C++ Constructs by Examples

Jan Faigl

Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague

Lecture 12

B3B36PRG – C Programming Language

Overview of the Lecture

- Part 1 – C++ constructs in `class` Matrix example

Part I

Part 1 – C++ constructs in class Matrix example

Class as an Extended Data Type with Encapsulation

- Data hiding is utilized to encapsulate implementation of matrix

```
class Matrix {
   private:
      const int ROWS;
      const int COLS;
      double *vals;
};
```

1D array is utilized to have a continuous memory. 2D dynamic array can be used in C++11.

- In the example, it is shown
  - How initialize and free required memory in constructor and destructor
  - How to report an error using exception and try-catch statement
  - How to use references
  - How to define a copy constructor
  - How to define (overload) an operator for our class and objects
  - How to use C function and header files in C++
  - How to print to standard output and stream
  - How to define stream operator for output
  - How to define assignment operator

Example – Class `Matrix` – Constructor

- Class `Matrix` encapsulate dimension of the matrix
- Dimensions are fixed for the entire life of the object (const)

```
class Matrix {
   public:
      Matrix(int rows, int cols);
      ~Matrix();
   private:
      const int ROWS;
      const int COLS;
      double *vals;
};
```

```
Matrix::Matrix(int rows, int cols) :
   ROWS(rows), COLS(cols)
{
   vals = new double[ROWS * COLS];
}

Matrix::~Matrix()
{
   delete[] vals;
}
```

Notice, for simplicity we do not test validity of the matrix dimensions.

- Constant data fields `ROWS` and `COLS` must be initialized in the constructor, i.e., in the initializer list

We should also preserve the order of the initialization as the variables are defined

Example – Class `Matrix` – Hidding Data Fields

- Primarily we aim to hide direct access to the particular data fields
- For the dimensions, we provide the so-called "accessor" methods
- The methods are declared as `const` to assure they are read only methods and do not modify the object (compiler checks that)
- Private method `at()` is utilized to have access to the particular cell at *r* row and *c* column

`inline` is used to instruct compiler to avoid function call and rather put the function body directly at the calling place.

```
class Matrix {
   public:
   inline int rows(void) const { return ROWS; } // const method cannot
   inline int cols(void) const { return COLS; } // modify the object
   private:
      // returning reference to the variable allows to set the variable
      // outside, it is like a pointer but automatically dereferenced
      inline double& at(int r, int c) const
      {
         return vals[COLS * r + c];
      }
};
```


Example – Class `Matrix` – Using Reference

- The `at()` method can be used to fill the matrix randomly
- The `random()` function is defined in `<stdlib.h>`, but in C++ we prefer to include C libraries as `<cstdlib>`

```
class Matrix {
   public:
      void fillRandom(void);
   private:
      inline double& at(int r, int c) const { return vals[COLS * r + c]; }
};
#include <cstdlib>
void Matrix::fillRandom(void)
{
   for (int r = 0; r < ROWS; ++r) {
      for (int c = 0; c < COLS; ++c) {
         at(r, c) = (rand() % 100) / 10.0; // set vals[COLS * r + c]
      }
   }
}
```

*In this case, it is more straightforward to just fill 1D array of `vals` for i in 0..(ROWS * COLS).*

Example – Class `Matrix` – Getters/Setters

- Access to particular cell of the matrix is provided through the so-called *getter* and *setter* methods

```
class Matrix {
   public:
      double getValueAt(int r, int c) const;
      void setValueAt(double v, int r, int c);
};
```

- The methods are based on the private `at()` method but will throw an exception if a cell out of `ROWS` and `COLS` would be requested

```
#include <stdexcept>
double Matrix::getValueAt(int r, int c) const
{
   if (r < 0 or r >= ROWS or c < 0 or c >= COLS) {
      throw std::out_of_range("Out of range at Matrix::getValueAt");
   }
   return at(r, c);
}
void Matrix::setValueAt(double v, int r, int c)
{
   if (r < 0 or r >= ROWS or c < 0 or c >= COLS) {
      throw std::out_of_range("Out of range at Matrix::setValueAt");
   }
   at(r, c) = v;
}
```


Example – Class `Matrix` – Exception Handling

- The code where an exception can be raised is put into the **try-catch** block
- The particular exception is specified in the catch by the class name
- We use the program standard output denoted as `std::cout`

We can avoid `std::` by `using namespace std;`

Or just `using std::cout;`

```
#include <iostream>
#include "matrix.h"
int main(void)
{
   int ret = 0;
   try {
      Matrix m1(3, 3);
      m1.setValueAt(10.5, 2, 3); // col 3 raises the exception

      m1.fillRandom();
   } catch (std::out_of_range& e) {
      std::cout << "ERROR: " << e.what() << std::endl;
      ret = -1
   }
   return ret;
}
```

lec12cc/demo-matrix.cc

Example – Class Matrix – Printing the Matrix

- We create a `print()` method to nicely print the matrix to the standard output
- Formatting is controlled by i/o stream manipulators defined in `<iomanip>` header file

```
#include <iostream>
#include <iomanip>

#include "matrix.h"

void print(const Matrix& m)
{
   std::cout << std::fixed << std::setprecision(1);
   for (int r = 0; r < m.rows(); ++r) {
      for (int c = 0; c < m.cols(); ++c) {
         std::cout << (c > 0 ? " " : "") << std::setw(4);
         std::cout << m.getValueAt(r, c);
      }
      std::cout << std::endl;
   }
}
```


Example – Class Matrix – Printing the Matrix

- Notice, the matrix variable `m1` is not copied when it is passed to `print()` function because of passing reference

```
#include <iostream>
#include <iomanip>
#include "matrix.h"

void print(const Matrix& m);

int main(void)
{
   int ret = 0;
   try {
      Matrix m1(3, 3);
      m1.fillRandom();
      std::cout << "Matrix m1" << std::endl;
      print(m1);
...
```

- Example of the output

```
clang++ --pedantic matrix.cc demo-matrix.cc && ./a.out
Matrix m1
 1.3  9.7  9.8
 1.5  1.2  4.3
 8.7  0.8  9.8
```

lec12cc/matrix.h, lec12cc/matrix.cc, lec12cc/demo-matrix.cc

Example – Class Matrix – Copy Constructor

- We may overload the constructor to create a copy of the object

```
class Matrix {
   public:
      ...
      Matrix(const Matrix &m);
      ...
};
```

- We create an exact copy of the matrix

```
Matrix::Matrix(const Matrix &m) : ROWS(m.ROWS), COLS(m.COLS)
{  // copy constructor
   vals = new double[ROWS * COLS];
   for (int i = 0; i < ROWS * COLS; ++i) {
      vals[i] = m.vals[i];
   }
}
```

- Notice, access to private fields is allowed within in the class

We are implementing the class, and thus we are aware what are the internal data fields

Example – Class Matrix – Dynamic Object Allocation

- We can create a new instance of the object by the `new` operator
- We may also combine dynamic allocation with the copy constructor
- Notice, the access to the methods of the object using the pointer to the object is by the `->` operator

```
ratrix m1(3, 3);
m1.fillRandom();
std::cout << "Matrix m1" << std::endl;
print(m1);

Matrix *m2 = new Matrix(m1);
Matrix *m3 = new Matrix(m2->rows(), m2->cols());
std::cout << std::endl << "Matrix m2" << std::endl;
print(*m2);
m3->fillRandom();
std::cout << std::endl << "Matrix m3" << std::endl;
print(*m3);

delete m2;
delete m3;
```

lec12cc/demo-matrix.cc

Example – Class Matrix – Sum

- The method to sum two matrices will return a new matrix

```
class Matrix {
   public:
      Matrix sum(const Matrix &m2);
}
```

- The variable `ret` is passed using the copy constructor

```
Matrix Matrix::sum(const Matrix &m2)
{
   if (ROWS != m2.ROWS or COLS != m2.COLS) {
      throw std::invalid_argument("Matrix dimensions do not match at
     Matrix::sum");
   }
   Matrix ret(ROWS, COLS);
   for (int i = 0; i < ROWS * COLS; ++i) {
      ret.vals[i] = vals[i] + m2.vals[i];
   }
   return ret;
}
```

We may also implement sum as addition to the particular matrix

- The `sum()` method can be then used as any other method

```
Matrix m1(3, 3);
m1.fillRandom();
Matrix *m2 = new Matrix(m1);
Matrix m4 = m1.sum(*m2);
```


Example – Class Matrix – Operator +

- In C++, we can define our operators, e.g., `+` for sum of two matrices
- It will be called like the `sum()` method

```
class Matrix {
   public:
      Matrix sum(const Matrix &m2);
      Matrix operator+(const Matrix &m2);
}
```

- In our case, we can use the already implemented `sum()` method

```
Matrix Matrix::operator+(const Matrix &m2)
{
   return sum(m2);
}
```

- The new operator can be applied for the operands of the `Matrix` type like as to default types

```
Matrix m1(3,3);
m1.fillRandom();
Matrix m2(m1), m3(m1 + m2); // use sum of m1 and m2 to init m3
print(m3);
```


Example – Class Matrix – Output Stream Operator

- An output stream operator `<<` can be defined to pass `Matrix` objects directly to the output stream

```
#include <ostream>
class Matrix { ... };
std::ostream& operator<<(std::ostream& out, const Matrix& m);
```

- It is defined outside the `Matrix`

```
#include <iomanip>
std::ostream& operator<<(std::ostream& out, const Matrix& m)
{
   if (out) {
      out << std::fixed << std::setprecision(1);
      for (int r = 0; r < m.rows(); ++r) {
         for (int c = 0; c < m.cols(); ++c) {
            out << (c > 0 ? " " : "") << std::setw(4);
            out << m.getValueAt(r, c);
         }
         out << std::endl;
      }
   }
   return out;
}
```

"Outside" operator can be used in an output stream pipeline with other data types. In this case, we can use just the public methods. But, if needed, we can declare the operator as a `friend` method to the class, which can access the private fields.

Example – Class Matrix – Example of Usage

- Having the stream operator we can use `+` directly in the output

```
std::cout << "\nMatrix demo using operators" << std::endl;
Matrix m1(2, 2);
Matrix m2(m1);
m1.fillRandom();
m2.fillRandom();
std::cout << "Matrix m1" << std::endl << m1;
std::cout << "\nMatrix m2" << std::endl << m2;
std::cout << "\nMatrix m1 + m2" << std::endl << m1 + m2;
```

- Example of the output operator

```
Matrix demo using operators
Matrix m1
 0.8  3.1
 2.2  4.6

Matrix m2
 0.4  2.3
 3.3  7.2

Matrix m1 + m2
 1.2  5.4
 5.5 11.8
```

lec12cc/demo-matrix.cc

Example – Class Matrix – Assignment Operator =

- We can defined the assignment operator `=`

```
class Matrix {
   public:
      Matrix& operator=(const Matrix &m)
      {
         if (this != &m) { // to avoid overwriting itself
            if (ROWS != m.ROWS or COLS != m.COLS) {
               throw std::out_of_range("Cannot assign matrix with
                   different dimensions");
            }
            for (int i = 0; i < ROWS * COLS; ++i) {
               vals[i] = m.vals[i];
            }
         }
         return *this; // we return reference not a pointer
      }
};
// it can be then used as
Matrix m1(2,2), m2(2,2), m3(2,2);
m1.fillRandom();
m2.fillRandom();
m3 = m1 + m2;
std::cout << m1 << " + " << std::endl << m2 << " = " << std::endl
     << m3 << std::endl;
```

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);
```

lec12cc/matrix.h

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.

Example Matrix – Identity Matrix

- Implementation of the function set the matrix to the identity 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
```

lec12cc/demo-matrix.cc

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;
};
```


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 – 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 refering the superclass

   public:
   MatrixExt(int r, int c) : Base(r, c) {} // base constructor

   void setIdentity(void);
   Matrix operator*(const Matrix &m2);
};
```

lec12cc/matrix_ext.h

Example MatrixExt – Identity and Multiplication Operator

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

`Matrix` does not have any `protected` members

```
#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;
      }
   }
}
Matrix MatrixExt::operator*(const Matrix &m2)
{
   Matrix m3(rows(), m2.cols());
   for (int r = 0; r < rows(); ++r) {
      for (int c = 0; c < m2.cols(); ++c) {
         m3(r, c) = 0.0;
         for (int k = 0; k < cols(); ++k) {
            m3(r, c) += (*this)(r, k) * m2(k, c);
         }
      }
   }
   return m3;
}
```

lec12cc/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
```

lec12cc/demo-matrix_ext.cc

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;
```

lec12cc/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

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

    Together with encapsulation!
  - Inheritance is foundation for the **polymorphism**

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 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
         }
      }
   }
}
```

lec12cc/matrix_ext.h, lec12cc/matrix_ext.cc

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;
```

lec12cc/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 identity the object type at runtime for the **polymorphism of the methods**

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;
```

```
clang++ demo-novirtual.cc
./a.out
Object of the class A
Object of the class B
Object of the class A
```

lec12cc/demo-novirtual.cc

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;
```

```
clang++ demo-virtual.cc
./a.out
Object of the class A
Object of the class B
Object of the class B
```

lec12cc/demo-virtual.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 – Virtual Destructor 1/4

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

lec12cc/demo-virtual_destructor.cc

Example – Virtual Destructor 2/4

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

lec12cc/demo-virtual_destructor.cc

Example – Virtual Destructor 3/4

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

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

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

lec12cc/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 desctructors `Derived` and `Base` are called

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
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
Base::~Base -- release data
```

Only the desctructor of `Base` is 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 – 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 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 – 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

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.

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!

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 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`)

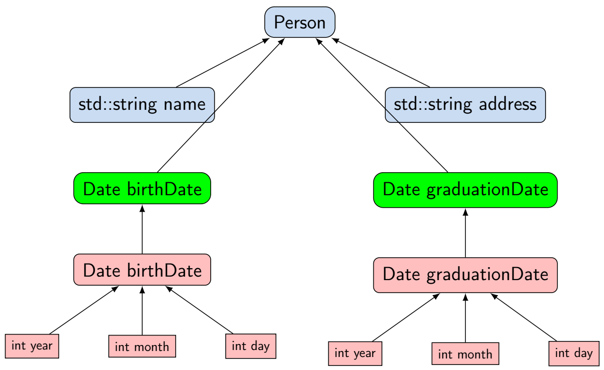
Example – Composition 2/3

```
#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 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 an inheritance
  - „Is" test – a symptom of inheritance (**is-a**)
  - „Has" test – a symptom of composition (**has**)

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`

Summary of the Lecture

Topics Discussed

- 2D Matrix – Examples of C++ constructs
  - Overloading constructors
  - References vs pointers
  - Data hidding – getters/setters
  - Exception handling
  - Operator definition
  - Stream based output
- Operators
  - Subscripting operator
- Relationship between objects
  - Aggregation
  - Composition
- Inheritance – properties and usage in C++
- Polymorphism – dynamic binding and virtual methods
- Inheritance and Composition