Advanced Programming

Part I: Objects in `C++`

Alexander Breuer, Tobias Weinzierl

January 09, 2014

Getting an Overview

Lectures

.le. 11/21/13 Heap, Cache, Scopes, ..

11/28/13 - 12/19/13 Performance Oriented Programming

.ge. 01/09/14 `C++` High Level (From Objects to `C++` 11)

Objects: Overview

Basics and Recap Getting a Solid Foundation

- Definition of an Object
- Relationships between Objects
- Pros and Cons

Objects in `C++` and C Syntax and Concepts

- Skeleton of a class with member variables, member functions and access control
- Implementation of the class: Constructor/destructor and member functions
- Runner: Using classes

Objects in `C++` continued Advanced Syntax and Concepts

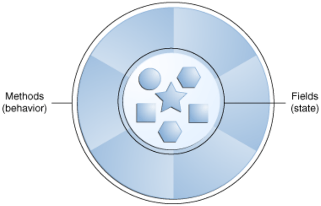
- Copy Constructor
- Assignment Operator
- Static & Const Members

OO-oriented MD SoA with Dynamic Memory Allocation as Object

Objects: What's that about?

- ”In C, a struct is an agglomeration of data [...] with functions in the package, the structure becomes a new creature, capable of describing both **characteristics** (like a C struct does) and **behaviors**.” – *Bruce Eckel, Thinking In C++, Volume One: Introduction to Standard C++*
- ”Learning the **fundamentals** of a programming language is one thing; learning how to design and implement **effective** programs in that language is something else entirely. This is especially true of C++, a language boasting an uncommon range of power and expressiveness. Built atop a full-featured conventional language (C), it also offers a wide range of **object-oriented** features, as well as support for **templates** and **exceptions**. [...] Used without **discipline**, C++ can lead to code that is incomprehensible, unmaintainable, inextensible, inefficient, and just plain wrong.” – *Scott Meyers, Effective C++*

Object: Definition

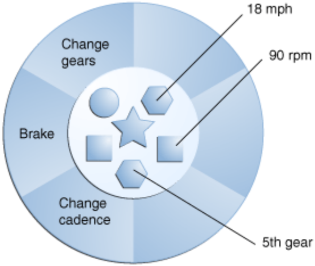


Oracle, The Java™ Tutorials

Definition

- **State**: Encapsulated in **member variables**/fields (built-in datatypes and other objects)
- **Behavior**: **Member functions**/methods operate on the internal state of the object, primary mechanism for object-to-object communication
- **Information-hiding**: Internal state usually hidden, interaction/modification via member functions

Object: Example

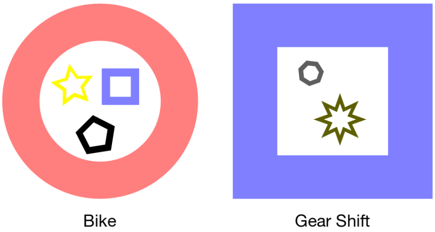


Oracle, The Java™ Tutorials

Example: Bike

- State: Speed, Revolutions per Minute, Gear, ..
- Behavior: Change gears, change cadence, brake, ..
- Information-hiding: User interacts with gear shift, brake, cadence; Doesn't modify Speed, RPMs, .. directly

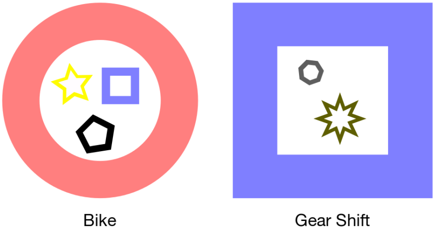
Relationships between Objects: ”Part-Of”



Part-Of releationship

Object A is with object B in an **part-of releationship**, if A is part of B (A is one of the parts the machinery of B is assembled of); Example: Object ”gear shift” can be modeled in a part-of relation to the object ”bike”

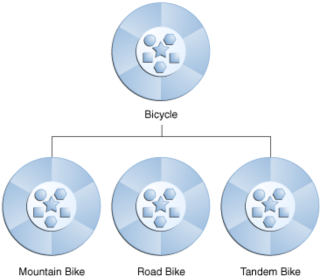
”Object Member Variables” model ”Part-Of”



Object Member Variables

OO-programming allows objects to be build of other objects not only built-in data-types: This allows for a modular implementation and code reuse

Relationships between Objects: ”Is-A”

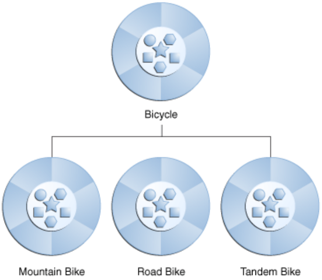


Oracle, The Java™ Tutorials

Is-A releationship

Object A is with object B in an **is-a releationship**, if A behaves like B, but adds additional features (B is a blueprint for A); Example: Objects ”mountain bike” and ”road bike” are modeled to be in a is-a relation to the object ”bicycle”

"Inheritance" models "Is-A"



Oracle, The Java™ Tutorials

Inheritance

OO-programming allows to inherit states and behavior from other objects; allows for **abstraction/specialization** in the implementation

Why should('nt?) you program OO?

Pros

- **Modularity**: Implementation of different objects usually written independently.
- Information-hiding: Internal implementation remains **hidden** from the outside world (can be changed easily even for heavy used objects)
- **Code reuse**: To (re)use an object you only have to understand the member-functions visible to you; Interfaces in software projects and/or libraries

Con

- Performance: Wrong usage of OO-language constructs might come with a significant overhead; You have to understand what happens under the hood!

Objects: Overview

Basics and Recap Getting a Solid Foundation

- Definition of an Object
- Relationships between Objects
- Pros and Cons

Objects in `C++` and C Syntax and Concepts

- Skeleton of a class with member variables, member functions and access control
- Implementation of the class: Constructor/destructor and member functions
- Runner: Using classes

Objects in `C++` continued Advanced Syntax and Concepts

- Copy Constructor
- Assignment Operator
- Static & Const Members

OO-oriented MD SoA with Dynamic Memory Allocation as Object

Objects in `C++`

Bikes as Objects

- Task: Model a bike in `C++`
- Goal: Learn the `C++` concepts on the way

Design

Gear Shift
- State: Current gear, #gears
- Behavior: Increase gear, decrease gear, get current gear

Bike
- State: Current Speed, revolutions per minute, Gear shift
- Behavior: Change gear, brake, change cadence

Overview: Bikes as Objects

Design

Gear Shift
- State: Current gear, #gears
- Behavior: Increase gear, decrease gear, get current gear

Bike
- State: Current Speed, revolutions per minute, Gear shift
- Behavior: Change gear, brake, change cadence

Overview

Next Skeleton (header file) of the bike class with member variables, member functions and access control.

The ”class” keyword

Source

```
#ifndef BIKE_H_

#include <cmath>
#include <iostream>

#include "GearShift.h"

class Bike {

};

#endif
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/bike/Bike.h`

Keyword

class defines an object in `C++` and builds the frame for the OO-concepts (member variables, member functions, inheritance, ..)

State: Member Variables

Source

```
class Bike {
  //private:
    //! current speed
    float m_speed;

    //! current rpm
    float m_revolutions;

    //! gear shift
    GearShift m_gearShift;

};
```

code: https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/bike/Bike.h

State

Member variables define the state of an object. Member variable can bee built-in or custom objects.

Behavior: Member Functions

```
class Bike {

  public:
    /**
     * Constructor
     *
     * @param i_numberOfGears #gears
     **/
    Bike( int i_numberOfGears );

    /**
     * Destructor
     **/
    ~Bike();

    /**
     * Changes the gear to the closest possible gear.
     *
     * @param i_gear gear to change to.
     **/
    void changeGear( int i_gear );

};
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/bike/Bike.h`

Behavior: Member Functions (2)

```
class Bike {

  public:

    /**
     * Brakes the bike.
     **/
    void brake();

    /**
     * Accelerates the bike.
     *
     * @param i_rpm revolutions per minute.
     **/
    void changeCadence( float i_rpm );

};
```

code: https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/bike/Bike.h

Behavior

Member functions operate on the internal state of the object, primary mechanism for object-to-object communication.

Behavior: Member Functions (3)

Source

```
class Bike {

  public:


    /**
     * Prints status of our bike.
     **/
    void printInformation() const;
};
```

code: https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/bike/Bike.h

Behavior

Member functions operate on the internal state of the object, primary mechanism for object-to-object communication.

Access Control

Source

```
class Bike {
  //private:
  public:
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/bike/Bike.h`

Accessibility

- Restrictive access control helps to write clean code and uses the compiler to enforce your interfaces.
- **private**: Members and **friends** (later)
- **protected**: Members and friends of the class & of **derived** classes (later)
- **public**: Anyone

Overview: Bikes as Objects

Design

Gear Shift
- State: Current gear, #gears
- Behavior: Increase gear, decrease gear, get current gear

Bike
- State: Current Speed, revolutions per minute, Gear shift
- Behavior: Change gear, brake, change cadence

Overview

Previous Skeleton (header file) of the bike class with member variables, member functions and access control.

Next Implementation (source file) of the bike class & a runner.

Implementation: Constructor

Source

```cpp
#include "Bike.h"

Bike::Bike( int i_numberOfGears ):
  m_speed(0),
  m_revolutions(0),
  m_gearShift(i_numberOfGears) {
  std::cout << "bike with " << i_numberOfGears
            << " gears constructed" << std::endl;
};
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/bike/Bike.cpp`

Properties

- Called whenever an object is created
- More than one constructor (with different syntax) possible; no return type
- If none is specified, the compiler generates one
- Next slide: Some members, i.e. const members, have to be initialized in the **member initialization list**

Member Initialization List

Source

```
Bike::Bike( int i_numberOfGears ):
  m_speed(0),
  m_revolutions(0),
  m_gearShift(i_numberOfGears) {
```

code: https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/bike/Bike.cpp

Properties

- Per default the default constructor is called for all members
- Some member types have to be initialized in the member initialization list:
  - references
  - non-static const member
  - class members w\o default constructors
  - base class members
- Performance gains possible: Construction vs. assignment

Implementation: Destructor

Source

```
Bike::~Bike() {
  std::cout << "bike destroyed :(" << std::endl;
}
```

code: https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/bike/Bike.cpp

Properties

- Called whenever an object is destroyed
- Clean up of the object (could free dyn. allocated memory for example)
- Only destructor without arguments or return type
- If none is specified, the compiler generates one

Implementation: Member Functions

```
void Bike::changeGear( int i_gear ) {
  if( i_gear < m_gearShift.getCurrentGear() ) {
    // decrease gears until best possible gear is reached
    while( m_gearShift.decreaseGear() ) {
      if( i_gear == m_gearShift.getCurrentGear() ) break;
    }
  }
  else if( i_gear > m_gearShift.getCurrentGear() ) {
    // increase gearse until best possible gear is reached
    while( m_gearShift.increaseGear() ) {
      if( i_gear == m_gearShift.getCurrentGear() ) break;
    }
  }
}

void Bike::brake() {
  m_speed = std::max( 0.f, m_speed - 5.f );
  m_revolutions = m_speed * 25.f;
}

void Bike::changeCadence( float i_rpm ) {
  m_revolutions = std::max( 0.f, i_rpm );
  m_speed = m_revolutions * m_gearShift.getCurrentGear() / 25.f;
}
```

Implementation: Member Functions (2)

Source

```
void Bike::printInformation() const {
  std::cout << "speed: " << m_speed << std::endl
            << "rpms: " << m_revolutions << std::endl
            << "gear: " << m_gearShift.getCurrentGear()
            << std::endl;
}
```

Properties

Member functions can access and modify member variables; **const** as above forbids modification of member variables (details later and in the tutorials)

Runner: Dynamic Memory Allocation

Source

```cpp
#include "Bike.h"

int main() {
  // construct two bikes
  Bike l_myBike( 20 );
  Bike *l_myFriendsBike = new Bike( 15 );


  delete l_myFriendsBike;
}
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/bike/runner.cpp`

Properties

Dynamic memory allocation in `C++` should be done via the **new**-keyword: More typesafe and calls constructors
Hint: Always delete what you new and free what you malloc. Don't mix new with free or malloc with delete.

Runner: Using bikes

```
#include "Bike.h"

int main() {
  // construct two bikes
  Bike l_myBike( 20 );
  Bike *l_myFriendsBike = new Bike( 15 );

  // print their initial status
  l_myBike.printInformation();
  l_myFriendsBike->printInformation();

  // go on a bike tour with a friend
  l_myBike.changeGear( 16 );
  l_myBike.changeCadence( 16.5 );
  l_myFriendsBike->changeGear( 9 );
  l_myFriendsBike->changeCadence( 29.8 );

  // print information
  l_myBike.printInformation();
  l_myFriendsBike->printInformation();

  // free dynamic memory
  delete l_myFriendsBike;
}
```

Objects: Overview

Basics and Recap Getting a Solid Foundation

- Definition of an Object
- Relationships between Objects
- Pros and Cons

Objects in `C++` and C Syntax and Concepts

- Skeleton of a class with member variables, member functions and access control
- Implementation of the class: Constructor/destructor and member functions
- Runner: Using classes

Objects in `C++` continued Advanced Syntax and Concepts

- Copy Constructor
- Assignment Operator
- Static & Const Members

OO-oriented MD SoA with Dynamic Memory Allocation as Object

"Objects" in C

Bikes as Objects

- Task: Model a bike in C
- Goal: Understand the features `C++` brings to C and how `C++` is works under the hood.

Design

Gear Shift
- State: Current gear, #gears
- Behavior: Increase gear, decrease gear, get current gear

Bike
- State: Current Speed, revolutions per minute, Gear shift
- Behavior: Change gear, brake, change cadence

C-Skeleton: Header

Source

```c
#include <stdlib.h>
#include <math.h>
#include <stdio.h>

#include "GearShift.h"

typedef struct {
  float m_speed;
  float m_revolutions;
  GearShift m_gearShift;
} Bike;
```

code: https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/bike_c/Bike.h

Structs

Member variables can be stored in struct, however we are not able to have member functions → Need for an explicit **this**-Pointer (this is defined in a C++ -class as well).

C-Skeleton: Header (2)

Source

```
Bike *Bike_constructor( Bike *io_this, int i_numberOfGears );

void Bike_destructor( Bike *io_this, bool i_dynamic );

void Bike_changeGear( Bike *io_this, int i_gear );

void Bike_brake( Bike *io_this );

void Bike_changeCadence( Bike *io_this, float i_rpm );

void Bike_printInformation( Bike *io_this );
```

code: https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/bike_c/Bike.h

Description

Member variables can be stored in struct, however we are not able to have member functions → Need for an explicit **this**-Pointer (this is defined in a `C++` -class as well).

C-Implementation: Constructor

Source

```c
#include "Bike.h"

Bike *Bike_constructor( Bike *io_this, int i_numberOfGears ) {
  // check if memoy for (non-static) member variables is allocated
  if( io_this == NULL ) {
    // allocate memory
    io_this = (Bike*) malloc( sizeof(Bike) );
  }


  return io_this;
}
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/bike_c/Bike.c`

Description

In C we have to immitate the behavior of the **new** `C++` -keyword for dynamic memory allocation: A **NULL**-pointer results in a **malloc** for the underlying struct of the class.

C-Implementation: Constructor (2)

Source

```
Bike *Bike_constructor( Bike *io_this, int i_numberOfGears ) {


  // initialize member variables
  if( io_this ) {
    io_this->m_speed = 0;
    io_this->m_revolutions = 0;
    GearShift_constructor( &(io_this->m_gearShift), i_numberOfGears );
  }

  printf( "bike with %i gears constructed\n", i_numberOfGears );

  return io_this;
}
```

Description

As in `C++` after memory allocation the **member initialization** follows.
Note: In contrast to `C++` we have to call all constructors explicitly.

C-Implementation: Destructor

Source

```
void Bike_destructor( Bike *io_this, bool i_dynamic ) {
  if( i_dynamic ) {
    free( io_this );
  }

  printf( "bike destroyed :(\n" );
}
```

code: https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/bike_c/Bike.c

Description

In C we have to immitate the behavior of **delete**: We tell the destructor about dynamic memory allocation and **free** the memory.

C-Implementation: Member Functions

```c
void Bike_changeGear( Bike *io_this, int i_gear ) {
  if( i_gear < GearShift_getCurrentGear(&(io_this->m_gearShift)) ) {
    while( GearShift_decreaseGear(&(io_this->m_gearShift)) ) {
      if( i_gear == GearShift_getCurrentGear(&(io_this->m_gearShift)) ) break;
    }
  }
  else if( i_gear > GearShift_getCurrentGear(&(io_this->m_gearShift)) ) {
    while( GearShift_increaseGear(&(io_this->m_gearShift)) ) {
      if( i_gear == GearShift_getCurrentGear(&(io_this->m_gearShift)) ) break;
    }
  }
```

Description

Implementation of member functions is, besides **pointer arithmetics**, identical.

C-Implementation: Member Functions (2)

```
void Bike_brake( Bike *io_this ) {
  io_this->m_speed = fmaxf( 0.f, (io_this->m_speed - 5.f) );
  io_this->m_revolutions = io_this->m_speed * 25;
}

void Bike_changeCadence( Bike *io_this, float i_rpm ) {
  io_this->m_revolutions = fmaxf( 0.f, i_rpm );
  io_this->m_speed = io_this->m_revolutions * GearShift_getCurrentGear(&(io_this->m_gearShift)) / 25.f;
}

void Bike_printInformation( Bike *io_this ) {
  printf( "speed: %f\nrpms: %f\ngear: %i\n",
          io_this->m_speed,
          io_this->m_revolutions,
          GearShift_getCurrentGear(&(io_this->m_gearShift)) );
}
```

Description

Implementation of member functions is, besides **pointer arithmetics**, identical.

Runner: Using Bikes in C

```
int main() {
  // construct two bikes
  Bike l_myBike;
  Bike_constructor( &l_myBike, 20 );
  Bike* l_myFriendsBike = Bike_constructor( NULL, 15 );

  // print their initial status
  Bike_printInformation( &l_myBike );
  Bike_printInformation( l_myFriendsBike );

  // go on a bike tour with a friend
  Bike_changeGear( &l_myBike, 16 );
  Bike_changeCadence( &l_myBike, 16.5f );
  Bike_changeGear( l_myFriendsBike, 9 );
  Bike_changeCadence( l_myFriendsBike, 29.8f );

  // print information
  Bike_printInformation( &l_myBike );
  Bike_printInformation( l_myFriendsBike );

  // call destructors
  Bike_destructor( &l_myBike, false );
  Bike_destructor( l_myFriendsBike, true );
}
```

A First Overview: C and `C++`

- C is a part `C++` : You can always use **low-level C-functionality** (or even assembly) in `C++` , if necessary
- `C++` has a higher level of **abstraction** than C (on the way to a **high-level-language**)
- C follows a procedural paradigm, while `C++` is object-oriented
- `C++` has a stronger **type system** than C
- The object-oriented approach of `C++` allows for **access control**
- to be extended [...]

Objects: Overview

Basics and Recap Getting a Solid Foundation

- Definition of an Object
- Relationships between Objects
- Pros and Cons

Objects in `C++` and C Syntax and Concepts

- Skeleton of a class with member variables, member functions and access control
- Implementation of the class: Constructor/destructor and member functions
- Runner: Using classes

Objects in `C++` continued Advanced Syntax and Concepts

- Copy Constructor
- Assignment Operator
- Static & Const Members

OO-oriented MD SoA with Dynamic Memory Allocation as Object

Implicit Copy Constructor

Source

```
class SimpleClass {
  // private:
    int m_a;
    std::string m_b;

  public:
    void setValues( int i_a, const std::string &i_b ) { m_a=i_a; m_b=i_b; };
    void printValues() { std::cout << m_a << ", " << m_b << std::endl; };
};

int main() {
  SimpleClass l_simple1;
  l_simple1.setValues( 1, "simple1" );
  l_simple1.printValues();

  // call copy constructor
  SimpleClass l_simple2( l_simple1 );
  l_simple2.printValues();
}
```

code: https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/copy_constructor/implicit.cpp

Implicit Copy Constructor (2)

Source

```
class SimpleClass {
  // private:
    int m_a;
    std::string m_b;

  public:
    void setValues( int i_a, const std::string &i_b ) { m_a=i_a; m_b=i_b; };
    void printValues() { std::cout << m_a << ", " << m_b << std::endl; };
};
```

code: https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/copy_constructor/implicit.cpp

Description

The compiler generates an implicit **copy constructor**, which is used when a new object is created from another and provides a member-wise copy of the source object

Explicit Copy Constructor

```
class SimpleClass {
  // private:
    int m_a;
    std::string m_b;

  public:
    SimpleClass(){};

    SimpleClass( const SimpleClass &i_src ):
      m_a( i_src.m_a ),
      m_b( i_src.m_b ) {
      std::cout << "copy␣constructor␣called" << std::endl;
    };

};
```

code: https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/copy_constructor/explicit.cpp

Description

As already seen for the constructor and destructor, a manual implementation is possible and sometimes necessary; besides the example above different signatures are possible

Implicit Assignment Operator

Source

```
class SimpleClass {
  // private:
    int m_a;
    std::string m_b;

  public:
    void setValues( int i_a, const std::string &i_b ) { m_a=i_a; m_b=i_b; };
    void printValues() { std::cout << m_a << ", " << m_b << std::endl; };
};

int main() {
  SimpleClass l_simple1, l_simple2;
  l_simple1.setValues( 1, "simple1" );
  l_simple1.printValues();

  // call the assignment operator
  l_simple2 = l_simple1;
  l_simple2.printValues();
}
```

code: https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/assignment_operator/implicit.cpp

Implicit Assignment Operator

Source

```cpp
class SimpleClass {
  // private:
    int m_a;
    std::string m_b;

  public:
    void setValues( int i_a, const std::string &i_b ) { m_a=i_a; m_b=i_b; };
    void printValues() { std::cout << m_a << ", " << m_b << std::endl; };
};
```

code: https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/assignment_operator/implicit.cpp

Description

An implicit **assignment operator** is generated by the compiler, which does a member-wise assignment of each source member

Explicit Assignment Operator

Source

```
class SimpleClass {
  // private:
    int m_a;
    std::string m_b;

  public:
    SimpleClass(){};

    SimpleClass &operator=( const SimpleClass &i_src ) {
      m_a = i_src.m_a;
      m_b = i_src.m_b;
      std::cout << "assignment operator called" << std::endl;
      return *this;
    };

  SimpleClass l_simple3( l_simple1 = l_simple2 );
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/assignment_operator/explicit.cpp`

Description

Again a manual implementation is possible and sometimes required; different signatures are possible

Static Members

```
class SimpleClass {
  // private:
    static int s_simpleCounter;

  public:
    SimpleClass() {
      s_simpleCounter++;
      std::cout << s_simpleCounter << std::endl;
    };

    ~SimpleClass() {
      s_simpleCounter--;
      std::cout << s_simpleCounter << std::endl;
    };
};

int SimpleClass::s_simpleCounter;

int main() {
  SimpleClass l_simple1;
  {
    SimpleClass l_simple2;
  }
  SimpleClass l_simple3;
}
```

Static Members (2)

Source

```
class SimpleClass {
  // private:
    static int s_simpleCounter;

  public:
    SimpleClass() {
      s_simpleCounter++;
      std::cout << s_simpleCounter << std::endl;
    };

    ~SimpleClass() {
      s_simpleCounter--;
      std::cout << s_simpleCounter << std::endl;
    };
};

int SimpleClass::s_simpleCounter;
```

code: https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/static_member_variable.cpp

Description

A class can have **static member variables**, which are single locations in memory shared by all instances of the class

Static Member Functions

```cpp
class SimpleClass {
  // private:
    static int s_simpleCounter;

  public:
    static void increaseCounter(){ s_simpleCounter++; };
    static void printCounter(){ std::cout << s_simpleCounter << std::endl; };
};

int SimpleClass::s_simpleCounter = 1;

int main() {
  SimpleClass::printCounter();
  SimpleClass::increaseCounter();
  SimpleClass::printCounter();
}
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/static_member_function.cpp`

Description

Static member functions are accessible independently of a single instance of a class; Consequently non-static members or the this-pointer can't be accessed

Const Members

Source

```
class SimpleClass {
  // private:
    const int m_a;

  public:
    SimpleClass( int i_a ): m_a( i_a ) {
      // forbidden assignment
      // m_a = 10;
    };
};

int main() {
  SimpleClass l_simple( 15 );
}
```

code: https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/const_member_variable.cpp

Description

A class can have **const member variables**, which can't be changed after initialization; thus have to be set in the member initialization list.

Const Member Functions

Source

```
class SimpleClass {
  // private:
    int m_a;

  public:
    void simpleFunctions() const {
      // forbidden assignment
      // m_a = 5;
    }
};

int main() {
  SimpleClass l_simple;
  l_simple.simpleFunctions();
}
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/const_member_function.cpp`

Description

A class can have **const member functions**, which are not allowed to change the member variables/state of an object.
Hint: The position of the const qualifier is crucial (→ tutorials).

Objects: Overview

Basics and Recap Getting a Solid Foundation

- Definition of an Object
- Relationships between Objects
- Pros and Cons

Objects in `C++` and C Syntax and Concepts

- Skeleton of a class with member variables, member functions and access control
- Implementation of the class: Constructor/destructor and member functions
- Runner: Using classes

Objects in `C++` continued Advanced Syntax and Concepts

- Copy Constructor
- Assignment Operator
- Static & Const Members

OO-oriented MD SoA with Dynamic Memory Allocation as Object

OO-MD: Header

```
#ifndef MOLECULES_H_

#include <cassert>
#include <iostream>

/**
 * Collection of Molecules
 **/
class Molecules {
  // private:
    //! number of molecules under control of this collection
    const int m_numberOfMolecules;

    //! time step width deltaT
    const double m_deltaT;

    //! positions in x-, y- and z-direcion
    double *m_x, *m_y, *m_z;


};

#endif
```

code: https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/md_collection/Molecules.h

OO-MD: Header(2)

```
class Molecules {
  // private:

    // velocities in x-, y- and z-direction
    double *m_velocityX, *m_velocityY, *m_velocityZ;

  public:
    /**
     * Constructor, which allocates memory for all molecules.
     *
     * @param i_numberOfMolecules number of molecules under control of the collection.
     */
    Molecules( int    i_numberOfMolecules,
               double i_deltaT );

    /**
     * Destructor, which frees the allocated memory.
     **/
    ~Molecules();

};
```

code: https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/md_collection/Molecules.h

OO-MD: Header(3)

```
class Molecules {

  public:

    /**
     * Sets the the initial values of the molecule with the given ID.
     *
     * @param i_moleculeId id of the molecule.
     * @param i_initialValues initial values:
     *           0-2: position in x-, y- and z-direction
     *           3-5: velocity in x-, y- and z-direction
     **/
    void setInitialValues(       int    i_moleculeId,
                           const double i_values[6] );

    /**
     * Computes the next time step.
     **/
    void computeTimeStep();

};
```

code: https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/md_collection/Molecules.h

OO-MD: Header(4)

```
class Molecules {

  public:

    /**
     * Prints the values of the molecule with the given ID.
     *
     * @param i_moleculeId id of the molecule.
     **/
    void printValues( int i_moleculeId ) const;
};
```

code: https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/md_collection/Molecules.h

OO-MD: Implementation

```
#include "Molecules.h"

Molecules::Molecules( int    i_numberOfMolecules,
                      double i_deltaT ):
  m_numberOfMolecules( i_numberOfMolecules ),
  m_deltaT( i_deltaT ) {
  std::cout << "allocating memory for "
            << i_numberOfMolecules
            << " molecules" << std::endl;

  // allocate memory for the molecules
  m_x = new double[i_numberOfMolecules];
  m_y = new double[i_numberOfMolecules];
  m_z = new double[i_numberOfMolecules];
  m_velocityX = new double[i_numberOfMolecules];
  m_velocityY = new double[i_numberOfMolecules];
  m_velocityZ = new double[i_numberOfMolecules];
}

Molecules::~Molecules() {
  std::cout << "freeing Memory" << std::endl;

  delete[] m_x,         m_y,         m_z,
           m_velocityX, m_velocityY, m_velocityZ;
}
```

OO-MD: Implementation (2)

```
void Molecules::setInitialValues(       int    i_moleculeId,
                                  const double i_values[6] ) {
  // assert that the id is in bound
  assert( i_moleculeId < m_numberOfMolecules );

  m_x[i_moleculeId] = i_values[0];
  m_y[i_moleculeId] = i_values[1];
  m_z[i_moleculeId] = i_values[2];

  m_velocityX[i_moleculeId] = i_values[3];
  m_velocityY[i_moleculeId] = i_values[4];
  m_velocityZ[i_moleculeId] = i_values[5];
}

void Molecules::computeTimeStep() {
  // iterate over all molecules of the collection
  for( int l_moleculeId = 0; l_moleculeId < m_numberOfMolecules; l_moleculeId++ ) {
    m_x[l_moleculeId] += m_deltaT * m_velocityX[l_moleculeId];
    m_y[l_moleculeId] += m_deltaT * m_velocityY[l_moleculeId];
    m_z[l_moleculeId] += m_deltaT * m_velocityZ[l_moleculeId];
  }
  // usually here would be same velocity computation based on molecular forces
}
```

OO-MD: Implementation (3)

```cpp
void Molecules::printValues( int i_moleculeId ) const {
  // assert the id is in bound
  assert( i_moleculeId < m_numberOfMolecules );

  std::cout << "x: "  << m_x[i_moleculeId] << std::endl
            << "y: "  << m_y[i_moleculeId] << std::endl
            << "z: "  << m_z[i_moleculeId] << std::endl
            << "vX: " << m_velocityX[i_moleculeId] << std::endl
```

code: https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/md_collection/Molecules.cpp

OO-MD: Runner

```cpp
#include "Molecules.h"

int main() {
  // initial dummy values
  double l_initialMolecule[6] = { 0.1, 0.2, 0.3,
                                  0.4, 0.5, 0.6 };
  int l_numberOfMolecules = 5000;
  double l_deltaT = 0.0001;

  std::cout << "creating a new molecules collection"
            << std::endl;
  Molecules l_molecules( 5000, 0.01 );

  std::cout << "occupied size: " << sizeof(l_molecules)
            << std::endl;

  std::cout << "initializing molecules"
            << std::endl;
  for( int l_moleculeId = 0; l_moleculeId < l_numberOfMolecules; l_moleculeId++ ) {
    l_molecules.setInitialValues( l_moleculeId,
                                  l_initialMolecule );
  }
```

code: https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/md_collection/md_runner.cpp

OO-MD: Runner (2)

```
  std::cout << "occupied␣size:␣" << sizeof(l_molecules)
            << std::endl;

  std::cout << "iterating␣over␣time"
            << std::endl;
  for( double l_time = 0.0; l_time < 0.5; l_time += l_deltaT ) {
    l_molecules.computeTimeStep();
  }

  std::cout << "finished." << std::endl;
}
```

code: https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/md_collection/md_runner.cpp

Voluntary Homework: Autovectorization

```
..B5.3:                          # Preds ..B5.1 ..B5.3
mov       rdx, QWORD PTR [40+rdi]                        #60.37
vmovsd    xmm0, QWORD PTR [8+rdi]                        #60.26
mov       rcx, QWORD PTR [16+rdi]                        #60.5
vmulsd    xmm1, xmm0, QWORD PTR [rdx+rax*8]              #60.37
vaddsd    xmm2, xmm1, QWORD PTR [rcx+rax*8]              #60.5
vmovsd    QWORD PTR [rcx+rax*8], xmm2                    #60.5
mov       rsi, QWORD PTR [48+rdi]                        #61.37
vmovsd    xmm3, QWORD PTR [8+rdi]                        #61.26
mov       r8, QWORD PTR [24+rdi]                         #61.5
vmulsd    xmm4, xmm3, QWORD PTR [rsi+rax*8]              #61.37
vaddsd    xmm5, xmm4, QWORD PTR [r8+rax*8]               #61.5
vmovsd    QWORD PTR [r8+rax*8], xmm5                     #61.5
mov       r9, QWORD PTR [56+rdi]                         #62.37
vmovsd    xmm6, QWORD PTR [8+rdi]                        #62.26
mov       r10, QWORD PTR [32+rdi]                        #62.5
vmulsd    xmm7, xmm6, QWORD PTR [r9+rax*8]               #62.37
vaddsd    xmm8, xmm7, QWORD PTR [r10+rax*8]              #62.5
vmovsd    QWORD PTR [r10+rax*8], xmm8                    #62.5
inc       rax                                            #59.66
movsxd    r11, DWORD PTR [rdi]                           #59.45
cmp       rax, r11                                       #59.45
jl        ..B5.3        # Prob 82%                       #59.45
```

code: https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/md_collection/Molecules_mod.s

Voluntary Homework: Autovectorization (2)

```
Molecules.cpp(28): (col. 39) remark: vectorization support: call to function _Znam cannot be vectorized.
Molecules.cpp(28): (col. 3) remark: loop was not vectorized: existence of vector dependence.
Molecules.cpp(19): (col. 12) remark: vectorization support: call to function _Znam cannot be vectorized.
Molecules.cpp(19): (col. 12) remark: loop was not vectorized: existence of vector dependence.
Molecules.cpp(57): (col. 35) remark: routine skipped: no vectorization candidates.
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/md_collection/vec_report_intel.log`

Task

Can you convince a compiler of your choice to autovectorize computeTimeStep()?

Solutions until 01/16/14 to breuera@in.tum.de

Assertions

```
void Molecules::setInitialValues(       int    i_moleculeId,
                                  const double i_values[6] ) {
  // assert that the id is in bound
  assert( i_moleculeId < m_numberOfMolecules );

  m_x[i_moleculeId] = i_values[0];
  m_y[i_moleculeId] = i_values[1];
  m_z[i_moleculeId] = i_values[2];

  m_velocityX[i_moleculeId] = i_values[3];
  m_velocityY[i_moleculeId] = i_values[4];
  m_velocityZ[i_moleculeId] = i_values[5];
}
```

code: `https://github.com/TUM-I5/advanced_programming/tree/master/lectures/objects/md_collection/Molecules.cpp`

Properties

- If the expression is false: Message to standard error and abort
- Assertions are disabled if the pre-processor macro **NDEBUG** is defined → Helps to find programming errors
- Use assertions whenever applicable, no performance impacts in release mode (NDEBUG defined)