Introduction to Object Oriented Programming in C++

Jan Faigl

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

Lecture 10

PRG – Programming in C

Overview of the Lecture

Part I

Part 1 – Brief Overview of C89 vs C99 vs C11

Differences between C89 and C99

- *Comments* – In C99 we can use a line comment that begins with `//`
- *Identifiers* – C89 requires compilers to remember the first 31 characters vs. 63 characters in C99
  - Only the first 6 characters of names with external linkage are significant in C89 (no case sensitive)
  - In C99, it is the first 31 characters and case of letters matters
- *Keywords* – 5 new keywords in C99: `inline`, `restrict`, `_Bool`, `_Complex`, and `_Imaginary`
- *Expressions*
  - In C89, the results of `/` and `%` operators for a negative operand can be rounded either up or down. The sign of `i % j` for negative *i* or *j* depends on the implementation.
  - In C99, the result is always truncated toward zero and the sign of `i % j` is the sign of *i*.

Differences between C89 and C99

- *Bool type* – C99 provides `_Bool` type and macros in `stdbool.h`
- *Loops* – C99 allows to declare control variable(s) in the first statement of the `for` loop
- *Arrays* – C99 has
  - **designated initializers** and also allows
  - to use **variable-length arrays**
- *Functions* – one of the directly visible changes is
  - In C89, declarations must precede statements within a block. In C99, it can be mixed.
- *Preprocessor* – e.g.,
  - C99 allows macros with a variable number of arguments
  - C99 introduces `__func__` macro which behaves as a string variable that stores the name of the currently executing function
- *Input/Output* – conversion specification for the *`printf()` and *`scanf()` functions has been significantly changed in C99.

Differences between C89 and C99 – Additional Libraries

- `<stdbool.h>` – macros `false` and `true` that denote the logical values 0 and 1, respectively
- `<stdint.h>` – integer types with specified widths
- `<inttypes.h>` – macros for input/output of types specified in `<stdint.h>`
- `<complex.h>` – functions to perform mathematical operations on complex numbers
- `<tgmath.h>` – type-generic macros for easier call of functions defined in `<math.h>` and `<complex.h>`
- `<fenv.h>` – provides access to floating-point status flags and control modes

Further changes, e.g., see K. N. King: Appendix B

Overview of Changes in C11 – 1/2

- Memory Alignment Control – `_Alignas`, `_Alignof`, and `aligned_alloc`, `<stdalign.h>`
- Type-generic macros – `_Generic` keyword
- `_Noreturn` keyword as the function specifier to declare function does not return by executing return statement (but, e.g., rather `longjmp`) – `<stdnoreturn.h>`
- `<threads.h>` – multithreading support
- `<stdatomic.h>` – facilities for uninterruptible objects access
- Anonymous structs and unions, e.g., for nesting union as a member of a struct

Overview of Changes in C11 – 2/2

- Unicode support – `<uchar.h>`
- Bounds-checking functions – e.g., `strcat_s()` and `strncpy_s()`
- `gets()` for reading a while line from the standard input has been removed.
  - It has been replaced by a safer version called `gets_s()`

    In general, the bound-checking function aims to that the software written in C11 can be more robust against security loopholes and malware attacks.
- `fopen()` interface has been extended for exclusive create-and-open mode (`"..x"`) that behaves as `O_CREAT|O_EXCL` in POSIX used for lock files
  - `wx` – create file for writing with exclusive access
  - `w+x` – create file for update with exclusive access
- Safer `fopen_s()` function has been also introduced

Generic Selection

- In C11, we can use a generic macros, i.e., macros with results that can be computed according to type of the pass variable (expression)

```
double f_i(int i)
{
   return i + 1.0;
}
double f_d(double d)
{
   return d - 1.0;
}

#define fce(X) _Generic((X),\
int: f_i,\
double: f_d\
)(X)
```

```
int main(void)
{
   int i = 10;
   double d = 10.0;

   printf("i = %d; d = %f\n", i, d);
   printf("Results of fce(i) %f\n", fce(i));
   printf("Results of fce(d) %f\n", fce(d));
   return EXIT_SUCCESS;
}
```

lec10/demo-matrix.cc

```
clang -std=c11 generic.c -o generic && ./generic
i = 10; d = 10.000000
Results of fce(i) 11.000000
Results of fce(d) 9.000000
```

- A function is selected according to the type of variable during compilation.

  Static (parametric/compile-time) polymorphism

Part II

Part 2 – Introduction to Object Oriented Programming

C

- C was developed by Dennis Ritchie (1969–1973) at AT&T Bell Labs
- C is a **procedural (aka structural) programming language**
- C is a subset of C++
- The solution is achieved through a sequence of procedures or steps
- C is a **function driven language**

C++

- Developed by **Bjarne Stroustrup** in 1979 with C++'s predecessor "C with Classes"
- C++ is **procedural** but also an **object oriented programming language**
- C++ can run most of C code
- C++ can model the whole solution in terms of objects and that can make the solution better organized
- C++ is an **object driven language**

C

- Concept of virtual functions is not present in C
- No operator overloading
- Data can be easily accessed by other external functions
- C is a *middle level language*
- C programs are divided into **modules and procedures**
- C programs use *top-down approach*

C++

- C++ offers the facility of using **virtual functions**
- C++ allows **operator overloading**
- Data can be put inside objects, which provides better data security
- C++ is a high level language
- C++ programs are divided into **classes and functions**
- C++ programs use *bottom-up approach*

C

- Does not provide namespaces
- **Exception handling** is not easy in C
- Inheritance is not possible
- Function overloading is not possible
- Functions are used for input/output, e.g., `scanf()` and `printf()`
- Does not support reference variables
- Does not support definition (overloading) operators

C++

- **Namespaces** are available
- **Exception handling** through **Try** and **Catch** block
- **Inheritance** is possible
- **Function overloading** is possible (i.e., functions with the same name)
- Objects (streams) can be use for input/output, e.g., `std::cin` and `std::cout`
- Supports **reference variables**, using `&`
- C++ supports definition (overloading) of the **operators**

C

- Provides `malloc()` (`calloc()`) for dynamic memory allocation
- It provides `free()` function for memory de-allocation
- Does not support for virtual and friend functions
- Polymorphism is not possible
- C supports only built-in data types
- Mapping between data and functions is difficult in C
- C programs are saved in files with extension .c

C++

- C++ provides `new` operator for memory allocation
- It provides `delete` and (`delete[]`) operator for memory de-allocation
- C++ supports **virtual** and **friend** functions
- C++ offers **polymorphism**
- It supports both built-in and user-defined data types
- In C++ data and functions are easily mapped through objects
- C++ programs are saved in files with extension `.cc`, `.cxx` or `.cpp`

http://techwelkin.com/difference-between-c-and-c-plus-plus

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

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	Access Derived Class	Access "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 ∼ as a prefix

Constructor Overloading

- An example of constructor for creating an instance of the complex number
- In an object initialization, we may specify only real part or both the real and imaginary part

```
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"

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 `rand()` 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;
}
```

lec10/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 matrix variable `m1` is not copied as it is passed as reference to `print()` function

```
#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
```

lec10/matrix.h, lec10/matrix.cc, lec10/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;
```

lec10/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
```

lec10/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;
```


Summary of the Lecture

Topics Discussed

- C89 vs C99 vs C11 – a brief overview of the changes
- C vs C++ – a brief overview of differences
- Object oriented programming in C++
  - Introduction to OOP
  - Classes and objects
  - Constructor
  - Examples of C++ constructs
    - Overloading constructors
    - References vs pointers
    - Data hiding – getters/setters
    - Exception handling
    - Operator definition
    - Stream based output
- Next: OOP – Polymorphism, inheritance, and virtual methods.