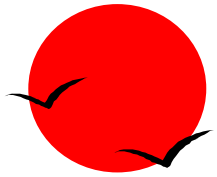


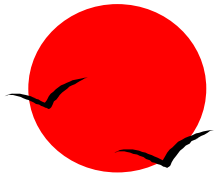
Middleware technology

Chapter 3 Interoperation between two binary executable modules in object-oriented programming paradigm

- CSEE Hunan university -

Jin-Min Yang

2016.02



同一进程内模块之间的互操作

互操作中的2个问题：1）**代码和数据定位；2）参数传递；**

计算机**计算原理**：以数据和代码的内存地址来定位；

挑战：代码要求数据和代码的内存地址，以及数据和代码的内存地址无法事先确定（原因：模块的独立性，版本升级性）；

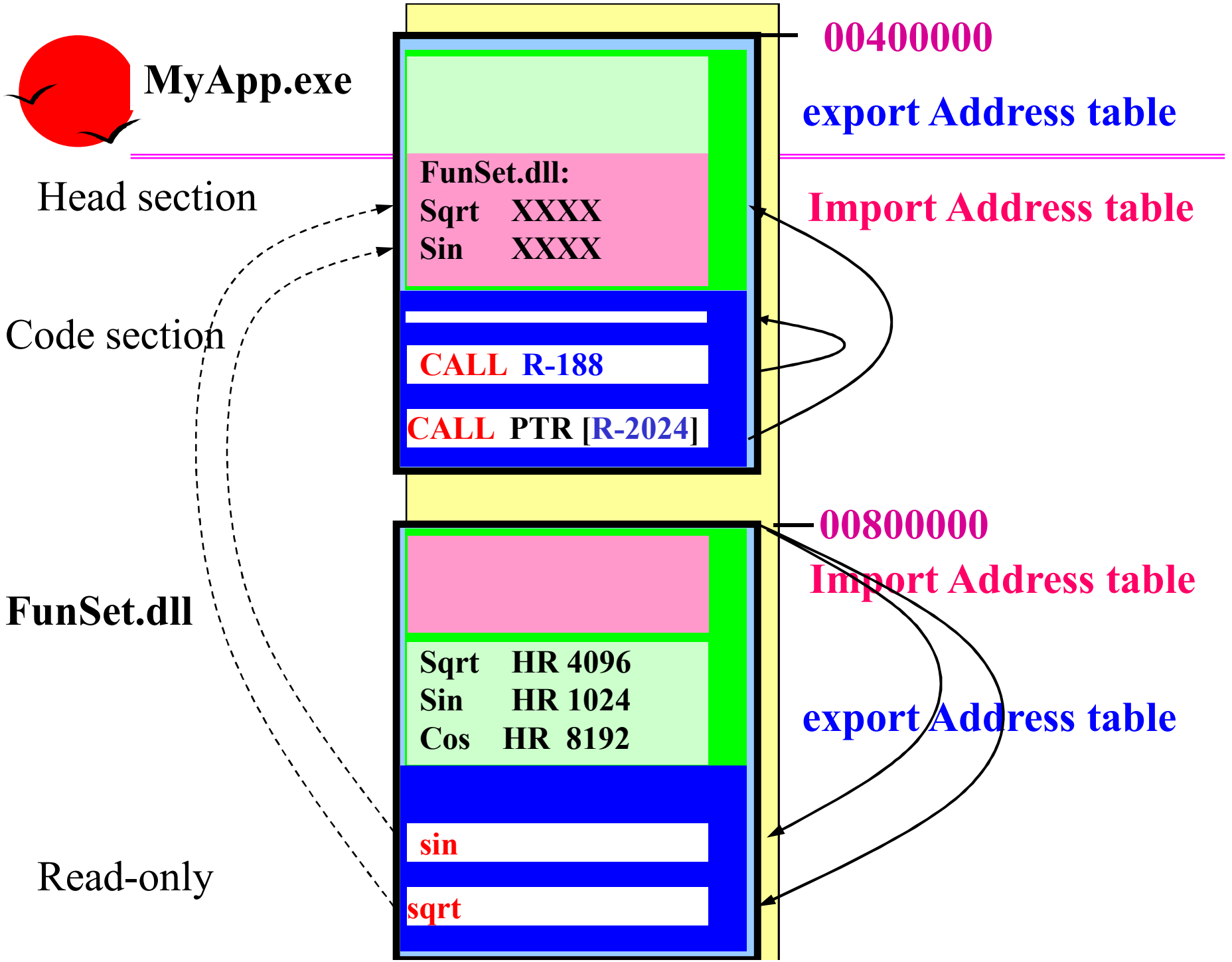
1) 参数传输： 媒介，长度，顺序；
2) 返回值：媒介，和长度；
3) 谁负责**清除**栈中传递的参数；
4) 函数命名规则；
5) 寄存器的保存与恢复；

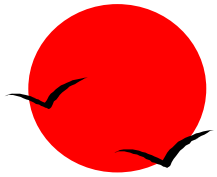
参数传递**媒介**： stack, register

通过名字定位；

策略： 导出地址表(callee)；
导入地址表(caller)；

实现： Compiling；
Linking；
Loading；


MyApp.exe
Head section
Code section
FunSet.dll
Read-only
00400000
export Address table
FunSet.dll:
Sqrt XXXX
Sin XXXX
Import Address table
CALL R-188
CALL PTR [R-2024]
00800000
Import Address table
Sqrt HR 4096
Sin HR 1024
Cos HR 8192
export Address table
sin
sqrt



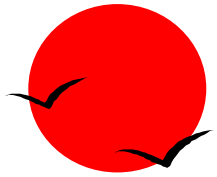
二进制模块之间的互操作

两个**二进制模块**在一个进程中来进行互操作：

互操作**形式**：**函数调用；**

互操作**契约**(共识)：**函数调用规范；**

契约**载体**：**.h 文件；**



互操作契约

Vendor:

在服务方的头文件(support.h)中：

```
int __declspec(dllexport) __cdecl MyFunc( char *p, double f );
```

6要素

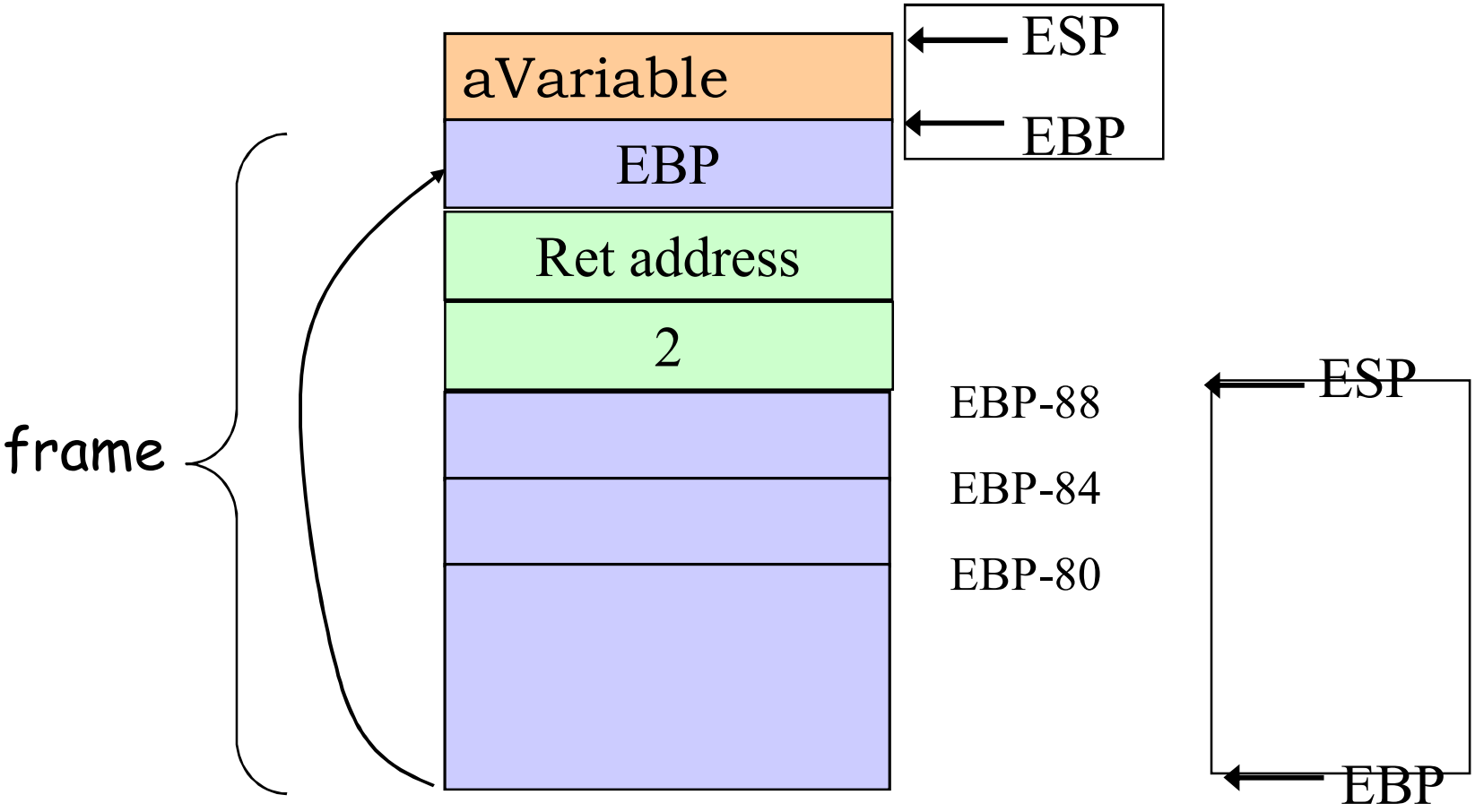
Client:

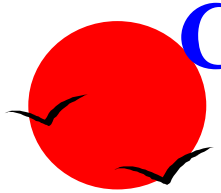
提供给客户的头文件(support.h)中：

```
int __declspec(dllimport) __cdecl MyFunc( char *p, double f );
```

Stack结构－链表结构

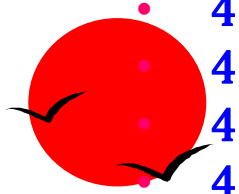
Stack frame(栈帧)





Code re-entry, function recursive calling, stack ,Thread

```
int g_myGlobalVariable;
int main( int argc, char *argv[ ] )  {
   char szBuf [128];
   char *psz = "Hello";
   unsigned long   p = 2;

   g_myGlobalVariable = 0x12345678;
//Procedure invocation :
  MyFunction( p );
}
```

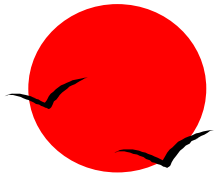


超越微软

```
401000:   PUSH    EBP
401001:   MOV     EBP,  ESP
401003:   SUB     ESP, 00000088
401009:   PUSH    EDI
40100A:   MOV     DWORD PTR [EBP- 00000084],  00406030
401014:   MOV     DWORD PTR [EBP- 00000088],  00000002
401036:   MOV     DWORD PTR [004088E8], 12345678
4010E1:   MOV     EAX, DWORD PTR [EBP - 00000088]; PUSH EAX
4010F9:   CALL    DWORD PTR [0x00405030]
4010FE:   ADD     ESP, 4
```

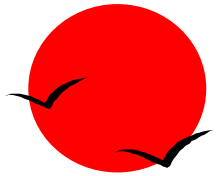
超越的意义何在?

```
401000:   PUSH    EBP
401001:   MOV     EBP,  ESP
401003:   SUB     ESP, 00000088
401009:   PUSH    EDI
40100A:   MOV     DWORD PTR [EBP- 00000084],  EDX +6030
401014:   MOV     DWORD PTR [EBP- 00000088],  00000002
401036:   MOV     DWORD PTR [EDX + 78E8], 12345678
          PUSH    EDX
          MOV     ECX, EDX
4010E1:   MOV     EAX, DWORD PTR [EBP - 00000088]; PUSH EAX
          MOV     EDX, DWORD PTR [ECX + 4096]
4010F9:   CALL    DWORD PTR [ECX + 4030]
4010FE:   ADD     ESP, 4
          POP     EDX
```



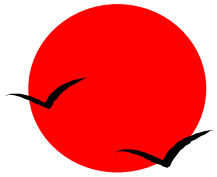
Microsoft platform

Web service		WSDL (XML) + UDDI
.NET library	MSCorEE.dll	Metadata + reflection
COM library	OLE32.DLL	Type library: IDL
MFC library	Mfc**.dll	.h 文件
C runtime library	mscrt40.DLL	Tight couple
OS API	kernel32.DLL	



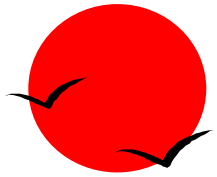
Contents

- ✓ Basic concepts in object-oriented programming;
- ✓ Problems interoperation in binary-level in object-oriented paradigm;
- ✓ Solution scheme: component model:
  - Separating interface class from implementation class;
  - Construction and deconstruction of object;
  - Extension of interface;
  - Object lifetime management;



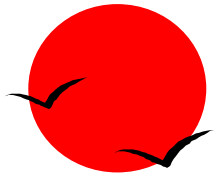
Why is the concept of object-oriented programming brought up?

- In order to **decrease the cost** of software development, **cut down the time** of software development, **ease the degree of difficulty** of software development, and **improve the quality** of software, software reusability is desired, especially in binary level.
- The background of Procedure-oriented paradigm:
  - A programmer only use several modules from external world. Due to small scale of application program, it is enough for a programmer to be concerned with the linear **procedure** and its **sections** in an application;



Why is the concept of object-oriented programming brought up?

- Since a application becomes **more and more compound in its procedures and composite elements**, and a programmer use **more and more modules from various sources** , **the flat organization structure** of the functions in procedure-oriented paradigm leads to such problems as **naming conflictions** and **inconsistency**.
- The flat organization structure of the functions in procedure-oriented must give place to **a hierarchical organization structure** to solve **the problem of function positioning**;

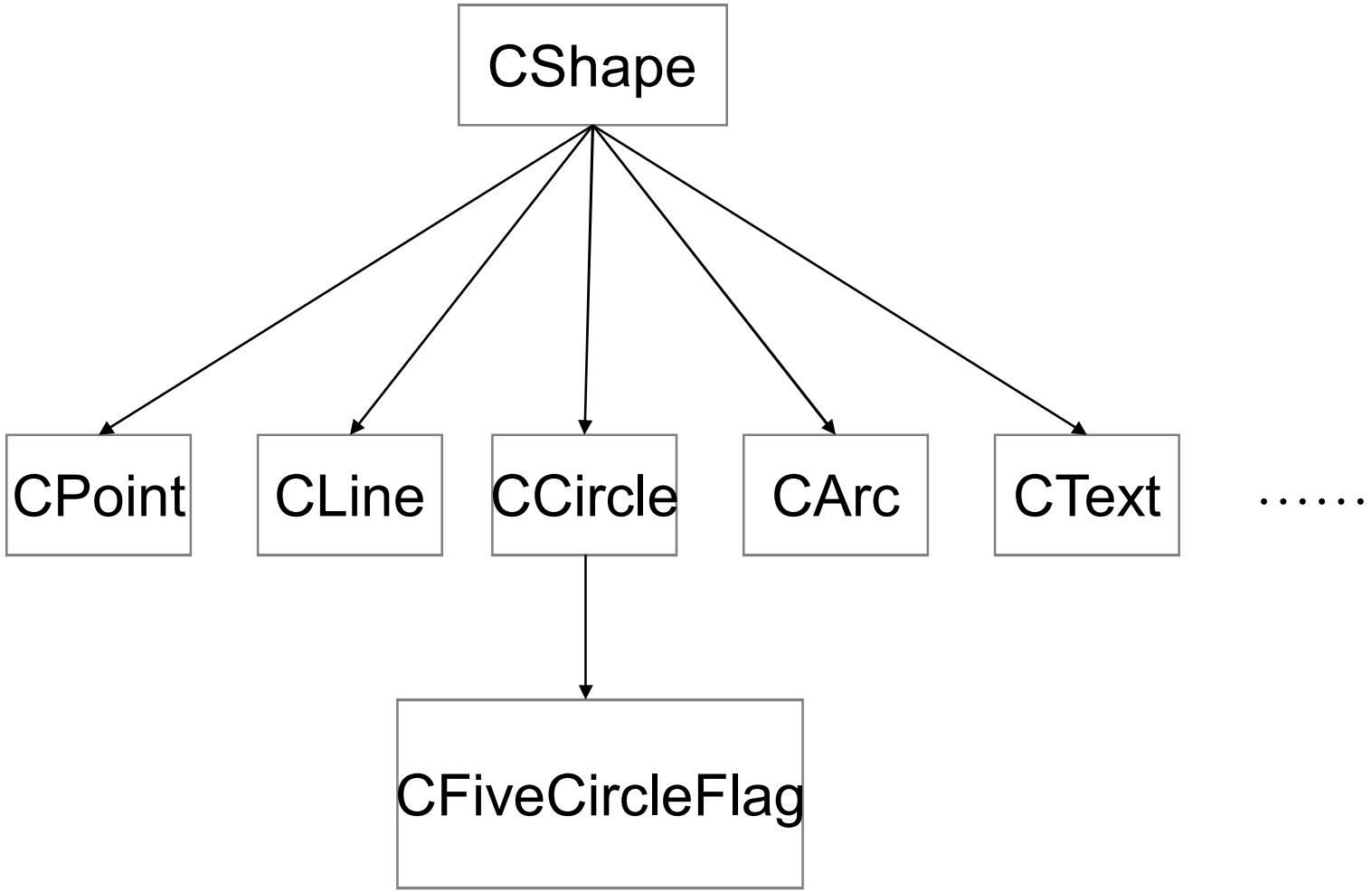


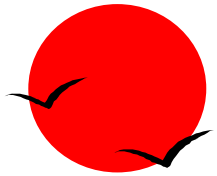
Basic concepts in object-oriented programming

In C++:

- the constructor and deconstructor functions of a class: have no return type and return value ;
- Types of a data member of a class: static, nonstatic;
- Types of a member function of a class: static、nonstatic (conventional, virtual, pure virtual);
- polymorphism;
- encapsulation;
- Object lifetime;

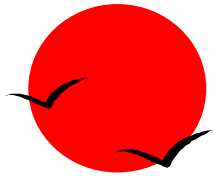
Demonstration of Object-Oriented Programming





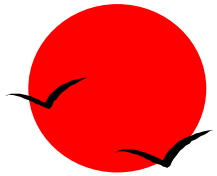
Polymorphism and virtual function

- Polymorphism is the core concept of object-oriented programming.
- We use the following example to demonstrate its essence.



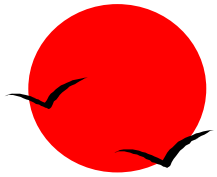
Demonstration of Polymorphism essence

```
Class CShape {
        public:
          virtual void Function1(    )   { };
          virtual void Display(CDC* pDC)   { };
}
Class CCircle: public Class CShape {
       public:
          CPoint CenterPoint;
           int  Radius;
          virtual void Function1(    )   { };
          virtual void Display(CDC* pDC);
}
Class CFiveCircleFlag: public Class CCircle {
        public:
            Cpoint CCP;
            int CR;
            virtual void Function1(    );
           virtual void Display(CDC* pDC);
}
```



Demonstration of Polymorphism essence (cont.)

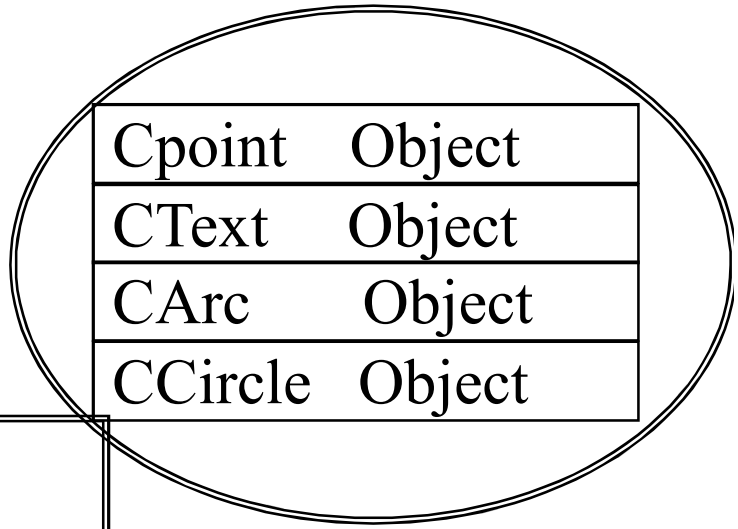
```
virtual void CCircle::Display (CDC* pDC)   {
        pDC->DrawCircle(CenterPoint.x, CenterPoint.y,
                Radius;
}

virtual void CFiveCircleFlag:: Display (CDC* pDC)   {
        pDC->DrawRectangle(CCP.x, CCP.y, CR,CR);
        CCircle::Display(pDC);
        …;
}
```



The significance of Polymorphism

- Polymorphism application:
- void CDocument::OnDraw(CDC* pDC) {

```
    CShape  *pShape ;
    pShape = GetFirstShape( );
     while (pShape <> null)  {
            pShape->Display( pDC);
            pShape = GetNextShape( );
     }
}
```



Question: what is the display result ?

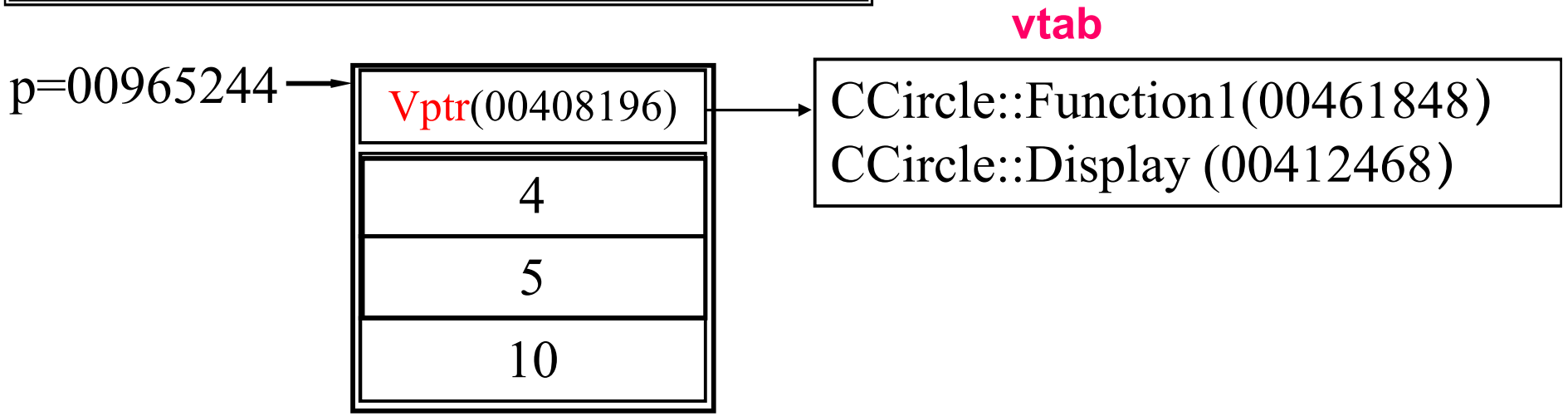
If Display( ) is not defined as a virtual function， what is the display result of the document?

Polymorphism implementation technique (Vptr/vtab)

CCircle *p = new CCircle(Cpoint(4,5),10);

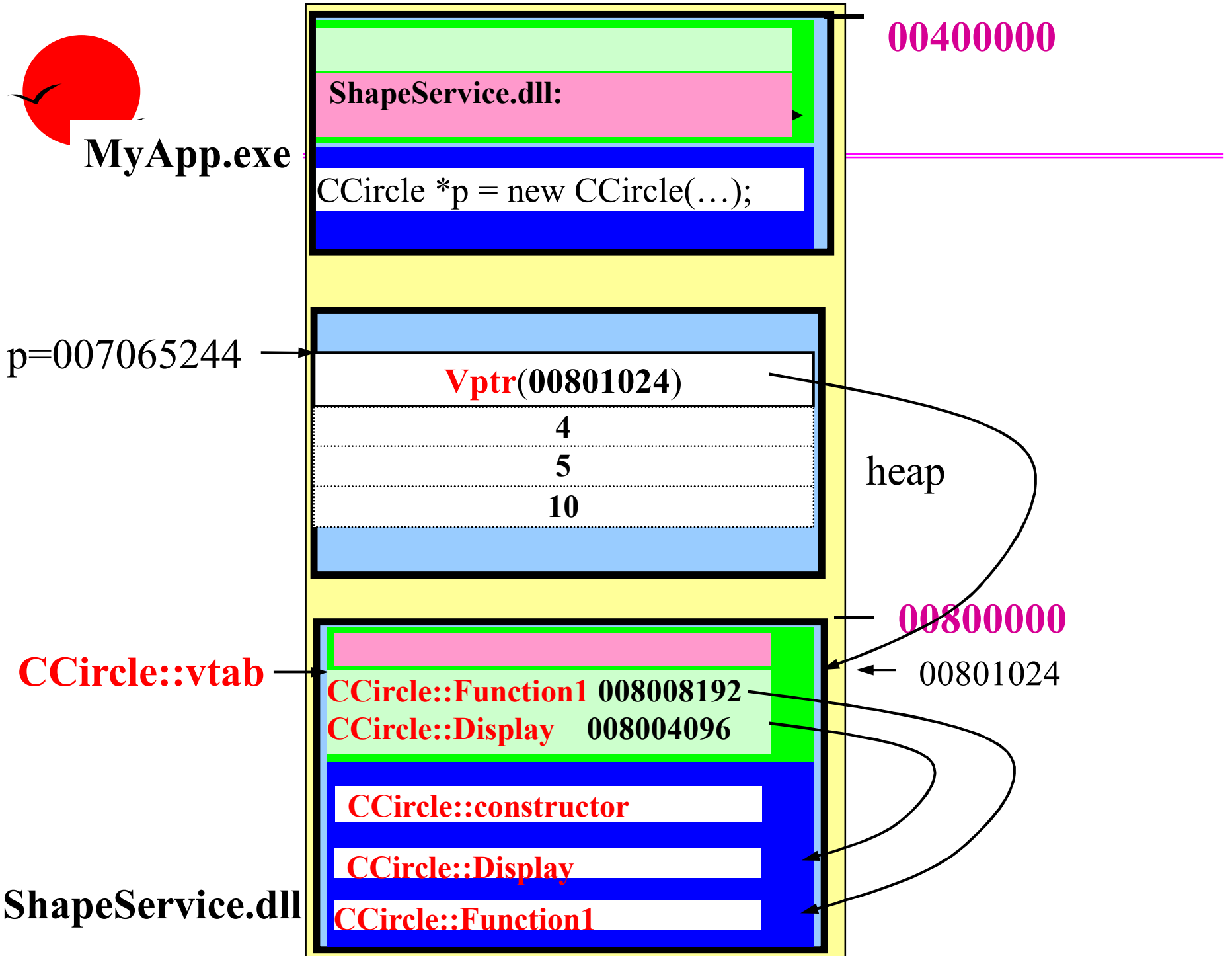
CShape *p0;
p0=p;

Object memory Layout:



Question: in which kind of memory is function code, Vtab, object , respectively?

Who is responsible for filling in the Vptr in a object?


00400000
MyApp.exe
ShapeService.dll:
CCircle *p = new CCircle(…);
p=007065244
Vptr(00801024)
4
5
10
heap
00800000
00801024
CCircle::vtab
CCircle::Function1 008008192
CCircle::Display 008004096
CCircle::constructor
CCircle::Display
CCircle::Function1
ShapeService.dll

Polymorphism implementation technique (indirect calling)

```
void CDocument::OnDraw(CDC* pDC)   {
    CShape  *pShape ;
   pShape = GetFirstShape(  );
    while (pShape <> null)  {
          pShape->Display( pDC);
           pShape = GetNextShape(  );
    }
}
```

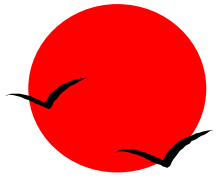
For a virtual function，the assembly generated by compiler as follows :

```
MOV CDX, DWORD PTR [pShape];
CALL DWORD PTR[PTR [CDX] +4];
```

Indicate the second virtual function

For non-virtual function，the assembly generated by compiler as follows :

```
CALL  the real address of Cshape::Display( );
```

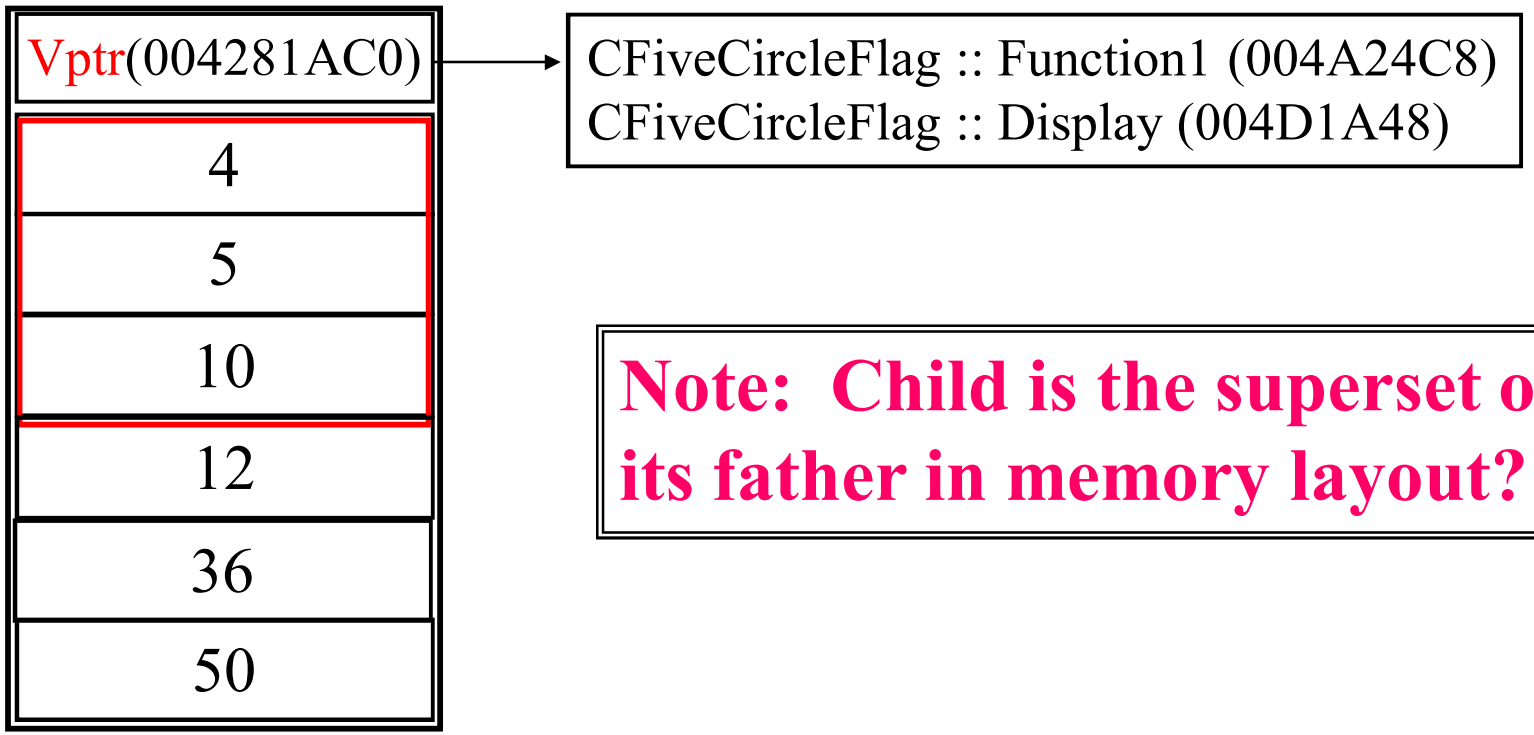


Polymorphism implementation technique (Vptr/vtab)

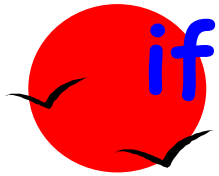
CFiveCircleFlag *p = new CFiveCircleFlag(Cpoint(12,36),50);

Object memory Layout:

p=009ABC20



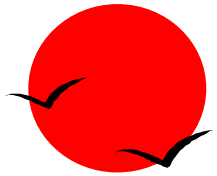
Note: Child is the superset of its father in memory layout?



if no Polymorphism, how to program?

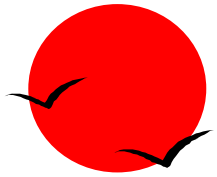
```
void CDocument::OnDraw(CDC* pDC)  {
   CShape  *pShape ;
    pShape = GetFirstShape( );
    while (pShape <> null)  {
            pShape->Display( pDC);
            pShape = GetNextShape( );
    }
}
```

Question: if no Polymorphism, how to get the desired display result by programming?



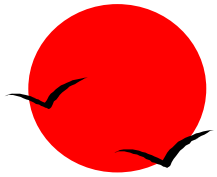
the solution scheme, when no Polymorphism

```
Class CShape  {
  public:
      enum Shape { LINE, POINT, CIRCLE, ARC, TEXT,
                   FIVECIRCLEFLAG};
      Shape m_nShape;

      void Function1(    )   { ……. };
      void Display(CDC* pDC)   { …… };
}
```



implementation code

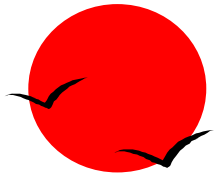
```
void CDocument::OnDraw(CDC* pDC)   {
    CShape  *pShape ;
   pShape = GetFirstShape(  );
    while (pShape <> null)  {
          switch (pShape->m_nShape)  {
              case LINE:
                  (CLine *)pShape->Display( pDC);
                  break;
              case POINT:
                  (CPoint *)pShape->Display( pDC);
                  break;
             case CIRCLE:
                  (CCircle *)pShape->Display( pDC);
              break;
              case FIVECIRCLEFLAG:
                  (CFiveCircleFlag *)pShape->Display( pDC);
              break;
             ……;
          }
            pShape = GetNextShape(  );
    }   }
```



Drawbacks of this code

When a new descendant class appear, it must be distributed to clients. Clients must **modify the above code** by adding 3 line, and then **re-compile** and **re-link** to include the new class. Finally, **distribute** the new application program to users.

That is to say, When a new descendant class appear, **it couple with the client source code together**, failing to separating it from the client source code. This **violate the modulization idea**;

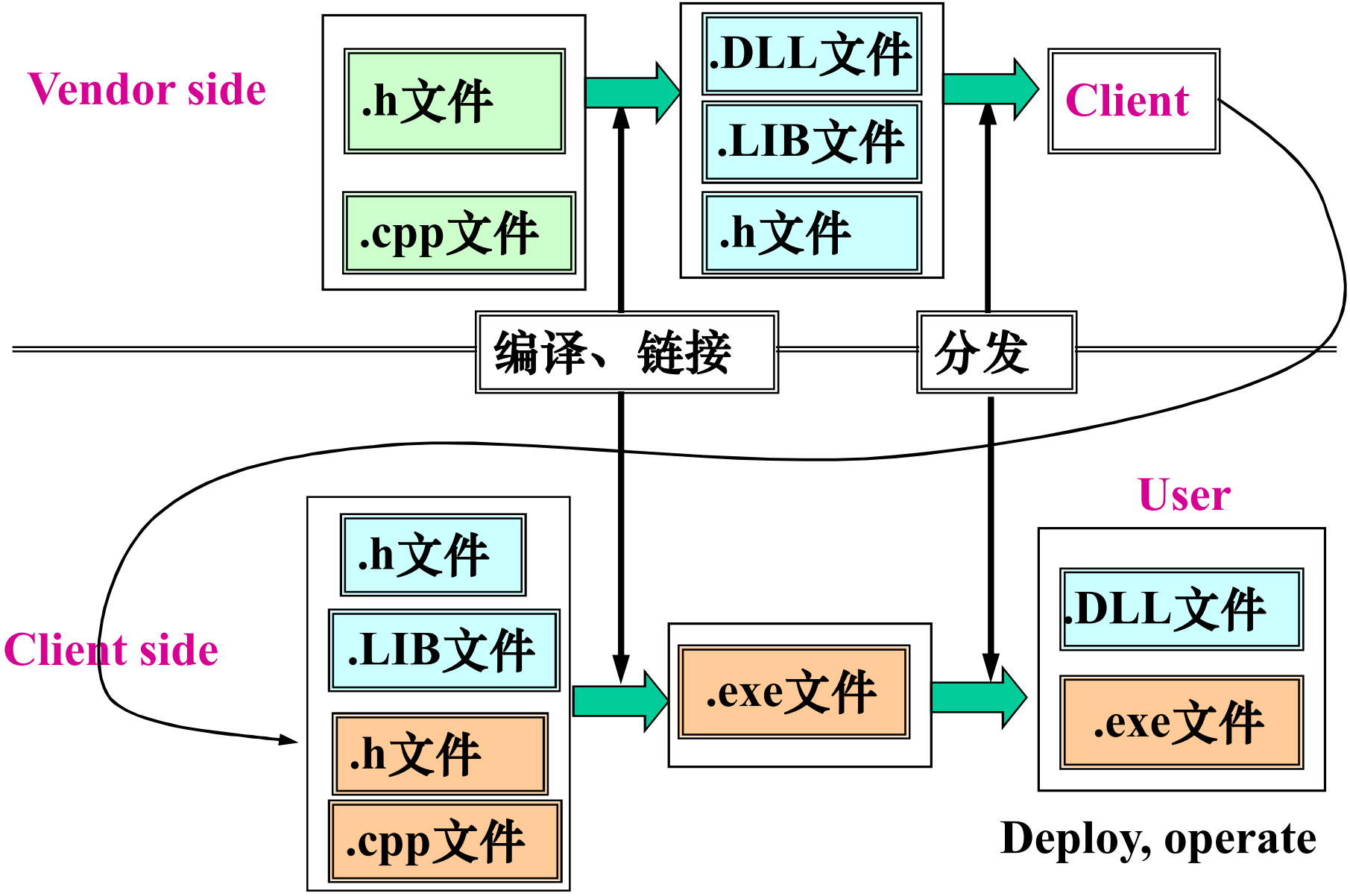


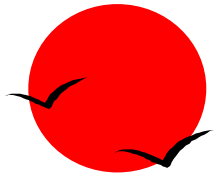
Polymorphism enables program active

- Without Polymorphism, the program is hard, not soft, not extensive, no **dynamic self-adaptability**.
- Polymorphism solve the above problems;
- Polymorphism enables program active;

This is why the name is named as *software*?

Interoperation in objected-oriented

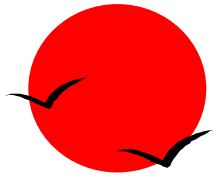




Interoperation Example

- **The vendor first defines a class in faststring.h file:**

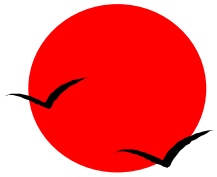
```
class __declspec(dllexport) FastString {
        Char *m_psz;
        public:
        FastString(char *psz);
        ~FastString(void);
        int Length(void); // returns # of characters
        int Find(char *psz); // returns offset
};
```



Interoperation example (cont.)

the vendor then implement the member functions in a separate source file faststring.cpp :

```
FastString::FastString(char *psz)
        : m_psz(new char[strlen(psz) + 1])
         {      strcpy(m_psz, psz);   }

FastString::~FastString(void)
 {       delete[ ] m_psz;   }

int FastString::Length(void)  {
         return strlen(m_psz);
}

int FastString::Find(char *psz)
{   // lookup code deleted for clarity    }
```

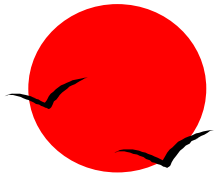


Interoperation example (cont.)

Compile and link the class to generate FastString.dll and FastString.lib that exposes the methods symbols.

The vendor packages 3 files to sell them to clients. The FastString.h is as follows:

```
class __declspec(dllimport) FastString    {
        char *m_psz;
        public:
        FastString(char *psz);
        ~FastString(void);
        int Length(void);    // returns #of characters
        int Find(const char *psz);  // returns offset
    };
```



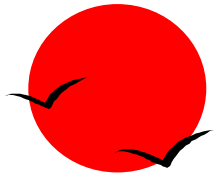
Client Program

Include “FastString.h”

```
FastString *fso = new Fastring(“abcdefghijk”);
int i = Fso->Length( );
int j = fso->Find(“cd”);
Delete fso;
```

then compile and link with Fastring.lib to generate a executable file App.exe;

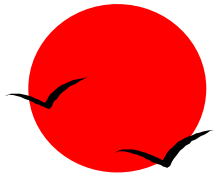
Deploy App.exe plus FastString.dll in User machine;



Problems in DLL Resue

- **Name resolution problems;**
- **Losing encapsulation;**
- **Coupling problems (Version issues);**
- **implememntation incompatibilities issues;**

  For eample, exception handling (we don't discuss it in this course);



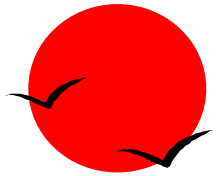
Name Resolution problems (linking problems)

The client may use different development Tool, for exampe, **GNU C++ , Borland C++, Microsoft Visual C++;**

To allow operator and function overloading (either with different argument types or in different scopes)**, C++ compilers typically use the naming methods of their own, resulting in name resolution problems**.

using extern "C" to disable symbolic mangling would not help in this case, as the DLL is exporting member functions, not global functions.

The vendor have to issues a DEF file for every kind of linker, to offer mapping between different imported symbols.



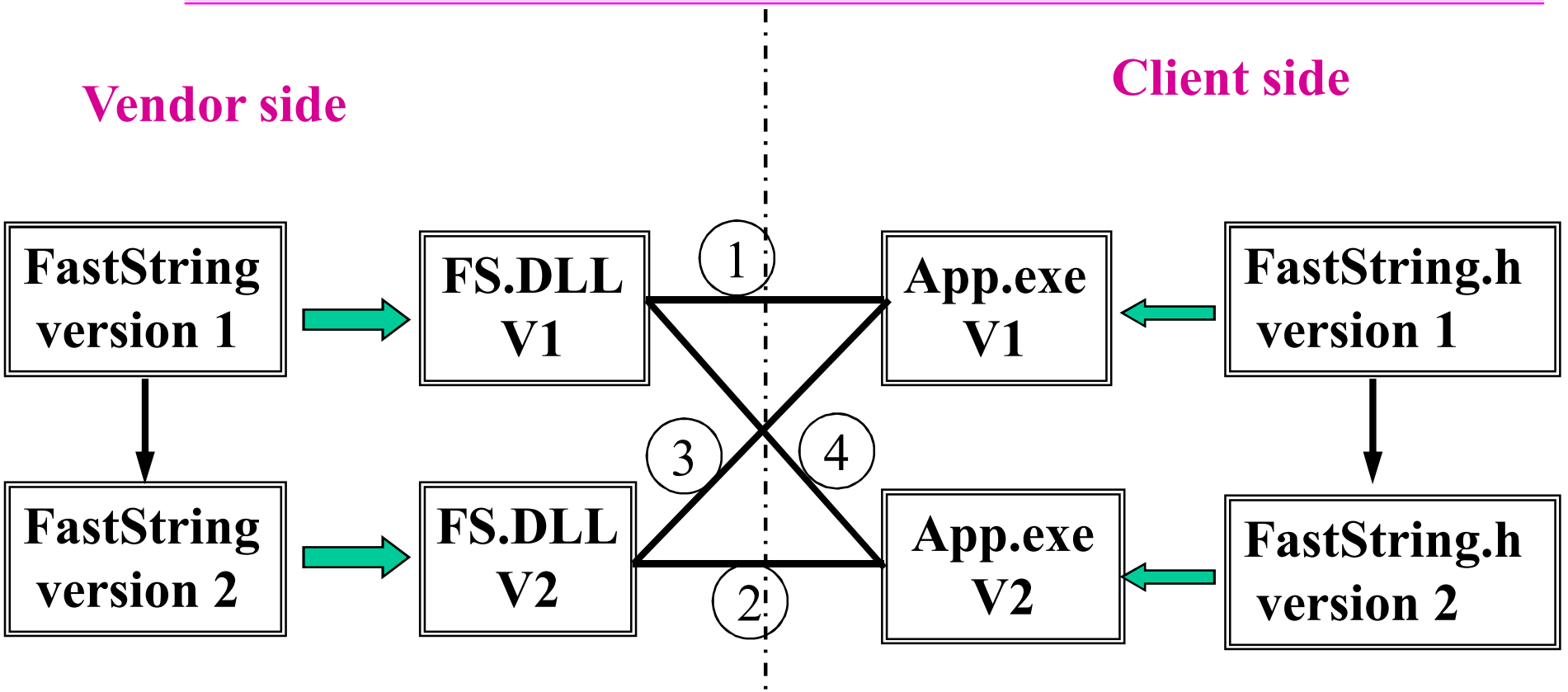
Losing encapsulation

```
FastString *fso = new Fastring("abcdefghijk");
```

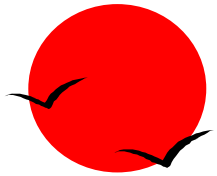
To support syntactic encapsulation via its private and protected keywords, no binary encapsulation.

the compilation model of C++ requires the client's compiler to have access to all information regarding object layout in order to instantiate an instance of a class or to make nonvirtual method calls and access to public member variables. This includes information about the size and order of the object's private and protected data members.

Version problems (Coupling problems)



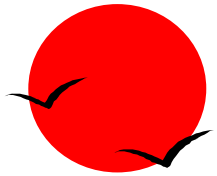
aa



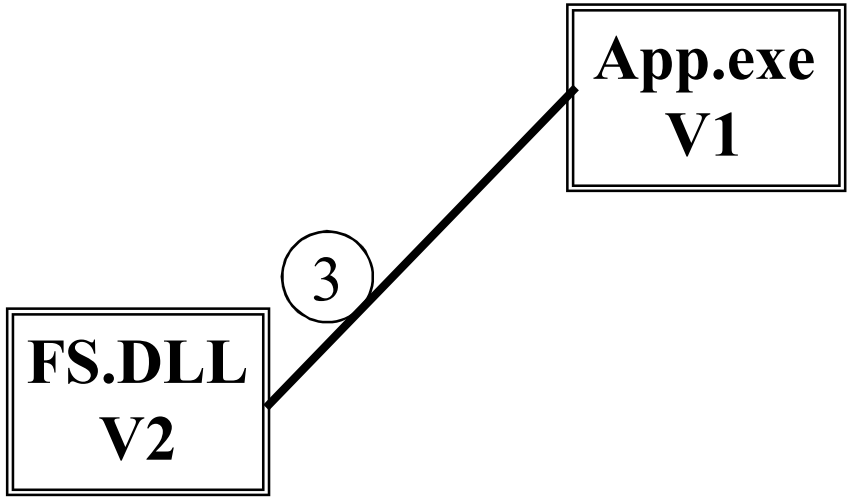
Coupling Issues

Version 2 of class FastString:

```
class __declspec(dllexport) FastString {
      const int m_cch;   // count of characters  //new member
         in V2
      char *m_psz;
      public:
      FastString(char *psz);
      ~FastString(void);
      int Length(void); // returns # of characters
      int Find(char *psz); // returns offset
};
```



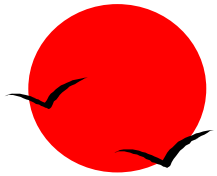
Coupling issues - version issues



Application allocates 4 bytes of memory to pass to the class's constructor. But FS.DLL's constructor, destructor assume that the client has allocated 8 bytes per instance .

One common solution to the versioning problem is to rename the DLL each time a new version is produced. This is the strategy taken by MFC.

Tremendous versions of a DLL in a machine.

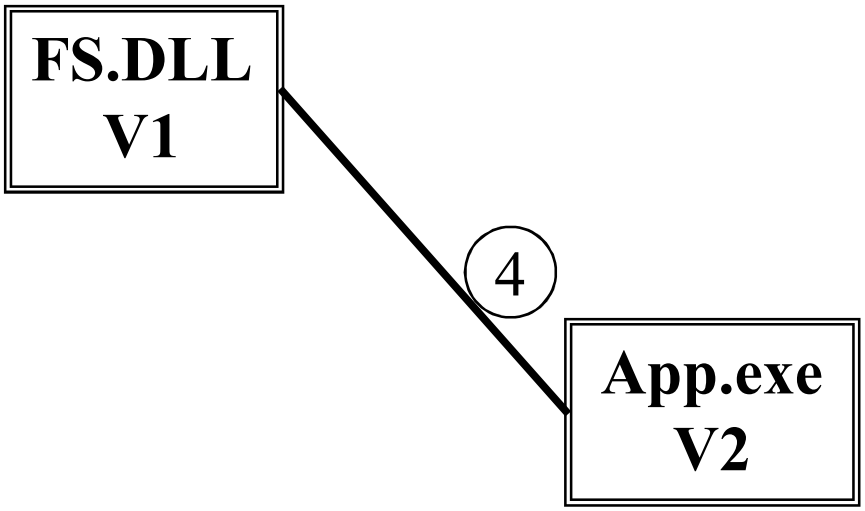


Version Coupling Issues

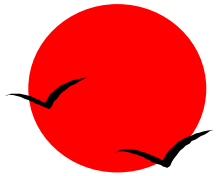
Version 2 of class FastString:

```
class __declspec(dllexport) FastString {
    const int m_cch;   // count of chars  //new member in V2
    char *m_psz;
    public:
    FastString(char *psz);
    ~FastString(void);
    int Length(void); // returns # of characters
    int Find(char *psz); // returns offset
};
```

Question



In the case, Is there problems?

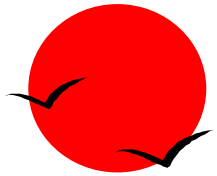


Explain to Coupling issue

- this versioning problem is rooted in the compilation model , which was not designed to support independent binary components.

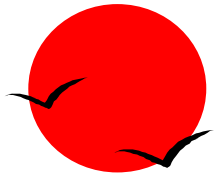
- Clients requires the knowledge of object layout, a tight binary coupling between the client and object executables is introduced .

- Normally, binary coupling is preferred, as it allows compilers to produce extremely efficient code. Unfortunately, this tight binary coupling prevents class implementations from being replaced without client recompilation.



Sub-conclusion

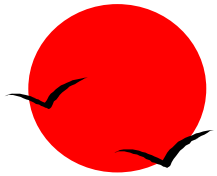
With the issues of coupling and the compiler and linker incompatibilities, simply exporting C++ class definitions from DLLs does not provide a reasonable binary component architecture.



Separating Interface from Implementation

- **The concept of encapsulation is based on separating what an object looks like (its interface) from how it actually works (its implementation).**

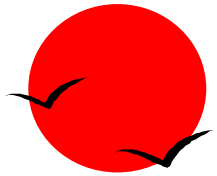
- **The problem with object-oriented programming is that this principle does not apply at a binary level, as a class is both interface and implementation simultaneously.**



interface

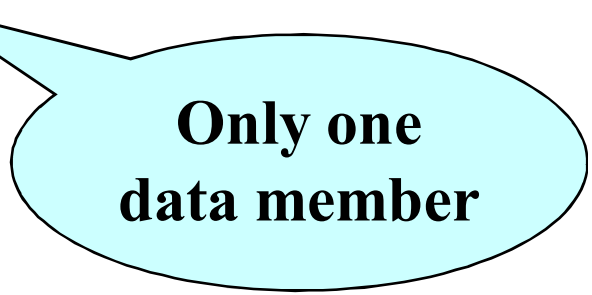
- the C++ interface class should not contain any of the data members that will be used in the implementation of the object.

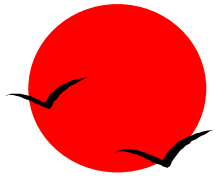
- The C++ implementation class will contain the actual data members required to implement the object's functionality.



The first way: proxy interface

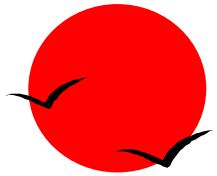
```
class __declspec(dllexport) FastStringItf {
    class FastString; // introduce name of implementation class
    FastString *m_pThis; // opaque pointer
    // (size remains constant)
    public:
    FastStringItf(char *psz);
    ~FastStringItf(void);
    int Length(void); // returns # of characters
    int Find(char *psz); // returns offset
};
```





Interface implementation

```
// faststringitf.cpp      // (part of DLL, not client) //
#include "faststringitf.h"
FastStringItf::FastStringItf(char *psz)
      : m_pThis(new FastString(psz)) {
      assert(m_pThis != 0);
}
int FastStringItf::LengthItf(void)  {
    return m_pThis->Length();
}
```

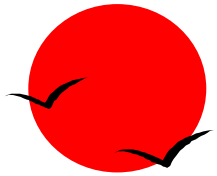


Interface distribution

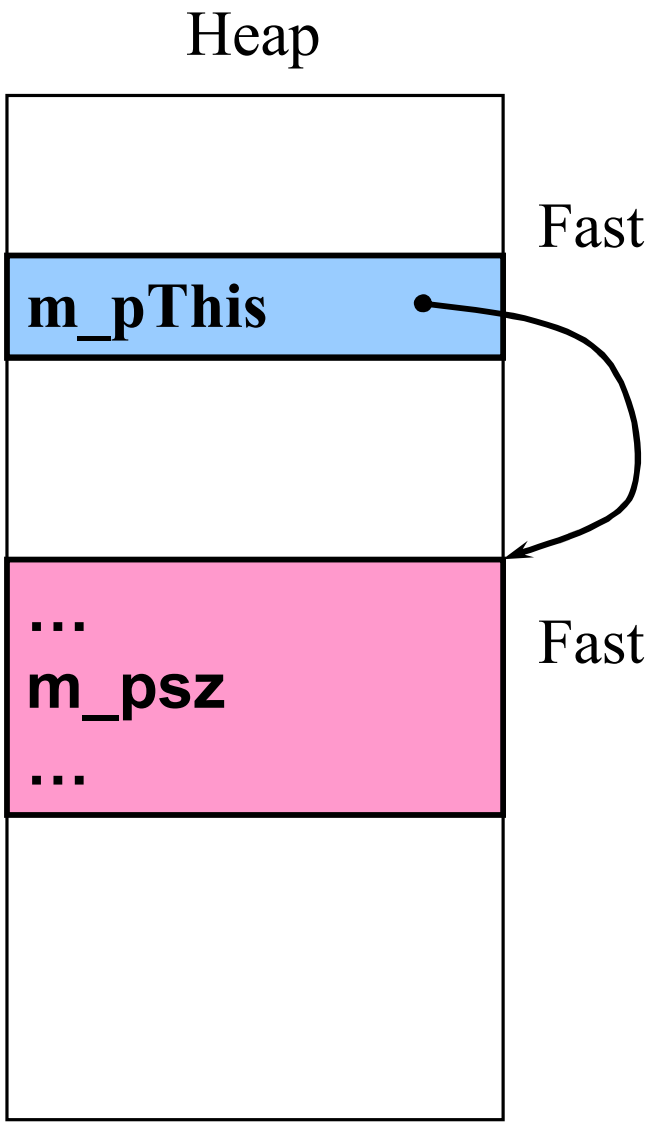
The vendor only distributes the faststringitf.h file, not FastString.h to clients, so clients never know the class definition of the implementation class FastString, thus solves both encapsulation problem and coupling problem, but not name resolution problems;

The code in DLL (faststringitf .cpp):

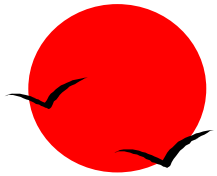
```
FastStringItf::FastStringItf(char *psz)
        : m_pThis(new FastString(psz)) {
        assert(m_pThis != 0);
}
```



Memory layout

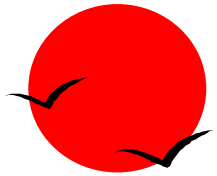



```
FastStringItf::FastStringItf(char *psz)
: m_pThis(new FastString(psz))
{assert(m_pThis != 0);}
```



overview

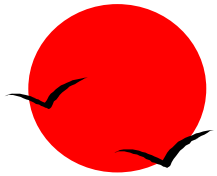
- To the object-oriented interoperation in the last lecture.



Interoperation Example:service side

- **faststring.h :**

```
class __declspec(dllexport) FastString {
        Char *m_psz;
        public:
        FastString(char *psz);
        ~FastString(void);
        int Length(void); // returns # of characters
        int Find(char *psz); // returns offset
};
```

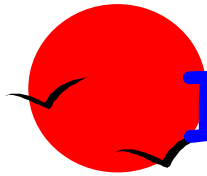


Interoperation Example:service side

faststring.cpp :

```
FastString::FastString(char *psz)
        : m_psz(new char[strlen(psz) + 1])
         {      strcpy(m_psz, psz);   }

FastString::~FastString(void)
 {       delete[ ] m_psz;   }

int FastString::Length(void)  {
          return strlen(m_psz);
}

int FastString::Find(char *psz)
{   // lookup code deleted for clarity    }
```

编译生成FastString.dll

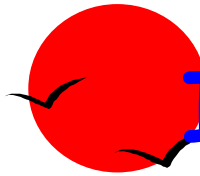


Interoperation example :client side

FastString.h :

```
class __declspec(dllimport) FastString    {
        char *m_psz;
        public:
        FastString(char *psz);
        ~FastString(void);
        int Length(void);   // returns #of characters
        int Find(const char *psz);  // returns offset
    };
```

FastString.dll

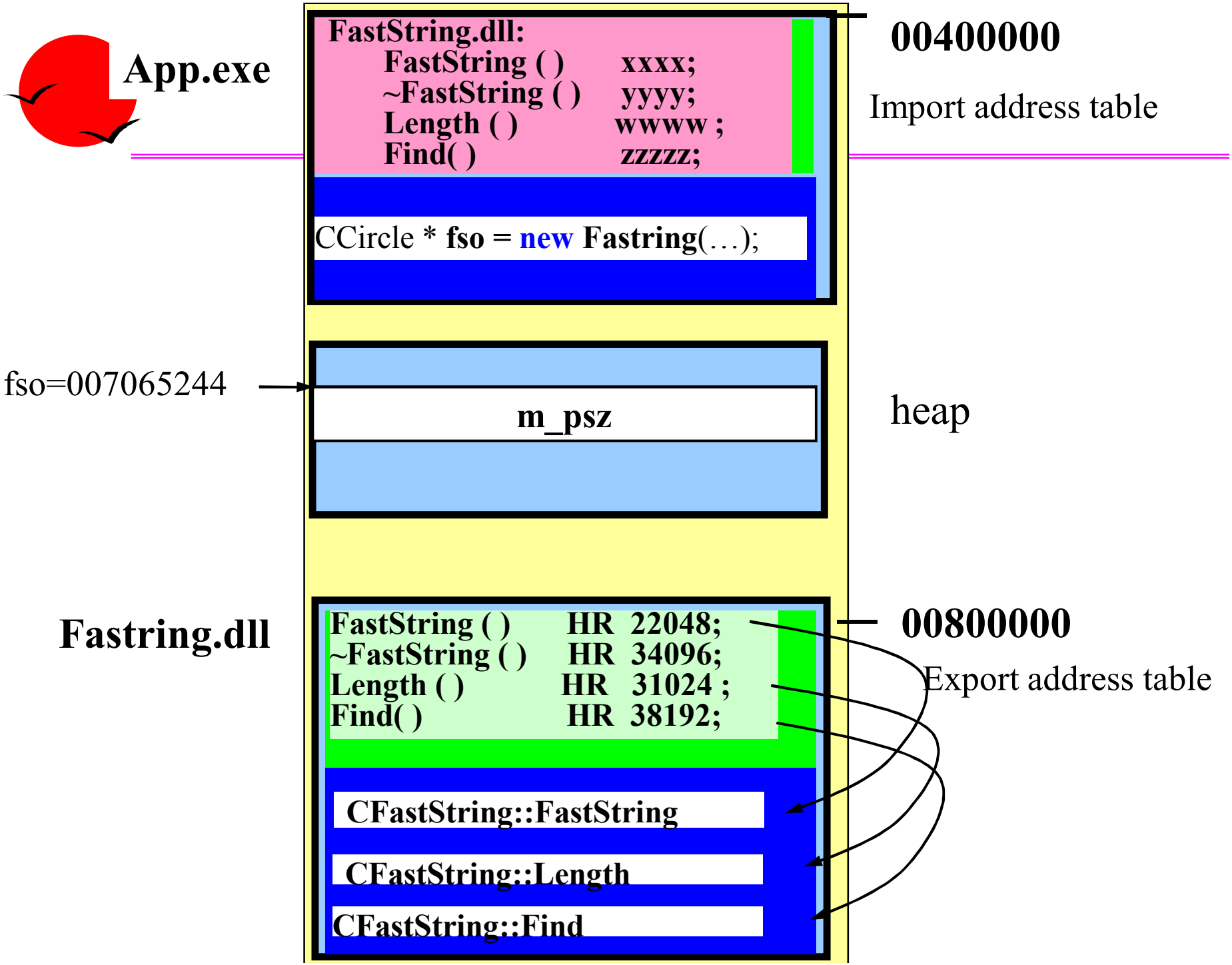


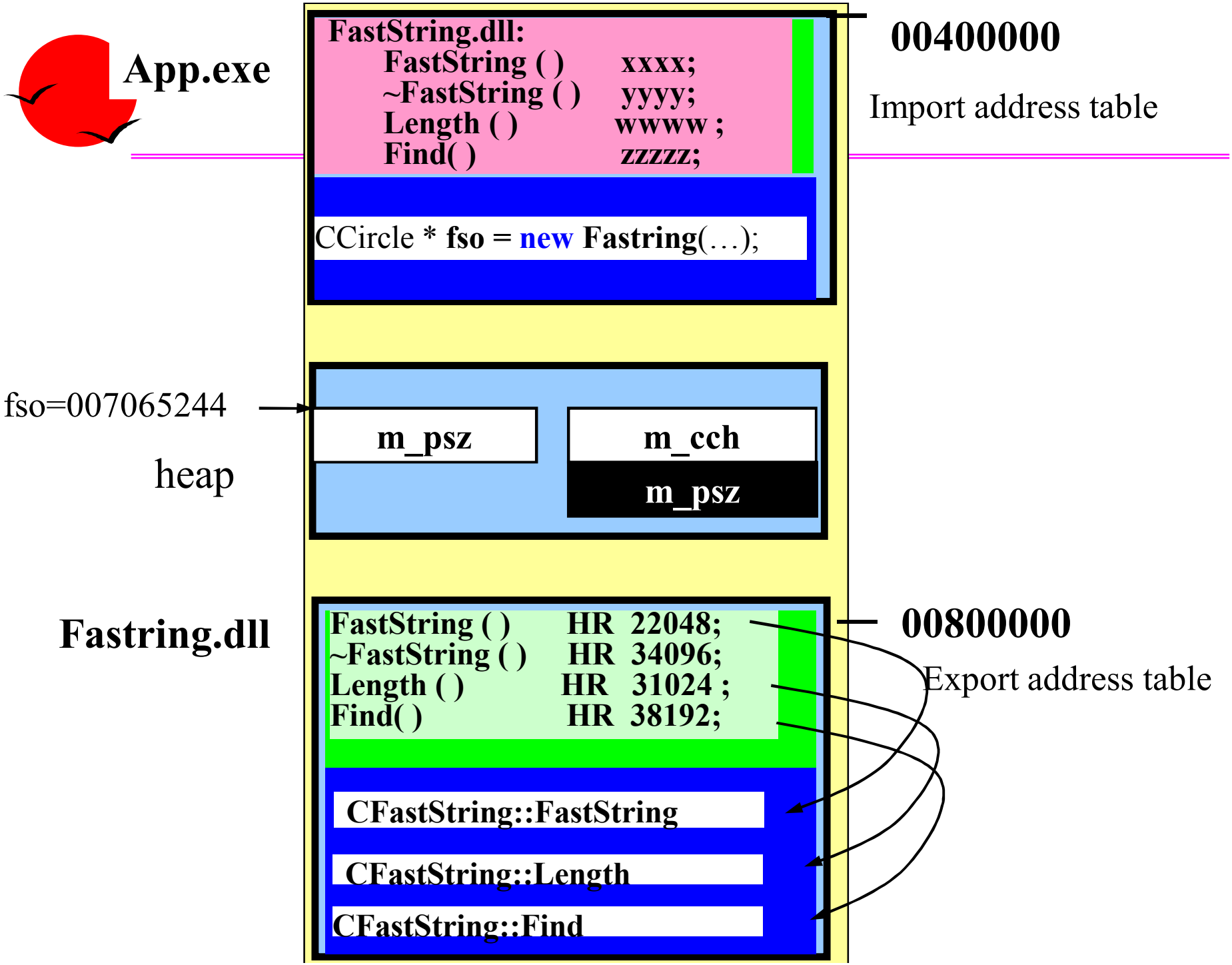
Interoperation example :client side

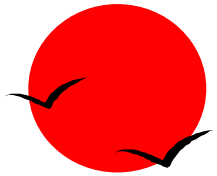
App.cpp:

```
Include “FastString.h”

FastString *fso = new Fastring(“abcdefghijk”);
int i = Fso->Length( );
int j = fso->Find(“cd”);
Delete fso;
```

App.exe
FastString.dll;


App.exe
FastString.dll:
FastString ( ) xxxx;
~FastString ( ) yyyy;
Length ( ) wwww ;
Find( ) zzzzz;
00400000
Import address table
CCircle * fso = new Fastring(…);
fso=007065244
m_psz
heap
Fastring.dll
FastString ( ) HR 22048;
~FastString ( ) HR 34096;
Length ( ) HR 31024 ;
Find( ) HR 38192;
00800000
Export address table
CFastString::FastString
CFastString::Length
CFastString::Find

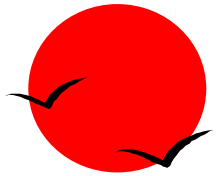

App.exe
FastString.dll:
FastString ( ) xxxx;
~FastString ( ) yyyy;
Length ( ) wwww ;
Find( ) zzzzz;
00400000
Import address table
CCircle * fso = new Fastring(…);
fso=007065244
heap
m_psz
m_cch
m_psz
Fastring.dll
FastString ( ) HR 22048;
~FastString ( ) HR 34096;
Length ( ) HR 31024 ;
Find( ) HR 38192;
00800000
Export address table
CFastString::FastString
CFastString::Length
CFastString::Find



proxy approach: service side

- **FastStringItf.h:**
- **class __declspec(dllexport)** FastStringItf {

```
        class FastString;
        FastString *m_pThis;
        public:
        FastStringItf(char *psz);
        ~FastStringItf(void);
        int Length(void);
        int Find(char *psz);
};
class FastString {
        const int m_cch;
        char *m_psz;
        public:
        FastString(char *psz);
        ~FastString(void);
        int Length(void);
        int Find(char *psz);
};
```

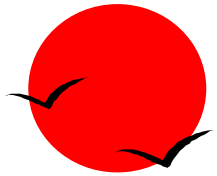


proxy approach: service side

faststringitf.cpp :

```
#include "faststringitf.h"
FastStringItf::FastStringItf(char *psz)
  : m_pThis(new FastString(psz)) {
  assert(m_pThis != 0);
}
int FastStringItf::LengthItf(void)  {
    return m_pThis->Length();
}
FastString::FastString(char *psz)
    : m_psz(new char[strlen(psz) + 1])  {
    strcpy(m_psz, psz);
  }
int FastString::Length(void)  {
        return strlen(m_psz);
}
```

编译生成FastStringItf.dll

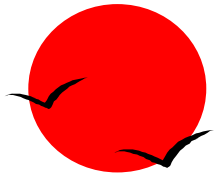


In client side

- **FastStringItf.h:**
- **class __declspec(dllimport)** FastStringItf {

```
    class FastString;
    FastString *m_pThis;
    public:
    FastStringItf(char *psz);
    ~FastStringItf(void);
    int Length(void);
    int Find(char *psz);
};
```

FastStringItf.dll

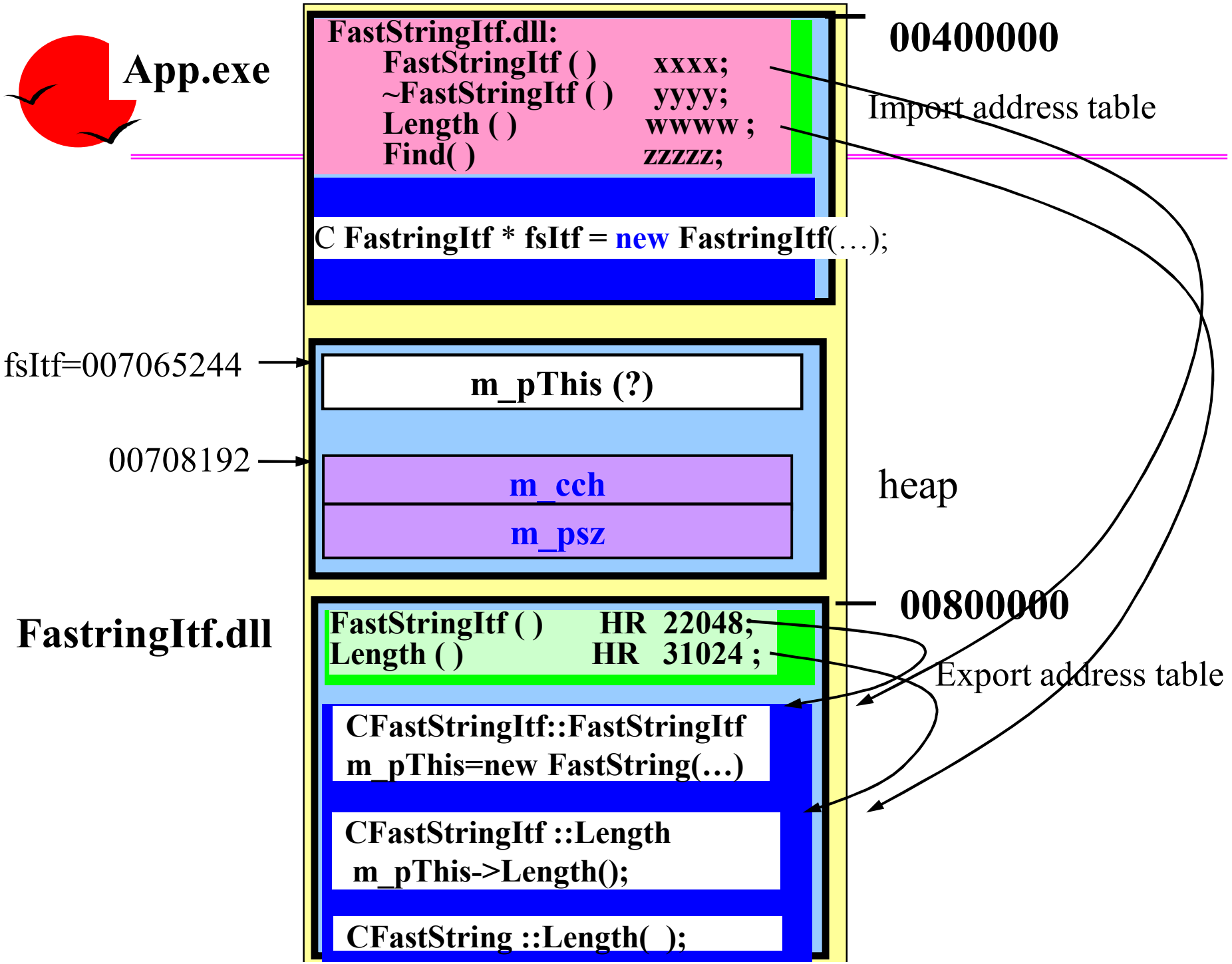


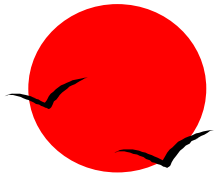
In client side

App.cpp:

```
Include “FastStringItf.h”

FastStringItf *fsitf = new FastStringItf(“abcdefghijk”);
int i = fsitf ->Length( );
int j = fsitf ->Find(“cd”);
Delete fsitf;
```

FastStringItf.dll


App.exe
FastStringItf.dll:
FastStringItf ( ) xxxx;
~FastStringItf ( ) yyyy;
Length ( ) wwww ;
Find( ) zzzzz;
00400000
Import address table
C FastringItf * fsItf = new FastringItf(…);
fsItf=007065244
m_pThis (?)
00708192
m_cch
m_psz
heap
FastringItf.dll
FastStringItf ( ) HR 22048;
Length ( ) HR 31024 ;
00800000
Export address table
CFastStringItf::FastStringItf
m_pThis=new FastString(…)
CFastStringItf ::Length
m_pThis->Length();
CFastString ::Length( );



Reviews to this approach

Although the approach of using handle classes can safely expose a class from a DLL, thus solves two problems:

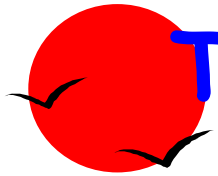
[1] encapsulation problem;

[2] coupling problem;

However, it also has its weaknesses:

[1] It doesn't't solve the name resolution problems;

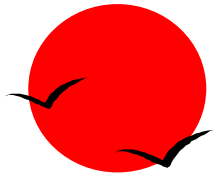
[2] increase performance cost .



The second way: abstract interface

Idea: Separating interface from implementation:

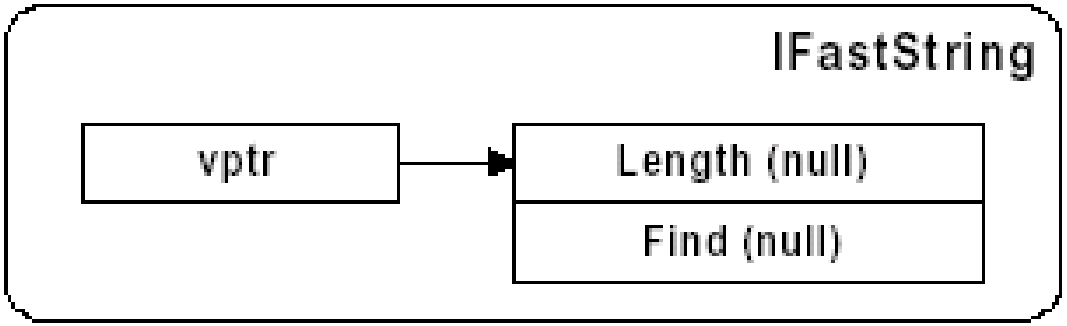
To devise a **technique** for **hiding** compiler/ linker implementation details **behind** some sort of binary interface, this would make a C++-based DLL usable by a much larger clients.

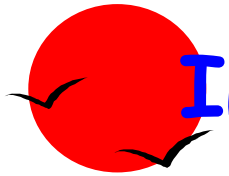


Pure virtual functions

•The purpose of **pure virtual** is to **informs the compiler that no implementation of these methods is required from the interface class**. When the compiler generates the **vtbl** for the interface class, the entry for each pure virtual function will either be null.

If a method is not declared as pure virtual, the compiler would have attempted to populate the corresponding **vtbl** entry with the interface class's method implementation, which of course does not exist. This would result in **a link error**.



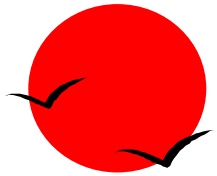


Interface class Vs implementation class

- The **interface class** is an abstract class.

- The **implementation class** must derive from **the interface class** and **override each of the pure virtual functions** with meaningful implementations.

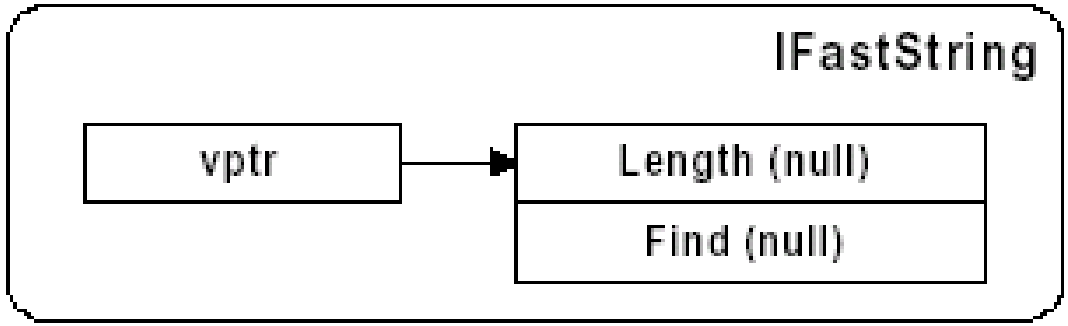
- This inheritance relationship will result in objects that have an **object layout** that is a **binary superset** of **the layout of the interface class** (which ultimately is just a **vptr/vtbl**).



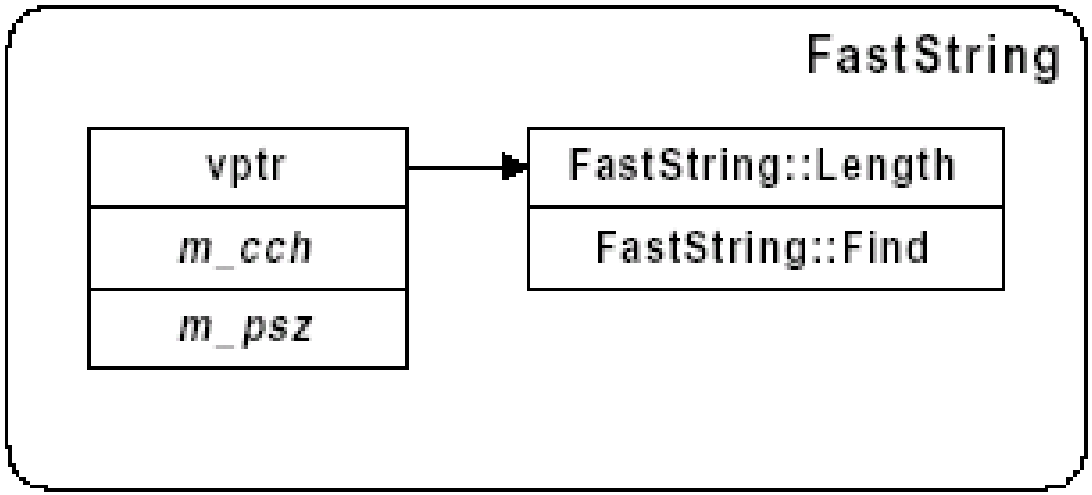
Abstract class Demeostration

```
// ifaststring.h :
class IFastString  {
        public:
        virtual int Length(void) = 0;
        virtual int Find(const char *psz)  = 0;
};
class FastString : public IFastString {
        int m_cch; // count of characters
        char *m_psz;
        public:
        FastString(char *psz);
        ~FastString(void);
        int Length(void); // returns # of characters
        int Find(char *psz); // returns offset
};
```

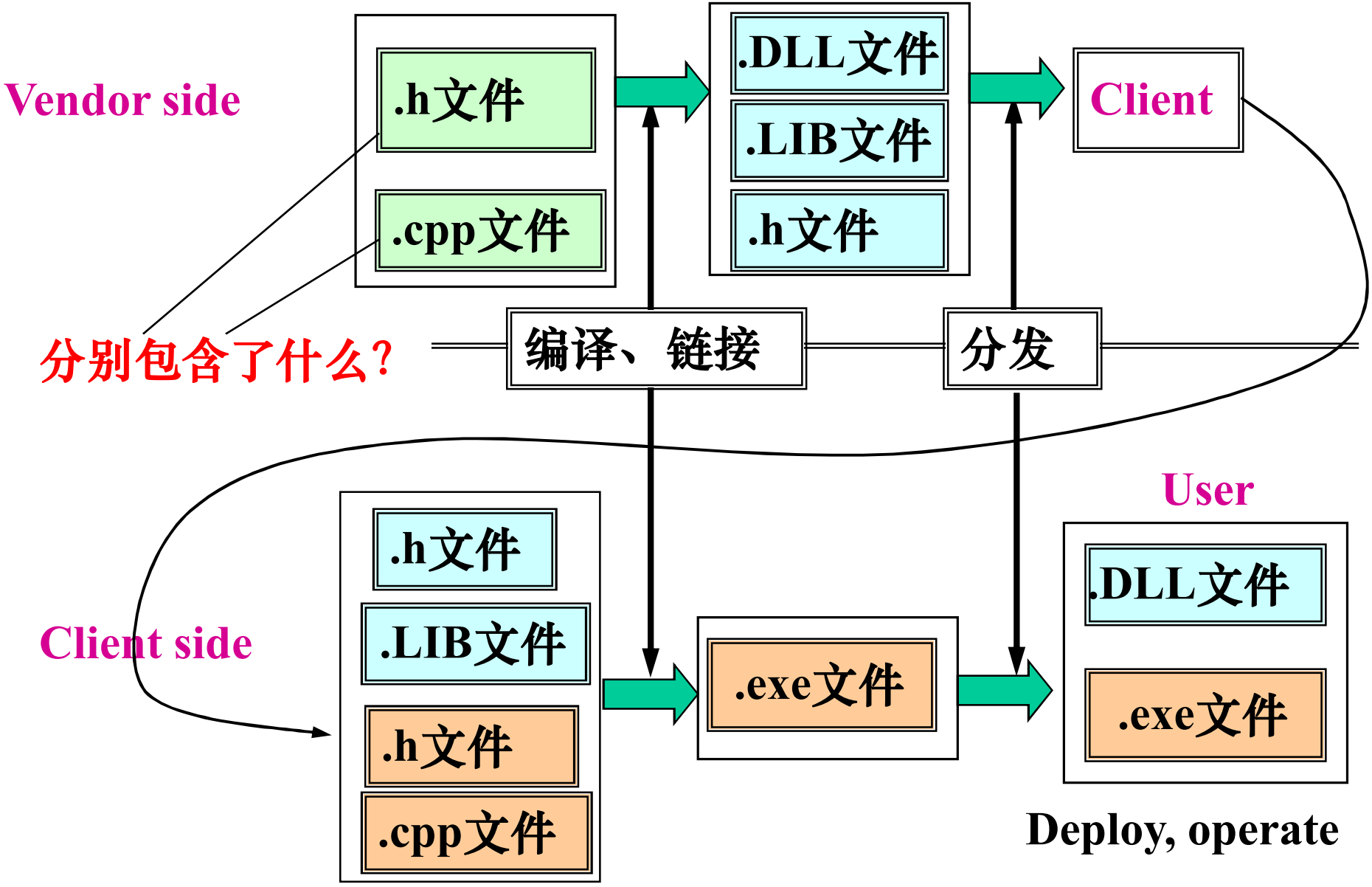
Interface/Implementation Classes in Binary



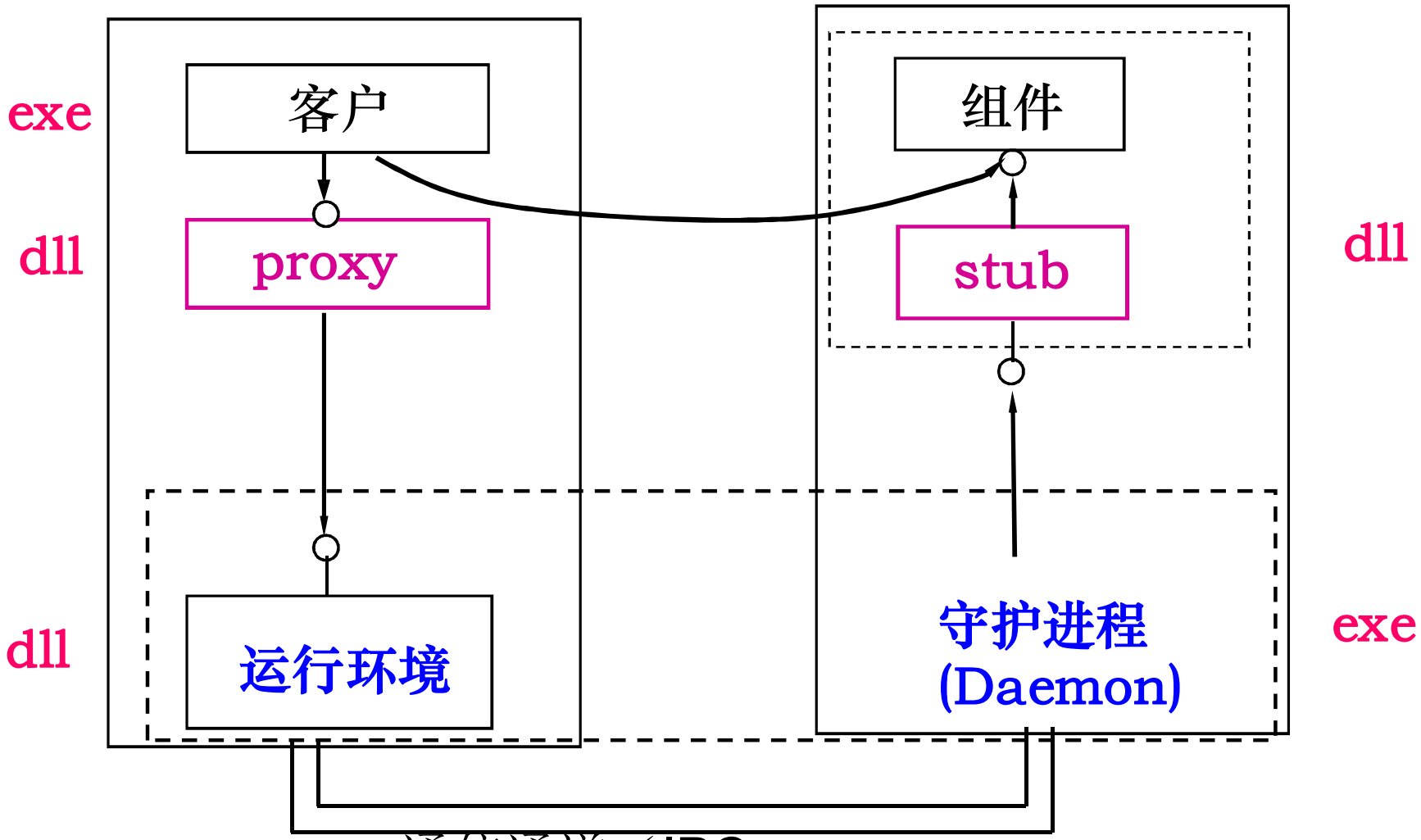
偷梁换柱之计!

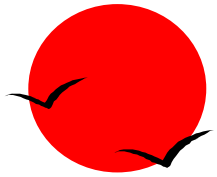


Interoperation in objected-oriented



Interoperation in objected-oriented





面向对象中互操作的问题

Version 2 of class FastString:

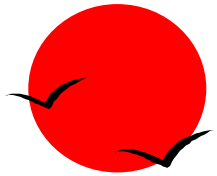
```
class __declspec(dllexport) FastString {
    int m_cch;   // count of characters  //new member in V2
    char *m_psz;
    public:
    FastString(char *psz);
    ~FastString(void);
    int Length(void); // returns # of characters
    int Find(char *psz); // returns offset
};
```

Problem 1：version issues

新版本FS.DLL的 构造、析构 函数假定客户分配了8字节的内存；

由于App.exe用到的是老版本的class FastString 定义，因此 new Fastring( ) 时，对象只分配4字节的内存；

问题发生在什么时候?



problem 2：Name resolution

```
int FastString::Length()
 { }
int FastString::Find(char *psz)
{  }
```

FS.dll

Export address table

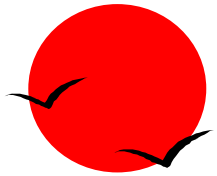
```
FastString *fso = new
     Fastring("abcdefghijk");
int i = Fso->Length( );
int j = fso->Find("cd");
Delete fso;
```

App.exe

Import address table

如何定位外部函数?

问题发生在什么时候？链接时?



problem 3： Losing encapsulation

```
int FastString::Length()
 { }
int FastString::Find(char *psz)
{  }
```

FS.dll

```
FastString *fso = new
     Fastring("abcdefghijk");
int i = Fso->Length( );
int j = fso->Find("cd");
Delete fso;
```

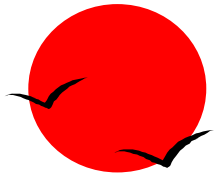
App.exe

Export address table

Import address table

对数据成员的访问：
[this + offset];

什么性质的问题？



The first way: proxy

```
•class __declspec(dllexport)  FastStringItf  {
        class FastString;
        FastString *m_pThis; // opaque pointer
        int Length(void); // returns # of characters
        int Find(char *psz); // returns offset
};
/ / faststringitf.cpp      // (part of DLL, not client) //
FastStringItf::FastStringItf(char *psz)
        : m_pThis(new FastString(psz)) {
        assert(m_pThis != 0);
}
int FastStringItf::LengthItf(void) {
    return m_pThis->Length();
}
```

还有：//faststring.cpp　　// **(part of DLL, not client)**

?

保证正确互操作，有什么要求?

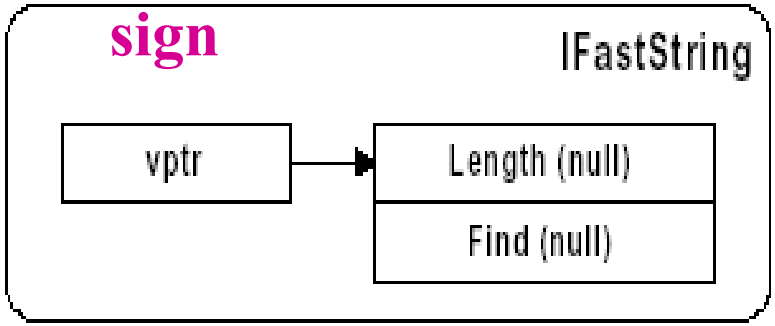
解决了什么问题?

没有解决什么问题?

The second way: Separating Interface from Implementation

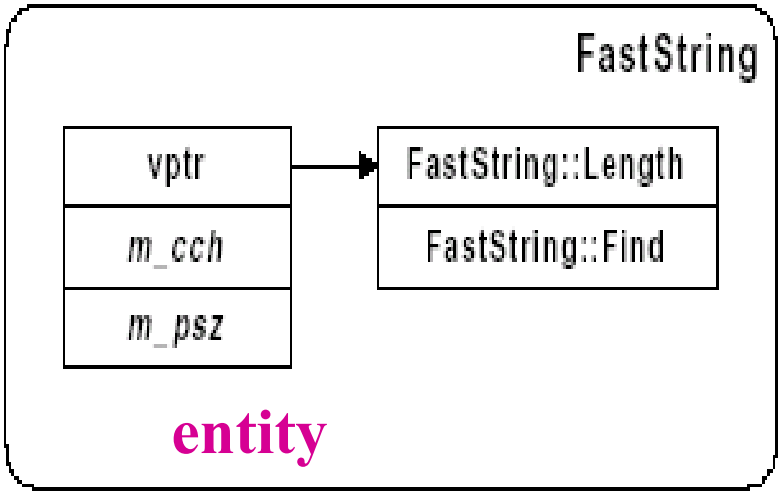
```
// ifaststring.h :
class IFastString  {
        public:
        virtual int Length(void) = 0;
        virtual int Find(const char *psz) = 0;
};
class FastString : public IFastString {
        int m_cch; // count of characters
        char *m_psz;
        public:
        FastString(char *psz);
        ~FastString(void);
        virtual int Length(void);
        virtual int Find(char *psz);
        };
```

Interface class



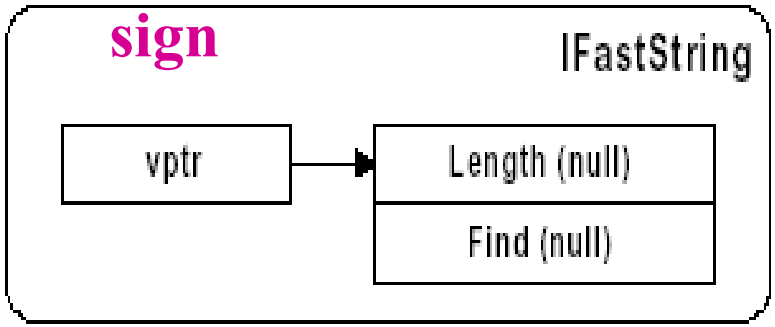
A kind of reflection

Implementation class



The second way: Implementation technology

```
// ifaststring.h :
class IFastString  {
   public:
   virtual int Length(void) = 0;
   virtual int Find(const char *psz) = 0;
};
```

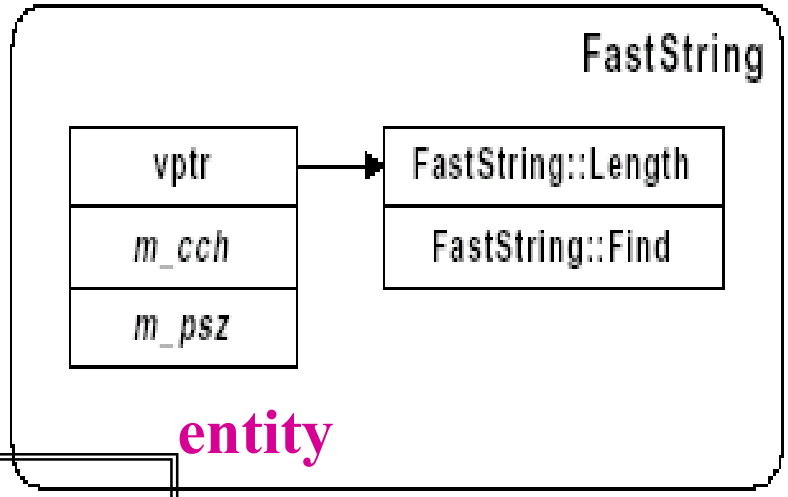


A kind of reflection

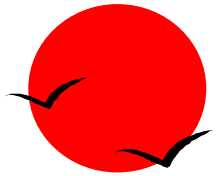
Client calls the methods of the interface:

int i = pFSO->Length( );
int j = pFSO ->Find("cd");

Assembly generated by compiler:
MOV CDX, DWORD PTR [this];
CALL DWORD PTR [PTR [CDX] +4]



Question: 1) which function is called?
2) Does class IfastString need to be exported?



Issues 1: Object Construction

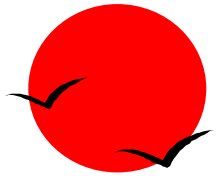
```
// ifaststring.h :
class IFastString  {
        public:
        virtual int Length(void) = 0;
        virtual int Find(char *psz) = 0;
};
class FastString : public IFastString {
        int m_cch; // count of characters
        char *m_psz;
        public:
        FastString(char *psz);
        ~FastString(void);
        virtual int Length(void);
        virtual int Find(char *psz);
        };
```

Interface class, as contract;

Implementation class

How do clients get the pointer to the interface?

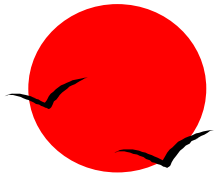
Even if there is no implementation class, client can also program.



Issues 1: Object Construction

Revealing the implementation class definition to the client would bypass the **binary encapsulation** of the interface, which would **defeat the purpose of using an interface class**.

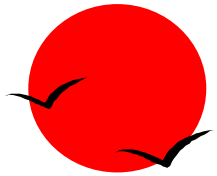
One reasonable technique for allowing clients to instantiate FastString objects would be to export a global function from the DLL that would call the new operator **on behalf of the client**. Provided that this routine is exported using **extern "C"**, it would still be accessible from any C++ compiler.



Object Construction

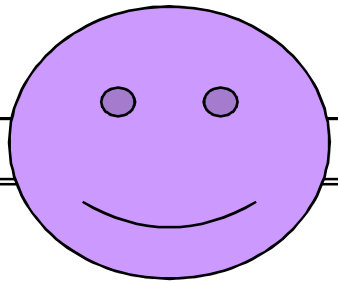
```
class IFastString   {
        public:
        virtual int Length(void) const = 0;
        virtual int Find(const char *psz) const = 0;
};

extern "C” IFastString *CreateFastString(const char *psz);
// faststring.cpp (part of DLL) //
IFastString *CreateFastString (const char *psz) {
            return new FastString(psz);
}
```

Question:
Whether can this strategy be used in original scheme or not? 从耦合问题来看，我们在版本升级时，要注意什么？



fastString.h to clients

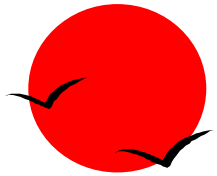
```
class IFastString   {
public:
      virtual int Length(void) const = 0;
      virtual int Find(const char *psz) const = 0;
};

extern "C" __declspec(dllimport) IFastString *
CreateFastString(const char *psz);
```



Question:

1) If need class IFastString _declspec(dllimport) or not?

2) Which are The other 2 files to clients?



Issue 2: Object destruction

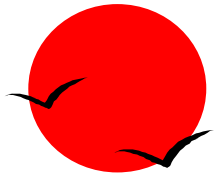
The following client code will compile, but the results are unexpected:

```
int f(void)   {
      IFastString *pfs = CreateFastString("Deface me");
      int n = pfs->Find("ace me");
      delete pfs; //delete  (FastString *)pfs;    but don't give its definition
      return n;
 }
```

leakhole! !

Reasons: **destructor for the interface class is not virtual.** This means that the call to the delete operator will not dynamically find the most derived destructor and recursively destroy the object from the outermost type to the base type.

Therefore, the **FastString destructor** is never called, leading to **buffer memory leaks** used to hold the string "Deface me" and the object.

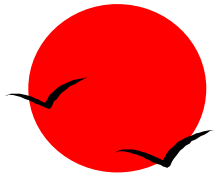


Solution to Object destruction

•**One obvious solution** to this problem is to **make the destructor virtual in the interface class**.

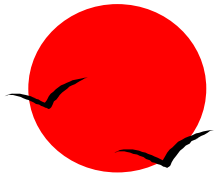
•Unfortunately, this pollutes the compiler independence of the interface class, as **the position of the virtual destructor in the vtbl can vary from compiler to compiler**.

•**Another workable solution** to this problem is to add an explicit Delete method to the interface as another pure virtual function and have the derived class delete itself in its implementation of this method. This results in the correct destructor being executed.



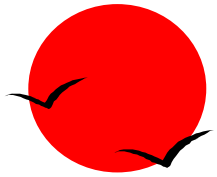
Solution to Object destruction

```
class IFastString {
        public:
        virtual void Delete(void) = 0;
        virtual int Length(void) const = 0;
        virtual int Find(const char *psz) const = 0;
};
extern "C“ IFastString *CreateFastString (const char *psz);
class FastString : public IFastString   {
        int m_cch; // count of characters
        char *m_psz;
        public:
        FastString(char *psz);
        ~FastString(void);
        void Delete(void); // deletes this instance
        virtual int Length(void); // returns # of characters
        virtual int Find(char *psz); // returns offset
};
```



Solution to Object destruction

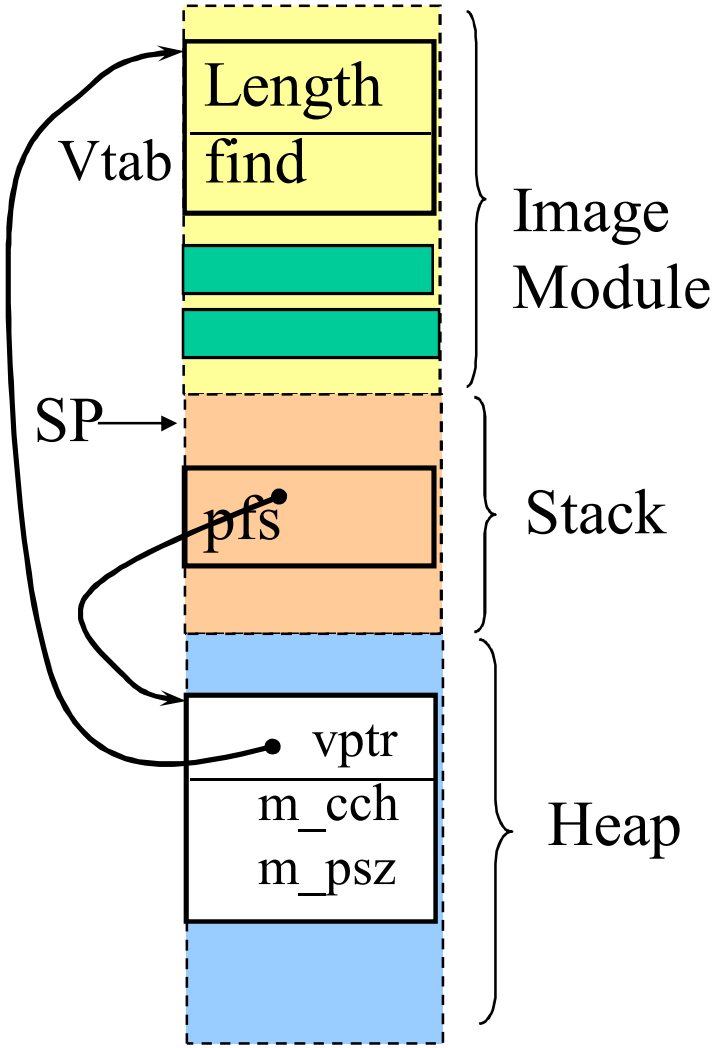
```
//vendor code of implementation of Delete:
void FastString::Delete(void) {
        delete this;
}

//Client code:
int f(void) {
    IFastString *pfs = CreateFastString("Hi Bob!");
    if (pfs)
        n = pfs->Find("ob");
        pfs->Delete();
}
```



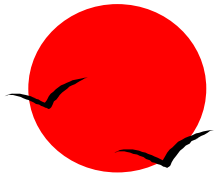
Solution to Object destruction

```
void FastString::Delete(void) {
        delete this;
}
int f(void) {
      IFastString *pfs =
         CreateFastString("Hi Bob!");
      if (pfs)
         n = pfs->Find("ob");
         pfs->Delete();
}
delete pfs;
```

MOV CDX, DWORD PTR [pfs +0];
CALL DWORD PTR [CDX +4];



Vptr由谁填写?



(2) Interface approach: service side

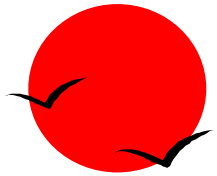
IFastString.h:

```
class IFastString {
        public:
        virtual void Delete(void) = 0;
        virtual int Length(void) const = 0;
        virtual int Find(char *psz) const = 0;
};
```

Java语言：

```
interface FastString   {
        void Delete( );
        int Length( );
        int Find(char *psz);
};
```

```
extern "C“__declspec(dllexport) IFastString *CreateFastString (const char *psz);

class FastString : public IFastString   {
        int m_cch; // count of characters
        char *m_psz;
        public:
        FastString(char *psz);
        ~FastString(void);
        virtual void Delete(void);
        virtual int Length(void);
        virtual int Find(char *psz);
        };
```

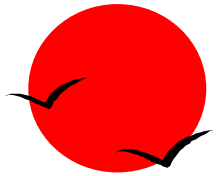


(2) Interface approach: service side

IFastString.cpp:

```
extern "C"__declspec(dllexport) IFastString *CreateFastString (const char *psz) {
        return new FastString(psz);
}
FastString::FastString(char *psz)
    : m_psz(new char[strlen(psz) + 1])
     {    strcpy(m_psz, psz);   }

FastString::~FastString(void)
 {          delete[ ] m_psz;   }

virtual void FastString::Delete(void) {
        delete this;
}
virtual int FastString::Length(void)  {
        return strlen(m_psz);
}
virtual int FastString::Find(char *psz)
{   // lookup code deleted for clarity    }
```

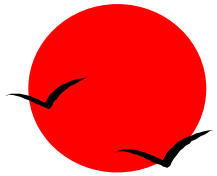
编译和链接生成IFastString.dll



(2) Interface approach: client side

IFastString.h:

```
class IFastString {
      public:
      virtual void Delete(void) = 0;
      virtual int Length(void) const = 0;
      virtual int Find(const char *psz) const = 0;
};

extern "C“__declspec(dllimport) IFastString *CreateFastString
(const char *psz);
```

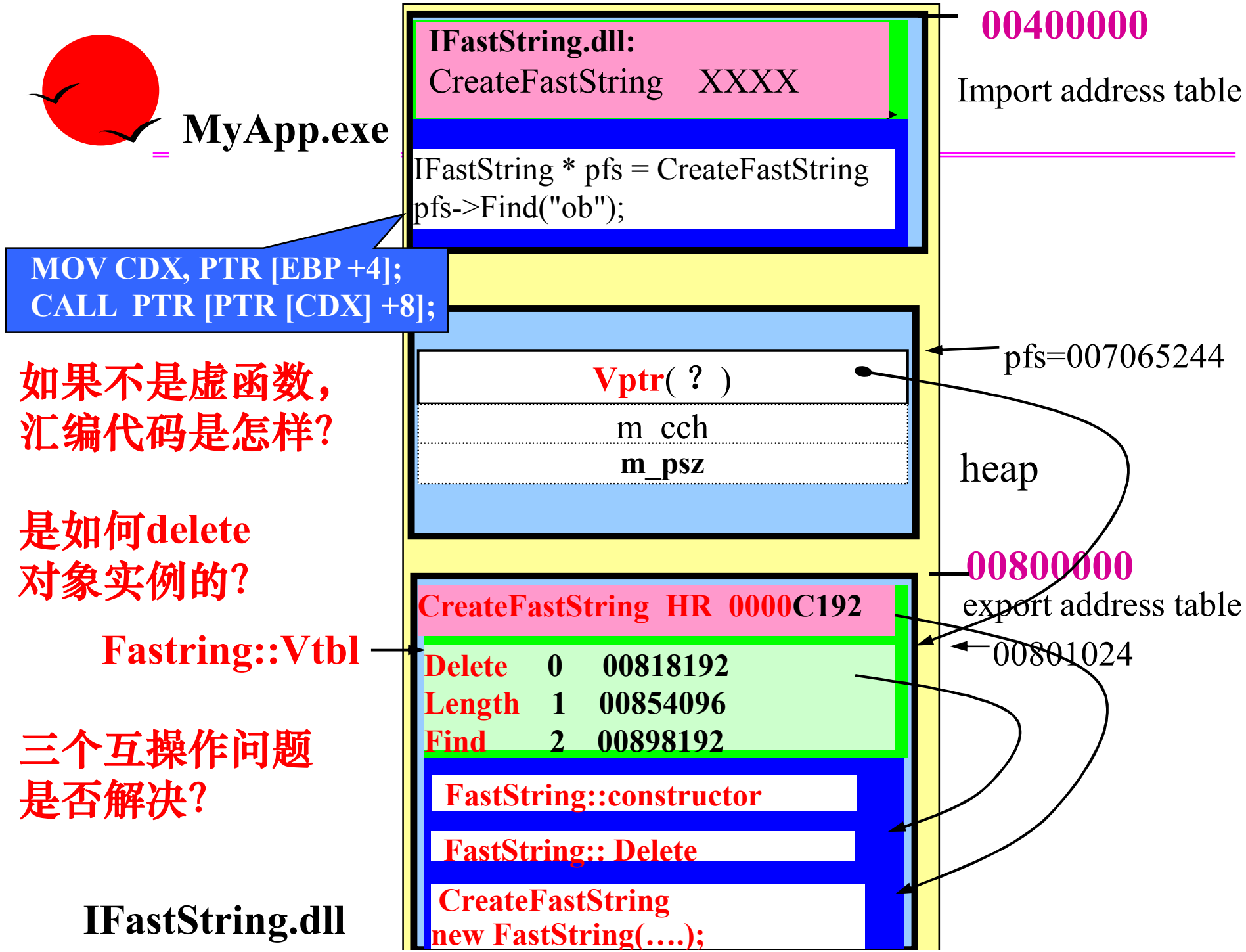


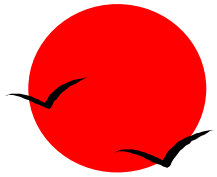
(2) Interface approach: client side

MyApp.cpp:

```
#include “IFastString.h”

int f(void) {
        IFastString *pfs = CreateFastString("Hi Bob!");
        if (pfs)
                int n = pfs->Find("ob");
                int k = pfs->Length( );
                pfs->Delete();
}
```

编译和链接生成MyApp.exe

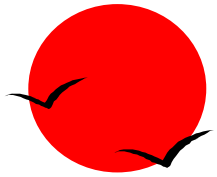

MyApp.exe
IFastString.dll:
CreateFastString XXXX
00400000
Import address table
IFastString * pfs = CreateFastString
pfs->Find("ob");
MOV CDX, PTR [EBP +4];
CALL PTR [PTR [CDX] +8];
如果不是虚函数，
汇编代码是怎样?
Vptr(?)
m_cch
m_psz
pfs=007065244
heap
是如何delete
对象实例的?
00800000
export address table
00801024
Fastring::Vtbl
CreateFastString HR 0000C192
Delete 0 00818192
Length 1 00854096
Find 2 00898192
FastString::constructor
FastString:: Delete
CreateFastString
new FastString(….);
三个互操作问题
是否解决?
IFastString.dll



Observations - the name resolution problem is solved

interface methods are **immune to name resolution differences between compilers**.

- The only entry point that is explicitly linked against by name is **CreateFastString, the global function exported using extern "C"**, which **suppresses symbolic mangling**. This implies that all C++ compilers expect the import library and DLL to export the same symbol.

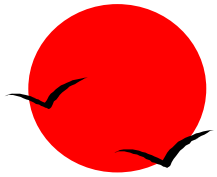


Issue 3: Runtime Polymorphism

The vendor can only distribute 2 files: .H file and .DLL file.

Deploying class implementations **using abstract base classes as interfaces** opens up a new world of possibilities in terms of what can happen at runtime.

Note that the FastString **DLL exports only one symbol**, CreateFastString. This makes it fairly trivial for the client to load the DLL dynamically on demand using **LoadLibrary** and resolve that one entry point using **GetProcAddress**:

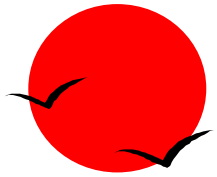


Runtime Polymorphism

```
IFastString *CallCreateFastString(const char *psz)   {
      static IFastString * (*pfn)(const char *) = 0;
      if (!pfn) {
      HINSTANCE h = LoadLibrary("FastString.DLL");
      if (h)
          *(FARPROC *  )&pfn = GetProcAddress(h, "CreateFastString");
      return pfn ? pfn(psz) : 0;
}
```

This technique has several possible applications.

Which applicatipons?

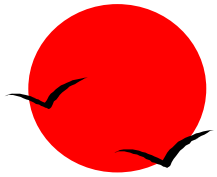


What are advantages of this kind of runtime polymorphism

There are two kinds of polymorphism: 1) via **configuration paramerizing**; 2) **Vptr/Vtab**.

The advantages include:

1) Separating the **design / development** and **deployment**;
2) Be able to use various implementations for the same interfaces;
3) Flexibility of configurations:
    1) in-process interoperation;
    2) out-of-process interoperation;
4) The generalization of the code (require the constructor function has no parameter or only a parameter of void * type.



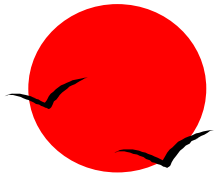
Issue 4: Object Extensibility

the implementation class can evolve over time without requiring client recompilation;

the interface class must keep invariable. This is because the interface is the semantic contract between the client and the vendor.

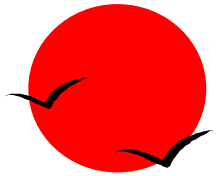
Any changes to the interface definition require that the client recompile to adapt it.

Worse yet, changing the interface definition completely violates the encapsulation of the object (its public interface has changed) and would break all existing clients.



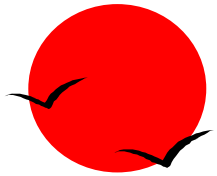
Object Extensibility

- The solution is to allow an implementation class to **expose more than one interface**.
- This can be achieved either by **designing an interface to derive from another related interface** or by **allowing an implementation class to inherit from several unrelated interface classes**.
- In either case, the client could use **C++'s Runtime Type Identification (RTTI) feature** to interrogate the object at runtime to ensure that the requested functionality is indeed supported by the object currently in use.



Object Extensibility

```
class IFastString {
        public:
        // version 1.0
        virtual void Delete(void) = 0;
        virtual int Length(void) = 0;
        virtual int Find(const char *psz) = 0;
        // version 2.0
        virtual int FindN(const char *psz, int n) = 0;
};
class IFastString2 : public IFastString {  // real version 2.0
        public:
        virtual int FindN(const char *psz, int n) = 0;
};
```

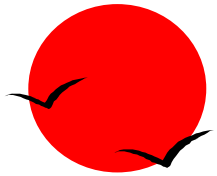


Object Extensibility

- Clients can interrogate the object reliably at runtime to discover whether or not the object is IFastString2 compatible by using C++'s dynamic_cast operator:

```
int Find10thBob(IFastString *pfs) {
    IFastString2 *pfs2 = dynamic_cast<IFastString2*>(pfs);
    if (pfs2)   // the object derives from IFastString2
        return pfs2->FindN("Bob", 10);
    else   // object doesn't derive from IFastString2
        return  –1;
}
```

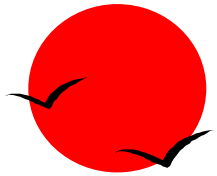
注意



Problems in the above method

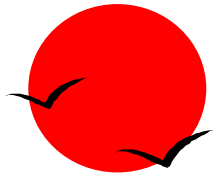
- This technique works because the compiler has enough knowledge about the **layout** and **type hierarchy** of **the implementation class** to examine an object **at runtime** to determine whether it in fact derives from IFastString2.

- **RTTI is a very compiler-dependent feature**. Again, Although there are the syntax and semantics specifications for RTTI, but each compiler vendor's implementation of RTTI is unique and proprietary. This effectively destroys the compiler independence that has been achieved by using abstract base classes as interfaces. This is unacceptable for a vendor-neutral component architecture.



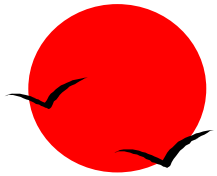
Solution approach: extending object interface: add new interface

```
class IFastString {
        public:
        virtual void *Dynamic_Cast(const char *pszType) =0;
        virtual void Delete(void) = 0;
        virtual int Length(void) = 0;
        virtual int Find(const char *psz) = 0;
};

class IPersistentObject {
        public:
        virtual void *Dynamic_Cast(const char *pszType) =0;
        virtual void Delete(void) = 0;
        virtual bool Load(const char *pszFileName) = 0;
        virtual bool Save(const char *pszFileName) = 0;
};
```



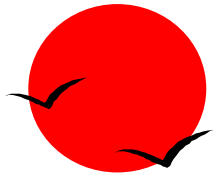
Hierarchical structure

```
class IExtensibleObject {
        public:
        virtual void *Dynamic_Cast(const char* pszType) =0;
        virtual void Delete(void) = 0;
};
class IPersistentObject : public IExtensibleObject {
        public:
        virtual bool Load(const char *pszFileName) = 0;
        virtual bool Save(const char *pszFileName) = 0;
};
class IFastString : public IExtensibleObject {
        public:
        virtual int Length(void) = 0;
        virtual int Find(const char *psz) = 0;
        }
```

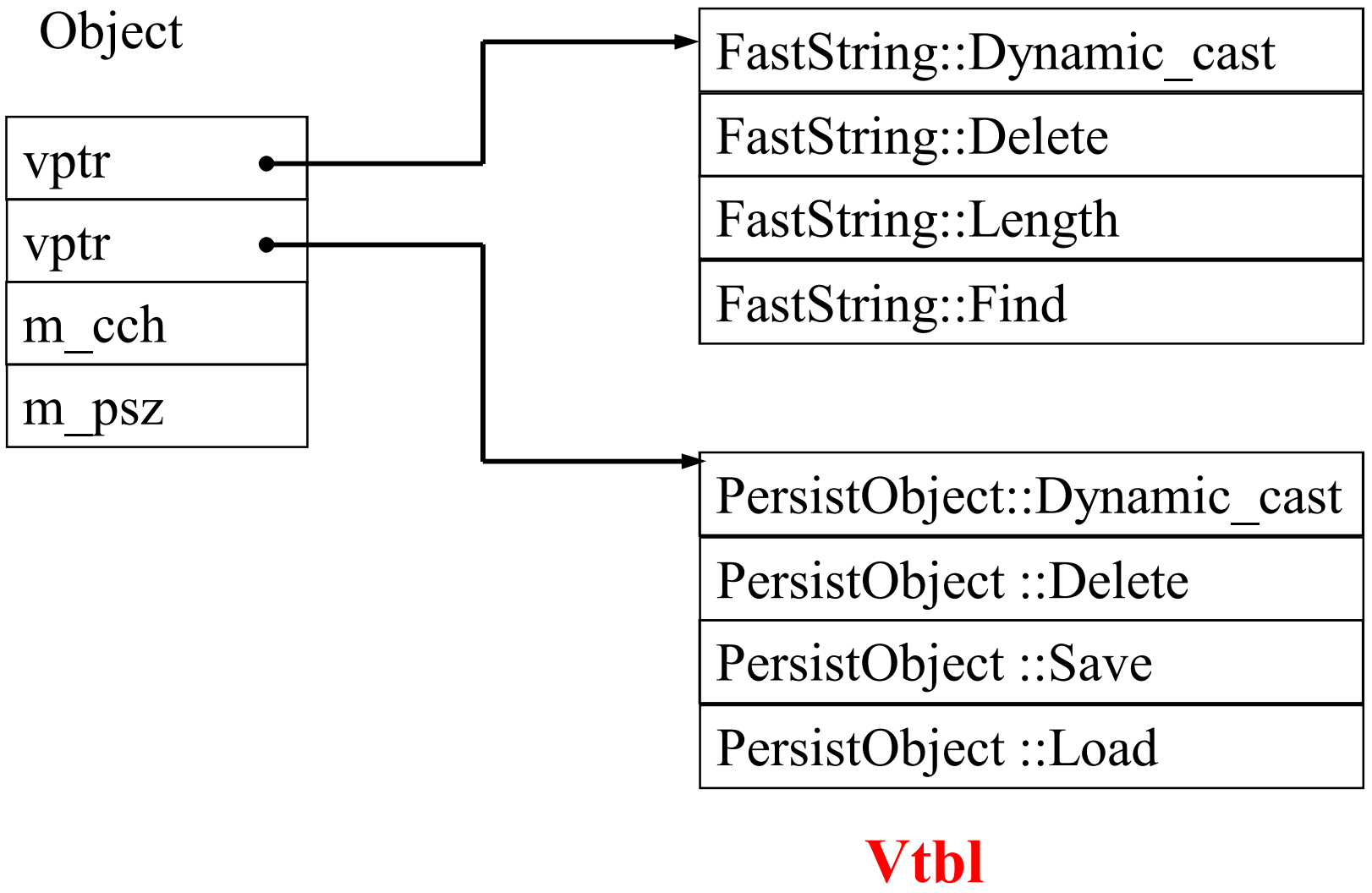


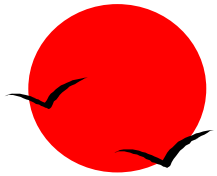
Implement multiple interfaces

```
class FastString : public IFastString, public IPersistentObject {
  ……….
   // IExtensibleObject methods:
       void *Dynamic_Cast(const char *pszType);
        void Delete(void); // deletes this instance

  // IFastString methods:
        int Length(void) const; // returns # of characters
        int Find(const char *psz) const; // returns offset

  // IPersistentObject methods:
       bool Load(const char *pszFileName);
       bool Save(const char *pszFileName);
  };
```



Memory layout



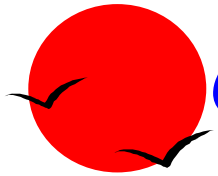


The implementation of Dynamic_Cast

The implementation of Dynamic_Cast needs to simulate the effects of RTTI by navigating the type hierarchy of the object.

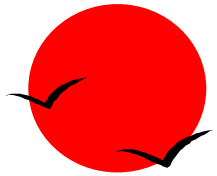
```
void *FastString::Dynamic_Cast(const char *pszType) {
    if (strcmp(pszType, "IFastString") == 0)
        return static_cast<IFastString*>(this);
    else if (strcmp(pszType, "IPersistentObject") == 0)
        return static_cast<IPersistentObject*>(this);
    else if (strcmp(pszType, "IExtensibleObject") == 0)
        return static_cast<IFastString*>(this);
    else
        return 0; // request for unsupported interface
}
```

Question: What is the difference between dynamic_cast and static_cast? `IFastString2 *pfs2 = dynamic_cast<IFastString2*>(pfs);`



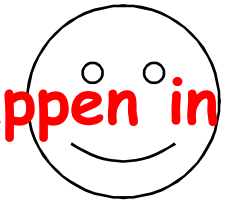
Client access to multiple interfaces

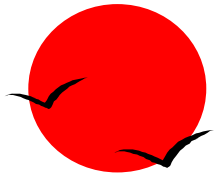
```
//client code:
bool SaveString(IFastString *pfs, const char *pszFN)  {
      bool bResult = false;
      IPersistentObject *ppo = (IPersistentObject*)
      pfs->Dynamic_Cast( "IPersistentObject");
      if (ppo)
          bResult = ppo->Save(pszFN);   return bResult;
}
```



Issue 5: Resource Management

```
//Client code:
void f(void) {
        IFastString *pfs = 0;
        IPersistentObject *ppo = 0;
        pfs = CreateFastString("Feed B0B");
        if (pfs)  {
            ppo = (IPersistentObject *)pfs->Dynamic_Cast(
                                              "IPersistentObject");
           if (ppo)     {
                 ppo->Save("C:\\autoexec.bat");
                 ppo->Delete();
           }
            pfs->Delete();
        }
}
```





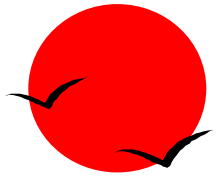
Resource Management

- **Call Delete only once per object!**

- **managing these relationships becomes quite complex and error prone.**

- **One way to simplify the client's task is to push the responsibility for managing the lifetime of the object down to the implementation.**

- **After all, allowing the client to delete an object explicitly betrays yet another implementation detail: the fact that the object is allocated on the heap.**

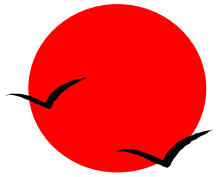


Solution to Resource Management

One simple solution to this problem is to have **each object maintain a reference count** that **is incremented when an interface pointer is duplicated and decremented when an interface pointer is destroyed.**

This means changing the definition of IExtensibleObject from:

```
class IExtensibleObject {
    public:
    virtual void Delete(void) = 0;
    virtual void *Dynamic_Cast(const char* pszType) = 0;
    virtual void DuplicatePointer(void) = 0;
    virtual void DestroyPointer(void) = 0;
};
```



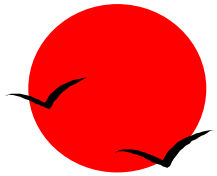
Reference count

all users of IExtensibleObject must now adhere to the following two mandates:

(1) When an interface pointer is duplicated, a call to DuplicatePointer is required.

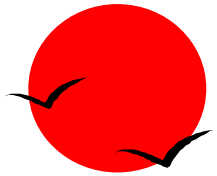
(2) When an interface pointer is no longer in use, a call to DestroyPointer is required.

These methods could be implemented in each object simply by noting the number of live pointers and destroying the object when no outstanding pointers remain:



Implement reference count

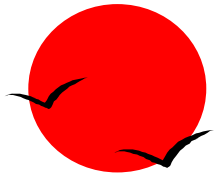
```
class FastString : public IFastString, public IPersistentObject {
        ……
        int m_cPtrs; // add a data member for reference count
        ……
      void DuplicatePointer(void);
      void DestroyPointer(void)
}
```



Implement reference count

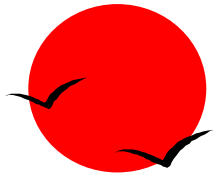
```
//implementation:
// initialize pointer count to zero in constructor:
FastString::FastString(const char *psz) : m_cPtrs(0) { …..}

// increment count:
void FastString:: DuplicatePointer(void)  {
      ++m_cPtrs;
}
// decrement count and destroy object when last pointer destroyed:
void FastString::DestroyPointer(void)  {
     if (--m_cPtrs == 0)
          delete this;
}
```

Note: What problem will occur when multiple clients access to it concurrently?



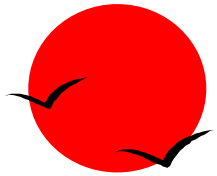
Corresponding modification to entry point

```
IFastString *CreateFastString (const char *psz)   {
        IFastString * pfsResult = new FastString(psz);
         if (pfsResult )
                pfsResult ->DuplicatePointer();
        Return pfsResult;
}
```



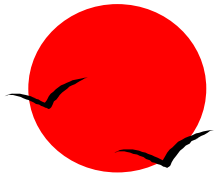
Corresponding modification to interface query

```
void *FastString::Dynamic_Cast(const char *pszType) {
      Void * pfsResult = 0 ;
      if (strcmp(pszType, "IFastString") == 0)
            pfsResult = static_cast<IFastString*>(this);
      else if (strcmp(pszType, "IPersistentObject") == 0)
            pfsResult = static_cast<IPersistentObject*>(this);
      else if (strcmp(pszType, "IExtensibleObject") == 0)
            pfsResult = static_cast<IFastString*>(this);
      else
            return 0; // request for unsupported interface
      ((IExtensibleObject *)pfsResult)->DuplicatePointer();
      Return  pfsResult;
}
```



Corresponding modification to Client access code

```
//Client code:
void f(void) {
      IFastString *pfs = 0;
      IPersistentObject *ppo = 0;
      pfs = CreateFastString("Feed B0B");
      if (pfs)  {
          ppo = (IPersistentObject *)
          pfs->Dynamic_Cast( "IPersistentObject");
         if (ppo)     {
                ppo->Save("C:\\autoexec.bat");
                ppo->DestroyPointer();
         }
          pfs-> DestroyPointer();;
      }
}
```



Perspective

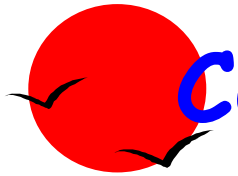
Now each pointer is now treated as an autonomous entity with respect to lifetime in clients' perspective. Thus the client does not need to be concerned with which pointer refers to which object.

Instead, the client simply follows the two simple rules and allows the object to manage its own lifetime. with a rule moved to the implementation, clients only care about the other rule.

Now all interfaces are peer;

But the problem is not completely solved, **Why?**

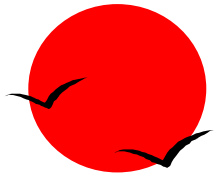
When DestroyPointer() is called repeatly,
the process will terminate unexpectively.



Component technology: IUnKnown

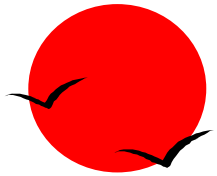
IUnKnown的C++定义：

```
Extern “C” const IId IID_IUnKnown;
Interface IUnKnown  {
        virtual void Delete(void) = 0;
        Virtual  bool  QueryInterface(REFIID riid, void **ppv) = 0;
        Virtual  int  AddRef( void ) = 0;
        Virtual int  Release( void ) = 0;
}
```



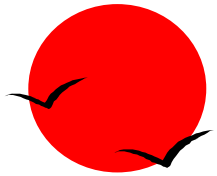
Client access code

```
//Client code:
void f(void) {
        IFastString *pfs = 0;
        IPersistentObject *ppo = 0;
        pfs = CreateFastString("Feed B0B");
        if (pfs)  {
            ppo = (IPersistentObject *)
            pfs->Dynamic_Cast( "IPersistentObject");
           if (ppo)     {
                  ppo->Save("C:\\autoexec.bat");
                  ppo->DestroyPointer();
                  ppo = 0;
           }
            pfs-> DestroyPointer();;
            pfs = 0;
        }
}
```



Chapter Summary

- ✓The problems with object-oriented programming in interoperation:
  - Name resolution problems;
  - Losing encapsulation;
  - Coupling problems (Version issues);
  - implememntation incompatibilities issues**;**
- ✓Separating **interfaces** from **implementations** allows the object's layout to evolve without requiring client recompilation.
- ✓Interfaces took the form of a **vptr** and **vtbl**.
- ✓Two Key issues in the Component Object Model:
  1) Interface and object **extensibility**;
  2) Object **lifetime** management. (**construction**, **destruction**, **lifetime determination**)



Component Technology

Component technology: Object-oriented Programming +

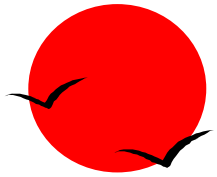
Interface +

binary Reuse.

Component Object Model:

Idea: Separating interface from implementation. Characteristics:

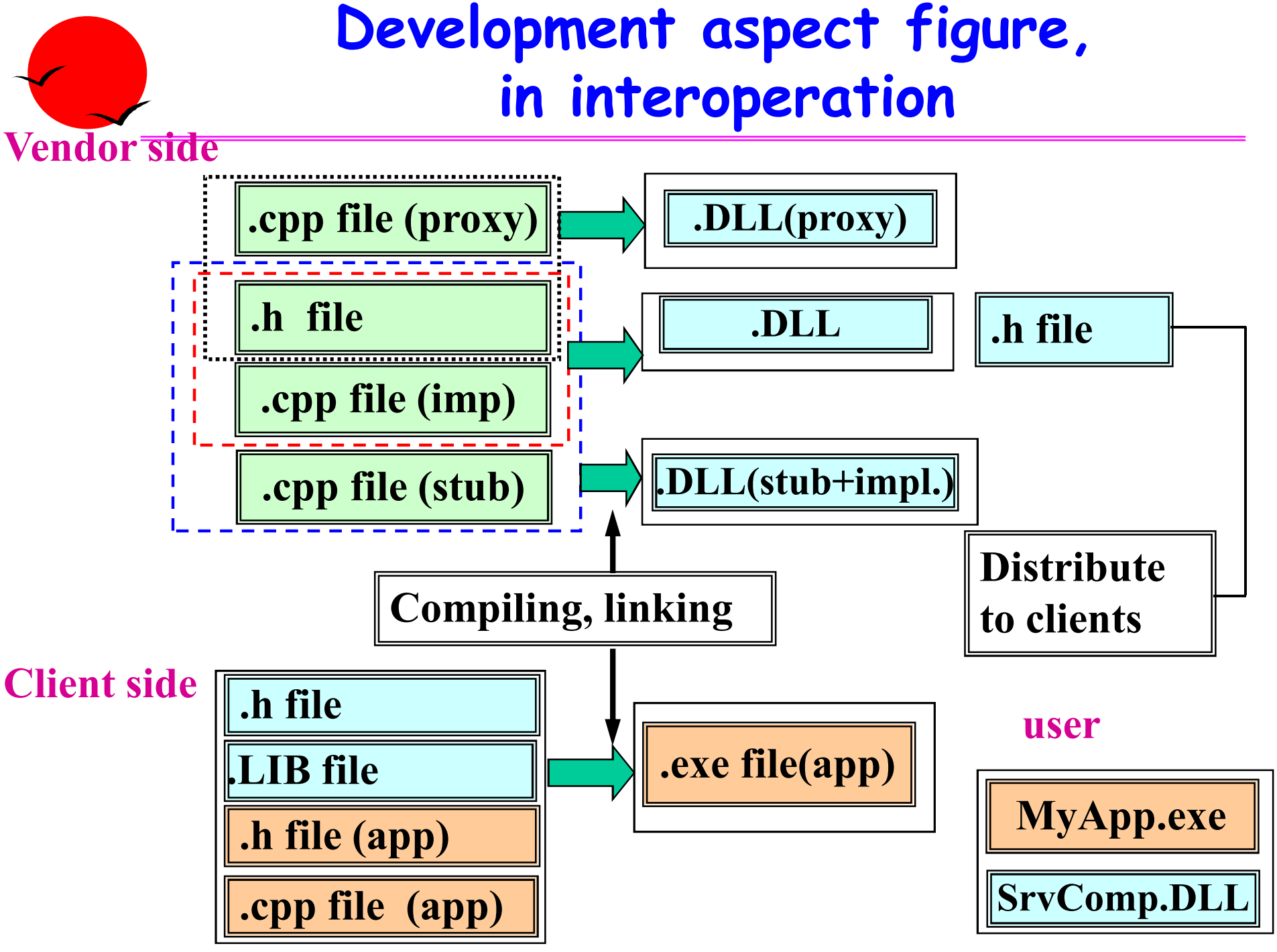
1) **loose coupling** : the change of implementation class do not the recompilation of client application;
2) **Keep encapsulation** feature in object-oriented;
3) **Bypass name resolution** problems;
4) Achieve **Binary interoperation** by utilizing **vptr/vtbl**;
5) After solving the above problems, binary interoperation becomes **independent on C++ compilers**.



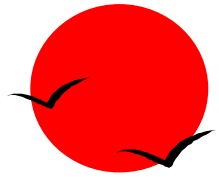
Interface characteristics

- 二进制特性；
- 接口不变性；
- 继承性(纵向），和扩展性（横向）；
- 全球范围内的标识性；
- 多态性——运行过程中的多态性；
- 对等性；
- 完全联通性；

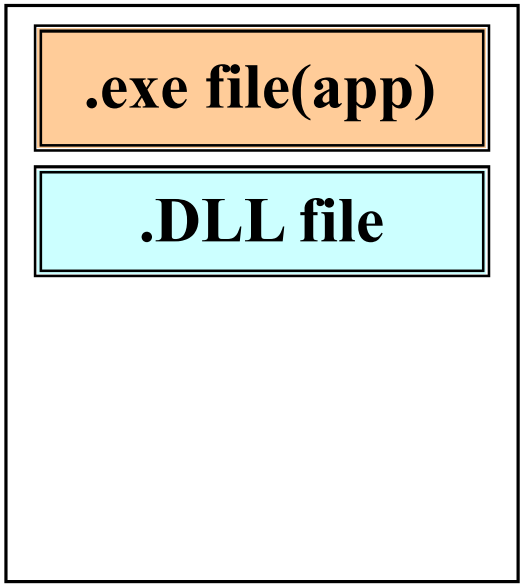
Development aspect figure, in interoperation



Deploying aspect figure, in interoperation

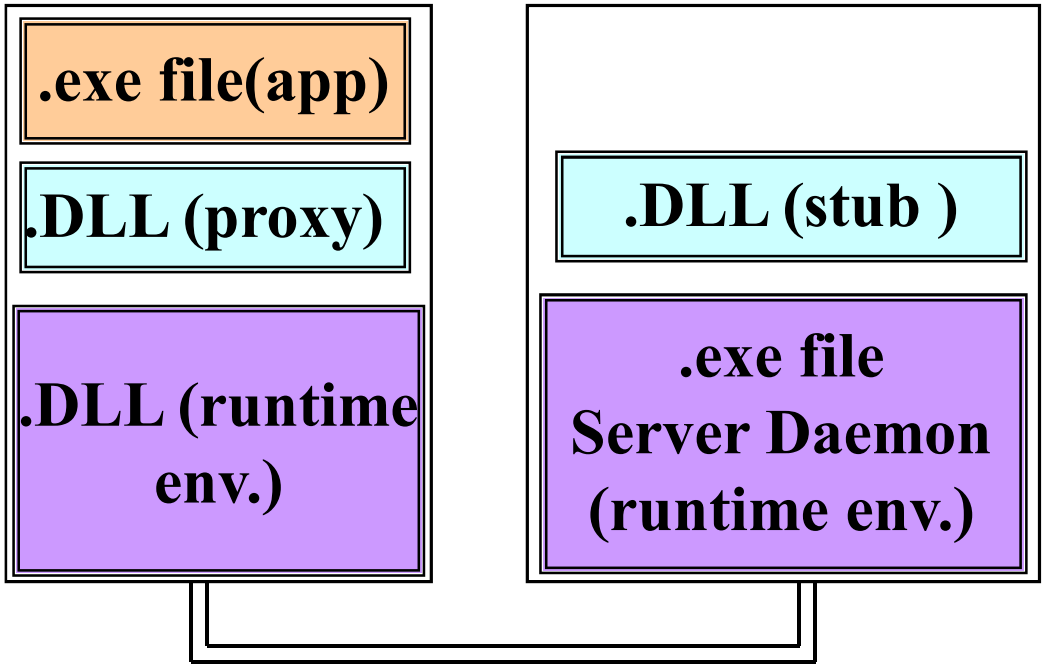


Case 1: in-process pattern



process

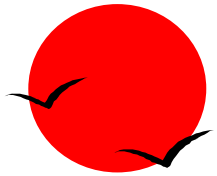
Case 2: inter-process/machine



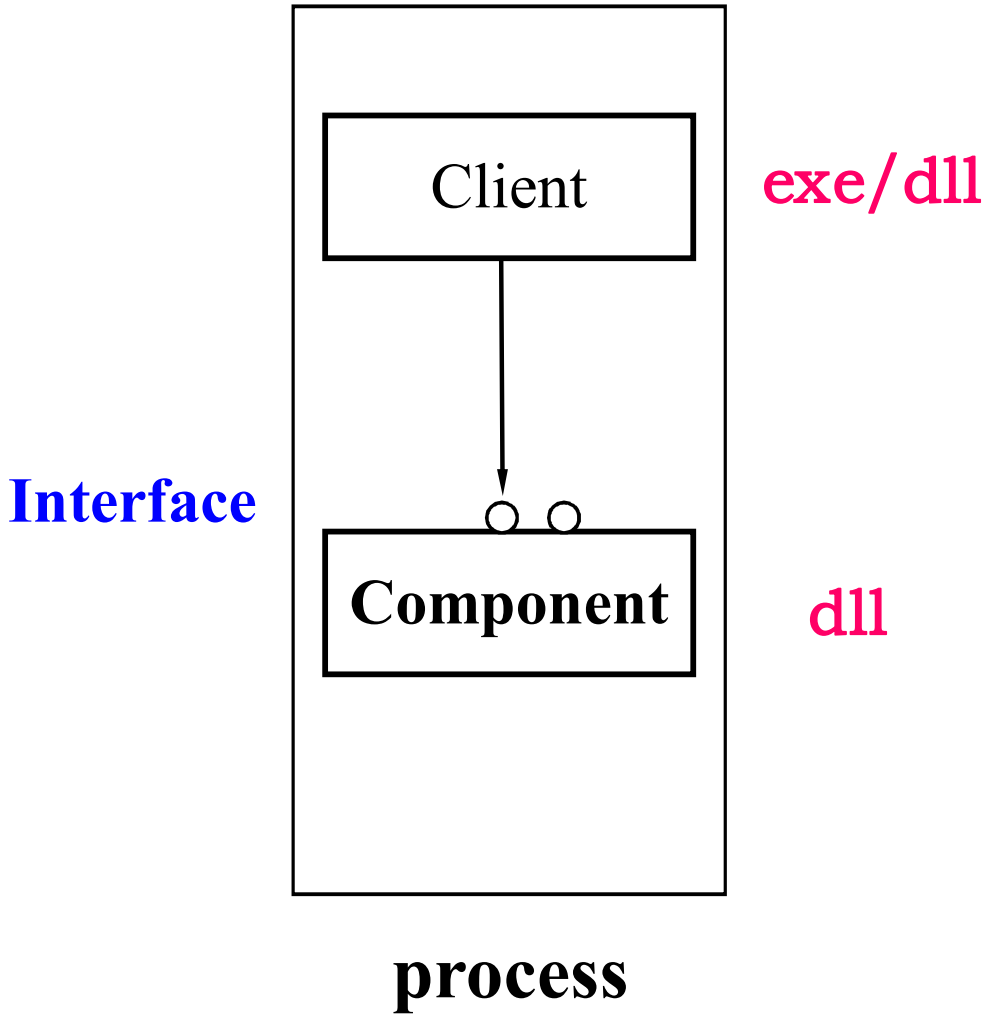
process

process

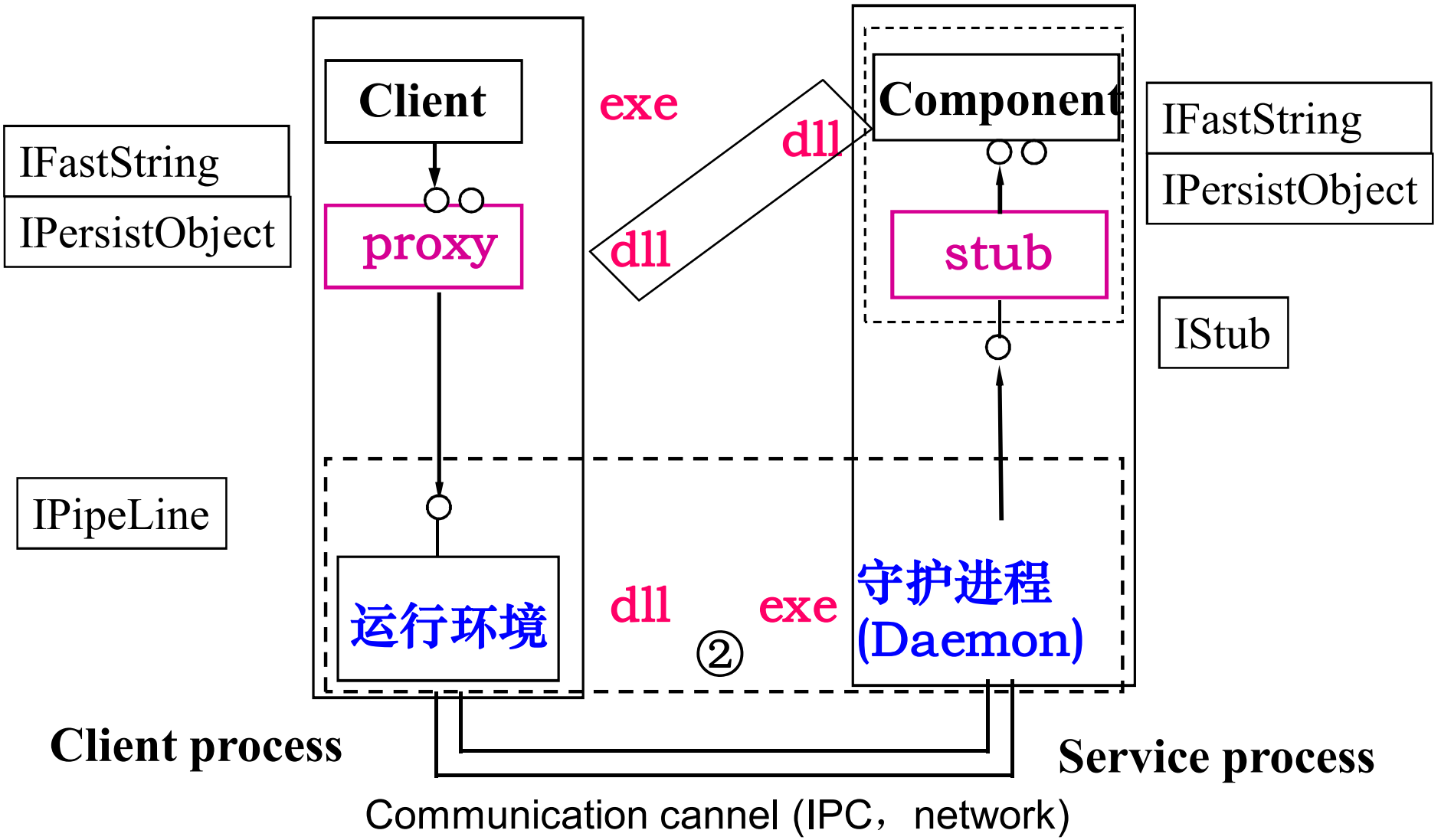
User side

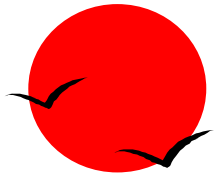


Model aspect figure, interoperation case 1: in-process

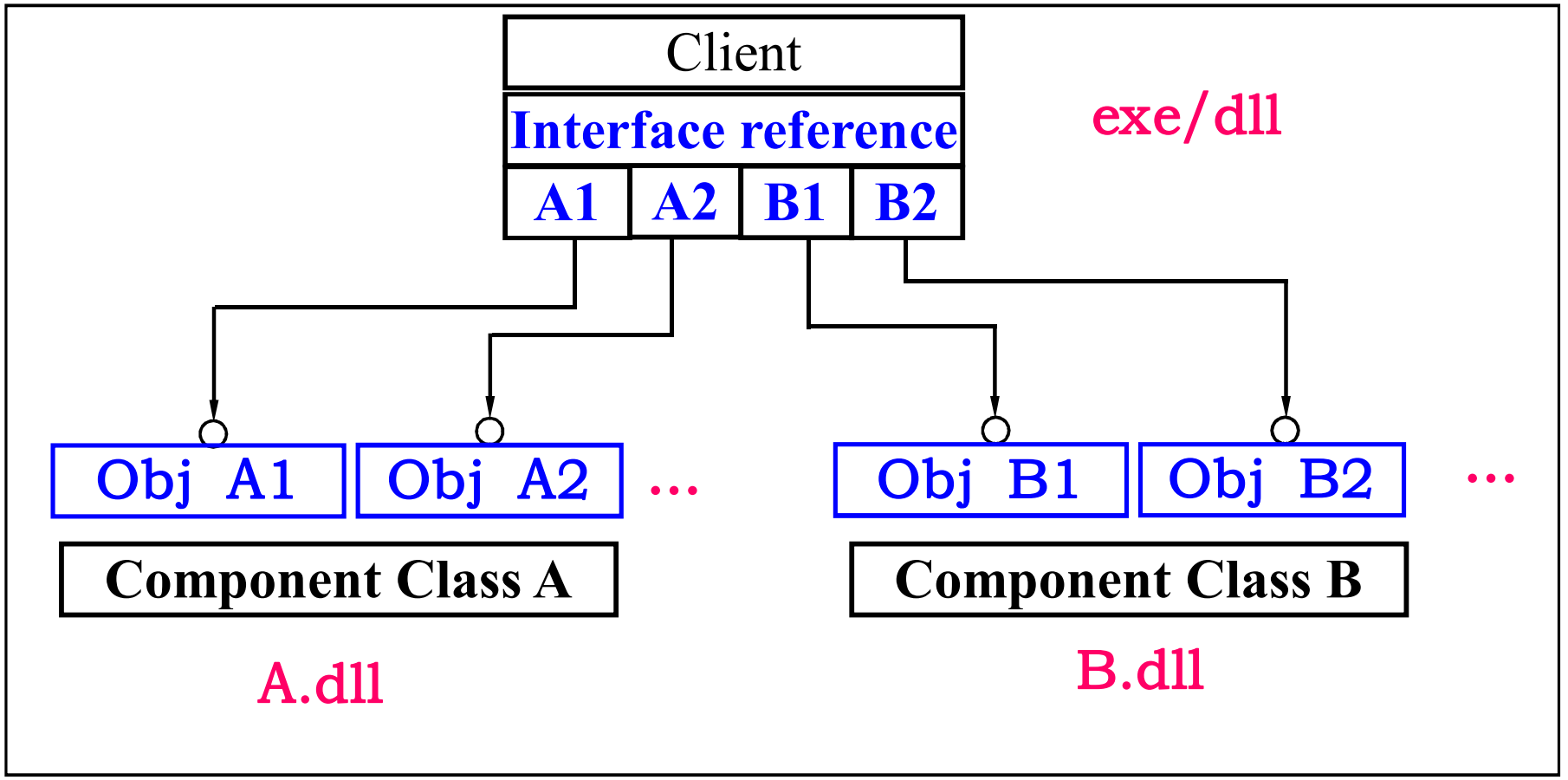


Model aspect figure, interoperation case 2: inter-process



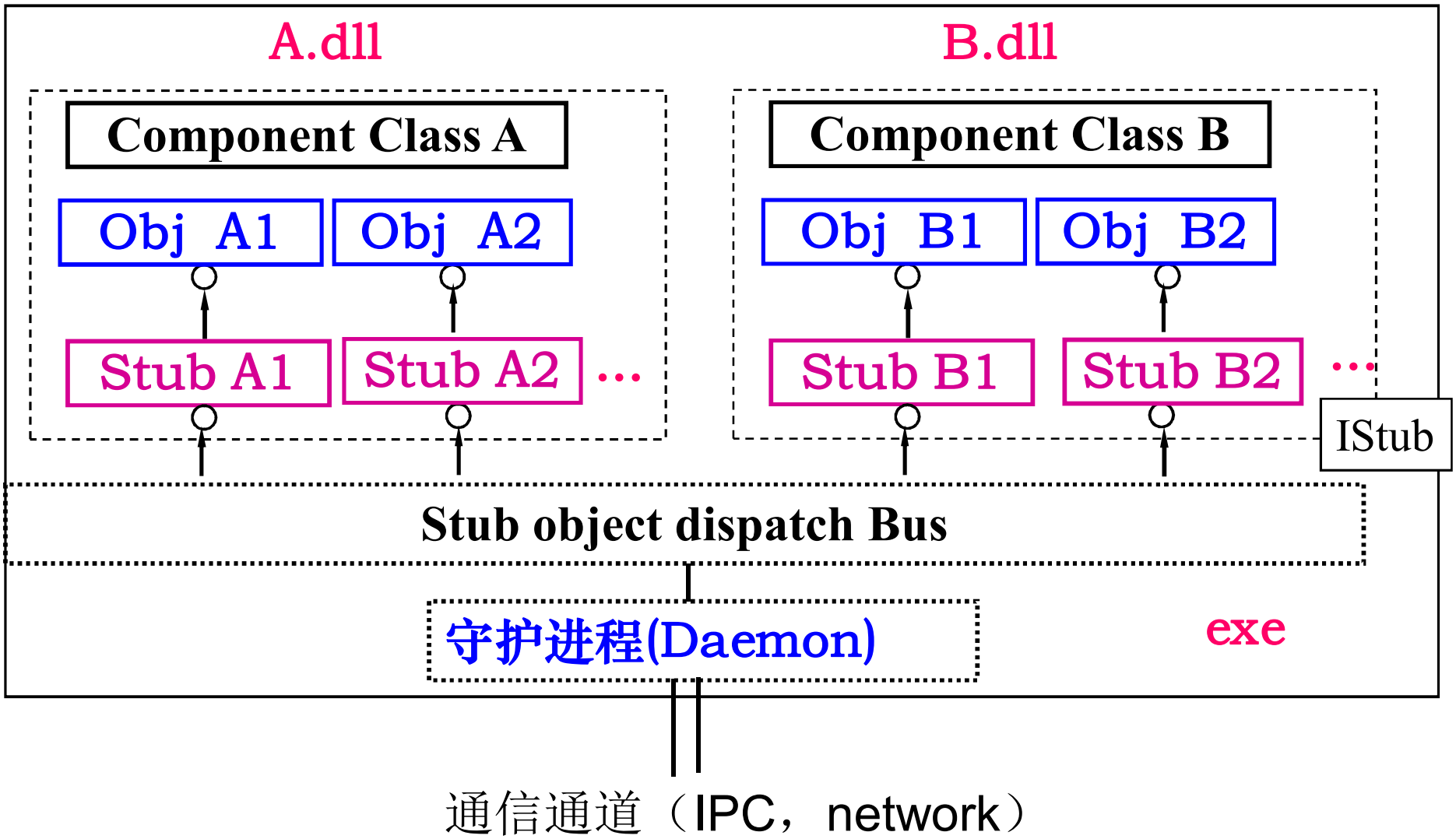


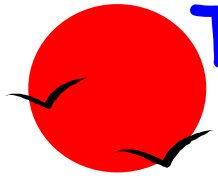
Runtime aspect figure, interoperation case 1: in-process



in-process

Runtime aspect figure, interoperation case 2 :Service process



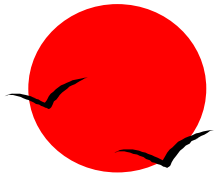


The original Interface definition for interoperation

例如：

UnKnown.idl：

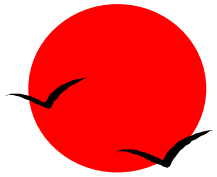
```
//禁止为该接口产生网络代码。
[local, object,  uuid(00000000-0000-C000-000000000046]
 Interface IUnKnown  {
        void Delete(void);
     HRESULT QueryInterface([in] REFIID riid, [out] void **ppv);
        ULONG AddRef( void ) ;
        ULONG Release( void );
}
```



IUnKnown与C++定义的映射

IUnKnown的C++定义：

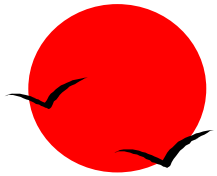
```
Extern “C” const IId IID_IUnKnown;
Interface IUnKnown  {
        virtual void STDMETHODCALLTYPE Delete(void) = 0;
        Virtual HRESULT STDMETHODCALLTYPE
                QueryInterface(REFIID riid, void **ppv) = 0;
        Virtual ULONG STDMETHODCALLTYPE AddRef( void ) = 0;
        Virtual ULONG STDMETHODCALLTYPE Release( void ) = 0;
}

#define  STDMETHODCALLTYPE  _stdcall
```



C++中简写IUnKnown

IUnKnown的C++定义：

```
Extern “C” const IId IID_IUnKnown;
Interface IUnKnown  {
        virtual void Delete(void) = 0;
        Virtual  bool  QueryInterface(REFIID riid, void **ppv) = 0;
        Virtual  int  AddRef( void ) = 0;
        Virtual int  Release( void ) = 0;
}
```



Demo: Interface definition in interoperation

运行环境要提供如下接口，供proxy来调用：

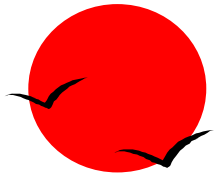
```
class IPipeLine: public IUnknown  {
   virtual int SendMessage(void *pmsg, int MsgLen) = 0;
   virtual int ReceiveMessage(void *pmsg) = 0;
}
```

Stub要提供如下接口，供服务器来调用：

```
class IStub: public IUnknown  {
    virtual int  Invoke(void *pmsg, void *Result, int ResultLen) = 0;
}
```

服务组建要提供如下接口，供客户来调用：

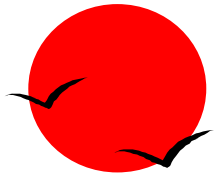
```
class IFastString : public IUnknown {
    virtual int Length(void) = 0;
    virtual int Find(const char *psz) = 0;
}
class IPersistObject : public IUnknown {
    virtual bool Load(const char *pszFileName) = 0;
    virtual bool Save(const char *pszFileName) = 0;
};
```



Demo：运行环境提供商提供

Pipelin.h：

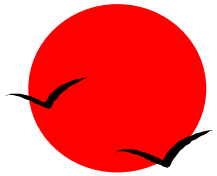
```
class IPipeLine: public IUnknown  {
   virtual int SendMessage(void *pmsg, int MsgLen) = 0;
   virtual int ReceiveMessage(void *pmsg) = 0;
}

extern "C“__declspec(dllimport) IPipeline *CreatePipeline (char
*pSrvIP, int port);
```



Demo：运行环境提供商提供

Pipeline_impl.h：

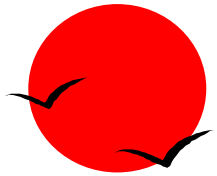
```
Class CPipeline:  public IPipeLine  {

        int m_cPtrs;

        INT_SOCKET    hSocket;

       CPipeline( char *pSrvIPAddr,  int port);
        ~Cpipeline(  );

         virtual void Delete(void) = 0;
        virtual  bool  QueryInterface(REFIID riid, void **ppv) = 0;
         virtual  int  AddRef( void ) = 0;
         virtual int  Release( void ) = 0;
       virtual int SendMessage(void *pmsg, int MsgLen);
       virtual int ReceiveMessage(void *pmsg);
}

extern "C“__declspec(dllexport) IPipeline *CreatePipeline (char *pSrvIP, int port);
```



Demo：运行环境提供商提供

Pipeline_impl.cpp：

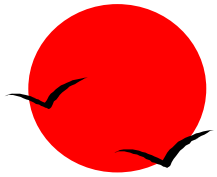
```
CPipeline::CPipeline( char *pSrvIPAddr,  int port)  {  .....  }
Void  CPipeline :: Delete(void)  {
        delete this;   }
bool CPipeline :: QueryInterface(REFIID riid, (void **)  ppv)  {
        If (riid == IID_IPIPELINE || riid == IID_IUnKnown)
            *ppv = static_cast<IPipeline *>(this);
        else {
              *ppv = 0;
               return false;  }
        Reinterpret_cast <IUnKnown *> (*ppv)->AddRef();
        Return ture;   }
 int CPipeline :: AddRef(void)  {
        return InterlockedIncrement(&m_cPtrs);}
 int CPipeline :: Release(void)  {
          LONG res = InterlockedDecrement(&m_cPtrs);
          if res == 0)   delete this;
          return res; }
IPipeline *CreatePipeline (char *pSrvIPAddr, int port)   {
          return new CPipelineg(pSrvIPAddr,  port);
}
```



组件提供商提供

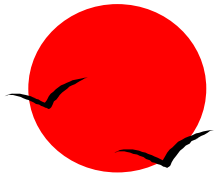
FastString.h：

```
class IFastString : public IUnknow {
    virtual int Length(void) = 0;
    virtual int Find(const char *psz) = 0;
}
class IPersistObject: public IUnknow {
    virtual bool Load(const char *pszFileName) = 0;
    virtual bool Save(const char *pszFileName) = 0;
};

extern "C“__declspec(dllimport) IPipeline *CreateFastString (char
*pSrvIP, int port);
```



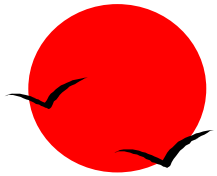
Proxy implementation class

```
Class FastString_proxy: public IFastString, IPersistObject  {
    int m_cPtrs;

    int   StubObjectID;
    IPipeLine *pChannel;

    FastString_proxy(const char *psz);
    ~ FastString_proxy(void);

    virtual void Delete(void) = 0;
    virtual  bool  QueryInterface(REFIID riid, void **ppv) = 0;
    virtual  int  AddRef( void ) = 0;
    virtual int  Release( void ) = 0;

   virtual int    Length(void);
   virtual int    Find(const char *psz);
   virtual bool  Load(const char *pszFileName);
   virtual bool  Save(const char *pszFileName);
}
```



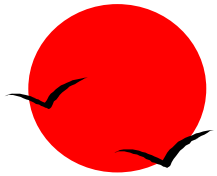
Implementing FastString_proxy constructor

```
Int FastString_proxy:: FastString_proxy(char *psz)   {
        IPipeLine *   (*pfn)(char *, int) = 0;
        HINSTANCE h = LoadLibrary("PipeLine.dll");
         if (h)
            *(FARPROC* )&pfn = GetProcAddress(h, "CreatePipeLine");
         If (pfn)
              pChannel = pfn("102.98.105.67", 80);
        sprintf(Buffer, "initial; FastString.dll; CreateFastString; %s", psz);
        pChannel ->SendMessage(Buffer, strlen(Buffer);
        pChannel ->ReceiveMessage(& StubObjectID);
}
```



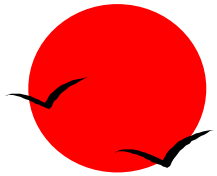
Implementing FastString_proxy constructor

```
virtual int FastString_proxy:: Find(char *psz)   {
        if  (StubObjectID! =null)   {
              sprintf(Buffer, "calling; %d; IFastString; Find; %s",
                        StubObjectID, psz);
               pChannel ->SendMessage(Buffer, strlen(Buffer);
               int len = pChannel ->ReceiveMessage(Buffer);
               return  atoi(Buffer, len);
        }
         else  return null;
}
```



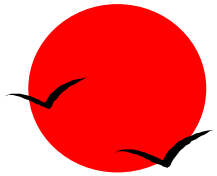
Stub implementation class

```
Class FastString_stub: public IStub  {
        int m_cPtrs;
      IFastString *pfs;
        FastString_stub(const char *psz);
        ~ FastString_stub(void);
      virtual void Delete(void) = 0;
      virtual  bool  QueryInterface(REFIID riid, void **ppv) = 0;
      virtual  int  AddRef( void ) = 0;
      virtual int  Release( void ) = 0;
       virtual int Invoke(void *pmsg, void *Result, int ResultLen);
}
extern "C" __declspec(dllexport) IStub * CreateFastString(void *psz);
```



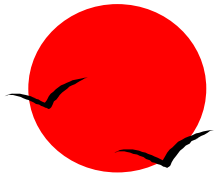
Implementing FastString_stub constructor

```
extern "C" __declspec(dllexport)  IStub * CreateFastString(void
*parameter)
{
      return new FastString_stub((char *) parameter );
}

FastString_stub::FastString_stub(char *psz)   {
      pfs = new FastString(psz);
}
```



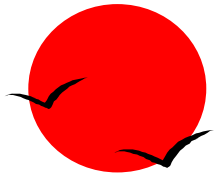
Client calling a component

```
# Include “IFastString.h”
int f(void) {
        static IFastString * (*pfn)(const char *) = 0;
        Char pszDll[80];
        if (!pfn) {
        ReadDeploymentParameter(“ComponentDllName”, pszDll);
        HINSTANCE h = LoadLibrary(pszDll);
        if (h)
            *(FARPROC *  )&pfn = GetProcAddress(h, "CreateFastString");
             if (pfn)   {
                    IFastString *pfs = pfn( ("Hi Bob!");
                    if (pfs)   {
                          n = pfs->Find("ob");
                         pfs->Delete();
}
```



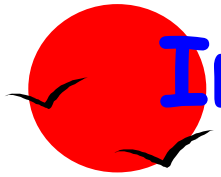
Implementing server Daemon

```
void _ServerThread(    )   {
  unsigned char szDllName[128], szFunName[128], pBuffer[4096]
  IStub  * (*pfn)( void *) = 0;
  Pipeline = CreatePipelineBus("interoperation");
  while (1)   {
       WaitforSingleObject(hCallEvent);
       if   (Pipeline.FetchFlag( ) =="initial")   {
            Pipeline.FetchDllName(szDllName);
            Pipeline. FetchFunctionName(szFunName);
            Pipeline. FetchParameters(pBuffer);
            HINSTANCE h = LoadLibrary(szDllName);
            if (h)
                 *(FARPROC* )&pfn = GetProcAddress(h, szFunName);
            If (pfn)
                  IStub *InterfaceReference = pfn(pBuffer);
            Pipeline.AddReferenceValue(InterfaceReference );
        }
```



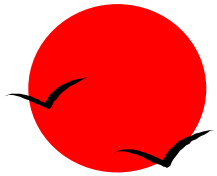
Implementing server Daemon

```
        Else  {
                IStub *p;
                Pipeline. FetchInterfaceReference(&p);
                Pipeline. FetchParameters(pBuffer);
                p->Invoke(pBuffer, &Result, &ResultLen);
                Pipeline.AddResult(Result, ResultLen );
        }
      TriggerEvent(hReturnEvent);
   }  //while
}
```



Implement FastString_stub::Invoke

```
Int FastString_stub::Invoke(void *pmsg, void *Result,  int &ResultLen)   {
        InterfaceName = AbstractInterfaceName(pmsg);
        FunctionName = AbstractFunctionName(pmsg);
        If  (InterfaceName = = "IFastString")  {
          if (FunctionName = = "Find")  {
              char * pParameter = AbstractString(pmsg);
               ret = p->Find(p pParameter);
               sprintf (Result,"%d",ret);
               ResultLen = strlen(Result);
               return 1;
         else if (FunctionName = = "Length") {
              …….;
          }
        }
        Else If  (InterfaceName = = "IPersitObject")  {
           IPersistObject *p2 = p-> Dynamic_Cast("IPersitObject") ;
           ……;
        }
```



Homework 1

In the demonstration example:

```
class CFastString : public IFastString, IPersistObject {
    .......
}
```

Suppose three cases: 1) in-process interoperation; 2) out-of-process interoperation in a machine; 3) out-of-process interoperation in different machine;

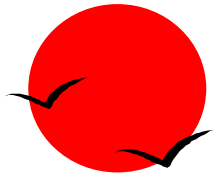
Use C++ to complete the following tasks:

1) Write a PipleLine as a part of runtime environment to implement communication channel between two processes; (a .dll file)

2) Write a server daemon as a part of runtime environment to implement interoperation between two modules in different processse; (a .exe file)

3) Complete coding of its **proxy and stub, respectively;**

4) Write a client program and a service program to implement interoperation betwwen two module.

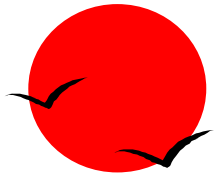


Homework 2

Calling DestroyPointer( ) is of dangerous, when DestroyPointer() is called extra more one time, the process will likely terminate unexpectedly. How do you think to solve this problem?

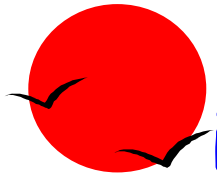
In addition, How to automate reference counting to a object to void memory leak? Consider it with respect to program model, compiler, and separation strategy.

How to enable object lifetime management transparent to client programmer?



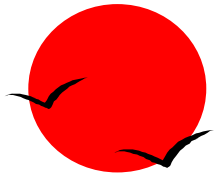
Analysis

Require client programmers to set the interface pointer variable to zero when defining it (compiler can help to achieve it on behalf of programmers). After having used a interface pointer variable, client programmers should reset it to zero again as long as it is not used any longer (compiler can help to achieve it on behalf of programmers if the interface pointer variable is defined as a local variable or a member variable of a class). if the interface pointer variable is defined as a global variable or a static variable, compiler cannot recognize what time the interface is not used any more. In this case, memory leaking could occur.



Seminar: how to make binary code independent on its position in runtime, in terms of Vptr/Vtab

When absolute address scheme is employed to the addressing of resource or data in binary executable code, if a module is not loaded in the expected location in process virtual address space, all addresses of the resources in the code of this module need to be modified to the new address. This situation fails to make code independent on its position, losing the characteristic of completely read-only. The technique we have devised for procedure-oriented programming cannot apply to this situation. The reason is that, when calling a virtual function in runtime, the caller don't know which module the callee is located in.



A scheme to achieve code independent on its position in runtime in object-oriented programming

We add a address variable just before every *vtab* to store the loaded address of the module. When a module is not loaded in the expected address in process address space, all address variables mentioned above are modified to the really loaded address. Thus, the technique in procedure-oriented programming can be applied to object-oriented programming. In the address of *Vptr* – 4, the caller can get the loaded address of the module the callee is located in.