EL2310 – Scientific Programming

Lecture 16: STL, C++1y

Ramviyas Parasuraman (ramviyas@kth.se)

Royal Institute of Technology – KTH

Overview

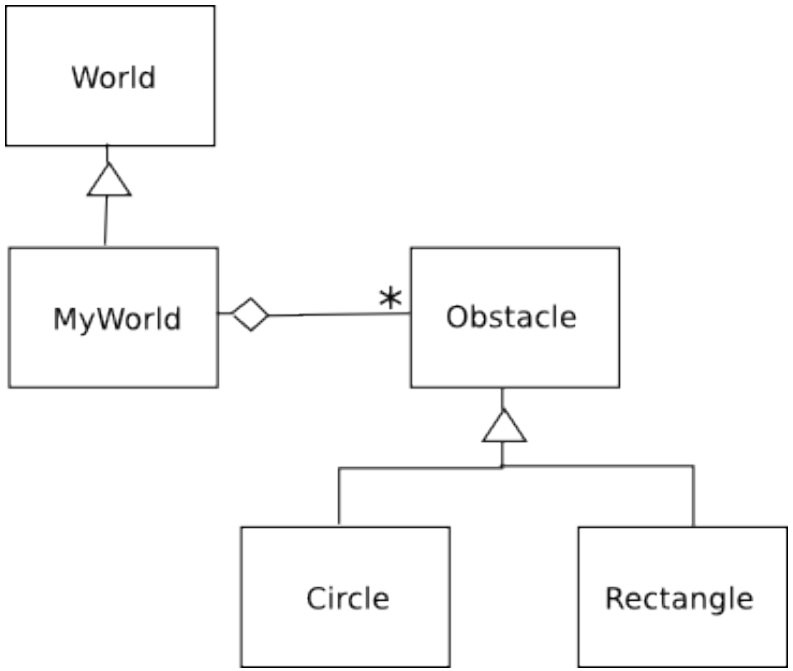
Lecture 16: STL and C++1y

- Reminders
- Wrap up
- The Standard Template Library (STL)
- C++11

The help sessions and deadline

- C++ help session: Fri 24.10.2015, 15:00-17:00, Room ”22:an”, Teknikringen 14
- C++ project deadline: Mon 26.10.2015, 20:00

Tips for C++ project



Lecture 16: STL and C++1y

Accessing methods in inherited class

- ```
  class Vehicle
  {
    void drive();
  }
  class Car:  public Vehicle
  {
    void openTrunk();
  }
  ```
- `Vehicle *v = new Car();`
- `v->drive();` runs drive() from the Vehicle part of the Car
- `v->openTrunk();` NOT POSSIBLE!
- But: `((Car *)v)->openTunk();` WORKS!
  ```

Overloading in sub-classes

- We can overload a method in a sub-class

```
class Vehicle {
  void drive();
}
class Car:  public Vehicle {
  void drive();
}
```

- `Vehicle *v1 = new Vehicle();`
- `Vehicle *v2 = new Car();`
- `Car *c = new Car();`
- `v1->drive();` and `v2->drive();` run drive() from the Vehicle
- `c->drive();` runs drive() from the Car

`virtual` functions

- What if we want the object know what it “really” is and run the correct `drive()` method?
- Declare the method with the keyword `virtual`

```
class Vehicle {
  virtual void drive();
}
class Car:  public Vehicle {
  virtual void drive();
}
```

- `Vehicle *v1 = new Vehicle();`
- `Vehicle *v2 = new Car();`
- `v1->drive();` runs drive() from the Vehicle
- `v2->drive();` runs drive() from the Car

Polymorphism with `virtual` functions

- What `virtual` function to run is determined at run-time
- Depends on the “real” type of objects
- Works for both pointers and references

Interfacing: Abstract class

- In C++, abstract classes provides interfaces
- Not to be confused with data abstraction
- To make a class abstract : declare at least one of its functions as pure ”virtual” function.
- A pure virtual function is specified by placing ”= 0”
- ```
  class Car
  {
  public:
    virtual double getNrWheels() = 0; // pure
  virtual function
  private:
    double NrWheels
  };
  ```
  ```

Abstract class

- Abstract classes cannot be instantiated
- Purpose : A base classes which could be inherited in other classes
- Inherited classes have to overload each of the virtual functions in the base class
- Meaning: B (inherits the base class A) supports the interface provided by A.

Lecture 16: STL and C++1y

Template

- Templates offers a way to write code compatible with any data types
- Use it when you want a generic function or class that can work on many different data types
- We can write both template classes and functions
- Example of a template function:

```
template <typename T>
T getMax (T a, T b)
{
  if(a>b) {return a}
  else {return b}
}
```

- `getMax<int>(4,5)` returns 5
- `getMax<double>(4.2, 4.1)` returns 4.2

Standard Template Library: STL

- The Standard Template Library (STL) provides classes for:
  - Collections: lists, vectors, sets, maps
- Defined as templates: can store data of any type!
- Examples:
  - `std::list<T>`
    Ex: `std::list<std::string> names;`
  - `std::vector<T>`
    Ex: `std::vector<double> values;`
  - `std::set<T>`
    Ex: `std::set<std::string> nameOfPerson;`
  - `std::map<T1,T2>`
    Ex: `std::map<int, std::string> nameOfMonth;`
    Ex: `std::map<std::string, int> monthNumberByName;`

Standard Template Library: STL

- Different collections are optimized for different use, e.g.:
  - `std::list<T>`
    Cannot access elements with `x[i]`, need to use so called *iterators* to step through the list, can add/remove elements at low cost
  - `std::vector<T>`
    Can access elements with `x[i]`, but resizing is more costly
  - `std::set<T>`
    Does not allow for redundant elements
  - `std::map<T1,T2>`
    Provides a mapping from one object to another
- More in C++ Library Reference, e.g.
  `http://www.cplusplus.com/reference/stl/`

Often used: vector (from C++ reference)

```
// erasing from vector
#include <iostream>
#include <vector>
using namespace std;

int main ()
{
  unsigned int i;
  vector<unsigned int> myvector;

  // set some values (from 1 to 10)
  for (i=1; i<=10; i++) myvector.push_back(i);
}
```

Often used: vector (from C++ reference)

```
// erase the 6th element
myvector.erase(myvector.begin()+5);

// erase the first 3 elements:
myvector.erase(myvector.begin(),
myvector.begin()+3);

cout << "myvector contains:";
for (i=0; i<myvector.size(); i++) {
 cout << " " << myvector[i];
 cout << endl;
}
return 0;
```

Often used: iterators (from C++ reference)

```
using namespace std;
vector<int> v;
vector<int>::iterator vIt;
v.push_back(2);
v.push_back(3);
for(vIt = vIt.begin(); vIt != vIt.end(); vIt++) {
    cout<<*vIt;
}
```

Standard Template Library: STL

- What would be suitable STL structures for:
  - A queue of real time messages?
  - A set of options?
  - Modelling a shipment of products?
- Differences are important:
  - How to insert an element?
  - How to access/find an element?
  - How to remove an element?

Standard Template Library: STL

- What would be suitable STL structures for:
  - A queue of real time messages?
  - A set of options?
  - Modelling a shipment of products?
- Differences are important:
  - How to insert an element?
  - How to access/find an element?
  - How to remove an element?

STL Algorithm Library (from C++ reference)

```
#include <algorithm>
int myints[] = { 10, 20, 30 ,40 };
int * p; // pointer to array element:
p = std::find (myints,myints+4,30);
++p;
std::cout << "The elem. following 30 is ";
std::cout << *p << '\n';
```

STL Algorithm Library (from C++ reference)

```
#include <algorithm>
#include <vector>
int myints[] = {32,71,12,45,26,80,53,33};
std::vector<int> myvector (myints, myints+8);
// 32 71 12 45 26 80 53 33
// using default comparison (operator <):
std::sort (myvector.begin(), myvector.begin()+4);
//(12 32 45 71)26 80 53 33
```

STL Algorithm Library (from C++ reference)

```
#include <algorithm>
#include <vector>
int myints[] = {32,71,12,45,26,80,53,33};
std::vector<int> myvector (myints, myints+8);
// 32 71 12 45 26 80 53 33
// using default comparison (operator <):
bool myfunction (int i,int j) { return (i>j); }
std::sort(myvector.begin()+4,
          myvector.end(), myfunction);
// 32 71 12 45 (80 53 33 26)
```

Other parts of the STL and boost:

- pairs
- queues
- stacks
- the BOOST libraries, boost.org

Lecture 16: STL and C++1y

C++11

- A new revision from 2011 of C++, supported by g++
- Many improvements to C++
- activate using -std=c++11 (default?)
- example: `g++ -std=c++11 main.cpp -o main`

C++11

- Variable type inference:
- `auto a = 42;`
- `auto b = 42.01;`
- `auto c = new MyObject();`
- `auto d = myfunction(a,b,c);`

C++11

- Lambda functions: Ex: `auto func = [] () { cout << "Hello KTH"; };`

```
int main() {
    vector<int> x;
    for(int i=1;i<10;i++) { x.push_back(i);}
    auto pos = std::find_if(std::begin(x),
          std::end(x), [](int n) {return n%2==0;} );
    cout << *pos; // is 2
};
```

C++11

easy looping

```
using namespace std;
int main() {
    vector<int> x;
    for(int i=1;i<10;i++) { x.push_back(i);}
    for(auto v:x) {cout << v <<" ";};
    // 1 2 3 4 5 6 7 8 9
};
```

C++14

- Minor improvements over C++11 and bug fixes
- Extension of ”auto” data type to all functions (not just lambda)
- Digit separators: ’ character, Ex: `auto fpnum = 0.113'343`
- Template for variables as well
- Ex: `template<typename T> constexpr T pi = T(3.14159265358979323846264643L);`

Other tools for Scientific Programming:

- Python + Numpy - Simple, interpreted (no compilation)
- Java - general purpose, portable, no pointers!
- Mathematica - powerful computation
- Maple - extensive analytics
- R - majorly used in statistics
- Note: Matlab now has OO features!

A zoo of programming languages:

- Prolog
- Scala
- Haskell
- Lisp
- Ruby
- ...