Object-oriented Programming

for Automation & Robotics

Carsten Gutwenger

LS 11 Algorithm Engineering

Lecture 13 • Winter 2011/12 • Jan 24

technische universität dortmund

department of computer science

Today’s Agenda

- **Smart Pointers**
  - pointer-like objects with “automatic garbage collection”
- **Function Objects**
  - … with applications in C++ standard library **algorithms**!
  - overloading the function call operator
- **Final Exam FAQ**
  - Topics & organizational stuff

3rd place in voting

Smart Pointers

- **Smart pointers**
  - are data types that simulate a pointer.
  - provide additional features like automatic deletion of the object they point to.
- Main benefits
  - Avoid typical programming errors like dangling pointers and memory leaks.
  - Express e.g. who is responsible for the objects pointed to (*Who needs to delete the object when a function returns a pointer?*)
    → Explicit transfer of ownership
- Different variants
  - **Unique pointers:** Implement strict ownership (explicit transfer of ownership is possible).
  - **Shared pointers:** Use reference counting for deciding when to delete the object pointed to.

Smart Pointers: History in C++

- **“Old”** C++-standard (**C++ 98**, **C++ 03**)
  - class `std::auto_ptr`
  - deprecated in the latest standard
- **“New”** C++-standard: **C++ 11**
  - class `std::unique_ptr`
  - class `std::shared_ptr`
  - VS 2008:
    Only `std::tr1::shared_ptr` available (**C++ TR1**)
- In this lecture:
  - We use VS 2010 / **C++ 11**

Unique Pointers: unique_ptr

- **`unique_ptr<`*`type`*`>`**
  - a smart pointer that retains sole ownership of an object through a pointer.
  - no copy possible:
    no two instances of **`unique_ptr`** can manage the same object!
  - stores a pointer to an object (allocated with **`new`**), or a 0-pointer.
- **Transfer of ownership**
  - Use function **`std::move`**.
  - Member function **`swap`** exchanges the pointers stored in two unique pointers.
- **Automatic deletion** of the object pointed to
  - When the unique pointer is destroyed (e.g. goes out of scope).
  - Using member function **`reset`**.

unique_ptr: Example

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

int main() {
    unique_ptr<int> p1( new int(10) );
    unique_ptr<int> p2( new int(20) );
    unique_ptr<int> p3 = p1; // compiler error

    unique_ptr<int> q1 = move(p1); // p1 now empty!
    q1.swap(p2);
    p1.reset( new int(30) );

    // prints "30 20 10"
    cout << *p1 << " " << *p2 << " " << *q1 << endl;

    q1.reset();  // explicit "delete"; q1 now empty

    return 0;
}
```

contains definitions for smart pointers

transfer ownership

Output:

```
30 10 20
```

Shared Pointers: shared_ptr

- **`shared_ptr<`*`type`*`>`**
  - similar as **`unique_ptr`**, but allows several owners.
    → copying shared pointers is possible.
  - maintains a reference count, which counts how many shared pointers point to that object.
  - object is deleted when the last shared pointer pointing to that object is destroyed.

shared_ptr: Example

```
…
   shared_ptr<int> p1( new int(10) );
   shared_ptr<int> p2( new int(20) );
   shared_ptr<int> p3 = p1; // copy possible!

   shared_ptr<int> q1 = move(p1); // p1 now empty!
   q1.swap(p2);
   p1.reset( new int(30) );

   // prints "30 10 10 20"
   cout << *p1 << " " << *p2 << " " << *p3 << " " << *q1 << endl;

   cout << *p3 << ": use count = " << p3.use_count() << endl;
   p2.reset();  // decreases use count for "10"
   cout << *p3 << ": use count = " << p3.use_count() << endl;
…
```

Output:

```
30 10 10 20
10: use count = 2
10: use count = 1
```

Function Objects

- A function object (functor) is an object that can be invoked using the same syntax as for invoking a function.

```
IsGreaterThan compare;
cout << boolalpha <<
            "4 > 2 ? " << compare(4,2) << endl;
```

- How does this work?
  - We have overloaded the function call operator in the structure **`IsGreaterThan`**, such that it takes two **`int`**s as input and returns a **`bool`**.
  - **`compare(4,2)`** is short for **`compare.operator()(4,2)`**

Overloading the Function Call Operator

- Overloading (as usual)

```
struct IsGreaterThan {
    bool operator()(int x, int y) {
        return x > y;
    }
};
```

- Any number of parameters is possible (0, 1, 2, ...).
- We could also implement several function call operators.
  [Same rules as for overloading functions apply.]
- Advantage compared to a function
  - We have access to local data members of the function object (e.g. these can be initialized when constructing the function object).

Example: Sorting in descending order

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

struct IsGreaterThan {
   bool operator()(int x, int y) {
      return x > y;
   }
};

ostream &operator<<(ostream &os, const vector<int> &v) {
   for(vector<int>::const_iterator it = v.begin();
      it != v.end(); ++it)
      os << *it << endl;

   return os;
}
```

Example: Sorting in descending order

```
int main() {
    // create a vector of random integers
    vector<int> v;  srand(4711);
    for(int i = 0; i < 8; ++i)
       v.push_back(rand() % 100);

    cout << v;

    IsGreaterThan compare;
    sort(v.begin(), v.end(), compare);

    cout << "----------" << endl;
    cout << v;

    return 0;
}
```

Output:

```
22
2
26
96
71
69
26
53
-----
96
71
69
53
26
26
22
2
```

Predicates

- A predicate is a function object that returns a `bool` (`true` or `false`)
- Predicates are widely used in the C++ standard library.
- Examples:
  - `IsGreaterThan` is a binary predicate defining an order.
  - Algorithms:
    `sort`, `stable_sort`, `nth_element`, `binary_seach`, `merge`
    `min_element`, `max_element`
  - Unary predicate: IsOdd
  - Algorithms:
    `find_if`, `count_if`, `replace_if`, `remove_if`

Example: replace_if

```
struct IsOdd
{
   bool operator()(int x)
   {
      return x % 2 == 1;
   }
};
```

```
int main() {
   const vector<int>::size_type n = 16;
   vector<int> v(n);

   vector<int>::size_type i;
   for(i = 0; i < n; ++i)
      v[i] = i+1;

   IsOdd is_odd;
   replace_if(v.begin(), v.end(), is_odd, 0);

   for(i = 0; i < n; ++i)
      cout << v[i] << " ";
   cout << endl;

   return 0;
}
```

Output:

```
0 2 0 4 0 6 0 8 0 10 0 12 0 14 0 16
```

Final Exam: FAQ

- ***Who can attend the final exam?***
  - Everyone with three successful exam sheets.
  - No registration required.
- ***Where will the final exam take place?***
  - January 31
  - Group A: 10:30-12:30, Retina pool 108a & 108b
  - Group B: 13:30-15:30, Retina pool 108b
  - There will be a list assigning you to group A or B. Go to that group!
  - You will have 90 minutes for solving the exercises, plus extra time for filling out name, matriculation number etc.
- ***What do you need?***
  - Your student ID and passport
  - A pen

Final Exam: FAQ

- ***Which additional material can you use?***
  - Only the printed lecture slides
- ***What is not allowed?*** **(→ Cheating = Failing the exam)**
  - Hand-written notes on the print-outs
  - Computers / laptops / smartphones / mobile phones
  - Any source-code, like the solutions to the assignment and exam sheets
- ***What should you do for preparation?***
  - The topics are listed on Assignment Sheet No. 12
  - You should carefully study and understand the solutions to the exercises listed there
  - Try to solve some of these exercises with pen & paper

Final Exam: FAQ

- ***Which tasks will you be given?***
  - Write C++ source code
    (Solutions will typically be short, sometimes part of the code is given.)
  - Read and understand a given piece of C++ source-code
    (Answer questions about the output of a program or the values of variables at “checkpoints”.)
- ***What is required to pass the final exam?***
  - Similar as for the other exam sheets:
    Solve at least half of the four exercises successfully!
  - Exercises will be rated with **0** / **0.5** / **1** points
    → You need at least 2 points in total
  - And: Write readable! *I am very bad in deciphering bad handwriting, and if I cannot read something I assume it is wrong.*