Summary of C++ Constructs

Jan Faigl

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

Lecture 12

PRG(A) – Programming in C

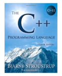
Overview of the Lecture

Part I

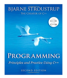
Part 1 – Summary of C++ Constructs

Resources – Books

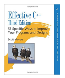
The C++ Programming Language, *Bjarne Stroustrup*, Addison-Wesley Professional, 2013, ISBN 978-0321563842



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	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
- 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"

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;
};
```


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

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
```

lec12/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
```

lec12/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>
class Base {
   public:
      Base(int capacity) {
         std::cout << "Base::Base -- allocate data" << std::endl;
         data = new int[capacity];
      }
      virtual ~Base() { // virtual destructor is important
         std::cout << "Base::~Base -- release data" << std::endl;
         delete[] data;
      }
   protected:
      int *data;
};
```

lec12/demo-virtual_destructor.cc

Example – Virtual Destructor 2/4

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

lec12/demo-virtual_destructor.cc

Example – Virtual Destructor 3/4

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

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

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

lec12/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`)

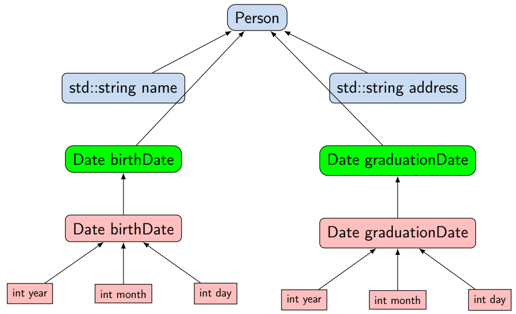
Example – Composition 2/3

```
#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`

Part II

Part 2 – Standard Template Library (STL)

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;
}
```

lec12/stl-vector.cc

Summary of the Lecture

Topics Discussed

- Classes and objects
- Constructor/destructor
- Templates and STL
- Relationship between objects
  - Aggregation
  - Composition
- Inheritance – properties and usage in C++
- Polymorphism – dynamic binding and virtual methods
- Inheritance and Composition