Module 3: Constructor - Destructor

Prof. Tran Minh Triet

Acknowledgement

- Slides
  - Course CS202: Programming Systems
    Instructor: MSc. Karla Fant,
    Portland State University
  - Course CS202: Programming Systems
    Instructor: Dr. Dinh Ba Tien,
    University of Science, VNU-HCMC
  - Course DEV275: Essentials of Visual Modeling with UML 2.0
    IBM Software Group

Outline

- Constructors
- The **this** pointer
- Destructor
- Member Initialization
- Copy constructor
- Assignment operator

Constructors

- Remember that when you define a local variable in C++, the memory is not automatically initialized for you.
- This could be a problem with classes and objects
- Luckily, with a constructor we can write a function to initialize our data members and have it implicitly be invoked whenever a client creates an object of the class

Constructors

- Constructor is a physical piece of code (in fact, it is a special type of method) that is used to construct and initialize objects.
- It is automatically invoked when a new object is created.
- There is no returned value, even a void.
- A class can have many constructors (overload)
- Name of the constructors must be the same as the class name.

Notes on constructors

- Default Constructor
- Constructor with no parameters
- Constructor with parameter(s)
- Constructor with default parameter(s)

Notes on constructors

- If no constructor is implemented, the compiler will issue a default constructor
- The default constructor:
  - No argument
  - Invoke other default constructors of data members if they are objects.
  - Doesn't initialize other data members if they are not objects.

Default constructor

- If there is at least one constructor, the default constructor will not be created by the compiler

```
class CDate
{
public:
   CDate(int iNewDay);
   ...
private:
   ...
};
```

```
int main()
{
    CDate today; //error
    ...
    return 0;
}
```

- Advice: always define your own default constructor!

Other constructors

- They allow users different options to create a new object

```
class  CDate
{
private:
   int m_iDay, m_iMonth, m_iYear;
public:
   CDate();
   CDate(int, int);
   CDate(int, int, int);
   …
};
```

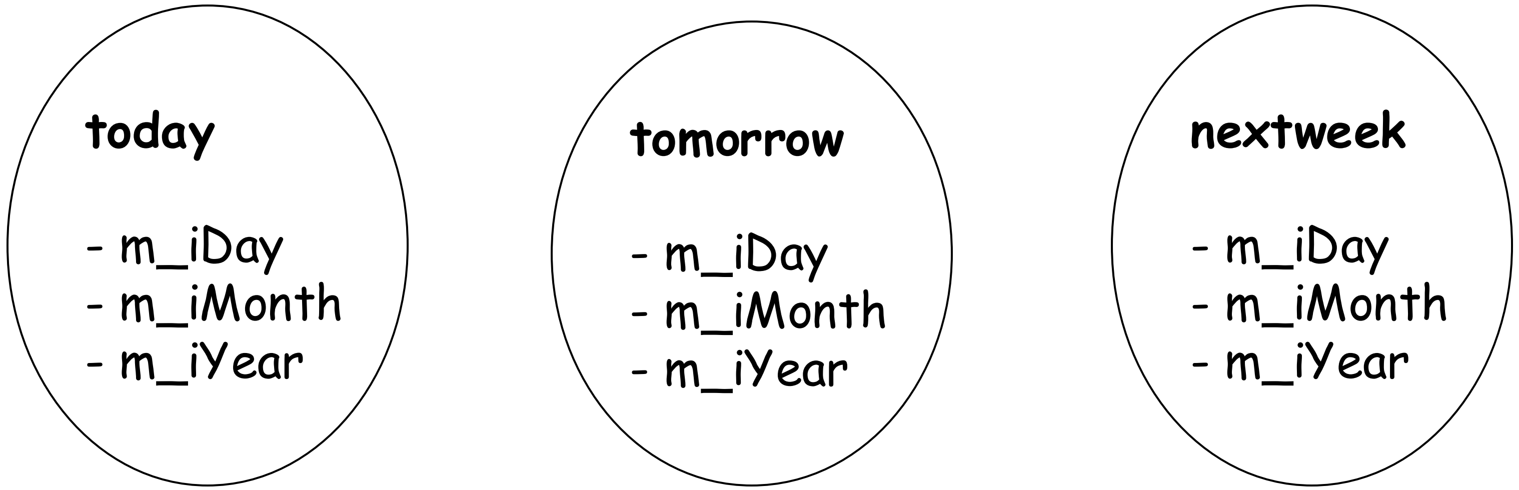
The this pointer

- Check the following lines of code, are they correct in terms of: syntax? semantics? useful?

```
CDate::CDate(int m_iDay, int m_iMonth, int m_iYear)
{
        m_iDay = m_iDay;
        m_iMonth = m_iMonth;
        m_iYear = m_iYear;
}
```

```
CDate today(4, 11, 2009);
```

The this pointer



```
int CDate::GetMonth()
{
    return m_iMonth;
}
```

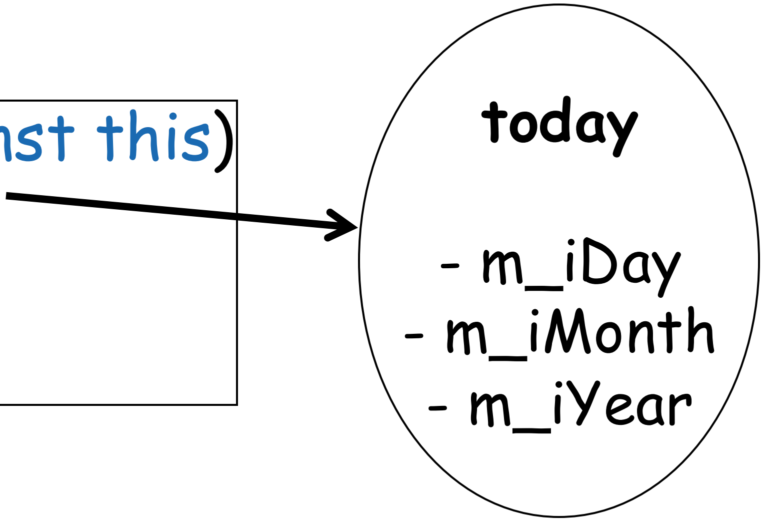
- How can we know **day**, **month** or **year** of which object are using?

The this pointer

- C++ adds an implicit function parameter - the pointer to the current object instance: **this**
- **this** is a constant pointer, you cannot modify it within a member function.

```
int month = today.GetMonth();
```

```
int CDate::GetMonth(CDate* const this)
{
    return this->m_iMonth;
}
```



The this pointer

```
CDate::CDate(int m_iDay, int m_iMonth, int m_iYear)
{
        m_iDay = m_iDay;
        m_iMonth = m_iMonth;
        m_iYear = m_iYear;
}
```

- Syntax: correct
- Semantic: legal
- Useful: NO!!!

The code should be

```
CDate::CDate(int iDay, int iMonth, int iYear)
{
        this->m_iDay = iDay;
        this->m_iMonth = iMonth;
        this->m_iYear = iYear;
}
```

Destructor

- We can deallocate the memory when the lifetime of a list object is over
- When is that?
- Luckily, when the client's object of the list class lifetime is over (at the end of the block in which it is defined) -the destructor is implicitly invoked
- So, all we have to do is write a destructor to deallocate our dynamic memory

Destructor

- Invoked automatically, when the variable is removed from memory (e.g. goes out of scope).
- Each class can have at most one destructor
- The destructor name is a name of a class preceded by a tilde sign (~).
- Destructor, the same as constructor, has no return type (even void)
- Destructor frees the resources used by the object (allocated memory, file descriptors, semaphores etc.)

Example

```
class CDate
{
private:
   int m_iDay, m_iMonth, m_iYear;

public:
   CDate();
   CDate(int, int, int);
   ~CDate();
   …
};
```

Members Initialization

- Distinguish between Assignment and Initialization

```
CDate::CDate(int iNDay, int iNMonth, int iNYear)
{
        m_iDay = iNDay;
        m_iMonth = iNMonth;
        m_iYear = iNYear;
}
```

- This is Assignment, not Initialization

Members Initialization

- This is members initialization

```
class CDate
{
private:
   int m_iDay, m_iMonth, m_iYear;
public:
   CDate();
   CDate(int iNDay, int iNMonth, int iNYear)
           : m_iDay(iNDay),
           m_ iMonth(iNMonth),
           m_ iYear(iNYear)
   {}
   virtual ~CDate();
};
```

Mandatory Members Initialization

- Const members
- References
- Sub-objects which require arguments in constructors

Mandatory Members Initialization

```
class CTest
{
private:
 Another&    refA; // reference member
 const int   MAX; // const member
 vector      arr;
public:
 CTest(Another& r) : refA(r), MAX(100), arr (MAX) {}
};
```

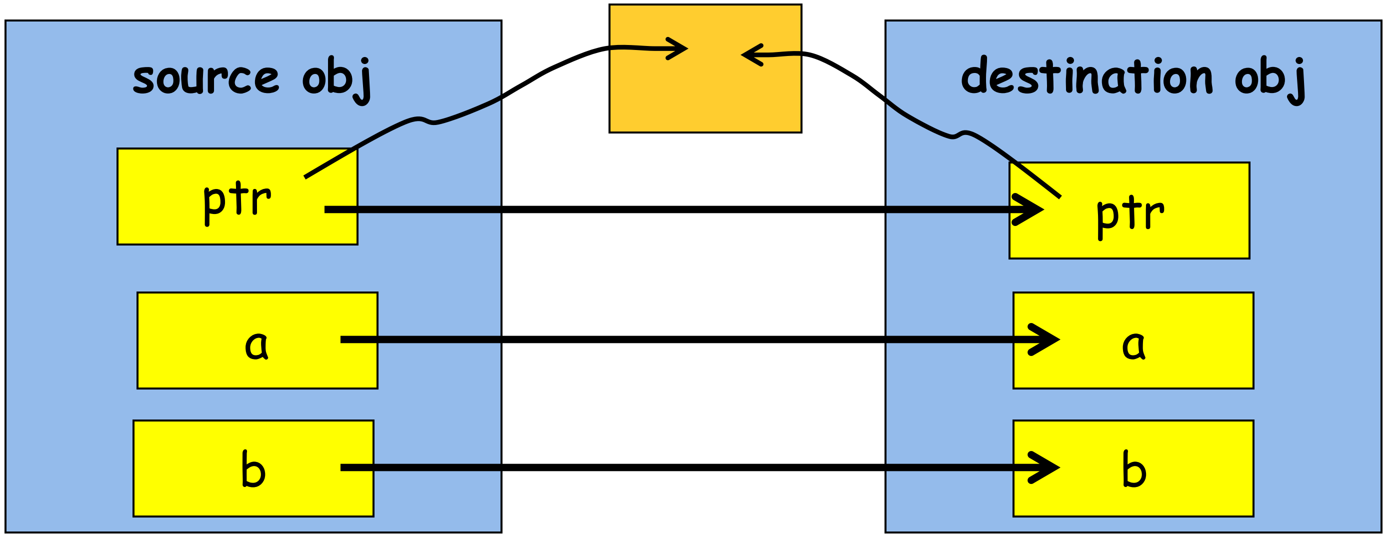
Default copy constructor

- In each class, if there is no copy constructor, a default copy constructor will be generated. It helps to create a new object of this class from another object. For example:

```
int main()
{
   CRectangle          a;
   CRectangle          b(a); // invoke copy constructor
   CRectangle          c = a; // invoke copy constructor
}
```

Default copy constructor (cont.)

- Default copy constructor performs a bitwise copy from the source to the current object:



Copy constructor

- Due to the bitwise copy of the default constructor, it will cause a serious problem if the copying takes place when the object has a member pointer with a dynamic allocated memory.
  - Pointers of the source obj and the destination obj will refer into the same memory

Copy constructor

- Depending on the members of the class to decide whether to have a copy constructor
  - When having dynamic allocated members

```
CTest::CTest(const CTest& src)
{
        iSize = src.iSize;
        ptr = new int [iSize];
        for (int i=0; i<iSize; ++i)
                ptr[i] = src.ptr[i];
};
```

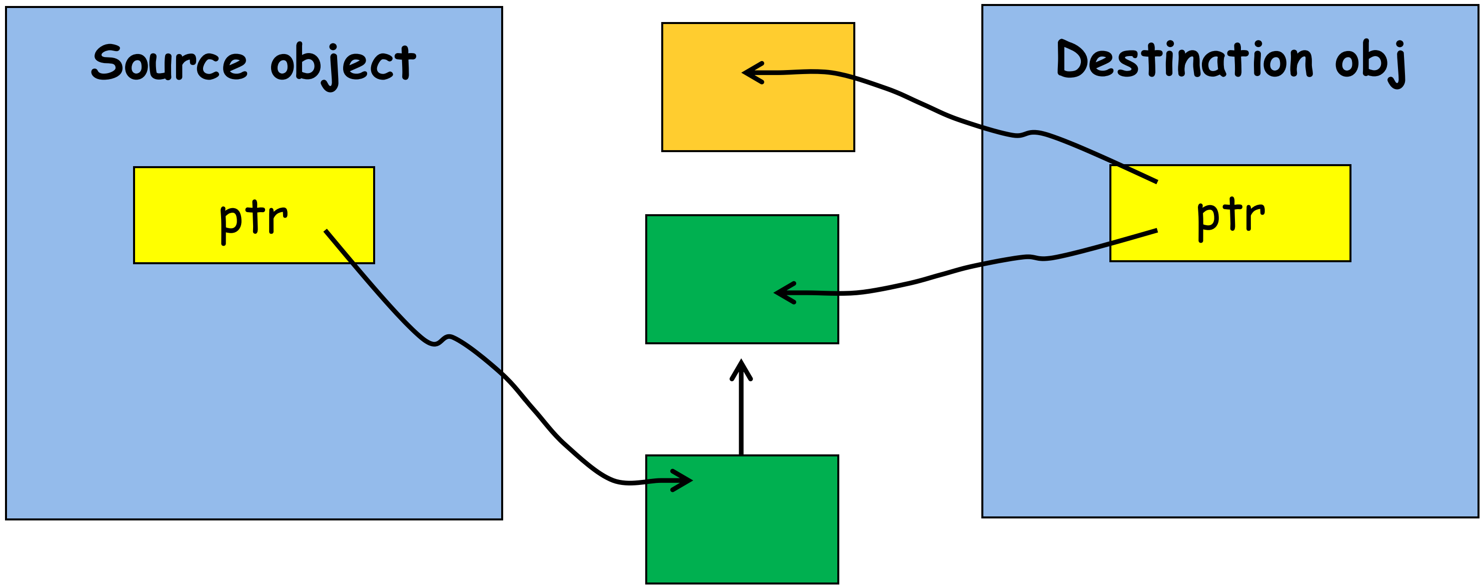
The default assignment operator

- Similar to the default copy constructor, in each class, if there is no assignment operator, a default assignment operator will be generated
- It also has a similar functionality of a default copy constructor, i.e. doing a bitwise copy from the source object to the destination object.

Assignment operator

- Thus, if there is a pointer member in the class, an assignment operator should be defined.
- Note: assignment operator is a bit different from the copy constructor:
  - Clean up the allocated memory that the pointer member is pointing to before being allocated with a new memory.
  - Remember to check for self-assignment

Assignment operator



- Clean up the memory it is pointing to
- Copy the memory to a new place

For example

```
CTest& CTest::operator=(const CTest& src)
{
   if (this != &src)
   {
         delete [] ptr;
         iSize = src.iSize;
         ptr = new int [iSize];
         for (int i=0; i<iSize; ++i)
               ptr[i] = src.ptr[i];
   }
   return *this;
}
```

Remember

The 3 following functions often go together:

- Copy constructor
- Assignment operator
- Destructor