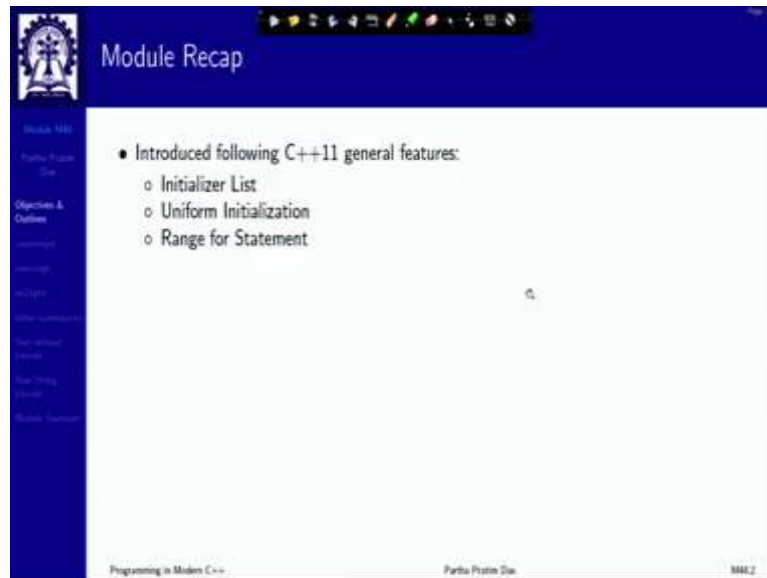


**Programming in Modern C++**  
**Professor Partha Pratim Das**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kharagpur**  
**Lecture 48**

**C++ 11 and beyond: General Features: Part 3**

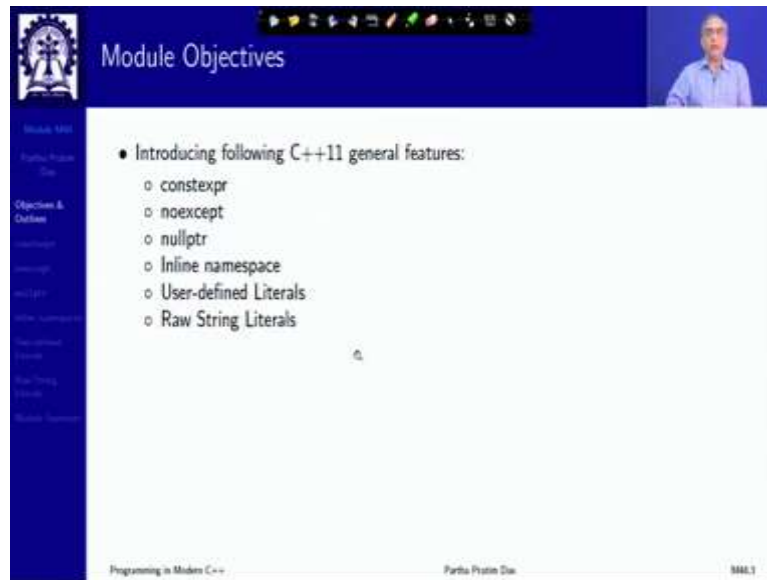
Welcome to Programming in Modern C++. We are in week 10 and we are going to discuss module 48.

(Refer Slide Time: 0:38)



In the last module we have introduced, continued on introducing some general features of C++ 11. We have talked about initializer list, the braced initialization and its consequences and also the uniform initialization mechanism, which can be uniformly used everywhere in C++ 11. And we have also seen a convenience mechanism for iteration over an entire data structure by using the range for statement.

(Refer Slide Time: 1:12)

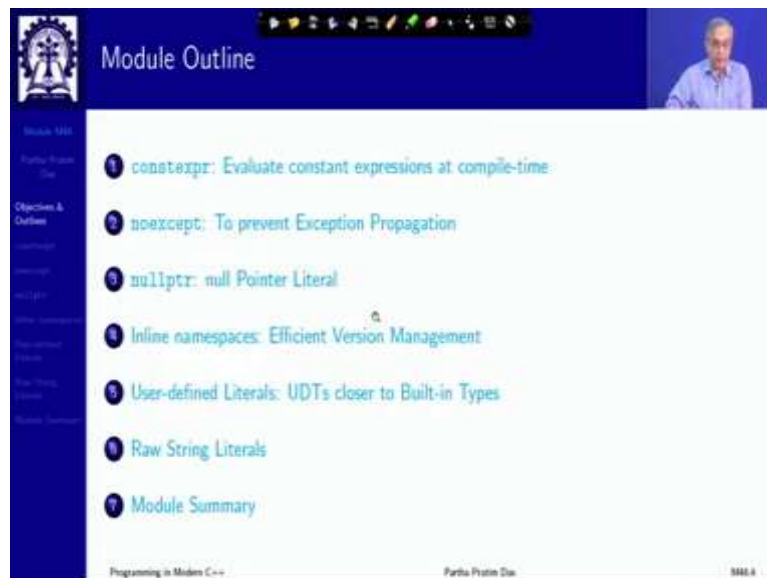


The slide is titled "Module Objectives" and features a blue header with a logo on the left and a small video inset of a speaker on the right. The main content area is white and contains a bulleted list of C++11 features. At the bottom, there is a footer with the text "Programming in Modern C++", "Partho Pratim Das", and "SML1".

- Introducing following C++11 general features:
  - constexpr
  - noexcept
  - nullptr
  - Inline namespace
  - User-defined Literals
  - Raw String Literals

We will continue on these and introduce several of other general features of C++ 11, which are on one side for convenience, for safety, for efficiency and as well as will become important for the later important features. So, these are the six features that we are going to talk about in this module.

(Refer Slide Time: 1:37)



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 topics. At the bottom, there is a footer with the text "Programming in Modern C++", "Partho Pratim Das", and "SML1".

- 1 constexpr: Evaluate constant expressions at compile-time
- 2 noexcept: To prevent Exception Propagation
- 3 nullptr: null Pointer Literal
- 4 Inline namespaces: Efficient Version Management
- 5 User-defined Literals: UDTs closer to Built-in Types
- 6 Raw String Literals
- 7 Module Summary

This will be our outline naturally which will be available on the left.

(Refer Slide Time: 1:43)

constexpr: Evaluate constant expressions at compile-time

Sources:

- [constexpr, isocpp.org](#)
- [An Overview of the New C++ \(C++11/14\)](#), Scott Meyers Training Courses
- [Understanding constexpr specifier in C++](#), [geekforgeeks.org](#)
- [Difference between 'constexpr' and 'const'](#), [stackoverflow.com](#), 2013
- [C++20 constexpr specifier](#)

Programming in Modern C++ Partho Pratap Das NMMS

So, let us start with const expression. Const expression is a feature to evaluate constant expressions at compile time.

(Refer Slide Time: 1:55)

constexpr

- The `constexpr` mechanism
  - provides more *general constant expressions*
  - allows constant expressions involving *user-defined types*
  - provides a way to guarantee that an *initialization is done at compile time*

```
enum Flags { good = 0, fail = 1, bad = 2, eof = 4 }; ✓
constexpr int operator|(Flags f1, Flags f2)
{ return Flags(INV(f1) | int(f2)); }

void f(Flags x) {
    switch (x) {
        case bad: /* ... */ break;
        case eof: /* ... */ break;
        case bad|eof: /* ... */ break;
        default: /* ... */ break;
    }
}
```

Programming in Modern C++ Partho Pratap Das NMMS

constexpr

- The constexpr mechanism
  - provides more *general constant expressions*
  - allows constant expressions involving *user-defined types*
  - provides a way to guarantee that an *initialization is done at compile time*

```

enum Flags { good = 0, fail = 1, bad = 2, eof = 4 };
constexpr int operator|(Flags f1, Flags f2)
{ return Flags(int(f1) | int(f2)); }

void f(Flags x) {
  switch (x) {
    case bad: /* ... */ break;
    case eof: /* ... */ break;
    case bad|eof: /* ... */ break;
    default: /* ... */ break;
  }
}

```

constexpr

- The constexpr mechanism
  - provides more *general constant expressions*
  - allows constant expressions involving *user-defined types*
  - provides a way to guarantee that an *initialization is done at compile time*

```

enum Flags { good = 0, fail = 1, bad = 2, eof = 4 };
constexpr int operator|(Flags f1, Flags f2)
{ return Flags(int(f1) | int(f2)); }

void f(Flags x) {
  switch (x) {
    case bad: /* ... */ break;
    case eof: /* ... */ break;
    case bad|eof: /* ... */ break;
    default: /* ... */ break;
  }
}

```

constexpr

- The constexpr mechanism
  - provides more *general constant expressions*
  - allows constant expressions involving *user-defined types*
  - provides a way to guarantee that an *initialization is done at compile time*

```

enum Flags { good = 0, fail = 1, bad = 2, eof = 4 };
constexpr int operator|(Flags f1, Flags f2)
{ return Flags(int(f1) | int(f2)); }

void f(Flags x) {
  switch (x) {
    case bad: /* ... */ break;
    case eof: /* ... */ break;
    case bad|eof: /* ... */ break;
    default: /* ... */ break;
  }
}

```

The screenshot shows a presentation slide with the title "constexpr". It contains the following text:

- The `constexpr` mechanism
  - provides more *general constant expressions*
  - allows constant expressions involving *user-defined types*
  - provides a way to guarantee that an *initialization is done at compile time*

```
enum Flags { good = 0, fail = 1, bad = 2, eof = 4 };
constexpr int operator|(Flags f1, Flags f2)
{ return Flags(int(f1) | int(f2)); }

void f(Flags x) {
  switch (x) {
    case bad: /* ... */ break;
    case eof: /* ... */ break;
    case bad|eof: /* ... */ break;
    default: /* ... */ break;
  }
}
```

Handwritten in blue ink on the right side of the slide is a binary diagram:

$$\begin{array}{r} 0010 \\ 1000 \\ \hline 1010 \end{array}$$

So, let us, see what it means is; const expression is in a way more, I mean. it gives us more than the general constant expressions. We have constant expression or const seen earlier, it will provide for somewhat more and with some differences. It allows constant expressions involving user defined types and provide a way to guarantee that an initialization is done at compile time for a const expression, given that you have the constants available at compile time.

So, here is a simple example of an enum of flags having four different possible innumerable values and we are defining an operator or for two flags, f1 and f2. Now, if you look at this then this is, it basically takes the flag f1 and the flag f2, converts each to int as you know normal enum can be converted to int, and then returns the value as an int value. So, that is what the flags are doing.

Now, what will you have? Since, if you know the values of the flags f1 and f2 at the time of compilation, naturally this entire expression can be evaluated at the compile time. So, as an illustration of that let us see we have a flag, we have a switch based on this flag type x. Now, as you know the cases of the switch need to be constants at the compile time, so it is quite obvious that I can have case bad which is basically case 2.

I can have case eof, which is ah case four, but I wanted to write case bad or eof. This is an operation which normally you will expect at the runtime, if you, but if that happens at the run time then this code cannot compile. This code will not compile because you will not know what is the value of bad or eof as a constant at the compile time and switch case will not allow that. So, without this const expression this code will not compile.

But what will happen with the const expression, with the const expression at the compile time f1 will be treated as bad, eof will be treated as f, I am sorry, f2 will be treated as eof and it will actually compute the o of these two and whatever is that odd value, so if I have 0 0 1 0 and eof is 4, so if I have or of that, so for this value this case will be applicable. So, this is possible because we can evaluate this expression at the constant, as a constant expression at the compiled time. So, this is the basic feature of the const expression as we have.

(Refer Slide Time: 5:07)

**constexpr**

- Here `constexpr` says that the function must be of a simple form so that it can be evaluated at compile time if given constant expressions arguments
- In addition to be able to evaluate expressions at compile time, we want to be able to require expressions to be evaluated at compile time
- `constexpr` in front of a variable definition does that (and implies `const`):

```
constexpr int x1 = bad|eof; // okay

void f(Flags f3) {
    constexpr int x2 = bad|f3; // error: cannot evaluate at compile time
    int x3 = bad|f3; // okay
}
```

- Recall the use of `constexpr` in `std::initializer_list` in Module 47

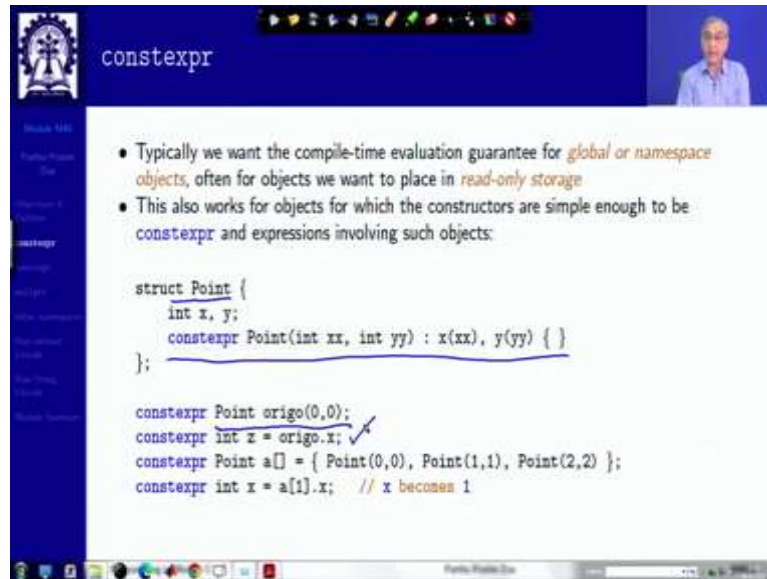
So, typically the const expression is a function that must be of simple form that it can be evaluated at compile time given the constant expression and it is not only that it gives us an ability to evaluate expressions at compile time but we want to be able to require expressions to be evaluated at compile time, because that gives better efficiency and of the code that we have. So, const expression in the form, in front of a variable definition will does that, and will do that and will also mean what we traditionally mean by const.

So, this is a different context, so we have const expression before int x and this is an expression which can be evaluated at compile time, so it will evaluate, have a constant value and will treat this as a constant. Similarly, but if we try to do say in flags, in this function if we try to do this, see f3 is a parameter and I am using that, Now, this is, this function will get the value of f3 at the runtime and only then it will know what is bad or f3.

So, you cannot evaluate this as a const expression, but if you just do bad or f3 you will be able to do that, in fact, if you do const int x3, bad or f3 that will also be valid because const does not need the evaluation to happen at the compiled time. It only says that it has to be a

constant expression. We have already seen the use of const expression in module 47 in the standard library initializer list.

(Refer Slide Time: 7:04)



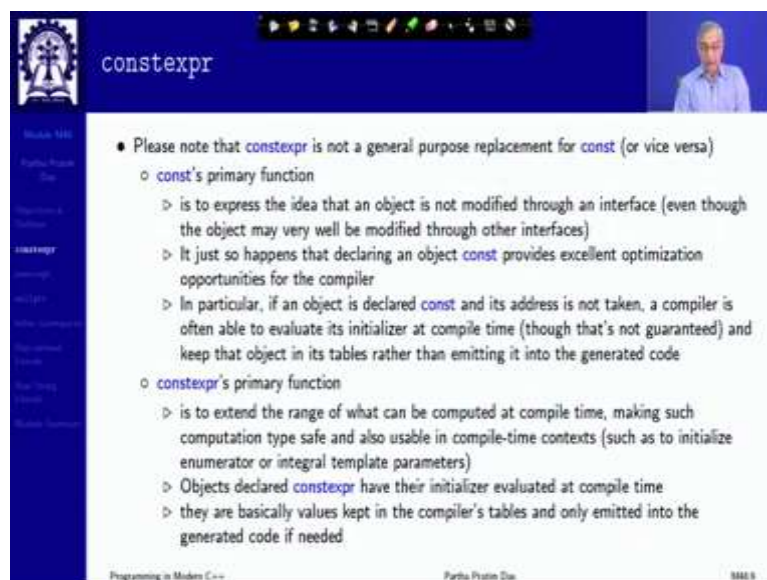
The slide is titled "constexpr" and features a blue header with a logo on the left and a small video feed of a speaker on the right. The main content area is white with a blue sidebar on the left containing navigation icons. The slide contains the following text:

- Typically we want the compile-time evaluation guarantee for *global or namespace objects*, often for objects we want to place in *read-only storage*
- This also works for objects for which the constructors are simple enough to be `constexpr` and expressions involving such objects:

```
struct Point {  
    int x, y;  
    constexpr Point(int xx, int yy) : x(xx), y(yy) { }  
};  
  
constexpr Point origo(0,0);  
constexpr int z = origo.x; ✓  
constexpr Point a[] = { Point(0,0), Point(1,1), Point(2,2) };  
constexpr int x = a[1].x; // x becomes 1
```

So, it can be used, const expression can be used with user defined types also. Here is an example, we have a struct point and I have specified the constructor to be const expression, so const expression point origo (0, 0) will actually at the compile time construct this object and will also set the values of x and y. Similarly, it can then set z as a const expression origo dot x which will actually get 0 and so on so forth, the arrays and so on. So, all of these different types of const expressions are possible.

(Refer Slide Time: 7:45)



The slide is titled "constexpr" and features a blue header with a logo on the left and a small video feed of a speaker on the right. The main content area is white with a blue sidebar on the left containing navigation icons. The slide contains the following text:

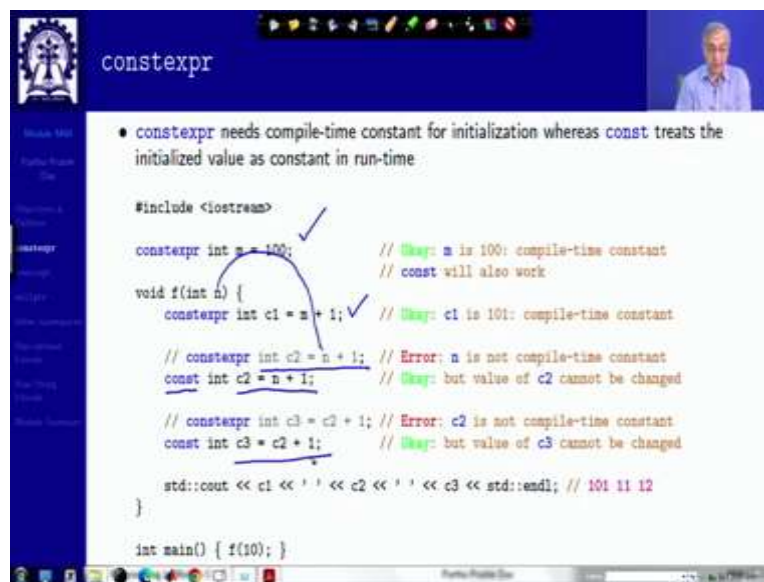
- Please note that `constexpr` is not a general purpose replacement for `const` (or vice versa)
- `const`'s primary function
  - ▷ is to express the idea that an object is not modified through an interface (even though the object may very well be modified through other interfaces)
  - ▷ It just so happens that declaring an object `const` provides excellent optimization opportunities for the compiler
  - ▷ In particular, if an object is declared `const` and its address is not taken, a compiler is often able to evaluate its initializer at compile time (though that's not guaranteed) and keep that object in its tables rather than emitting it into the generated code
- `constexpr`'s primary function
  - ▷ is to extend the range of what can be computed at compile time, making such computation type safe and also usable in compile-time contexts (such as to initialize enumerator or integral template parameters)
  - ▷ Objects declared `constexpr` have their initializer evaluated at compile time
  - ▷ they are basically values kept in the compiler's tables and only emitted into the generated code if needed

At the bottom of the slide, there is a footer with the text "Programming in Modern C++", "Partho Pratim Das", and "3885".

Note the difference between const and const expression. It is not in general, one is not a replacement of the other. Const's primary concern is to express the idea that the object is not modified through the const interface, though it may be modified through other means. So, it just so happens that telling that an object is const provides good opportunities for optimization for the compiler.

And for example, if an object is const and its address is not taken, then the compiler may not allocate any memory for that, it can just keep it in a read only table, whereas const expressions primary concern is to check what can be computed at compile time. So, making such computations, type safe and usable in any compile time context. For example, size of an array, initializing enumerators, passing as a int, template parameter and so on so forth. So, objects declare const expression have their initializer evaluated at the compiled time. So, that is the basic difference between these two.

(Refer Slide Time: 9:04)



```
constexpr
#include <iostream>
constexpr int m = 100; // Okay: m is 100: compile-time constant
// const will also work
void f(int n) {
    constexpr int c1 = m + 1; // Okay: c1 is 101: compile-time constant
    // constexpr int c2 = n + 1; // Error: n is not compile-time constant
    const int c2 = n + 1; // Okay: but value of c2 cannot be changed
    // constexpr int c3 = c2 + 1; // Error: c2 is not compile-time constant
    const int c3 = c2 + 1; // Okay: but value of c3 cannot be changed
    std::cout << c1 << ' ' << c2 << ' ' << c3 << std::endl; // 101 11 12
}
int main() { f(10); }
```

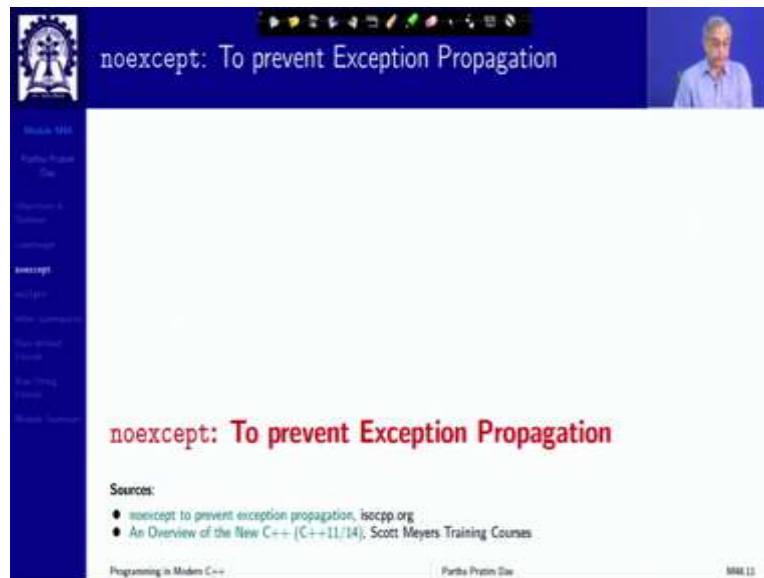
Here is another example to show the concept, so these are const expression for variable declaration m, so where const will also work actually. You can then define const expression c1 as m + 1 which will be fine because m will be const, compile time 100, so this is 101. But if you try to do say const expression int c2 of n + 1 where n is a parameter, this will not compile, because it is not known to be a constant at the compiled time.

Whereas you can do const on this, because all that it says if you say const is, it says that when this gets executed at the runtime, it is not required at the compile time, but the runtime when it gets executed whatever is the value of n + 1, will be the value with which you initialize c2 and that cannot be changed through this interface at least. So, similar thing is for c3 being



defined as  $c2 + 1$  and so on. So, that is the difference between these two const expression is a very powerful mechanism to optimize at the compile time.

(Refer Slide Time: 10:17)



The slide features a blue header with the title "noexcept: To prevent Exception Propagation" and a small video inset of the presenter in the top right. A vertical navigation menu is on the left. The main content area has the title in red and a "Sources:" section with two bullet points.

## noexcept: To prevent Exception Propagation

Sources:

- `noexcept` to prevent exception propagation, [isocpp.org](http://isocpp.org)
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses

Programming in Modern C++ Partha Pratim Das MM 11



The slide features a blue header with the title "noexcept" and a small video inset of the presenter in the top right. A vertical navigation menu is on the left. The main content area contains a list of bullet points and a code snippet.

## noexcept

- If a function cannot throw an exception or if the program is not written to handle exceptions thrown by a function, that function can be declared `noexcept`:  

```
extern "C" double sqrt(double) noexcept; // will never throw
```
- If a function declared `noexcept` throws (so that the exception tries to escape the `noexcept` function)
  - the program is terminated by a call to `std::terminate()`
  - the call of `terminate()` cannot rely on objects being in well-defined states, that is, there is
    - ▷ no guarantee that destructors have been invoked
    - ▷ no guaranteed stack unwinding, and
    - ▷ no possibility for resuming the program as if no problem had been encountered
  - This is deliberate and makes `noexcept` a simple, crude, and very efficient mechanism
    - ▷ much more efficient than the old dynamic `throw()` exception specification mechanism

```
// Not prepared to handle memory exhaustion
vector<double> my_computation(const vector<double>& v) noexcept {
    vector<double> res(v.size()); // might throw
    for(int i; i<v.size(); ++i) res[i] = sqrt(v[i]);
    return res;
}
```

Programming in Modern C++ Partha Pratim Das MM 11

## noexcept

- If a function cannot throw an exception or if the program is not written to handle exceptions thrown by a function, that function can be declared `noexcept`:
 

```
extern "C" double sqrt(double) noexcept; // will never throw
```

```
// Not prepared to handle memory exhaustion ...
vector<double> my_computation(const vector<double>& v) noexcept {
    vector<double> res(v.size()); // might throw
    for(int i; i<v.size(); ++i) res[i] = sqrt(v[i]);
    return res;
}
```
- If a function declared `noexcept` throws (so that the exception tries to escape the `noexcept` function)
  - the program is terminated by a call to `std::terminate()`
  - the call of `terminate()` cannot rely on objects being in well-defined states, that is, there is
    - > no guarantee that destructors have been invoked
    - > no guaranteed stack unwinding, and
    - > no possibility for resuming the program as if no problem had been encountered
  - This is deliberate and makes `noexcept` a simple, crude, and very efficient mechanism
    - > much more efficient than the old dynamic `throw()` exception specification mechanism

## noexcept

- If a function cannot throw an exception or if the program is not written to handle exceptions thrown by a function, that function can be declared `noexcept`:
 

```
extern "C" double sqrt(double) noexcept; // will never throw
```

```
// Not prepared to handle memory exhaustion
vector<double> my_computation(const vector<double>& v) noexcept {
    vector<double> res(v.size()); // might throw
    for(int i; i<v.size(); ++i) res[i] = sqrt(v[i]);
    return res;
}
```
- If a function declared `noexcept` throws (so that the exception tries to escape the `noexcept` function)
  - the program is terminated by a call to `std::terminate()`
  - the call of `terminate()` cannot rely on objects being in well-defined states, that is, there is
    - > no guarantee that destructors have been invoked
    - > no guaranteed stack unwinding, and
    - > no possibility for resuming the program as if no problem had been encountered
  - This is deliberate and makes `noexcept` a simple, crude, and very efficient mechanism
    - > much more efficient than the old dynamic `throw()` exception specification mechanism

## noexcept

- If a function cannot throw an exception or if the program is not written to handle exceptions thrown by a function, that function can be declared `noexcept`:
 

```
extern "C" double sqrt(double) noexcept; // will never throw
```

```
// Not prepared to handle memory exhaustion
vector<double> my_computation(const vector<double>& v) noexcept {
    vector<double> res(v.size()); // might throw
    for(int i; i<v.size(); ++i) res[i] = sqrt(v[i]);
    return res;
}
```
- If a function declared `noexcept` throws (so that the exception tries to escape the `noexcept` function)
  - the program is terminated by a call to `std::terminate()`
  - the call of `terminate()` cannot rely on objects being in well-defined states, that is, there is
    - > no guarantee that destructors have been invoked
    - > no guaranteed stack unwinding, and
    - > no possibility for resuming the program as if no problem had been encountered
  - This is deliberate and makes `noexcept` a simple, crude, and very efficient mechanism
    - > much more efficient than the old dynamic `throw()` exception specification mechanism

The second feature that we, these features are all kind of diverse because there are features on different aspects that are being talked off, so there is a feature called `noexcept` to declare that a function will not throw an exception or it cannot handle, it has not been written to handle exceptions thrown by functions within it. So, we are saying that let us say `extern C`, a C function from the standard math library, we say that `sqrt` function is `noexcept`.

Which means that the `sqrt` function will never throw if I call it. It will execute and give me a proper value. So, using that I am defining some function for a, my computation function for a vector, which I want to declare as `noexcept`. So, what it says? This `noexcept` says that this function my computation, either will not throw or it is not written to handle exceptions to be thrown. For example, it is using `sqrt`, so this will not throw.

But it is also using say `v.size` here, which might throw, but even if this throws, this function is not written, equipped to handle that exception. So, that is the basic idea of `noexcept`. So, naturally that opens the question as to what happens if a function specified to be `noexcept` if it throws. If it throws then naturally you are violating the basic guarantee given by the code.

So, this will in turn call a function in the standard library called `std::terminate`, which will terminate the function, it is like, I mean close to what a bot does, you can also register your own function for `terminate`. Now, this function `terminate` does not do the typical tasks of exception handling. For example, it does not guarantee that the destructors of the objects that are getting out of context will be called.

It does not guarantee that the stack will be properly unwinded or it does not keep the provision of resuming the computation if the program has been found to have no further problem. So, you will recall that in exception also, mechanism also, we had a way to say `no throw`, `no throw` in this way, that is it specifies what does a function throws and giving an empty parameter says that it does not throw of any kind. So, which is semantically similar to `noexcept`, but the fact is through this throws mechanism is actually run time, so it is far less efficient in terms of the exception specification in contrast to what `noexcept` can do which is a compile time feature being provided.

(Refer Slide Time: 13:33)

### noexcept

- It is possible to make a function *conditionally noexcept*. For example, an algorithm can be specified to be `noexcept` iff the operations it uses on a template argument are `noexcept`:

```
template<class T>
// can throw if f(v.at(0)) can
void do_f(vector<T>& v) noexcept(noexcept(f(v.at(0)))) {
    for(int i; i<v.size(); ++i)
        v.at(i) = f(v.at(i));
}
```

- Here, we use `noexcept` as an operator:
  - `noexcept(f(v.at(0)))` is true if `f(v.at(0))` cannot throw, that is, if the `f()` and `at()` used are `noexcept`
- The `noexcept()` operator is a
  - constant expression and
  - does not evaluate its operand

### noexcept

- It is possible to make a function *conditionally noexcept*. For example, an algorithm can be specified to be `noexcept` iff the operations it uses on a template argument are `noexcept`:

```
template<class T>
// can throw if f(v.at(0)) can
void do_f(vector<T>& v) noexcept(noexcept(f(v.at(0)))) {
    for(int i; i<v.size(); ++i)
        v.at(i) = f(v.at(i));
}
```

- Here, we use `noexcept` as an operator:
  - `noexcept(f(v.at(0)))` is true if `f(v.at(0))` cannot throw, that is, if the `f()` and `at()` used are `noexcept`
- The `noexcept()` operator is a
  - constant expression and
  - does not evaluate its operand

### noexcept

- It is possible to make a function *conditionally noexcept*. For example, an algorithm can be specified to be `noexcept` iff the operations it uses on a template argument are `noexcept`:

```
template<class T>
// can throw if f(v.at(0)) can
void do_f(vector<T>& v) noexcept(noexcept(f(v.at(0)))) {
    for(int i; i<v.size(); ++i)
        v.at(i) = f(v.at(i));
}
```

- Here, we use `noexcept` as an operator:
  - `noexcept(f(v.at(0)))` is true if `f(v.at(0))` cannot throw, that is, if the `f()` and `at()` used are `noexcept`
- The `noexcept()` operator is a
  - constant expression and
  - does not evaluate its operand

Now, the interesting fact is many a times the `noexcept` can be used as a conditionally and that becomes particularly useful for templates. So, what we want to say is suppose I have a function `do_f` templated by the type `t` and that type could be kind of anything. So, what I want to say is this needs to use `f(v.at(i))`. At `i` is basically accessing a vector location and so it is accessing the `i`th location of the vector and then using `f`.

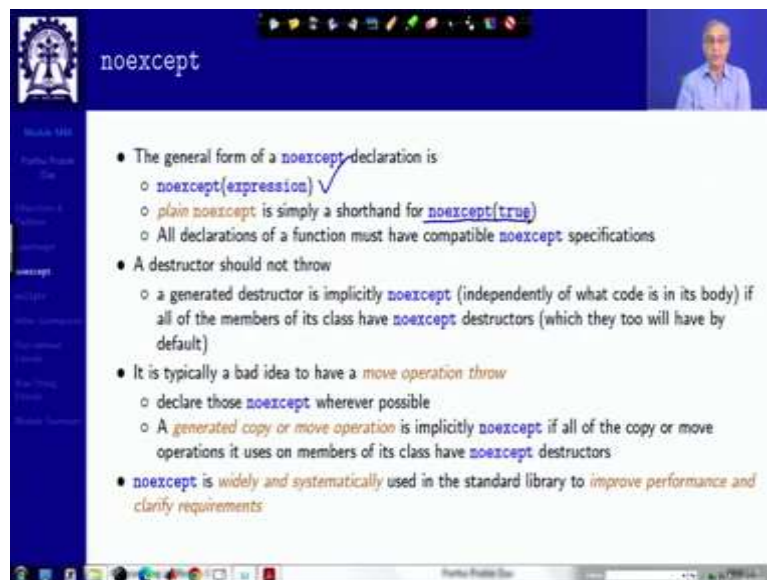
So, there are two function calls involved here, so the guarantee that we want to say, we cannot say it is `noexcept`, if we give `noexcept` then you are saying that it cannot handle anything. But what we want to say is this will be `noexcept` provided this expression that is the call to `at` and call to `f`, they do not throw, if they are `noexcept`.

So, it is kind of a conditional specification where as you can see easily that `noexcept` instead of just being used as a qualifying keyword it is being used kind of as a compile time operator. So, what does this operator do? It says that this will be true if `f(v.at(0))` cannot throw, that is if `f` as well as `at` are used as `noexcept`.

So, it will check that because do not know for what `f` is, I do not know for that vector type what `at`, or might have been. So, this will give me a conditional, so provided if this is `noexcept`, if this is true, then this is, then this becomes `noexcept(true)`, that means this function is `noexcept`. But if this is false, then then this does not hold any good.

So, the interesting factor about `noexcept` is it is a constant expression, that it is evaluated at the compiled time, that is the reason I need `at 0` and it does not evaluate its operand, it does not evaluate that, but it just checks the behavior, checks the specification of the corresponding functions and they are `noexcept` status.

(Refer Slide Time: 16:15)



The slide is titled "noexcept" and features a list of bullet points. A small video inset in the top right shows a man speaking. The slide content is as follows:

- The general form of a `noexcept` declaration is
  - `noexcept(expression)` ✓
  - *plain* `noexcept` is simply a shorthand for `noexcept(true)`
  - All declarations of a function must have compatible `noexcept` specifications
- A destructor should not throw
  - a generated destructor is implicitly `noexcept` (independently of what code is in its body) if all of the members of its class have `noexcept` destructors (which they too will have by default)
- It is typically a bad idea to have a *move operation* `throw`
  - declare those `noexcept` wherever possible
  - A *generated copy or move operation* is implicitly `noexcept` if all of the copy or move operations it uses on members of its class have `noexcept` destructors
- `noexcept` is *widely and systematically* used in the standard library to *improve performance and clarify requirements*

So, `noexcept` declaration would typically be in the form of `noexcept` followed by an expression and this will be true or false and the plain `noexcept` is just a short form of `noexcept true`. Now, normally destructors should not throw, because then the cleanup becomes a mess, so any destructor that is, that you write, you should write it as `noexcept`.

If the compiler provides the destructor, then it will be implicitly `noexcept` provided all objects being destroyed in that destructor, the member variables are also have destructors which are `noexcept`. Similarly, the move operations must not throw, they should typically be `noexcept`, we will talk about the move expression very soon, but not in this module. So, `noexcept` is widely and systematically used in the standard library to improve performance and to clarify the requirements of where do you really need to do checks for the exception handling and where you can skip those.

(Refer Slide Time: 17:27)

The slide shows a search for 'nullptr' in a presentation software. The search results are empty. Below the search bar, the text 'nullptr: null Pointer Literal' is displayed in red. Underneath, there is a 'Sources:' section with a list of references:

- `nullptr` - a null pointer literal, [isocpp.org](http://isocpp.org)
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- `NULL`, [cpreference.com](http://cpreference.com) (C)
- `NULL`, [cpreference.com](http://cpreference.com) (C++)

At the bottom, it says 'Programming in Modern C++' and 'Partha Pratim Das'.

The slide is titled 'nullptr' and contains the following content:

- `nullptr` is a literal denoting the null pointer
  - Literal of type `std::nullptr_t` in `<cstdint>`
  - Convertible to any pointer type and to `bool`, but nothing else
  - It is not an integer and cannot be used as an integral value
- `nullptr` is provided to replace the macro `NULL`
  - C implementations (`<stddef.h>`, and others)
    - `#define NULL 0 // C++ compatible ✓`
    - `#define NULL (0*2 - 20) // C++ incompatible ✓`
    - `#define NULL {(void*)0} // C++ incompatible ✓`
  - C++ implementations (`<cstdint>`, and others)
    - `#define NULL 0 // C++03, May be 0L in some compiler`
    - `#define NULL nullptr // C++11`
- `NULL` or `0` causes confusion in following cases that `nullptr` can resolve:
  - **Function Overload Resolution**
  - **Forwarding Templates**

The next feature is a null pointer. You will wonder as to why do we need null pointer, we already had null pointer. So, this is a new keyword and a literal. `nullptr` is a new keyword and a literal of a new type called `new ptr_t`, and this type is defined in `stdint.h` in C standard library so `cstdint` is where you will get it. So, this is a literal which is convertible to any pointer type and to `bool` but nothing else.

Now, if you look at the typical use of capital `NULL` as a null pointer, you will typically have a macro somewhere in `stdint.h` or `cstdint`, which is defined as a `0`. So, that basically tells you that it is an integer `0`, which is defined as a null. Now, some C, so this is what how C implements it, so which is compatible to C++.

Some C plus, C compilers do some, try to do something more intelligent like specifying this as an integer expression, making sure that you get the proper integer type. Some write it as a void\* null pointer and so on, note that these are not compatible to C++. C++ typically will implement this as NULL 0 or in C++ 11 NULL will be defined as nullptr. So, what is the advantage of having this nullptr? It solves basically two problems, one is a function overload resolution and one is of forwarding templates.

(Refer Slide Time: 19:24)

**nullptr: Function Overload Resolution & Forwarding**

```

// Simple Examples
int* q = nullptr; // q is null
char* p = nullptr; // p is null
char* p1 = 0; // 0 still works, p1 is null and p == p1
char* p2 = NULL; // p2 is null
if (p) ... // compiles but fails
if (p == p1) ... // compiles and succeeds
if (q == p2) ... // error: comparison between distinct pointer types int* and char*
void g(int);
g(nullptr); // error: nullptr is not an int, cannot convert std::nullptr_t to int
int i = nullptr; // error: nullptr is not an int, cannot convert std::nullptr_t to int

void f(int); void f(int*); // Function overload resolution
f(0); // call f(int)
f(nullptr); // call f(int*)
f(NULL); // error: call of overloaded f(NULL) is ambiguous for f(int) and f(int*)

void h(int*); // h(0) and h(nullptr) are okay
template<typename F, typename P> // Forwarding template
void logNoCall(F func, P param) {
    ... func(param); // make log entry ... then invoke func on param
}
logNoCall(h, 0); // error: P deduced as int, and h(int) invalid
logNoCall(h, NULL); // error: P deduced as long int, and h(long int) invalid
logNoCall(h, nullptr); // P deduced as std::nullptr_t, and h(std::nullptr_t) is okay

```

**nullptr: Function Overload Resolution & Forwarding**

```

// Simple Examples
int* q = nullptr; // q is null
char* p = nullptr; // p is null
char* p1 = 0; // 0 still works, p1 is null and p == p1
char* p2 = NULL; // p2 is null
if (p) ... // compiles but fails
if (p == p1) ... // compiles and succeeds
if (q == p2) ... // error: comparison between distinct pointer types int* and char*
void g(int);
g(nullptr); // error: nullptr is not an int, cannot convert std::nullptr_t to int
int i = nullptr; // error: nullptr is not an int, cannot convert std::nullptr_t to int

void f(int); void f(int*); // Function overload resolution
f(0); // call f(int)
f(nullptr); // call f(int*)
f(NULL); // error: call of overloaded f(NULL) is ambiguous for f(int) and f(int*)

void h(int*); // h(0) and h(nullptr) are okay
template<typename F, typename P> // Forwarding template
void logNoCall(F func, P param) {
    ... func(param); // make log entry ... then invoke func on param
}
logNoCall(h, 0); // error: P deduced as int, and h(int) invalid
logNoCall(h, NULL); // error: P deduced as long int, and h(long int) invalid
logNoCall(h, nullptr); // P deduced as std::nullptr_t, and h(std::nullptr_t) is okay

```



```

// Simple Examples
int* q = nullptr; // q is null
char* p = nullptr; // p is null
char* p1 = 0; // 0 still works, p1 is null and p == p1
char* p2 = NULL; // p2 is null
if (p) ... // compile but fails
if (p == p1) ... // compile and succeeds
if (q == p2) ... // error: comparison between distinct pointer types int* and char*
void g(int);
g(nullptr); // error: nullptr is not an int, cannot convert std::nullptr_t to int
int i = nullptr; // error: nullptr is not an int, cannot convert std::nullptr_t to int

void f(int); void f(int*); // Function overload resolution
f(0); // call f(int)
f(nullptr); // call f(int*)
f(NULL); // error: call of overloaded f(NULL) is ambiguous for f(int) and f(int*)

void h(int*); // h(0) and h(nullptr) are okay
template<typename F, typename P> // Forwarding template
void logAndCall(F func, P param) {
    ... func(param); // make log entry ... then invoke func on param
}
logAndCall(h, 0); // error: P deduced as int, and h(int) invalid
logAndCall(h, NULL); // error: P deduced as long int, and h(long int) invalid
logAndCall(h, nullptr); // P deduced as std::nullptr_t, and h(std::nullptr_t) is okay

```

```

// Simple Examples
int* q = nullptr; // q is null
char* p = nullptr; // p is null
char* p1 = 0; // 0 still works, p1 is null and p == p1
char* p2 = NULL; // p2 is null
if (p) ... // compile but fails
if (p == p1) ... // compile and succeeds
if (q == p2) ... // error: comparison between distinct pointer types int* and char*
void g(int);
g(nullptr); // error: nullptr is not an int, cannot convert std::nullptr_t to int
int i = nullptr; // error: nullptr is not an int, cannot convert std::nullptr_t to int

void f(int); void f(int*); // Function overload resolution
f(0); // call f(int)
f(nullptr); // call f(int*)
f(NULL); // error: call of overloaded f(NULL) is ambiguous for f(int) and f(int*)

void h(int*); // h(0) and h(nullptr) are okay
template<typename F, typename P> // Forwarding template
void logAndCall(F func, P param) {
    ... func(param); // make log entry ... then invoke func on param
}
logAndCall(h, 0); // error: P deduced as int, and h(int) invalid
logAndCall(h, NULL); // error: P deduced as long int, and h(long int) invalid
logAndCall(h, nullptr); // P deduced as std::nullptr_t, and h(std::nullptr_t) is okay

```

So, let us look at these with through some examples. So, first let us have some examples of the nullptr. So, I have a integer pointer q initialized to nullptr. The character pointed p initialized to nullptr initialize to 0, initialize to null, p, p1, p2 all of these we will work. Now, if I check p it will compile but obviously it will fail because p is a nullptr. So, you can see that nullptr, though is of nullptr\_t type it will get converted to bool for this if check.

I can compare it with another pointer, p can be equated to with p1, where p is initialized with nullptr and p one is initialized with 0, so this will compile and fail, this will compile as well as succeed because both of them are actually null. But if I try to do compare q with p2, q is int\*, p2 is char\*, both are null but their types are different, so that conversion will not be allowed, conversion between distinct pointer types are not allowed so therefore it will fail.

Similarly, if we have a function `g` which takes a parameter of type `int` and you try to call it with `nullptr` you will get an error because as we said `nullptr` cannot be converted to `int`. If you try to initialize an `int` value with `nullptr` you will also get an error, with capital `NULL`, both of these will actually work. Now, let us come to the function overload resolution. What is the problem? Suppose, I have a function `f` with two overloads, one is `int`, another is `int*`.

Now, often it becomes difficult to resolve a call to, for these overloads of the function. For example, if I write it say `f(0)` in C++ 11 it will call the `int` version, if I write it as `nullptr` it will write, it will get me the `int*` version, because `0` is a literal of `int` type, so `f(0)` is called to `f(int)`, `nullptr` is of type `nullptr_t`, so `f(nullptr)` is a pointer type so it will be, give me `f(int*)`.

(Refer Slide Time: 22:01)

```

// Simple Examples
int* q = nullptr; // q is null
char* p = nullptr; // p is null
char* p1 = 0; // 0 still works, p1 is null and p == p1
char* p2 = NULL; // p2 is null
if (p) ... // compiles but fails
if (p == p1) ... // compiles and succeeds
if (q == p2) ... // error: comparison between distinct pointer types int* and char*
void g(int);
g(nullptr); // error: nullptr is not an int, cannot convert std::nullptr_t to int
int i = nullptr; // error: nullptr is not an int, cannot convert std::nullptr_t to int

void f(int); void f(int*); // Function overload resolution
f(0); // call f(int)
f(nullptr); // call f(int*)
f(NULL); // error: call of overloaded f(NULL) is ambiguous for f(int) and f(int*)

void h(int*); // h(0) and h(nullptr) are okay
template<typename E, typename P> // Forwarding template
void logAndCall(F func, P param) {
    ... func(param); // make log entry ... then invoke func on param
}
logAndCall(h, 0); // error: P deduced as int, and h(int) invalid
logAndCall(h, NULL); // error: P deduced as long int, and h(long int) invalid
logAndCall(h, nullptr); // P deduced as std::nullptr_t, and h(std::nullptr_t) is okay
    
```

```

// Simple Examples
int* q = nullptr; // q is null
char* p = nullptr; // p is null
char* p1 = 0; // 0 still works, p1 is null and p == p1
char* p2 = NULL; // p2 is null
if (p) ... // compiles but fails
if (p == p1) ... // compiles and succeeds
if (q == p2) ... // error: comparison between distinct pointer types int* and char*
void g(int);
g(nullptr); // error: nullptr is not an int, cannot convert std::nullptr_t to int
int i = nullptr; // error: nullptr is not an int, cannot convert std::nullptr_t to int

void f(int); void f(int*); // Function overload resolution
f(0); // call f(int)
f(nullptr); // call f(int*)
f(NULL); // error: call of overloaded f(NULL) is ambiguous for f(int) and f(int*)

void h(int*); // h(0) and h(nullptr) are okay
template<typename E, typename P> // Forwarding template
void logAndCall(F func, P param) {
    ... func(param); // make log entry ... then invoke func on param
}
logAndCall(h, 0); // error: P deduced as int, and h(int) invalid
logAndCall(h, NULL); // error: P deduced as long int, and h(long int) invalid
logAndCall(h, nullptr); // P deduced as std::nullptr_t, and h(std::nullptr_t) is okay
    
```

## nullptr: Function Overload Resolution & Forwarding

```

// Simple Examples
int* q = nullptr; // q is null
char* p = nullptr; // p is null
char* p1 = 0; // 0 still works, p1 is null and p == p1
char* p2 = NULL; // p2 is null
if (p) ... // compiles but fails
if (p == p1) ... // compiles and succeeds
if (q == p2) ... // error: comparison between distinct pointer types int* and char*
void g(int);
g(nullptr); // error: nullptr is not an int, cannot convert std::nullptr_t to int
int i = nullptr; // error: nullptr is not an int, cannot convert std::nullptr_t to int

void f(int); void f(int*); // Function overload resolution
f(0); // call f(int)
f(nullptr); // call f(int*)
f(NULL); // error: call of overloaded f(NULL) is ambiguous for f(int) and f(int*)

void h(int*); // h(0) and h(nullptr) are okay
template<typename F, typename P> // Forwarding template
void logAndCall(F func, P param) {
    ... func(param); // make log entry ... then invoke func on param
}
logAndCall(h, 0); // error: P deduced as int, and h(int) invalid
logAndCall(h, NULL); // error: P deduced as long int, and h(long int) invalid
logAndCall(h, nullptr); // P deduced as std::nullptr_t, and h(std::nullptr_t) is okay

```

## nullptr: Function Overload Resolution & Forwarding

```

// Simple Examples
int* q = nullptr; // q is null
char* p = nullptr; // p is null
char* p1 = 0; // 0 still works, p1 is null and p == p1
char* p2 = NULL; // p2 is null
if (p) ... // compiles but fails
if (p == p1) ... // compiles and succeeds
if (q == p2) ... // error: comparison between distinct pointer types int* and char*
void g(int);
g(nullptr); // error: nullptr is not an int, cannot convert std::nullptr_t to int
int i = nullptr; // error: nullptr is not an int, cannot convert std::nullptr_t to int

void f(int); void f(int*); // Function overload resolution
f(0); // call f(int)
f(nullptr); // call f(int*)
f(NULL); // error: call of overloaded f(NULL) is ambiguous for f(int) and f(int*)

void h(int*); // h(0) and h(nullptr) are okay
template<typename F, typename P> // Forwarding template
void logAndCall(F func, P param) {
    ... func(param); // make log entry ... then invoke func on param
}
logAndCall(h, 0); // error: P deduced as int, and h(int) invalid
logAndCall(h, NULL); // error: P deduced as long int, and h(long int) invalid
logAndCall(h, nullptr); // P deduced as std::nullptr_t, and h(std::nullptr_t) is okay

```

## nullptr: Function Overload Resolution & Forwarding

```

// Simple Examples
int* q = nullptr; // q is null
char* p = nullptr; // p is null
char* p1 = 0; // 0 still works, p1 is null and p == p1
char* p2 = NULL; // p2 is null
if (p) ... // compiles but fails
if (p == p1) ... // compiles and succeeds
if (q == p2) ... // error: comparison between distinct pointer types int* and char*
void g(int);
g(nullptr); // error: nullptr is not an int, cannot convert std::nullptr_t to int
int i = nullptr; // error: nullptr is not an int, cannot convert std::nullptr_t to int

void f(int); void f(int*); // Function overload resolution
f(0); // call f(int)
f(nullptr); // call f(int*)
f(NULL); // error: call of overloaded f(NULL) is ambiguous for f(int) and f(int*)

void h(int*); // h(0) and h(nullptr) are okay
template<typename F, typename P> // Forwarding template
void logAndCall(F func, P param) {
    ... func(param); // make log entry ... then invoke func on param
}
logAndCall(h, 0); // error: P deduced as int, and h(int) invalid
logAndCall(h, NULL); // error: P deduced as long int, and h(long int) invalid
logAndCall(h, nullptr); // P deduced as std::nullptr_t, and h(std::nullptr_t) is okay

```

Now, if I had done `f(null)` then which function would get called? The problem is that the compiler will say I am confused, because `null` is actually a value `0`. So, considering `null` to be a value `0`, this function should get called, but a value `0` can also be interpreted as a `null` void pointer. So, this value should get called. This function should get called, so both of these overloads have equal weightage and therefore, it cannot be resolved, so using `null` this kind of a function overload resolution was not possible.

But with `null` pointer this will now be possible because you are specifically saying that I want the pointer version, when you use `0` you specifically say that I want the integer version. This gets more involved in a forwarding template. So, let us consider what it talks about. We need a function `h`, which takes an integer pointer. So, you can, as you can easily understand `h(0)` and `h(null pointer)` both are okay.

Now, I have a template of 2 type parameters `f` and `p`, and with that I templatize a function `log` and `call`, where `f` is actually a function parameter and `p` is the parameter to that function. So, it is kind of an application function, which takes a function pointer, it takes a parameter and applies that function on the function parameter. So, it tries to make this call. So, it is probably is doing some `log` entries and then it invokes this `func` on the parameter.

Now, what will happen if you call this with `h(0)`? Now, with `h(0)` the `h` is the `func` and the parameter is passed as `0`, so you want a template deduction to happen. By template deduction this is `int`, `0` is `int`, so it will be deduced as `int` and you are expecting a call to `h int` which does not exist, `h` is `int*`, so this fails. The same thing happens with `null`.

If you pass it as `null`, then whatever is the type of `#define null 0`, maybe it is `int`, maybe it is defined as `0 l`, in which case it is `long int`, so the compiler that I was using, the `g++` online `gcc` as I talked of, it takes it as `long int` the `null`, so `p` is deduced as `long int` and `h long int` is searched, it does not exist and therefore, it fails.

But if I call this with `null pointer`, `nullptr`, then it correctly succeeds because then `param` is deduced as `nullptr_t` type and therefore, `h` is of this `std::nullptr_t` type that you are trying to call and you have `int*` which said can automatically convert, so this function matches and the template will get forwarded easily. So, these are the two problems that `nullptr`, introduction of `nullptr` solves in terms of function overload resolution and in terms of forwarding template.

(Refer Slide Time: 25:41)



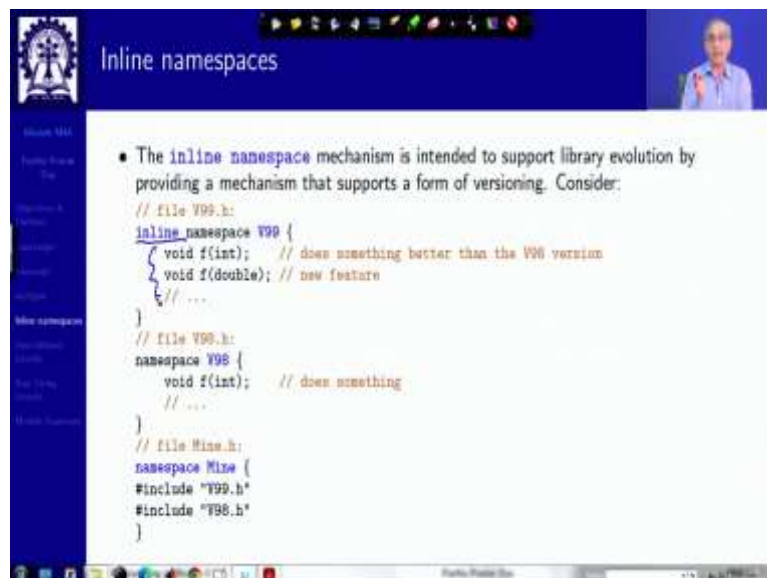
(Refer Slide Time: 27:04)



The slide is titled "Inline namespaces" and features a small video inset of a speaker in the top right corner. The main content consists of a bulleted list and a code snippet. The code snippet shows the inclusion of "Mine.h" and the use of the "Mine" namespace. It illustrates how to call functions from different versions (V98 and V99) and a default version (f(1)).

- We here have a namespace `Mine` with both the latest release (`V99`) and the previous one (`V98`). If we want to be specific, we can:  

```
#include "Mine.h"
using namespace Mine;
// ...
V98::f(1); // old version
V99::f(1); // new version
f(1);     // default version
```
- The point is that the `inline` specifier makes the declarations from the nested namespace appear exactly as if they had been declared in the enclosing namespace.
- This is a very *static* and *implementer-oriented facility* in that the `inline` specifier has to be placed by the designer of the namespaces – thus making the choice for all users.
  - It is not possible for a user of `Mine` to say:  
▷ *I want the default to be V98 rather than V99*



The slide is titled "Inline namespaces" and features a small video inset of a speaker in the top right corner. The main content consists of a bulleted list and code snippets for three files: V99.h, V98.h, and Mine.h. The code snippets show the structure of the inline namespace V99, the namespace V98, and the namespace Mine which includes both V99.h and V98.h.

- The `inline namespace` mechanism is intended to support library evolution by providing a mechanism that supports a form of versioning. Consider:  

```
// file V99.h:
inline namespace V99 {
    void f(int); // does something better than the V98 version
    void f(double); // new feature
    // ...
}
// file V98.h:
namespace V98 {
    void f(int); // does something
    // ...
}
// file Mine.h:
namespace Mine {
    #include "V99.h"
    #include "V98.h"
}
```

That is very simply in `Mine.h`, where you have included both these headers, you say you are using this `Mine` namespace and you say that if you want the old version you say `V98::f1`, if you say the new version, then you say `V99::f1`, no surprise till, but if you just say `f1`, that is if you do not make any changes, then it defaults to `V99`. Why? Why does it default to `V99`? Because `V99` is inline.

So, that means that anything that is inline is available in the enclosing namespace, enclosing namespace is global, so these are available in the enclosing namespace, unless you specifically use some other namespace. So, this default behavior is what is critical of the inline namespace feature. So, the company not only can provide the versions but also can, is

enforcing that if you are not consciously using the older version, then you will get automatically defaulted to the newer version.

(Refer Slide Time: 28:32)

**Inline namespaces.**  
namespace in C++03 vs. inline namespace in C++11

```
// C++03: nested namespaces
#include <iostream>
using namespace std;

namespace ns1 { int v1 = 2;
  namespace ns2 { int v2 = 3;
    namespace ns3 { int v3 = 5;
      }
    }
  }

int main() { // Fully qualified names must
  cout << ns1::v1 << ' ';
  cout << ns1::ns2::v2 << ' ';

  cout << ns1::ns2::ns3::v3 << endl;
}
2 3 5
```

```
// C++11: inline namespaces
#include <iostream>
using namespace std;

// inline namespace ns1; for global access
namespace ns1 { int v1 = 2;
  inline namespace ns2 { int v2 = 3;
    inline namespace ns3 { int v3 = 5;
      }
    }
  }

int main() { // Qualified by enclosing namespace
  cout << ns1::v1 << ' ';
  cout << ns1::v2 << ' ';
  cout << ns1::ns2::v3 << ' ';
  cout << ns1::v3 << endl;
}
2 3 5 5
```

- Note: If the outermost namespace (ns1) is inline, then the symbols within ns1 are available in the global namespace. For example, ns1::ns2::ns3::v3 can be accessed as v3 in main() besides as ns1::ns2::v3 and ns1::v3. This property is used in Version Control

You can see the comparison of the inline namespace between C++ 03 and C++ 11. I have three namespaces with three variables and I have to give complete qualification of these variables to be accessed properly. Whereas if I inline the inner two namespaces, if I inline namespace ns3, then every symbol in ns3 is automatically available in the enclosing namespace ns2. If we inline ns2, all are available in ns1.

So, actually using ns1 I can use, refer any of these variables implicitly. In addition, if I had done an inline of the outermost namespace, then the symbols would be available globally also the feature that we are using in version control. The question would be could we not have done this in C++ 03 we have using.

(Refer Slide Time: 29:30)

**Inline namespaces.**  
inline namespace effects by using namespace in C++03

```
// C++03: nested namespaces
#include <iostream>
using namespace std;

namespace ns1 { int v1 = 2;
  namespace ns2 { int v2 = 3;
    namespace ns3 { int v3 = 5;
      }
    }
  }

int main() { // Fully qualified names must
  cout << ns1::v1 << ' ';
  cout << ns1::ns2::v2 << ' ';

  cout << ns1::ns2::ns3::v3 << endl;
}
2 3 5
```

```
// C++03: inline namespace effect by using
#include <iostream>
using namespace std;

namespace ns1 { int v1 = 2;
  namespace ns2 { int v2 = 3;
    namespace ns3 { int v3 = 5;
      }
    }
  }
  using namespace ns1;
  using namespace ns2;
  // using namespace ns1; for global access
int main() { // Qualified by using namespace
  cout << ns1::v1 << ' ';
  cout << ns1::v2 << ' ';
  cout << ns1::ns2::v3 << ' ';
  cout << ns1::v3 << endl;
}
2 3 5 5
```

- Note: With using namespace ns1 before main() the symbols within ns1 will be in the global namespace. Like, ns1::ns2::ns3::v3 can be accessed as ns1::ns2::v3, ns1::v3, and v3
- However, using namespace ns1 belongs to the application space. Hence, the choice of putting it belongs to the user and default version cannot be forced. inline namespace addresses this default enforcement for Version Control



So, here is a comparison of the C++ 03 namespaces the same code as before on the left-hand side and here I have introduced two using, which gives you the same effect. You can use, like you can use everything from ns1 because using namespace ns3 as a part of ns2 says that anything in ns3 is available in ns2, and so similarly for here goes to ns1. Well, if I do that then till up to this point things are comparable.

Now, how do I make this available in global by default? What I will need to do is to put using name space ns1. If I put using name space ns1, then even simply writing v3 will mean this variable, because using, using, using everything is globally available. But that does not address the basic question, because this using namespace ns1 is in the code of the application, so it is up to the choice of the user to put it or not put it. Whereas when you do inline namespace, it is in the space of the library, so it is the choice of the library designer. That is the subtle difference for which the inline namespace feature is very important for proper version control.

(Refer Slide Time: 30:57)

**User-defined Literals: UDTs closer to Built-in Types**

**User-defined Literals: UDTs closer to Built-in Types**

**Sources:**

- User-defined literals, [isocpp.org](http://isocpp.org)
- User Defined Literals in C++, [geeksforgeeks.org](http://geeksforgeeks.org), 2018
- User-defined literals, [microsoft.com](http://microsoft.com), 2021
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses

Programming in Modern C++ Part 16: Primitives 16/21

**User-defined Literals**

- C++ has always provided literals for a variety of built-in types:
 

```

123 // int
1.2 // double
1.2F // float
'a' // char
1ULL // unsigned long long
0xD0 // hexadecimal unsigned
"as" // string

```
- However, in C++03 there are no literals for user-defined types. This violates the principle that UDTs should be supported as well as built-in types are. Common requests include:
 

```

"Hi!"s // std::string, not
        // "zero-terminated array of char"
1.2i // imaginary
123.4567891234df // decimal floating point (IBM)
101010111000101b // binary
123s // seconds
123.56km // not miles! (units)
1234567890123456789012345678901234567890x // extended-precision

```

Programming in Modern C++ Part 6: Primitives 38/64

The next feature is user defined literals where the concern is very simple that we say that user defined types, I mean, in C++ we are building user defined types, but we cannot have literals of the types that we build. We can have only the literals of types that are already given, some examples are given here. Even if I want to have, take certain values and want to give them a specific meaning, say 123s, I want to mean that 123s is 123 seconds, I cannot do that.

(Refer Slide Time: 31:29)

**User-defined Literals: Literal Operators**

- C++11 supports *user-defined literals* through the notion of *literal operators* that map literals with a given suffix into a desired type. For example:
 

```

constexpr complex<double> operator ""_i(long double d) { // imaginary literal
    return complex<double>{ 0.0, static_cast<double>(d) }; // complex is a literal type
} // Note the use of constexpr to enable compile-time evaluation

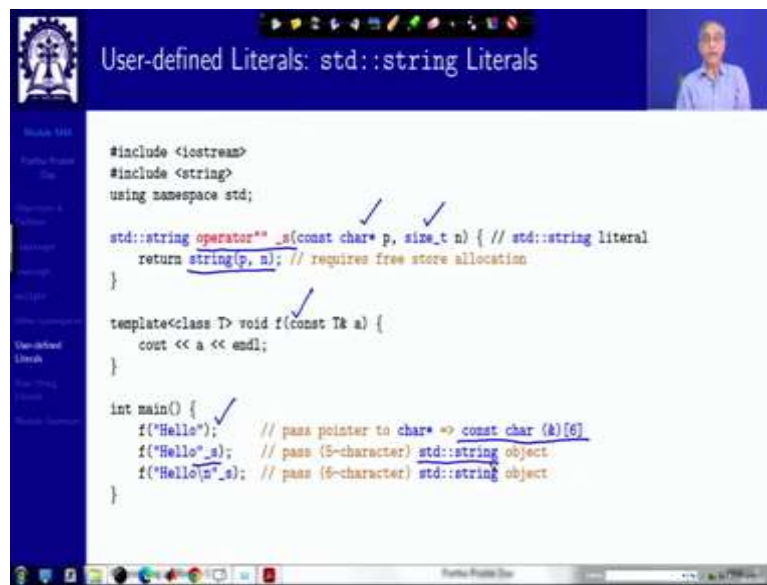
```
- Literal operator has the syntax: `<ReturnType> operator "" <Suffix> (<Parameters>):`
  - `ReturnType` can be anything including void
  - `Suffix` must start with an underscore (`_`). Only the Standard Library is allowed to define literals without the underscore. Suffixes will tend to be short (like `_s` for `string`, `_i` for `imaginary`, `_m` for `meter`, and `_x` for `extended`), so different uses could easily clash. Use namespaces to prevent clashes:
  - Parameters can be any one of four kinds of literals
    - > *Integer literal*: unsigned long long int ✓
    - > *Floating-point literal*: long double ✓
    - > *String literal*: (const char\*, size\_t), (const wchar\_t\*, size\_t), (const char16\_t\*, size\_t), (const char32\_t\*, size\_t), or (char const\*)
    - > *Character literal*: char, wchar\_t, char16\_t, or char32\_t

Part 6: Primitives 39/64

So, user defined literal does that by introducing a new literal operator, which is written in this form, that is operator keyword followed by a pair of double quotes. And then a suffix to be used with the literal conversion. So, in general this operator, so this is a computation, so this operator will have this, then a suffix to actually, which is the name of the user defined literal operator, then it will have parameters of what you are converting from and the return type.

The restrictions are that the suffix has to start with an underscore otherwise non-underscore names are reserved for the standard library and parameters can be of certain types only, either an integer literal or a floating-point literal or a string literal or a character literal, but the return type could be anything you can take any of these and convert to a return type literal of your choice. So, let us see what it does.

(Refer Slide Time: 32:35)



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

std::string operator"" _s(const char* p, size_t n) { // std::string literal
    return string(p, n); // requires free store allocation
}

template<class T> void f(const T& a) {
    cout << a << endl;
}

int main() {
    f("Hello"); // pass pointer to char* => const char (&)[6]
    f("Hello"_s); // pass (5-character) std::string object
    f("Hello\n"_s); // pass (6-character) std::string object
}
```

Say, I define a string, operator literal for the string, so `_s`, all that I do is I just construct a string and pass on. I have a pointer to the character array. I need to pass the size because otherwise I will not be able to do the conversion. So, if I do that and call this templated function `f` with just the double quote, it is a pointer to character. It is an array of character, so it takes as `const char` reference size 6, 5, characters for `hello` and the terminator, but if I write it as `_s`, then it takes as a `std::string` object, it constructs an `std::string` object by calling this literal operator function.

(Refer Slide Time: 33:32)

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

// Note the use of constexpr to enable compile-time evaluation
constexpr complex<double> operator"" _i(long double d) { // imaginary literal
    return complex<double>(0.0, static_cast<double>(d)); // complex is a literal type
}

int main() {
    auto z = 3.0 + 4.0_i;           // complex(3.0, 4.0)
    auto y = 2.3 + 5.0_i;         // complex(2.3, 5.0)
    cout << "z + y = " << z+y << endl; // z + y = (5.3,9)
    cout << "z * y = " << z*y << endl; // z * y = (-13.1,24.2)
    cout << "abs(z) = " << abs(z) << endl; // abs(z) = 5
}
```

Similar thing can be done for say a complex, so I am using an `_i`, the value is taken as long double and I construct a complex object of the standard library, so and then I use it like this. So, I am assuming that the real part is 0 and the imaginary part is provided by this literal conversion, so this literal conversion `3 + 4.0 i` gives me a complex number, 3.0 plus, 3.0, 4.0 because `4.0 _i` literal gives me 0, 4.0, then that is added to 3.00, and I get that.

(Refer Slide Time: 34:21)

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

// user defined literals: kg, g, and mg
long double operator"" kg(long double x) { // Kilogram: to gram
    return x * 1000;
}
long double operator"" g(long double x) { // Gram
    return x;
}
long double operator"" mg(long double x) { // Milligram: to gram
    return x / 1000;
}

int main() {
    long double weight = 3.6_kg;
    cout << weight << endl; // 3600
    cout << setprecision(8) << (weight + 2.3_g) << endl; // 3600.0023
    cout << (32.3_kg / 2.0_g) << endl; // 16150
    cout << (32.3_mg * 2.0) << endl; // 0.0646
}
```

Conversions can be done with meaningful extension symbols like kg, I say okay. I have to deal with the kilogram, gram, milligram, so I have literals of every kind and I will put a value and put the literal operator. So, the basic representation is in gram. For kilogram we multiply that value by thousand, for milligram I divide it by 1000. So, I say 3.6 underscore kg is my

literal, so that means 3600, because it will call this literal operator and convert into grams which is 3600 grams and similar for milligrams and all those, so this makes the code naturally lot more readable and understandable.

(Refer Slide Time: 35:04)



```
#include <iostream>
#include <string>
using namespace std;
// User-defined Date class
class Date { int date, month, year;
public: Date(int d = 1, int m = 1, int y = 0): date(d), month(m), year(y) { }
    friend ostream operator<<(ostream os, const Date& d) {
        os << d.date << "/" << d.month << "/" << d.year; return os;
    }
};
// Literal operator for Date
Date operator ""_ad(const char* s, size_t) { // representation of date as "dd/mm/yyyy", format
    // parsing s into dd, mm, yyyy as int
    char *str = strdup(s); // copy needed as s is const char* - strtok cannot work on s
    char *date_str = strtok(str, "/"); int date = atoi(date_str);
    date_str = strtok(NULL, "/"); int month = atoi(date_str);
    date_str = strtok(NULL, "/"); int year = atoi(date_str);
    free(str);

    return Date{ date, month, year }; // Date is a literal type
}

int main() {
    auto myDate = "08/02/2022"_ad; // Date object created from literal
    cout << myDate << endl;
}
```

And you can actually use this to convert to user defined types like here run this example and check for yourself, I have defined a Date type, date, month, year and I want that this kind of a date literal I would have dd slash yy slash mm slash yyyy and convert it directly to date object. This conversion function, the operator literal conversion function allows me to do that. Just go through this carefully it has a lot of parsing from the string, it takes it as a string, does a parsing, take out different parts, convert them into integer and finally, it constructs the date object as a literal.

(Refer Slide Time: 35:49)



**User-defined Literals**

- The basic (implementation) idea is
  - After parsing what could be a *literal*, the compiler always checks for a *suffix*
  - The user-defined literal mechanism simply *allows the user to specify a new suffix* and what is to be done with the literal before it
  - *It is not possible to redefine the meaning of a built-in literal suffix or augment the syntax of literals*
  - A literal operator can request to get its (preceding) literal passed
    - ▷ as *cooked* (with the value it would have had if the new suffix had not been defined) or
    - ▷ as *uncooked* (as a string) by simply requesting a single `const char*` argument:

```
Bignum operator"" x(const char* p) {  
    return Bignum(p);  
}  
void f(Bignum);  
f(1234567890123456789012345678901234567890x);
```

Here the C-style string "1234567890123456789012345678901234567890" is passed to `operator"" x()`. Note that we did not explicitly put those digits into a string

Programming in Modern C++ Partha Pratim Das 33/33

So, user defined literal is a very, very powerful mechanism, there are ways to either take the converted literal form or the raw string form in which it is provided, all of these examples are given here.

(Refer Slide Time: 36:05)



**Raw String Literals**

**Sources:**

- Raw string literals, [isocpp.org](http://isocpp.org)
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses

Programming in Modern C++ Partha Pratim Das 33/33

**Raw String Literals**

- String literals where special characters are *not* special:
  - For example, escaped characters and double quotes:
 

```
std::string noNewlines(R"(\n\n)");
std::string cmd(R"(ls /home/docs | grep ".pdf")");
```
  - For example, newlines:
 

```
std::string withNewlines(R"(
    Line 1 of the string...
    Line 2...
    Line 3)");
```
- Rawness may be added to any string encoding:
  - LR (Raw Wide string literal \t (without a tab))
  - u8R (Raw UTF-8 string literal \n (without a newline))
  - uR (Raw UTF-16 string literal \\ (with two backslashes))
  - UR (Raw UTF-32 string literal 2620 (w/o a skull & crossbones))
- Raw text delimiters may be customized:
  - Useful when `]"` is in raw text, for example, in regular expressions:
 

```
std::regex re1(R"(*operator!*operator->)*"); // "operator()!*operator->"
std::regex re2(R"xyzzy*([A-Za-z_]\w*)*xyzzy"); // "(identifier)* \w"
```

Before I end the last feature, I would like to mention is raw string, you know that c strings have lot of escape characters, like `\t` is written to mean tab, `\n` is written to mean new line, and so on. C++ allows you to define a raw string. It is written with a prefix capital R before the double quotes, which means that the escape characters will not be interpreted, they will not be treated as special.

So, this will be `\n` and `\n` not to new line. Similarly, I can write a string like this, which spreads over multiple lines. There are new lines inside. If you try to do this in a normal string you will have error, but here you will be able to do that because everything is being taken just as the character is without any special interpretation being given to that. So, this becomes useful in some of the standard library features as we will see later on.

(Refer Slide Time: 37:06)

**Module Summary**

- Introduced following C++11 general features:
  - constexpr
  - noexcept
  - nullptr
  - Inline namespace
  - User-defined Literals
  - Raw String Literals

Programming in Modern C++ | Part 6: Primitives | 888/31

So, we have introduced a number of general features, these six, so please practice them through programming. Thank you very much for your attention and we will meet in the next module.