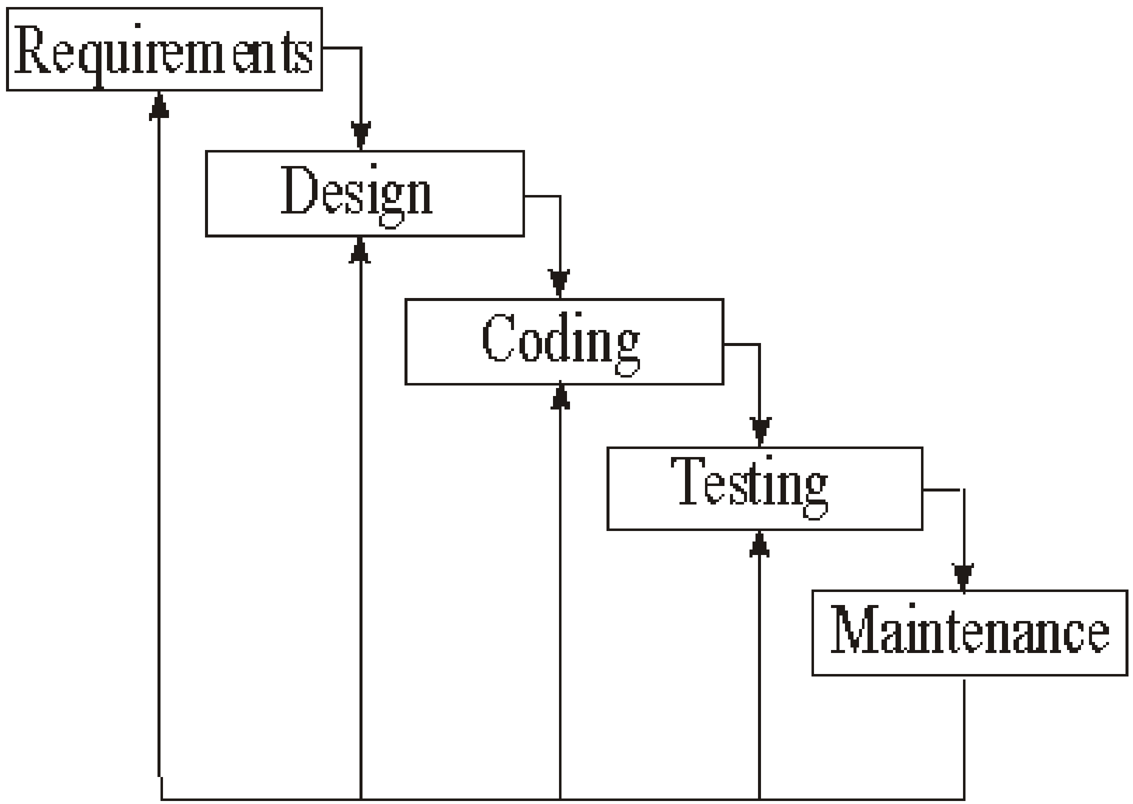
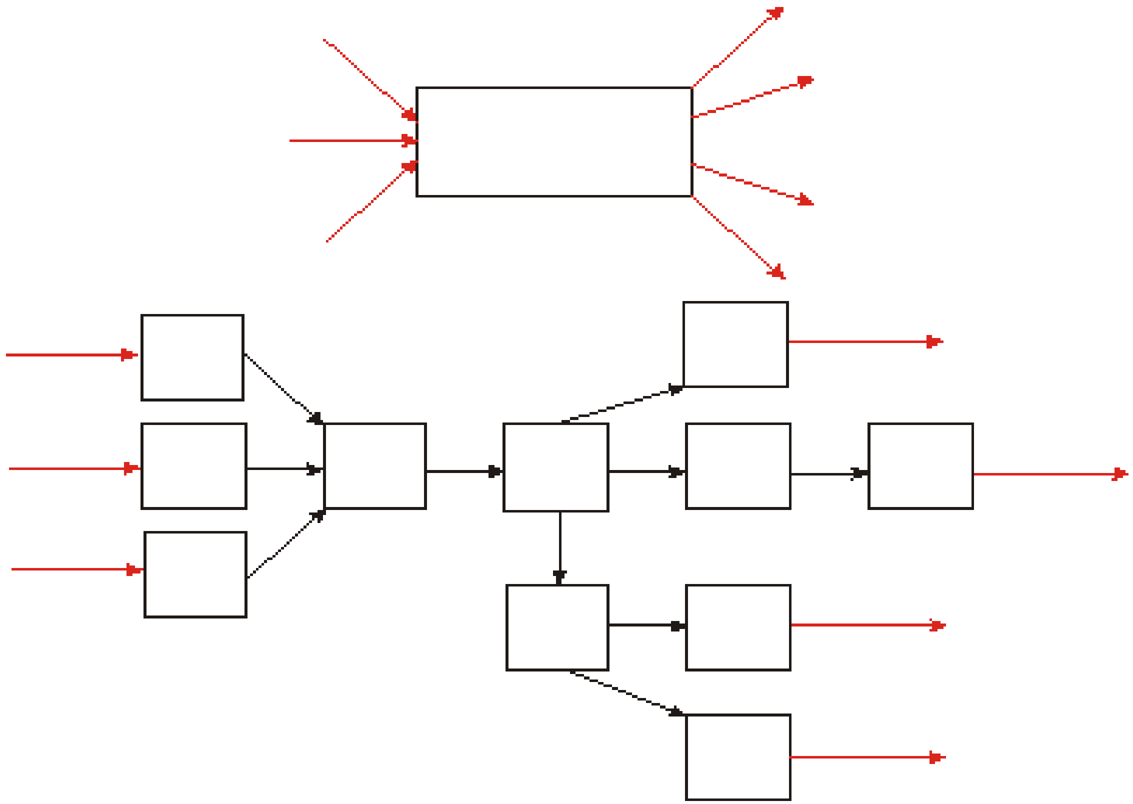
Waterfall model



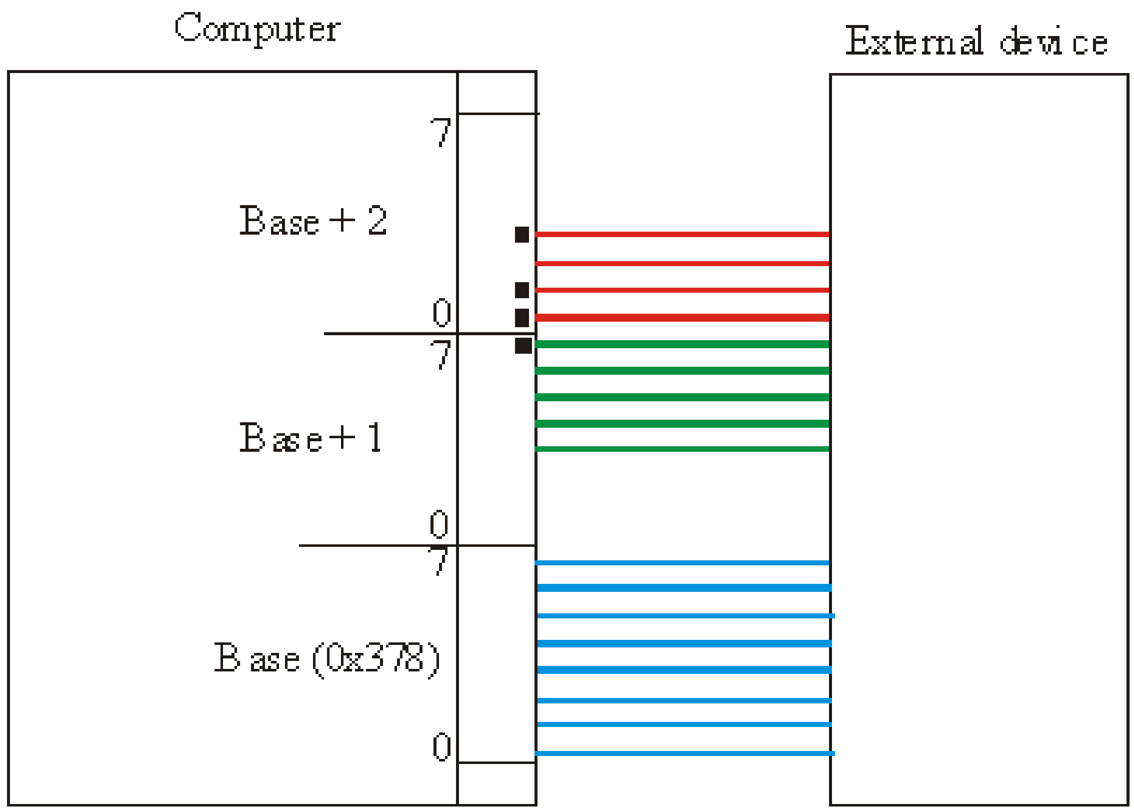
Structured approach



Object-oriented approach

Keywords:
1. Application area, object, attribute, value, state, message, method or function, class, member, software model.
2. Encapsulation, inheritance, polymorphism and virtual functions.
3. Object-oriented approach, object-oriented design, object-oriented programming, object-oriented language

Parallel port as an example (1)



25-pin connector. Blue wires are for output, green wires for input, red wires for the both.
Wires are controlled by accessing the bytes located in the I/O address space.
Bit 7 in (Base + 1) and bits 3, 1 and 0 in (Base + 2) are inverted. Use XOR to get the actual value.

Parallel port as an example (2)

Direct access of bytes from the I/O address space is not possible:

```
#include "dos.h"
#define BASE 0x378LU

unsigned char data_to_write;
unsigned char read_data;

outportb(BASE, data_to_write);

read_data = inportb(BASE + 1) ^ 0x80;
read_data = inportb(BASE + 2) ^ 0x0B;

// 0 ^ 0 = 0;  1 ^ 1 = 0;  1 ^ 0 = 1;  0 ^ 1 = 1
```

Defining a class

File ParallelPort.h:

```
class ParallelPort
{
  public:  DWORD m_BaseAddr;
            // m_BaseAddr is an attribute (data member)
            // prefix "m_" for attibute names is recommended by Microsoft
            void SetBaseAddr(DWORD);
            // SetBaseAddr is a method (member function)
            void WritePort0(BYTE); // Port 0 - BASE
            void WritePort2(BYTE); // Port 2 - BASE + 2
            BYTE ReadPort1(); // Port 1 - BASE + 1
            BYTE ReadPort2();
}; // Do not forget the semicolon
```

Implementing a class (1)

File ParallelPort.cpp:

```
#include "dos.h"
#include "Windows.h"
#include "ParallelPort.h"

void ParallelPort::SetBaseAddr(DWORD a)
{
  m_BaseAddr = a;
}
void ParallelPort::WritePort0(BYTE data)
{
  outportb(m_BaseAddr, data);
 }
void ParallelPort::WritePort2(BYTE data)
{
  outportb(m_BaseAddr + 2, data ^ 0x0B);
 }
```

Implementing a class (2)

File ParallelPort.cpp continues:

```
BYTE ParallelPort::ReadPort1()
{
  return inportb(m_BaseAddr + 1) ^ 0x80;
 }
BYTE ParallelPort::ReadPort2()
{
  return inportb(m_BaseAddr + 2) ^ 0x0B;
 }
```

Using a class (1)

```
#include "Windows.h"
#include "ParallelPort.h"

ParallelPort Port1;   // We got 4 bytes for storing the value of m_BaseAddr
// This memory field is accessed with expression Port1.m_BaseAddr
Port1.SetBaseAddr(0x378);
            // 0x378 is written on the memory field reserved for m_BaseAddr
            // Actually, we performed Port1.m_BaseAddr = 0x378;
ParallelPort Port2;
Port2.SetBaseAddr(0x3BC);
            // We got another 4 bytes for storing the value of m_BaseAddr
            // and write onto it 0x3BC
Port1.WritePort0('A');
            // Actually, we performed outportb(0x378, 'A');
Port2.WritePort0('B');
            // Actually, we performed outportb(0x3BC, 'B');
```

Using a class (2)

```
#include "Windows.h"
#include "ParallelPort.h"

ParallelPort *pPort1;
pPort1 = new ParallelPort;   // We got 4 bytes for storing the value of m_BaseAddr
// This memory field is accessed with expression pPort1->m_BaseAddr
pPort1->SetBaseAddr(0x378);
        // 0x378 is written on the memory field reserved for m_BaseAddr
        // Actually, we performed pPort1->m_BaseAddr = 0x378;
ParallelPort *pPort2;
pPort2 = new ParallelPort;
pPort2->SetBaseAddr(0x3BC);
        // We got another 4 bytes for storing the value of m_BaseAddr
        // and write onto it 0x3BC
pPort1->WritePort0('A');
        // Actually, we performed outportb(0x378, 'A');
pPort2->WritePort0('B');
        // Actually, we performed outportb(0x3BC, 'B');
```

Access control (1)

Access control or encapsulation.

```
class ParallelPort
{
  private:  DWORD m_BaseAddr;
  public:   void SetBaseAddr(DWORD); // accessor function
            DWORD GetBaseAddr(); // accessor function
            void WritePort0(BYTE);
            void WritePort2(BYTE);
            BYTE ReadPort1();
            BYTE ReadPort2();
};
DWORD ParallelPort::GetBaseAddr()
{
   return m_BaseAddr;
}
```

Keywords private and public are access modifiers. Public members are accessible without any restrictions. Private members can be accessed only within functions from the same class.

Access control (2)

To access private attributes, use accessor functions:

```
ParallelPort Port1;
Port1.m_BaseAddr = 0x378; // ERROR
printf("%lu", Port1.m_BaseAddr); // ERROR
Port1.SetBaseAddr(0x378);
printf("%lu", Port1.GetBaseAddr());
```

Public attributes are not recommended.

Main reasons for encapsulation:

- data hiding
- data protecting

Public methods comprise the class interface.

Constructor (1)

```
class ParallelPort
{
  private:  DWORD m_BaseAddr;
  public:   ParallelPort(DWORD); // constructor, always public
            void SetBaseAddr(DWORD);
            DWORD GetBaseAddr();
            void WritePort0(BYTE);
            void WritePort2(BYTE);
            BYTE ReadPort1();
            BYTE ReadPort2();
};

ParallelPort::ParallelPort(DWORD a)
{
   m_BaseAddr = a;
 }

ParallelPort Port1(0x378); // constructor is called, m_BaseAddr is initialized to 0x378
ParallelPort *pPort2 = new ParallelPort(0x3BC);
                                // constructor is called, m_BaseAddr is initialized to 0x3BC
```

Constructor (2)

```
class ParallelPort
{
  private:  DWORD m_BaseAddr;
  public: void SetBaseAddr(DWORD); // the class has no user-defined constructors
              ………………………………………………………
};

ParallelPort Port1;
// the default constructor is called, no initializations

ParallelPort Ports[3];
// the default constructor is called 3 times, no initializations

ParallelPort *pPort2 = new ParallelPort;
// the default constructor is called, no initializations

ParallelPort *pPorts = new ParallelPort[3];
// the default constructor is called 3 times, no initializations
```

Constructor (3)

```
class ParallelPort
{
  private:  DWORD m_BaseAddr;
  public:   ParallelPort(DWORD); // user-defined constructor
            ……………………………………………………
};
ParallelPort Port1, Ports[3], *pPort2 = new ParallelPort, *pPorts = new ParallelPort[3];
// Error, because the default constructor is created only when there are no user-
// defined constructors.
```

To solve the problem write an additional empty constructor

```
ParallelPort::ParallelPort() {     }
```

```
ParallelPort Port1(0x378); // user-defined constructor is called
ParallelPort Port2;   // user-defined empty constructor is called, do not write "Port2();"
```

Constructor (4)

```
class ParallelPort
{
  private:  DWORD m_BaseAddr;
  public:   ParallelPort(DWORD = 0x378); // constructor with default arguments
            void SetBaseAddr(DWORD);
            DWORD GetBaseAddr();
            ……………………………………………………
};

ParallelPort Port1,
             *pPort1 = new ParallelPort,
                   // constructor is called, m_BaseAddr is initialized to 0x378
                   // do not write "new ParallelPort()"
              Port2(0x3BC),
              *pPort2 = new ParallelPort(0x3BC),
                   // constructor is called, m_BaseAddr is initialized to 0x3BC
              Ports[3], *pPorts = new ParallelPort[3];
                   // constructors are called, in all the ports the m_BaseAddr is
                   // initialized to 0x378
```

Constructor (5)

```
class ParallelPort
{
  private:  DWORD m_BaseAddr;
  public:   ParallelPort(DWORD); // constructor 1
            ParallelPort(); // constructor 2
             .......................................................
};
ParallelPort::ParallelPort(DWORD a)
{            // constructor 1
   m_BaseAddr = a;
 }
ParallelPort::ParallelPort()
{            // constructor 2
   m_BaseAddr =0x378;
 }
ParallelPort Port1, *pPort1 = new ParallelPort,
// constructor 2 is called, m_BaseAddr is initialized to 0x378
 Port2(0x3BC),   *pPort2 = new ParallelPort(0x3BC),
// constructor 1 is called, m_BaseAddr is initialized to 0x3BC
Ports[3], *pPorts = new ParallelPort[3];
// constructors 2 are called, in all the ports the m_BaseAddr is initialized to 0x378
```

Inline member functions

File ParallelPort.h:

```
class ParallelPort
{
  private:  DWORD m_BaseAddr;
  public:   ParallelPort(DWORD a ) {  m_BaseAddr = a; }
            ParallelPort() { }
            void SetBaseAddr(DWORD a ) {  m_BaseAddr = a; }
            DWORD GetBaseAddr() {  return m_BaseAddr;  }
            void WritePort0(BYTE d) { outportb(m_BaseAddr, d); }
            void WritePort2(BYTE d)  { outportb(m_BaseAddr + 2, data ^ 0x0B); }
            BYTE ReadPort1() { return inportb(m_BaseAddr + 1) ^ 0x80; }
            BYTE ReadPort2() { return inportb(m_BaseAddr + 2) ^ 0x0B; }
};
```

As here all the methods are inline functions, file ParallelPort.cpp is not needed.

Aggregation (1)

File DAC.h (digital-analog convertor):

```
class DAC
{
  private: ParallelPort m_Port;
  public:   DAC(DWORD a ) : m_Port(a) {  }
            // There are no initializations for DAC itself. But the constructor of
            // ParallelPort must be called.
            void SetBaseAddr(DWORD a ) {  m_Port.SetBaseAddr(a); }
            DWORD GetBaseAddr() {  return m_Port.GetBaseAddr();  }
            void Write (BYTE d) { m_Port.WritePort0(d); }
};
```

Aggregation option 1: an attribute of a class (the container) is an object from another class.

Here DAC is the container class involving attribute m_Port – an instance of class ParallelPort.

```
DAC *pDAC = new DAC(0x378);
pDAC->Write(0xFF);
```

Aggregation (2)

File DAC.h:

```
class DAC
{
  private: ParallelPort *m_pPort;
  public:   DAC(DWORD a = 0x378) { m_pPort = new ParallelPort(a);  }
            ~DAC()  {  delete m_pPort;  } // destructor
            void SetBaseAddr(DWORD a ) {  m_pPort->SetBaseAddr(a); }
            DWORD GetBaseAddr() {  return m_pPort->GetBaseAddr();  }
            void Write (BYTE d) { m_pPort->WritePort0(d); }
};
```

Aggregation option 2: an attribute of a class (the container) is a pointer to object from another class. This object is created in the container and must be destroyed by the container.

```
DAC *pDAC = new DAC;
pDAC->Write(0xFF);
delete pDAC;  // destructor is called

DAC *pDACs = new DAC[3];
delete[] pDACs; // destructors for all the DACs are called
```

Destructors

Destructors are called:

- when a local object (auto memory class) stops existing;
- when a global object stops existing (i.e. the application terminates);
- when a dynamically created object is deleted.

Tasks for destructor:

- release memory allocated by the member functions of the current class
- close files, sockets and the other resources opened or created by the functions of the current class.

Destructors are called automatically. Never call the destructor directly from your code.

Association

File DAC.h:

```
class DAC
{
  private: ParallelPort *m_pPort;
  public:   DAC(ParallelPort *p) { m_pPort = p;  }
            void SetBaseAddr(DWORD a ) {  m_pPort->SetBaseAddr(a); }
            DWORD GetBaseAddr() {  return m_pPort->GetBaseAddr();  }
            void Write (BYTE d) { m_pPort->WritePort0(d); }
};
```

Association: an attribute of a class is a pointer to object from another class. This object, however, exists outside of the container (i.e. has its own lifetime).

```
ParallelPort pPort = new ParallelPort(0x3BC);
DAC pDAC = new DAC(pPort);
pDAC->Write(0xFF);
delete pDAC;
delete pPort;
```

```
class Node
{
private:
   void *m_pRecord;
   Node *m_pNext;
public:
   Node() { m_pRecord = NULL;  m_pNext = NULL; }
   Node(void *p1, Node *p2 = NULL) { m_pRecord = p1;  m_pNext = p2; }
   void SetRecord(void *p) { m_pRecord = p; }
   void *GetRecord() { return m_pRecord; }
   void SetNext(Node *p) { m_pNext = p; }
   Node *GetNext(){ return m_pNext; }
};

class LinkedList
{
private:
   Node *m_pHead;
   Node *m_pTail;
   int m_nRecords;
public:
   LinkedList() { m_pHead = m_pTail = NULL; m_nRecords = 0; }
   BOOL IsEmpty() { m_pHead ? FALSE : TRUE; }
   int GetMeasure() { return m_nRecords; }
   void *GetHead() {  return m_pHead->GetRecord(); }
   void *GetTail() {  return m_pTail->GetRecord(); }
   void *Get(int); // returns pointer to the record on the given position
   BOOL InsertToHead(void *); // the new record will be the first
```

```
    BOOL InsertToTail(void *); // the new record will be the last
    BOOL Insert(void *, int);   // the new record will be on the given position
    void *RemoveFromHead();
    void *RemoveFromTail();
    void *Remove(int); // remove record from the given position
    Node *operator[](int);
};
```

```cpp
#include "stdafx.h"
#include "Windows.h"
#include "LinkedList.h"

void *LinkedList::Get(int i)
{
    if (i < 0 || i > m_nRecords - 1)
        return NULL;
    Node *pTemp = m_pHead;
    for (int j = 0;  j != i;  j++)
        pTemp = pTemp->GetNext();
    return pTemp->GetRecord();
}

BOOL LinkedList::InsertToHead(void *p)
{
    if (!p)
        return FALSE;
    if (!m_nRecords)
        m_pHead = m_pTail = new Node(p);
    else
        m_pHead = new Node(p, m_pHead);
    m_nRecords++;
    return TRUE;
}
```

```
BOOL LinkedList::InsertToTail(void *p)
{
    if (!p)
        return FALSE;
    if (!m_nRecords)
        return InsertToHead(p);
    else
    {
        Node *pNew;
        m_pTail->SetNext(pNew = new Node(p));
        m_pTail = pNew;
    }
    m_nRecords++;
    return TRUE;
}

BOOL LinkedList::Insert(void *p, int i)
{
    if (!i)
        return InsertToHead(p);
    if (i == m_nRecords)
        return InsertToTail(p);
    if (i < 0 || i > m_nRecords || !p)
        return FALSE;
    Node *pTemp = m_pHead, *pNew;
    for (int j = 0;  j != i - 1;  j++)
        pTemp = pTemp->GetNext();
```

```
        pTemp->SetNext(pNew = new Node(p, pTemp->GetNext()));
        m_nRecords++;
        return TRUE;
    }

    void *LinkedList::RemoveFromHead()
    {
        if (!m_nRecords)
            return NULL;
        Node *pTemp = m_pHead;
        m_pHead = m_pHead->GetNext();
        m_nRecords--;
        if (!m_nRecords)
            m_pTail = NULL;
        void *p = pTemp->GetRecord();
        delete pTemp;
        return p;
    }

    void *LinkedList::RemoveFromTail()
    {
        if (m_nRecords <= 1)
            return RemoveFromHead();
        Node *pTemp1 = m_pHead, *pTemp2 = m_pTail;
        for (int j = 0;  j != m_nRecords - 2;  j++)
            pTemp1 = pTemp1->GetNext();
        pTemp1->SetNext(NULL);
        m_pTail = pTemp1;
```

```
    m_nRecords--;
    void *p = pTemp2->GetRecord();
    delete pTemp2;
    return p;
}

void *LinkedList::Remove(int i)
{
    if (i < 0 || i > m_nRecords - 1)
        return NULL;
    if (i == 0)
        return RemoveFromHead();
    if (i == m_nRecords - 1)
        return RemoveFromTail();
    Node *pTemp1 = m_pHead, *pTemp2;
    for (int j = 0;  j != i - 1;  j++)
        pTemp1 = pTemp1->GetNext();
    pTemp2 = pTemp1->GetNext();
    pTemp1->SetNext(pTemp2->GetNext());
    m_nRecords--;
    void *p = pTemp2->GetRecord();
    delete pTemp2;
    return p;
}
```

```
Node *LinkedList::operator[](int i)
{
    if (i < 0 || i > m_nRecords - 1)
        return NULL;
    Node *pTemp = m_pHead;
    for (int j = 0;  j != i;  j++)
        pTemp = pTemp->GetNext();
    return pTemp;
}
```

```cpp
#include "stdafx.h"
#include "Windows.h"
#include "LinkedList.h"

int _tmain(int argc, _TCHAR* argv[])
{
    LinkedList List;
    List.InsertToHead(_T("Jaan"));
    LinkedList *pList = new LinkedList;
    pList->InsertToHead(_T("Jaan"));
    return 0;
}
```

Inheritance (1)

File DAC.h:

```
class DAC : public ParallelPort
{
  public:   DAC(DWORD a) : ParallelPort(a) {  }
            void Write (BYTE d) { WritePort0(d); }
};
```

ParallelPort is the base class, DAC is the derived class. All the members of base class are also the members of derived class.

```
DAC pDAC = new DAC(0x3BC);
printf("%lu", pDAC->GetBaseAddr()); // GetBaseAddr is inherited from ParallelPort
pDAC->Write(0xFF);
delete pDAC;
```

Inheritance (2)

```
class LEDs : public ParallelPort // bank of 8 LEDs, only one of them can be on
{
  private: int m_Lit;  // index of the LED that is lit. –1 means that all the LEDs are off
  public:  LEDs(DWORD a) : ParallelPort(a) { m_Lit = -1; }
           LEDs(DWORD a, int i) : ParallelPort(a) { TurnOn(i); }
                    // The constructor of base class is called first
           LEDs() { m_Lit = -1; }
           void TurnOn(int i)
           {
              if (i >= 0 && i <= 7)
              {
                WritePort0(0x01 << i);
                 m_Lit = i;
               }
           }
           void TurnOff()  {  WritePort0(0);  m_Lit = -1;  }
           int WhichIsLit() { return m_Lit; }
};
```

Inheritance (3)

Problems with constructors:

The base class constructor is always called before the derived class constructor. The parameter list of derived class constructor must also contain the parameters for the base class constructor:

ddd::ddd(full_list_of_parameters) : bbb(list_of_base_class_constructor_parameters)
{……………………………...}

If the base class constructor does not have parameters, the derived class constructor is simply written as

ddd::ddd(full_list_of_parameters) {……………………………...}

Here *ddd* is the derived class and *bbb* is the base class.

Problems with destructors:

The derived class destructor is always called before the base class destructor.

Problems with matching names:

Suppose that class bbb has attribute *m_Attr* ja method *void fun()*. Suppose that class ddd has also attribute *m_Attr* ja method *void fun()*. Then in software written for class *ddd*:

- To access *m_Attr* and *fun()* from class *ddd* write simply *m_Attr* and *fun().*
- To access *m_Attr* and *fun()* from class *bbb* write *bbb::m_Attr* and *bbb::fun()*.

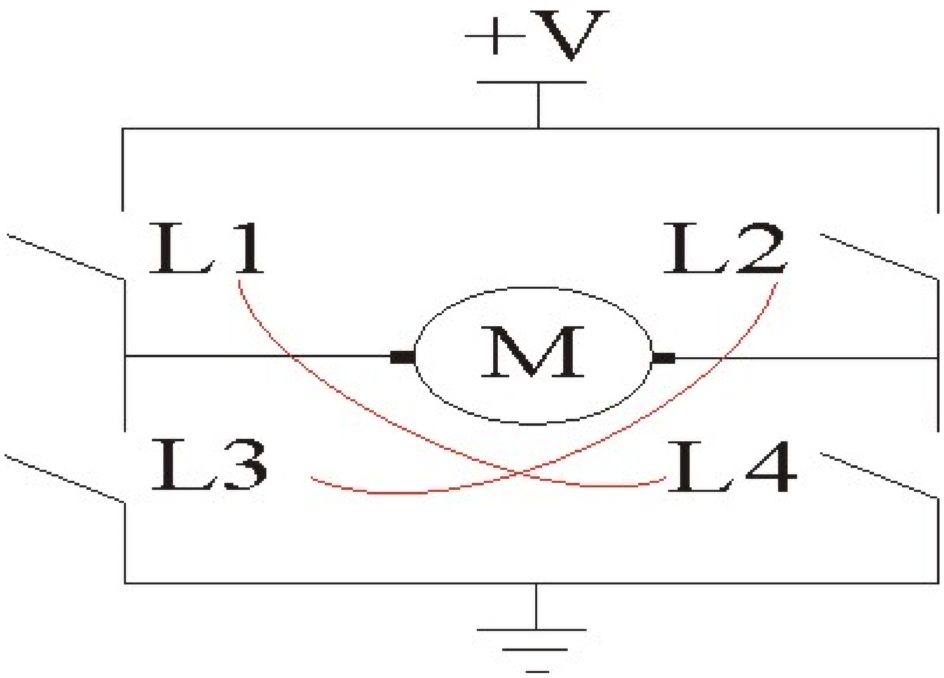
Futher, suppose that the application has global variable *m_Attr* ja global function *fun()* (they are not members of some of the classes). To access them write *::m_Attr* and *::fun()*.

Inheritance and access

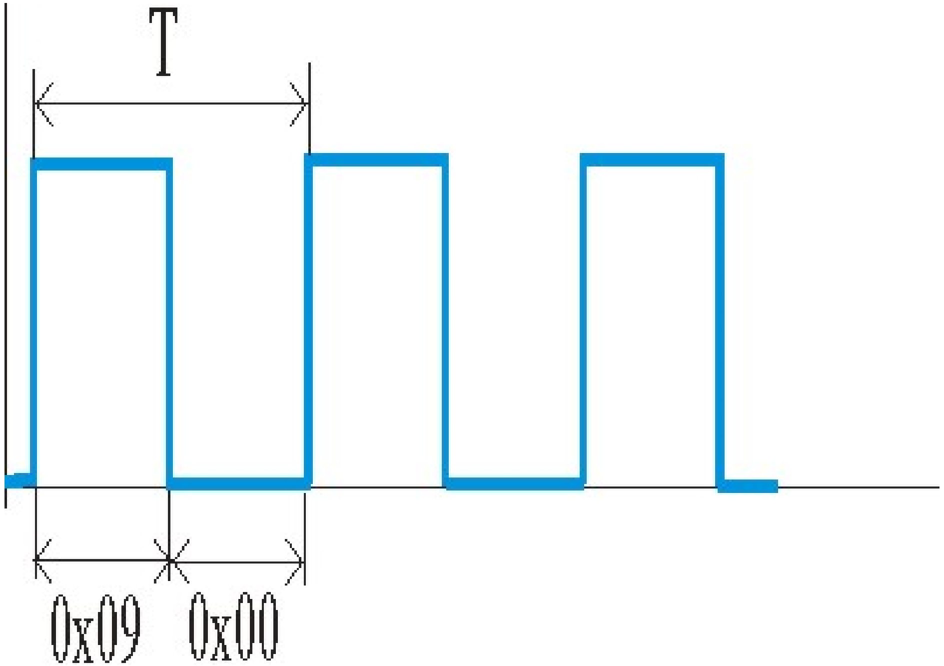
The private members are accessible only by the member functions of the same class. Although they are inherited by derived classes, the member functions declared in derived classes have no access to them.
The protected members are accessible for the member functions of the same class as well as for the member functions of derived classes. They are not accessible for functions outside the current inheritance chain.
The public members are accessible without any restrictions.

In the public inheritance *class ddd : public bbb { …}* the members of the base class keep their access level also in the derived class.
In the protected inheritance *class ddd : protected bbb { …}* the public members of the base class become protected in the derived class, the others keep their access level.
In the private inheritance *class ddd : private bbb { …}* the public and protected members of the base class become private in the derived class, the private members stay private. Consequently, the members of classes derived from *ddd* cannot access members of *bbb*.

DC motor example



In H-bridge:
Rotate forward: L1 & L4 closed,
L2 & L3 open.
Rotate backward: L1 & L2 open,
L3 & L4 closed.
[L1 : L4] are controlled by parallel port bits [0 : 3] (bit 1 means closed).
Output word 0x09 – forward; 0x06 – backward; 0x00 - stopp

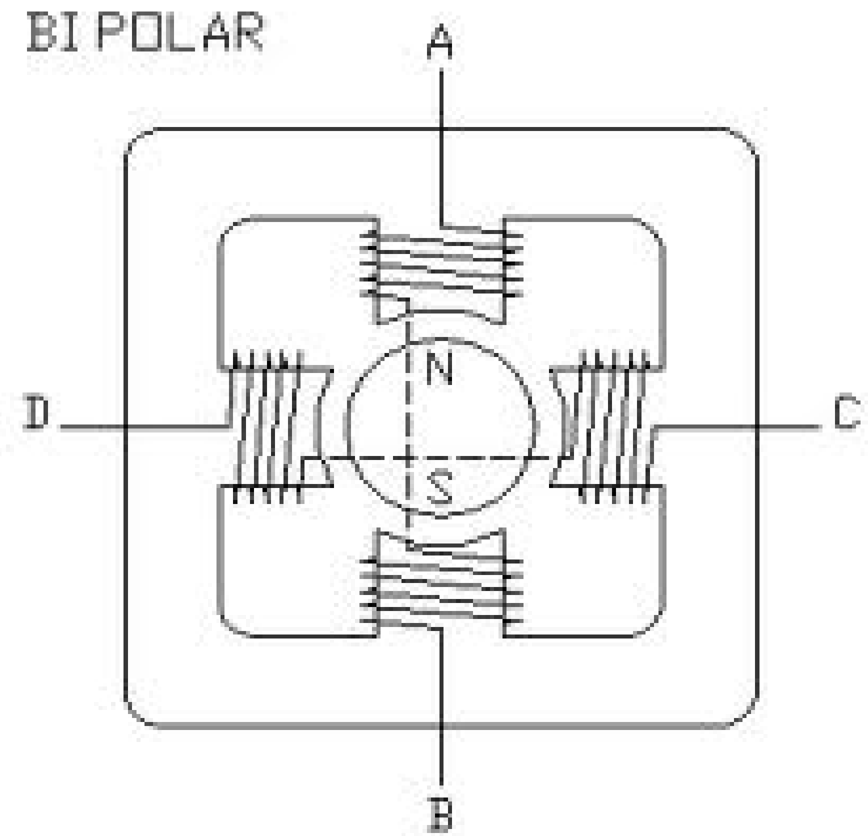


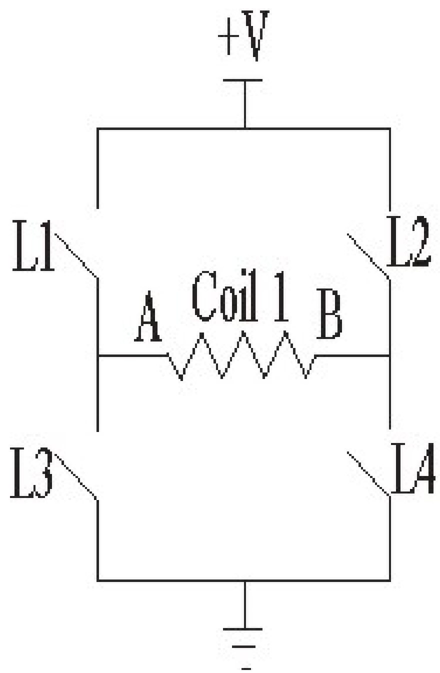
Power supply: sequence of pulses

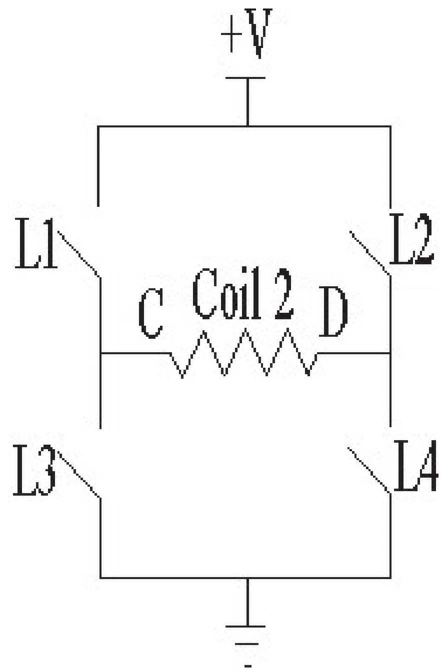
```
while (!StopRequest)
{ // runs in a separate thread
    for (int i = 0;  i < 256;  i++)
        outportb(BASE, i < 128 ? 0x09 :
            x00);
}
```

Rotating forward, half speed.

Bipolar stepper motor example

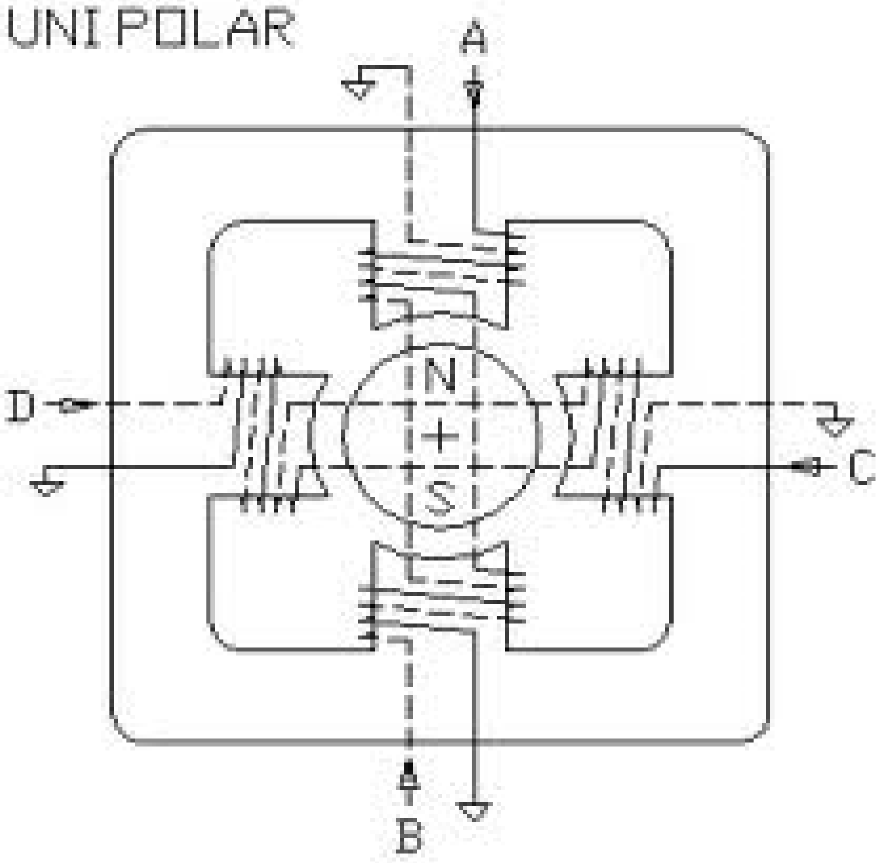


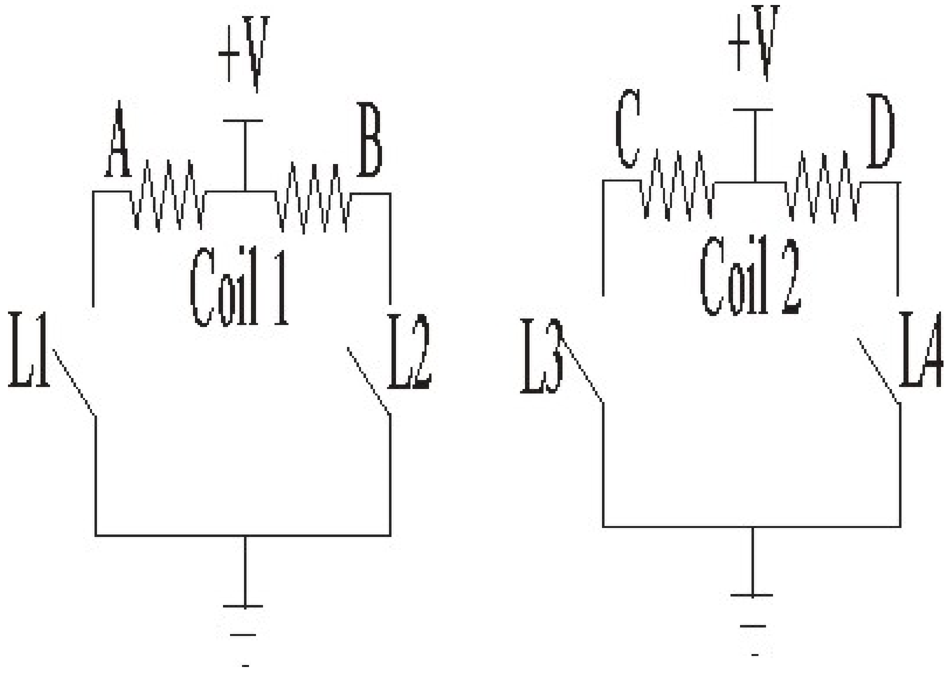




0x09	0x69	0x60	0x66	0x06	0x96	0x90	0x99
45º	90º	135º	180º	225º	270º	315º	360º

Unipolar stepper motor example



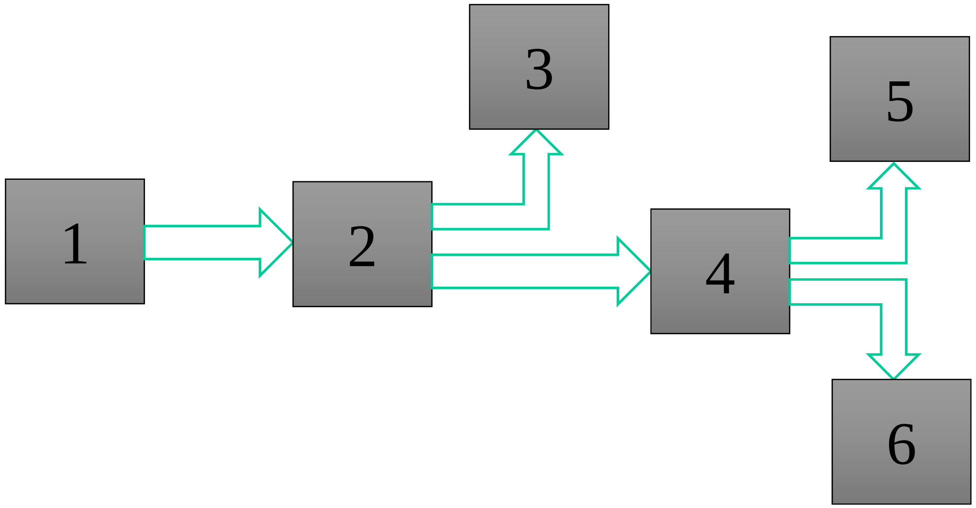


0x11	0x10	0x12	0x02	0x22	0x20	0x21	0x01
45º	90º	135º	180º	225º	270º	315º	360º

Abstract classes

```
class Motor : public ParallelPort
{
  public:   Motor(DWORD a) : ParallelPort(a) {     }
            void Forward() {     }
            void Backward() {     }
            void Off() {   WritePort0(0x00);  }
};
```

The abstract class is a base class concentrating the common features of its successor classes. In the most cases some or even all the methods of an abstract class are empty. C++ allows to declare objects of abstact classes, but usually nobody does it.



1. ParallelPort
2. Motor (abstract)
3. DCMotor
4. StepperMotor (abstract)
5. BipolarStepperMotor
6. UnipolarStepperMotor

Inheritance (4)

```
class DCMotor: public Motor
{
 private:   double m_Speed;
 public:    DCMotor(DWORD a) : Motor(a) { m_Speed = 0; }
            BOOL SetSpeed(double d)  {  if (d < 0 || d > 1)
                                            return FALSE;
                                        m_Speed = d;
                                        return TRUE;  }
            double GetSpeed() {  return m_Speed; }
            void Forward()  {   int i = 0, Limit = (int)(m_Speed * 255);
                                for (; i < 256;  i++)
                                    WritePort0(i < Limit ? 0x09 : 0x00);  }
            void Backward()  { int i = 0, Limit = (int)(m_Speed * 255);
                                for (; i < 256;  i++)
                                    WritePort0(i < Limit ? 0x06 : 0x00);  }
};
```

Inheritance (5)

```
DCMotor *pDCMotor = new DCMotor(0x378);
BOOL volatile Rotate = TRUE;

pDCMotor->SetSpeed(0.5);
while (Rotate)  // polling, Rotate is changed in another thread
    pDCMotor->Forward();
pDCMotor->Off();
```

Virtual functions (1)

```
class StepperMotor: public Motor
{
 private:    int m_Index;
 public:     StepperMotor(DWORD a) : Motor(a) { m_Index = 0; }
 protected: virtual int GetStepWord(int i) {  return 0; }
             void Forward()  {  if (++m_Index >= 8)
                                   m_Index = 0;
                                 WritePort0(GetStepWord(m_Index));  }
             void Backward() {  if (--m_Index < 0)
                                   m_Index = 7;
                                 WritePort0(GetStepWord(m_Index));  }
};
```

Virtual functions (2)

```
class UnipolarStepperMotor : public StepperMotor
{
 private: BYTE m_Words[8];
 public:   UnipolarStepperMotor(DWORD a) : StepperMotor(a)
           { // constructor
             BYTE Words[] = { 0x11, 0x10, 0x12, 0x02, 0x22, 0x20, 0x21, 0x01 };
             memcpy(m_Words, Words, 8);
           }
 protected: virtual int GetStepWord(int i) { return m_Words[i]; }
};

class BipolarStepperMotor : public StepperMotor
{
 private: BYTE m_Words[8];
 public:   BipolarStepperMotor(DWORD a) : StepperMotor(a)
           {// constructor
             BYTE Words[] = { 0x09, 0x69, 0x60, 0x66, 0x06, 0x96, 0x90, 0x99 };
             memcpy(m_Words, Words, 8);
           }
 protected: virtual int GetStepWord(int i) { return m_Words[i]; }
};
```

Virtual functions (3)

```
UnipolarStepperMotor *pUnipolar = new UnipolarStepperMotor(0x378);
pUnipolar->Forward(); // rotate by 45°
```

Forward() inherited from class StepperMotor calls GetStepWord():

```
void Forward()
{  if (++m_Index >= 8)
        m_Index = 0;
   WritePort0(GetStepWord(m_Index));
 }
```

If GetStepWord() is not virtual, early binding is applied: as there is GetStepWord() in class StepperMotor, Forward() always calls GetStepWord() from its own class, i.e. from StepperMotor and GetStepWord() from UnipolarStepperMotor is ignored.

If GetStepWord() is virtual, late binding is applied: Forward() calls GetStepWord() defined in class UnipolarStepperMotor. The late binding means that the selection of function to be called depends on the type of object for which it is called.

```
BipolarStepperMotor *pBipolar = new BipolarStepperMotor(0x378);
pBipolar->Forward();
```

Now Forward() calls GetStepWord() defined in class BipolarStepperMotor.

Virtual functions (4)

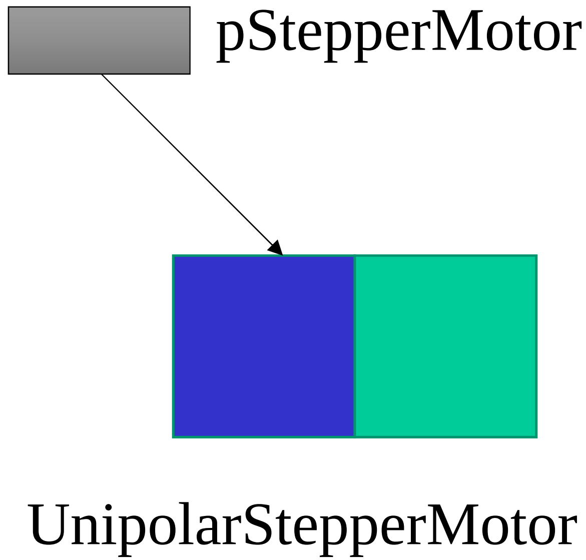
```
#define UNIPOLAR          1
#define BIPOLAR           2

StepperMotor *pStepperMotor;
if (GetConfiguration() == UNIPOLAR)
   pStepperMotor = new UnipolarStepperMotor(0x378);
else
   pStepperMotor = new BipolarStepperMotor(0x378);
pStepperMotor->Forward();
```

We may write:

```
UnipolarStepperMotor Unipolar(0x378);
StepperMotor *pStepper = &Unipolar;
```

But we cannot write:

```
StepperMotor Stepper(0x378);
UnipolarStepperMotor *pUnipolar = &Stepper;
```



Blue – defined in base class StepperMotor, common for all stepper motors
Green – defined only in class UnipolarStepperMotor

Virtual functions (5)

```
class StepperMotor: public Motor
{
 private:    int m_Index;
 public:     StepperMotor(DWORD a) : Motor(a) { m_Index = 0; }
 protected: virtual int GetStepWord(int) = 0; // pure virtual function
             void Forward()  {  if (++m_Index >= 8)
                                    m_Index = 0;
                                 WritePort0(GetStepWord(m_Index));  }
             void Backward() {  if (--m_Index < 0)
                                    m_Index = 7;
                                 WritePort0(GetStepWord(m_Index));  }
};
```

If a class contains pure virtual functions, then:

1. It is not possible to create objects of that class.
2. Classes derived from that class must implement the pure virtual functions as non-pure functions or define them once more as pure functions.

Remark that for example

```
virtual void fun() { }
```

is not a pure virtual function.

Virtual functions (6)

Suppose that classes StepperMotor, UnipolarStepperMotor and BipolarStepperMotor have destructors. Then

```
StepperMotor *pStepperMotor;
if (GetConfiguration() == UNIPOLAR)
   pStepperMotor = new UnipolarStepperMotor(0x378);
else
   pStepperMotor = new BipolarStepperMotor(0x378);
……………………………………………………….
delete pStepperMotor;
```

only the destructor of class StepperMotor is called because we have early binding here. To solve the problem we have to define the destructor as a virtual function:

```
virtual ~StepperMotor();
```

Recommendation: if you know that your class will be used as a base class for derivations, always include into it a virtual destructor. Its body may stay empty.

Struct in C++

For C++, the only difference between struct and class is that by default in struct the members are public and in class private.

```
class ParallelPort
{
  DWORD m_BaseAddr;  // by default private (although the "private" specifier is missing)
  public:   ParallelPort(DWORD); // specified as public
  ……………………………………….
};
struct ParallelPort
{
  DWORD m_BaseAddr;  // by default public
  ParallelPort(DWORD);
  ……………………………………….
};
```

In practice, however, the struct is used for small classes containing only attributes. Or in other words – struct in C++ has the same meaning as in C.

Copy constructor (1)

```
class Date
{
   private: int m_Day,  m_Month, m_Year;
   public: Date(int d, int m, int y) { m_Day = d;  m_Month = m;  m_Year = y; }
……………………………………………………………………
};
void PrintDate(Date d)
{
  printf("%d.%d.%d\n", d.GetDay(), d.GetMonth(), d.GetYear());
}

Date d1(27, 5, 2012);
Date d2 = d1; // default copy constructor, copies attribute by attribute
Date *pd3 = new Date(27, 5, 2012);
Date d4 = *pd3; // de-referencing the pointer and copying
PrintDate(d1);  // copies d1 to d
PrintDate(*pd3); // de-references pd3 and copies to d
```

Turn attention to the following expression:

```
PrintDate(Date(27, 5, 2012)); // contructs an object without name and copies it to d
```

Copy constructor (2)

```
class Date
{
private:   int m_Day;
           char *m_pMonth;
           int m_Year;
public:    Date(int d, const char *mp, int y)
           {
             m_Day = d; m_Year = y;
             m_pMonth = new char[strlen(mp) + 1];   strcpy(m_pMonth, mp);
           }
           ~Date() { delete m_pMonth; }
……………………………………
};
static Date d1(27, “May”, 2012); // global lifetime
Date d2 = d1; // local lifetime
```

As the default copy constructor copies attribute by attribute, the pointers d1.m_pMonth and d2.m_pMonth point to the same place. Consequently, when d2 as a local variable is deleted, d1 looses its attribute “Month”.

Copy constructor (3)

```
class Date
{
private: int m_Day, m_Year;
          char *m_pMonth;
public: Date(int d, const char *mp, int y)
        {
            m_Day = d; m_Year = y;
            m_pMonth = new char[strlen(mp) + 1];   strcpy(m_pMonth, mp);
        }
        Date (const Date &Original)
        {  // overloads the default copy constructor
            m_Day = Original.m_Day;  m_Year = Original.m_Year;
            m_pMonth = new char[strlen(Original.m_pMonth) + 1];
            strcpy(m_pMonth, Original.m_pMonth);
        }
        ~Date() { delete m_pMonth; }
        ………………………………………………………………..
};
Date d2 = d1; // When the copy constructor is working, Original is the synonym of d1;
              // m_Day, m_Year and m_pMonth are the members of d2.
```

Pointer "this"

By default, each class has a member called as "this". It is a pointer which points to the object itself. For example,

```
void ParallelPort::WritePort0(BYTE data)
{
    outportb(m_BaseAddr, data); // equivalent with this ->outportb(m_BaseAddr, data);
 }

class Date
{
    ………………………………………………………………..
    Date (const Date &Original)
    {  // overloads the default copy constructor
        *this = Original; // At first copy everything, then correct
                          // Default assignment operator is applied
        m_pMonth = new char[strlen(Original.m_pMonth) + 1];
        strcpy(m_pMonth, Original.m_pMonth);
    }
    …………………………………………………………………..
};
```

Constant objects

```
class ParallelPort
{
  private:  DWORD m_BaseAddr;
  public:   ParallelPort(DWORD a ) {  m_BaseAddr = a; }
            void SetBaseAddr(DWORD a ) {  m_BaseAddr = a; }
            DWORD GetBaseAddr() {  return m_BaseAddr;  }
            ………………………………………….
};

const ParallelPort Port1(0x378);
Port1.SetBaseAddr(0x3BC);  // error
printf("%lu\n", Port1.GetBaseAddr()); // also error
```

Solution:

```
DWORD GetBaseAddr() const {  return m_BaseAddr;  }
printf("%lu\n", Port1.GetBaseAddr()); // now OK
```

Friends (1)

```
class Date
{
   private: int m_Day,  m_Month, m_Year;
   public: Date(int d, int m, int y) { m_Day = d;  m_Month = m;  m_Year = y; }
               Date() { }
               void SetDate(int d, int m, int y) { m_Day = d;  m_Month = m;  m_Year = y; }
               void GetDate(int *pd, int *pm, int *py) const
                              { *pd = m_Day; *pm = m_Month;  *py = m_Year; }
   friend Timestamp; // class Date declares that class Timestamp is its friend class
};
class Time
{
   private: int m_Sec,  m_Min, m_Hour;
   public: Time(int h, int m, int s) { m_Sec = s;  m_Min = m;  m_Hour = h; }
               Time() { }
               void SetTime(int h, int m, int s) {m_Sec = s;  m_Min = m;  m_Hour = h; }
               void GetTime(int *ph, int *pm, int *ps) const
                              { *ph = m_Hour; *pm = m_Min;  *ps = m_Sec; }
   friend Timestamp; // class Time declares that class Timestamp is its friend class
};
```

Friends (2)

```
class Timestamp
{
   private: Date m_Date;
            Time m_Time;
   public:  Timestamp(Date d, Time t);
            void PrintTimestamp();
……………………………………….
};
void Timestamp::PrintTimestamp()
{ // due to friendship has access to the Date and Time private attributes
  printf("%d.%d.%d %d:%d:%d\n", m_Date.m_Day, m_Date.m_Month, m_Date.m_Year,
  m_Time.mHour, m_Time.m_Min, m_Time.m_Sec);
}
```

If class A declares that class B is its friend, class B has free access to all the members of class A. But it does not mean that A can also access non-public members of B. Here classes Time and Date allow class Timestamp to work with its private attributes. As Timestamp has not declared friendship with Time and Date, those classes have no free access to Timestamp private and protected members.

Friendship is not inherited. Also, if B declares that C is its friend, C has access to non-public members of B but not to non-public members of A.

Friends (3)

```
class Date
{
   private: int m_Day,  m_Month, m_Year;
   public: Date(int d, int m, int y) { m_Day = d;  m_Month = m;  m_Year = y; }
               Date() { }
               void SetDate(int d, int m, int y) { m_Day = d;  m_Month = m;  m_Year = y; }
               void GetDate(int *pd, int *pm, int *py) const
               { *pd = m_Day; *pm = m_Month;  *py = m_Year; }
   friend voidTimestamp::PrintTimestamp();
  // Only PrintTimestamp() from Timestamp will have access to private members of Date
   friend void ::PrintDate(Date *);
 // Also, function PrintDate not belonging to classes will have access to private members
 // of Date
};

void PrintDate(Date *pd)
{
  printf("%d.%d.%d \n", pd->m_Day, pd-> m_Month, pd-> m_Year);
}
```

Operator overloading (1)

```
class complex
{
  public:  double m_Re, m_Im; // real part and imaginary part
             complex(double d1 = 0, double d2 = 0) { m_Re = d1;  m_Im = d2; }
             complex operator+(complex &c)
                        { return complex (m_Re + c.m_Re, m_Im + c.m_Im); }
             int operator==(complex &c)
                        { return m_Re == c.m_Re && m_Im == c.m_Im ? 1 : 0; }
             complex operator!() { return complex(m_Re, -m_Im); }
             ………………………………………..
};

complex x(5, 6), y(1,2); // x = 5 + j6, y = 1 + j2
complex z1 = x + y; // Actually z1 = x.operator+(y);  we get z = 6 + j8.
            // When the operator method is working, c is the synonym of y; m_Re and
            // m_Im are the members of x.  The return value is a new nameless
            // complex  number. From it the default copy constructor creates z1.
if (x == y) // actually x.operator==(y)
   printf("Equal\n");
complex z2 = !x;  // actually z2 = x.operator!(); we get z2 = 5 - j6 (conjugate of x)
```

Operator overloading (2)

```
class complex
{
  public:  double m_Re, m_Im;
           complex(double d1 = 0, double d2 = 0) { m_Re = d1;  m_Im = d2; }
           friend complex operator+(complex &, complex &);
           friend int operator==(complex &, complex &);
           friend complex operator!(complex &);
           ………………………………………..
};

complex x(5, 6), y(1,2); // x = 5 + j6, y = 1 + j2
complex z1 = x + y; // actually z1 = operator+(x, y);  we get z1 = 6 + j8
if (x == y) // actually operator==(x, y)
   printf("Equal\n");
complex z2 = !x;  // actually z2 = operator!(x); we get z2 = 5 - j6
```

Operator overloading (3)

```
complex operator+(complex &a, complex &b)
{
  return complex(a.m_Re + b.m_Re, a.m_Im + b.m_Im);
}

int operator==(complex &a, complex &b)
{
  if (a.m_Re == b.m_Re && a.m_Im == b.m_Im)
      return 1;
  else
      return 0;
}

complex operator!(complex &a)
{
  return complex(a.m_Re, -a.m_Im);
}
```

Operator overloading (4)

It is not possible to:

1. Introduce new operators not specified in C++ standard
2. Change the priorities
3. Overload the sizeof operator, the scope resolution operator (::), the conditional operator (?:) and the member selection operator (.)

Overloading of operators like new, delete, function call (( )), array element reference ([ ]), comma (,), assignment (=) and type cast may be tricky.

```
class Date
{
  private: int m_Day,  m_Month, m_Year;
  public: Date(int d = 0, int m = 0, int y = 0) { m_Day = d;  m_Month = m;  m_Year = y; }
……………………………………………………………………
};
```

```
Date d1(20, 10, 2012); // constructor called
Date d2 = d1; // default copy constructor called
Date d3; // constructor called, default arguments are used
d3 = d1; // here we need operator overloading function for assignment
```

Each class has default assignment overloading function providing bitwise copy

Operator overloading (5)

```
class Date
{
private: int m_Day, m_Year;
         char *m_pMonth;
public:  Date &operator =(const Date &Right)  // & - specifies the reference type
      {
       if (this == &Right) // & - address operator
         return *this; // necessary for expressions like d1 = *pd where pd points to d1
        m_Day = Right.m_Day;  m_Year = Right.m_Year;
        if (m_pMonth) delete m_pMonth;
        m_pMonth = new char[strlen(Right.m_pMonth) + 1];
        strcpy(m_pMonth, Right.m_pMonth);
        return *this;
       }
……………………………………
};
d1 = d2; // actually d1.operator=(d2); i.e. this points to d1; Right is the synonym of d2
d1 = d2 = d3; // d1 = d2.operator=(d3) ➔d1.operator=(d2.operator=(d3));
```

Therefore void Date::operator=(Date &Right) {…} does not work – the operator= function must return the object.

Operator overloading (6)

```
class Date
{
private:    int m_Day;
            char *m_pMonth;
            int m_Year;
            char m_Buf[100];
public:     operator char *()    // no return value, the word "operator" is followed
                                 // by the new type specifier
            {
               sprintf(m_Buf, "%d %s %d", m_Day, m_pMonth, m_Year);
               return m_Buf;
            }
……………………………………
};
Date d (27, "May", 2012);
if (strcmp(d, "28 June 2013")) // actually the operator char *() function is called
   printf("Not match\n");
```

Operator overloading (7)

In class LinkedList:

```
Node *operator[](int);

Node *LinkedList::operator[](int i)
{
  if (i < 0 || i > m_nRecords - 1)
    return NULL;
  Node *pTemp = m_pHead;
  for (int j = 0;  j != i;  j++)
    pTemp = pTemp->GetNext();
  return pTemp;
}

LinkedList Names;
```

`Names[i]` is handled as `Names.operator[](i)` and it gives us the pointer to the i-th node. Consequently

```
Names[i]->GetRecord();
```

gives us the record associated with the i-th node.

Operator overloading (8)

LinkedList *pNames;

(*pNames)[3] is handled as (*pNames).operator[](3). It returns the pointer to the third node. Asterisk (*) here is the dereference operator!
The record associated with this node is expressed as
(*pNames [3])->GetRecord()

If the record is a string, we have to convert the result to TCHAR *:
(TCHAR *)(((*pNames)[3])->GetRecord());

To print this string:
_tprintf(_T("Got %s\n"), (TCHAR *)(((*pNames)[3])->GetRecord()));

Static members(1)

```
class Base
{
public:     static int m_Counter;   // declaration, but definition is also needed
            Base() { m_Counter++; }
            ~Base() { m_Counter--; }
};

int Base::m_Counter=10; // definition, must be outside of functions and class declarations

Base b;
printf("%d\n", b.m_Counter);  // may be error
printf("%d\n", Base:: m_Counter); // correct
```

Static attributes get memory only once. They are shared between all the objects of that class and also objects of classes derived from that class. The static attributes exist even when there are no any objects defined yet.

Static members(2)

```
class Base
{
public:     static int m_Counter;
            Base() { m_Counter++; }
            ~Base() { m_Counter--; }
};

class Derived_1 : public Base {………………..};
……………………………………………………..
class Derived_n : public Base {………………..};
Derived_i di;
printf("%d\n", di.m_Counter); // may be error
printf("%d\n", Derived_i:: m_Counter); // correct
```

Here m_Counter presents the current total number of objects of class Base plus objects of classes Derived_1…Derived_n.

Static members(3)

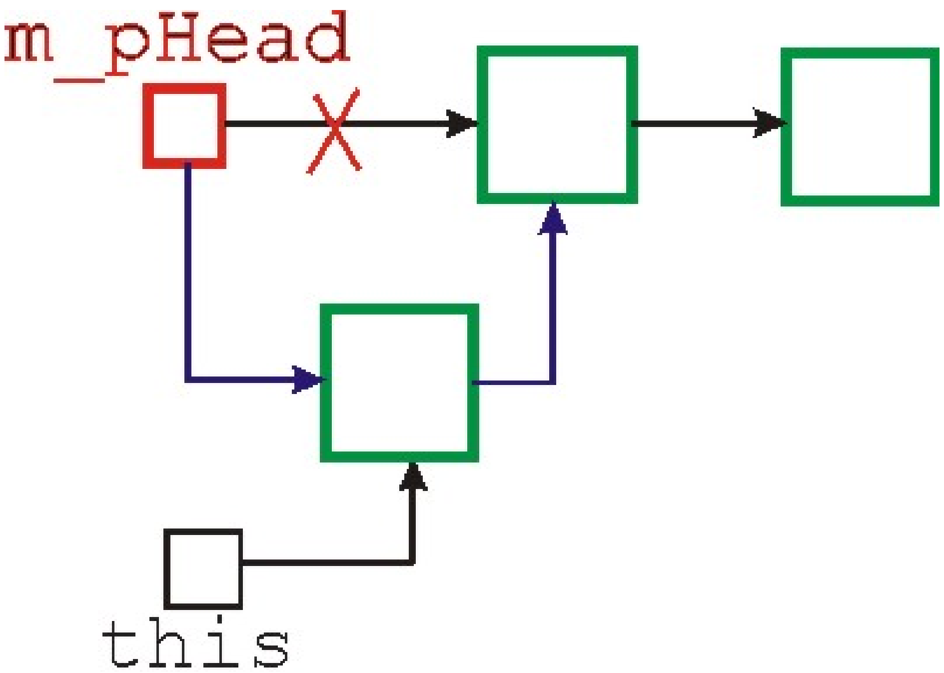
```
class Base
{
private:   static int m_Counter;
public:     Base() { m_Counter++; }
           ~Base() { m_Counter--; }
           static int GetCounter() { return m_Counter; }
};
int Base::m_Counter=0; // although private
class Derived_1 : public Base {………………..};
……………………………………………………..
class Derived_n : public Base {………………..};
Derived_i di;
printf("%d\n", di.GetCounter()); // correct
printf("%d\n", Derived_i:: GetCounter()); // correct
```

Static functions of a class cannot operate with non-static members of that class. They can be called even when there are no any objects defined yet.

All the non-static functions have access to any of the static members, the restrictions depend only on the access specifiers (public, private, protected).

Static members(4)

```
class Base
{
private:   static Base *m_pHead;
           Base *m_pNext;
public:     Base();
           ~Base();
           static Base *GetHead() { return m_pHead; }
           Base *GetNext() { return m_pNext; }
};
```

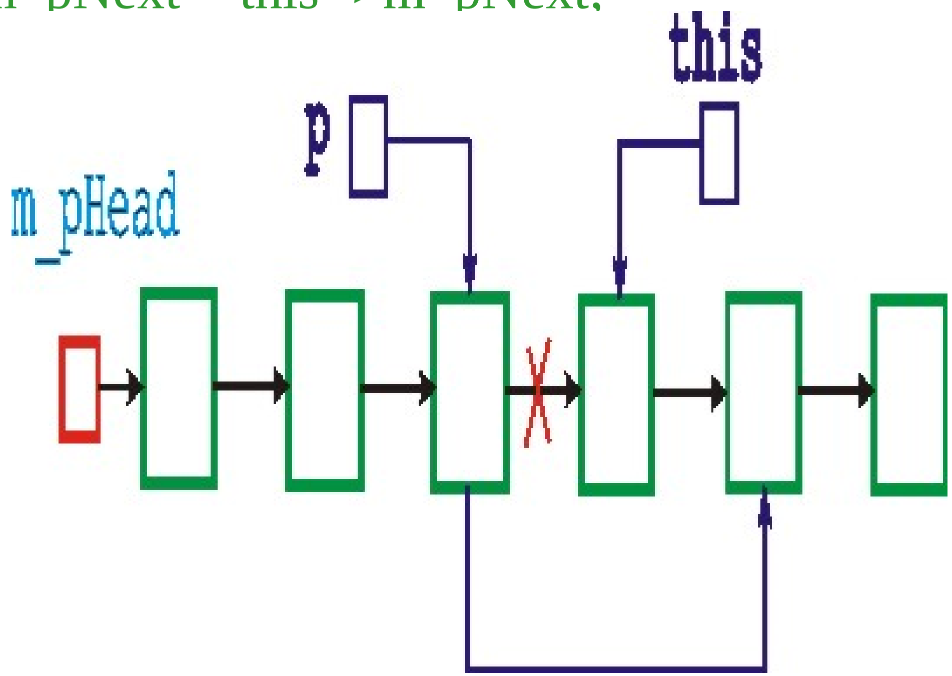



```
Base Base::m_pHead=NULL;

Base::Base()
{// All the objects of classes derived from
  // base will be automatically inserted into
  // list
  if (!m_pHead)
      m_pNext = NULL;
  else
      this->m_pNext = m_pHead;
   m_pHead = this;
}
```

Static members(5)

```
Base::~Base()
{
  if (this == m_pHead)
      m_pHead = m_pNext;
  else
  {
    Base *p;
    for (p = m_pHead;  p->m_pNext != this;  p = p->m_pNext);
    p->m_pNext = this ->m_pNext;
  }
}
```

When an object is destroyed, it will also automatically removed from the list.



Exceptions (1)

```
class Time
{
  private: int m_Hour, m_Min, m_Sec;
  public:  Time(int h, int m, int s)
             {
               if (h < 0 || h > 23) throw _T("Hours wrong\n"); // jump to the catch block
               m_Hour = h;
               if (m < 0 || m > 59) throw _T("Minutes wrong\n"); // jump to the catch block
               m_Min = m;
               if (s < 0 || s > 59) throw _T("Seconds wrong\n"); // jump to the catch block
               m_Sec = s;
             }
…………………………………………………….
};

Time *pTime = NULL;
try { pTime = new Time(25, 23, 23); }
catch (TCHAR * pText) { _tprintf(pText); }
```

The Microsoft structured exception handling (keywords __try, __except, __finally) is not a part of C++ standard.

Exceptions (2)

```
class Time
{
  private: int m_Hour, m_Min, m_Sec;
  public:  Time(int h, int m, int s)
               {
                  if (h < 0 || h > 23) throw 1;
                  m_Hour = h;
                  if (m < 0 || m > 59) throw 2;
                  m_Min = m;
                  if (s < 0 || s > 59) throw 3;
                  m_Sec = s;
               }
…………………………………………………………….
};

Time *pTime = NULL;
try { pTime = new Time(25, 23, 23); }
catch (int ErrorCode) { _tprintf(_T("Object not created, error %d\n"), ErrorCode); }
```

Exceptions (3)

```
void LinkedList::Insert(void *p, int i)
{
  if (i < 0) throw i; // throws integer
  if (!p) throw p; // throws pointer
  …………………………..
}

LinkedList *pList;
try { pList->Insert(_T("Hello"), 10); }
catch(int Index) { _tprintf(_T("Failure, index is %d\n"), Index); } // catches integers
catch(void *pObject) { _tprintf(_T("Failure, no object")); } // catches pointers

try { pList->Insert(_T(“Good bye”), 20); ); }
catch(…) // Handles any exceptions. If several catches, catch(…) must be the last
{
  _tprintf(_T(“Failure, wrong input parameters”));
}
```

Namespaces (1)

File ParallelPort.h:

```
namespace ControlSystem
{ class ParallelPort {…..}; } // semicolon not needed
```

File Motors.h:

```
namespace ControlSystem
{ class Motor: public ParallelPort  {…..};
  class StepperMotor: public Motor {…..};
  class UnipolarStepperMotor : public StepperMotor {…..};
}
```

File ControlSystem.cpp:

```
namespace ControlSystem
{
      void SomeFunction()
      {
       UnipolarStepperMotor *pMotor = new UnipolarStepperMotor(0x378);
      // The complete name is ControlSystem::UnipolarStepperMotor, but here
      // we know that class UnipolarStepperMotor belongs to the same namespace
      ………………………………….
      }
}
```

Namespaces (2)

```
namespace ControlSystem
{
    void SomeFunction()
    {
     std::wstring abc(_T("ABC")); // create object abc of standard class wstring
    // complete name is obligatory as wstring belongs to namespace std and
    // SomeFunction() to namespace ControlSystem
    ………………………………………………
    }
}
```

Namespace not specified - i.e. we are in the global namespace

```
int _tmain(int argc, _TCHAR* argv[])
{
 std::wstring abc(_T("ABC"));
// complete name is obligatory as wstring belongs to namespace std and main() to
// the global namespace
…………………………………………
}
```

Namespaces (3)

```
using namespace ControlSystem;
using namespace std;
int _tmain(int argc, _TCHAR* argv[]) // _tmain is in the global namespace
{
  wstring abc(_T("ABC")); // now the class wstring is searched first from the current
// (i.e. global) namespace, then from namespace ControlSystem and at last from
// namespace std.
……………………………………………
}
namespace ControlSystem
{ // nested namespaces
   namespace Motors
   {
      class Motor: public ParallelPort  {…..};  // ControlSystem::Motors::Motor
      ……………………………………………
    }
   namespace Convertors
   {
     class DAC: public ParallelPort  {…..}; // ControlSystem::Convertors::DAC
   }
}
```

Templates (1)

```
class Array
{
protected: int m_Size,  *m_pArray;
public:     Array(int n) { m_Size = n;  m_pArray = new int[n]; }
            virtual ~Array()  { delete m_pArray; }
            int GetSize() { return m_Size; }
            int Get(int);
            void Set(int, int);
};

int Array::Get(int i)
{
  if (i < 0 || i > m_Size - 1)  throw _T("Illegal index");
  else return *(m_pArray + i);
}
void Array::Set(int Value, int i)
{
  if (i < 0 || i > m_Size - 1) throw _T("Illegal index");
  else *(m_pArray + i) = Value;
}
```

Templates (2)

Generic programming: how to write class Array so that one of the users could apply it as a container of double numbers, another user for storing of pointers to strings, etc.

The class template defines a class where the types of some attributes, methods and/or parameters of methods are specified as parameters.

```
template<typename T> class Array // deprecated: template<class T>
{ // template<typename T> is the template specifier.
  // Word "Array" here is the class template (not the class) name.
  // T is the placeholder for actual types like int, double, etc
protected: int m_Size;
           T *m_pArray;
public:    Array(int n) { m_Size = n;  m_pArray = new T[n]; }
           virtual ~Array()  { delete m_pArray; }
           int GetSize() { return m_Size; }
           T Get(int);
           void Set(T, int);
};
```

Templates (3)

```
template<typename T> T Array<T>::Get(int i)
{// template<typename T> is the template specifier, it says that we have a template,
 // not a traditional class
 // Array<T> refers to class template with parameter T and name Array
 // Name Array without following to it <T> is meaningless
 // T Array<T>::Get(int i) means that Get() is a member function of class template
 // Array<T> and T is the type of Get() return value.
 if (i < 0 || i > m_Size - 1)  throw _T("Illegal index");
 else return *(m_pArray + i);
}

template<typename T> void Array<T>::Set(T Value, int i)
{
 if (i < 0 || i > m_Size - 1) throw _T("Illegal index");
 else *(m_pArray + i) = Value;
}
```

Templates (4)

```
int _tmain(int argc, _TCHAR* argv[])
{
 Array<int> IntArr(10); // instantiating the template, other examples:
// Array<DCMotor> *pDCMotorArr = new Array<DCMotor>(3);
// Array<TCHAR *> StringArr(100);
  try
  {
    for (int i = 0;  i < 10;  i++)
          IntArr.Set(i, i);
    _tprintf(_T("%d\n"), IntArr.Get(5));
  }
  catch(TCHAR *pMsg)
  {
    _tprintf(pMsg);
  }
  return 0;
}
```

Important: the compiler checks the template code syntax, but does not compile it. The compiling is performed when the actual type is specified. Therefore, in the example above the compiler needs the complete code of template Array<T>.

Templates (5)

```
template<typename T> class Array
{
……………………………………………………..
  Array<T>(const Array<T> &Original)
  { // copy constructor
    m_Size = Original.m_Size;
    m_pArray = new T[m_Size];
    memcpy(m_pArray, Original.m_pArray, sizeof(T) * m_Size);
  }
  Array<T> &operator=(const Array<T> &Right)
  { // overloading =
    m_Size = Right.m_Size;
    delete m_pArray;
    m_pArray = new T[m_Size];
    memcpy(m_pArray, Right.m_pArray, sizeof(T) * m_Size);
    return *this;
  }
…………………………………………………
}:
```

Templates (6)

```
template<typename T, int SIZE> class Array
{ // non-type parameters can only be integrals (char, int, etc.), pointers and references
protected: T *m_pArray;
public:    Array() { m_pArray = new T[SIZE]; } // constructor
           Array<T, SIZE>(const Array<T, SIZE> &Original) // copy constructor
           { m_pArray = new T[SIZE];
             memcpy(m_pArray, Original.m_pArray, sizeof(T) * SIZE); }
           virtual ~Array()  { delete m_pArray; } // destructor
           Array<T, SIZE> &operator=(const Array<T, SIZE> &Right) // overloading =
           { memcpy(m_pArray, Right.m_pArray, sizeof(T) * SIZE);
             return *this; }
           int GetSize() { return SIZE; } // get the number of elements
           T Get(int i) // get an element
           { if (i < 0 || i > SIZE) throw _T("Illegal index");
             else return *(m_pArray + i); }
           void Set(T Value, int i) // set value to an element
           { if (i < 0 || i > SIZE) throw _T("Illegal index");
             else *(m_pArray + i) = Value; }
};
// Array<int, 10> IntArr; // array of integers, the length is always 10
```

Templates (7)

C++ supports also templates for functions:

```
template<typename T> T Larger(T a, T b)
{
	return a > b ? a : b;
}
```

Usage:

```
double x, y, z;
z = Larger<double>(x,y);
```

The functions is applicable for types for which the "greater than" operation is defined.

The arguments and return values may be from different types:

```
template <typename T1, typename T2, typename T3> void Fun(T1 a, T2 b, T3 c)
{
…………….
}
```

Usage:

```
double x, y;
int i;
Fun<double, int, double>(x, i, y);
```

New variable types (1)

In C any variable of any type is interpreted as false if its value is zero and as true if its value is not zero. This is still true in C++.

To improve the readability of code, preprocessor definitions like

```
#define TRUE  1
#define FALSE 0
```

are used. In C++ there is an additional built-in type: bool

```
bool b1 = true, b2 = false;
```

Actually, b1 is stored as integer 1 and b2 as integer 0. Boolean variables are implicitly (i.e. automatically) converted into integers and vice versa:

```
int i = b1; // i is now 1
b1 = 10; // b1 is now true
```

Examples of usage:

```
while (b1 == true) {……}
while (b1) {……}
while (!b2) {……}
bool fun()
{ …..
  return true; }
```

New variable types (2)

Length depends on the implementation of compiler:

long long int ll; // introduced in C++ v11, in Visual Studio 64 bits
unsigned long long int ull; // introduced in C++ v11, in Visual Studio 64 bits
wchar_t wct; // in Visual Studio 16 bits
long double ld; // in Visual Studio 64 bits, i.e. the same as double

Length is specified in standard:

char16_t c16; // introduced in C++ v11, 16-bit character
char32_t c32; // introduced in C++ v11, 32-bit character

Additional built-in types defined by Microsoft:

__int8 i8; // 8-bit integer
__int16 i16; // 16-bit integer
__int32 i32; // 32-bit integer
__int64 i64; // 64-bit integer

Visual Studio does not support 128 bit variables.

New variable types (3)

In C and C++ prior version 11 keyword auto meant that the variable has automatic duration (i.e. it will be created and destroyed automatically):

```
auto int i; // "auto" was almost always omitted
```

In C++ v11 and later keyword auto means that the compiler has to deduct the actual type:

```
auto i = 10; // i is of type int
auto j = 10L; // j is of type long int
auto k; // error -  compiler is unable to deduct the type
```

```
template <typename T1, typename T2> void Fun(T1 a, T2 b)
{
      auto c = a + b;
      ………………..
}
```

If T1 and T2 are both int, c is also int. But if T1 is double and T2 is int, c is double. Consequently, when writing the code, we do not know the type of c and therefore using the auto deduction is the only way out.

New variable types (4)

Pointer that points to nothing has value 0:

```
char *p = 0;
```

Rather often:

```
#define NULL 0
char *p = NULL;

void fun(char *p) {……….}
void fun(int i) {……….}
```

Problem:

```
fun(0); // as 0 is an integer, always the second function is called
```

Solution:

```
fun(nullptr); // the first function is called
fun(0); // the second function is called
```

nullptr is introduced in C++ v 11. Advised to use instead 0 when working with pointers.

Casts (1)

The traditional explicit C cast (*new type) expression* is still in use:

double d 5.6;
int i;
i = (int)d;

C++ standard has defined 4 new casting operators:

static_cast <*new type*> (*expression*)
dynamic_cast <*new type*> (*expression*)
reinterpret_cast <*new type*> (*expression*)
const_cast <*new type*> (*expression*)

The C-style cast is suitable for conversions between primitive data types. For conversions between pointers the C++ casting operators are preferred.

Generally, the static, reinterpret ja const casts do the same as the C-style cast but allow more control over how the conversion should be performed. They are also easier to find in the source code.

Dynamic cast correctness is checked during run-time.

Casts (2)

The static cast checks a bit more that C-style cast and is therefore more secure.

```
double d = 5.6;
int i;
i = static_cast<int>(d) // the same as i = (int)d;

class Base {…..};
class Derived : public Base { …..};
Derived *pd = new Derived;
Base *pb = pd; // implicit cast
pd = pb; // compile error, implicit cast not allowed
pd = (Derived *)pb; // legal, but also a possible source of run-time errors
pd = static_cast<Derived *>(pb); // legal and possible source of run-time errors

int *pi;
double *pd = (double *)pi; // legal, but also a source of run-time errors
*pd = 5.5; // writes 8 bytes to four-byte field, run-time error
double *pd = static_cast<double *>(pi); // compile error
```

The static cast checks whether the pointer and pointee data types are compatible.

Casts (3)

The reinterpret cast checks nothing and allows to cast a pointer to any other type of pointer (exactly as C-style cast).

```
int *pi;
double *pd = reinterpret_cast<double *>(pi); // legal
```

Using the reinterpret cast instead of C-style cast the programmer emphasizes that he knows about the possible risks. If the program crashes, the reinterpret casts are good start points for searching the bugs.

The const cast is used to convert a constant to non-constant.

```
void proc (char *); // a third-party function we have to use
void fun (const char *p)
{  // our function, by specification its argument must be const char *
  ……………………
  proc(const_cast<char *>(p));
  ……………………..
}
```

Casts (4)

The dynamic cast provides pointers run-time check (not compile-time as the other casts) on casts within an inheritance hierarchy.

```
class Base
{
  virtual void base_fun(); // the hierarchy must contain at least one virtual method
   ………………..
};
class Derived : public Base { …..};
Derived *pd = new Derived;
pd = static_cast<Derived *>(pb); // legal and possible source of run-time errors
pd = dynamic_cast<Derived *>(pb); // no compile error but when the program
                                   // runs, the result is null-pointer
if (!pd)
{
   ………………….
}
pb = dynamic_cast<Base *>(pd); // legal, no any errorrs
```

If the hierarchy does not contain virtual functions, a compile error will follw.

Initializing (1)

```
class ParallelPort
{
  private: DWORD m_BaseAddr;
  public:  ParallelPort(DWORD a) { m_BaseAddr = a; }
           ParallelPort() { m_BaseAddr = 0x378; }
  …………………………..
 };
```

Starting from C++ v 11, the member variables may be initialized directly in the class definition:

```
class ParallelPort
{
  private: DWORD m_BaseAddr = 0x378;
  public:  ParallelPort(DWORD a) { m_BaseAddr = a; }
           ParallelPort() { m_BaseAddr = 0x378; } // now not needed
  …………………………..
 };
```

```
ParallelPort *p1 = new ParallelPort; // m_BaseAddr is set to 0x378
ParallelPort *p2 = new ParallelPort(0x3BC); // m_BaseAddr is overwritten to 0x3BC
```

Initializing (2)

```
class Motor : public ParallelPort
{
   private: double m_Speed;
   public:  Motor(DWORD a, double s)  :  ParallelPort(a) { m_Speed = s; }
                  ………………………..
 };
```

Constructor with initializer list:

```
Motor(DWORD a, double s)  :  ParallelPort(a), m_Speed(s) { }
```

Usage:

```
Motor *pMotor = new Motor(0x3BC, 0.75);
```

Attributes specified as references must be always initialized by initializer list.

C++ standard library

Standard classes for:

- Input and output
- String processing
- Exception handling
- Operating with containers (vectors, linked lists, etc.)
- Clocks and timers
- Multithreading
- Threads synchronization
- Random numbers
- Complex numbers
- Internationalization
- …………………………………..

I/O streams (1)

To stdout: printf, wprintf, _tprintf

```
printf("%d\n", i);
wprintf (L"%d\n", i);
_tprintf (_T("%d\n"), i);
```

To a stream: fprintf(FILE *stream, *printf parameters*); fwprintf, _ftprintf

```
fprintf(stderr, "%d\n", i);
fwprintf(stderr, L"%d\n", i);
_ftprintf(stderr, _T("%d\n"), i);
```

To a memory field: sprintf(pointer *, *printf parameters*); swprintf, _stprintf

```
char pc[100];
sprintf(pc, "%d\n", i);
wchar_t pwc[100];
swprintf(pwc, L"%d\n", i);
TCHAR ptc[100];
_stprintf(ptc, _T("%d\n"), i);
```

I/O streams (2)

#include <iostream> // obligatory

#include <iomanip> // may be needed for manipulators

cin – global object of class istream, cout and cerr – global objects of class ostream.

Operator overloading functions operator<< and operator>>

Basic types like char, char *, int, double

Manipulators, for example endl, hex, dec, setw(n), setprecision(n)

Methods, for example write(), put(), flush()

int i = 10, j = 20; double d = 3.14159; char *p = "abc";

cout << i; // printf("%d", i);

cout << i << ' ' << d << ' '<< p << endl; // printf("%d %lg %s\n", i, d, p);

// possible because the return value of operator<< is ostream&

cout << i << endl << d << endl << p << endl; // printf("%d\n %lg\n %s\n", i, d, p);

cout << 10 << ' ' << 3.14 << ' ' << "abc" << endl; // printf("%d %lg abc\n", 10, 3.14);

cerr << "Unable to open file" << endl; // fprintf(stderr, "Unable to open file\n");

cout << hex << i << endl; // printf("%x\n", i); hex stays valid until manipulator dec

cout << setw(6) << i << ' ' << j << endl; // printf("%6d %d\n", i, j);

cout << setprecision(4) << d << endl; // printf("%.4lg\n", d); we get 3.142

cout.write(p, 2); // prints the first 2 characters

cout.put(*p); // prints one character

I/O streams (3)

Use cout in case of ASCII strings, wcout in case of Unicode strings:

```
int i = 10, j = 20; double d = 3.14159; wchar_t *p = L"abc";
wcout << i << L' ' << d << L' '<< p << endl;
```

_tcout is not defined but everybody can do it himself / herself:

```
#if defined(UNICODE) || defined(_UNICODE)
#define _tcout std::wcout
#else
#define _tcout std::cout
#endif                    // Do not forget to include this definition into your homework
int i = 10, j = 12; double d = 3.14159; TCHAR *p = _T("abc");
_tcout << i << _T(' ') << d << _T(' ') << p << endl;
```

```
int n;
string name; // object of class string
cout << "Enter the number of arguments: ";
cin >> n;
cout << "Enter the name: ";
cin >> name;
```

I/O streams (4)

```
class ParallelPort
{
  private:  DWORD m_BaseAddr;
  public:   ParallelPort(DWORD); // constructor 1
            ParallelPort(); // constructor 2
            friend ostream &operator<<(ostream &, const ParallelPort &);
             ...........................................................
};

ostream &operator<<(ostream &ostr, const ParallelPort &pp)
{
  ostr << "Base address is 0x"  << hex << pp. m_BaseAddr  << dec << endl;
  return ostr;
}

ParallelPort pPort = new ParallelPort(0x378);
cout << *pPort;
```

I/O streams (5)

```
#include <fstream>
fstream File; // File is an object of class fstream
File.open(_T("c:\\temp\\data.bin"), fstream::out | fstream::in);
```

fstream modes:

- app - Set the stream position indicator to the end of the stream before each output operation.
- ate - Set the stream position indicator to the end of the stream on opening.
- binary - Consider stream as binary rather than text.
- in - Allow input operations on the stream.
- out - Allow output operations on the stream.
- trunc – Discard the current content, assume that on opening the file is empty.

Filename: const char *, in Windows also const wchar_t *.

```
int iArray[10], i;
for (i = 0;  i < 10;  i++)
     File << iArray [i];  // writes into file
for (i = 0;  i < 10;  i++)
     File >> iArray [i];  // reads from file

File.close();
```

I/O streams (6)

```
File.write((const char*) &iArray [0], sizeof iArray); // write a block of data
File.read((char*) &iArray [0], sizeof iArray); // read a block of data
```

Shift the cursor marking the position for writing:

```
int n;
File.seekp(ios_base::beg + n); // n bytes from the beginning
File.seekp(ios_base::end - n); // n bytes before the end
File.seekp(ios_base::cur + n); // n bytes after the current position
File.seekp(ios_base::cur - n); // n bytes before the current position
n = File.tellp(); // returns the current position
```

Shift the cursor marking the position for reading: use seekg() and tellg().

```
File.open(_T("c:\\temp\\data.bin"), fstream::out | fstream::in);
if (!File.good())  // failure, file was not open
   return;
File.write((const char*) &iArray [0], sizeof iArray);
if (!File.good())  // writing failed
   return;
```

More: see http://www.cplusplus.com/reference/iostream/

C++ strings (1)

```
#include <string> //See http://www.cplusplus.com/reference/string/
```

Some of the constructors:

```
string s1("abc"), // s1 contains characters a, b and c
       s2(s1, 1), // s2 is "bc"
       s3(5, 'a'), // s3 is "aaaaa"
       s4 = s1,   // copying, s4 is also "abc"
       s5;        // empty string, alternative is s5("");
string *ps1 = new string("abc"),
       *ps2 = new string(*ps1, 1), // ps2 points to string "bc"
       *ps3 = new string(5, 'a'), // ps3 points to string "aaaaa"
       *ps4 = new string(*ps1), // ps4 points to string "abc"
       *ps5 = new string; // ps5 points to empty string, alternative is new string("")
```

In case of Unicode use wstring:

```
wstring ws1(L"abc"), wps3 = new wstring(5, L'a');
```

Input and output:

```
cout << s1<<endl;
cin >> *ps5;  // when typing, press Enter to mark the end
```

C++ strings (2)

Arithmetics:

```
string s6 = s1 + s2; // s6 is " abcbc"
s6 += s3; // s6 is now "abcbcaaaaa"
```

Comparisons:

```
if (s1 == s2) // also !=, >, <,  >=, <=
{……………………}
```

Length:

```
int n = ps1->length(); // number of characters in string
```

Access:

```
char c1 = s1.at(0); // c1 gets value 'a'. If index is out of scope, throws out_of_range exeption
char c2 = s1[0]; // c2 gets value 'a'. If index is out of scope, the behavior is undefined
s1[0] = 'x'; // s1 is now "xbc"
s1[4] = 'y'; // error, corrupts the memory, use s1 += "y"
char c3 = s1.front(); // the first character
char c4 = s1.back(); // the last character
```

C++ strings (3)

Inserting:

```
s1.insert(1, "xx"); // s1 is " axxbc"
s2.insert(0, s3); // s2 is "aaaaabc"
```

Deleting:

```
s1.erase(1); // s1 is "a", all the characters from position 1 are removed
s2.erase(0, 5); // s2 is "bc", 5 characters starting from position 0 are removed
```

Replacing:

```
s3.replace(1, 3, s4, 1, 2); ); // s3 is "abca", characters 1, 2 and 3 are replaced with characters
                               // on positions 1 and 2 in string s4
```

Substrings:

```
cout << s4.substr(1) << endl; // prints "bc" (all the characters from position 1)
cout << s4.substr(0, 2) << endl; // prints "ab" (2 characters from position 0)
```

Searching:

```
s1 = "axxbc";
int n = s1.find("xx"); // returns 1 as is the beginning of substring "xx"
n = s1.find("zz"); // returns -1 (i.e. not found)
```

C++ strings (4)

Conversions to string:

string to_string(arg); // arg can be any int, float, double

Conversions from string (throw exception if converting impossible):

int stoi(string);

long stol(string);

unsigned long stoul(string);

float stof(string);

double stod(string);

Get C string:

const char *p = s1.c_str(); // valid until s1 is not changed

Actually typedef std::basic_string<char> string; basic_string is a template

typedef std::basic_string<wchar_t> wstring;

Everybody can define _tstring:

typedef std::basic_string<TCHAR> _tstring;

// Do not forget to include this definition into your homework

_tstring(_T(“abc”));

String streams

```
#include<sstream> // See http://www.cplusplus.com/reference/sstream/stringstream
```

Input/output is from/to strings.

```
cout << "Failure, error is " << GetLastError() << endl; // prints in command prompt window
stringstream sout; // not a predefined object
sout << "Failure, error is " << GetLastError() << endl; // string is stored in sout
sout << "Press ESC to continue, ENTER to break" << endl;
cout << sout.str(); // prints the previous two rows
sout.str(""); // clears the text stored is sout
sout.str("Data:\n"); // stores the row in sout, alternative to sout << "Data:" <<endl
```

String output streams format data exactly as ordinary output streams. But instead of immediate output they store the formatted data allowing to output them later.

String input streams are mostly used for parsing:

```
void fun(string name) // name like "John Smith"
{
   string first_name, last_name;
   stringstream name_stream(name);
   name_stream >> first_name >> last_name;
             // now first_name is "John", last_name is "Smith"
   ………………………………
```

C++ standard exceptions (1)

```
#include <exception> // See http://www.cplusplus.com/reference/exception/
void LinkedList::Insert(void *p, int i)
{
  if (i < 0) throw out_of_range("Index is negative"); // the argument must be const char *
  if (!p) throw invalid_argument (string("No object")); // also, the argument may be string &

  ………………………..
}
LinkedList *pList;
try
{
   pList->Insert(_T("Hello"), 10);
}
catch(const out_of_range &e1)
{
   cout << e1.what() << endl;
}
catch (const invalid_argument &e2)
{
    cout << e2.what() << endl;
}
```

C++ standard exceptions (2)

```
#include <exception>
void LinkedList::Insert(void *p, int i) throw (out_of_range, invalid_argument)
{            // throw list informs the user about excpetions the function may throw
  if (i < 0) throw out_of_range(string("Failure, index is negative"));
  if (!p) throw invalid_argument (string("Failure, no object"));
  …………………………..
}
```

The throw list is not compulsory. If present, it must be included also into the prototype:

```
void LinkedList::Insert(void *, int) throw (out_of_range, invalid_argument);
```

To emphasize that the current function does not throw exceptions, you may write:

```
void Fun() noexcept;
```

The throw list may cause compiler warnings. To suppress them write

```
#pragma warning( disable : 4290 )
```

All the standard exception classes are derived from a base class called exception.

C++ standard exceptions (3)

```
#include <exception>
void LinkedList::Insert(void *p, int i) throw (out_of_range, invalid_argument,
                                                runtime_error)
{   ……………………….. }

LinkedList *pList;
try
{
    pList->Insert(_T("Hello"), 10);
}
catch(const out_of_range &e1)
{
    cout << e1.what() << endl;
}
catch (const exception &e2)
{ // all the standard exceptions have the same base class called "exception"
     cout << e2.what() << endl;
}
```

Time handling (1)

```
#include <time.h>
time_t CurrentTime = time(nullptr); // raw time, alternative is time(&CurrentTime);
cout << ctime (&CurrentTime); // string like Wed Aug 30 10:51:00 2017
struct tm *pLocalTime = localtime(&CurrentTime);
```

where

```
struct tm {int tm_sec, tm_min, tm_hour, tm_mday, tm_wday, tm_yday, tm_mon, tm_year,
tm_isdast};
```

Curiosities:

```
cout << pLocalTime->tm_year << endl; // years since 1900, for example 117
cout << pLocalTime->tm_mon << endl; // month as integer, January is 0
cout << pLocalTime->tm_mday << endl; // day as integer, starting from 1
cout << pLocalTime->tm_wday << endl; // day in week as integer, Sunday is 0
cout << asctime(pLocalTime) << endl; // string like Wed Aug 30 10:51:00 2017
```

For custom data formatting

```
char Buf[100];
strftime(Buf, sizeof Buf, "%d-%m-%Y %H:%M:%S", pLocalTime);
cout << Buf << endl; // string like 30-08-2017 10:51:00
```

To get raw time other than the current time change members of pLocalTime and

```
time_t OtherTimePoint = mktime(pLocalTime);
```

Time handling (2)

```
#include <chrono> // See http://www.cplusplus.com/reference/chrono/
using namespace std::chrono;
```

Namespace chrono includes five components: system_clock, steady_clock, high_resolution_clock, time_point and duration. Duration and timepoint are components of clocks.

- *system_clock* represents timepoints associated with the computer usual real-time clock.
- *steady_clock* guarantees that it never gets adjusted.
- *high_resolution_clock* represents the clock with the shortest possible tick period. In Visual Studio equivalent with the *system_clock*.

Usage:

```
system_clock::time_point CurrentTime = system_clock::now();
time_t CurrentTime_t = system_clock::to_time_t(CurrentTime);
struct tm *pLocalTime = localtime(&CurrentTime_t);
// after that apply the C standard time handling functions
```

A time_point is always associated with a clock:

```
time_point t; // error
time_point<system_clock> t; // correct
```

The time_point has epoch (or origin, 01.01.1601 in case of Windows, 01.01.1970 in case of Linux). Its value is actually the duration from the epoch (measured in 100ns units in case of Windows and seconds in case of Linux).

Time handling (3)

Duration is specified by various templates. Examples:

```
seconds Duration1(20);             // declares time interval 20s
hours Duration2(24);               // declares time interval 24 hours
milliseconds Duration3(1500);      // declares time interval 1500ms
```

Examples of operator functions:

```
milliseconds Duration4(1000);
milliseconds Duration5(2000);
milliseconds Duration6 = Duration4 + Duration5; // get time interval 3000ms
if  (Duration4 < Duration5)
{ …………………… }
seconds Duration7(1);
milliseconds Duration8 = Duration6 + Duration7; // different units, we get 4000ms
cout << Duration8.count() << endl;  // prints 4000, cout << Duration8; does not work
milliseconds Duration9 = (milliseconds)Duration7; // casting, get time interval 1000ms
```

Output with iostream and sstream:

```
system_clock::time_point when = system_clock.now();
time_t when_t = system_clock::to_time_t(when);
struct tm *pLocalTime = localtime(&when_t);
cout << put_time(pLocalTime, "%d-%m-%Y %H:%M:%S") << endl; // from C++ v 11
```

STL threads (1)

```
#include <thread> // See http://www.cplusplus.com/reference/thread/
using namespace std::thread;
```

To declare a thread and launch it, write:

```
thread thread_object_name(reference_to_entry_point_function, list_of_input_parameters);
```

Example: for entry point function

```
void Compute(int Arg1, int Arg2, int *pResult) ) {.............................}
```

the thread may be

```
int Result;
thread ThreadCompute(Compute, 10, 20, &Result);
```

The entry point function may have any set of input parameters, but no return value.

When the main function terminates, all the threads will also terminate.

```
void Process() {.............................}
int main()
{
    thread ProcessThread(Process);
    ProcessThread.join(); // waits until Process has done its job
    return 0;
}
```

STL threads (2)

If the entry point function is a member of a class:

```
thread thread_object_name(reference_to_entry_point_function,
                          pointer_to_object_of_that_class,
                          list_of_input_parameters);
```

Example:

```
class ThreadWrapper
{
  public:
      void Compute(int Arg1, int Arg2, int *pResult) ) {...........................}
};
.......................................................
ThreadWrapper pWrap = new ThreadWrapper;
int Result;
thread ThreadCompute(&ThreadWrapper::Compute, pWrap, 10, 20, &Result);
```

STL threads (3)

Suppose we have

```
void Compute(int Arg1, int Arg2, int *pResult) throw (invalid_argument)
{
   ............................
   if (.....) throw invalid_argument("..... ");
}
int main()
{
 int Result;
 thread ThreadCompute(Compute, 10, 20, &Result);
   // we need to process the exception, but the ordinary try{} catch() {} expression
   // does not help here
 .........................
 }
```

For solution we need:

An object of class exception_ptr, used for storing the exception

Function current_exception() creating from ordinary exception an object of class *exception_ptr* (actually storing the exception).

Function rethrow_exception() allowing to transform the object of class exception_ptr back to ordinary exception.

```
#include "stdafx.h"
#include "Windows.h"
#include <string>
#include <iostream>
#include <exception>
#include <thread>
#include "math.h"

#if defined(UNICODE) || defined(_UNICODE)
#define _tcout std::wcout
#define _tcin std::wcin
#else
#define _tcout std::cout
#define _tcout std::cin
#endif

using namespace std;
typedef std::basic_string<TCHAR> _tstring;

void Compute(int, int, double *) throw (invalid_argument);
void ThreadMain(int, int, double *, exception_ptr *);

int _tmain(int argc, _TCHAR* argv[])
{
   double roots[2];
   exception_ptr ExPtr = NULL;
   // object of class exception_ptr, actually the reference to exception
```

```
    // operator functions of this calls allow us to write expressions "ExPtr = NULL" and
     // "if (ExPtr == NULL)"
    thread ComputeThread(ThreadMain, 5, 26, roots, &ExPtr); // launch the thread
    ComputeThread.join(); // wait until end of thread (normal or due to exception)
    try
    {
        if (ExPtr != NULL) // i.e. the thread has thrown an exception
            rethrow_exception(ExPtr); // rethrow to apply the classic try-catch mechanism
    }
    catch(const exception& ex)
    {
        cout << ex.what() << endl;
        return 1;
    }
     _tcout << roots[0] << endl; // no exceptions, results are got
     _tcout << roots[1] << endl;
    return 0;
}

void Compute(int a1, int a0, double *pResult) throw (invalid_argument)
{ // the body of thread, the actual work is performed here
    if (((a1 * a1) / 4.0 - a0) < 0)
        throw invalid_argument("Wrong coefficients");
    *pResult = a1 / 2.0 + sqrt((a1 * a1) / 4.0 - a0);
    *(pResult +1) = a1 / 2.0 - sqrt((a1 * a1) / 4.0 - a0);
}
```

```
void ThreadMain(int a1, int a0, double *pResult, exception_ptr *pEx)
{ // wraps the thread body, stores the exceptions
    try
    {
        Compute(a1, a0, pResult);
    }
    catch(const exception& ex)
    {
        *pEx = current_exception(); // remember the exception
    }
    return;
}
```

STL threads (4)

Instead of interlocked functions we may use atomic variables:

```
#include <atomic>
using namespace std;
atomic <int> i(0);
atomic <char> c('A');
atomic <long> l(100000);
.............................. // all the integral types are allowed
atomic <int *> p(NULL); // atomic pointer
```

i, c , l etc. are objects. Their initialization is compulsory. Due to operator functions you may write simply:

```
atomic <int> i = 0;
atomic <char> c = 'A';
atomic <long> l = 100000;
```

Functions declared in atomic classes are interlocked functions. For example

```
i.store(1);                  // or due to operator functions i = 1;
if (i.load() == 1) .....     // or due to operator functions if (i == 1)
```

The operations with object "i" are atomic: when the store and load functions are active, no other threads can access this object.

Some examples about the other atomic class functions:

```
i.fetch_add(1);              // or due to operator functions i++;
i.fetch_add(10);             // or due to operator functions i+=10;
i.fetch_sub(1);              // or due to operator functions i--;
```

```cpp
#include "stdafx.h"
#include "Windows.h"
#include "conio.h"
#include <string>
#include <iostream>
#include <atomic>
#include <thread>
#if defined(UNICODE) || defined(_UNICODE)
#define _tcout std::wcout
#define _tcin std::wcin
#else
#define _tcout std::cout
#define _tcout std::cin
#endif
using namespace std;
typedef basic_string<TCHAR> _tstring;
void RunKeyboard();

atomic<int> Stop(FALSE); // or Stop = FALSE;

int _tmain(int argc, _TCHAR* argv[])
{
    thread KeyboardThread(RunKeyboard);
    while (Stop.load() != TRUE) // or Stop != TRUE
        _tcout << _T("Print ESC to stop program") << endl;
    KeyboardThread.join();
    return 0;
}
```

```
//
// Keyboard thread
//
void RunKeyboard()
{
    while (_getch() != 27);    // 27 - ESC
        _tcout << _T("Terminating") << endl;
    Stop.store(TRUE);    // or Stop = TRUE;
}
```

STL threads (5)

For synchronization use mutexes (very similar to critical sections):

```
#include <mutex>
using namespace std;
mutex mx;
..............................................................
mx.lock();  // like entering into the critical section
..............................................................
mx.unlock();  // like leaving the critical section
```

Additional possibilities:

```
while (TRUE)
{
       if (mx.try_lock())
       {
              .................................. // critical section operations
              mx.unlock();
              break;
       }
       else
       {
              .................................... // do something else, then try once more to enter
       }
}
```

STL threads (6)

```
#include <mutex>
using namespace std;
timed_mutex tmx;
chrono::milliseconds timeout(1000);

if (tmx.try_lock_for(timeout))
{           // successful locking
            .................................. // critical section operations
            tmx.unlock();
}
else
{           // 1000ms has elapsed but locking is still not possible
            cout << " Problems: the other thread has frozen" << endl;
            return;
}
```

Timed mutexes can operate as ordinary mutexes, i.e. you may write simply

```
tmx.lock();
```

and

```
tmx.try_lock();
```

```cpp
#include "stdafx.h"
#include "Windows.h"
#include "conio.h"
#include <string>
#include <iostream>
#include <fstream>
#include <atomic>
#include <thread>
#include <mutex>
#include <vector>
#include <chrono>
#if defined(UNICODE) || defined(_UNICODE)
#define _tcout std::wcout
#define _tcin std::wcin
#else
#define _tcout std::cout
#define _tcout std::cin
#endif
using namespace std;
using std::chrono::system_clock;

typedef std::basic_string<TCHAR> _tstring;

void RunKeyboard();
void RunProducer();
void RunConsumer();
atomic<int> Stop(FALSE);
```

```
mutex mx;
vector<int> Buf(32, 0);

int _tmain(int argc, _TCHAR* argv[])
{
   thread KeyboardThread(RunKeyboard);
   thread ProducerThread(RunProducer);
   thread ConsumerThread(RunConsumer);
   while (Stop.load() != TRUE)
      this_thread::sleep_for(chrono::milliseconds(0));  // use a primitive spinlock here
   KeyboardThread.join();
   ProducerThread.join();
   ConsumerThread.join();
   return 0;
}

void RunKeyboard()  // Keyboard thread
{
   while (_getch() != 27);
   _tcout << _T("Terminating") << endl;
   Stop.store(TRUE);
}

void RunProducer()// Producer thread produces data (array of random numbers)
{
   int i, delay;
   srand((unsigned)time(NULL)); // initialize the generator of random numbers
   while (!Stop.load())
```

```
    {
        mx.lock();  // 1
        delay = (int)((double)rand() / (RAND_MAX + 1) * (5000 - 1000) + 1000);
            // RAND_MAX is defined as 0x7FFF
            // here the random delay is between 1000ms and 5000ms
        this_thread::sleep_for(chrono::milliseconds(delay));
        for (i = 0;  i < 32;  i++)
            Buf[i] = rand();
        mx.unlock();  // 2
    }
}

void RunConsumer() // Consumer thread consumes the data (computes the checksum)
{
    long int sum = 0;
    int i;
    while (!Stop.load())
    {
        mx.lock(); // 3
        for (i = 0, sum = 0;  i < 32;  i++)
            sum += Buf[i];
        _tcout << sum << endl;
        mx.unlock(); // 4
    }
}
// This code is not 100% correct. The problem is as follows.
// Although the Producer starts before the Consumer, it may
```

```
// happen that row 3 is reached before row 1. In that case
// the first printed value is 0 (the Consumer calculates
// the checksum using the initial values in Buf, i.e. zeroes).
```

STL threads (7)

To simplify locking and unlocking with mutexes use class lock_guard:

```
lock_guard<mutex> lock_guard_name(mutex_name);
```

Example:

```
#include <mutex>
using namespace std;
mutex mx;
lock_guard<mutex> lock (mx); // we call constructor for object lock
```

The lock_guard has only two methods: contructor and destructor.
When the constructor is called, the lock() method of the corresponding mutex is also called.
When the destructor is applied, the unlock() method of the corresponding mutex is also called.

With manual locking, you have to ensure that the mutex is unlocked correctly on every exit path from the region where you need the mutex locked, including when the region exits due to an exception. Having a local lock_guard object (i.e. auto memory class) it is not a problem: even if an exception is thrown, the destructor is applied and the mutex released.

```
#include "stdafx.h"
#include "Windows.h"
#include <string>
#include <iostream>
#include <fstream>
#include <atomic>
#include <thread>
#include <mutex>

#if defined(UNICODE) || defined(_UNICODE)
#define _tcout std::wcout
#define _tcin std::wcin
#else
#define _tcout std::cout
#define _tcout std::cin
#endif
using namespace std;
typedef std::basic_string<TCHAR> _tstring;

void RunKeyboard();

_tstring sCommand = _T("");

mutex CommandProcessed;
atomic<int> bStop = FALSE;

int _tmain(int argc, _TCHAR* argv[])
```

```
{
   thread KeyboardThread(RunKeyboard);
   while (TRUE)
      {
         lock_guard<mutex> lock(CommandProcessed);
            // Object lock is visible only within the while(TRUE) {} loop. Each
            // time when the loop starts, the lock_guard costructor creates object
            // lock for mutex CommandProcessed. During this, CommandProcessed.lock()
            // is applied
         if (sCommand == _T("exit"))
         {
            bStop = TRUE;
               // here we jump out of the loop. Object lock is destroyed
               // and during this operation CommandProcessed.unlock()
               // is applied.
            break;
         }
            // Object lock is visible only within the while(TRUE) {} loop.
            // Each time when the loop ends, lock_guard destructor is called.
            // Inside this, CommandProcessed.unlock() is applied.
      }
   KeyboardThread.join();
   return 0;
/* Alternative solution:
while(TRUE)
{
   CommandProcessed.lock();
   if (sCommand == _T("exit")
```

```
    {
        bStop = TRUE;
        CommandProcessed.unlock();
        break;
    }
    CommandProcessed.unlock();
}
*/
}
//
// Keyboard thread
//
void RunKeyboard()
{
    while (TRUE)
    {
        lock_guard<mutex> lock(CommandProcessed);
        if (bStop)
            return;
        _tcin >> sCommand;
    }
}
```

STL threads (8)

The unique_lock is more flexible:

```
unique_lock<mutex> lock_guard_name(mutex_name);
```

// as lock_guard – the constructor locks the associated mutex.

```
unique_lock<mutex> lock_guard_name(mutex_name, std::defer_lock);
```

// the constructor does not lock the associated mutex

You may later call the mutex lock() or try_lock() methods to lock.
The destructor of unique_lock unlocks the associated mutex. But you may also unlock the mutex with unlock() method.

Example:

```
#include <mutex>
using namespace std;
mutex mx;
unique_lock<mutex> lock (mx, defer_lock);
............................................................
lock.lock();
.......................... // critical section
lock.unlock();
```

STL threads (9)

The unique_lock associated with a timed_mutex is also allowed . Example:

```
#include <mutex>
using namespace std;
timed_mutex tmx;
unique_lock<timed_mutex> timed_lock(tmx);
..............................................................
timed_lock.try_lock_for (chrono::milliseconds(3000));
.......................... // critical section
lock.unlock();
```

Some terminology:

```
mutex mx;
........................................................
mx.lock();      // the thread acquires the mutex
...................... // the thread owns the mutex
mx.unlock();  // the thread releases the mutex

unique_lock<mutex> lock (mx, defer_lock);
..............................................................
lock.lock();          // the thread acquires the lock
..........................   // the thread owns the lock
lock.unlock();      // the thread releases the lock
```

STL threads (10)

Inter-thread communication in C++ versioon 11 is implemented by conditional variables.

Let us have two threads, thread2 can process data only when thread1 has prepared it.

```
#include <condition_variable>
#include <mutex>
using namespace std;
mutex mx; // global variables
condition_variable cv;
```

In thread 1:

```
lock_guard<mutex> lock1(mx);
…………………………. // do something, prepare data
cv.notify_one();   // orders thread2 to stop waiting
```

In thread 2:

```
unique_lock<mutex> lock2(mx);
cv.wait(lock2);  // blocks the thead, waits for the notification,
…………………………. // processes data
```

```
#include "stdafx.h"
#include "Windows.h"
#include "conio.h"
#include <string>
#include <iostream>
#include <fstream>
#include <atomic>
#include <thread>
#include <mutex>
#include <vector>
#include <chrono>
#include <condition_variable>
#if defined(UNICODE) || defined(_UNICODE)
#define _tcout std::wcout
#define _tcin std::wcin
#else
#define _tcout std::cout
#define _tcout std::cin
#endif
using namespace std;
using std::chrono::system_clock;

typedef std::basic_string<TCHAR> _tstring;

void RunKeyboard();
void RunProducer();
void RunConsumer();
```

```
atomic<int> Stop(FALSE);
mutex mx;
condition_variable cv;
vector<int> Buf(32, 0);

int _tmain(int argc, _TCHAR* argv[])
{
    thread KeyboardThread(RunKeyboard);
    thread ProducerThread(RunProducer);
    thread ConsumerThread(RunConsumer);
    KeyboardThread.join();
    ProducerThread.join();
    ConsumerThread.join();
    return 0;
}

void RunKeyboard() // Keyboard thread
{
    while (_getch() != 27);
    _tcout << _T("Terminating") << endl;
    Stop.store(TRUE);
}

void RunProducer()// ProducerThread produces data (array of random numbers)
{   int i, delay;
    srand((unsigned)time(NULL)); // initialize the generator of random numbers
    while (!Stop.load())
    {
```

```
        lock_guard<mutex> lock(mx); // 2
        delay = (int)((double)rand() / (RAND_MAX + 1) * (5000 - 1000) + 1000);
        this_thread::sleep_for(chrono::milliseconds(delay));
        for (i = 0;  i < 32 && !Stop.load();  i++)
            Buf[i] = rand();
        cv.notify_one(); // 3
    }
}

void RunConsumer()// ConsumerThread consumes the data (computes the checksum)
{
    long int sum = 0;
    int i;
    while (!Stop.load())
    {
        unique_lock<mutex> lock(mx); // 4
        cv.wait(lock); // 5
        for (i = 0, sum = 0;  i < 32;  i++)
            sum += Buf[i];
        _tcout << sum << endl;
        lock.unlock(); // 6
    }
}
// The locks guarantee that when one of the threads works with the buffer,
// another thread cannot access it (see the STLMutexExample). But the
// Consumer must not start to compute the sum when the buffer is empty.
// If the Consumer thread tries to do it, on row 5 the wait function:
```

```
// a) releases the mutex, thus allowing the Producer to work
// b) blocks the Consumer until the Producer has not signalled (row 3)
//    that the buffer is full.
```

STL containers

Containers:

- vector
- list
- forward_list (from C++ 11)
- queue (FIFO)
- deque (double-ended queue)
- priority_queue
- stack (LIFO)
- map
- set
- hash_tables (several, from C++ 11)

Some algorithms implemented on containers:

- copy
- replace
- remove
- search
- sort
- merge
- count

Vectors (1)

```
#include <vector> // See http://www.cplusplus.com/reference/stl/
using namespace std;
vector<int> iVector(10); // array for 10 integers as object iVector
vector<double> dVector(10, 1.0); // with initialization
vector<string> sVector(10);
vector<DCMotor> dcmVector(10); // array of 10 DCMotors as object dcmVector
vector<DCMotor> *pdcmVector = new vector<DCMotor>(10);
                                    // dynamically allocated array of 10 DCMotors as object
delete pdcmVector; // not delete[]
```

To use vectors for a user-defined class, this class must contain the following methods:

- Copy constructor (obligatory)
- Destructor (obligatory)
- operator= (obligatory)
- Constructor without arguments (not obligatory, but if not present, some of the vector methods like resize() will fail)
- operator== (as above)
- operator< (as above)

Vectors (2)

```
DCMotor MotorA (0x378);
MotorA.SetSpeed(0.5);
vector<DCMotor> dcmVector(2);
dcmVector[0] = MotorA; // operator[], the vector will contain the copy of MotorA
MotorA.SetSpeed(0.75);
cout << dcmVector[0].GetSpeed() << endl; // prints 0.5
cout << MotorA.GetSpeed() << endl; // prints 0.75

DCMotor MotorB= dcmVector[0]; // MotorB is the copy of vector element
MotorB.SetSpeed(0.75);
cout << dcmVector[0].GetSpeed() << endl; // prints 0.5
cout << MotorB.GetSpeed() << endl; // prints 0.75

dcmVector[0].SetSpeed(0.75);
cout << dcmVector[0].GetSpeed() << endl; // prints 0.75

dcmVector.at(0).SetSpeed(0.75); // as dcmVector[0].SetSpeed(0.75); but if the
                                // index is wrong, throws the out_of_range exception
DCMotor MotorC= dcmVector.front(); // gets the first element
DCMotor MotorD= dcmVector.back(); // gets the last element
```

Vectors (3)

```
vector<int> iVector1(10, 0); // array for 10 integers initialized to 0
iVector1.push_back(0); // appends a new element 0, now the size is 11
iVector1.pop_back(); // removes the last element, now the size is 10 again
cout << iVector1.size() << endl; // prints the size
iVector1.resize(20, 0); // sets the size to 20, new elements are 0
iVector1.resize(5); // sets the size to 5, the last 15 elements are removed
iVector1.clear(); // sets the size to 0, removes all the elements
if (iVector1.Empty())
   cout << "Empty" << endl;

vector<int> iVector2 = iVector1; // copy constructor
vector<int> iVector3; // array with size 0
iVector3 = iVector1; // operator=

if (iVector1 == iVector2) // operator==, there are also operators for !=, >, >=, <, <=
    cout << "Equal" << endl;
```

Vectors (4)

```
int iArray[100];  // C-style array
for (int i = 0;  i < 100;  i++)     cout << iArray[i] << endl;
```

or

```
for (int *p = &iArray[0];    p != &iArray[100];    p++)   cout << *p << endl;

vector<int> iVector(100); // C++ vector
for (int i = 0;  i < 100;  i++)     cout << iVector[i] << endl;
```

or

```
for (vector<int>::iterator it = iVector.begin();   it != iVector.end();  ++it)
     cout << *it << endl; // begin() returns iterator to the first element, end() to the
                          // first non-existing element.
```

An iterator is any object that, pointing to some element in an array or other range of elements, has the ability to iterate through the elements of that range using a set of operators (at least, the (++) increment and (*) dereference). In C-style array the simplest iterator is the pointer. For C++ vectors and other containers the iterators are objects of certain classes. Overloaded operations for iterators are ++, --, *, += and + for adding integers. Thus, the iterator objects and ordinary pointers have the same set of functionalities. Otherwise, the iterator is a smart pointer.

Vectors (5)

```
vector<int> *piVector = new vector<int>(100);   // pointer to dynamically allocated
                                                // C++ vector
for (int i = 0;  i < 100;  i++)
   cout << piVector->at(i) << endl;
or
for (vector<int>::iterator it = piVector->begin();   it != piVector->end();  ++it)
    cout << *it << endl;

vector<DCMotor> dcmVector(2);
for (vector< DCMotor >::iterator it = dcmVector.begin();   it != dcmVector.end();  ++it)
    it->SetSpeed(0.75);

vector<int> iVector(100);
for (vector<int>::const_iterator cit = iVector.begin();   cit != iVector.end();  ++cit)
    cout << *cit << endl;
but
for (vector<int>::const_iterator cit = iVector.begin();   cit != iVector.end();  ++cit)
   *cit = 10; // error, modifying with const_iterator is not possible
```

Vectors (6)

```
vector<int> iVector(10);
vector<int>::iterator it;
int k = 0;
for (it = iVector.begin();  it != iVector.end();  ++it)
    *it = k++; // members of vector are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
iVector.insert(iVector.begin() + 2, 10); // new element 10  to position 2
// members of vector are 0, 1, 10, 2, 3, 4, 5, 6, 7, 8, 9
iVector.insert(iVector.begin() + 3, 3, 11); // 3 new elements, value 11, from position 3
// members of vector are 0, 1, 10, 11, 11, 11, 2, 3, 4, 5, 6, 7, 8, 9
iVector.insert(iVector.begin() + 6, iVector.begin() + 12, iVector.begin() + 14);
// 2 new elements, insert from position 6, values taken from positions 12 and 13 (not 14)
// members of vector are 0, 1, 10, 11, 11, 11, 8, 9, 2, 3, 4, 5, 6, 7, 8, 9
// the second argument is the first in range, the third argument is the first not in range
iVector.erase(iVector.begin() + 2); // removes element from position 2
// members of vector are 0, 1, 11, 11, 11, 8, 9, 2, 3, 4, 5, 6, 7, 8, 9
iVector.erase(iVector.begin() + 1, iVector.begin() + 4);
// removes elements from positions 1 to 3 (not 4)
// members of vector are 0, 11, 8, 9, 2, 3, 4, 5, 6, 7, 8, 9
```

Lists

list – doubly linked; forward_list – single linked, moving backwards not possible

```
list<DCMotor> dcmList (4);
DCMotor MotorA= dcmList.front(); // gets the first element
DCMotor MotorB= dcmList.back(); // gets the last element
DCMotor MotorC= dcmList.at(1); // error, random access not possible, use iterators
DCMotor MotorD= dcmList.[2]; // not possible, use iterators
list< DCMotor >::iterator it;
int i;
for (it = dcmList.begin(), i = 0;    it != dcmList.end();  ++it,  i++)
{ // to get DCMotor with index 2
    if (i == 2)
    {
       it->SetSpeed(0.75);  // or (*it).SetSpeed(0.75)
       break;
    }
}
```

Functions push_back(), pop_back(), clear(), size(), empty(), insert(), erase(), operator overloading functions, etc. are as in vectors.

Iterators are also as the vector iterators. For forward_list the operaator--() is not implemeted.

Range-base *for* loop (1)

```
int iArray[] = {1, 2, 3, 4, 5};
for (int i : iArray)
    cout << i << endl; // prints all the members
```

Introduced in C++version 11. But range-base for loop is tricky

```
for (int i : iArray)
    i++; // has no efect, the array is still 1, 2, 3, 4, 5
```

Correct is

```
for (int& i : iArray)
    i++; // now the array is 2, 3, 4, 5, 6
```

If the intention is to modify the values stored in the container, the loop variable must be a reference.

The range-base for loop is applicable *for* looping over STL containers:

```
vector<int> iVector(10);
for (int& i : iVector)
    i = 10;
vector<DCMotor> motors(5);
for (DCMotor& m : motors)
    m.SetSpeed(0.75); // here m is a member of vector motors
```

Range-base *for* loop (2)

Range-base *for* loop helps to avoid tinkering with iterators.

```
vector<DCMotor> motors(5);
for (int i = 0;  i < 5;  i++)
    cout << motors[i].GetSpeed() << endl; // OK, because we have a vector
for (int i = 0;  i < 5;  i++)
    cout << motors.At(i).GetSpeed() << endl; // OK, because we have a vector

list<DCMotor> motors(5);
for (int i = 0;  i < 5;  i++)
    cout << motors[i].GetSpeed() << endl; // error, because we have a list
for (int i = 0;  i < 5;  i++)
    cout << motors.At(i).GetSpeed() << endl; // error, because we have a list

for (list< DCMotor >::iterator it = motors.begin();   it != motors.end();  ++it)
    cout << it->GetSpeed() << endl; // OK, but complicated
```

Comfortable and simple solution:

```
for (DCMotor& m : motors)
    cout << m.GetSpeed() << endl;
```

Algorithm *find_if*

```
list<DCMotor> motors(5);
list< DCMotor >::iterator it = motors.begin(); // or simply auto it = motors.begin()
for (;   it != motors.end();  ++it
{
     if (it->GetSpeed() == 0.75)
         break;
}
if (it == motors.end()
    cout << "Not found" << endl;
else
    …… // it points now to motor with speed 0. 75
```

Simpler solution uses STL algorithms:

```
#include <algorithm>
auto end = motors.end();
auto it = find_if(motors.begin(), end, check); // it points to motor with speed 0. 75
if (it == motors.end()
    cout << "Not found" << endl;
else
    …… // it points now to motor with speed 0.75
```

where

```
bool check(DCMotor dm) {  return dm.GetSpeed() == 0.75; }
```

Lambda expressions (1)

```
#include <algorithm>
auto end = motors.end();
auto it = find_if(motors.begin(), end, [ ] (DCMotor dm) { return dm.GetSpeed() == 0.75; });
```

Call to function *check* is replaced by lambda expression.

The lambda (the term is from LISP language) is a short nameless function defined in the body of another function. The syntax of the simplest lambda is [ ] (*parameter list*) { *body* }. The type of return value is specified by the expression following the *return* keyword. If there is no *return* statement, the return type is void.

Lambda is treated similary to pointers to functions:

```
int *pArray = new int[100];
..............................
auto check = [ ] (int *p, int n, int m) // check is a pointer to lambda
   { int i = 0;  for (;  i < n && *(p + i) != m;  i++); return i != n; };
bool b = check(pArray, 100, 50); // true if pArray contains an integer equal with 50
```

Rather senseless but possible alternative:

```
bool b = [ ] (int *p, int n, int m)
   { int i = 0;  for (;  i < n && *(p + i) != m;  i++); return i != n; } (pArray, 100, 50);
```

Lambda expressions (2)

If the body contains several *return* statements or specifying the return type simply by the expression following the *return* keyword is not acceptable, the syntax to be used is
[ ] (*parameter list*) -> *return-type* { *body* }.

```
auto x = [ ] (int *p, int n, int m) -> int
   { int i = 0;  for (;  i < n && *(p + i) != m;  i++); return i != n; } (pArray, 100, 50);
```

Lambda can use variables from the enclosing block.

```
double max_speed = 0.75;
auto it = find_if(motors.begin(), end,
              [max_speed] (DCMotor dm) { return dm.GetSpeed() == max_speed; });
```

// *max_speed* in the lambda is the copy of *max_speed* defined in the outer scope
The brackets are to specify the capture block: the list of variables from outer scope that the lambda must use in its body. [=] means that the list absorbs all the outer scope variables.

```
double max_speed = 0.75;
auto end = motors.end();
auto it = find_if(motors.begin(), end,
              [max_speed&] (DCMotor dm) { return dm.GetSpeed() == max_speed; });
```

// *max_speed* in the lambda and *max_speed* in the outer scope are the same.
[&] means that the list absorbs references to all the outer scope variables.

Algorithm *for_each*

```
list<DCMotor> motors(5);
for (auto it = motors.begin();   it != motors.end();  ++it)
{
    cout << it->GetSpeed()  << endl;
}
```

Simpler solution uses STL algorithms and lambdas:

```
#include <algorithm>
for_each(motors.begin(), motors.end(),
            [ ] (DCMotor dm) { cout << dm.GetSpeed() << endl; });
```

The lambda specifies what to do with each value stored in the container.

```
double needed_speed = 0.75;
for_each(motors.begin(), motors.end(),
            [needed_speed] (DCMotor dm) { dm.SetSpeed(needed_speed); });
// error, the speed keeps its current value
for_each(motors.begin(), motors.end(),
            [needed_speed] (DCMotor& dm) { dm.SetSpeed(needed_speed); });
```

Maps (1)

A map consists of pairs <key, data>.

```
#include <map>
class Description
{
   private: int m_Size;
              string m_Unit;
   ……………………………………
    public: Description (int s, string u) : m_Size(s), m_Unit(u) { }
             Description() { } // constructor without arguments is obligatory
             string ToString( ) { return to_string(m_Size) + " bytes, unit is  " + m_Unit }
    ……………………………………
 };
map <string, Description> PhysicalValues =
{
      { "Temperature", Description(sizeof(double), "°C") },
      { "Pressure", Description(sizeof(double), "atm") },
      { "Concentration", Description(sizeof(int), "%") }
       ……………………………………………………………
};
```

Maps are not ordered (the members have no indeces or positions). Each member must have its own unique key.

Maps (2)

Inserting a new member:

```
auto ret = PhysicalValues.insert( { "Viscosity", Description(sizeof(double), "cSt") } );
cout << (ret.second ? "Success" : "Failure") << endl;
```

If the key already exists, the operation fails (even if the data component is different).
ret.first is the iterator to the new member or, in case of failure, to the existing member.
The iterator itself has members *first* pointing to the key and *second* pointing to the data component:

```
cout << ret.first->first << " " << ((Description)ret->first.second).ToString() << endl;
```

Alternative:

```
PhysicalValues["Viscosity"] = Description(sizeof(double), "cSt") ;
```

If the key already exists, the data component will be replaced by the new one.

Removing an existing member:

```
PhysicalValues.erase("Temperature"); // just mark the key
```

Searching:

```
auto it = PhysicalValues.find("Temperature"); // it is the iterator as above
if (it == Physicalvalue.end())
    cout << "Not found" << endl;
```

Maps (3)

Iterating:

```
for (auto it = PhysicalValues.begin();  it != PhysicalValues.end(); ++it)
   cout << it->first << " " << ((Description)it->second).ToString() << endl;
```

Alternative:

```
for (auto& p : PhysicalValues)
   cout << p.first << " " << ((Description)p.second).ToString() << endl;
```

More complicated maps:

- Multimaps allow several members with the same key
- Maps with comparison allow to specify another method for comparison of keys.