

An Introduction to C++

Part 4

Introduction to classes

Classes

- ◆ Are a method to create new data types
 - ◆ E.g. a vector or matrix type
- ◆ Object oriented programming:
 - ◆ Instead of asking: “What are the subroutines?”
 - ◆ We ask:
 - ◆ What are the abstract entities?
 - ◆ What are the properties of these entities?
 - ◆ How can they be manipulated?
 - ◆ Then we implement these entities as classes
- ◆ Advantages:
 - ◆ High level of abstraction possible
 - ◆ Hiding of representation dependent details
 - ◆ Presentation of an abstract and simple interface
 - ◆ Encapsulation of all operations on a type within a class
 - ◆ allows easier debugging

A first simulation: biological aging

- ◆ a simple model for death: $dN = -\lambda N dt$
- ◆ what about aging?
 - ◆ the remaining lifetime does not depend on current age.
 - ◆ true for radioactive decay
 - ◆ not true for biology
- ◆ what about age distribution?
 - ◆ exponential distribution!
 - ◆ also not what is seen in nature!
- ◆ what is missing?
 - ◆ some kind of aging
 - ◆ we need to develop a model containing aging

The Penna model

- ◆ A very simple model of biological aging
 - ◆ T.J.P. Penna, J. Stat. Phys **78**, 1629 (1995)
- ◆ Three important assumptions
 - ◆ finite age of adulthood
 - ◆ mutations of genetic material
 - ◆ limited resources
- ◆ This allows to model many features of biological population dynamics:
 - ◆ pacific salmon dies after giving birth
 - ◆ redwood trees have offsprings for hundreds of generations
 - ◆ catastrophic decline of cod in Atlantic due to small increase in fishing
- ◆ All these issues cannot be modeled without aging effects!

Details of the Penna model

- ◆ Each animal contains genes determining the survival rate
 - ◆ Each gene relevant for one year of its life
 - ◆ animal dies when it has collected T bad genes
- ◆ Limitation of resources
 - ◆ An animal that would survive because of its genes dies with a probability of N/N_0
 - ◆ N ...current population
 - ◆ N_0 ... maximum sustainable population
- ◆ Children
 - ◆ from an age of R years an animal gets a child asexually with a probability b (birthrate)
- ◆ Mutations
 - ◆ The children have the genes of the parents but with M random mutations

Program for the Penna model

- ◆ First step: find the entities
- ◆ What are the abstract ideas?
 - ◆ Genes
 - ◆ Animal
 - ◆ Population
- ◆ Exercise: write a list of the properties of each of these entities
 - ◆ Representation (internal state)
 - ◆ Properties
 - ◆ Operations
 - ◆ Construction/destruction

What are classes?

- ◆ Classes are collections of “members”, which can be
 - ◆ functions
 - ◆ data
 - ◆ types
- ◆ representing one entity
- ◆ These members can be split into
 - ◆ **public**, accessible interface to the outside
 - ◆ should not be modified later!
 - ◆ **private**, hidden representation of the concept
 - ◆ can be changed without breaking any program using the class
 - ◆ this is called “data hiding”
- ◆ Objects of this type can be modified only by these member functions -> easier debugging

How to design classes

- ◆ ask yourself some questions
- ◆ what are the logical entities (**nouns**)?
 - ◆ ® classes
- ◆ what are the internal state variables ?
 - ◆ ® private data members
- ◆ how will it be created/initialized and destroyed?
 - ◆ ® constructor and destructor
- ◆ what are its properties (**adjectives**)?
 - ◆ ® public constant member functions
- ◆ how can it be manipulated (**verbs**)?
 - ◆ ® public operators and member functions

A first class example: a traffic light

- ◆ Property
 - ◆ The state of the traffic light (green, orange or red)
- ◆ Operation
 - ◆ Set the state
- ◆ Construction
 - ◆ Create a light in a default state (e.g. red)
 - ◆ Create a light in a given state
- ◆ Destruction
 - ◆ Nothing special needs to be done
- ◆ Internal representation
 - ◆ Store the state in a variable
 - ◆ Alternative: connect via a network to a real traffic light

A first class example: a traffic light

- ◆ Converting the design into a class

```
class Trafficlight {  
};
```

A first class example: a traffic light

- ◆ Add a public type member

```
class Trafficlight {
    public: // access declaration
        enum light { green, orange, red}; // type member

};
```

A first class example: a traffic light

- ◆ Add a private data member (variable) of that type:

```
class Trafficlight {
    public: // access declaration
        enum light { green, orange, red}; // type member

    private: // this is hidden
        light state_; // data member
};
```

A first class example: a traffic light

- ◆ Add a const member function to access the state

```
class Trafficlight {
    public: // access declaration
        enum light { green, orange, red}; // type member

        light state() const; //function member

    private: // this is hidden
        light state_; // data member
};
```

A first class example: a traffic light

- ◆ Add a non-const member function to change the state

```
class Trafficlight {
    public: // access declaration
        enum light { green, orange, red}; // type member

        light state() const; //function member
        void set_state(light);

    private: // this is hidden
        light state_; // data member
};
```

A first class example: a traffic light

- ◆ Add a default constructor to initialize it in the default way
a constructor has the same name as the class

```
class Trafficlight {
    public: // access declaration
        enum light { green, orange, red}; // type member
        Trafficlight(); // default constructor

        light state() const; //function member
        void set_state(light);
    private: // this is hidden
        light state_; // data member
};
```

A first class example: a traffic light

- ◆ Add a second constructor to construct it from a light

```
class Trafficlight {
    public: // access declaration
        enum light { green, orange, red}; // type member
        Trafficlight(); // default constructor
        Trafficlight(light); // constructor

        light state() const; //function member
        void set_state(light);
    private: // this is hidden
        light state_; // data member
};
```


A first class example: a traffic light

- ◆ And finish by adding a destructor (called to cleanup at destruction) a destructor has the same name as the class, prefixed by ~

```
class Trafficlight {
    public: // access declaration
        enum light { green, orange, red}; // type member
        Trafficlight(); // default constructor
        Trafficlight(light); // constructor
        ~Trafficlight(); // destructor

        light state() const; //function member
        void set_state(light);
    private: // this is hidden
        light state_; // data member
};
```

Data hiding and access

- ◆ The concept expressed through the class is **representation - independent**
- ◆ Programs using a class should thus also not depend on representation
- ◆ Access declarators
 - ◆ **public**: only representation-independent interface, accessible to all
 - ◆ **private**: representation-dependent functions and data members
 - ◆ **friend** declarators allow related classes access to representation
- ◆ Note: Since all data members are representations of concepts (numbers, etc.) they should be hidden (private)!
- ◆ By default all members are private
In a **struct** by default all are public

Member access

```
class Trafficlight {
public:
    enum light
    { green, orange, red};

    Trafficlight();
    Trafficlight(light);
    ~Trafficlight();

    light state() const;
    void set_state(light);
private:
    light _state;
};
```

◆ Usage:

```
Trafficlight
x(Trafficlight::green);
Trafficlight::light l;

l = x.state();
l = Trafficlight::green;
```

◆ Members accessed with

variable_name.member_name

◆ Type members accessed with

class_name::member_name

as they are not bound to specific object but common to all.

Special members

◆ Constructors

- ◆ initialize an object
- ◆ same name as class

◆ Destructors

- ◆ do any necessary cleanup work when object is destroyed
- ◆ have the class name prefixed by ~

◆ Conversions

◆ Operators

Illustration: a point in two dimensions

- | | |
|---|---|
| <ul style="list-style-type: none"> ◆ Internal state: <ul style="list-style-type: none"> ◆ x- and y- coordinates ◆ is one possible representation ◆ Construction <ul style="list-style-type: none"> ◆ default: (0,0) ◆ from x- and y- values ◆ same as another point ◆ Properties: <ul style="list-style-type: none"> ◆ distance to another point ◆ x- and y- coordinates ◆ polar coordinates ◆ Operations <ul style="list-style-type: none"> ◆ Inverting a point ◆ assignment | <pre> class Point { private: float x_,y_; public: Point(); // (0,0) Point(float, float); Point(const Point&); float dist(const Point&) const; float x() const; float y() const; float abs() const; float angle() const; void invert(); Point& operator=(const Point&); }; </pre> |
|---|---|

Constructors and Destructors

- ◆ Let us look at the point example:
 - ◆ `public:`

```

        Point(); // default constructor
        Point(float, float); // constructor from two numbers
        Point(const Point&); // copy constructor

```
- ◆ Most classes should provide a default constructor
- ◆ Copy constructor automatically generated as memberwise copy, unless otherwise specified
- ◆ Destructor normally empty and automatically generated
- ◆ Nontrivial destructor only if resources (memory) allocated by the object. This usually also requires nontrivial copy constructor and assignment operator. (example: array class)

Default members

- ◆ Some member functions are implicitly created by the compiler
 - ◆ Copy constructor


```
A::A(A const&);
```

 defaults to member-wise copy if not specified
 - ◆ Assignment operator


```
A::operator=(A const&);
```

 also defaults to member-wise copy
 - ◆ Destructor


```
A::~~A()
```

 defaults to empty function

Declaration, Definition and Implementation

◆ Declaration

```
class Point;
```

◆ Definition

```
class Point {
private:
    float x_,y_;
public:
    Point(); // (0,0)
    Point(float, float);
    ...
};
```

◆ Implementation

```
float Point::abs() const {
    return std::sqrt(x_*x_+y_*y_);
}
```

◆ Constructors

```
Point::Point(float x, float y)
    : x_(x), y_(y)
{} // preferred method
```

◆ or

```
Point::Point(float x, float y)
{ x_ = x; y_ = y;}
```

Initializing a reference or a const member

- ◆ The simple-minded way fails

```
class A {
private:
    int& x;
    const int y;
public:
    A(int& r, int s) {
        x=r; // does not work
            // what does x refer to ?
        y=s; // does not compile
            // y is const!
    }
};
```

- ◆ We need the initialization syntax

```
class A {
private:
    int& x;
    const int y;
public:
    A(int& r, int s)
        : x(r),
          y(s)
    {
    }
};
```

- ◆ Stylistic advice: initialize all members in this way

const and volatile

- ◆ **const**

- ◆ Variables or data members declared as `const` cannot be modified
- ◆ Member functions declared as `const` do not modify the object
- ◆ Only `const` member functions can be called for `const` objects

- ◆ **volatile**

- ◆ Volatile variables
 - `volatile int x;`
 - can be modified from outside the program!
- ◆ Examples: multi-threaded programs and I/O ports
- ◆ No optimization or caching allowed!
- ◆ Only member functions declared `volatile` can be called for `volatile` objects

mutable

◆ Problem:

- ◆ want to count number of calls to `age()` function of `Animal`

◆ Original source:

```
class Animal() {
public:
    age_t age() const;
private:
    long    cnt_;
    age_t age_;
};

age_t Animal::age() const {
    cnt_++; // error: const!
    return age_;
}
```

◆ Solution:

- ◆ mutable qualifier allows modification of member even in `const` object!

◆ Modified source:

```
class Animal() {
    ...
private:
    mutable long cnt_;
    ...
};

age_t Animal::age() const {
    cnt_++; // now OK!
    return age_;
}
```

friends

◆ Consider geometrical objects: points, vectors, lines,...

```
◆ class Point {
    ...
private:
    float x,y,z;
};
```

```
class Vector {
    ...
private:
    float x,y,z;
};
```

- ◆ For an efficient implementation these classes should have access to each others internal representation

◆ Using friend declaration this is possible:

```
◆ class Vector;
class Point {
    ...
private:
    float x,y,z;
    friend class Vector;
};
class Vector {
    ...
private:
    float x,y,z;
    friend class Point;
};
```

◆ also functions possible:

```
◆ friend Point::invert(...);
```

this

- ◆ Sometimes the object needs to be accessed from a member function

- ◆ `this` is a pointer to the object itself:

- ◆

```
const Array& Array::operator=(const Array& o) {
    if(this!=&o) {// do nothing if x=x;
        // do assignment
        ...
    }
    return *this;
}
```

Inlining of member functions

- ◆ For speed issues member functions can be inlined
- ◆ Avoid excessive inlining as it leads to code-bloat

- ◆ Either in-class definition:

```
class complex {
    float re_, im_;

public:
    float real() const
    {return re_;}

    float imag() const
    {return im_;}
    ...
};
```

- ◆ or out-of-class:

```
class complex {
    float re_, im_;

public:
    inline float real() const;
    inline float imag() const;
    ...
};

float complex::real() const
{
    return re_;
}

float complex::imag() const
{
    return im_;
}
```

Static members

- ◆ are **shared by all objects** of a type
- ◆ Act like global variables in a name space
- ◆ exist even without an object, thus :: notation used:
`Genome::gene_number`
`Genome::set_mutation_rate(2);`
- ◆ Static member functions can only access static member variables!
Reason: which object's members to use???
- ◆ must be declared and defined!
 - ◆ will not link otherwise

```
class Genome {
public:
    Genome(); // constructor

    static const unsigned short
        gene_number=64;
        // static data member
    Genome clone() const;

    static void set_mutation_rate
        (unsigned short);

private:
    unsigned long gene_;
    static unsigned short
        mutation_rate_;
};

// in source file:
unsigned short
    Genome::mutation_rate_=2; //
    definition
```

Class templates

- ◆ same idea as function templates, classes for any given type T
- ◆ Learn it by studying examples:
 - ◆ Array of objects of type T
 - ◆ Complex numbers based on real type T
 - ◆ Statistics class for observables of type T

- ◆ Take care with syntax, where <T> must be used!

```
◆ template <class T> class Array {
    ...
    Array(const Array<T>&); // constructor!
    ...
};
◆ template <class T>
    Array<T>::Array(const Array<T>& cp)
    { ... }
```


The complex template

- ◆ The standard complex class is defined as a template
- ◆ `template <class T> class complex;`
- ◆ It is specialized and optimized for
 - ◆ `complex<float>`
 - ◆ `complex<double>`
 - ◆ `complex<long double>`
- ◆ but in principle also works for `complex<int>`, ...
- ◆ it is a good exercise in template class design to look at the `<complex>` header

Do not avoid **typedef**!

- ◆ Do not store the age of an animal in an int
- ◆ Instead define a new type `age_type`
 - ◆

```
class Animal {
public:
    typedef unsigned short age_t;
    age_t age() const;
private:
    age_t age_;
};
```
- ◆ Allows easy modifications. If we want to allow older ages, just change the typedef to:
 - ◆ `typedef unsigned long age_t;`
- ◆ The rest of the code can remain unchanged!