

# Component Object Model (COM): Problem of C++ Binary Standard

Seong Jong Choi  
chois@mmlab.net

[Multimedia Lab.](#)

Dept. of Electrical and Computer Eng.  
University of Seoul  
Seoul, Korea

# 1. Main Objectives

---

- Understand the problems related to using C++ as component substrate.

# Problem

---

- An **application developer** adopts a component from a **component developer**.
- C++ Software distribution method
  - Case 1: Source code distribution
  - Case 2: Dynamic Link Library (DLL) distribution

# Const: 3cases

---

- In declaration

```
Const int a;
```

```
Char * const b;
```

- Function arguments

```
Foo( const char *FastString);
```

- Class member function

```
Class FastString {
```

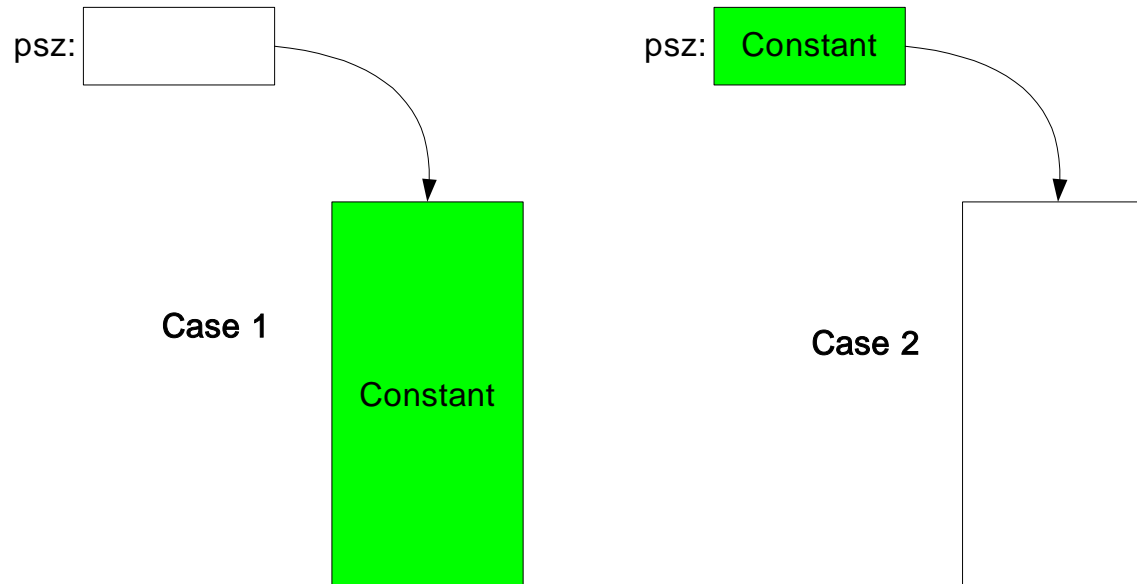
```
    ...
```

```
    int foo(void) const;
```

```
}
```

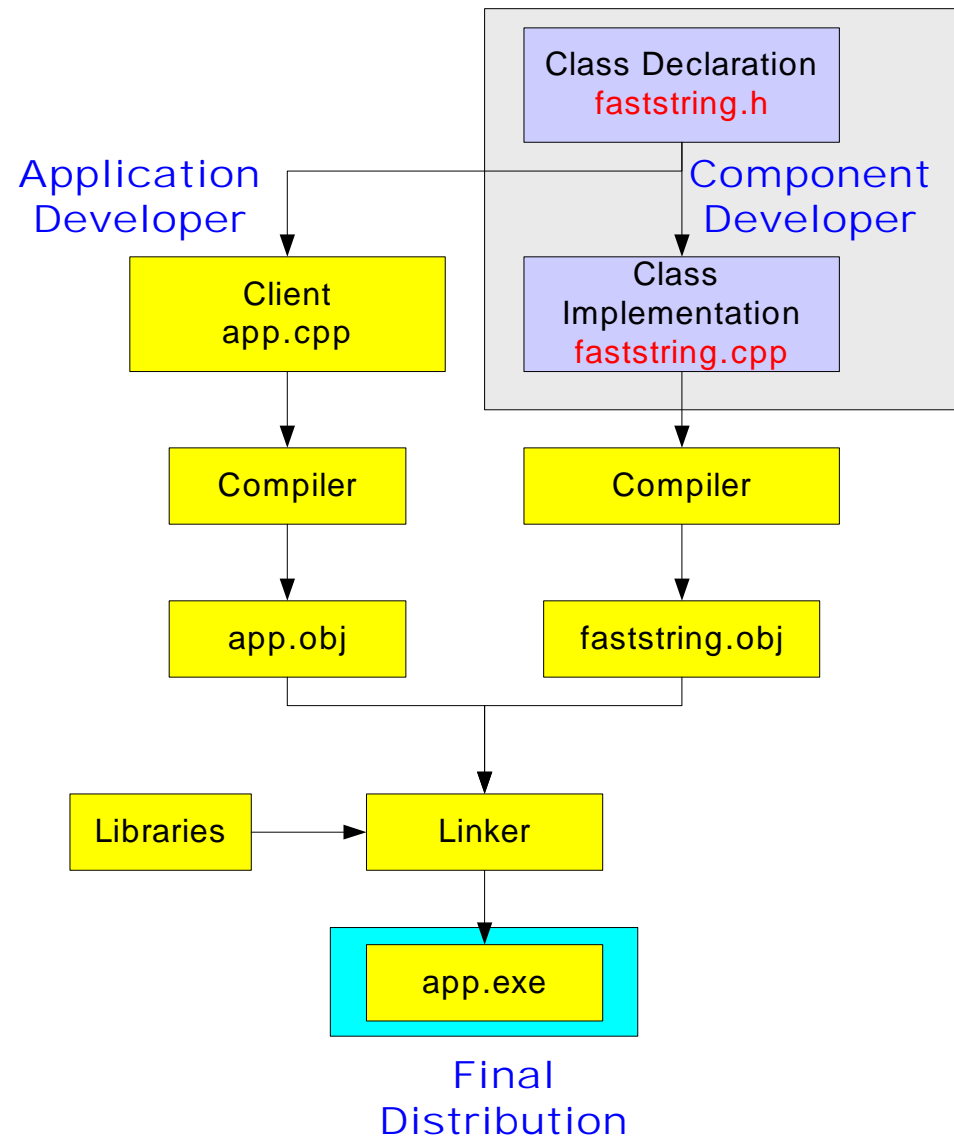
# const - declaration

- ```
const int a;  
#define const debugging
```
- **Pointer and constants**  
Case 1: `const char *psz;` (== `char const* psz;`)  
Case 2: `char * const psz;`



# Source Code Distribution: Method

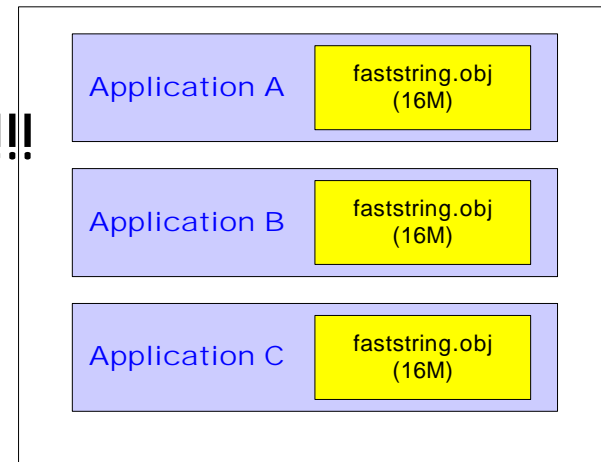
- Static Library
- Application Developer
  - Application source
  - developing Environment
- Component Developer
  - Class definition
  - Implementation



# Source Code Distribution: Discussion

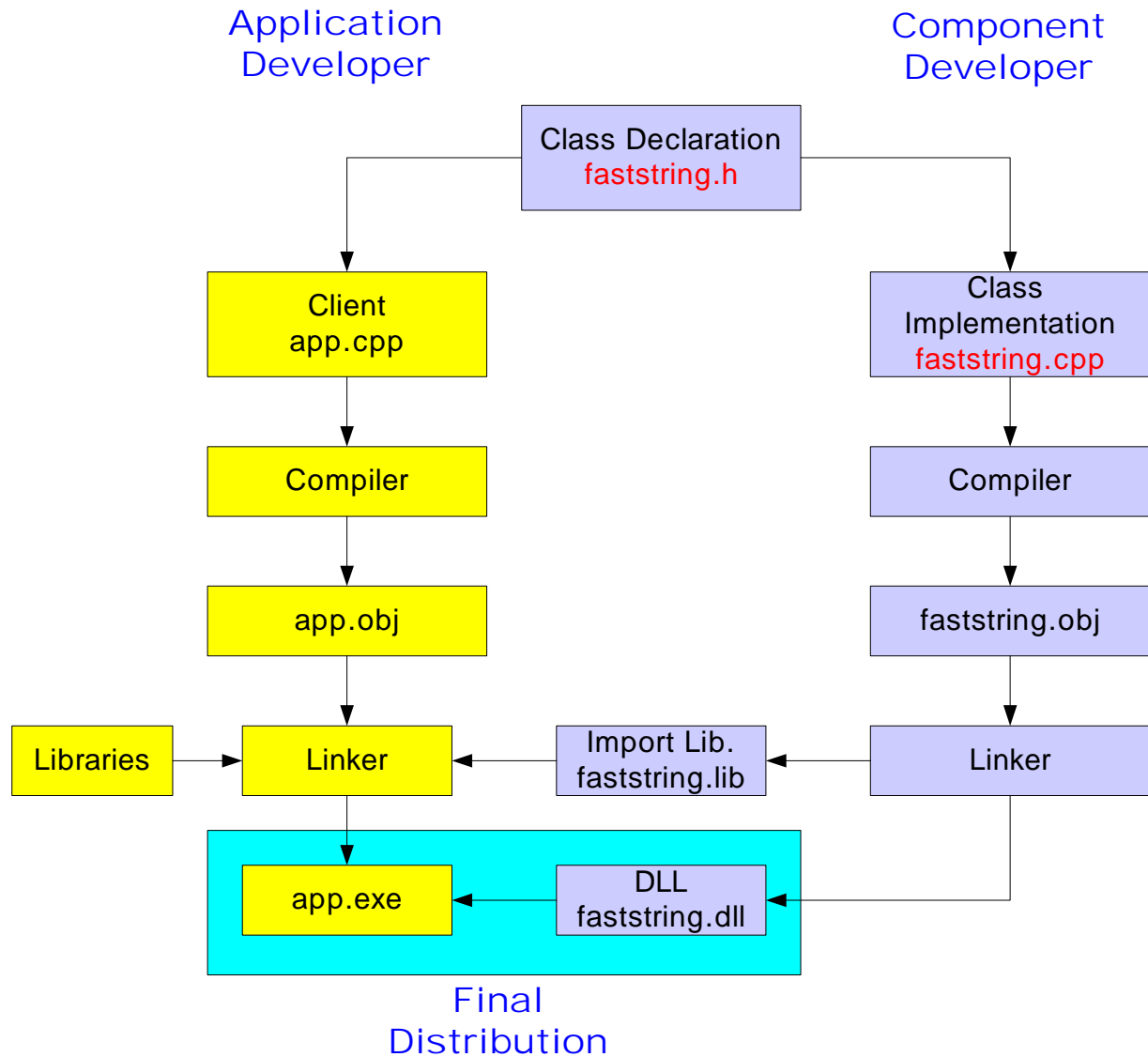
- Perfectly workable if the library sources are compatible with the new compiler
- However
  - It take too much spaces.
  - Library source are open to others.
  - If they want to modify the library source, then they have to rebuild and redistribute the application.

- **No binary modularity** of the library!!!



A Client

# DLL Distribution: Method



# DLL Distribution: Discussion

---

- Close the library source code
- Save resource (Memory, HDD)
- If they want to modify the library source, then just redistribute a new DLL (**theoretically!!!**).
  - Really???
  - We will spend next several slide that the above statement is not easy to achieve.

# Problems with DLL Distribution

---

- Case 1: Problem caused by different compilers
  - Problem arises when the component developer and application developer use two **different C++ compilers**.
- Case 2: Problem with a single compiler
  - When the component developer and application developer use the **same C++ compilers**.
  - Problem arises when the component developer **revised the internal implementation detail**, leaving the interface unchanged.
  - C++ encapsulation does not guarantee binary compatibility.

# Case 1: Problem by different compilers

---

- Different C++ compilers use different **name mangling techniques**.
- A solution to the name mangle conflict
  - Using Module Definition File (DEF)
    - DEF allow exported symbols to be aliased.
  - But, this is tedious.

# Problem by different compilers

---

- More Problems with different compilers
  - ISO/ANSI C++ Draft Wording Paper (DWP) does not provide **standardization for binary runtime model of C++**.
  - There are more obstacles for binary compatibility.
  - Proprietary ways to implement objects.
- Example
  - C++ exception in MS compiler is not compatible with Watcom compiler.

# Name Mangling

---

- To allow C++ overloading, compilers mangle the symbolic name of each entry point.
- Name mangle for **global** function is disabled using **extern "C"** keyword.
- But, you cannot declare a **member function** with **extern "C"**.

# Name Mangling - Example

---

- Function prototype

1. `__declspec(dllexport) int fnwmangle(void);`
2. `__declspec(dllexport) int fnwmangle(int); //function overloading`
3. `extern "C" __declspec(dllexport) int fnwomangle(void);`

- MAP file

|               |                   |            |                |
|---------------|-------------------|------------|----------------|
| 0001:000000f0 | ?fnwmagle@@YAHXZ  | 100010f0 f | FastString.obj |
| 0001:00000120 | ?fnwmagle@@YAHH@Z | 10001120 f | FastString.obj |
| 0001:00000150 | _fnwomangle       | 10001150 f | FastString.obj |

# Problem by different compilers: Conclusion

---

- The lack of a C++ binary standard limits compatibility across DLL boundaries.
- Simply exporting C++ member function from DLLs is not enough to create vendor-independent component.

# Case 2: Problem with a single compiler

---

- Assume that the library developers and application developers use the same developing environment.
- C++ encapsulation is not enough to guarantee binary compatibility.
- Note
  - **Encapsulation** guarantees that the clients cannot access implementation details, i.e., private data members.

# Problem with a single compiler - Example

---

```
class __declspec(dllexport) FastString {
private:
    int m_cch; // new line for version 2.0
    char * m_psz;

public:
    FastString(const char *str);
    ~FastString(void);
    int Length(void) const;
    int Find(const char* psubstr) const;
};
```

# Calling Member Function

- Steps for member function call
  1. Push input arguments to stack
  2. Push object pointer to stack
  3. call Member function
  4. reset stack pointer
- object model for version 1

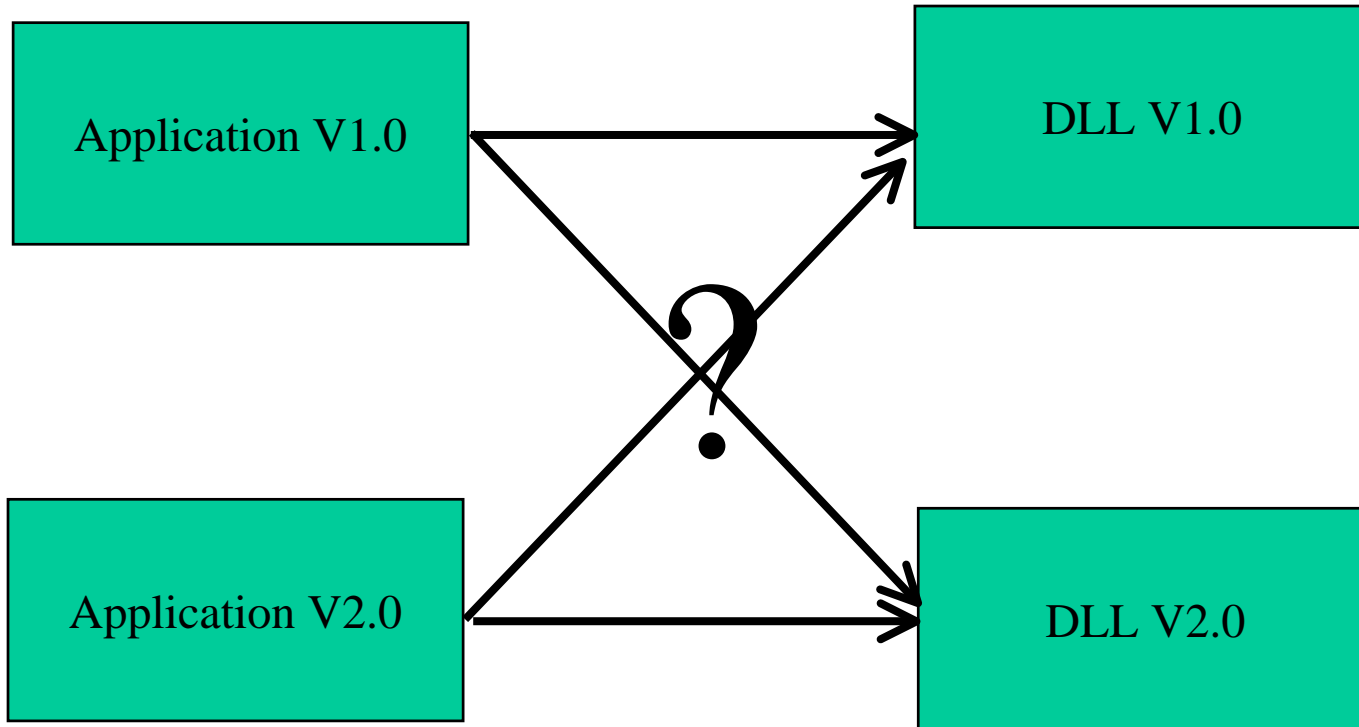
| offset | data member |
|--------|-------------|
| 0-3    | m_psz       |

- Object model for version 2

| offset | data member |
|--------|-------------|
| 0-3    | m_cch       |
| 4-7    | m_psz       |

# Problems

---



# Problem with a single compiler:Discussion

---

- C++ supports syntactic encapsulation, but no support for binary encapsulation.
- A solution to this
  - Add version number to the DLL file
    - Ex) FastString10.dll, FastString20.dll
  - MFC adopts this solution
  - Unnecessary files will be accumulated.

# Here Comes COM

---

- If they want to modify the library source, then just redistribute a new DLL (**theoretically!!!**).
- As explained, there is so many obstacle to achieve the previous statement.
- But there is a way, Component Object Model (COM)

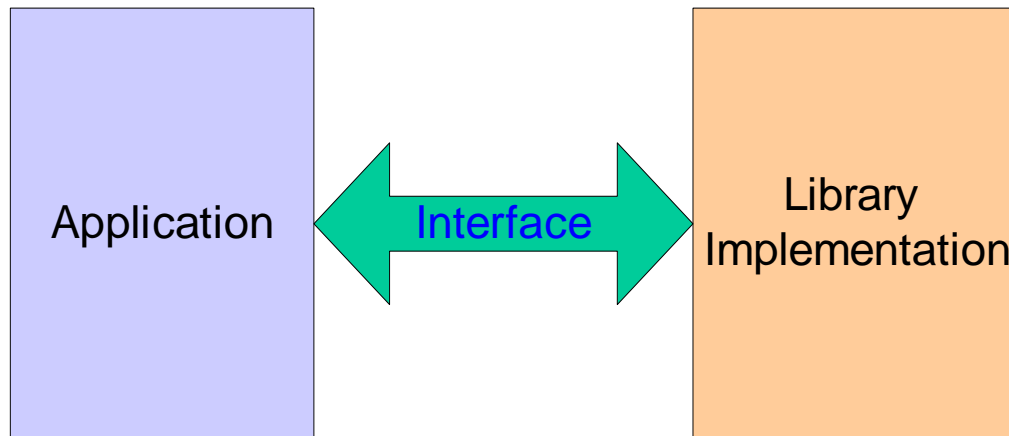
# Component Object Model

---

- Separate interface from implementation
- Runtime Polymorphism
- Object extensibility
- Resource management

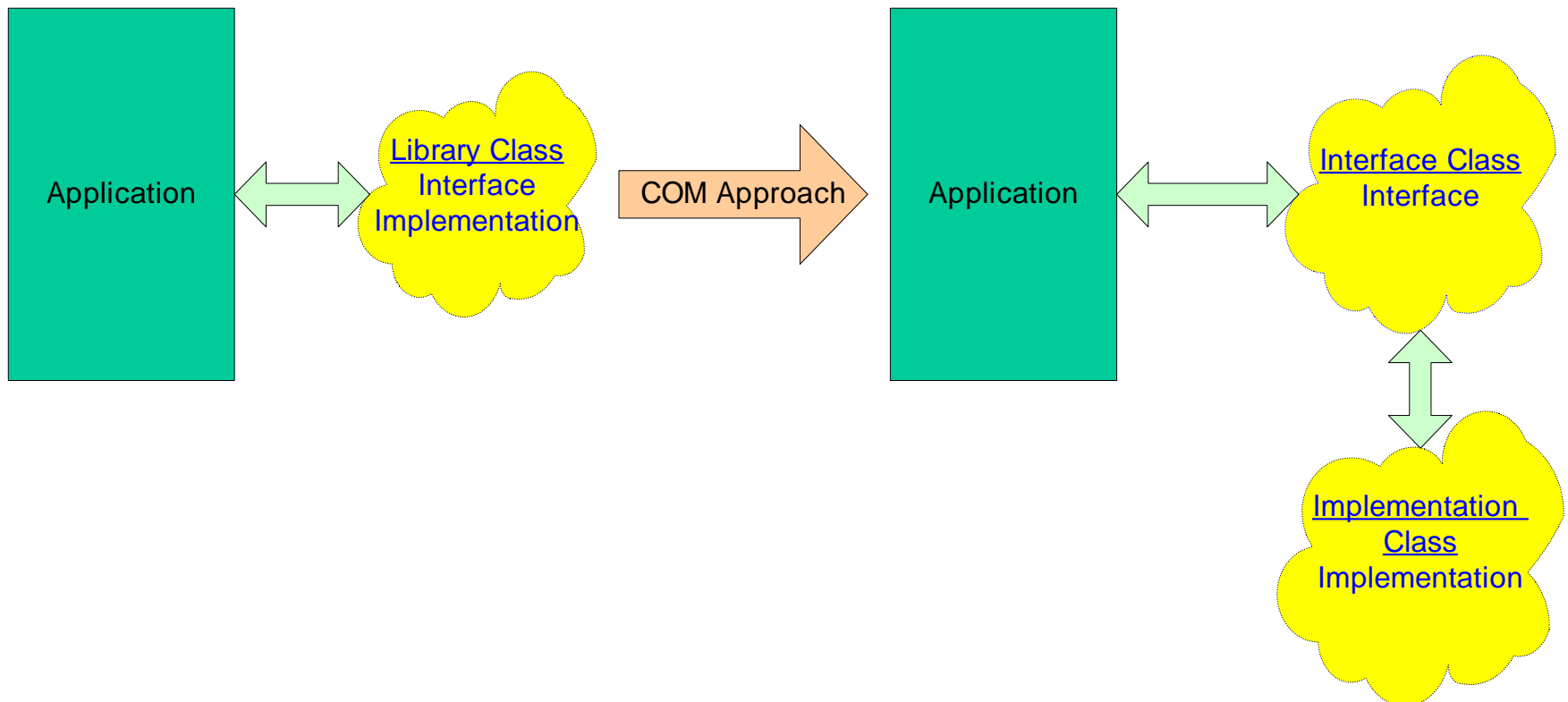
# Separating Interface from Implementation

- Information hiding as well as encapsulation principle of C++ enable separation of “**interface**” and “**implementation**”
  - The problem is that these principles does not apply at a binary level.
  - In general, C++ class is both interface and implementation simultaneously.



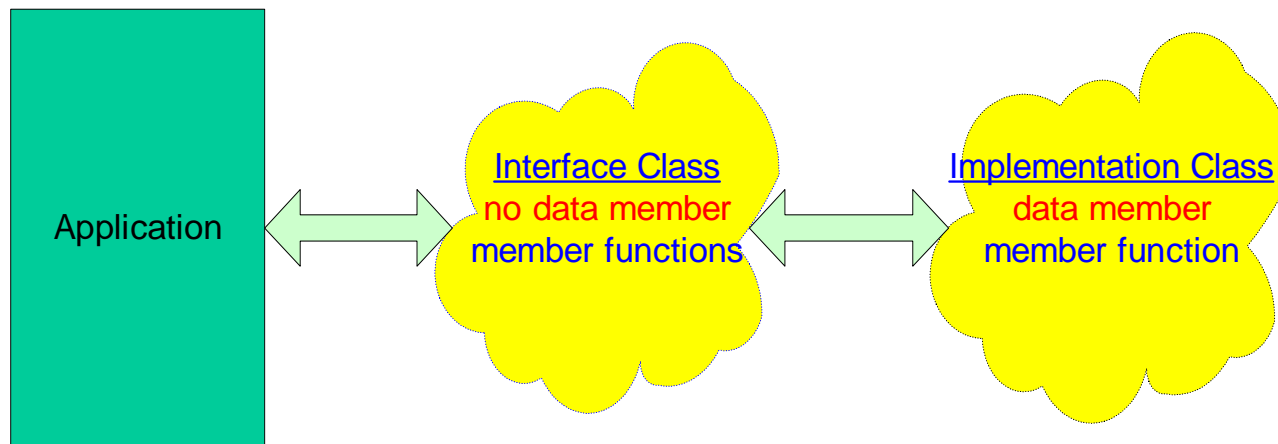
# Solution for binary encapsulation

- Let's make two classes
  - **Interface class** which represents (binary) interface to a data type
  - **Implementation class** which represents the actual implementation of the data type



# Solution for binary encapsulation

- Interface class should not contain any data member, only member functions.
- Implementation class contain data member for implementation.
- There two ways
  - Use Handle class as the interface
  - Use abstract base class as the interface

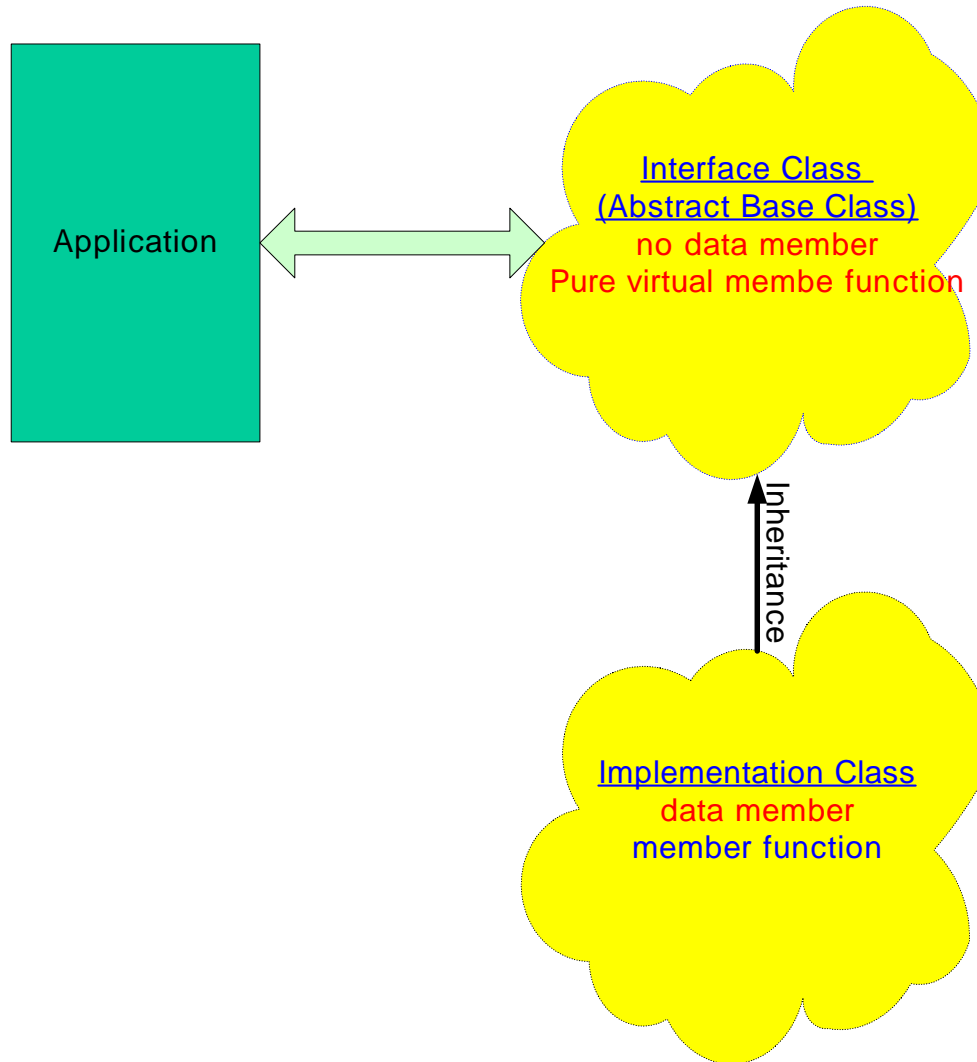


# Handle class as the interface

---

- Some people still use this method  
example: W3 implementation of html parser
- If there are many public member functions, using this is tedious and error prone

# Abstract Base Class as Binary Interface

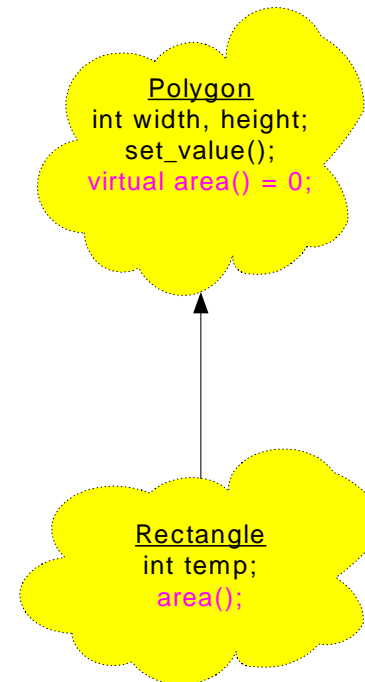


# Implementation of Virtual Function

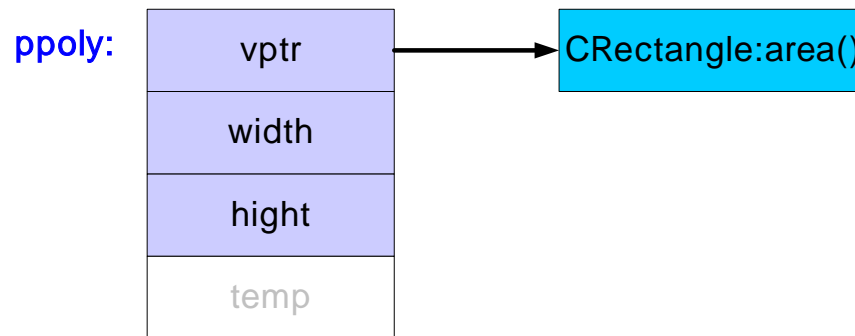
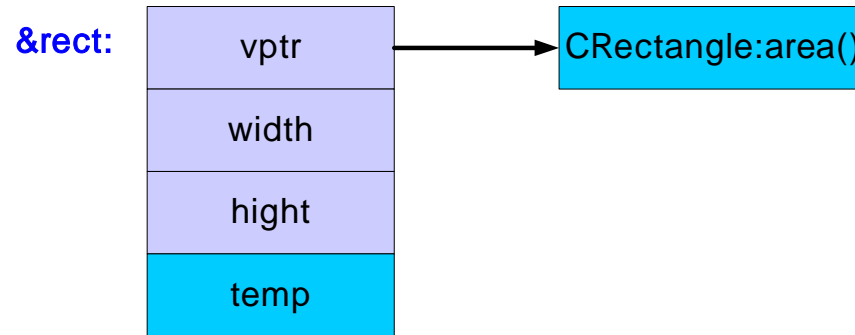
```
class CPolygon {
protected:
    int width, height;
public:
    void set_values (int a, int b)
        { width=a; height=b; }
    virtual int area (void) =0;
};

class CRectangle: public CPolygon {
public:
    int temp;
    int area (void)
        { return (width * height); }
};

void main () {
    CRectangle rect;
    CPolygon * ppoly = &rect;
    ppoly->set_values (4,5);
    ppoly->area();
}
```



# Implementation of Virtual Function



# Abstract Base Class as Binary Interface

---

- Class with pure virtual member function.

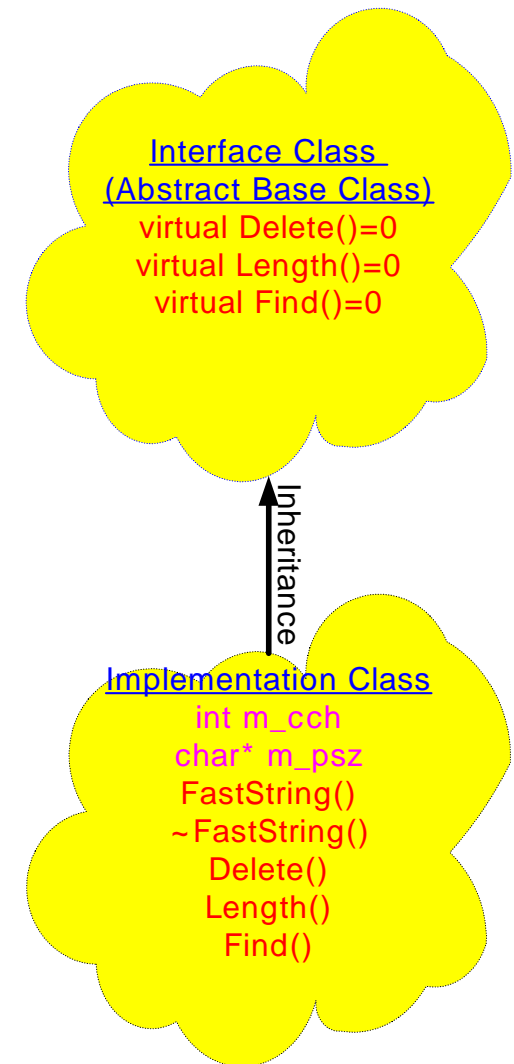
- Interface Class:

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

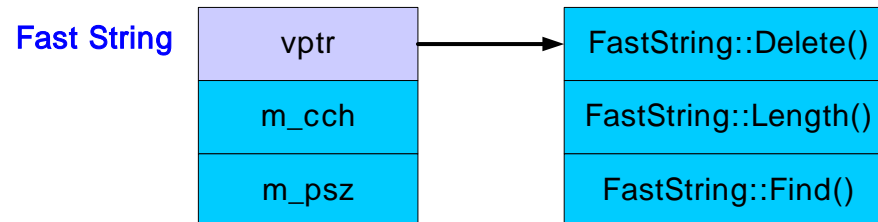
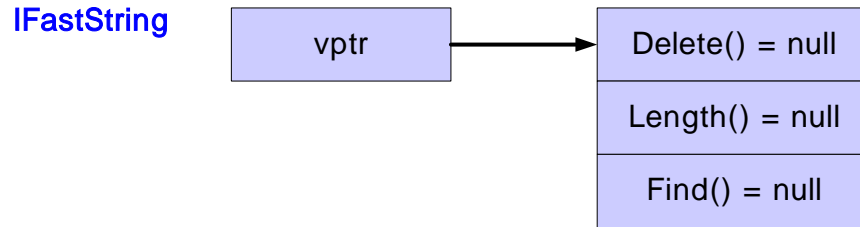
# Implementation Class

- Implementation of the Library

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



# FastString Using Abstract Base as Interface



# Object Creation

---

- DLL exports a global function that call the constructor and return object pointer

`extern "C"`

```
I FastString *CreateFastString(const char *psz) {  
    return new FastString(psz);  
}
```

- This is the **only function that the DLL exports**
- By using `extern "C"`, we can **overcome name mangling** (multi-compiler dependency).

# Object Destruction

---

- Problem
  - The client wants to destroy the implementation class object.
  - But the client only have the pointer to the interface class object, I.e., base class.
  - using "delete" with the pointer, the results are unexpected.
- Solution
  - Add "Delete" virtual member function to the interface.
  - Implement the "Delete" member function in the implementation class.

# Object Destruction - Example

---

```
class IFastString {
public:
    virtual void Delete(void) = 0;
    ...
};
class FastString : public IFastString {
    ~FastString(void);
    void Delete(void);
    ...
};

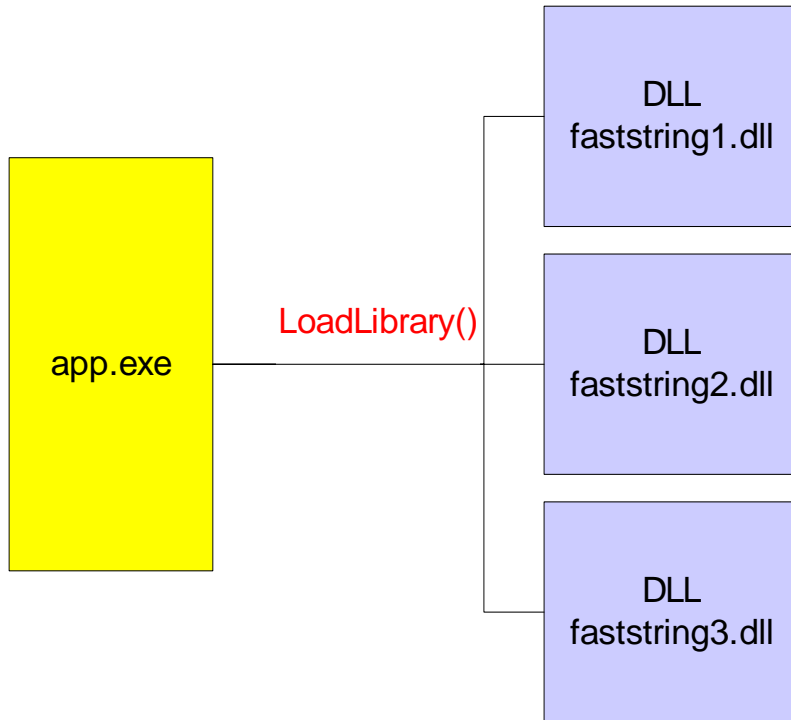
void FastString::Delete(void) {
    delete this;
}

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

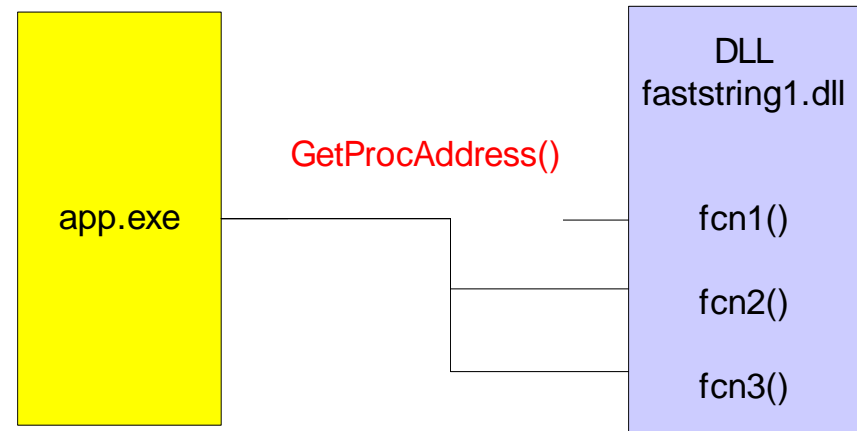
# Runtime Polymorphism

- During runtime, the client can
  - Select a DLL: `LoadLibrary()`
  - Select a function: `GetProcAddress()`

Application can select DLL



Application can select function

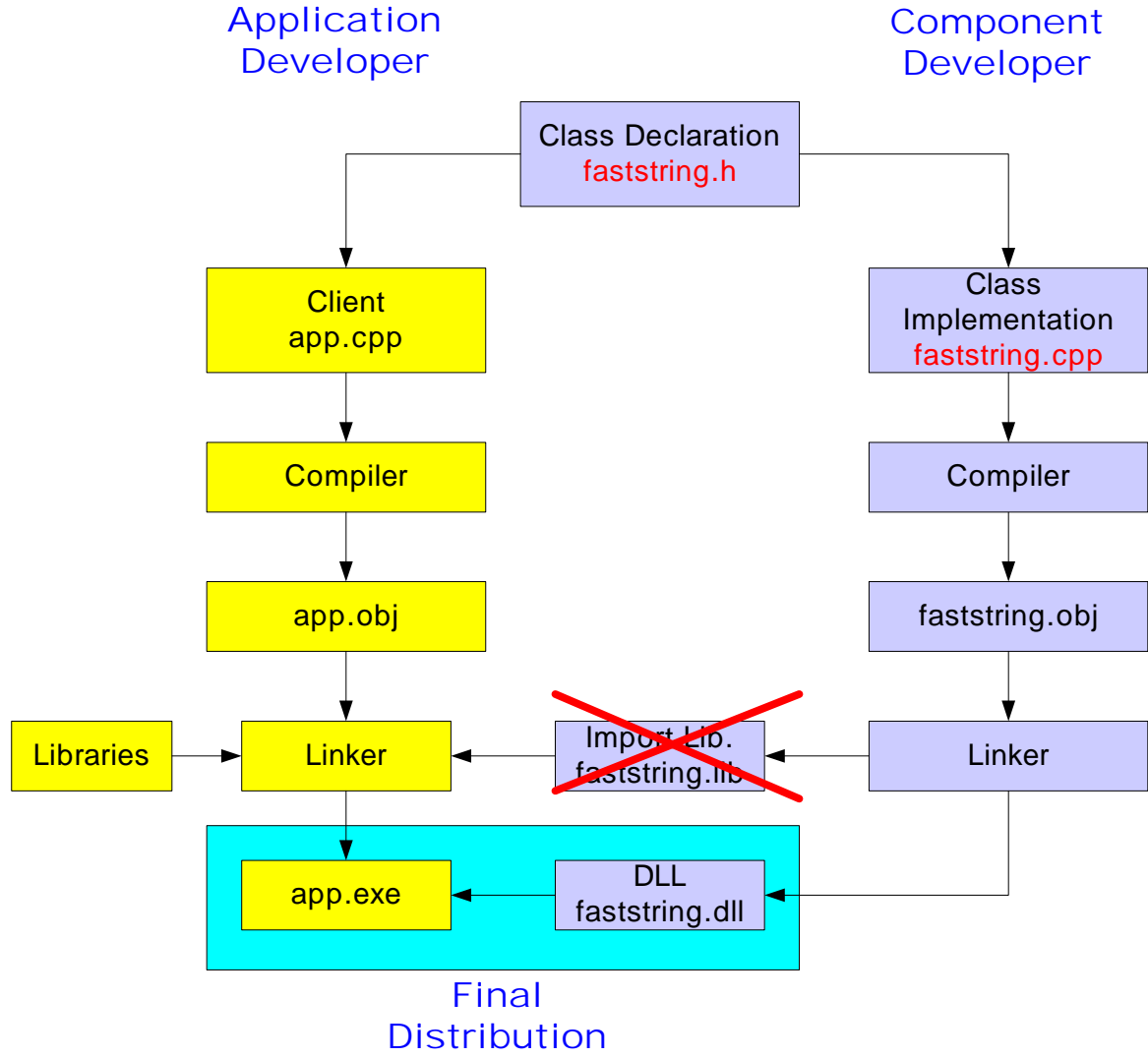


# Runtime Polymorphism: Merits

---

- The client can run without the DLL.
- The DLL is loaded only when they are called.
- No need for the DLL's import library.
  
- Client can select “**dynamically**” (during runtime) between various implementations of the same interface.
  
- Example
  - word searching from right to left or left to right.

# Runtime Polymorphism



# Object Extensibility

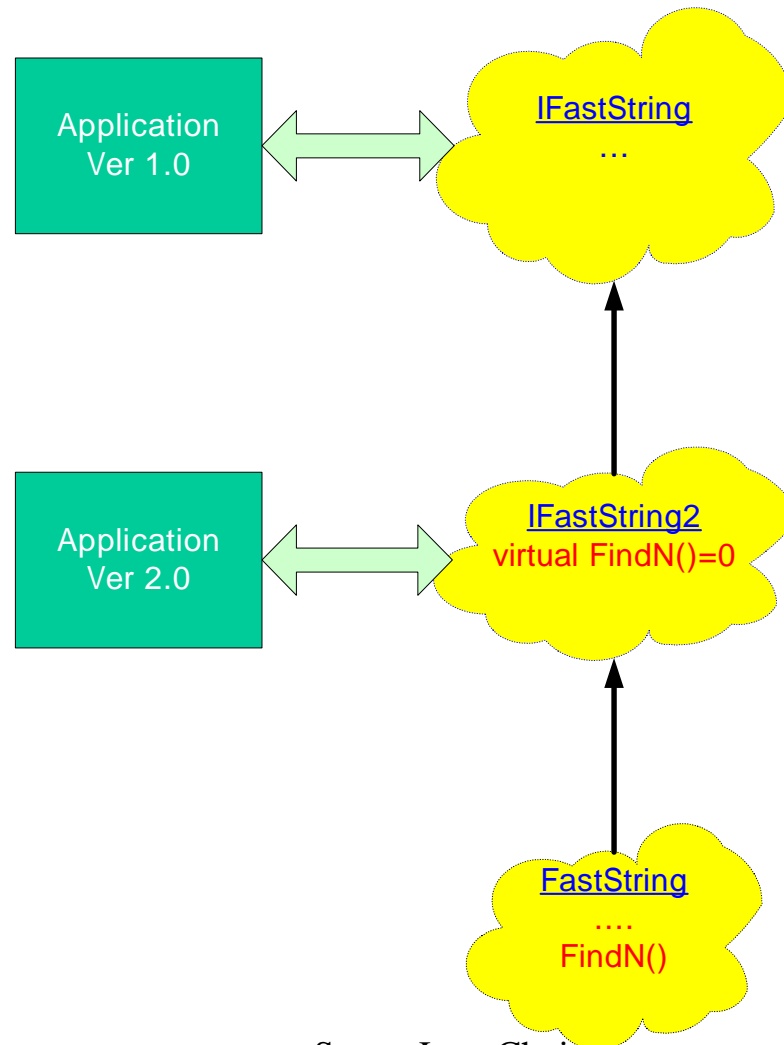
---

- Sometimes, we need to **change the interface definition** after the complete implementation.
- How do we build a DLL that supports both the old and new clients.
- Only extending the old interface is possible.
- Two scenarios for the object extending.
  - Extending existing versions of an interface.
  - Exposing multiple unrelated interfaces from a single object.

# Extending existing versions of an interface

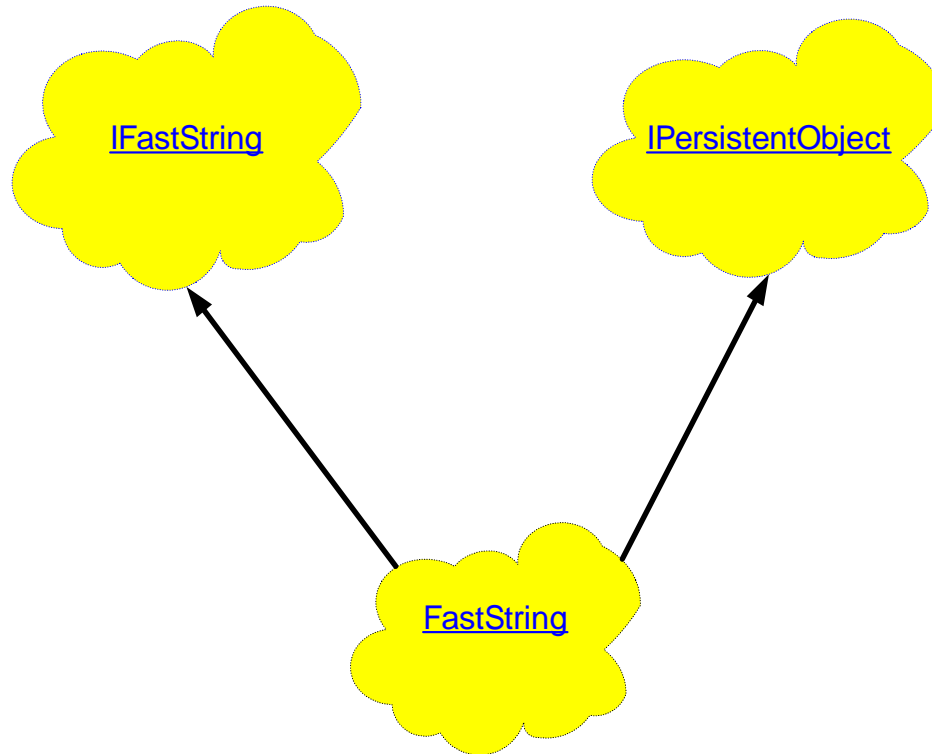
```
I FastString *pfs = CreateFastString("hi bob");
```

```
I FastString2 *pfs2 = dynamic_cast<I FastString2*> (pfs);
```



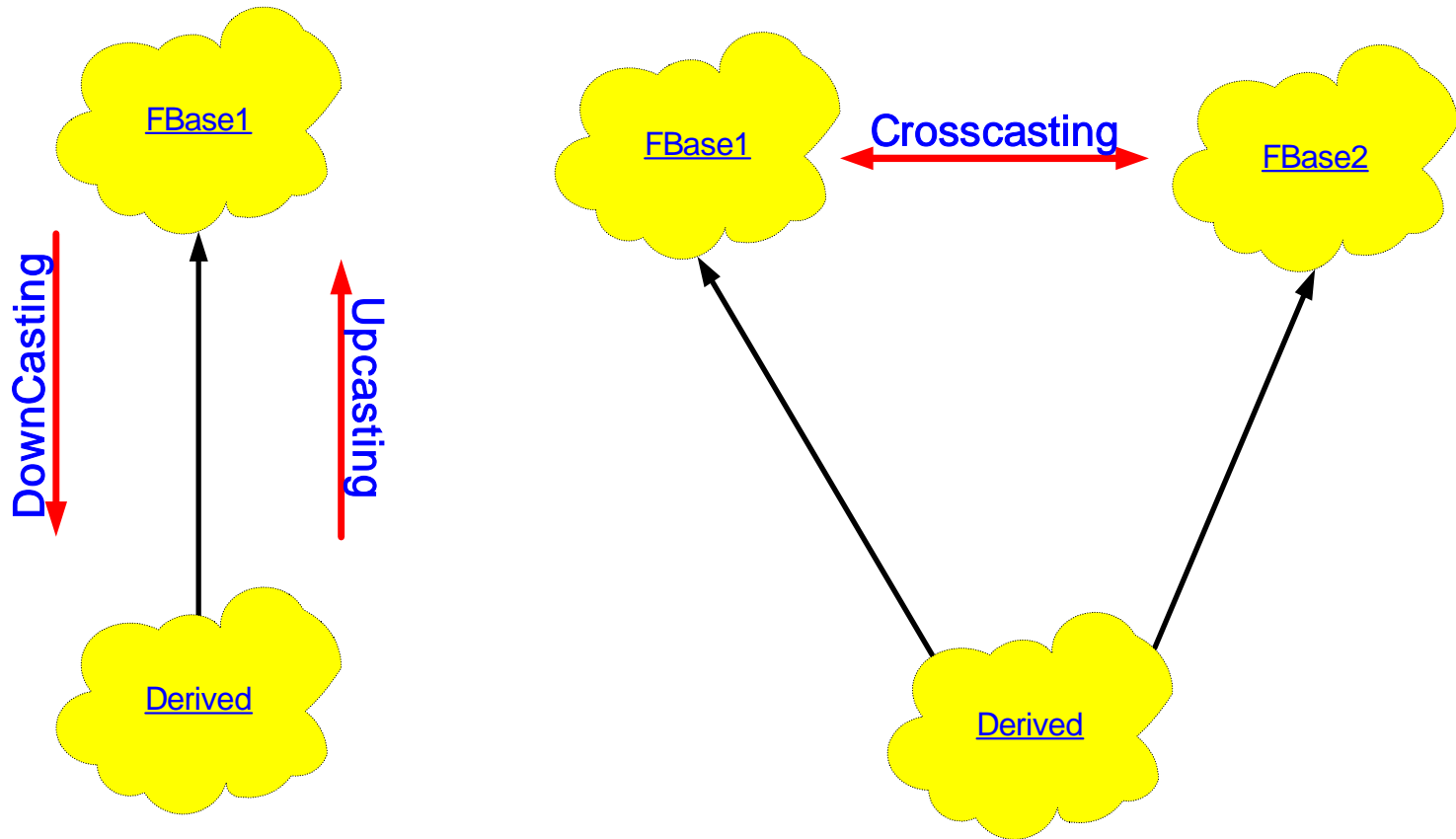
# Extending existing versions of an interface

```
I FastString *pfs = CreateFastString("hi bob");  
I PersistentObject *pfs2 = dynamic_cast<I PersistentObject*> (pfs);
```



# Casting between Classes

- DownCasting; Upcasting; Crosscasting



# Casting between Classes

- Runtime Type Identification (RTTI)
  - Ensure casting supported by an object at runtime
  - return null if fails
  - `dynamic_cast <T *> (P) P → T*`



# 4. Summary or Conclusion

---

- What are the problems with componentware?
  1. Problems with different C++ compilers
  2. Name Mangling
  3. Problems with different versions of DLL.
- How do we solve the problems? Answer: COM
  1. Separation between interface and implementation:  
Abstract class and virtual member function.
  2. run-time polymorphism
- Basic functions of COM
  1. object creation and deletion
  2. object extention