Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

Classes in C++

Michael Hanke

School of Engineering Sciences

Program construction in C++ for Scientific Computing

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

Outline

1 Classes

2 Constructors and Destructors

3 Summary

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

What is a Class?

- An abstract data type is a (nonempty, finite) set and the collection of operations defined on this set.
- A C++ class is the programmatic description of a data type.
- An object is an instance of a class.

An abstract data type is a suitable model for implementing abstract mathematical structures.

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

Formal Class Declaration

C++ class declaration

```
class identifier {
public:
  // Public class members
protected:
  // Protected class members
private:
  // Private class members
}; // Do not forget the semicolon here!!
```

where identifier is the name of the class.

- The members can be basic data types, other classes or functions.
- Public members can be accesses from anywhere in the program.
- Private members can only be accessed from member functions of the class.
- Protected members can be accessed from derived classes additionally to member functions of the class.

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

Class Declaration (cont)

- Instead of class, the reserved word struct can be used. The difference lies in the default access behavior.
- Default access behavior: class = private; struct = public.
- *Convention*: Names of classes start usually with a capital letter.

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

A Simple Class

- The mathematical notion: *Points in the two-dimensional Cartesian plane*
- The implementation of this mathematical notion should look to the user as if it were a standard type.
- The user of the class does not need to know how the internals look like.
- *Example*: The user should be able to write something like

```
Point P;
Point W(1.0,2.0);
Point Q = P;
```

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

A C-Style Implementation

```
class Point {
  public:
    double x;
    double y;
};
```

Note: The keyword class can be replaced by struct. The latter is the way one would do it in C.

- The coordinates can be accessed via P.x and P.y using explicitly the implementation.
- *What if we instead would use polar coordinates in the implementation?* The user must rewrite his/her code!

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

A C++-Style Implementation

```cpp
class Point {
  private:       // Can be omitted here
    double x;
    double y;
  public:
    double X() { return x; }  // return x coordinate
    double Y() { return y; }  // return y coordinate
    void zero() { x = y = 0.0; } // set point to origin
};
```

The user can access the Cartesian coordinates via P.X() and P.Y(),
respectively.

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

Another Implementation

```
class Point {
  private:
    double r;
    double phi;
  public:
    double X() { return r*std::cos(phi); }
    double Y() { return r*std::sin(phi); }
    void zero() { r = phi = 0.0; }
};
```

The user interface did not change!

- The variables r, phi are called data members of the class.
- The functions X, Y, zero are the member functions of the class.

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

Programming Style: Separation of Interface and Implementation

The interface file point.hpp may look like this:

```cpp
#ifndef POINT_HPP
#define POINT_HPP

class Point {
    double x;
    double y;
  public:
    double X();
    double Y();
    void zero();
};

#endif
```

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

Implementation

```
#include ''point.hpp''

double Point::X() {
  return x;
}
double Point::Y() {
  return y;
}
void Point::zero() {
  x = y = 0.0;
}
```

The user of the class will most probably never see the implementation!

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

Efficiency Considerations: Inlining

- The principle of data hiding leads often to very many small member functions.
- Calling a function includes an overhead compared with the simple data member access (e.g., P.x).
- The overhead can lead to low efficiency if calls happen rather often (inside innermost loops).
- This overhead can be avoided by function inlining.
- Note: Inlining is a hint to the compiler. The compiler can do it or not.
- Function bodies defined in header files are inlined be default, while functions defined in the implementation are not. (*Guess why?*)

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

Efficiency Considerations: const

- A compiler can often optimize the code much better if it can use additonal assumptions about the function behavior.
- One important property is if certain objects are constant.
- Example: In the definition

      const int N = 10;

  the variable N will never change its value. Doing so will result in a compilation error.
- As a byproduct, the user interface may become safer.

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

const And Pointers

- Consider the definition

    const double *p;

- This construct indicates that the double the pointer p *is pointing to* will never change its value.

- Consider instead

    double *const p = &q;

- Here, *the pointer* p will never change its value.

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

Efficiency Considerations: point
Class

For efficiency, the header file should look like this:

```
#ifndef POINT_HPP
#define POINT_HPP
class Point {
  private:      // Can be omitted here
    double x;
    double y;
  public:
    double X() const { return x; }
    double Y() const { return y; }
    void zero() { x = y = 0.0; }
};
#endif
```

The keyword const indicates that the object will not change its state
when queuried for the coordinates.

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

Constructors

- Constructors determine what happens if an instance of a class (an object) is created.
- Built-in data types have default constructors: E.g., a statement `int i;` reserves memory for one instance of type integer.
- The initial value of an instance of a built-in type is undefined!
- A definition of the type `int i = 0;` invokes another type of constructor, the so-called copy constructor.
- A definition of the kind *class variable*; invokes a constructor

    class::class()

  as a member function of the instance *variable*. (the so-called default constructor)

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

Constructors (cont)

- If no constructors are defined in a class, the so-called *synthesized default constructor* is automatically defined by the compiler.
- The synthesized default constructor invokes recursively the default constructors of the data members.
- As soon as at least one constructor is defined in the class, the default constructor is not available (unless it is explicitly required by class() = default;)
- Be careful: The synthesized default constructor might not be what you want! (Shallow vs deep copy)

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

`point` Class Constructors

- We want something like

    ```
    Point();
    Point(double xx, double yy);
    ```

- The default constructor is "do nothing but reserve memory":

    ```
    Point() {}
    ```

- The next one seems also easy:

    ```
    Point(double xx, double yy) { x = xx; y = yy; }
    ```

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

point Class Constructors (cont)

- A more efficient way: Use initialization lists:

      Point(double xx, double yy) : x(xx), y(yy) { }

  (uses the copy constructors)

- And finally: A versatile version (even replacing the default constructor):

      Point(double xx = 0.0, double yy = 0.0) :
                  x(xx), y(yy) { }

- Now, we can define:

      Point P(3.0,5.0);
      Point Q(3.0);
      Point W;

  but even:

      Point *p; p = new Point(2.0);

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

Constructors: Intialization Lists

We *must* use the constructor initializer list to provide values for members that are const, reference, or of class type that does not have a default constructor.
Example:

```
class ConstRef {
  public:
    ConstRef(int ii);
  private:
    int i;
    const int ci;
    int &ri;
};
```

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

Initialization Lists (cont)

Correct

```
ConstRef::ConstRef(int ii): i(ii), ci(ii), ri(i) { }
```

Errorneous
```
ConstRef::ConstRef(int ii) {
  i = ii;  // ok
  ci = ii; // wrong since ci is const
  ri = i;  // wrong: ri was never initialized
}
```

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

The Copy Constructor

- *Aim*: Initialize an instance of a class by another instance of the same class:

  ```
  Point P(3.0,5.0);
  Point Q(P);
  Point W = Q;
  ```

- The creation of the objects Q and W are handled by the *copy constructor*.

- The copy constructor is invoked when
  - objects are defined by = or class(object of that class)
  - objects are passed as actual parameters for non-reference arguments
  - return object from a function that has a non-reference return type.

- This explains why the argument must be of reference type! (*Why?*)

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

The Default Copy Constructor

- The default copy constructor invokes the copy constructors of all data members.
- For built-in types, this is a simple copy.
- In our example, it is equivalent to:

      Point(const Point& Q): x(Q.x), y(Q.y) { }

  Note: This is not identical to

      Point(const Point& Q) {x = Q.x; y = Q.y; }

  Why?
- If the class manages its own dynamic memory (e.g. using new type[n]), one must most probably define its own copy constructor!
- Discussion: Should one define one's own copy constructor?

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

Remark

- In the following, Q is constructed via the copy constructor:

      Point P(3.0,5.0);
      Point Q = P;

- Compare:

      Point P(3.0,5.0), Q;
      Q = P;

  This case is handled by the *copy-assignment* constructor! This is different from the previous one!

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

Copy Constructor: Efficiency

Consider the following ordinary (non-member) function:

```
const Point negative(const Point P) {
  return Point(-P.X(),-P.Y());
}
```

- This version is very expensive, since it uses the constructor 3 times!

It's demo time!

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

Efficiency (cont)

- Better:

    ```
    const Point negative(const Point& P) {
      return Point(-P.X(),-P.Y());
    }
    ```

- Note: The return type cannot be const Point&! *Why?*

- The C++11 and later standards have means to avoid certain copies of temporary objects (move and move-assignment constructors).

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

The Destructor

- Inverse operation of constructors.
- Destructors do whatever work is needed to free the resources used by an object.
- The destructor is a member function with empty argument list with the name of the class prefixed by a tilde:

    ```
    ~Point() { }
    ```

    In our simple example, it is a no-op. The runtime system releases the memory.

- *In general, releasing resources must be handled very carefully in order to avoid memory leaks etc!*

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

Type Conversion

- What happens in the following situation?

      double d = 1;

  The constant "1" is int, the variable defined of type double.

- The integer constant is implicitely converted to type double (1.0) and then assigned.

- In case of the definition

      Point P = 1.0;

  the constructor Point(1.0) is invoked.

- *This way, the constructor includes an implicit type conversion!*

- Note: Explicit type conversion ("*type casting*") is included in this mechanism:

      Point P; P = (Point) 1.0;

  Be careful! Avoid explicit type casting!

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

static Class Members

- Any member of a class can be static.
- A static member exists only once for each class. Thus, it is not bound to a concrete object.
- A static member function does not contain a this pointer. It can only use static class members.
- Definition of a static member outside of a class body: Omit the static keyword.
- Static data members must be initialized outside the class (No constructor will be called!)
- constexpr static data members will be initialized in the class definition.

Introduction

Michael
Hanke

Classes

Constructors
and
Destructors

Summary

Summary

What we learned:

- Basic definitions of classes
- Private and public members
- Constructors and destructors
- Constructors: Efficiency considerations

- What comes next:
  - Operator overloading