7. Object-based Programming

June 14, 2010
Outline

- Recapitulation
- Operations on Structs: Methods
- Constructors and Destructors
- The Const Modifier
- Encapsulation & Information Hiding
Recapitulation

```c
int a;
int* b;
a = 3;
b = &a;
b = 4;
*b = 5;
```

- Explain/sketch what the individual lines do.
- What does \texttt{b++};
- What happens if we write

```c
int* a, b;
```

- What happens if we call a function \texttt{foo(int*)} with \texttt{b}. 
—7.1. Operations on Structs—

```c
/* Represents a date. 
 * Each month shall have 30 days. 
 */
struct Date {
    int month, day;
};

void switchToNextDay(
    Date& date
) {
    ...
}
```
Class Diagrams with the Unified Modelling Language

- The Unified Modelling Language is a graphical representation of your system design.
- Operation and the data here go hand-in-hand. UML illustrates this fact.
- Object-based paradigm: Model whole system in terms of structs and operations acting on these structs.
- Remember: Structs can hold other structs as *attributes*.
- Remember: Structs can hold pointers to other structs.
Aggregation and Composition

ImportantNote

Date
- int month
- int day
- switchToNextDay()
/** Represents a date. Each month shall have 30 days */
struct Date {
    int month, day;
};

// Declaration
void switchToNextDay(Date& date) {
    ...
}

/*** Represents a date. Each month shall have 30 days **/
struct Date {
    int month, day;
    // Belongs to the struct it is embedded into
    void switchToNextDay();
};
Operations on Structs Rewritten

```
// Date.h
struct Date {
    int month, day;
    void switchToNextDay();
};

// Date.cpp
#include "Date.h"

void Date::switchToNextDay() {
    day++;
    ...
}
```

- Syntax is similar to namespaces.
- It is now clear, how operations and data belong together.
- A good object always works solely on data of its own. If it has to manipulate data from another object, it should use this object’s functions (operations, methods).
- Internally, it is still your straightforward C realisation.
Invoke Operations on Structs

```c
// Date.h
struct Date {
    int month, day;
    void switchToNextDay();
};

...  
Date myDate;
myDate.month = 7;
myDate.day = 5;
myDate.switchToNextDay();
```

- Create a variable: *instantiate*.
- Variable: *object*.
- Variables of a struct: *attributes*.
- Call a struct's operation: invoke a *method*.
---7.2. Constructors and Destructors---

```c
// Date.h
struct Date {
    int month, day;
    void switchToNextDay();
};

...

Date myDate;
myDate.month = 7;
myDate.day = 5;
myDate.switchToNextDay();
```

- The two set operations also belong to the struct itself.
- It might be good design to write an initialisation operation.
- Add a method `init(int month, int day);`.
Constructors

- A constructor is a special type of operation.
- Its name equals the struct name.
- It has no return type.
- It uses initialisation lists.

```c++
// Date.h
struct Date {
    int _month, _day;
    void switchToNextDay();
    Date(int month, int day);
};

// Date.cpp
Date::Date(int month, int day):
    _month(month),
    _day(day) {
    // some checks
}
```
Creating an Instance

```c
// Date.h
struct Date {
    int _month, _day;
    void switchToNextDay();
    Date(int month, int day);
    Date();
    Date(const char* stringRepresentation);
};

// Another file
Date myDate1(7, 5);
Date myDate2(); // doesn't work
Date myDate3;
Date myDate4("5 July");
```

- We can overload constructors.
- The default constructor is invoked without brackets.
- There is no need for a default constructor.
- If you don’t provide a constructor at all, C++ automatically (in the background) generates a default constructor.
Object Destruction

```
...  
while (something) {
    Date myDate(...);
    // we do something
}
```

- Instance of Date is destroyed at the end of the scope.
- If there is a constructor, why isn’t there a counterpart?
Destructor Syntax

```c++
// Date.h
struct Date {
    int _month, _day;
    void switchToNextDay();
    Date(int month, int day);
    Date();
    ~Date();
};

// Date.cpp
Date::~Date() {
}
```

- Destructor is called always at the end of the scope containing the object.
- We cannot overload destructors.
- Destructors don’t have a return value.
- A destructor is never called explicitly.
- If you don’t define a destructor, C++ automatically generates one.
Objects on the Heap

```c
// Date.h
struct Date {
    int _month, _day;
    void switchToNextDay();
    Date(int month, int day);
    Date();
    ~Date();
};

...
Date* myDate = new Date(7,5);
...
myDate->switchToNextDay();
...
delete myDate;
```

- new reserves memory (as it does in C), and
- new invokes the constructor.
- delete invokes the destructor, and
- it frees the memory (as it does in C).
- Remember of linked list: If we delete the first list entry, this entry can also free the subsequent entries. It is much easier to enforce the consistency with these new data structures.
Arrays of Objects

```cpp
// Date.h
struct Date {
    int _month, _day;
    void switchToNextDay();
    Date(int month, int day);
    Date();
    ~Date();
};

Date myDates[10];
...
myDate[4] -> switchToNextDay();
...
```

- Arrays of objects are supported by C++.
- However, such data structures have to have a standard constructor.
Explicit Constructor Calls

```c
// Date.h
struct Date {
    int _month, _day;
    void switchToNextDay();
    Date(int month, int day);
    Date();
    ~Date();
};

Date myDate1 = Date(7, 5);
Date myDate2 = Date();
...
```

- Here, we need the parentheses.
- Obviously, the constructor is kind of a function which returns a struct.
- This is bad style, as it induces a bit-wise copy.
—7.3. Const Methods—

... why does this snippet not work?

```c
struct IntegerEntry {
    int value;
    IntegerEntry* next;
    // Print this entry to terminal and continue with next entry.
    void printList();
    // Make argument next argument.
    void append(IntegerEntry* nextEntry);
};

void doSomething(const IntegerEntry& entry) {
    ...
    entry.printList();
}
```

Call-by-value is expensive, as it copies the whole object bit-wise. This lead to a performance breakdown in this code. Thus, we used call-by-const-reference. However, the code now does not compile anymore.
Const Keyword

```c
struct IntegerEntry {
    int value;
    IntegerEntry* next;
    // Print this entry to terminal and continue with next entry.
    void printList() const;
    // Make argument next argument.
    void append(IntegerEntry* nextEntry);
};

...  

void doSomething(const IntegerEntry& entry) {
    ...
    entry.printList();
}
```
Semantics of Const

```c
struct IntegerEntry {
    int value;
    IntegerEntry* next;
    // Print this entry to terminal and continue with next entry.
    void printList() const;
};

void IntegerEntry::printList() const {
    value = 2; // error
    return value; // o.k.
}
```

- `const` operations may not alter object state.
- `const` operations may not call non-const methods.
- `const` operations can be invoked on objects passed by call-by-const-reference.
- `const` operations allow the compiler to optimise.
- `const` operations allow user to enforce encapsulation and to write safer apps.

⇒ use the `const` modifier whenever possible.
Const Variants

```c
struct IntegerEntry {
    int value;
    IntegerEntry* next;

    const int getValue();
    const int getValue() const;

    // variant A
    int getValue() const;
    int getValue();

    // variant B
    const int& getValue() const;
    const int& getValue();
};
```
Const Variants

```c
struct IntegerEntry {
    int value;
    IntegerEntry* next;

    const int getValue();
    const int getValue() const;

    // variant A
    int getValue() const;
    int getValue();

    // variant B
    const int& getValue() const;
    const int& getValue();
};
```

- `const` belongs to the signature, i.e. we can overload with respect to `const`.
- `const` after the operation enforces the operation not to manipulate object state.
- `const` before the return argument does not allow programmer to manipulate result.
- The compiler tries to use `const` operations before it falls back to non-const operations.
With the new technique at hand, we can encapsulate data and operations as these two things go hand in hand.

We can write a couple of setter and getter operations to allow the user to manipulate our brand new data structure.

However, the use still can reset attributes manually. We can not forbid this.

Consequently, we “need“ an alterantive, new technique to forbid this.

Furthermore, it would be nice if the user doesn’t even see the attributes, as it might be reasonable that the user can’t even read attributes if we don’t provide the corresponding getters (next pointer in our list example, e.g.).
Our Beloved Colleagues

```c
/* * * Represents a date. */
/* * Each month shall have 30 days. */

struct Date {
    int month, day;
    void switchToNextDay;
};

// this is the code our colleague wrote
Date myDate( ... );
myDate.day++;
Encapsulation

```cpp
/* *\* Represents a date.
 * Each month shall have 30 days.
 */
class Date {
  private:
    int month, day;
  public:
    void switchToNextDay();
};
```

- In principle, class is an alias for struct,
- i.e. we can do all the things we can do with structs with classes, too.
- However, for classes we can create public and private sections. Only add them in the header.
- Private attributes and operations are not available from outside, but only within the operations of the class (*encapsulation*).
Encapsulation at Work

```cpp
/**
 * Represents a date.
 * Each month shall have 30 days.
 */
class Date {
    private:
        int _month, _day;
    public:
        void switchToNextDay();
};

...  

void Date::switchToNextDay() {
    _day++;  // o.k.
}

Date myDate;
myDate._day++;  // doesn't work.
```
Encapsulation and the Object Lifecycle

```cpp
/** Represents a date.
 * Each month shall have 30 days.
 */

class Date {
    private:
        int _month, _day;
    Date();

    public:
        void switchToNextDay();
    Date(int month, int day);
};
```

- We can make constructors private and, thus, forbid everybody to create an instance of our object.
- We could make the destructor private, too. However, this never makes sense.
namespace calendar {
    /** Represents a date.
     * Each month shall have 30 days.
     */
    class Date {
        private:
            int _month, _day;
        Date();
        public:
            void switchToNextDay();
            Date( int month, int day );
    };
}

void calendar::Date::switchToNextDay() {
    ...
}

// fully qualified argument here not necessary
// we can access private arguments of other instances
void calendar::Date::copyFromOtherDate(const calendar::Date& date) {
    ...
}
Remark: Classes and Recursion

```cpp
class IntegerEntry {
    private:
        int value;
        IntegerEntry* next;
    public:
        void append(IntegerEntry* newEntry);
};

void IntegerEntry::append(IntegerEntry* newEntry) {
    if (next == 0) {
        next = newEntry;
    } else {
        next->append(newEntry);
    }
}
```

- This is not recursion, as the operation is invoked on another object.
- However, recursion and object-based programming work together.