Quick Introduction to C++

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 12

B0B36PRP – Procedurální programování

Overview of the Lecture

Část I

Part 1 – Quick Introduction to 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 ∼ 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;
}
```

lec12cc/stl-vector.cc

Summary of the Lecture

Topics Discussed

- Classes and objects
- Constructor/destructor
- Templates and STL
- Next: C++ constructs (polymorphism, inheritance, and virtual methods, etc.) in examples