Advanced Software Development

Funktionsobjekte

Übersicht

- Funktionsadapter
- Lambda Expressions
- std::function

Funktionsadapter

- Binders: bind1st, bind2nd

```
nrNegative = count_if(v.begin(), v.end(),
                      bind2nd(less<int>(), 0));
```

- Negators: not1, not2

```
struct IsPositive: public unary_function<int, bool> {
  bool operator()(int const x) const { return x >= 0; }
};
nrNegative = count_if(v.begin(), v.end(), not1(IsPositive()));
```

- Function Pointer Adaptor: ptr_fun
- Member Function Pointer Adaptors: mem_fun, mem_fun_ref

Function Pointer Adaptor (ptr_fun)

- Ermöglicht die Verwendung von Function Pointers mit anderen Adaptern

```
bool IsPositive(int const x) {
  return x >= 0;
}

// Folgendes funktioniert nicht
nrNegative = count_if(v.begin(), v.end(), not1(IsPositive));
// Mit ptr_fun funktioniert's
nrNegative = count_if(v.begin(), v.end(), not1(ptr_fun(IsPositive)));
```

- Für Funktionen mit 1 oder 2 Parametern

Member Function Pointer Adaptors (1)

- Adapter für Methoden (Member Functions)

```
class MyClass {
public:
  void Print() const;
  void Add(int x);
};

vector<MyClass*> objects;
for_each(objects.begin(), objects.end(), mem_fun(&MyClass::Print));
for_each(objects.begin(), objects.end(),
         bind2nd(mem_fun(&MyClass::Add), 1));
```

- Für Methoden mit 0 oder 1 Parameter
- Auch für virtuelle Methoden

Member Function Pointer Adaptors (2)

- mem_fun: für Aufruf über Pointer
- mem_fun_ref: für Aufruf über Referenz

```
vector<MyClass> objects;
for_each(objects.begin(), objects.end(),
         mem_fun_ref(&MyClass::Print));
```

Member Function Pointers

```
// Deklaration Member Function Pointer Type und Variable
typedef void (MyClass::*MemFunType)(int);
MemFunType myMemFun;

// Zuweisung Methoden-Adresse
myMemFun = &MyClass::Add;

// Aufruf mit Objekt
MyClass obj;
(obj.*myMemFun)(1);

// Aufruf mit Pointer
MyClass *pObj = new MyClass;
(pObj->*myMemFun)(2);
```

std::mem_fn

- Neu in C++ 11
- Verallgemeinerung von mem_fun und mem_fun_ref
- Funktioniert mit Pointern, Referenzen und Smart Pointern

```
list<shared_ptr<GraphicObject>> objects;
for_each(objects.begin(), objects.end(),
        mem_fn(&GraphicObject::Draw));
```

- Für Methoden mit beliebig vielen Parametern

std::bind

- Neu in C++ 11
- Verallgemeinerung von bind1st und bind2nd
- Für beliebig viele Parameter (implementierungsabhängig)
- Für Funktoren, Function Pointers und Member Function Pointers

std::bind – Verwendung

- Platzhalter (_1 bis _9) für ungebundene Parameter (Namespace std::placeholders)
- Beispiele:

```
bind(less<int>(), _1, 0)  // 2. Parameter gebunden (wie bind2nd)

void MyFunction(int x, int y, int z);
bind(MyFunction, 1, 2, _1)     // x und y gebunden
bind(MyFunction, _1, _2, 100)  // z gebunden

class MyClass {
public:
  void Add(int x);
};
bind(&MyClass::Add, _1, 1)     // this ungebunden, x gebunden
```

std::bind – Beispiel

```
class GraphicObject {
public:
  …
  void Rotate(double angle);
  void Move(double dx, double dy);
  …
};

list<GraphicObject*> objects;
…
// Alle Objekte um 90° rotieren
for_each(objects.begin(), objects.end(), bind(&GraphicObject::Rotate, _1, 90));
// Alternativvariante mit bind2nd und mem_fun:
// for_each(objects.begin(), objects.end(),
//          bind2nd(mem_fun(&GraphicObject::Rotate), 90));

// Alle Objekte um (10,10) verschieben
for_each(objects.begin(), objects.end(), bind(&GraphicObject::Move, _1, 10, 10));
```

std::bind – Call by reference

- bind kopiert standardmäßig alle Parameter
- Übergabe von Referenzen mittels std::ref bzw. std::cref

```
template <typename T>
void Add(T &x, T const &y) {
  x += y;
}

int sum = 0;
for_each(numbers.begin(), numbers.end(),
         bind(Add<int>, ref(sum), _1));

string suffix = …;
for_each(strings.begin(), strings.end(),
         bind(Add<string>, _1, cref(suffix)));
```

Lambda Expressions

- Lambda Expression (Lambda-Ausdruck) = anonyme Funktion
- Grundelement funktionaler Programmiersprachen
- Zunehmend auch in imperativen Programmiersprachen zu finden, z.B. C# 3.0, Java 8 und C++ 11

Lambda Expressions in C++ 11

- Beispiel: Absteigend sortieren

```
vector<int> v;
…
sort(v.begin(), v.end(), [](int x, int y) { return x > y; });
```

- Bestandteile einer Lambda Expression
  - […]: Lambda Introducer: Erlaubt „Capture“ von Variablen des umgebenden Blocks
  - (…): Parameter
  - {…}: Block mit Anweisungen
  - Rückgabetyp (optional wenn Block nur eine return-Anweisung enthält): `[…] (…) -> int { … }`

Beispiele: „Capture“ von Variablen

```
vector<int> v;
…

int f = …;
// Multiplikation aller Elemente mit f
for_each(v.begin(), v.end(), [f](int &x) { x *= f; });


// Summenberechnung
int sum = 0;
for_each(v.begin(), v.end(), [&sum](int x) { sum += x; });
```

Speichern von Lambdas

- Lambda-Ausdrücke haben einen unbenannten, compiler-internen Typ
- Zuweisung an Variable mittels *auto*

```
auto isEven = [](int x) { return x % 2 == 0; };
…
int x = …;
if (isEven(x)) { … }
…
vector<int> v;
…
int nrEven = count_if(v.begin(), v.end(), isEven);
```

Generische Lambdas

- Angabe von auto als Parametertyp

```
vector<pair<int, int>> v;
…
auto iter = find_if(v.begin(), v.end(),
  [](auto elem) { return elem.first < 0; });
```

- Speichern von generischen Lambdas

```
auto square = [](auto x) { return x * x; };
int x = …;
int xs = square(x);
double d = …;
double ds = square(d);
```

Capture: Varianten

- Zugriff auf Members und Methoden

```
[this](int x) -> bool {
  this->SomeMethod();
  return x > mSomeMember;
};
```

- Capture aller verfügbaren Variablen
  - [=]: alle Variablen als Wert (inkl. *this*)
  - [&]: alle Variablen als Referenz (*this* als Wert)
  - [=, &x]: alle Variablen als Wert, *x* als Referenz
  - [&, x]: alle Variablen als Referenz, *x* als Wert

Capture: Initialisierung

- Initialisierung mit beliebigen Ausdrücken

```
int x = 5;
auto lambda = [y = x + 1] {
  …
};
```

- Ermöglicht Capture by Move

```
auto ptr = make_unique<string>("abc");
auto lambda = [ptr = move(ptr)] {
  …
};
```

std::function (1)

- Ermöglicht das Speichern von Funktionsobjekten, Function und Member Function Pointers, Lambda-Funktionen

```
int SumSquares(int x, int y) {
  return x * x + y * y;
}

function<int(int, int)> f;
f = plus<int>();
f = SumSquares;
f = [](int x, int y) { return x * x + y * y; };
int x = f(3, 4);  // x = 25
```

std::function (2)

- Zuweisung aller kompatiblen Funktionen möglich

```
void MyFunction(double);
function<void(int)> f = MyFunction;
```

- Verwendung mit Member Function Pointer

```
class MyClass {
public:
  void MyMethod(int x);
  …
};

function<void(MyClass&, int)> f = &MyClass::MyMethod;
MyClass obj;
f(obj, 0);
```