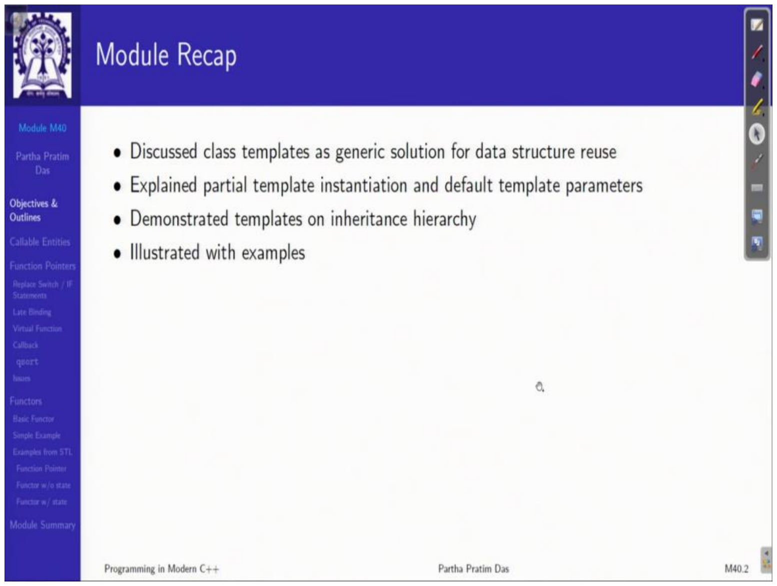
Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 40
Functors: Function Objects

Welcome to Programming in Modern C++. We are in week 8 and in the last module of week 8, module 40.

(Refer Slide Time: 0:37)



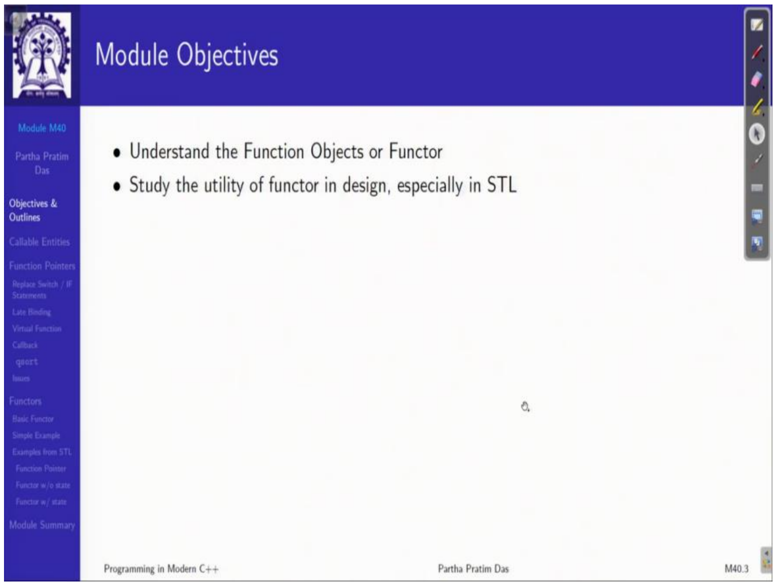
Module Recap

- Discussed class templates as generic solution for data structure reuse
- Explained partial template instantiation and default template parameters
- Demonstrated templates on inheritance hierarchy
- Illustrated with examples

Module M40
Partha Pratim Das
Objectives & Outlines
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
qsort
Issues
Functors
Basic Functor
Simple Example
Examples from STL
Function Pointer
Functor w/o state
Functor w/ state
Module Summary
Programming in Modern C++
Partha Pratim Das
M40.2

In the last two modules, we have discussed about templates as a generic solution for data structure reuse, and looked at various algorithms for template instantiation and template parameters and how to use class template in inheritance hierarchy had examples.

(Refer Slide Time: 0:54)



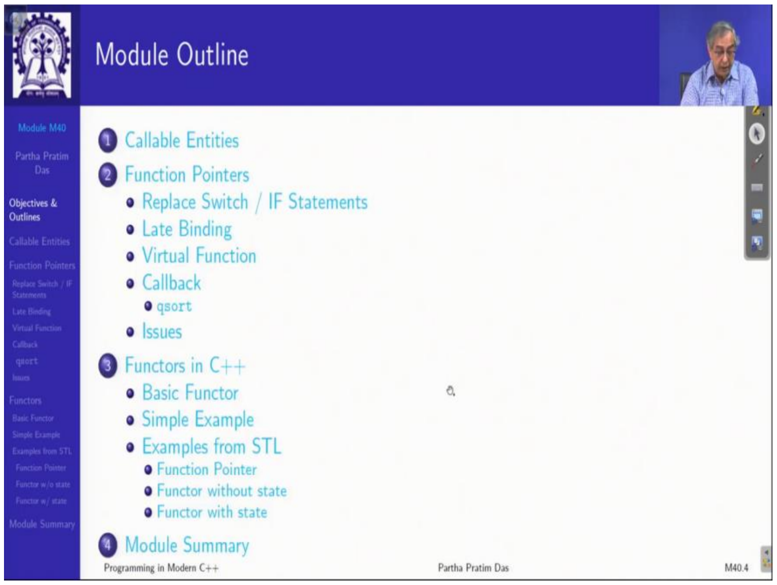
Module Objectives

- Understand the Function Objects or Functor
- Study the utility of functor in design, especially in STL

Programming in Modern C++ Partha Pratim Das M40.3

In the present module, we are going to look at something new, new for your thoughts that is called Functional Objects or Functors. And what is their utility in the Standard Template Library.

(Refer Slide Time: 01:11)



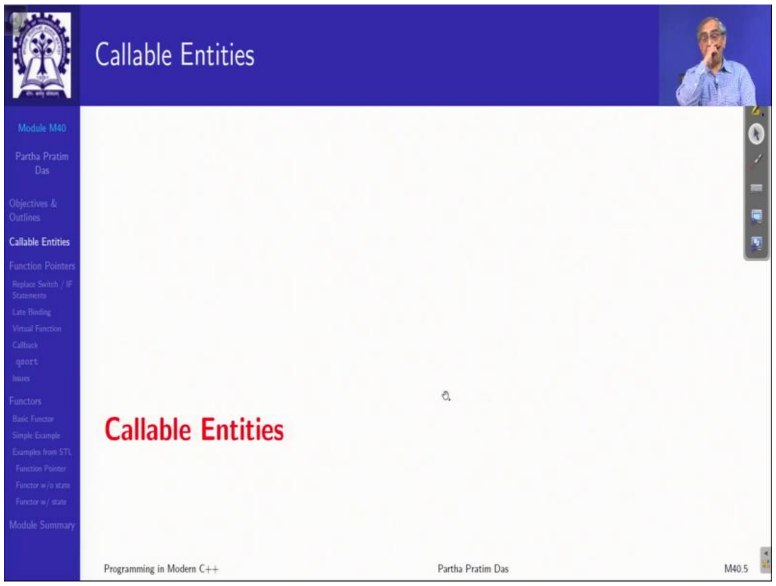
Module Outline

1. Callable Entities
2. Function Pointers
   - Replace Switch / IF Statements
   - Late Binding
   - Virtual Function
   - Callback
     - qsort
   - Issues
3. Functors in C++
   - Basic Functor
   - Simple Example
   - Examples from STL
     - Function Pointer
     - Functor without state
     - Functor with state
4. Module Summary

Programming in Modern C++ Partha Pratim Das M40.4

This will be the outline which will be available.

(Refer Slide Time: 01:14)



Callable Entities

Module M40

Partha Pratim Das

Objectives & Outlines

Callable Entities

Function Pointers

Replace Switch / IF Statements

Late Binding

Virtual Function

Callback

qsort

Issues

Functors

Basic Functor

Simple Example

Examples from STL

Function Pointer

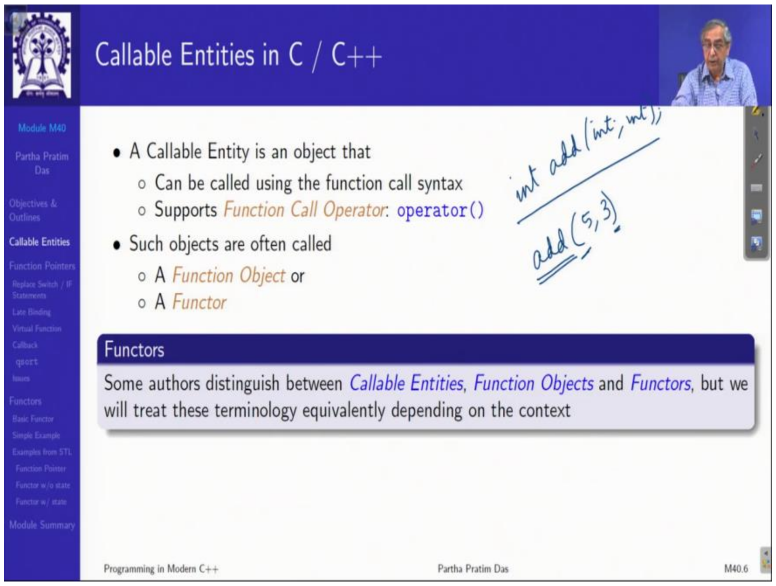
Functor w/o state

Functor w/ state

Module Summary

Callable Entities

Programming in Modern C++ Partha Pratim Das M40.5



Callable Entities in C / C++

- A Callable Entity is an object that
  - Can be called using the function call syntax
  - Supports *Function Call Operator*: `operator()`
- Such objects are often called
  - A *Function Object* or
  - A *Functor*

int add(int, int);

add(5, 3)

Functors

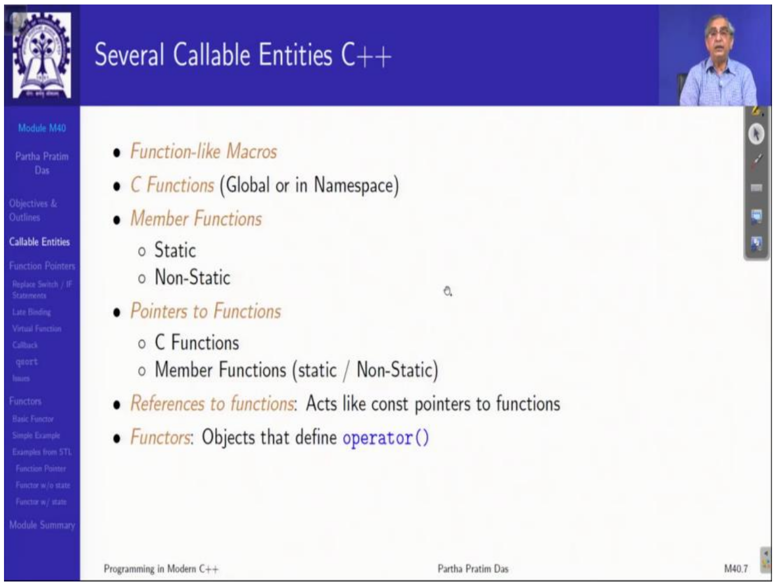
Some authors distinguish between *Callable Entities*, *Function Objects* and *Functors*, but we will treat these terminology equivalently depending on the context

Programming in Modern C++ Partha Pratim Das M40.6

Now, first as we mentioned about function, we will use a little bit of a generic term say callable entities, something that can be called. So, I say a callable entity is an object that can be used in the same syntax as a function call and it will support in a way the function call operator simple parenthesis that is how we get to know that something is a function.

We say int add(int, int) this is the signature and then we say add(5, 3). So, what tells me that add is a callable entity this pair of parentheses so, that is called the function call operator. These are typically known as function objects or functors also. Some authors distinguish between them with subtle differences in meaning, we will use them interchangeably depending on the context.

(Refer Slide Time: 02:22)

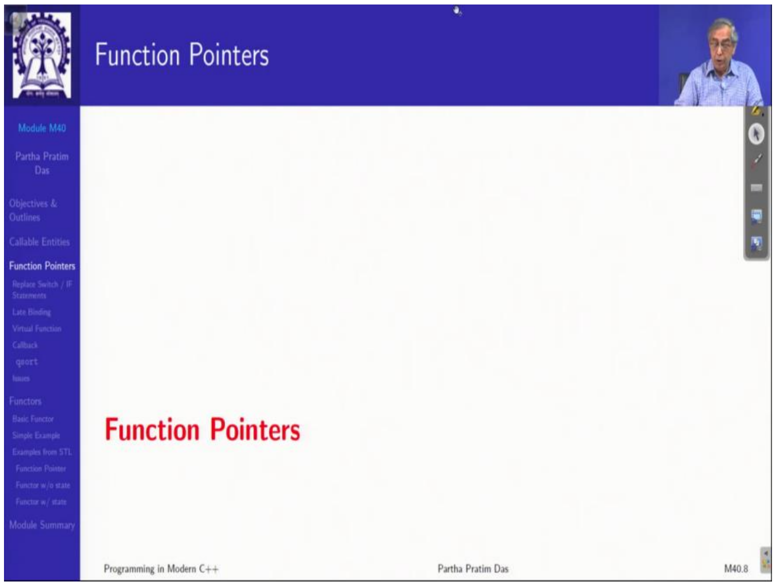


Several Callable Entities C++

- *Function-like Macros*
- *C Functions* (Global or in Namespace)
- *Member Functions*
  - Static
  - Non-Static
- *Pointers to Functions*
  - C Functions
  - Member Functions (static / Non-Static)
- *References to functions*: Acts like const pointers to functions
- *Functors*: Objects that define `operator()`

Now, there are several callable entities in C++ which look like a function and call something and does similar stuff, naturally, you have seen function-like macros that are C functions global or in the in the namespace, you have member functions you have static, non-static pointer to function, member functions which are static, non-static and so on. So for pointers to member functions that are static, non-static you can have references and you have functors which are objects that define a function call operator.

(Refer Slide Time: 03:06)



Function Pointers

Module M40

Partha Pratim Das

Objectives & Outlines

Callable Entities

Function Pointers

Replace Switch / IF Statements

Late Binding

Virtual Function

Callback

qsort

Issues

Functors

Basic Functor

Simple Example

Examples from STL

Function Pointer

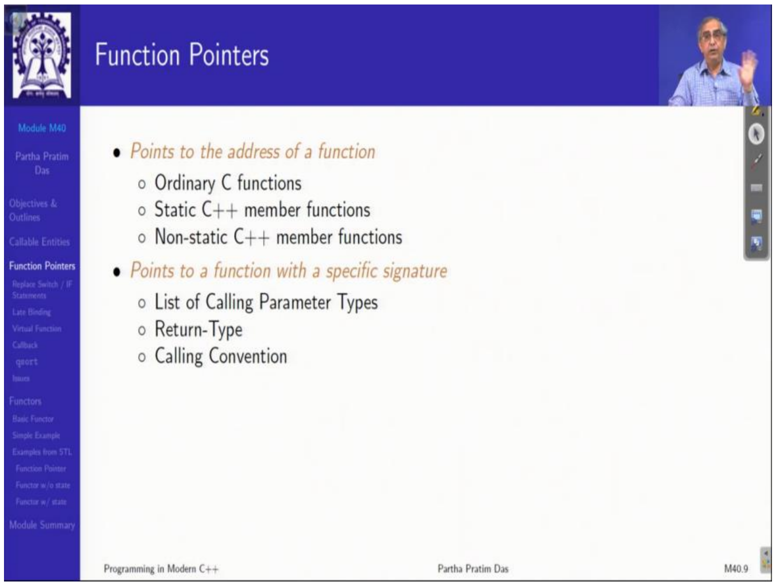
Functor w/o state

Functor w/ state

Module Summary

Function Pointers

Programming in Modern C++ Partha Pratim Das M40.8



Function Pointers

Module M40

Partha Pratim Das

Objectives & Outlines

Callable Entities

Function Pointers

Replace Switch / IF Statements

Late Binding

Virtual Function

Callback

qsort

Issues

Functors

Basic Functor

Simple Example

Examples from STL

Function Pointer

Functor w/o state

Functor w/ state

Module Summary

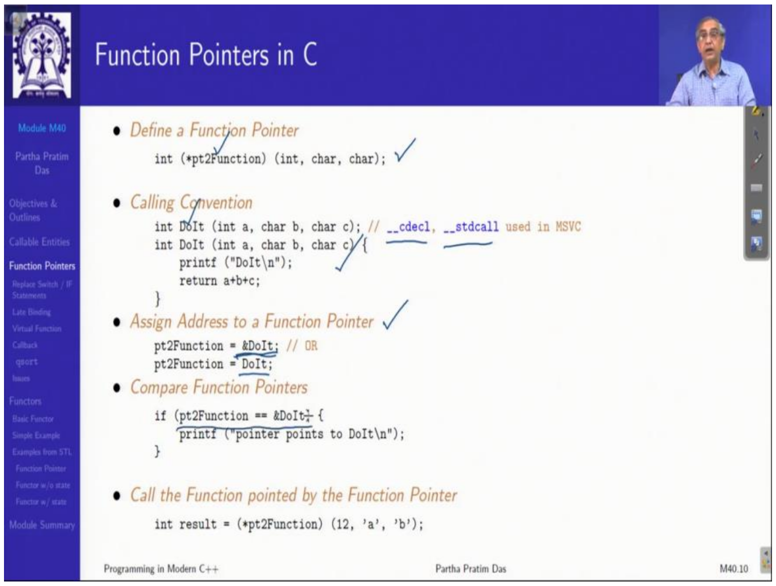
- *Points to the address of a function*
  - Ordinary C functions
  - Static C++ member functions
  - Non-static C++ member functions
- *Points to a function with a specific signature*
  - List of Calling Parameter Types
  - Return-Type
  - Calling Convention

Programming in Modern C++ Partha Pratim Das M40.9

So, firstly let us look at function pointers which have which have a very similar behavior. So, what is a function pointer, we have already studied it is address of a function, it could be address of an ordinary function that is a C global function. It could be static C++ member function, it could be non-static C++ member functions.

So, these are the three categories that we have seen. Leaving aside macros which are more syntactically like a function, but it is not exactly a function because it does not go to the stack frame and all those protocols. So, a function pointer points to a function with specific signature, it needs to know what is the signature, that is list of calling parameter types, the return type, the calling convention, whether it is a C convention or a C++ convention or some other specific convention, which is defined by the vendor and so on, so forth.

(Refer Slide Time: 04:51)



Function Pointers in C

Module M40

Partha Pratim Das

Objectives & Outlines

Callable Entities

Function Pointers

Replace Switch / IF Statements

Late Binding

Virtual Function

Callback

qsort

Issues

Functors

Basic Functor

Simple Example

Examples from STL

Function Pointer

Functor w/o state

Functor w/ state

Module Summary

- *Define a Function Pointer*

```
int (*pt2Function) (int, char, char);
```

- *Calling Convention*

```
int DoIt (int a, char b, char c); // __cdecl, __stdcall used in MSVC
int DoIt (int a, char b, char c) {
    printf ("DoIt\n");
    return a+b+c;
}
```

- *Assign Address to a Function Pointer*

```
pt2Function = &DoIt; // OR
pt2Function = DoIt;
```

- *Compare Function Pointers*

```
if (pt2Function == &DoIt) {
    printf ("pointer points to DoIt\n");
}
```

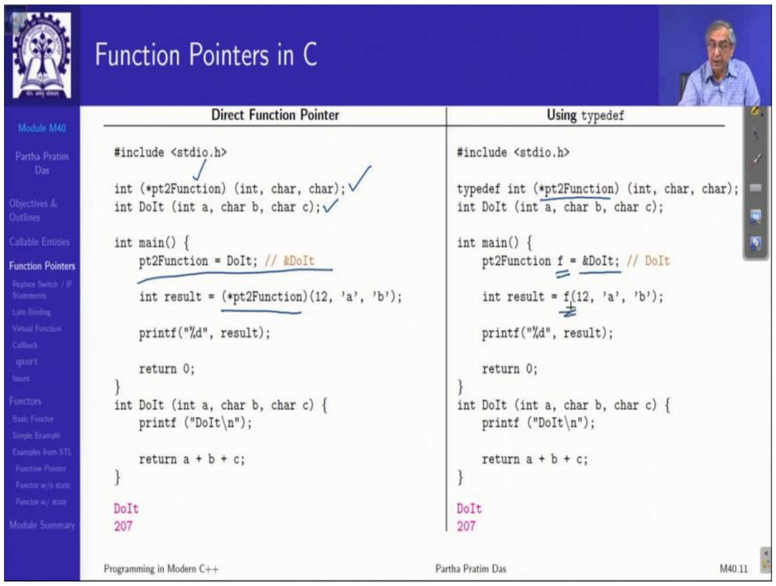
- *Call the Function pointed by the Function Pointer*

```
int result = (*pt2Function) (12, 'a', 'b');
```

Programming in Modern C++ Partha Pratim Das M40.10

So, these are typical function pointers in C, you can define a function pointer and use that you can use different calling conventions here there will be C calling convention which is C declaration. Microsoft Visual C++ use some additional calling conventions for the Windows API particularly, you can assign address to a function pointer, you have a function pointer here you have a function, you can assign that address by doing ampersand and I had mentioned earlier as a shortcut, since, you can only take the address of a function nothing else if you just write the name of the function also it is taken to be the address of the function. Then you can compare two function pointers and get true or false and call a function invoke a function pointed to by a function pointer. So, these are these are the things that a function pointer typically will allow you to do.

(Refer Slide Time: 05:41)



Function Pointers in C

Module M40

Partha Pratim Das

Objectives & Outlines

Callable Entities

Function Pointers

Replace Switch / IF Statements

Late Binding

Virtual Function

Callback

qsort

Issues

Functors

Basic Functor

Simple Example

Examples from STL

Function Pointer

Functor w/o state

Functor w/ state

Module Summary

Direct Function Pointer

```
#include <stdio.h>

int (*pt2Function) (int, char, char);
int DoIt (int a, char b, char c);

int main() {
    pt2Function = DoIt; // &DoIt

    int result = (*pt2Function)(12, 'a', 'b');

    printf("%d", result);

    return 0;
}
int DoIt (int a, char b, char c) {
    printf ("DoIt\n");

    return a + b + c;
}
```

```
DoIt
207
```

Using typedef

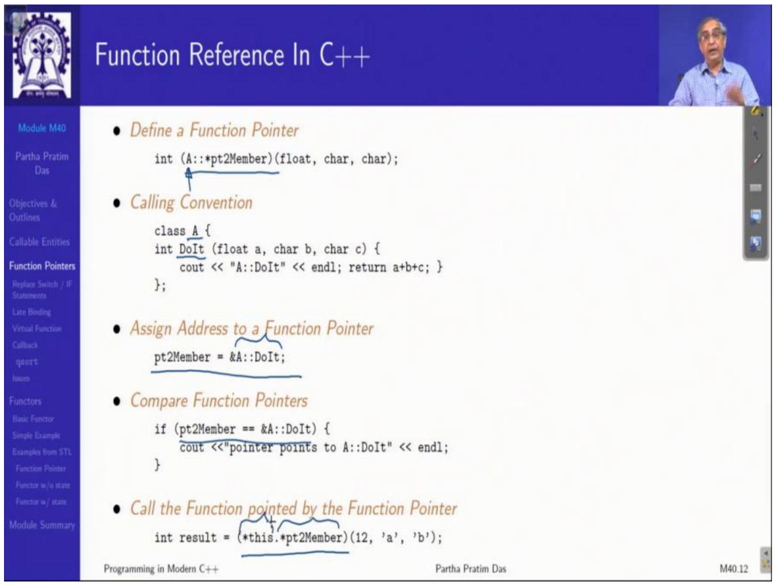
```
#include <stdio.h>

typedef int (*pt2Function) (int, char, char);
int DoIt (int a, char b, char c);

int main() {
    pt2Function f = &DoIt; // DoIt

    int result = f(12, 'a', 'b');

    printf("%d", result);

    return 0;
}
int DoIt (int a, char b, char c) {
    printf ("DoIt\n");

    return a + b + c;
}
```

```
DoIt
207
```

Programming in Modern C++ Partha Pratim Das M40.11

So, function pointers in in C, you will do it in two ways possibly one is directly you have defined a function pointer, you have a function you assign it to the function pointer that you have defined here and then use that function pointer to dereference and invoke the function or you can type def it by this function pointer name. So, you are aliasing with the type then define a function pointer f of that type set a function pointer to that set a function to that and invoke f. So, these are the two typical ways that you will be using it in C.

(Refer Slide Time: 07:27)



Function Reference In C++

Module M40

Partha Pratim Das

Objectives & Outlines

Callable Entities

Function Pointers

Replace Switch / IF Statements

Late Binding

Virtual Function

Callback

qsort

Issues

Functors

Basic Functor

Simple Example

Examples from STL

Function Pointer

Functor w/o state

Functor w/ state

Module Summary

- *Define a Function Pointer*

```
int (A::*pt2Member)(float, char, char);
```

- *Calling Convention*

```
class A {
int DoIt (float a, char b, char c) {
    cout << "A::DoIt" << endl; return a+b+c; }
};
```

- *Assign Address to a Function Pointer*

```
pt2Member = &A::DoIt;
```

- *Compare Function Pointers*

```
if (pt2Member == &A::DoIt) {
    cout <<"pointer points to A::DoIt" << endl;
}
```

- *Call the Function pointed by the Function Pointer*

```
int result = (*this.*pt2Member)(12, 'a', 'b');
```

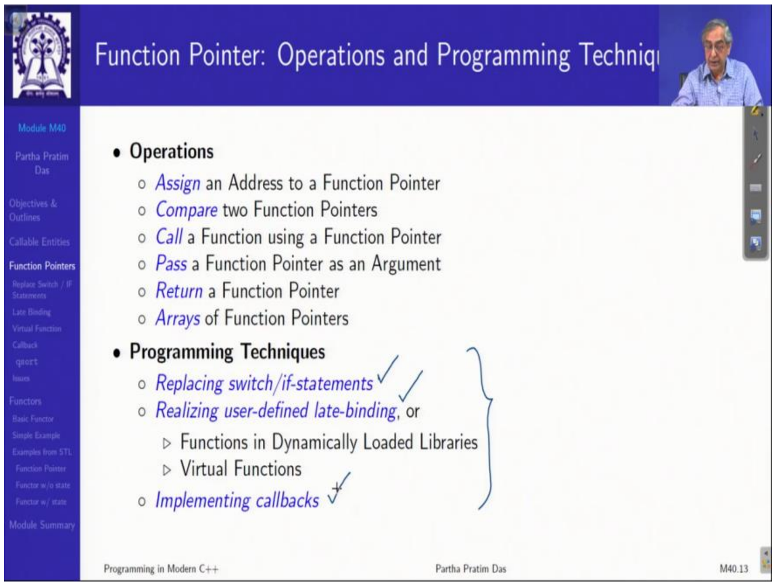
Programming in Modern C++ Partha Pratim Das M40.12

How about in C++, in C++ you can have function pointers as well as you can have function references. So, you can have function pointers like you had in C, but you can also have function pointers, which are of a certain specific class a class member. So, you are saying that this is a function pointer A::*pt2Member says that pt2Member is a function pointer to a member function of class A because a member function of class A only can be referred to as A:: where A is some class.

So, in that if I define a class A and have a member function there, I will have a calling convention of C++ in this case, then I can assign the actual DoIt function to the function pointer, by taking its address the name of the function is A::DoIt I do ampersand to get the address and put it to the function pointer, mind you this function pointer is defined for a member function of a so, I cannot take a member function of another class B and assign it here that type error.

I can compare two function pointers provided they match in their type, I can dereference the function pointer and I can actually use it for invocation. So, what I am doing is I am dereferencing this to get the function, I am dereferencing this pointer to get the object, then the object (.) dot member function is an invocation. That is a simple way of doing this just passing back into the basic syntax.

(Refer Slide Time: 08:26)



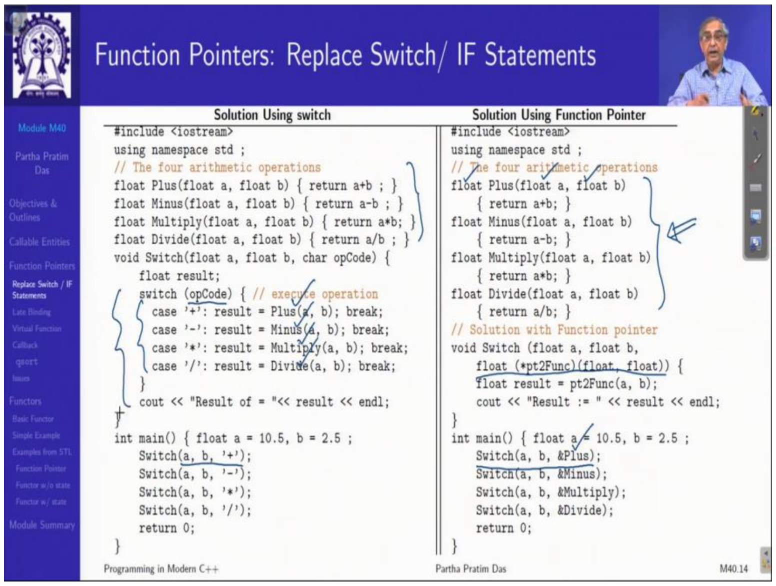
Function Pointer: Operations and Programming Techniq

- **Operations**
  - *Assign* an Address to a Function Pointer
  - *Compare* two Function Pointers
  - *Call* a Function using a Function Pointer
  - *Pass* a Function Pointer as an Argument
  - *Return* a Function Pointer
  - *Arrays* of Function Pointers
- **Programming Techniques**
  - *Replacing switch/if-statements* ✓ ✓
  - *Realizing user-defined late-binding*, or
    - ▷ Functions in Dynamically Loaded Libraries
    - ▷ Virtual Functions
  - *Implementing callbacks* ✓

Programming in Modern C++ Partha Pratim Das M40.13

So, if we now summarize that, what are the things that the function pointer can do the operations we see that we can assign, we can compare two function pointers, we can call a function, we can pass a function pointer as a parameter, we can return a function pointer from a function, we can have an area of function pointers, and these are involved in a variety of programming techniques that we can replace and switch if-statement. If this type of this kind of things can be replaced by function pointers, it can be used for late binding or it can be used in the case of callbacks. So, these are all very similar mechanisms, but all of them use function pointer and used in different programming context.

(Refer Slide Time: 10:03)



Function Pointers: Replace Switch/ IF Statements

Module M40

Partha Pratim Das

Objectives & Outlines
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
qsort
Issues
Functors
Basic Functor
Simple Example
Examples from STL
Function Pointer
Functor w/o state
Functor w/ state
Module Summary

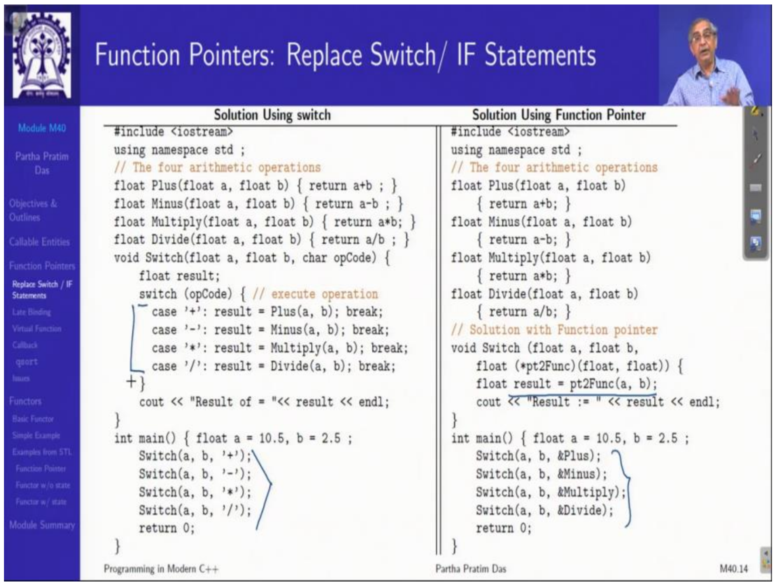
Solution Using switch

```
#include <iostream>
using namespace std ;
// The four arithmetic operations
float Plus(float a, float b) { return a+b ; }
float Minus(float a, float b) { return a-b ; }
float Multiply(float a, float b) { return a*b; }
float Divide(float a, float b) { return a/b ; }
void Switch(float a, float b, char opCode) {
    float result;
    switch (opCode) { // execute operation
      case '+': result = Plus(a, b); break;
      case '-': result = Minus(a, b); break;
      case '*': result = Multiply(a, b); break;
      case '/': result = Divide(a, b); break;
    }
    cout << "Result of = "<< result << endl;
}
int main() { float a = 10.5, b = 2.5 ;
    Switch(a, b, '+');
    Switch(a, b, '-');
    Switch(a, b, '*');
    Switch(a, b, '/');
    return 0;
}
```

Solution Using Function Pointer

```
#include <iostream>
using namespace std ;
// The four arithmetic operations
float Plus(float a, float b)
    { return a+b; }
float Minus(float a, float b)
    { return a-b; }
float Multiply(float a, float b)
    { return a*b; }
float Divide(float a, float b)
    { return a/b; }
// Solution with Function pointer
void Switch (float a, float b,
    float (*pt2Func)(float, float)) {
    float result = pt2Func(a, b);
    cout << "Result := " << result << endl;
}
int main() { float a = 10.5, b = 2.5 ;
    Switch(a, b, &Plus);
    Switch(a, b, &Minus);
    Switch(a, b, &Multiply);
    Switch(a, b, &Divide);
    return 0;
}
```

Programming in Modern C++ Partha Pratim Das M40.14



Function Pointers: Replace Switch/ IF Statements

Module M40

Partha Pratim Das

Objectives & Outlines
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
qsort
Issues
Functors
Basic Functor
Simple Example
Examples from STL
Function Pointer
Functor w/o state
Functor w/ state
Module Summary

Solution Using switch

```
#include <iostream>
using namespace std ;
// The four arithmetic operations
float Plus(float a, float b) { return a+b ; }
float Minus(float a, float b) { return a-b ; }
float Multiply(float a, float b) { return a*b; }
float Divide(float a, float b) { return a/b ; }
void Switch(float a, float b, char opCode) {
    float result;
    switch (opCode) { // execute operation
      case '+': result = Plus(a, b); break;
      case '-': result = Minus(a, b); break;
      case '*': result = Multiply(a, b); break;
      case '/': result = Divide(a, b); break;
    }
    cout << "Result of = "<< result << endl;
}
int main() { float a = 10.5, b = 2.5 ;
    Switch(a, b, '+');
    Switch(a, b, '-');
    Switch(a, b, '*');
    Switch(a, b, '/');
    return 0;
}
```

Solution Using Function Pointer

```
#include <iostream>
using namespace std ;
// The four arithmetic operations
float Plus(float a, float b)
    { return a+b; }
float Minus(float a, float b)
    { return a-b; }
float Multiply(float a, float b)
    { return a*b; }
float Divide(float a, float b)
    { return a/b; }
// Solution with Function pointer
void Switch (float a, float b,
    float (*pt2Func)(float, float)) {
    float result = pt2Func(a, b);
    cout << "Result := " << result << endl;
}
int main() { float a = 10.5, b = 2.5 ;
    Switch(a, b, &Plus);
    Switch(a, b, &Minus);
    Switch(a, b, &Multiply);
    Switch(a, b, &Divide);
    return 0;
}
```

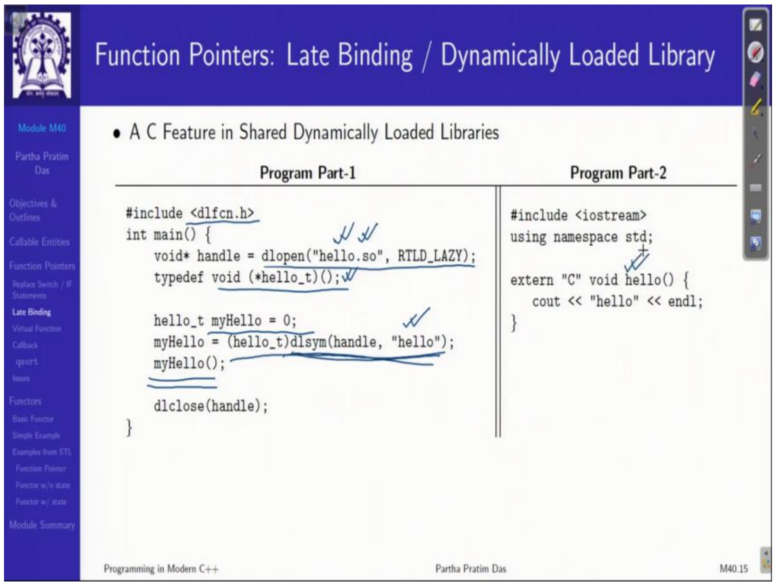
Programming in Modern C++ Partha Pratim Das M40.14

So, these examples we have seen before, but I am just bringing them in the context of the of the function pointers. So, here, I am trying to evaluate an expression involving the different numbers and operators. And I have 4 different functions for doing that. And I have an opCode, which says which operator is this, depending on the operator, I call one of these functions. In the other case, again, we have all these functions but what we do is we define a function pointer generically, which takes two floats and returns a float, which is the common type signature of all these 4 different operation functions.

And then what I do is, instead of doing a switch on the just type, we say that we will do a switch on this function, ampersand plus so instead of writing it through the code, within quotes, plus, I write it as a function pointer for plus, so I am saying that take a b and do a plus take a b, do a minus right that. So, this kind of switch which is a nesting of if-else statement is replaced here, because here you see there is actually no switch remaining.

All that is remaining is just call this function pointer, because in that function pointer you have already specified the actual function. So, here what you need as a two level, first specifying the code and then doing an if-else can be simplified and made more semantically meaningful. So, this is the first context in which function pointers can be used.

(Refer Slide Time: 13:11)



Function Pointers: Late Binding / Dynamically Loaded Library

- A C Feature in Shared Dynamically Loaded Libraries

Program Part-1

```
#include <dlfcn.h>
int main() {
    void* handle = dlopen("hello.so", RTLD_LAZY);
    typedef void (*hello_t)();

    hello_t myHello = 0;
    myHello = (hello_t)dlsym(handle, "hello");
    myHello();

    dlclose(handle);
}
```

Program Part-2

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

extern "C" void hello() {
    cout << "hello" << endl;
}
```

Programming in Modern C++ Partha Pratim Das M40.15

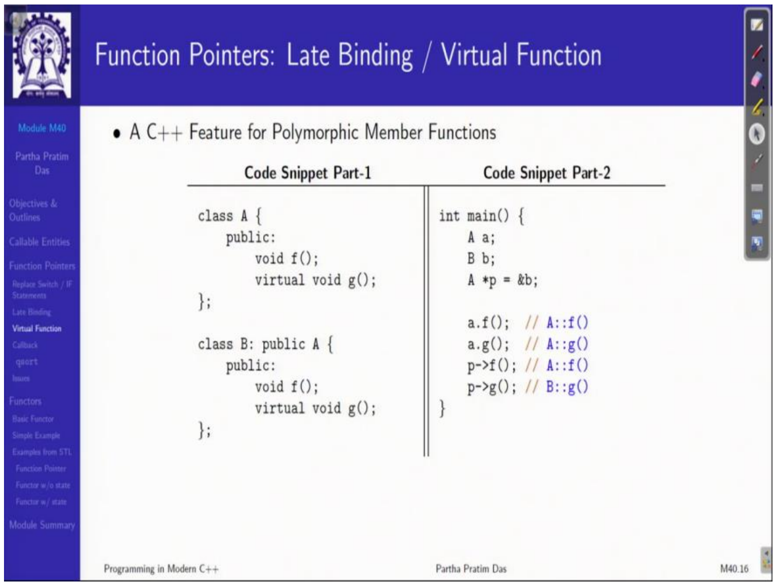
The other is what is known as late binding. Late binding is I talked about this in some other context as well and we have a separate tutorial on this as well. So, well that we are saying is, if I am trying to do a dynamic binding of my function, dynamic linking of my function, what does that mean that I do not want to put the function as a part of my executable code, I just want to know that this is my signature and at the runtime, I will go to the system and find out what is the available implementation of that function and call that.

So, the normal function binding is static, it is early because as soon as I see the function, I need to know what is a function body here I am saying no, all that I need to know is just like the compile time I just need to know what is the signature, but the function body can be supplied later on, which means it can be replaced even after I have written the code.

So, for that the mechanism is to use a specific header dynamic linking function is the name of the of the header dlfcn.h and with that, you open your library and get a define a function pointer which takes a void gives me a void because you do not know in general what the function can be so, you assume that it is only can be void. And then what you do you provide the name of the function that you want so that it can go to the library, go to the library and find out a function by that name and give you that pointer and then you call that function.

So, this function which you find in the library may have been generated after you have written the code for calling it which is not the case with the early binding. So, when you do that, you have got it refines and returns a void* pointer which you cast to your function pointer type and then you call that particular based on that particular function pointer, which will actually call the hello function that you have provided at a later stage. So, this is the, this late binding is required to provide any user defined linkages to any user defined code at a later stage.

(Refer Slide Time: 13:27)



Function Pointers: Late Binding / Virtual Function

- A C++ Feature for Polymorphic Member Functions

Code Snippet Part-1

```
class A {
    public:
        void f();
        virtual void g();
};

class B: public A {
    public:
        void f();
        virtual void g();
};
```

Code Snippet Part-2

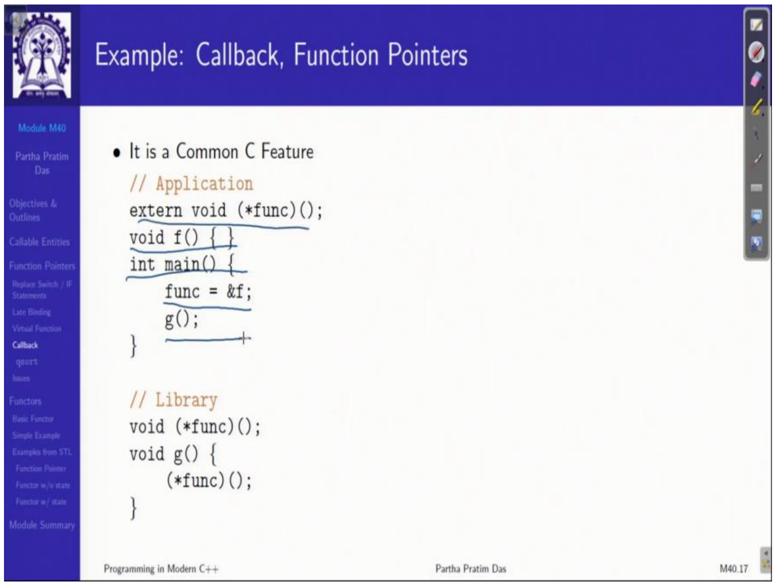
```
int main() {
    A a;
    B b;
    A *p = &b;

    a.f();  // A::f()
    a.g();  // A::g()
    p->f(); // A::f()
    p->g(); // B::g()
}
```

Programming in Modern C++ Partha Pratim Das M40.16

You have also seen late binding in terms of virtual functions, because you have seen that if you have a virtual function, then you do not actually know the compiler actually does not know which function it is going to invoke, whether it is the base class function or the derived class function, it depends on the type of object that it is actually pointing to.

So, both of these situations are in two different contexts, but both of them actually give you the same meet the same functionality of late binding, which means that the function to be called is not decided at the compile time, but it is decided at the runtime given what particular function body is available. In the first case, you find out by searching the library explicitly, in the second case is happens to the virtual function table and the runtime type of an object, but they are the same. So, this is the other context where function pointers are used.

(Refer Slide Time: 14:49)



Example: Callback, Function Pointers

- It is a Common C Feature

```
// Application
extern void (*func)();
void f() { }
int main() {
    func = &f;
    g();
}

// Library
void (*func)();
void g() {
    (*func)();
}
```

Module M40

Partha Pratim Das

Objectives & Outlines

Callable Entities

Function Pointers

Replace Switch / IF Statements

Late Binding

Virtual Function

Callback

qsort

Issues

Functors

Basic Functor

Simple Example

Examples from STL

Function Pointer

Functor w/o state

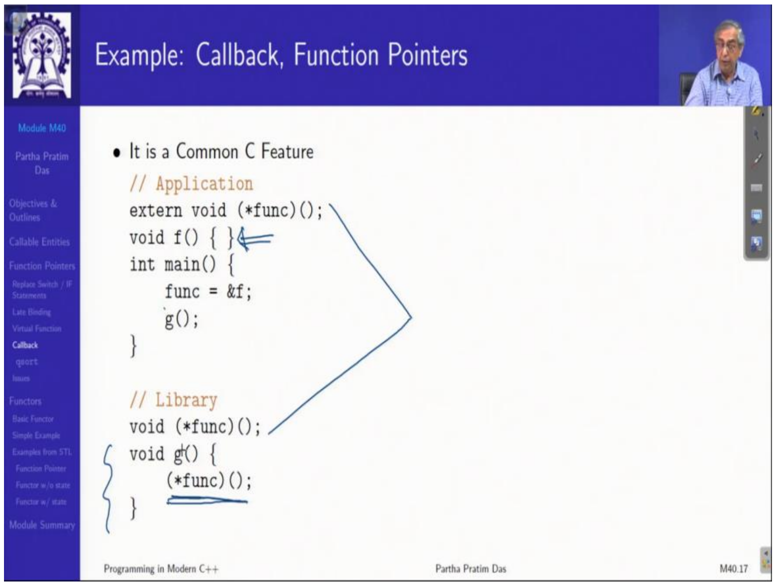
Functor w/ state

Module Summary

Programming in Modern C++ Partha Pratim Das M40.17

So, combining this you can also have callbacks in the applications using function pointers. So, what do I have, I have a main, I have a function f, which actually does nothing. I take that I have a function pointer and I call the function g. Now, this is my this is my main, so this is my application code.

(Refer Slide Time: 15:56)



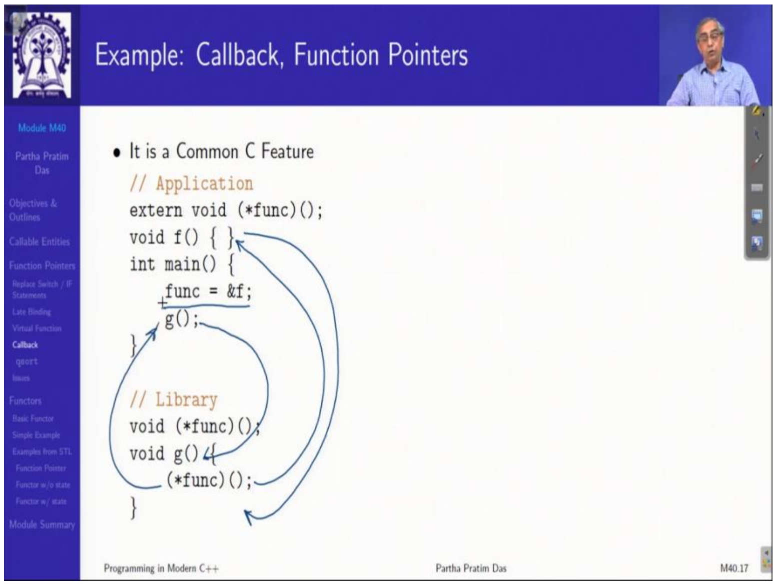
Example: Callback, Function Pointers

- It is a Common C Feature

```
// Application
extern void (*func)();
void f() { }
int main() {
    func = &f;
    g();
}

// Library
void (*func)();
void g() {
    (*func)();
}
```

In the library what do we have I have the same function pointer and the signature is very simple - take a void return a void this is all that it has. And I say that the library has provided me a function g. Now, this function g needs to know the function func which is the only the user knows. So, until the user has provided that the library cannot actually do anything. Now, naturally you cannot make it a function directly because then to build the library you will require the body of that function which is does not exist. So, you are u making use of this function pointer here. Now, what happens if main knows that the library will need to know a function pointer which is known as func.

(Refer Slide Time: 16:30)



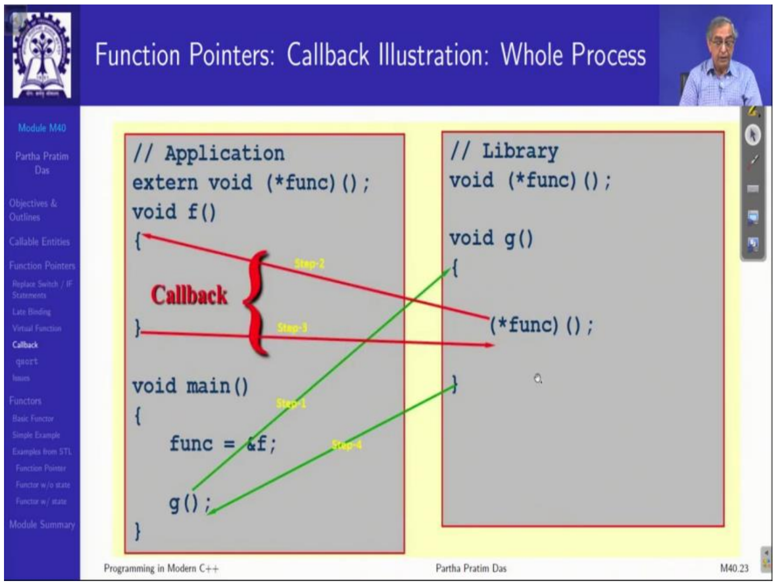
Example: Callback, Function Pointers

- It is a Common C Feature

```
// Application
extern void (*func)();
void f() { }
int main() {
    func = &f;
    g();
}

// Library
void (*func)();
void g() {
    (*func)();
}
```

So, the user application programmer will need to set that func to the function which wants get called from the library. So, the way it happens is here from main you call g and g calls this function back it goes back and then it comes up. So, the library is achieving its objectives by using a function which you provide to it at a later point of time and in invocation you can have a different search function. So, this is exactly what we saw in terms of things like quicksort that is provided by the C standard library.

(Refer Slide Time: 17:58)



Function Pointers: Callback Illustration: Whole Process

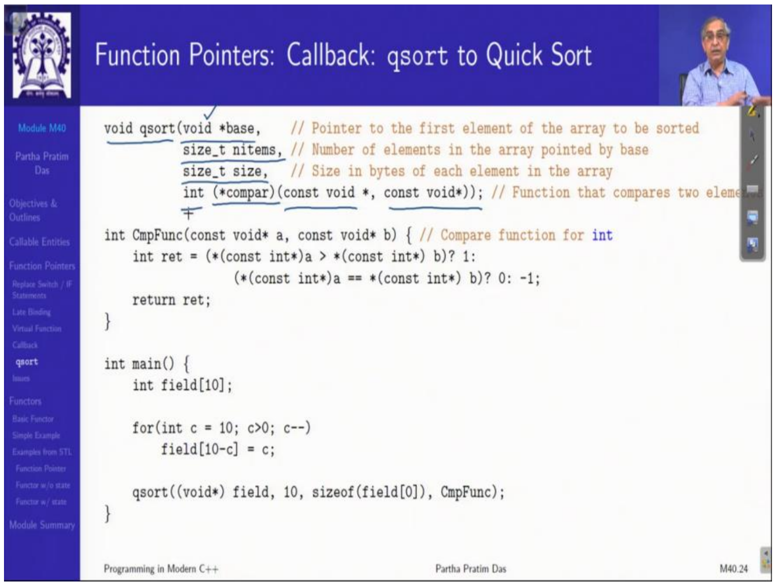
```
// Application
extern void (*func)();
void f()
{

}

void main()
{
    func = &f;

    g();
}
```

```
// Library
void (*func)();

void g()
{

    (*func)();

}
```

So, this is how it goes, you start doing main, main calls g control goes into the library, then it comes it does something comes to the function pointer. So, it tries to invoke the function which we have set to f here, which you have set to f here. So, it actually calls f and as you do that, then as you do that your function f is executing, as f executes, it comes to an end, the control returns back to the library. So, this is this is the callback happening.

So, what is happened is the library has called back into the application code and has got a functionality done embedded in it. And once this is done, then g comes back to main and the task has been completed. So, again, using the function pointer, you can have this kind of callback mechanisms. So, you can replace the if switch, you can do late binding, you can do callbacks, all of these you can do with function pointers, which is the most which gives you a lot of flexibility in terms of the way you do the entire thing.

(Refer Slide Time: 19:48)



Function Pointers: Callback: qsort to Quick Sort

```
void qsort(void *base,    // Pointer to the first element of the array to be sorted
           size_t nitems, // Number of elements in the array pointed by base
           size_t size,   // Size in bytes of each element in the array
           int (*compar)(const void *, const void*)); // Function that compares two eleme

int CmpFunc(const void* a, const void* b) { // Compare function for int
    int ret = (*(const int*)a > *(const int*) b)? 1:
              (*(const int*)a == *(const int*) b)? 0: -1;
    return ret;
}

int main() {
    int field[10];

    for(int c = 10; c>0; c--)
        field[10-c] = c;

    qsort((void*) field, 10, sizeof(field[0]), CmpFunc);
}
```

Module M40 | Partha Pratim Das | Objectives & Outlines | Callable Entities | Function Pointers | Replace Switch / IF Statements | Late Binding | Virtual Function | Callback | qsort | Issues | Functors | Basic Functor | Simple Example | Examples from STL | Function Pointer | Functor w/o state | Functor w/ state | Module Summary

Programming in Modern C++ | Partha Pratim Das | M40.24

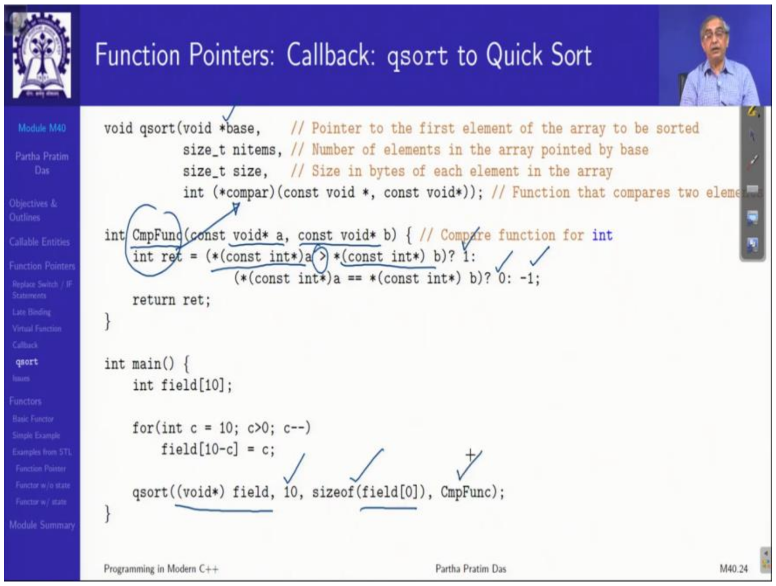
So, C uses this for actually giving you a generic sort function, which is qsort which is available in the standard library of C where you give a pointer to base which is an array containing a set of elements you do not know well, at the time of writing quick sort you do not know whether you would want to sort an array of integer or of double or of a student or what.

So, what you specify is, you specify how many elements are there, which you need to know, you specify the size of every element. Look at you are you are referring to it as void*. So, it what you can get is only you know, you have a pointed to something you do not know. And then you are saying this is the size.

So, you can take the size one after the other. So, many bytes together form one element. But to able to do quick sort you need to be able to compare two elements of the type you have given. So, you need to provide a compare function. And you provide that as a function pointer because the library at the time of writing qsort does not know the type that you will compare.

So, the library cannot write that. But all that it can say that it takes two pointers to two elements does not know its type. So, the only way it can refer to it is that constant void* that is pointed to constant objects of no known type. And the result should be an int, which gives you whether it is less so it is more or is equal.

(Refer Slide Time: 21:12)



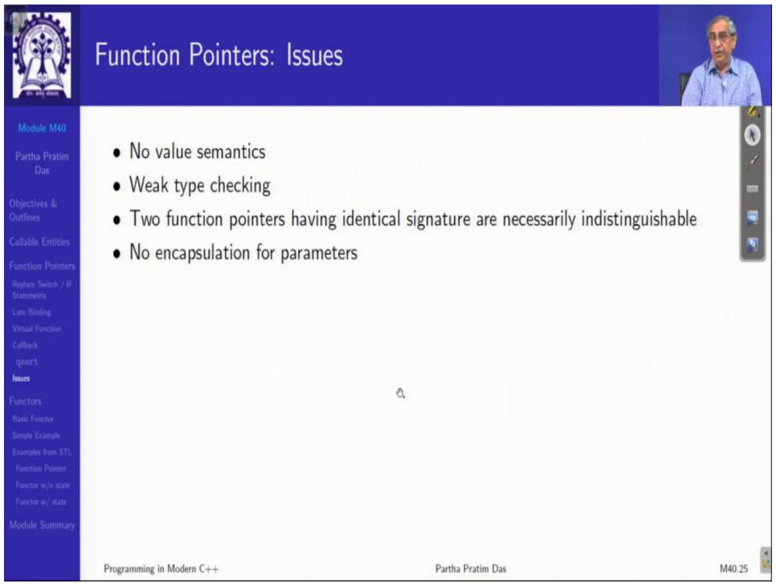
Function Pointers: Callback: qsort to Quick Sort

```
void qsort(void *base,    // Pointer to the first element of the array to be sorted
           size_t nitems, // Number of elements in the array pointed by base
           size_t size,   // Size in bytes of each element in the array
           int (*compar)(const void *, const void*)); // Function that compares two eleme

int CmpFunc(const void* a, const void* b) { // Compare function for int
    int ret = (*(const int*)a > *(const int*) b)? 1:
              (*(const int*)a == *(const int*) b)? 0: -1;
    return ret;
}

int main() {
    int field[10];

    for(int c = 10; c>0; c--)
        field[10-c] = c;

    qsort((void*) field, 10, sizeof(field[0]), CmpFunc);
}
```

Now, when the application programmer actually wants to use it, the application programmer has to provide that function it has to say that this is my function and the application programmer knows the type of element it has to compare. So, it casts into those types from void* does the comparison here it is cast to int. So, in comparison will happen if it is greater sets 1 as a return value otherwise if it is equal 0 if it is less minus 1.

So, this function now becomes eligible to work as a compare function pointer. So, this is something that the user is providing. So, you are going to use this as a qsort is going to use this as a callback to ask them that okay you have given me elements, but how to compare them. So, that is precisely what the qsort does it takes your array as void* to match this takes the number of elements take the size of every element, which you have taken size of filled 0, you could have done size of int also and then it takes this function pointer which it calls back to. So, this is the basic mechanism by which you can have user provided functionality to the library, this is what C has done always.

(Refer Slide Time: 21:25)



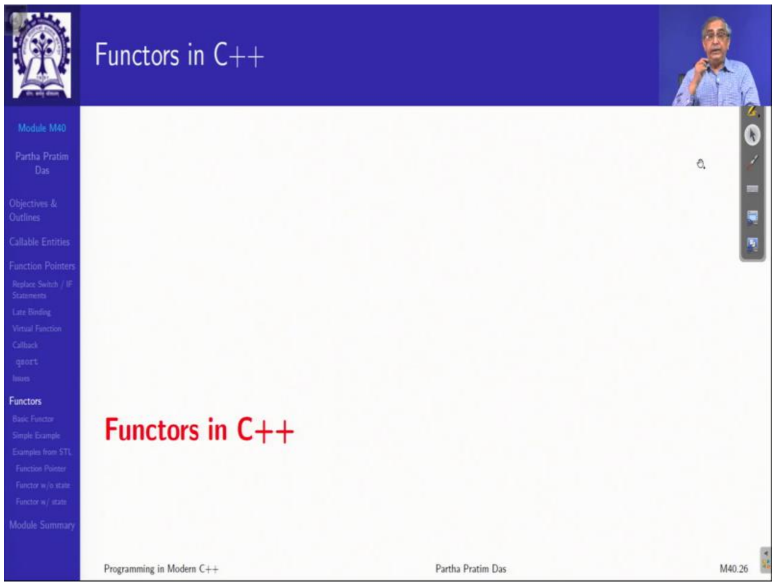
Function Pointers: Issues

- No value semantics
- Weak type checking
- Two function pointers having identical signature are necessarily indistinguishable
- No encapsulation for parameters

Programming in Modern C++ Partha Pratim Das M40.25

Now, there are a lot of issues with that because there is no value semantics there is no proper way to compare all functions which take a void and returns a void are considered to be of the same type which is which may not be what you want. And the type checking is almost not there, to this is the point I mentioned two function pointers having identical signature is necessarily indistinguishable, and you cannot capture the parameters.

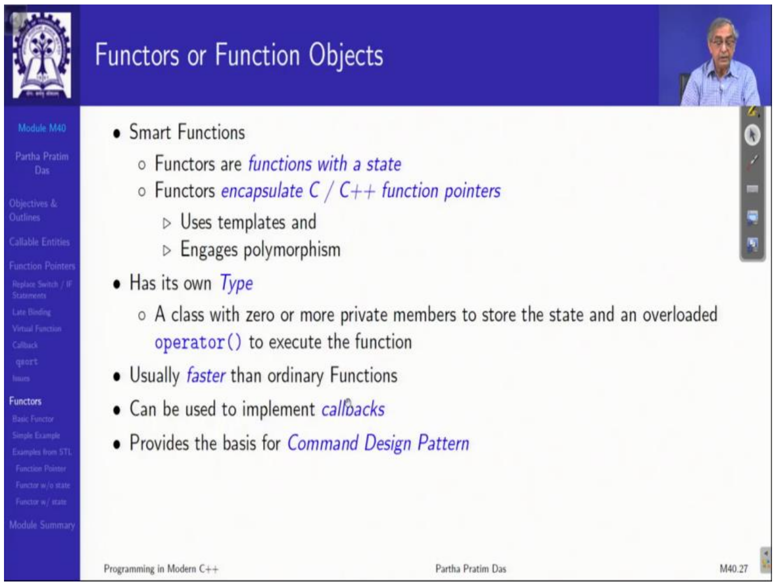
(Refer Slide Time: 21:55)



Functors in C++

Functors in C++

Programming in Modern C++ Partha Pratim Das M40.26



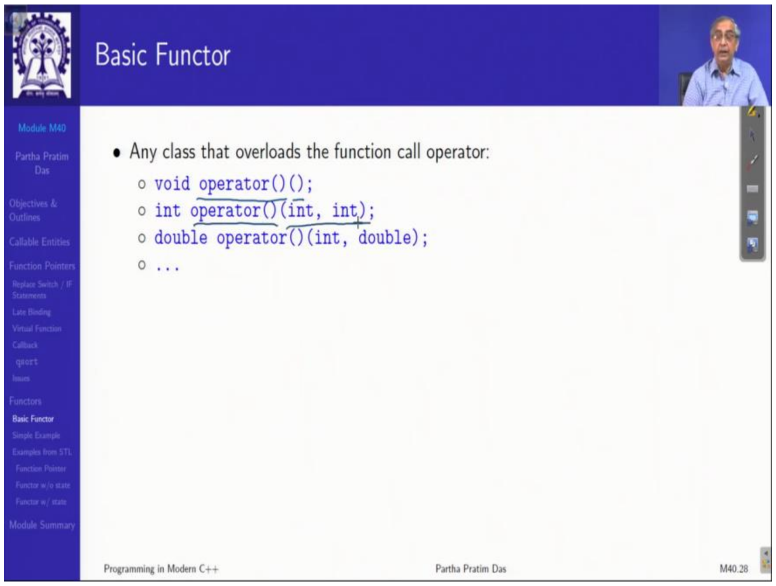
Functors or Function Objects

- Smart Functions
  - Functors are *functions with a state*
  - Functors *encapsulate C / C++ function pointers*
    - Uses templates and
    - Engages polymorphism
- Has its own *Type*
  - A class with zero or more private members to store the state and an overloaded `operator()` to execute the function
- Usually *faster* than ordinary Functions
- Can be used to implement *callbacks*
- Provides the basis for *Command Design Pattern*

Programming in Modern C++ Partha Pratim Das M40.27

Now, that is the these are the issues that functors solve. So, what is a functor, it is a smart function, it is a it is an object, which may or may not have states and it encapsulate a C or C++ function pointer. It uses template. It can engage in polymorphism. It has its own type and the most important it overloads the function invocation operator. Typically, function functors are faster than ordinary functions, they can be used to implement, callbacks and we will we will see later on that they are part of a design pattern called Common Design Pattern.

(Refer Slide Time: 23:05)

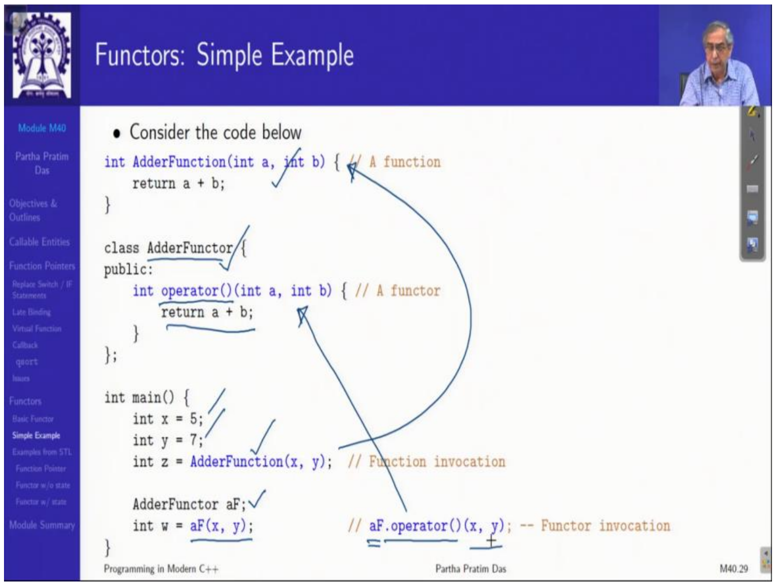


Basic Functor

- Any class that overloads the function call operator:
  - `void operator()();`
  - `int operator()(int, int);`
  - `double operator()(int, double);`
  - ...

So, these are these are some of the basic functors. For example, here just note the notation, this is a name of the operator and this is a function parameter. So, the operator function call takes nothing gives you nothing whereas here, the operator function call takes two integers gives you an integer. So, like, like the way double does. So, these are the basic functors that you have. But we will be interested in something which is more interesting.

(Refer Slide Time: 24:40)



Functors: Simple Example

- Consider the code below

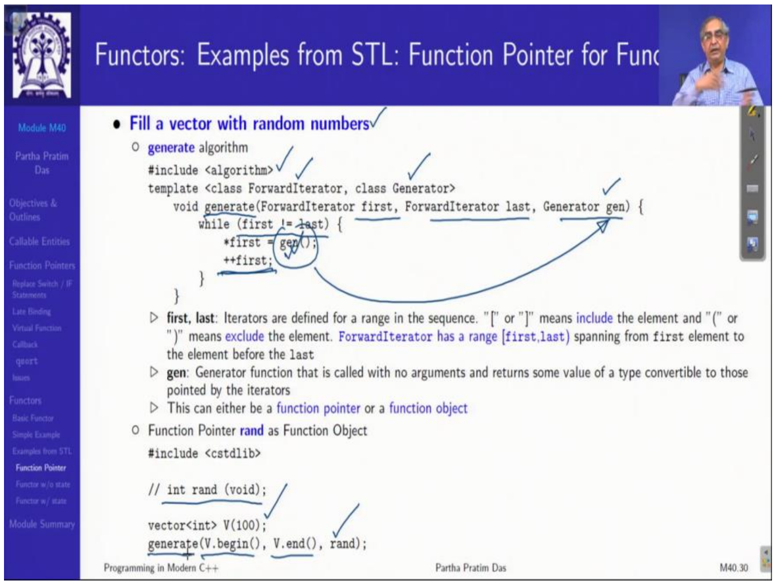
```
int AdderFunction(int a, int b) { // A function
    return a + b;
}

class AdderFunctor {
public:
    int operator()(int a, int b) { // A functor
        return a + b;
    }
};

int main() {
    int x = 5;
    int y = 7;
    int z = AdderFunction(x, y);  // Function invocation

    AdderFunctor aF;
    int w = aF(x, y);            // aF.operator()(x, y); -- Functor invocation
}
```

So, this is, this is the one slide memory that you should keep in mind. So, there is an AdderFunction, which is written in classical way, a global function. This is an AdderFunctor where what I have done is I have overloaded the function call operator to do this addition operator addition operator.

Now, let us see what happens. We have x, y, we call the adder function. This will call this function at the two numbers give me the result. We all know that. I have a class adder functor. So, I have instantiated that as a AdderFunctor object aF. And then I call aF(x, y) which is a notation which will shock you because aF is an object, but I am using function call syntax with that, how what does that mean, that means simply that aF being an object will take its operators as a shortcut of the detent notation.

So, aF, this operator means that aF dot (,) operator function points function call, so which means basically this function and the function will get invoked. The advantage now is that you can actually overload this operator for different types. You can overload this operator for different number of parameters. And added to that since this aF is an object, you can actually have a state of that, you can remember what that function was doing earlier and what is going to do next. So, this is a big, big, big advantage.

(Refer Slide Time: 26:47)



Functors: Examples from STL: Function Pointer for Func

- **Fill a vector with random numbers**
  - generate algorithm

```
#include <algorithm>
template <class ForwardIterator, class Generator>
    void generate(ForwardIterator first, ForwardIterator last, Generator gen) {
        while (first != last) {
            *first = gen();
            ++first;
        }
    }
```

  - ▷ **first, last**: Iterators are defined for a range in the sequence. "[" or "]" means include the element and "(" or ")" means exclude the element. ForwardIterator has a range [first,last) spanning from first element to the element before the last
  - ▷ **gen**: Generator function that is called with no arguments and returns some value of a type convertible to those pointed by the iterators
  - ▷ This can either be a function pointer or a function object
- Function Pointer **rand** as Function Object

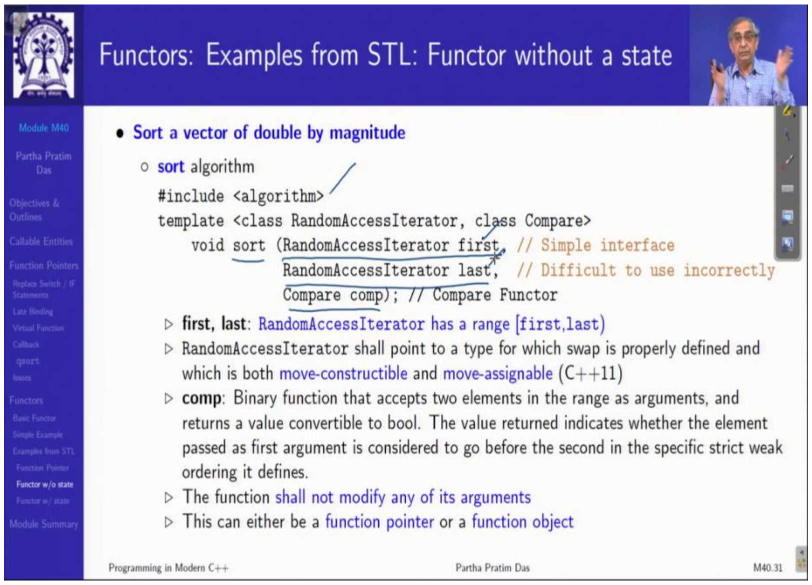
```
#include <cstdlib>

// int rand (void);

vector<int> V(100);
generate(V.begin(), V.end(), rand);
```

So, let me show you some examples from STL. STL has what is known as a, so this is the task, I want to fill up a vector with random numbers. I am using an STL component known under algorithm which gives me different algorithms. This algorithm is a generate algorithm. What does it take templatized takes a ForwardIterator that it will start from the left, go to the other end. And it takes a class Generator.

Now, generate takes the operators beginning and end. And this generator, which actually is a functor, which will do an operation, which will do a callback. And then what it does is it goes from first to last every time it is actually incrementing it. It is an iterator. So, it is going off to one after the other. And at every point, it invokes this function pointer, functor actually, to generate a number and gets that.

So, using this, and say, using a very simple generator like a, like the rand function in the library, you can write for a vector V, this is your iterator, begin this is your iterator end call rand as your functor, which will be called every time you go over this generation, and do generate. So, what it will do, it will start from the beginning of the vector keep on calling rand at every element and putting that value, the vector will get filled with the random values. So, that is how a functor can be used to write very simple and intuitive and compact code for us.

(Refer Slide Time: 27:58)



Functors: Examples from STL: Functor without a state

Module M40

Partha Pratim Das

Objectives & Outlines
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
qsort
Issues
Functors
Basic Functor
Simple Example
Examples from STL
Function Pointer
Functor w/o state
Functor w/ state
Module Summary

- Sort a vector of double by magnitude
  - sort algorithm

```
#include <algorithm>
template <class RandomAccessIterator, class Compare>
    void sort (RandomAccessIterator first, // Simple interface
               RandomAccessIterator last,  // Difficult to use incorrectly
               Compare comp); // Compare Functor
```

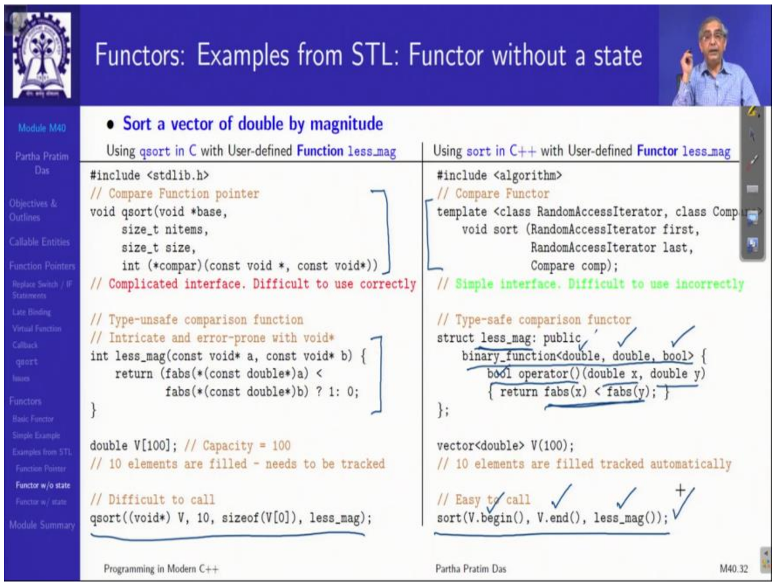
- ▷ **first, last**: RandomAccessIterator has a range [first,last)
- ▷ RandomAccessIterator shall point to a type for which swap is properly defined and which is both move-constructible and move-assignable (C++11)
- ▷ **comp**: Binary function that accepts two elements in the range as arguments, and returns a value convertible to bool. The value returned indicates whether the element passed as first argument is considered to go before the second in the specific strict weak ordering it defines.
- ▷ The function shall not modify any of its arguments
- ▷ This can either be a function pointer or a function object

Programming in Modern C++ Partha Pratim Das M40.31

Let us look at the sort. Suppose I want to sort a vector of double by magnitude. I again, include algorithm which has a template algorithm for sort. What does it need to know, it needs to know the beginning and end of the iterator, I am using random iterators here, because I need to know what part, some sequence - it could be an array, it could be a list, whatever, I do not care.

But I will iterate and go from here to here and to be able to sort that and I need to know a compare comparison functor like we needed in qsort to be given in a complicated manner. So, what it will do in this first and last specified, it will go over this entire collection and actually use the comp functor to do the comparison and do the sorting. So, that is a nice beautiful solution and we can compare it with the quick sort solution to see what you get this is left side is the solution in C and C library use in the quick sort example.

(Refer Slide Time: 30:57)



Functors: Examples from STL: Functor without a state

Module M40

Partha Pratim Das

Objectives & Outlines
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
qsort
Issues
Functors
Basic Functor
Simple Example
Examples from STL
Function Pointer
Functor w/o state
Functor w/ state
Module Summary

- Sort a vector of double by magnitude

Using qsort in C with User-defined Function less_mag

```
#include <stdlib.h>
// Compare Function pointer
void qsort(void *base,
    size_t nitems,
    size_t size,
    int (*compar)(const void *, const void*))
// Complicated interface. Difficult to use correctly

// Type-unsafe comparison function
// Intricate and error-prone with void*
int less_mag(const void* a, const void* b) {
    return (fabs(*(const double*)a) <
            fabs(*(const double*)b) ? 1: 0;
}

double V[100]; // Capacity = 100
// 10 elements are filled - needs to be tracked

// Difficult to call
qsort((void*) V, 10, sizeof(V[0]), less_mag);
```

Using sort in C++ with User-defined Functor less_mag

```
#include <algorithm>
// Compare Functor
template <class RandomAccessIterator, class Compare>
    void sort (RandomAccessIterator first,
               RandomAccessIterator last,
               Compare comp);
// Simple interface. Difficult to use incorrectly

// Type-safe comparison functor
struct less_mag: public
    binary_function<double, double, bool> {
        bool operator()(double x, double y)
        { return fabs(x) < fabs(y); }
};

vector<double> V(100);
// 10 elements are filled tracked automatically

// Easy to call
sort(V.begin(), V.end(), less_mag());
```

Programming in Modern C++ Partha Pratim Das M40.32

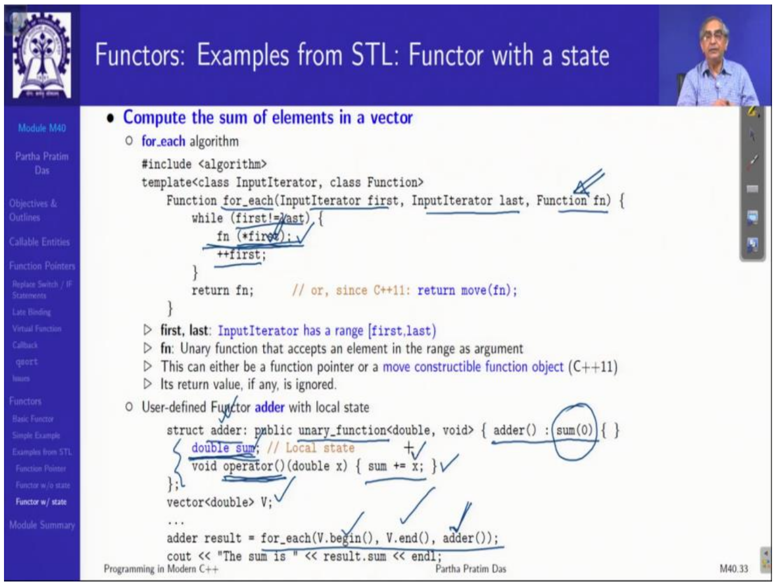
This is your quicksort signature. Quite complicated users do not know what it means, this is your sort signature, which is very simple and straightforward. You tell the range and the compare functor. That is all you need to know to compare to sort a range of elements. This is your comparison functor less magnitude, this is how you have to write it in C take it in void* cast it to double* then dereference double* and all complicated. Here you write it very simply as a Boolean operator because it has to it is a functor. So, what it will do it will do a function invocation taking two double and returning a bool and all that you need to do is to actually just do the comparison of the double.

Now, the question is how I sort know that this kind of a function, this kind of an interface exists that is why the library provides a template binary function which has three parameters, three type parameters, binary functions, so it will get two parameters for that function. So, types of rows and this is a return type.

So, it becomes very easy for the sort to know that you are providing what kind of a binary function for doing the comparison. So, actual code, you just look at the call here, quick sort this we have already seen. And here, it actually says what it does start at the beginning go up to the end, use less_mag, I have just used struct here for place of class because I need this to be public. So, having written struct, I may not actually write this as I am sorry, this is this is different this is for the inheritance actually.

So, it publicly inherits this less_mag inherits from the binary function template, which says that it is a binary function taking two double and a bool and implements it. And the interface that you actually have to call it is so, so simple. So, this is a this is a very, very powerful use of functors in the context of the library, know that this does not have a state because it is a functor did not need to use any data member.

(Refer Slide Time: 31:14)



Functors: Examples from STL: Functor with a state

- **Compute the sum of elements in a vector**
  - for_each algorithm

```
#include <algorithm>
template<class InputIterator, class Function>
    Function for_each(InputIterator first, InputIterator last, Function fn) {
        while (first!=last) {
            fn (*first);
            ++first;
        }
        return fn;      // or, since C++11: return move(fn);
    }
```

  - ▷ first, last: InputIterator has a range [first,last)
  - ▷ fn: Unary function that accepts an element in the range as argument
  - ▷ This can either be a function pointer or a move constructible function object (C++11)
  - ▷ Its return value, if any, is ignored.
  - User-defined Functor adder with local state

```
    struct adder: public unary_function<double, void> { adder() : sum(0) { }
        double sum; // Local state
        void operator()(double x) { sum += x; }
    };
    vector<double> V;
    ...
    adder result = for_each(V.begin(), V.end(), adder());
    cout << "The sum is " << result.sum << endl;
```

We can also have states in this. So, for example, I want to compute the sum of n elements in a vector, vectors n elements, I want to compare that. So, what I do use here is an algorithm called for each, that is do for every element as you go in the iteration from first to last and apply this functor fn or it can be used for any purpose. So, it goes from first to last just applies this fn on the current element goes to the next element. I am going to use that this for_each to add the elements of a vector.

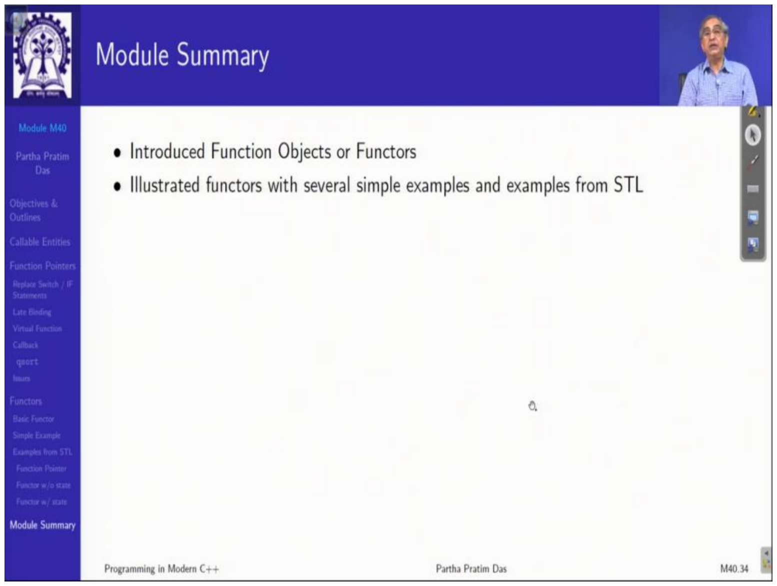
So, I am defining a functor adder deriving from the template of unary_function that is I just need one parameter and it will return a void that it will not return anything. So, what does that do that simply constructs an adder and what does the adder have, the adder has a local state a data member sum that is it will have two things, one is it has the function call operator to make a call every time and it has a local state sum where it can keep the sum and I encapsulate both of them in terms of this adder functor adder.

So, I define a vector I do for_each on this this part is routine go from here to here and the as functor I pass adder, So, what will happen? The first when this is called at the beginning adder is got constructed of called cumulate here, so, sum is initialized to 0 this is initialized to 0. And then it will call this fn which is this function which is adder the object function, which is the function call operator here.

So, it calls that, so whatever is a sum x gets added to it what is x what has been passed the current value that is added to its keys as a part of the functor I do not need to have a separate sum variable, I again go to the next in the iteration. Again, this is get invoked. So, the sum which had the first element now, to that the second element gets added, it goes on in this way.

And all elements get added over this time. So, this is a very beautiful way to see how functors can be used with state. We will have a separate tutorial on this as well because this is a little bit involved. And we will have lot of illustrations of this when we get to the Modern C++ part.

(Refer Slide Time: 34:48)



Module Summary

- Introduced Function Objects or Functors
- Illustrated functors with several simple examples and examples from STL

Module M40

Partha Pratim Das

Objectives & Outlines

Callable Entities

Function Pointers

Replace Switch / IF Statements

Late Binding

Virtual Function

Callback

qsort

Issues

Functors

Basic Functor

Simple Example

Examples from STL

Function Pointer

Functor w/o state

Functor w/ state

Module Summary

Programming in Modern C++ Partha Pratim Das M40.34

But I just wanted to introduce this basic concept of function objects where objects can have function call operator and can be used for invoking functions with or without state. Thank you very much for your attention. We will meet in the next module next week.