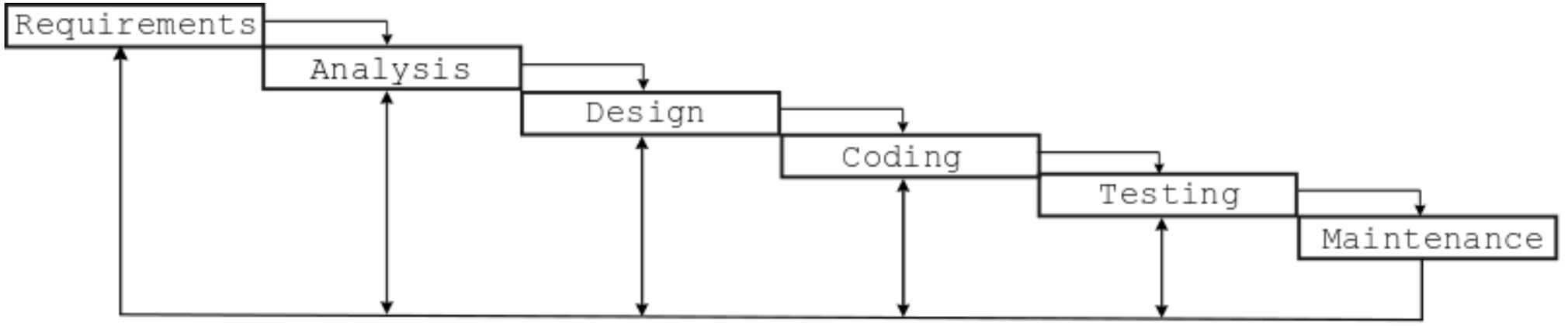
Software development life cycle

Software is developed through a series of steps. The sequence of steps is referred as the software development life cycle (SDLC).

There are several software development life cycle models. The simplest of them is the waterfall model:

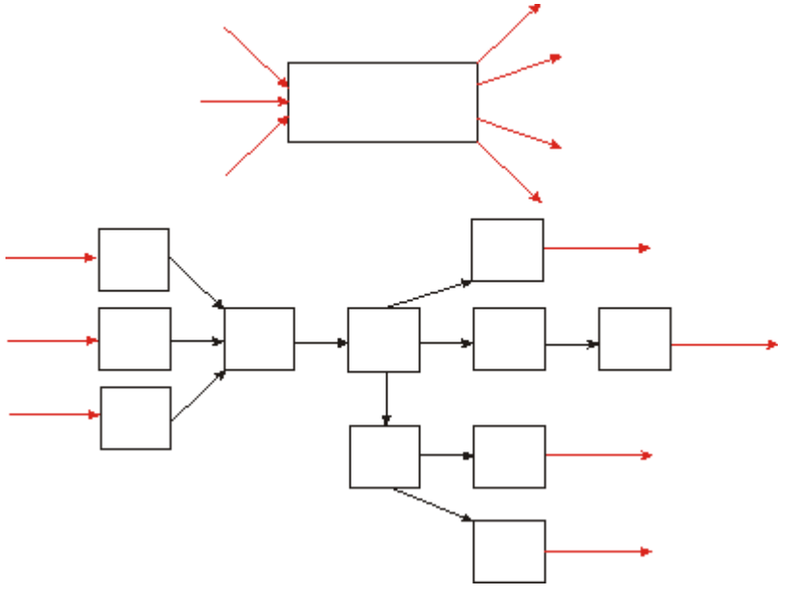


The software development cycle itself is endless. We may from any stage go back to an earlier stage.

The stages are not absolutely separated from each other. For example, rather often the coding and testing are performed concurrently: if a modul seems to be ready, it will tested immediately. Design is partly performed by coding tools (specifying the function prototypes, etc.).

Traditional approach: partitioning

The software product we have to impement gets some input data and from it generates some output data. So we may depict it as a box with some input arrows and some output arrows.



We divide the initial box into several smaller boxes, then repeat it with each smaller box, etc. Thus we get the partition of our softwere into modules. At last we attach to each module an accountable developer or a group of developers and the coding may start.

The main problem: the initial partition mirrors exactly and only the initial task. Sometimes even small changes in the problem specification can make it almost useless. The software modules written according to the specific partition are often so tightly connected to it that their usage in other projects without significant changes is not possible.

Objects and classes

In object-oriented programming (OOP) an object is a software model of real world objects (students, buildings, cars, etc.) as well as abstract concepts (stack, screen window, matrix, etc.).

Any real-world or abstract object has:

- State – set of characteristics describing it in a specific moment (color, weight, dimensions, etc.).
- Behavor – set of actions the object is able to perform.

Popular example: the state of a dog is specified by its name, color of fur, breed, etc. The behavor is running, barking, biting, etc.

Consequently, the software model of a real-world or abstract object must contain:

- Data members (in OOP mostly called attributes or member variables) for storing the state. The attributes have values: numbers, words, sets of numbers, long texts, pointers to other objects, etc.
- Functions (in OOP mostly called methods) to implement the behavor.

Objects having the same set of attributes (the values may be, of course, different) and the same set of methods belong to the same class. In class declaration we have to list the attributes and implement the code of methods. Classes as generalizations of objects are the building blocks of OOP program. The objects are just instances of their class. Each object is a member of a certain class.

Objects and classes in C++ (1)

Let us take a simple C struct:

```
struct Date
{
  int Day;
  int iMonth; // range 1…12, prefix i is to emphasize that month is an integer
  int Year;
};
```

It is a software model of abstract concept "date". But it presents just the state but not behavor.

Let us add a method and declare the class:

```
class Date
{
  int Day;
  int iMonth;
  int Year;
  char sDate[12]; // for date as string in format dd-mmm-yyyy
  void ToString(); // prototype of function that fills sDate
}; // do not forget the semicolon
```

In C++ a class is mostly implemented with 2 files: *class_name.h* for declaration and *class_name.cpp* for implementation of methods.

Objects and classes in C++ (2)

```
void Date::ToString() // scope operator, we have to define that this function belongs to Date
{
  const char MonthNames[12][4] =
 {
   "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug",  "Sep", "Oct", "Nov", "Dec"
  };
  sprintf_s(sDate, 12, "%02d %s %d", Day, MonthNames[iMonth - 1], Year);
} // sprintf_s instead of printing stores the string in buffer, formatting is as in printf
   // non-secure sprintf does not have argument presenting the length of output buffer
```

Turn attention that the function uses the members of its own class (*Day*, *iMonth*, *sDate*, *Year*) in the same way as the local variables (*MonthNames*).

Create an object:

```
Date d; // or class Date d; but the compiler itself understands what is Date
```

As in C, we get memory (here 24 bytes). Operator for accessing class members is also as in C (i.e. point). But

```
d.Day = 5; // error
```

This is because in class all the members are by default private. A private member can be accessed only by functions from the same class (like *ToString()* in our example). In other terms, a private member is encapsulated.

To allow access by any function we have to declare them as public.

Objects and classes in C++ (3)

```
class Date
{
public: // access modifier
  int Day, iMonth, Year;
  char sDate[12];
  void ToString();
};
```

Now:

```
Date d1, d2;
d1.Day = 5;   d1.iMonth = 3;    d1.Year = 2019;
d2.Day = 1;   d2.iMonth = 5;    d2.Year = 2019;
```

To get date as string write:

```
d1.ToString();
printf("%s\n", d1.sDate); // prints 05-Mar-2019
```

To call a member function we have to specify the corresponding object. *d1.ToString()* instructs function *ToString()* to use attribute values set for object *d1*. Similarly:

```
d2.ToString(); // uses values set to attributes of d2
printf("%s\n", d2.sDate); // prints 01-May-2019
```

If we write just *ToString();* the compiler supposes that we have a function *ToString()* that is not a class member (like *printf*)

Objects and classes in C++ (4)

Working with public data members is a bad idea. For example, if we do not forbid the user to set the value of member *iMonth* out of range 1…12, our *ToString()* may crash. To shield the data members they are mostly declared as private. Their accessing is performed by accessor functions.

```
class Date
{
private: // access modifier, valid until the next access modifier or until the end of declaration
  int Day,
      iMonth,
      Year;
  char sDate[12];
public:
  int GetDay(); // traditionally, the names of accessor functions start with words Get ans Set
  void SetDay(int);
  int GetMonth();
  void SetMonth(int);
  int GetYear();
  void SetYear(int);
  void ToString();
  char *GetDateString(); // no set function for sDate
};
```

Objects and classes in C++ (5)

In most cases the get-functions are very simple: they just return the value of attribute:

```
int Date::GetDay()
{
  return Day;
}
int Date::GetMonth()
{
  return iMonth;
}
int Date::GetYear()
{
  return Year;
}
char *Date::GetDateString()
{
  return &sDate[0];
}
```

Objects and classes in C++ (6)

Rather often the set-functions must check their input:

```
void Date::SetMonth(int m)
{
  if (m < 1 || m > 12)
     throw "Wrong month";
  iMonth = m;
}
void Date::SetYear(int y)
{
  if (y < 0)
     throw "Wrong year";
  Year = y;
}
```

Checking the day value is more complicated. For example, if the month is February, its max value may be 28 or 29 (leap years). Therefore it is wise to add into our class one more function:

```
private:
int IsLeap();
```

The users of our class do not need this function and must not know anything about it. If we declare it as private, only the functions of our class can call it.

Objects and classes in C++ (7)

```
void Date::SetDay(int d)
{
  if (d < 1 || d > 31)
      throw "Wrong day";
  if ((iMonth == 4 || iMonth == 6 || iMonth == 9 || iMonth == 11) && d == 31)
      throw "Wrong day";
   if (iMonth == 2)
  {
       if (IsLeap(Year))
      {
            if (d > 29)
               throw "Wrong day";
       }
       else
      {
            if (d > 28)
                throw "Wrong day";
       }
   }
   Day = d;
}
```

Objects and classes in C++ (8)

```
int Date::IsLeap(int y)
{ // see https://www.programiz.com/c-programming/examples/leap-year
 if (Year % 4 == 0)
{
     if (Year % 100 == 0)
    {
         if (Year % 400 == 0)
             return 1;
        else
            return 0;
     }
     else
         return 1;
     }
     else
         return 0;
}
```

Objects and classes in C++ (9)

Now we can operate with objects of class *Date* as shown in the following examples:

```
Date d1, d2, d3;
d1.SetMonth(3);    d1.SetYear(2019);   d1.SetDay(5);
d2.SetMonth(5);    d2.SetYear(2019);   d2.SetDay(1);
printf("%d\n", d1.GetYear());
d3.SetMonth(d2.GetMonth() + 1);
```

or using the dynamic memory allocation:

```
Date *pd1, *pd2, *pd3;
pd1 = new Date;
pd2 = new Date;
pd3 = new Date;
pd1->SetMonth(3);    pd1->SetYear(2019);   pd1->SetDay(5);
pd2->SetMonth(5);    pd2->SetYear(2019);   pd2->SetDay(1);
printf("%d\n", pd1->GetYear());
pd3->SetMonth(pd2->GetMonth() + 1);
delete pd1;
delete pd2;
delete pd3;
```

Objects and classes in C++ (10)

If we define an object of the current implementation of class *Date*, we can call *SetDay()* only after calls to *SetMonth()* and *SetYear*(). This is because there is no any initialization of data members and the values of *iMonth* and *Year* right after allocation of memory are occasional numbers. To solve this problem we may declare *SetDay()*, *SetMonth()* and *SetYear()* as private and add into class declaration a new public function:

```
public:
void SetDate(int, int, int);
```

```
void Date::SetDate(int d, int m, int y)
{
   SetYear(y);
   SetMonth(m);
   SetDay(d);
}
```

Usage:

```
Date d;
d.SetDate(5, 3, 2019);
Date *pd = new Date;
pd->SetDate(5, 3, 2019);
```

Objects and classes in C++ (11)

So we have now class:

```
class Date
{
private:
  int Day, iMonth, Year;
  char sDate[12];
public:
  void SetDate(int, int, int);
  int GetDay();
  int GetMonth();
  int GetYear();
  void ToString();
  char *GetDateString();
private:
   int IsLeap(int);
   void SetDay(int);
   void SetMonth(int);
   void SetYear(int);
};
```

Objects and classes in C++ (12)

Instead of defining an object (i.e. allocating memory for its data members) and then calling function(s) to initialize it we almost always use specific member functions called constructors:

- The name of constructor matches the name of class.
- The constructor does not have output value (i.e. formally it returns nothing). But it may throw exceptions.
- The constructor is always public.
- The constructor may have arguments.
- A class may have several constructors, but in that case their sets of arguments must be different.
- If the programmer has not defined his own constructor, the compiler automatically creates the default constructor that has no arguments and initializes nothing. On our previous slides class *Date* had such a constructor.
- If the programmer has defined at least one constructor, the default constructor (actually a function with empty body) is not created. If such an additional constructor is still needed, the programmer must write it himself.
- Defining of an object as local or global variable or allocating for it dynamical memory is always accompanied with automatical call to constructor. On our previous slides

  ```
  Date d;
  Date *pd = new Date;
  ```

  meant that the default constructor was called. *malloc()* does not call constructor.

Objects and classes in C++ (13)

```
class Date
{
private:
  int Day, iMonth, Year;
  char sDate[12];
public:
  Date(); // default constructor, here we need to create it ourselves
  Date(int, int, int); // initializing constructor
  void SetDate(int, int, int); // needed if we want to change the values later
  int GetDay();
  int GetMonth();
  int GetYear();
  void ToString();
  char *GetDateString();
private:
   int IsLeap(int);
   void SetDay(int);
   void SetMonth(int);
   void SetYear(int);
};
```

Objects and classes in C++ (14)

```
Date::Date()
{ // empty constructor, also the default constructor
}
Date::Date(int d, int m, int y)
{
  SetYear(y);
  SetMonth(m);
  SetDay(d);
}
```

Now:

```
Date d1; // the default constructor is called, no initializations. No parenthesis here!
Date week1[7]; // the default constructor is called 7 times, no initializations
Date *pd1 = new Date; // the default constructor is called, no initializations
Date *pWeek1 = new Date[7]; // the default constructor is called 7 times, no initializations
```

Those 4 variable declarations are correct only if the default constructor is explicitly defined or if the programmer has not defined no one constructor.

```
Date d2(5, 3, 2019); // the initializing constructor is called
Date *pd2 = new Date(5, 3, 2019);
```

but

```
Date week2(5, 3, 2019)[7]; // error
```

Objects and classes in C++ (15)

The constructor arguments may have default values, for example:

```
 class Date
{
 ……..
public:
  Date(int = 1, int = 1, int = 2020);
 ……..
};
```

Now:

```
Date d; // if the default constructor is not defined, we get Jan 1, 2020
            // no parenthesis here!
            // error if the default constructor is defined (ambiguity)
Date *pWeek = new Date[7];
  // if the default constructor is not defined, we get 7 times Jan 1, 2020
  // error if the default constructor is defined (ambiguity)
```

Objects and classes in C++ (16)

The constructor without arguments (default constructor) may be not empty, for example:

```
Date::Date()
{
    time_t Now; // typedef time_t is declared in time.h
    time(&Now); // reads the system timer
    struct tm Tm; // struct tm is declared in time.h, details will be discussed later
    localtime_s(&Tm, &Now);
    Day = Tm.tm_mday; // in struct tm 1...31
    iMonth = Tm.tm_mon + 1; // in struct tm 0…11
    Year = Tm.tm_year + 1900; //in struct tm current year - 1900
}
```

Now:

```
Date d1; // d1 presents the current date
Date *pWeek = new Date[7]; // 7 times the current date
```

Objects and classes in C++ (17)

Starting from C++ v 11, the attributes may be initialized directly in the class definition.
Example:

```
class Matrix
{
 private: int nRows = 0, nColumns = 0; // default values
          double **ppMatrix = nullptr; // default value
  public: Matrix () { }
          Matrix(int, int);
……….
};
Matrix *pm1 = new Matrix; // empty constructor is called, attributes get default values
Matrix *pm2 = new Matrix(10, 10); // attributes get values corresponding to the
                                  // constructor actual parameters
```

Default value may be presented by any expression that is executable when the object is created. Example:

```
class Time
{
  private: time_t Now = time(&Now);   // call to the system timer
 ……….
};
```

Objects and classes in C++ (18)

Objects defined as local or global variables are removed automatically. Objects located on heap (region for dynamical memory allocation) are removed when we apply operator *delete*. If some of the attributes are pointers to memory fields then it may happen that some of the member functions have allocated the memory but have not released it. For example, our *Date* class may have a buffer for string:

```
class Date
{
private:
  char *pBuf;
…….
};
```

The memory for buffer is allocated when the date is converted into string. To ensure that the buffer will disappear together with the object we have to write a specific function called destructor:

- The name of destructor matches the name of class, but has prefix ~ (for example ~*Date*).
- The destructor has neither output value nor arguments.
- The destructor is always public.
- The destructor is always called automatically. The programmer has no right to call it.
- The task of destructor is to release the memory fields that were not released earlier, close the I/O connections, etc.

Objects and classes in C++ (19)

```
class Date
{
private:
  char *pBuf = 0;
public:
  Date(int, int, int);
  ~Date();
…….
};
Date::~Date()
{
  if (pBuf) // this cheking is absolutely necessary
      delete pBuf;
}
```

To avoid releasing memory that is already released or not allocated at all (i.e. to avoid crashes), the initial values of pointer member variables must be zero. If some of the member functions releases memory, it must also set the pointer to zero.

Objects and classes in C++ (20)

Short member functions may be declared as inline functions. Here it means that they are declared and defined in class declaration (i.e. in *.h file):

```
class Date {
private:
  int Day = 0, iMonth, = 0,  Year = 0;
  char sDate[12] = { 0 };
public:
  Date() { } // inline
  Date(int, int, int);
  void SetDate(int, int, int);
  int GetDay() { return Day; } // inline
  int GetMonth() { return iMonth; } // inline
  int GetYear() { return Year; } // inline
  void ToString();
  char *GetDateString() { return &sDate[0]; } // inline
private:
   int IsLeap(int);
   void SetDay(int);
   void SetMonth(int);
   void SetYear(int);
};
```

Objects and classes in C++ (20)

Normally the inline member functions do not need more than three or four rows of code. The place of long member functions is in *.cpp file.

Sometimes in small classes all the functions may be declared as inline and thus *.cpp file is not needed at all.

Some programmers declare all his functions without any regard to their length as inline. It is not recommended.

Arrays of objets

Let us have

```
Date *pWeek = new Date[7];
for (int i = 0;  i < 7;  i++)
{
  (pWeek + i)->SetDate(4 + i, 3, 2020);
}
delete[] pWeek;
```

To release memory occupied by an array of objects it is better to use *delete[]* instead of *delete*. It ensures that the destructors for each element of array are called.

Aggregation (1)

Let us have class

```
class Project
{
private:
  char *pTitle;
  Date Deadline;
  ……..
public:
  Project(const char *p) // calls automatically constructor Date()
                { // correct if Date has constructor without arguments or constructor in which
                  // all the arguments have default values
                 int n;
                 pTitle = new char[n = strlen(p) + 1]; // destructor must release it!
                 strcpy_s(pTitle, n, p);
                }
    ……..
};
```

One of the attributes is object of class *Date* or in other words, class *Date* is aggregated into class *Project*.

Aggregation (2)

If we want also the set the deadline, we have to add one more constructor:

```
Project::Project(const char *p, int d, int m, int y) : Deadline(d, m, y)
{  // calls automatically constructor Date(int, int, int)
      int n;
       pTitle = new char[n = strlen(p) + 1];
       strcpy_s(pTitle, n, p);
  }
Project:: ~Project()
{// destructor of the aggregated class is called automatically
  delete pTitle;
// here the checking of pTitle is omitted because the memory was allocated by constructor
}
```

Now suppose that

```
class Project
{
private:
      Date *pDeadline;
  ……….
};
```

This is also aggregation but without automatic call to the constructor of aggregated class. The constructor of *Project* must explicitly allocate memory for deadline.

Aggregation (3)

```
Project::Project(const char *p, int d, int m, int y)
{
      int n;
      pTitle = new char[n = strlen(p) + 1];
      strcpy_s(pTitle, n, p);
      pDeadline = new Date(d, m, y);
  }
Project::Project(const char *p)
{
      int n;
      pTitle = new char[n = strlen(p) + 1];
      strcpy_s(pTitle, n, p);
      pDeadline = new Date;
  }
Project:: ~Project()
{ // checking not needed because memory is allocated in constructor
  delete pTitle;
  delete pDeadline;
}
```

Inheritance (1)

Let us have an 100% implemented class:

```
class Person
{
  char *pName;
  char * pAddress;
  Date Birthdate;
  char *pNationality;
  long long int Code;
  ……….
};
```

We need to create class *Student*. As all the software implemented in *Person* is applicable and needed in *Student*, we may use aggregation:

```
class Student
{
  Person *pPersonalData;
  ……. // speciality, examinations etc.
};
```

Inheritance (2)

Each student has a lot of common attributes. But for example a future engineer must work as intern in industry, a future trainer must have some achievements in sport, a future actor must have some role in theater or film, etc. Those specific attributes must also be presented in the class declaration. But

```
class FutureEngineer
{
  Student *pStudent;
  ……..
};
```

is a very awkward solution because we have now data on three levels: *FutureEngineer*, *Student* and *Person*, for example in class *FutureEngineer* to get the name we have to write

```
pFutureEngineer->GetStudent()->GetPersonalData()->GetName();
```

OPP has better solution: derive a new class from an existing class - the derived class inherits all the attributes and methods of the base class and may use them as its own attributes and methods. If we have derived *Student* from *Person*, attributes *pName*, *pAddress* etc. become automatically members of *Student*. If we continue in this way and derive *FutureEngineer* from *Student*, the *FutureEngineer* inherits all the attributes and methods from *Student* as well as from *Person*. So, attribute *pName* and method *GetName()* implemented in *Person* may be used in *FutureEngineer* in the same way as the attributes and methods declared in *FutureEngineer* itself, for example

```
printf("%s\n", pFutureEngineer->GetName());
```

Inheritance (3)

To declare a derived class write:

```
class derived_class_name : deriving_mode base_class_name
{ ..................... };
```

There are three deriving modes: *public* (used mostly), *private* and *protected*. Example:

```
class Employee {
  char *pName; // private by default
  public:
    Employee() {  pName = 0; }
    ~Employee() {
                  if (pName)
                      delete pName;
                 }
    const char *GetName() { return pName; }
    void SetName(const char * p) {
                  if (pName)
                      delete pName;  // remove the current name (if exists)
                  int n;
                  pName = new char[n = strlen(p) + 1];
                  strcpy_s(pName, n, p);
                 }
};
```

Inheritance (4)

```
class HourlyEmployee : public Employee
{
   int HourlyWage;
public:
    HourlyEmployee(int hw) { HourlyWage = hw; }
    int ToPay(int hours) { return hours * HourlyWage; }
};
```

Now:

```
HourlyEmployee *pJohn = new HourlyEmployee(10);
                 // constructor of Employee is called automatically
pJohn->SetName("John Smith");
                // SetName() is now also the member of HourlyEmployee
printf("%s gets $%d per week\n", pJohn->GetName(), pJohn->ToPay(40));
                // GetName() was declared in Employee
                // ToPay() was declared in HourlyEmployee
delete pJohn;
                // destructor of Employee is called automatically
```

Inheritance (5)

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

```
derived_class_name :: derived_class_name (full_list_of_parameters) :
                              base_class_name(sublist_of_base_class_constructor_parameters)
{ constructor_body }
```

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

```
derived_class_name :: derived_class_name (full_list_of_parameters)
{ constructor_body }
```

Example: rewrite the constructor of class *Employee*:

```
Employee(const char * p)
                {
                  int n;
                  pName = new char[n = strlen(p) + 1];
                  strcpy_s(pName, n, p);
                 }
```

Now we have to rewrite the constructor of *HourlyEmployee* too:

```
HourlyEmployee(int hw, const char *p) : Employee(p) {  HourlyWage = hw;  }
```

Inheritance (6)

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

In our example the base class *Employee* has destructor, but there is no need to write a destructor for derived class *HourlyEmployee*. But it does not mean that the destructor of *Eployee* is not called.

If the authors of base class and derived class are different (and rather often the programmer implementing derived class has no access to the code of base class, he has only the *.lib or *.obj file and a document describing the public attributes and methods), the member names of base class and derived class may match. Suppose that the base class *bbb* has attribute *Attr* ja method *void fun()*. Suppose that the derived class *ddd* has also attribute *Attr* ja method *void fun()*. Then in software written for class *ddd*:

- To access *Attr* and *fun()* from class *ddd* write simply *Attr* and *fun()*. It means that if we do not have access to base class code, we need not worry about matching names.
- To access *Attr* and *fun()* from class *bbb* write *bbb::Attr* and *bbb::fun()*.

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

Inheritance (7)

Users of a class cannot access private members. As the programmer implementing a derived class is also the user, we have a paradoxical situation: private members of base class are also members of derived class, but their usage is impossible. To solve the problem in OOP an additional access mode is introduced: a class member may be *protected*.

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 everywhere 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*.

Polymorphism (1)

Suppose we want to implement a package of functions for drawing figures consisting of geometric shapes like circle, triangle, rectangle, polygon, etc. All those figures have several attributes that are common for all of them, for example the color, width and type (continious, dashed) of line used for draw them, color to fill them, etc. They all have at least one point specifying their location. So we can create a class:

```
class Shape
{
 protected:
      int x, y,
          LineColor, FillColor, // for example like #define BLACK 0 #define RED 1
          LineWidth,
          LineType, // for example like #define CONTINIOUS 0 #define DASHED 1
          ….. ……..;
 public:
       Shape() { }
       Shape(int i1, int i2, …..) { x = i1; y = i2; …… }
       ……
};
```

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

Polymorphism (2)

The operations we want to perform with shapes are drawing, deleting, scaling, moving, etc. The algorithms of those operations are also common for all of the shapes, so we can add to our abstract class:

```
class Shape
{
  ……..
  void Draw() { } // empty because shape is just an abstraction
  void Delete() {
          int c1 = LineColor, c2 = FillColor;
          LineColor = FillColor = ::GetBackgroundColor();
          Draw();
          LineColor = c1;  FillColor = c2;
         }
   void Move(int newX, newY) {
           Delete();
           x = newX; y = newY;
           Draw();
          }
    ……..
};
```

Polymorphism (3)

Now we can derive our actual shapes, for example:

```
class Circle : public Shape
{
    int radius;
  public:
    Circle(int x, int y, int r, …..) : Shape(x, y, …..) { radius = r; }
    void Draw() { ……. } // overrides Draw() from base class
};
```

```
Circle *pCircle = new Circle(0, 0, 100);
pCircle->Draw(); // works, we can see the circle on screen
pCircle->Delete(); // does not work, the circle is still there
pCircle->Move(10, 10); // does not work, nothing is moved
```

The reason is that *Delete()* and *Move()* inherited from Shape call *Draw()* defined in *Shape*, but is does nothing.

The problem is: how to replace empty *Draw()* from *Shape* called in methods *Delete()*, *Move()*, etc. that are also from *Shape* with calls to *Draw()* from *Circle* when we are handling circles or with calls to *Draw()* from *Rectangle* when we are handling rectangles, etc.

Polymorphism (4)

Solution: declare *Draw()* as a virtual function:

```
virtual void Draw();
```

We have *Draw()* in base class and its overriding *Draw()* methods in derived classes. During application building, when the call to *Draw()* is encountered, the compiler and / or linker meet a problem: which *Draw()* to bind. They can do it (early or static binding) or left the question open (late or dynamic binding). In the last case the method to call is selected when the application is running.

If a function is declared as virtual, it and all its overriding functions will be bound dynamically. The method to call (*Draw()* from *Circle* or *Draw()* from *Rectangle*) depends on the type of object associated with that call. For example:

```
pCircle->Delete();
```

call to method *Delete()* from *Shape* is associated with object of type *Circle*. *Delete()* calls *Draw()* but as *Draw()* is virtual, bindings were open. As *pCircle* points to *Circle*, in runtime binding the call will be bound to *Draw()* from *Circle*. If on next row is expression

```
pRectangle->Delete();
```

then when executing it the call to *Draw()* will be bound to *Draw()* from *Rectangle*.

Generally, C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Polymorphism (5)

Why the selection which of the overrided methods to call is performed in runtime? The problem is that:

```
Circle *pCircle = new Circle(0, 0, 100, ….);
Rectangle *pRectangle = new Rectangle(10, 20, …..);
Shape *pShape;
if (….)
    pShape = pCircle; // without explicit cast!
                      // it is correct because all the attributes and methods from Shape are
                      // also the attributes and methods of Circle.
else
    pShape = pRectangle;
pShape->Delete();
  // correct but during building we do not know to which object pShape will point
delete pShape; // if the destructor of Shape is virtual, it is called and the destructor of Circle
                  // or Rectangle also. If not then due to early binding only the destructor
                  // of Shape is called
```

Recommendation: always include a virtual destructor into your class, even if it empty

Mark that

```
pCircle = pShape; // error
pCircle = (Circle *)pShape; // formally correct, but may lead to crash
```

Polymorphism (6)

If a virtual function has return value, it cannot be empty:

```
virtual int fun() { } // error
```

So we have to return something, for example

```
virtual int fun() { return 0; }
```

It is better to use pure virtual functions in which instead of body we write *=0;*

```
virtual int fun() = 0;
```

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.