02393 Programming in C++
Module 8: Classes and Objects III
Templates and Inheritance

Sebastian Mödersheim

October 24, 2016

Lecture Plan

#	Date	Topic
1	29.8.	Introduction
2	5.9.	Basic C++
3	12.9.	Data Types, Pointers
4	19.9.	
5	26.9.	Libraries and Interfaces; Containers
6	3.10.	Classes and Objects I
7	10.10.	Classes and Objects II
		Efterårsferie
8	24.10.	Classes and Objects III
9	31.10.	Recursive Programming
10	7.11.	Lists
11	14.11.	Trees
12	21.11.	Novel C++ features
13	28.11.	Summary
	5.12.	Exam

Copying vectors

- What will happen in the following code snippet?
- How are vectors copied?

```
vec f(vec v){...}
int main(){
  vec v1;
  ...
  vec v2=v1;
  ...
  v1=v2;
  ...
  v2=f(v1);
}
```

Copying vectors

- What will happen in the following code snippet?
- How are vectors copied?

```
vec f(vec v){...}
int main(){
  vec v1;
  ...
  vec v2=v1;
  ...
  v1=v2;
  ...
  v2=f(v1);
}
```

- Default behavior: C++ makes a copy of the member variables.

Copying vectors

- What will happen in the following code snippet?
- How are vectors copied?

```
vec f(vec v){...}
int main(){
  vec v1;
  ...
  vec v2=v1;
  ...
  v1=v2;
  ...
  v2=f(v1);
}
```

- Default behavior: C++ makes a copy of the member variables.
- How can we change that default behavior?

Copy Constructor

- Defining a copy constructor for class `vec`.
  `vec(const vec & v)`
  - ★ additional constructor that constructs a vector given an existing one.
  - ★ General form `classname(const classname & v)`
- Will be called:

```
vec f(vec v){...}
  //here for v (with argument v1 from main)
int main(){
  vec v1;
  ...
  vec v2=v1; // Here for v2 (with argument v1)
  ...
  v1=v2;
  ...
  v2=f(v1);
}
```

Assignment Operator

- Defining an assignment operator for class `vec`.

  `vec & operator=(const vec & v)`
  - ★ The "overwrite" the present vector with vector `v`.
  - ★ Result: reference to the present vector
  - ★ Pitfalls:
    - ▶ Check for self-assignment (so that `v=v;` does not crash)
    - ▶ Remember to de-allocate any allocated space of the old vector before overwriting variables.
  - ★ In general: `classname & operator=(const classname & v)`
- Will be called:

```
vec f(vec v){...}
int main(){
  vec v1;
  vec v2=v1;
  v1=v2; // Here for v1 (with argument v2)
  v2=f(v1); //Here for v2 (with the result
}           // of f(v1) as argument)
```

Abstract collection

- Dynamic size.
- Access through `operator[]`.
- Iterators (`begin` and `end`).
- Entries ordered from first to last.
- Traversing entries with `i++` and `i--`.
- Access entry with `*i`.

DRY

- We have defined a vector-class for integers.
- The class for vectors of strings for instance, would be very similar, exchanging just in several parts `int` with `string`.

DRY

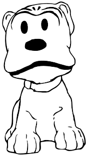
- We have defined a vector-class for integers.
- The class for vectors of strings for instance, would be very similar, exchanging just in several parts `int` with `string`.
- New Paradigm: DRY — Don't Repeat Yourself

DRY

- We have defined a vector-class for integers.
- The class for vectors of strings for instance, would be very similar, exchanging just in several parts `int` with `string`.
- New Paradigm: DRY — Don't Repeat Yourself
  - ★ Less code to write.
  - ★ Less code to understand.
  - ★ One fix applies to several areas.

DRY

- We have defined a vector-class for integers.
- The class for vectors of strings for instance, would be very similar, exchanging just in several parts `int` with `string`.
- New Paradigm: DRY — Don't Repeat Yourself
  - ★ Less code to write.
  - ★ Less code to understand.
  - ★ One fix applies to several areas.
  - ★ Every time you repeat yourself god kills a puppy.



Templates

- Example:

```
template <typename T>
class vec{...}
```

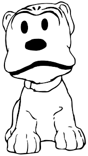
  defines that the class `vec` is parameterized over a type `T`.
- `T` can be used in the entire code of `vec` instead of `int`.
- We can declare `vec<int> v` and get `vec` where every occurrence of `T` is replaced by `int`.
- Similar for `vec<string> v` or `vec<vec<int> >`.
  - ★ Note you have to put a space between the two > (why?)

Templates

- Example:

  ```
  template <typename T>
  class vec{...}
  ```

  defines that the class `vec` is parameterized over a type `T`.
- `T` can be used in the entire code of `vec` instead of `int`.
- We can declare `vec<int> v` and get `vec` where every occurrence of `T` is replaced by `int`.
- Similar for `vec<string> v` or `vec<vec<int> >`.
  - ★ Note you have to put a space between the two > (why?)
- Many puppies saved!



Templates in .cpp

- Scenario: you have specified
  - header file: `template <typename T> class vec{...}` with only function prototypes
  - the actual implementation is in a separate `.cpp` file.
- This is how it should be, but does not work (in most compilers):
  - When compiling the `.cpp` file, the compiler does not know for which datatypes `T` the rest of the program will be using `vec<T>`.

Templates in .cpp

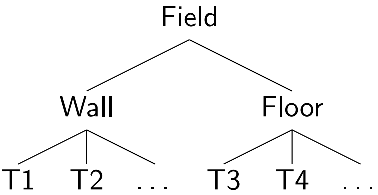
- Scenario: you have specified
  - header file: `template <typename T> class vec{...}` with only function prototypes
  - the actual implementation is in a separate `.cpp` file.
- This is how it should be, but does not work (in most compilers):
  - When compiling the `.cpp` file, the compiler does not know for which datatypes `T` the rest of the program will be using `vec<T>`.
- Possible solutions
  - insert the function definitions inside the class definition of the header file
    - Disadvantage: i.e. no separation of header from implementation)
  - use `include "..."` on the `.cpp` file whenever needed.
    - Disadvantage: Does not work with all compilers if several files of a project that include the library ...
  - write in the `.cpp` file which instances are needed, e.g.: `template class vec<int>;`
    - Disadvantage: this means listing as part of your library all datatypes for which it can be used...

Example: Enigma3D

The game framework Enigma3D is an example for making use of OOP:

- One can create new objects for the game in a modular way
  - ★ No change of other game code (almost), just add a new class!
  - ★ All objects are instances of a class `Field`.
  - ★ The main loop of the game deals only with objects of type `Field`, but not anything specific about your subclass.
- In general: allows many developers to work independently without synchronizing. A class gives a general framework to which everybody can specify their specific instances.
- With inheritance we can thus define an interface between different modules.

Inheritance for Enigma3D



- The game playground consists of different kinds of Fields.
  Goal: teams T_1, T_2,... can independently develop their fields.
- We use inheritance:
  - ★ The abstract class `Field` defines all methods that every field must implement.
  - ★ The abstract classes `Wall` and `Field` are subclasses of Fields, and further define the methods that every wall/floor field must implement.
  - ★ Each team develops subclasses of `Wall` or `Field`, implementing the required methods.
  - ★ Child class may overwrite methods of the parent class.

Live Programming

Review our maze class from exercise 3:

- Use the class structure to define different types of fields
- Moving "field-specific stuff" from main into the individual classes.

Inheritance: from subtypes to subclasses

Example of subclass/subtype relations:

- Every integer is a real;
- Every square is a rectangle;
- Every `HourlyEmployee` is an `Employee`;
- etc.

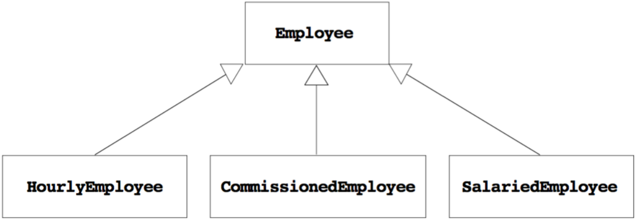
When is this useful?

- Bottom-up perspective (generalization)
  "we have classes for different kinds of Employees which share some functionalities... let us group them together."
- Top-down perspective (specialization)
  "the class of employees is full of specialized code for particular kinds of employees... let us separate them in different classes."

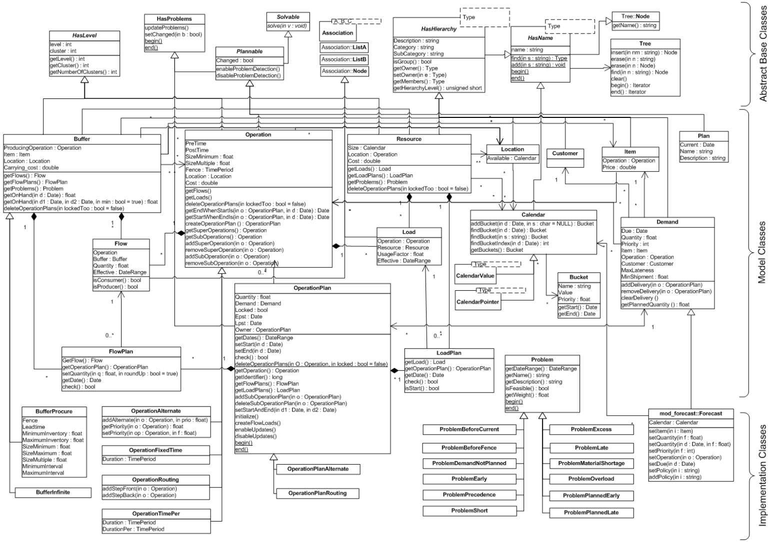
Advantages: modularity, clarity, maintanability, etc.

From "is-a" relations to class diagrams

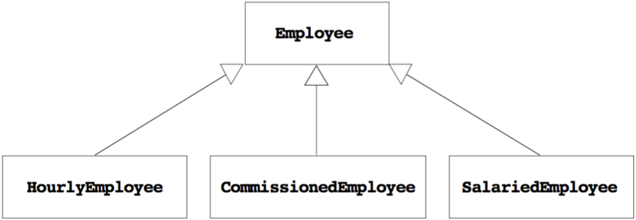
"Every `HourlyEmployee` *is an* `Employee`*."*
"Every `CommissionedEmployee` *is an* `Employee`*."*
"Every `SalariedEmployee` *is an* `Employee`*."*



Diagrams in real life



From diagrams to code



In C++ we write something like

```
class Employee {
    ...
}

class HourlyEmployee : Employee {
    ...
}

...
```

Inheritance: more

```
class B : A ...
```

What is actually inherited?

- `B` inherits (almost) all `public` and `protected` members.
- `B` does not inherit `private methods` of `A`.

What happens to the interface?

- It depends. Actually, we can write `class A :` *p* `B` with *p* being either `public`, `protected` or `private` (default).
- Depending on the choice of *p*, the members of *B* will be `public`, `protected` or `private`.

What happens to the methods?

- In some cases we can override/specialise them.
- In some cases we must override/specialise them.

Will my specialised method be used? ... not always!

Encapsulation

The access to members of a class can be controlled:

- `public` members are accessible by everyone;
- `protected` members are accessible by objects of the class and derived classes;
- `private` members are accessible by objects of the class and no one else (default).

This is useful to hide implementation details and also to protect the implementation from unintended or malicious use.

Encapsulation and Inheritance

class B: **public** A ...

- B inherits `public` members, which remain `public`;
- B inherits `protected` members, which remain `protected`.

class B : **protected** A ...

- B inherits `public` members, which become `protected`;
- B inherits `protected` members, which remain `protected`.

class B : **private** A ...

- B inherits `public` members, which become `private`;
- B inherits `protected` members, which become `private`.

Encapsulation and Inheritance

```
class A {
public:
    int x; // accessible to everyone
protected:
    int y; // accessible to all derived classes (A, B, C, D)
private:
    int z; // accessible only to A
};

class B : public A {
    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A {
    // x is protected
    // y is protected
    // z is not accessible from C
};

class D : private A {
    // x is private
    // y is private
    // z is not accessible from D
};
```

Refining methods

A method `f` inherited from `A` can be refined if we need to write specialized code for the subclass `B`.

If we want the *intuitive* behaviour (call `B::f` for objects of class `B`) `f` must be marked as `virtual` in `A`. This is realised by so-called *dynamic* dispatch.

Otherwise, the method dispatch is *static* (i.e. decided at static time by the compiler), which means that sometimes `A::f` may be called for objects of class `B`!

Static dispatch is more performant.

Refining Methods

```
class A {
public:
    void f(void) = { ... };
    virtual void g(void) = { ... };
};

class B : public A {
public:
    void f(void) = { ... };
    void g(void) = { ... };
};

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

    b.f();  // calls B::f()
    p->f(); // calls A::f()
            // due to static binding based on p's type

    b.g();  // calls B::g()
    p->g(); // calls B::g()
            // due to dynamic binding
}
```

Abstract Classes

Abstract classes

- Cannot be instantiated;
- Define an abstract interface for derived classes;
- Are specified by at least one *pure* virtual function
  `virtual void someMethod() = 0;`
  which *must* be overriden by derived classes

Example

```
class Employee {
public:
    String name(void);
    virtual double salary(void) = 0 ;
    ...
};

class HourlyEmployee : public Employee {
public:
    void double salary(void) = { ... };
};
```

Constructors and Inheritance

```
class B: A { ... }
```

Constructors and Inheritance can be tricky:

- Constructors are not inherited, B may need to define its own constructors;
- Before constructing an object `B` the constructor of class `A` needs to be invoked;