11. Generic Programming and Design Patterns

8. Juli 2011
Outline

- Recapitulation
- Template Programming
- An Overview over the STL
- Design Patterns (skipped)
- Evaluation/feedback
—11.1. Recapitulation—

Important Concepts (1/2)

- von-Neumann architecture, and, in particular, the linearised memory.
- Definition of an algorithm.
- Interplay of compiler, linker, interpreted programs, and machine language.
- Idea of a variable and its instantiation.
- Overview of built-in data types—please note the default-value paradigm in context with default constructors.
- Terms *syntax* and *semantic*.
- Comments.
- Assignment statements (equal sign) in combination with case distinction.
- An idea of the rounding error (normalisation process).
- Strict vs. non-strict boolean logic.
- Different variants of branches and loops.
Important Concepts (2/2)

- Scopes.
- Notion of a function and the three call paradigms.
- Keyword const in different context.
- Recursion and its different types.
- Enums.
- Split-up of an application into header and implementation files.
- Pointers, arrays, and strings.
- Concept of 0-terminated strings.
- Structures (n-tupels).
- Dynamic memory management.
- Encapsulation.
- Methods instead of functions on structs.
- Constructor and destructor concept.
- Inheritance.
11.2. Templates

```cpp
void swap(int & a, int & b) {
    a = a+b;
    b = a-b;
    a = a-b;
}

int a, b;
double c, d;
swap(a, b);
swap(c, d);
```

We need a `swap` routine for doubles as well. As we are not allowed to introduce a common superclass of `int` and `double` (this would be the C# way), we have to do it with overloading. Analyse your solution. Is there a bad smell in it?
A Solution with Overloading

```c
void swap(int & a, int & b) {
    a = a + b;
    b = a - b;
    a = a - b;
}

void swap(double & a, double & b) {
    a = a + b;
    b = a - b;
    a = a - b;
}
```

- This is duplicated code (bad).
- For each built-in type, we have to write yet another swap routine (worse).
- Now that we are able to overload the comparison operator, we even might have to write the swap for each user-defined class, e.g. if we are using our classes within a sorted list (worst case).
Templates

```cpp
template <class T> void swap ( T& a , T& b ) {
    T c(a); a=b; b=c;
}

int a, b;
swap(a,b);  // or
swap<int>(a,b);
```

- Templates are a technique to write code for an yet undefined type.
- We instantiate them implicitly whenever we use them.
- Templates have to be written into headers only (it's still cut-n-paste).
- The requirements on the datatype (we need a copy-constructor or there has to be a comparison operator) are given implicitly.
- We can resolve disambiguities explicitly.
- Template error messages are difficult to understand (you’ll see later on).
Templates Imply Copy ’n Paste

\[
\text{template } <\text{class T}> \ \text{void} \ \text{swap} ( \ T& \ a, \ T& \ b ) \ { \\
\quad \ T \ c(a); \ a=b; \ b=c; \\
\}
\]

\[
\text{int} \ a, \ b; \ \text{double} \ c, d; \\
\text{swap}(a,b); \\
\text{swap}(c,d);
\]

becomes internally

\[
\text{void} \ \text{swap} ( \ \text{double}& \ a, \ \text{double}& \ b ) \ { \\
\quad \ \text{double} \ c(a); \ a=b; \ b=c; \\
\}
\]

\[
\text{void} \ \text{swap} ( \ \text{int}& \ a, \ \text{int}& \ b ) \ { \\
\quad \ \text{int} \ c(a); \ a=b; \ b=c; \\
\}
\]

\[
\text{int} \ a, \ b; \ \text{double} \ c, d; \\
\text{swap}(a,b); \\
\text{swap}(c,d);
\]

and maps to overloading, i.e. templates blow up your code.
Generic Classes

```cpp
template <class T>
class Orbit {
    private:
        T _myCelestialBody;
    public:
        Orbit(CelestialBody body);
        void setTimeStep(double tau);
        void computeNextPosition() {
            ...
            ... = _myCelestialBody.getMass() * ...;
            ...
        }
    }

    Orbit<Sun> myOrbit(mySun);
```

- Whole classes can be defined with respect to an yet undefined data type.
- You can throw in any type (class) that has a `getMass()` operation.
Generic Classes vs. Interfaces

From an OO point of view, it would be better to make `Orbit` expect an implementation of an interface with the operation `getMass()`. However,

- then it would not work with built-in types, and
- we often don’t want hundreds of classes inherit from the same interface.
Technical Classes

```cpp
template <class T>
class OrderedSequence {
    private:
        T* _sequenceElements;
    public:
        void addEntry(T newEntry) {
            insert and do bubble sort
        }
    }
```

- Such a class should work for any (built-in) type that has a comparison operator.
- And there's lots of these technical classes (containers in particular).
—11.3. The Standard Template Library—

```cpp
namespace std {
  template <class T>
  void swap ( T& a, T& b ) {
    T c(a); a=b; b=c;
  }
}
```

- C++ comes along with a whole bunch of different technical classes contained in the namespace `std` (Standard).
- They are all contained realised as templates (Template),
- and are available with each C++ installation (Library).
The Most Famous Generic Stuff

- Strings (yes we can also handle something besides chars).
- Containers
- Iterators
- Standard algorithms (skipped)
- Math
#include <string>

... 
std::string s0 = "This is a";
std::string s1("test");
std::string s3 = s0 + "\n" + s1;

- size(): int
- clear(): void
- empty(): bool
- operator[]: char
- append(string)
- insert(string)
- replace(string,string)
Containers—Vector

A container is a data structure storing other objects. Our single linked list, e.g., was a container for integers. The STL containers all are templates, i.e. they work with any datatype.

```cpp
#include <vector>

... std::vector<double> myVector;

myVector.push_back(1.0);
myVector.push_back(2.0);
myVector.push_back(3.0);

if (myVector.empty()) {
    ...
if (myVector[2] == 4.3) {
    ...
myVector.clear();
```

The vector represents an array that can grow/shrink on-demand. Almost noone in C++ uses pure arrays anymore if there’s no need to do so. The dynamic memory management (due to new/delete) and a proper copying mechanism are hidden.
Containers—List

- A vector allows arbitrary elements access and
grows and shrinks (automatically), however
inserting and removing data is expensive as it induces copying.

```cpp
#include <list>
...
std::list<double> myList;
myList.push_back(1.0);
myList.push_back(2.0);
myList.push_back(3.0);
```
Containers—Stack and Queue

- A stack is a container where you can only add/remove elements at the end (LIFO).
- A queue is a container where you can remove elements only in the order you’ve inserted them (FIFO).
- vector basically gives you everything you need, but the stack and the queue class are faster.

```cpp
#include <stack>

... std::stack<double> myStack;
std::queue<double> myQueue;

myStack.push_back(1.0); myQueue.push_back(1.0);
myStack.push_back(2.0); myQueue.push_back(2.0);
myStack.push_back(3.0); myQueue.push_back(3.0);

myStack.back(); // gives you a 3.0
myQueue.front(); // gives you a 1.0

myStack.push_back();
myQueue.pop_back();
```
**Associative Containers—Maps**

```cpp
#include <map>
...

std::map<int, double> myMap;

myMap[1] = 0.3;
myMap[2] = 1.22;
myMap[16] = 5;
myMap[-4] = 9.99;

if (myMap.find(1) != myMap.end()) {
    ...
```

- Access similar to vector, but
- datastructure does not hold list but mappings from keys to values.
- There’s also variants of this map such as a hash map.
- Question: Why is it dangerous to have a double as key.
Iterators

- If you wanna traverse an array or a `std::vector`, you might use a simple for loop.
- However, most containers neither have an `[ ]` operator nor do they provide constant-time single element access (see our single linked list).
- The STL hence introduces the concept of an iterator.

```cpp
std::vector<double> myVector;

std::vector<double>::iterator p = myVector.begin();
while (p != myVector.end()) {
    if (⋆p == 0.4) {
        ...
        p++;
    }
}
```

Iterators have a signature similar to a pointer traversing an array, but they are classes and hide the implementation of the container.
Iterators

- If you wanna traverse an array or a `std::vector`, you might use a simple for loop.
- However, most containers neither have an `[ . ]` operator nor do they provide constant-time single element access (see our single linked list).
- The STL hence introduces the concept of an iterator.

```cpp
std::vector<double> myVector;

std::vector<double>::iterator p = myVector.begin();
while (p != myVector.end()) {
    if (*p == 0.4) {
        // ...  
        p++;
    }
}
```

Iterators have a signature similar to a pointer traversing an array, but they are classes and hide the implementation of the container.

```cpp
std::list<double> myVector;

std::list<double>::iterator p = myVector.begin();
while (p != myVector.end()) {
    if (*p == 0.4) {
        // ...  
        p++;
    }
}
Scientific computing

- We already used functions such as `std::sin` or `std::sqrt`.
- All of them are defined in `<cmath>`.
- There’s also a library for complex numbers (complex),
- and some tools for ranges:

```cpp
#include <numeric_limits>

double maxOfDouble = std::numeric_limits<double>::max();
int howManyDigits = std::numeric_limits<double>::digits();
bool isItSigned = std::numeric_limits<double>::is_signed();
double accuracy = std::numeric_limits<double>::epsilon();
```

The class `numeric_limits` does exist for each built-in datatype, and it provides exclusively static operations.
—11.4. Evaluation—

Hail to the international students in Munich

This all leads to ...
... Constructive Criticism

Das die Folien aus Englisch waren und durch die eher alltäglichen Inhalt war man zur Anwesenheit gezogen, obwohl man den Stoff nicht.

The sound of instructor is so peaceful, so that one can sleep in lecture.
About the slides

The slides are far from complete (and without bugs), however, they shall not and cannot replace any book. Their purpose is to guide you through books, the internet, tutorials, and to highlight some things that are important and are typcially not discussed in (introductionary) literature.
Literature

- Martin Fowler: *Refactoring*
- Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides. *Design Patterns: Elements of Reusable Software*
- Randall Hyde: *Write Great Code I: Understanding the machine*
- Randall Hyde: *Write Great Code II: Thinking Low-level, Writing High-level*
- Scott Meyers. *Effective C++*
- Scott Meyers. *More Effective C++*
- Walter Savitch. *Absolute C++*
- Bjarne Stroustrup. *The C++ Programming Language*
Feedback

I appreciate feedback written down as email if it's positive (please CC then the right persons, i.e. representatives, professors, the president, ... ;-)). However, perhaps there are also things/ideas how we could do something better. For such remarks, it is often useful to discussed them in the group:

- Speed and amount of content as well as level of difficulty.
- Correlation to current research, other courses, and day-to-day business.
- Structure, explanations, and examples.
- Quality of slides, preparation of lecturer, and poise.
- Would you recommend this lecture because of the content/because of the lecturer?
- Are there missing topics that should be discussed?

Inspired by the Vorlesungsumfrage of TUM.
Finally ...

Thank you for your participation!

Tobias Weinzierl
tobias.weinzierl@mytum.de