

# Object Oriented Programming in C++

Jan Faigl

Katedra počítačů  
Fakulta elektrotechnická  
České vysoké učení technické v Praze

Přednáška 14  
BAB36PRGA – Programování v C


## Část I

### Part 1 – Object Oriented Programming


## Overview of the Lecture

- Part 1 – Object Oriented Programming (in C++)
  - Resources
  - Objects and Methods in C++
  - Relationship
  - Inheritance
  - Polymorphism
  - Inheritance and Composition
- Part 2 – Standard Template Library (in C++)
  - Templates
  - Standard Template Library (STL)


## Books

 [The C++ Programming Language, Bjarne Stroustrup, Addison-Wesley Professional, 2013, ISBN 978-0321563842](#)



 [Programming: Principles and Practice Using C++, Bjarne Stroustrup, Addison-Wesley Professional, 2014, ISBN 978-0321992789](#)



 [Effective C++: 55 Specific Ways to Improve Your Programs and Designs, Scott Meyers, Addison-Wesley Professional, 2005, ISBN 978-0321334879](#)



## Example of Encapsulation

- Class `Matrix` encapsulates 2D matrix of `double` values

```
class Matrix {
public:
    Matrix(int rows, int cols);
    Matrix(const Matrix &m);
    ~Matrix();
    inline int rows(void) const { return ROWS; }
    inline int cols(void) const { return COLS; }
    double getValueAt(int r, int c) const;
    void setValueAt(double v, int r, int c);
    void fillRandom(void);
    Matrix sum(const Matrix &m2);
    Matrix operator+(const Matrix &m2);
    Matrix& operator=(const Matrix &m);
private:
    inline double& at(int r, int c) const { return vals[COLS * r + c]; }
private:
    const int ROWS;
    const int COLS;
    double *vals;
};
std::ostream& operator<<(std::ostream& out, const Matrix& m);
```

## Example Matrix – Identity Matrix

- Implementation of the `setIdentity()` using the matrix subscripting operator

```
void setIdentity(Matrix& matrix)
{
    for (int r = 0; r < matrix.rows(); ++r) {
        for (int c = 0; c < matrix.cols(); ++c) {
            matrix(r, c) = (r == c) ? 1.0 : 0.0;
        }
    }
}
Matrix m1(2, 2);
std::cout << "Matrix m1 -- init values: " << std::endl << m1;
setIdentity(m1);
std::cout << "Matrix m1 -- identity: " << std::endl << m1;
```

- Example of output

```
Matrix m1 -- init values:
0.0 0.0
0.0 0.0
Matrix m1 -- identity:
1.0 0.0
0.0 1.0
```

## Example – Matrix Subscripting Operator

- For a convenient access to matrix cells, we can implement operator `()` with two arguments `r` and `c` denoting the cell row and column

```
class Matrix {
public:
    double& operator()(int r, int c);
    double operator()(int r, int c) const;
};

// use the reference for modification of the cell value
double& Matrix::operator()(int r, int c)
{
    return at(r, c);
}

// copy the value for the const operator
double Matrix::operator()(int r, int c) const
{
    return at(r, c);
}
```

*For simplicity and better readability, we do not check range of arguments.*

## Relationship between Objects

- Objects can be in relationship based on the
  - Inheritance – is the relationship of the type **is**
    - Object of descendant class **is** also the ancestor class
    - One class is derived from the ancestor class
      - Objects of the derived class extends the based class
    - Derived class contains all the field of the ancestor class
      - However, some of the fields may be hidden
    - New methods can be implemented in the derived class
      - New implementation **override** the previous one
    - Derived class (objects) are specialization of a more general ancestor (super) class
  - An object can be part of the other objects – it is the **has** relation
    - Similarly to compound structures that contain other struct data types as their data fields, objects can also compound of other objects
    - We can further distinguish
      - Aggregation** – an object is a part of other object
      - Composition** – inner object exists only within the compound object

## Example – Aggregation/Composition

- Aggregation – relationship of the type “has” or “it is composed”
  - Let **A** be aggregation of **B C**, then objects **B** and **C** are contained in **A**
  - It results that **B** and **C** cannot survive without **A**

*In such a case, we call the relationship as composition*

### Example of implementation

```
class GraphComp { // composition
private:
    std::vector<Edge> edges;
};
class GraphComp { // aggregation
public:
    GraphComp(std::vector<Edge>& edges) : edges(
edges) {}
private:
    const std::vector<Edge>& edges;
};

struct Edge {
    Node v1;
    Node v2;
};
struct Node {
    Data data;
};
```

## Example MatrixExt – Extension of the Matrix

- We will extend the existing class **Matrix** to have identity method and also multiplication operator
- We refer the superclass as the **Base** class using **typedef**
- We need to provide a constructor for the **MatrixExt**; however, we used the existing constructor in the base class

```
class MatrixExt : public Matrix {
    typedef Matrix Base; // typedef for referring the superclass
public:
    MatrixExt(int r, int c) : Base(r, c) {} // base constructor
    void setIdentity(void);
    Matrix operator*(const Matrix &m2);
};
// lec14/matrix_ext.h
```

## Inheritance

- Founding definition and implementation of one class on another existing class(es)
- Let class **B** be inherited from the class **A**, then
  - Class **B** is subclass or the derived class of **A**
  - Class **A** is superclass or the base class of **B**
- The subclass **B** has two parts in general:
  - Derived part is inherited from **A**
  - New incremental part contains definitions and implementation added by the class **B**
- The inheritance is relationship of the type **is-a**
  - Object of the type **B** is also an instance of the object of the type **A**
- Properties of **B** inherited from the **A** can be redefined
  - Change of field visibility (protected, public, private)
  - Overriding of the method implementation
- Using inheritance we can create hierarchies of objects

*Implement general function in superclasses or creating abstract classes that are further specialized in the derived classes.*

## Example MatrixExt – Identity and Multiplication Operator

- We can use only the **public** (or **protected**) methods of **Matrix** class

```
#include "matrix_ext.h"
void MatrixExt::setIdentity(void)
{
    for (int r = 0; r < rows(); ++r) {
        for (int c = 0; c < cols(); ++c) {
            (*this)(r, c) = (r == c) ? 1.0 : 0.0;
        }
    }
}
// lec14/matrix_ext.cc
```

## Example MatrixExt – Example of Usage 1/2

- Objects of the class `MatrixExt` also have the methods of the `Matrix`

```

#include <iostream>
#include "matrix_ext.h"
using std::cout;
int main(void)
{
    int ret = 0;
    MatrixExt m1(2, 1);
    m1(0, 0) = 3; m1(1, 0) = 5;
    MatrixExt m2(1, 2);
    m2(0, 0) = 1; m2(0, 1) = 2;
    cout << "Matrix m1:\n" << m1 << std::endl;
    cout << "Matrix m2:\n" << m2 << std::endl;
    cout << "m1 * m2 =\n" << m2 * m1 << std::endl;
    cout << "m2 * m1 =\n" << m1 * m2 << std::endl;
    return ret;
}
    
```

```

clang++ matrix.cc matrix_ext.cc demo-
matrix_ext.cc && ./a.out
Matrix m1:
3.0
5.0
Matrix m2:
1.0 2.0
m1 * m2 =
13.0
m2 * m1 =
3.0 6.0
5.0 10.0
    
```

lec14/demo-matrix\_ext.cc

## Categories of the Inheritance

- Strict inheritance** – derived class takes all of the superclass and adds own methods and attributes. All members of the superclass are available in the derived class. It strictly follows the **is-a** hierarchy
- Nonstrict inheritance** – the subclass derives from the a superclass only certain attributes or methods that can be further redefined
- Multiple inheritance** – a class is derived from several superclasses

## Example MatrixExt – Example of Usage 2/2

- We may use objects of `MatrixExt` anywhere objects of `Matrix` can be applied.
- This is a result of the inheritance

And a first step towards polymorphism

```

void setIdentity(Matrix& matrix)
{
    for (int r = 0; r < matrix.rows(); ++r) {
        for (int c = 0; c < matrix.cols(); ++c) {
            matrix(r, c) = (r == c) ? 1.0 : 0.0;
        }
    }
}
    
```

```

MatrixExt m1(2, 1);
cout << "Using setIdentity for Matrix" << std::endl;
setIdentity(m1);
cout << "Matrix m1:\n" << m1 << std::endl;
    
```

lec14/demo-matrix\_ext.cc

## Inheritance – Summary

- Inheritance is a mechanism that allows
  - Extend data field of the class and modify them
  - Extend or modify methods of the class
- Inheritance allows to
  - Create hierarchies of classes
  - “Pass” data fields and methods for further extension and modification
  - Specialize (specify) classes
- The main advantages of inheritance are
  - It contributes essentially to the code reusability
  - Inheritance is foundation for the **polymorphism**

Together with encapsulation!

## Polymorphism

- Polymorphism can be expressed as the ability to refer in a same way to different objects  
*We can call the same method names on different objects*
- We work with an object whose actual content is determined at the runtime
- Polymorphism of objects** - Let the class **B** be a subclass of **A**, then the object of the **B** can be used wherever it is expected to be an object of the class **A**
- Polymorphism of methods** requires dynamic binding, i.e., static vs. dynamic type of the class
  - Let the class **B** be a subclass of **A** and redefines the method **m()**
  - A variable **x** is of the static type **B**, but its dynamic type can be **A** or **B**
  - Which method is actually called for **x.m()** depends on the dynamic type

## Example MatrixExt – Method Overriding 2/2

- We can call the method `fillRandom()` of the `MatrixExt`

```
MatrixExt *m1 = new MatrixExt(3, 3);
Matrix *m2 = new MatrixExt(3, 3);
m1->fillRandom(); m2->fillRandom();
cout << "m1: MatrixExt as MatrixExt:\n" << *m1 << std::endl;
cout << "m2: MatrixExt as Matrix:\n" << *m2 << std::endl;
delete m1; delete m2;
```

`lec14/demo-matrix_ext.cc`
- However, in the case of `m2` the `Matrix::fillRandom()` is called
 

```
m1: MatrixExt as MatrixExt:
-1.3  9.8  1.2
 8.7 -9.8 -7.9
-3.6 -7.3 -0.6
m2: MatrixExt as Matrix:
 7.9  2.3  0.5
 9.0  7.0  6.6
 7.2  1.8  9.7
```

We need a dynamic way to identify the object type at runtime for the **polymorphism of the methods**

## Example MatrixExt – Method Overriding 1/2

- In `MatrixExt`, we may override a method implemented in the base class `Matrix`, e.g., `fillRandom()` will also use negative values.
 

```
class MatrixExt : public Matrix {
    ...
    void fillRandom(void);
}
void MatrixExt::fillRandom(void)
{
    for (int r = 0; r < rows(); ++r) {
        for (int c = 0; c < cols(); ++c) {
            (*this)(r, c) = (rand() % 100) / 10.0;
            if (rand() % 100 > 50) {
                (*this)(r, c) *= -1.0; // change the sign
            }
        }
    }
}
```

`lec14/matrix_ext.h, lec14/matrix_ext.cc`

## Virtual Methods – Polymorphism and Inheritance

- We need a dynamic binding for polymorphism of the methods
- It is usually implemented as a **virtual method** in object oriented programming languages
- Override methods that are marked as **virtual** has a dynamic binding to the particular dynamic type

## Example – Overriding without Virtual Method 1/2

```
#include <iostream>
using namespace std;
class A {
public:
    void info()
    {
        cout << "Object of the class A" << endl;
    }
};
class B : public A {
public:
    void info()
    {
        cout << "Object of the class B" << endl;
    }
};
A* a = new A(); B* b = new B();
A* ta = a; // backup of a pointer
a->info(); // calling method info() of the class A
b->info(); // calling method info() of the class B
a = b; // use the polymorphism of objects
a->info(); // without the dynamic binding, method of the class A is called
delete ta; delete b;
                                lec14/demo-novirtual.cc
```

## Derived Classes, Polymorphism, and Practical Implications

- Derived class inherits the methods and data fields of the superclass, but it can also add new methods and data fields
  - It can extend and specialize the class
  - It can modify the implementation of the methods
- An object of the derived class can be used instead of the object of the superclass, e.g.,
  - We can implement more efficient matrix multiplication without modification of the whole program

*We may further need a mechanism to create new object based on the dynamic type, i.e., using the `newInstance` virtual method*
- Virtual methods** are important for the **polymorphism**
  - It is crucial to use a virtual **destructor** for a proper destruction of the object
 

*E.g., when a derived class allocate additional memory*

## Example – Overriding with Virtual Method 2/2

```
#include <iostream>
using namespace std;
class A {
public:
    virtual void info() // Virtual !!!
    {
        cout << "Object of the class A" << endl;
    }
};
class B : public A {
public:
    void info()
    {
        cout << "Object of the class B" << endl;
    }
};
A* a = new A(); B* b = new B();
A* ta = a; // backup of a pointer
a->info(); // calling method info() of the class A
b->info(); // calling method info() of the class B
a = b; // use the polymorphism of objects
a->info(); // the dynamic binding exists, method of the class B is called
delete ta; delete b;
                                lec14/demo-virtual.cc
```

## Example – Virtual Destructor 1/4

```
#include <iostream>
class Base {
public:
    Base(int capacity) {
        std::cout << "Base::Base -- allocate data" << std::endl;
        data = new int[capacity];
    }
    virtual ~Base() { // virtual destructor is important
        std::cout << "Base::~Base -- release data" << std::endl;
        delete[] data;
    }
protected:
    int *data;
};
                                lec14/demo-virtual_destructor.cc
```

### Example – Virtual Destructor 2/4

```
class Derived : public Base {
public:
    Derived(int capacity) : Base(capacity) {
        std::cout << "Derived::Derived -- allocate data2" << std::endl;
        data2 = new int[capacity];
    }
    ~Derived() {
        std::cout << "Derived::~Derived -- release data2" << std::endl;
        delete[] data2;
    }
protected:
    int *data2;
};

lec14/demo-virtual_destructor.cc
```

### Example – Virtual Destructor 4/4

- Without `virtual` destructor, e.g.,
 

```
class Base {
    ...
    ~Base(); // without virtualdestructor
};
Derived *object = new Derived(1000000);
delete object;
Base *object = new Derived(1000000);
delete object;
```
- Only both constructors are called, but only destructor of the `Base` class in the second case `Base *object = new Derived(1000000);`

Using Derived	Using Base
Base::Base -- allocate data	Base::Base -- allocate data
Derived::Derived -- allocate data2	Derived::Derived -- allocate data2
Derived::~Derived -- release data2	Base::~Base -- release data
Base::~Base -- release data	

*Only the destructor of Base is called*

### Example – Virtual Destructor 3/4

- Using `virtual` destructor all allocated data are properly released

```
std::cout << "Using Derived " << std::endl;
Derived *object = new Derived(1000000);
delete object;
std::cout << std::endl;
std::cout << "Using Base" << std::endl;
Base *object = new Derived(1000000);
delete object;

lec14/demo-virtual_destructor.cc

clang++ demo-virtual_destructor.cc && ./a.out
Using Derived
Base::Base -- allocate data
Derived::Derived -- allocate data2
Derived::~Derived -- release data2
Base::~Base -- release data
Using Base
Base::Base -- allocate data
Derived::Derived -- allocate data2
Derived::~Derived -- release data2
Base::~Base -- release data
Both destructors Derived and Base are called
```

### Inheritance and Composition

- A part of the object oriented programming is the object oriented design (OOD)
  - It aims to provide “a plan” how to solve the problem using objects and their relationship
  - An important part of the design is identification of the particular objects
  - their generalization to the classes
  - and also designing a class hierarchy
- Sometimes, it may be difficult to decides
  - What is the common (general) object and what is the specialization, which is important step for class hierarchy and applying the inheritance
  - It may also be questionable when to use composition
- Let show the inheritance on an example of geometrical objects

## Example – Is Cuboid Extended Rectangle? 1/2

```
class Rectangle {
public:
    Rectangle(double w, double h) : width(w), height(h) {}
    inline double getWidth(void) const { return width; }
    inline double getHeight(void) const { return height; }
    inline double getDiagonal(void) const
    {
        return sqrt(width*width + height*height);
    }
protected:
    double width;
    double height;
};
```

## Example – Inheritance Cuboid Extend Rectangle

- Class **Cuboid** extends the class **Rectangle** by the **depth**
  - **Cuboid** inherits data fields **width** a **height**
  - **Cuboid** also inherits „getters“ **getWidth()** and **getHeight()**
  - Constructor of the **Rectangle** is called from the **Cuboid** constructor
- The descendant class **Cuboid** extends (override) the **getDiagonal()** methods
 

*It actually uses the method **getDiagonal()** of the ancestor **Rectangle::getDiagonal()***
- We create a “specialization” of the **Rectangle** as an extension **Cuboid** class

**Is it really a suitable extension?**

What is the cuboid area? What is the cuboid circumference?

## Example – Is Cuboid Extended Rectangle? 2/2

```
class Cuboid : public Rectangle {
public:
    Cuboid(double w, double h, double d) :
        Rectangle(w, h), depth(d) {}
    inline double getDepth(void) const { return depth; }
    inline double getDiagonal(void) const
    {
        const double tmp = Rectangle::getDiagonal();
        return sqrt(tmp * tmp + depth * depth);
    }
protected:
    double depth;
};
```

## Example – Inheritance – Rectangle is a Special Cuboid 1/2

- Rectangle is a cuboid with zero depth

```
class Cuboid {
public:
    Cuboid(double w, double h, double d) :
        width(w), height(h), depth(d) {}
    inline double getWidth(void) const { return width; }
    inline double getHeight(void) const { return height; }
    inline double getDepth(void) const { return depth; }
    inline double getDiagonal(void) const
    {
        return sqrt(width*width + height*height + depth*depth);
    }
protected:
    double width;
    double height;
    double depth;
};
```



## Example – Inheritance – Rectangle is a Special Cuboid 2/2

```
class Rectangle : public Cuboid {
public:
    Rectangle(double w, double h) : Cuboid(w, h, 0.0) {}
};
```

- Rectangle is a “cuboid” with zero depth
- Rectangle inherits all data fields: `with`, `height`, and `depth`
- It also inherits all methods of the ancestor
 

*Accessible can be only particular ones*
- The constructor of the `Cuboid` class is accessible and it used to set data fields with the zero `depth`
- Objects of the class `Rectangle` can use all variable and methods of the `Cuboid` class

## Relationship of the Ancestor and Descendant is of the type “is-a”

- Is a straight line segment descendant of the point?
  - Straight line segment does not use any method of a point
 

*is-a?:* segment is a point ? → **NO** → segment is not descendant of the point
- Is rectangle descendant of the straight line segment?
 

*is-a?:* **NO**
- Is rectangle descendant of the square, or vice versa?
  - Rectangle “extends” square by one dimension, but it is not a square
  - Square is a rectangle with the width same as the height
 

*Set the width and height in the constructor!*

## Should be Rectangle Descendant of Cuboid or Cuboid be Descendant of Rectangle?

1. Cuboid is descendant of the rectangle
  - “Logical” addition of the depth dimensions, but methods valid for the rectangle do not work of the cuboid
 

*E.g., area of the rectangle*
2. Rectangle as a descendant of the cuboid
  - Logically correct reasoning on specialization
 

*“All what work for the cuboid also work for the cuboid with zero depth”*
  - Inefficient implementation – every rectangle is represented by 3 dimensions
 

**Specialization is correct**

*Everything what hold for the ancestor have to be valid for the descendant*

*However, in this particular case, usage of the inheritance is questionable.*

## Substitution Principle

- Relationship between two derived classes
- Policy
  - Derived class is a specialization of the superclass
 

*There is the is-a relationship*
  - Wherever it is possible to sue a class, it must be possible to use the descendant in such a way that a user cannot see any difference
 

*Polymorphism*
  - Relationship **is-a** must be permanent

## Composition of Objects

- If a class contains data fields of other object type, the relationship is called **composition**
- Composition creates a hierarchy of objects, but not by inheritance  
*Inheritance creates hierarchy of relationship in the sense of descendant / ancestor*
- Composition is a relationship of the objects – **aggregation** – **consists** / **is compound**
- It is a relationship of the type “**has**”

## Example – Composition 2/3

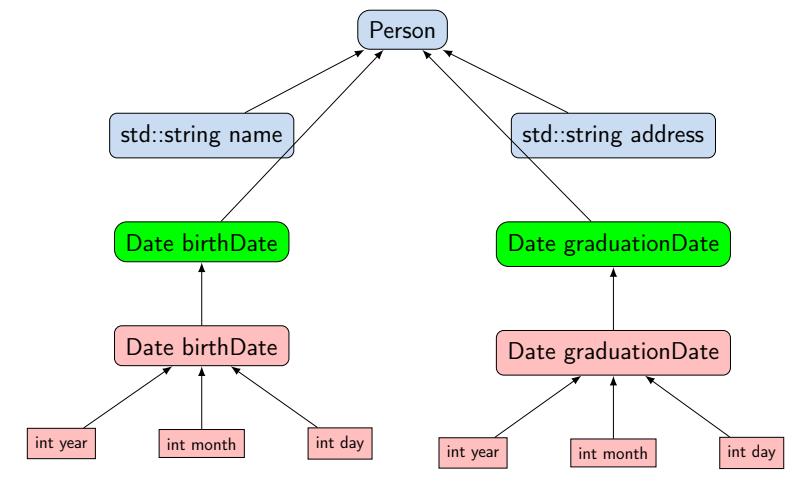
```
#include <string>
class Person {
public:
    std::string name;
    std::string address;
    Date birthDate;
    Date graduationDate;
};

class Date {
public:
    int day;
    int month;
    int year;
};
```

## Example – Composition 1/3

- Each person is characterized by attributes of the **Person** class
  - **name** (string)
  - **address** (string)
  - **birthDate** (date)
  - **graduationDate** (date)
- Date is characterized by three attributes Datum (class **Date**)
  - **day** (int)
  - **month** (int)
  - **year** (int)

## Example – Composition 3/3



## Inheritance vs Composition

- Inheritance objects:
  - Creating a derived class (descendant, subclass, derived class)
  - Derived class is a specialization of the superclass
    - May add variables (data fields) *Or overlapping variables (names)*
    - Add or modify methods
  - Unlike composition, inheritance changes the properties of the objects
    - New or modified methods
    - Access to variables and methods of the ancestor (base class, superclass) *If access is allowed (public/protected)*
- Composition of objects is made of attributes (data fields) of the object type *It consists of objects*
- A distinction between composition and inheritance
  - „Is” test – a symptom of inheritance (**is-a**)
  - „Has” test – a symptom of composition (**has**)

## Část II

### Part 2 – Standard Template Library (STL)

## Inheritance and Composition – Pitfalls

- Excessive usage of composition and also inheritance in cases it is not needed leads to complicated design
- Watch on literal interpretations of the relationship **is-a** and **has**, sometimes it is not even about the inheritance, or composition *E.g., Point2D and Point3D or Circle and Ellipse*
- Prefer composition and not the inheritance *One of the advantages of inheritance is the **polymorphism***
- Using inheritance violates the **encapsulation** *Especially with the access rights set to the **protected***

## Templates

- Class definition may contain specific data fields of a particular type
- The data type itself does not change the behavior of the object, e.g., typically as in
  - Linked list or double linked list
  - Queue, Stack, etc.
  - *data containers*
- Definition of the class for specific type would be identical except the data type
- We can use **templates** for later specification of the particular data type, when the instance of the class is created
- Templates provides **compile-time polymorphism** *In contrast to the run-time polymorphism realized by virtual methods.*

## Example – Template Class

- The template class is defined by the **template** keyword with specification of the type name

```
template <typename T>
class Stack {
public:
    bool push(T *data);
    T* pop(void);
};
```

- An object of the template class is declared with the specified particular type

```
Stack<int> intStack;
Stack<double> doubleStack;
```

## Example – Template Function

- Templates can also be used for functions to specify particular type and use type safety and typed operators

```
template <typename T>
const T & max(const T &a, const T &b)
{
    return a < b ? b : a;
}
double da, db;
int ia, ib;
std::cout << "max double: " << max(da, db) << std::endl;
std::cout << "max int: " << max(ia, ib) << std::endl;
//not allowed such a function is not defined
std::cout << "max mixed " << max(da, ib) << std::endl;
```

## STL

- Standard Template Library (STL) is a library of the standard C++ that provides efficient implementations of the data **containers**, algorithms, functions, and iterators
- High efficiency of the implementation is achieved by templates with compile-type polymorphism
- Standard Template Library Programmer's Guide – <https://www.sgi.com/tech/stl/>

## std::vector – Dynamic "C" like array

- One of the very useful data containers in the STL is **vector** that behaves like C array but allows adding and removing elements.

```
#include <iostream>
#include <vector>
int main(void)
{
    std::vector<int> a;
    for (int i = 0; i < 10; ++i) {
        a.push_back(i);
    }
    for (int i = 0; i < a.size(); ++i) {
        std::cout << "a[" << i << "] = " << a[i] << std::endl;
    }
    std::cout << "Add one more element" << std::endl;
    a.push_back(0);
    for (int i = 5; i < a.size(); ++i) {
        std::cout << "a[" << i << "] = " << a[i] << std::endl;
    }
    return 0;
}
```

lec14/stl-vector.cc

## Summary of the Lecture

## Topics Discussed

- Objects and Methods in C++ – example of 2D matrix encapsulation
  - Subscripting operator
- Relationship between objects
  - Aggregation
  - Composition
- Inheritance – properties and usage in C++
- Polymorphism – dynamic binding and virtual methods
- Inheritance and Composition
- Templates and STL