Programming in Modern C++
Functors

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

Feb 18, 2021

Table of Contents

Functors in C++

Callable Entities in C / C++

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

Some authors do distinguish between Callable Entities, Function Objects and Functors.

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()

Function Pointers

- 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

Function Pointers in C

- Define a Function Pointer

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

- Calling Convention

```
int DoIt (int a, char b, char c);
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');
```

Function Pointers in C

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
```

Function Reference In C++

- 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');
```

Function Pointer: 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

Function Pointer: Programming Techniques

- Replacing switch/if-statements
- Realizing user-defined late-binding, or
  - Functions in Dynamically Loaded Libraries
  - Virtual Functions
- Implementing callbacks.

Function Pointers – Replace Switch/ IF Statements

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 ;
}
```

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;
}
```

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

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

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

Example: Callback, Function Pointers

- It is a Common C Feature

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

Function Pointers: Callback Illustration (Step-1)


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


}

void main()
{
    func = &f;

    g();
}
```

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

void g()
{

    (*func)();

}
```

Function Pointers: Callback Illustration (Step-2)


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


}

void main()
{
    func = &f;

    g();
}
```

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

void g()
{

    (*func)();

}
```

Function Pointers: Callback Illustration (Step-3)


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


}

void main()
{
    func = &f;

    g();
}
```

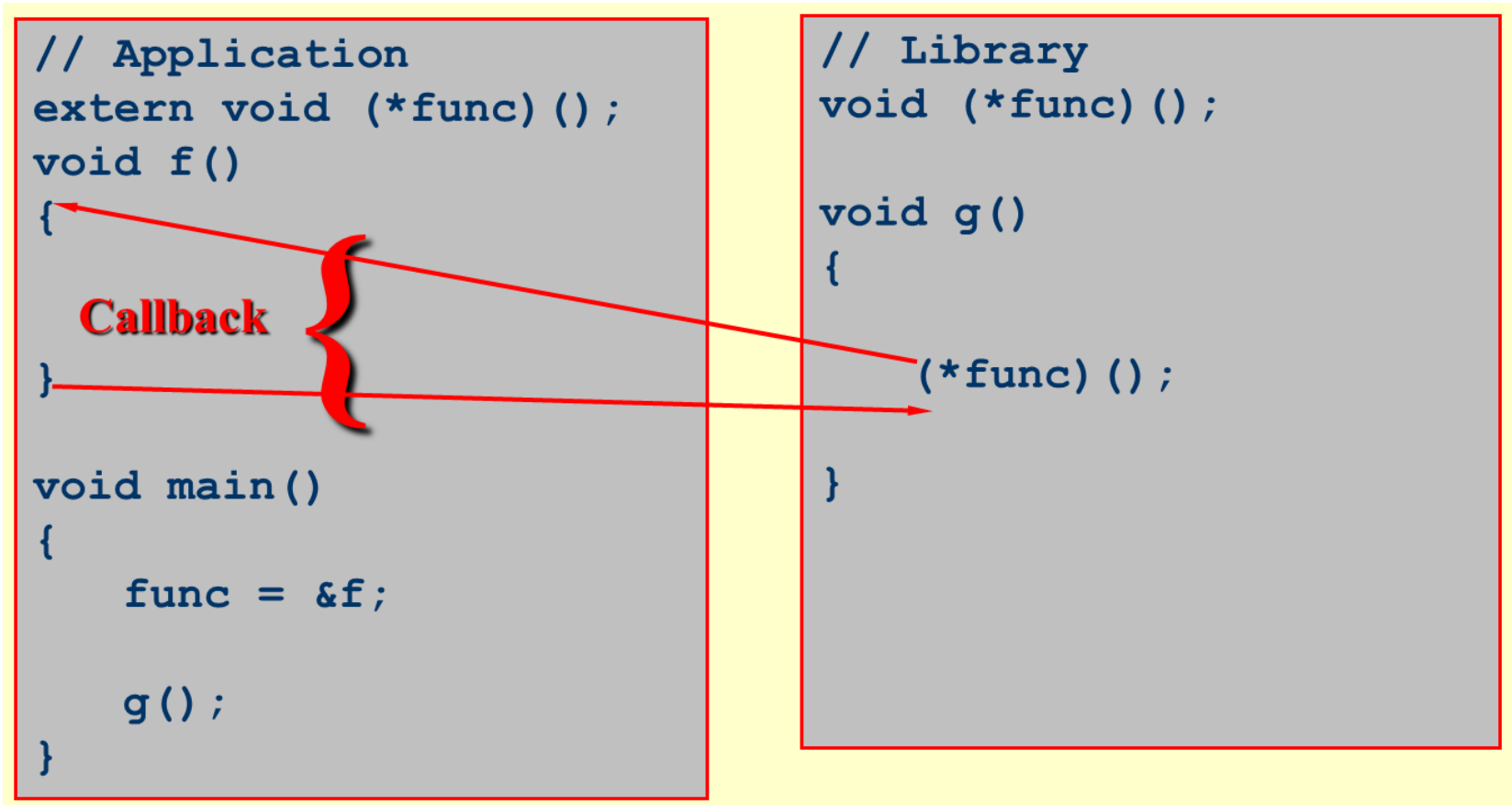
```
// Library
void (*func)();

void g()
{

    (*func)();

}
```

Function Pointers: Callback Illustration (Step-4)



Function Pointers: Callback Illustration (Step-Final)


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


}

void main()
{
    func = &f;

    g();
}
```

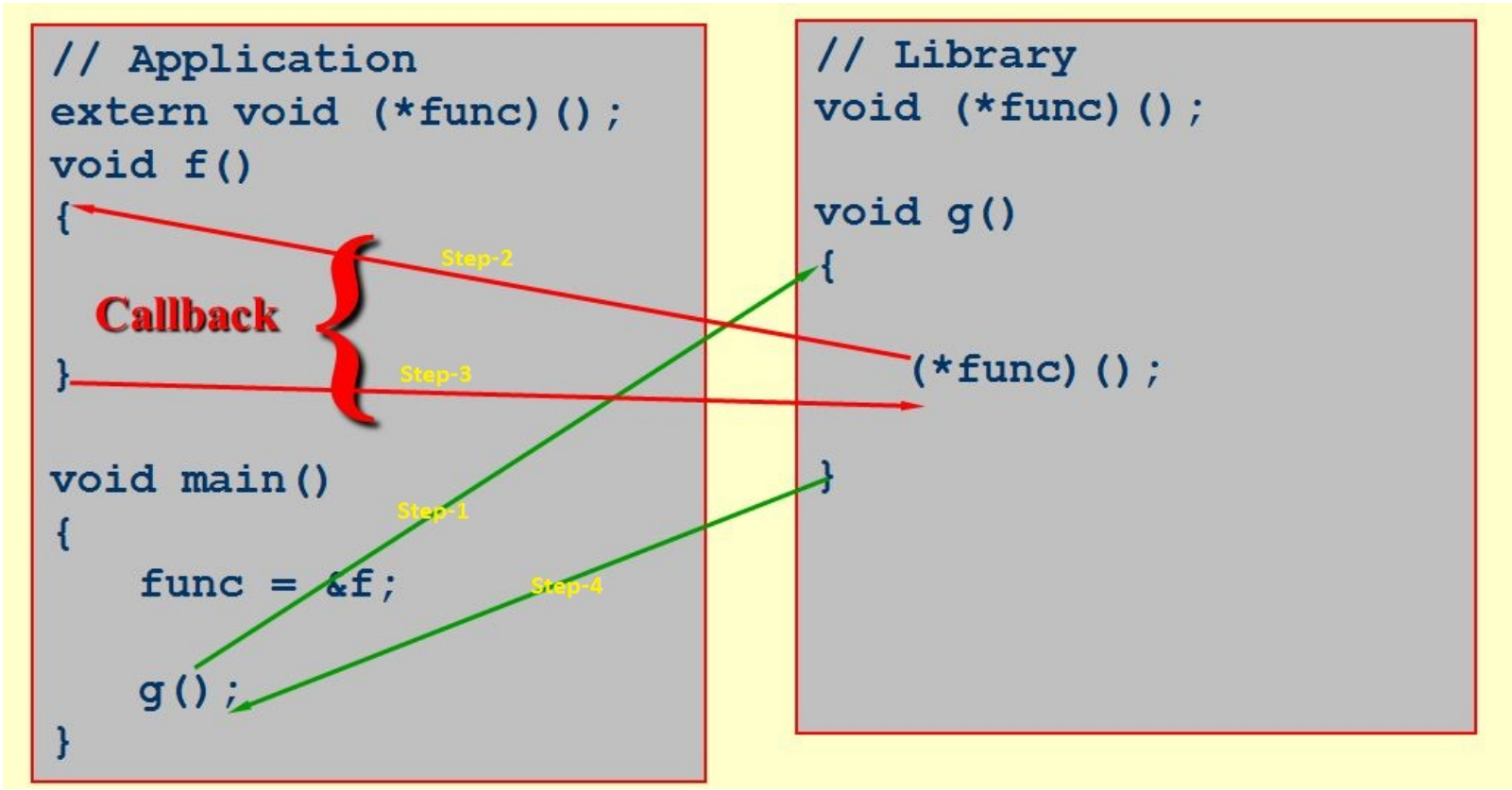
```
// Library
void (*func)();

void g()
{

    (*func)();

}
```

Function Pointers: Callback Illustration (whole Process)



Function Pointers–Callback: Quick Sort Implementation using callback in 'qsort'


```
void qsort(void *base,
           size_t nitems,
           size_t size,
           int (*compar)(const void *, const void*));

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

void main() {
    int field[10];

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

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

Function Pointers – Issues

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

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*

Basic Functor

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

Functors: Elementary Example

- Look at the code below

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

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

void main() {
    int x = 5;
    int y = 7;
    int z = AdderFunction(x, y);

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

Functors: Examples from STL: Function Pointer for Functor

- **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**: Forward iterators to the initial and final positions in a sequence. The range affected is [first,last), which contains all the elements between first and last, including the element pointed by first but not the element pointed by 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);
```

Functors: Examples from STL: Functor without a state

- **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**: Random-access iterators to the initial and final positions of the sequence to be sorted. The range used is [first,last), which contains all the elements between first and last, including the element pointed by first but not the element pointed by last.

      RandomAccessIterator shall point to a type for which swap is properly defined and which is both move-constructible and move-assignable.
    - **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.

Functors: Examples from STL: Functor without a state

- **Sort a vector of double by magnitude**
  - Using `qsort` in C with User-defined **Function** `less_mag`

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

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

double V[100]; // Capacity = 100
// 10 elements are filled. It needs to be tracked
qsort((void*) V, 10, sizeof(V[0]), less_mag); // Difficult call
```

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

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

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

vector<double> V(100);
// 10 elements are filled. Tracked automatically by vector
sort(V.begin(), V.end(), less_mag()); // Easy call
```

Functors: Examples from STL: Functor with a state

- **Find 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**: Input iterators to the initial and final positions in a sequence. The range used is [first,last), which contains all the elements between first and last, including the element pointed by first but not the element pointed by 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.

      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;
```