C++ Constructs by Examples

Jan Faigl

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

Lecture 11

PRG(A) – C Programming Language

Overview of the Lecture

Part I

Part 1 – Quick Overview of C++ (for C coders)

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

Objects Oriented Programming (OOP)

OOP is a way how to design a program to fulfill requirements and make the sources easy maintain.

- **Abstraction** – concepts (templates) are organized into classes
  - Objects are instances of the classes
- **Encapsulation**
  - Object has its state hidden and provides **interface** to communicate with other objects by sending messages (function/method calls)
- **Inheritance**
  - Hierarchy (of concepts) with common (general) properties that are further specialized in the derived classes
- **Polymorphism**
  - An object with some interface could replace another object with the same interface

C++ for C Programmers

- C++ can be considered as an "extension" of C with additional concepts to create more complex programs in an easier way
- It supports to organize and structure complex programs to be better manageable with easier maintenance
- **Encapsulation** supports "locality" of the code, i.e., provide only public interfance and keep details "hidden"
  - Avoid unintentional wrong usage because of unknown side effects
  - Make the implementation of particular functionality compact and easier to maintain
  - Provide relatively complex functionality with simple to use interface
- Support a tighter link between data and functions operating with the data, i.e., classes combine data (properties) with functions (methods)

From `struct` to `class`

- `struct` defines complex data types for which we can define particular functions, e.g., allocation(), deletion(), initialization(), sum(), print() etc.
- `class` defines the data and function working on the data including the initialization (**constructor**) and deletion (**destructor**) in a compact form
  - Instance of the class is an object, i.e., a variable of the class type

```
typedef struct matrix {
   int rows;
   int cols;
   double *mtx;
} matrix_s;

matrix_s* allocate(int r, int c);
void release(matrix_s **matrix);
void init(matrix_s *matrix);
void print(const matrix_s *matrix);

matrix_s *matrix = allocate(10, 10);
init(matrix);
print(matrix);
release(matrix);
```

```
class Matrix {
   const int ROWS;
   const int COLS;
   double *mtx;
   public:
   Matrix(int r, int c);
   ~Matrix(); //destructor
   void init(void);
   void print(void) const;
};
{
   Matrix matrix(10, 10);
   matrix.init();
   matrix.print();
} // will call destructor
```


Dynamic allocation

- `malloc()` and `free()` and standard functions to allocate/release memory of the particular size in C

```
matrix_s *matrix = (matrix_s*)malloc(sizeof(matrix_s));
matrix->rows = matrix->cols = 0; //inner matrix is not allocated
print(matrix);
free(matrix);
```

- C++ provides two keywords (operators) for creating and deleting objects (variables at the heap) `new` and `delete`

```
Matrix *matrix = new Matrix(10, 10); // constructor is called
matrix->print();
delete matrix;
```

- `new` and `delete` is similar to `malloc()` and `free()`, but
  - Variables are strictly typed and constructor is called to initialize the object
  - For arrays, explicit calling of `delete[]` is required

```
int *array = new int[100]; // aka (int*)malloc(100 * sizeof(int))
delete[] array; // aka free(array)
```

Reference

- In addition to variable and pointer to a variable, C++ supports references, i.e., a reference to an existing object
- Reference is an **alias** to existing variable, e.g.,

```
int a = 10;
int &r = a; // r is reference (alias) to a
r = 13; // a becomes 13
```

- It allows to pass object (complex data structures) to functions (methods) without copying them

Variables are passed by value

```
int print(Matrix matrix)
{// new local variable matrix is allocated
 // and content of the passed variable is copied
}
int print(Matrix *matrix) // pointer is passed
{
   matrix->print();
}
int print(Matrix &matrix)
{
   // reference is passed - similar to passing pointer
   matrix.print(); //but it is not pointer and .  is used
}
```


Class

Describes a set of objects – it is a model of the objects and defines:

- Interface – parts that are accessible from outside

  public, **protected**, **private**
- Body – implementation of the interface (methods) that determine the ability of the objects of the class

  Instance vs class methods
- Data Fields – attributes as basic and complex data types and structures (objects)

  Object composition
  - Instance variables – define the state of the object of the particular class
  - Class variables – common for all instances of the particular class

```
// header file - definition of the class type
class MyClass {
   public:
      /// public read only
      int getValue(void) const;
   private:
      /// hidden data field
      /// it is object variable
      int myData;
};
```

```
// source file - implementation of the methods
int MyClass::getValue(void) const
{
   return myData;
}
```


Object Structure

- The value of the object is structured, i.e., it consists of particular values of the object data fields which can be of different data type

  Heterogeneous data structure unlike an array
- Object is an abstraction of the memory where particular values are stored
  - Data fields are called attributes or instance variables
- Data fields have their names and can be marked as hidden or accessible in the class definition

  Following the encapsulation they are usually hidden

Object:

- Instance of the class – can be created as a variable declaration or by dynamic allocation using the **new** operator
- Access to the attributes or methods is using . or -> (for pointers to an object)

Creating an Object – Class Constructor

- A class instance (object) is created by calling a **constructor** to initialize values of the instance variables

  Implicit/default one exists if not specified
- The name of the constructor is identical to the name of the class

Class definition

```
class MyClass {
   public:
      // constructor
      MyClass(int i);
      MyClass(int i, double d);

   private:
      const int _i;
      int _ii;
      double _d;
};
```

Class implementation

```
MyClass::MyClass(int i) : _i(i)
{
   _ii =  i * i;
   _d = 0.0;
}
// overloading constructor
MyClass::MyClass(int i, double d) : _i(i)
{
   _ii =  i * i;
   _d = d;
}
```

```
{
   MyClass myObject(10); //create an object as an instance of MyClass
} // at the end of the block, the object is destroyed
MyClass *myObject = new MyClass(20, 2.3); //dynamic object creation
delete myObject; //dynamic object has to be explicitly destroyed
```

Relationship between Objects

- Objects may contain other objects
- Object aggregation / composition
- Class definition can be based on an existing class definition – so, there is a relationship between classes
  - Base class (super class) and the derived class
  - The relationship is transferred to the respective objects as instances of the classes

    By that, we can cast objects of the derived class to class instances of ancestor
- Objects communicate between each other using methods (interface) that is accessible to them

Access Modifiers

- Access modifiers allow to implement **encapsulation** (information hiding) by specifying which class members are private and which are public:
  - **public:** – any class can refer to the field or call the method
  - **protected:** – only the current class and subclasses (derived classes) of this class have access to the field or method
  - **private:** – only the current class has the access to the field or method

Modifier	Access		
	Class	Derived Class	"World"
public	✓	✓	✓
protected	✓	✓	✗
private	✓	✗	✗

Constructor and Destructor

- **Constructor** provides the way how to initialize the object, i.e., allocate resources

  Programming idiom – Resource acquisition is initialization (RAII)
- **Destructor** is called at the end of the object life
  - It is responsible for a proper cleanup of the object
  - Releasing resources, e.g., freeing allocated memory, closing files
- Destructor is a method specified by a programmer similarly to a constructor

  However, unlike constructor, only single destructor can be specified
  - The name of the destructor is the same as the name of the class but it starts with the character $\sim$ as a prefix

Constructor Overloading

- An example of constructor for creating an instance of the complex number
- Only a real part or both parts can be specified in the object initialization

```
class Complex {
   public:
      Complex(double r)
      {
         re = r;
      }
      Complex(double r, double i)
      {
         re = r;
         im = i;
      }
      ~Complex() { /* nothing to do in destructor */ }
   private:
      double re;
      double im;
};
```

Both constructors shared the duplicate code, which we like to avoid!

Example – Constructor Calling 1/3

- We can create a dedicated initialization method that is called from different constructors

```
class Complex {

   public:
      Complex(double r, double i) { init(r, i); }
      Complex(double r) { init(r, 0.0); }
      Complex() { init(0.0, 0.0); }

   private:

      void init(double r, double i)
      {
         re = r;
         im = i;
      }

   private:
      double re;
      double im;
};
```


Example – Constructor Calling 2/3

- Or we can utilize default values of the arguments that are combined with initializer list here

```
class Complex {
   public:
      Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
   private:
      double re;
      double im;
};

int main(void)
{
   Complex c1;
   Complex c2(1.);
   Complex c3(1., -1.);
   return 0;
}
```


Example – Constructor Calling 3/3

- Alternatively, in C++11, we can use delegating constructor

```
class Complex {
   public:
      Complex(double r, double i)
      {
         re = r;
         im = i;
      }
      Complex(double r) : Complex(r, 0.0)  {}
      Complex() : Complex(0.0, 0.0) {}

   private:
      double re;
      double im;
};
```


Constructor Summary

- The name is identical to the class name
- The constructor does not have return value

Not even `void`

- Its execution can be prematurely terminated by calling `return`
- It can have parameters similarly as any other method (function)
- We can call other functions, but they should not rely on initialized object that is being done in the constructor
- Constructor is usually **public**
- (**private**) constructor can be used, e.g., for:
  - Classes with only class methods

Prohibition to instantiate class

  - Classes with only constants
  - The so called singletons

E.g., "object factories"

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 constrast 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 STL is `vector` which behaves like C array but allows to add and remove 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;
}
```

lec11cc/stl-vector.cc

Part II

Part 2 – C++ constructs in class Matrix example

Class as an Extended Data Type with Encapsulation

- Data hidding 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 used to 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;
}
```

lec11/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

- The variable m1 is passed as reference to print() function and thus it is not copied

```
#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
```

lec11/matrix.h, lec11/matrix.cc, lec11/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

```
matrix 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;
```

lec11.cc/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 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       Matrix m2       Matrix m1 + m2
 0.8  3.1        0.4  2.3        1.2  5.4
 2.2  4.6        3.3  7.2        5.5 11.8
```

lec11/demo-matrix.cc

Example – Class Matrix – 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);
```

lec11/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 set 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
```

lec11/demo-matrix.cc

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);
};
```

lec11/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;
}
```

lec11/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
```

lec11/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;
```

lec11/demo-matrix_ext.cc

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
         }
      }
   }
}
```

lec11/matrix_ext.h, lec11/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;
```

lec11/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 object type identification at runtime for the **polymorphism of the methods**

Summary of the Lecture

Topics Discussed

- Quick Overview of C++ (for C coders)
- 2D Matrix – Examples of C++ constructs
  - Overloading constructors
  - References vs pointers
  - Data hidding – getters/setters
  - Exception handling
  - Operator definition
  - Stream based output