Çok İşlev, Çok Biçimlilik (Polymorphism)

Dr. Metin Özkan

Giriş

- Nesne tabanlı programlama teknolojileri şunları kapsar:
  - *Kapsülleme (Encapsulation) (Class)*
    - *Veri soyutlama (Data abstraction), bilgi saklama (information hiding)*
  - *Miras (Inheritance)*
    - *Tekrar kullanılabilirlik (Reusability)*
  - *Çok İşlev (Polymorphism)*
    - *Dinamik bağlama (Dynamic binding) olarak adlandırılan özel bir mekanizma ile bir fonksiyon ismine birden çok anlam bağdaştırma yeteneğidir.*
    - *Özel program "program in specific" yerine, genel program "program in general" yazmayı mümkün kılar.*

Statik ve Dinamik Bağlama (Static binding v.s. Dynamic binding)

- Programlama dillerinde, derleme ve çalışma zamanları iki ayrı süreçtir.
  - *Derleme, program söz diziminin (kodların) kontrol edilip, makine diline çevrildiği süreçtir.*
  - *Çalışma, makine diline çevrilmiş komutların çalıştırılma sürecidir.*
- Derleyici, program içerisinde hangi fonksiyonun çağrılacağını derleme zamanında belirlerse buna statik bağlama (static binding) denir.
  - *Böylece, program çalışmadan önce çağrılacak fonksiyonlar belirlenip, kod içine gömülmüş olur.*
- Program içinde, yaratılan nesneler çalışma zamanında farklılaşırsa, çalışma zamanından önce ilişkilendirme yapılmaması gerekir. Programcı derleyiciye hangi fonksiyonun çalışma zamanında bağlanacağını belirtebilir. Çalışma anında yapılan ilişkilendirmeye dinamik bağlama (dynamic binding) denir.

Çok İşlev (Polymorphism)

- Çok işlev (Polymorphism) yapısı, miras (inheritance) hiyerarşisini kullanır.

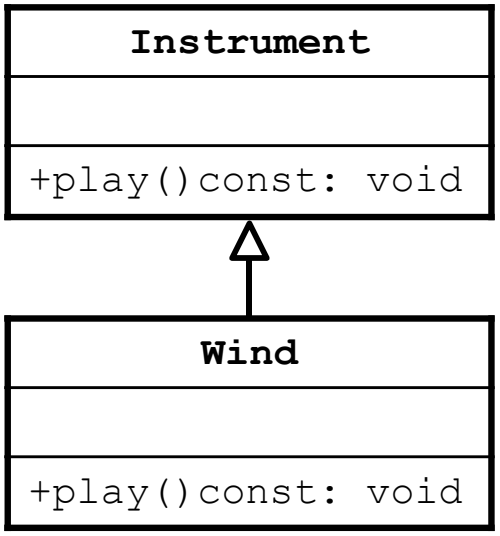
- Bu yapı, hiyerarşideki tüm sınıfların nesnelerinin, sanki taban sınıfının (base class) nesneleri gibi işlem görmesini sağlar.

- Her bir nesne, nesne tipinin gerektirdiği doğru görevi yerine getirir.
  - *Nesne tipine bağlı olarak, farklı eylemler gerçekleşir.*

- Hiyerarşiyi kullanan kodlarda az seviyede ya da hiç değişiklik yapmadan yeni sınıfların eklenmesi mümkün olur.

Çok İşlevsiz üye fonksiyon erişimi (Non-polymorphic member function access)

- Nesneler, taban sınıfının (base class) adresi üzerinden idare edilebilir.
- Bir nesnenin adresini (bir gösterge ya da referans) alıp, taban sınıfının adresi gibi muamele etmek yukarı çevirme (*upcasting*) denir.

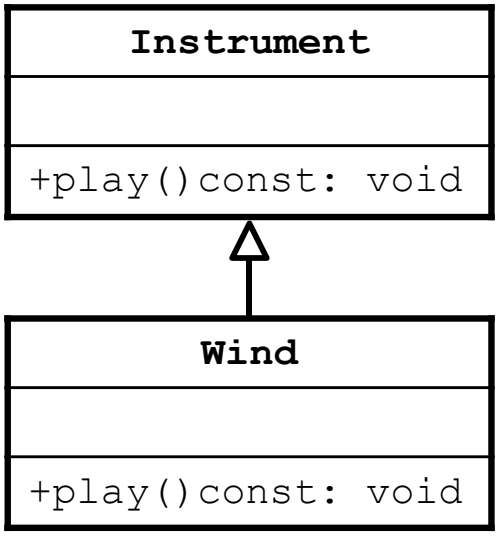



```
#include <iostream>
using namespace std;
class Instrument{
public:
      void play() const{
            cout<<"Instrument::play" << endl;
      }
};
class Wind : public Instrument {
public:
      void play() const{
             cout << "Wind::play" << endl;
      }
};
void tune(Instrument& i) {
            // ...
            i.play();
}
void tune(Instrument* i) {
            // ...
            i->play();
}

int main() {
            Wind flute;
            tune(flute); // Upcasting
            tune(&flute); // Upcasting
            return 0;
}
```

Çok İşlevsiz üye fonksiyon erişimi (Non-polymorphic member function access)

- `tune(flute);` yada `tune(&flute);` komutları çalıştığında, flute Wind tipinde bir nesne olmasına rağmen, tune fonksiyonunun parametresi olarak Instrument tipinde referans ya da göstergeye dönüşmektedir.
- Referans ya da gösterge Instrument tipinde olduğu için, flute nesnesinin sadece Instrument sınıfından miras aldığı üyelere erişilebilmektedir.




```
#include <iostream>
using namespace std;
class Instrument{
public:
      void play() const{
            cout<<"Instrument::play" << endl;
      }
};
class Wind : public Instrument {
public:
      void play() const{
             cout << "Wind::play" << endl;
      }
};
void tune(Instrument& i) {
            // ...
            i.play();
}
void tune(Instrument* i) {
            // ...
            i->play();
}

int main() {
            Wind flute;
            tune(flute); // Upcasting
            tune(&flute); // Upcasting
            return 0;
}
```

Çok İşlevsiz üye fonksiyon erişimi (Non-polymorphic member function access)

- Derleyici, gösterge/referans (pointer/reference) içeriğini dikkate almaz, gösterge/referans (pointer/reference) tipiyle eşleşen üye fonksiyonlara çağrı gerçekleştirir.
- Problem:
  - *Program çalıştırıldığında, ekrana çıktı olarak **Instrument::play** yazdırıldığı görünür.*
  - *Halbuki, bu beklenen çıktı değildir. Instrument tipindeki gösterge ya da referansın bir Wind türünde nesneyi gösterdiğini biliyorsunuz ve ekrana da wind sınıfındaki play fonksiyonu çağrılarak **Wind::play** yazdırılması beklenildi.*

Çok İşlevli üye fonksiyon erişimi (Polymorphic member function access)

Çözüm fonksiyon başında virtual anahtar kelimesini eklemek

- Bu durumda, çıktı beklendiği gibi **`Wind::play`** olacaktır.

```
#include <iostream>
using namespace std;
class Instrument{
public:
      virtual void play() const{
            cout<<"Instrument::play" << endl;
      }
};
class Wind : public Instrument {
public:
      void play() const{
            cout << "Wind::play" << endl;
      }
};
void tune(Instrument& i) {
            // ...
            i.play();
}
void tune(Instrument* i) {
            // ...
            i->play();
}

int main() {
            Wind flute;
            tune(flute); // Upcasting
            tune(&flute); // Upcasting
            return 0;
}
```

Dinamik Bağlama (Late Binding) (Virtual Functions)

- Belirli bir fonksiyon için dinamik bağlama gerçekleştirebilmek için, taban sınıfında o fonksiyonu beyan ederken **virtual** anahtar kelimesi kullanılmalıdır.
- Dinamik bağlama (Late binding), sadece sanal (**virtual)** fonksiyonlar için ve taban sınıfın adresi kullanılıyorsa gerçekleşir.
- Hangi sınıfın fonksiyonu çağrılacak
  - *Normalde*
    - *Gösterge/referans hangi sınıfın fonksiyonu çağrılacağını belirler.*
  - *Sanal (**virtual**) fonksiyonlarla*
    - *Gösterge/referans tipi değil, gösterilen nesnenin tipi, sanal fonksiyonun hangi versiyonunun çağrılacağını belirler.*
    - *Çalışma anında, dinamik olarak hangi fonksiyonun kullanılacağının program tarafından belirlenmesine imkan sağlar.*

      Bu durum, dinamik bağlama (dynamic binding) ya da geç bağlama (late binding) olarak adlandırılır.

Dinamik Bağlama (Late Binding) (Virtual Functions)

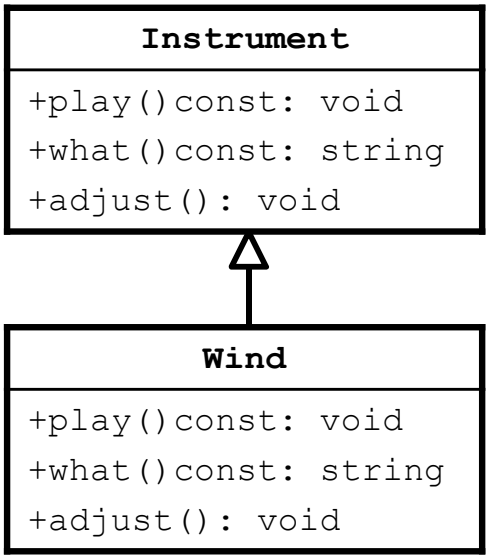
- Taban sınıfta virtual fonksiyon tanımlamaya ihtiyaç vardır.

- Türetilen sınıflarda da fonksiyon beyanında virtual kelimesi kullanılabilir. Bu hata değildir, ama gereksizdir.

Dinamik Bağlama (Late Binding) (Virtual Functions)

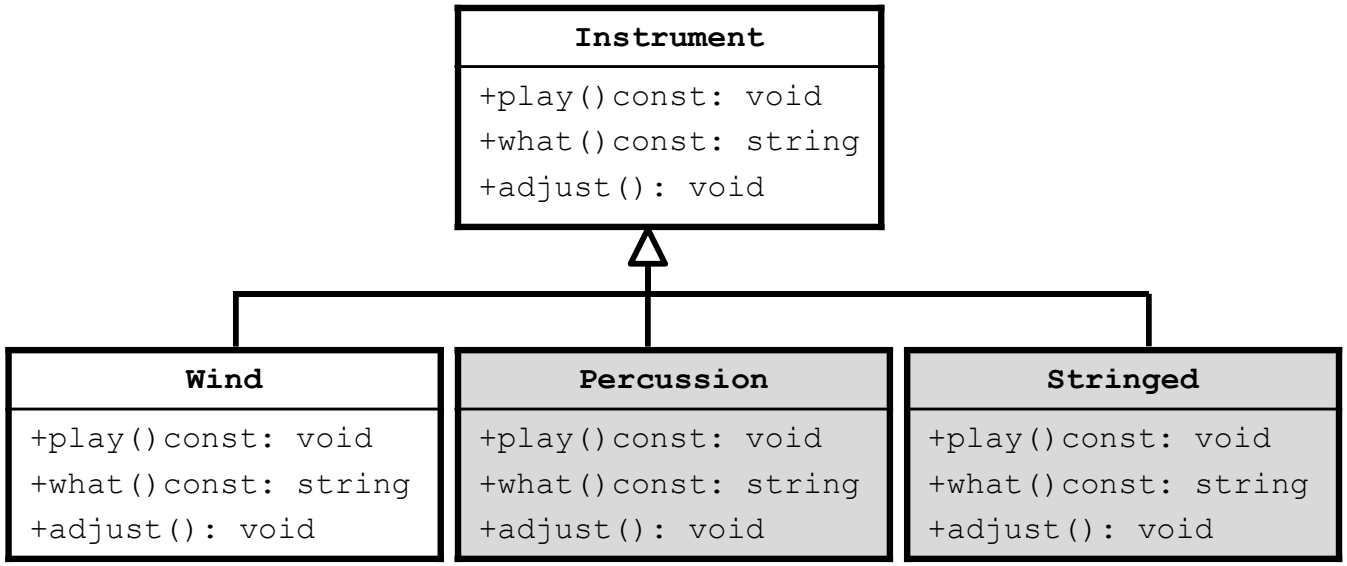
- Genişleyebilirdik
  - *Taban sınıfta (base class), **play**( ) fonksiyonunun sanal(**virtual**) olarak tanımlanması ile birlikte, bu enstrümanları kullanan **tune**( ) fonksiyonunu değiştirmeksizin yeni enstrümanlar taban sınıf (base class) altına eklenebilir ve kullanılabilir.*
  - *İyi tasarlanmış bir nesne tabanlı programda, birçok ya da bütün fonksiyonlar **tune**( ) fonksiyon modelini izler ve sadece taban sınıf (base class) ara yüzü ile haberleşir.*
  - *Böyle bir program, genişleyebilir **(extensible)** formdadır. Çünkü, yeni veri tiplerini ortak taban sınıfından türeterek, programa yeni fonksiyonlar kazandırılabilir.*

Dinamik Bağlama (Late Binding) (Virtual Functions)



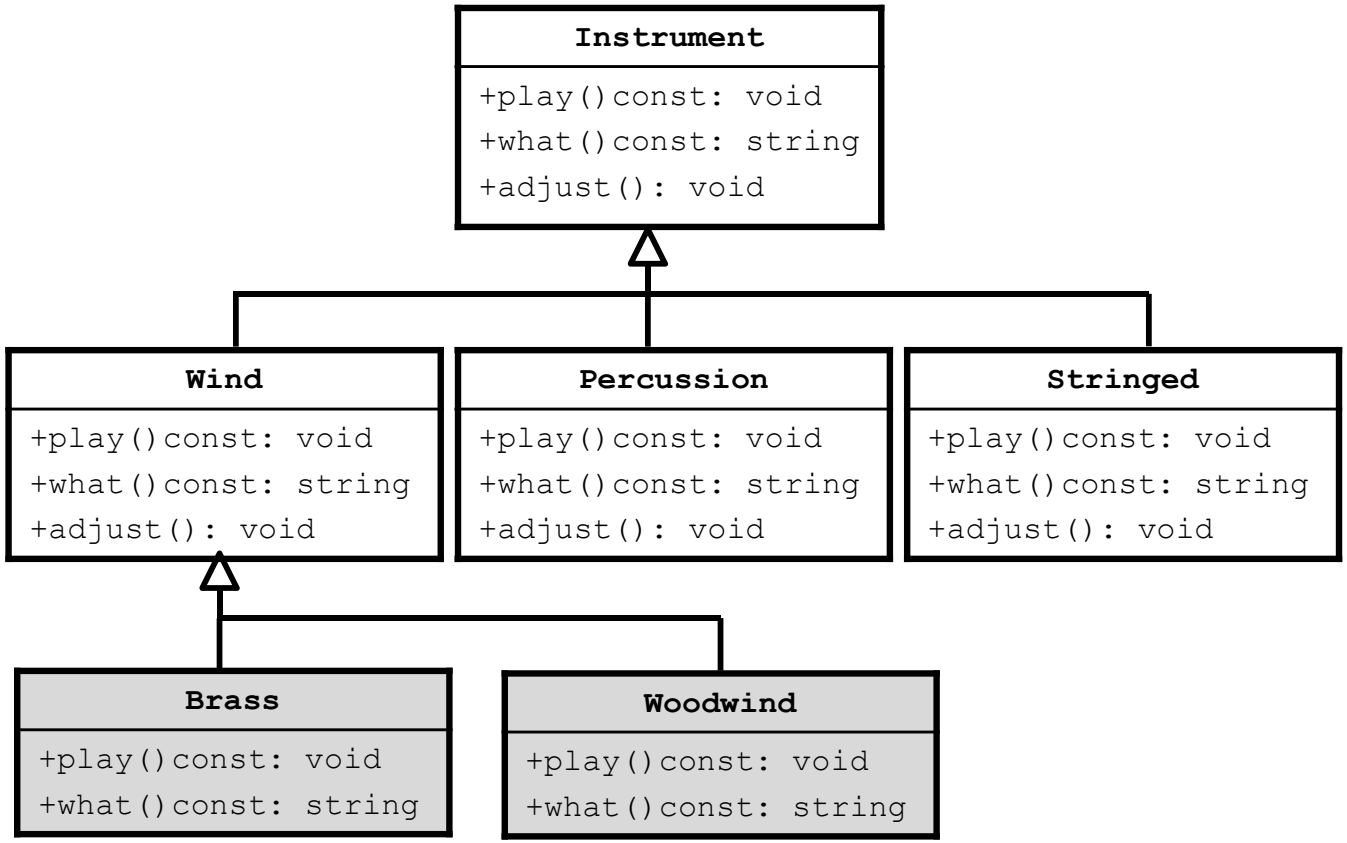

```
#include <iostream>
#include <string>
using namespace std;
class Instrument{
public:
      virtual void play() const{
            cout<<"Instrument::play" << endl;
      }
      virtual string what() const{
            return "Instrument";
      }
      virtual void adjust(){
            // Assume this will modify
            // the object
      }
};
class Wind : public Instrument {
public:
      void play() const{
             cout << "Wind::play" << endl;
      }
      string what() const{
            return "Wind";
      }
      void adjust(){
            // Assume this will modify
            // the object
      }
};
```

Dinamik Bağlama (Late Binding) (Virtual Functions)



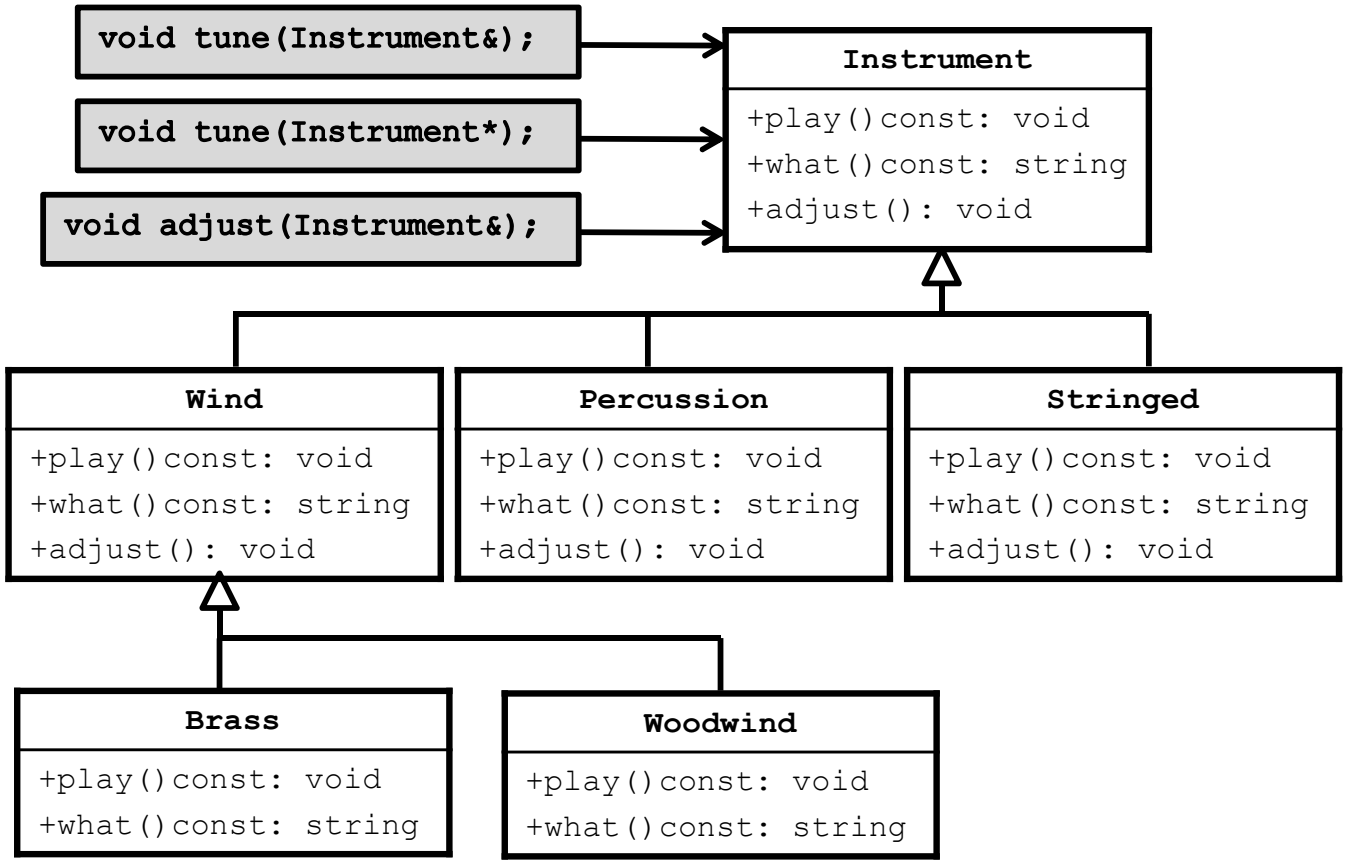

```
class Percussion : public Instrument
{
public:
      void play() const{
            cout<<"Percussion::play"
                << endl;
      }
      string what() const{
            return " Percussion ";
      }
      void adjust(){
            // Assume this will modify
            // the object
      }
};
class Stringed : public Instrument {
public:
      void play() const{
             cout<< "Stringed::play"
                 << endl;
      }
      string what() const{
            return " Stringed ";
      }
      void adjust(){
            // Assume this will modify
            // the object
      }
};
```

Dinamik Bağlama (Late Binding) (Virtual Functions)



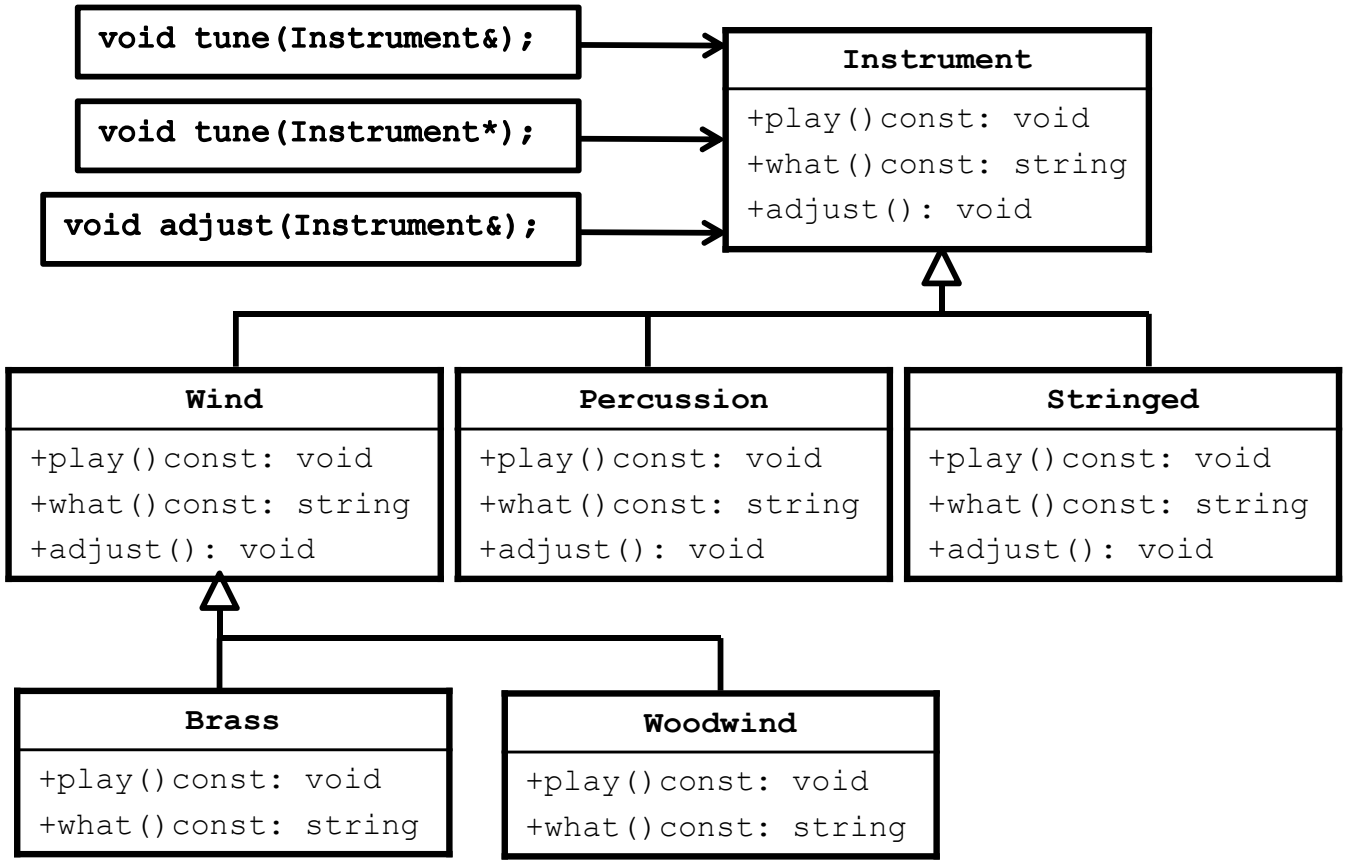

```
class Brass : public Wind {
public:
      void play() const{
            cout<<"Brass::play"
                << endl;
      }
      string what() const{
            return " Brass ";
      }
};
class Woodwind : public Wind {
public:
      void play() const{
             cout<< "Woodwind::play"
                 << endl;
      }
      string what() const{
            return " Woodwind ";
      }
};
```

Dinamik Bağlama (Late Binding) (Virtual Functions)




```
//Identical function from before:
void tune(Instrument& i) {
          // ...
          i.play();
}
void tune(Instrument* i) {
          // ...
          i->play();
}
//New function:
void adjust(Instrument& i){
          i.adjust();
}
//Upcasting for array initialization
Instrument* orchestra[]={
          new Wind,
          new Percussion,
          new Stringed,
          new Brass
};
```

Dinamik Bağlama (Late Binding) (Virtual Functions)




```
int main() {
          Wind flute;
          Percussion drum;
          Stringed violin;
          Brass flugelhorn;
          Woodwind recorder;

          tune(flute);
          tune(drum);
          tune(violin);
          tune(flugelhorn);
          tune(recorder);
          adjust(flugelhorn);
          return 0;
}
```

Çıktı:

```
Wind:play
Percussion::play
Stringed::play
Brass::play
Woodwind::play
```

Dinamik Bağlama (Late Binding) (Virtual Functions)

- Görüldüğü gibi bir başka miras hiyerarşisi (inheritance) **Wind** sınıfı altına eklendi. Ancak, sanal **(virtual)** mekanizma kaç seviye olduğuna bakmaksızın doğru bir şekilde çalıştı.

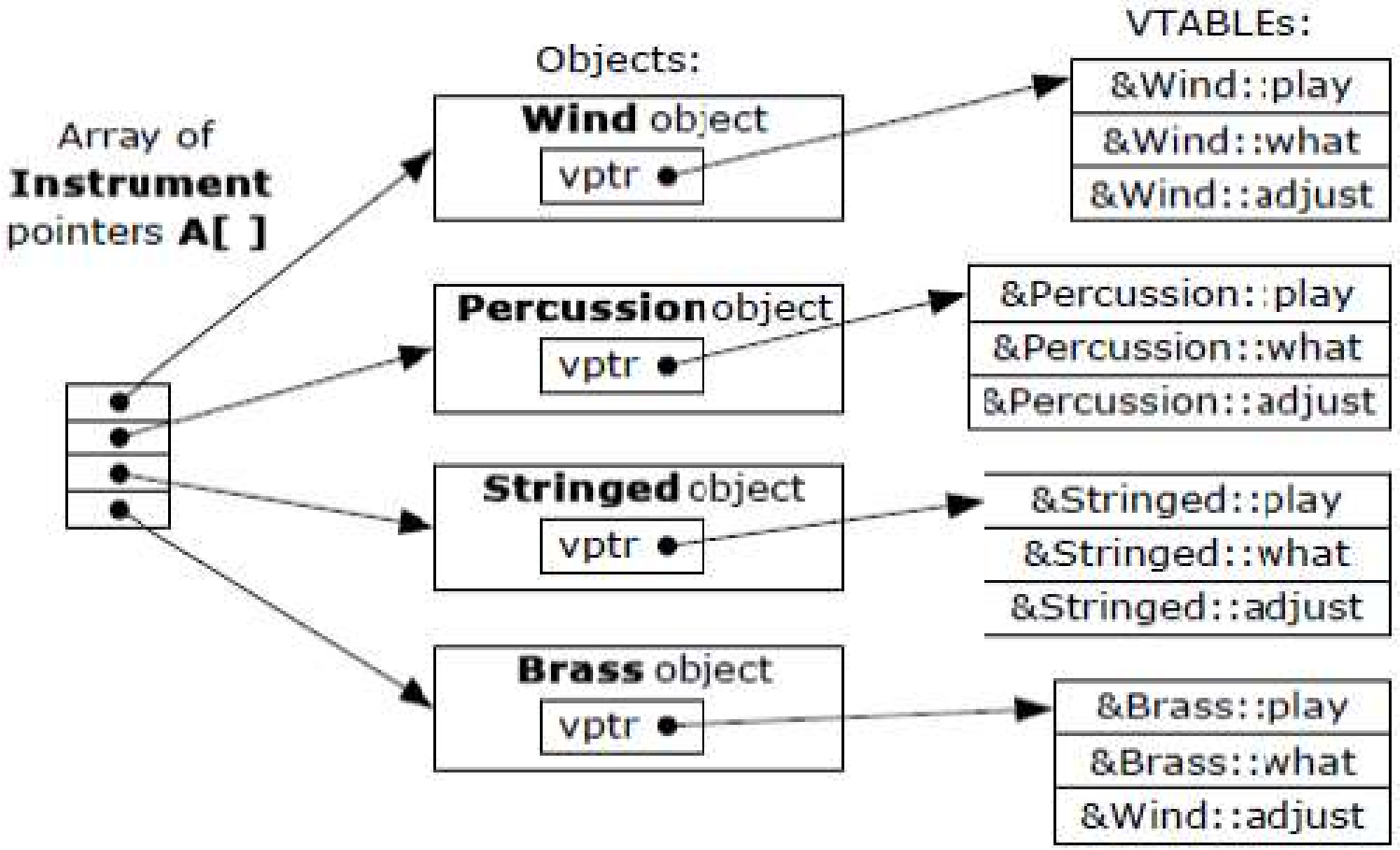
- **adjust( )** fonksiyonu, **Brass** sınıfı için geçersiz sayılmadı. Virtual fonksiyonu içermeyen bir sınıf için yukarı doğru hiyerarşide karşılaşılan ilk fonksiyon otomatik olarak kullanıldı. Derleyici, bir virtual fonksiyon için her zaman bir tanım olmasını garantiler. Bundan dolayı, hiçbir zaman bağlanmamış bir fonksiyon çağrısı ile karşılaşılmaz.

C++ dinamik bağlamayı (late binding) nasıl gerçekleştirir

- Bunu gerçekleştirmek üzere, derleyici virtual fonksiyonlar içeren her sınıf için bir tablo (VTABLE olarak adlandırılan) yaratır.
- Derleyici, virtual fonksiyonların adreslerini belirli sınıflar için VTABLE tablosuna yerleştirir.
- Virtual fonksiyon içeren her bir sınıfta, bu sınıftan türetilen nesne için VTABLE tablosunun yerini gösteren bir gösterge (*vpointer olarak adlandırılan,* VPTR olarak kısaltılan) yerleştirilir.
- Bir taban sınıf (Base class) göstergesi (pointer) üzerinden bir virtual fonksiyon çağrısı gerçekleştirildiğinde (çok işlev çağrısı, a polymorphic call), derleyici VPTR vasıtasıyla, VTABLE listesinde bulunan fonksiyon adresine bakar ve o adresteki fonksiyona çağrı gerçekleştirir. Böylece, dinamik bağlantı ile doğru fonksiyona çağrı gerçekleşir.

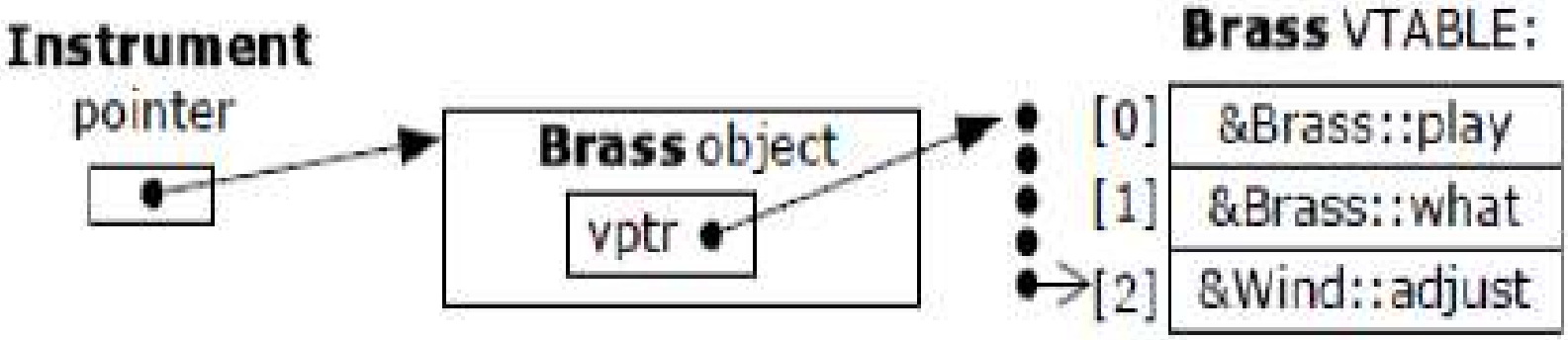
C++ dinamik bağlamayı (late binding) nasıl gerçekleştirir

- Burada, enstrüman örneği kullanılarak süreç görselleştirilmektedir.



C++ dinamik bağlamayı (late binding) nasıl gerçekleştirir

- Burada, bir **Brass** nesnesi için **adjust( )** çağrısının, **Instrument** göstergesi üzerinden nasıl göründüğü verilmektedir (Bir **Instrument** referans içinde aynı prosedür gerçekleşir.):



Soyut Taban Sınıfı (Abstract Base Class) ve Saf Sanal Fonksiyonlar (Pure Virtual Functions)

- Bir tasarımda sıklıkla, taban sınıfın (base class), sadece türetilen sınıflar (derived classes) için bir ara yüz oluşturması istenir.
- Öyle ki, taban sınıf tipinde bir nesne yaratılması istenmez.
  - *Örneğin, Instrument sınıfından bir nesne yaratıp, tune() fonksiyonunu çağırmanız çok anlamsız olacaktır. Bu fonksiyon, enstrümanla ne çalacaktır!*
- Bu durumda, taban sınıf soyut (*abstract)* yapılır. Bir sınıfın soyut olması için, en az bir tane fonksiyonun saf sanal fonksiyon (*pure virtual function)* olarak tanımlanması gerekir.

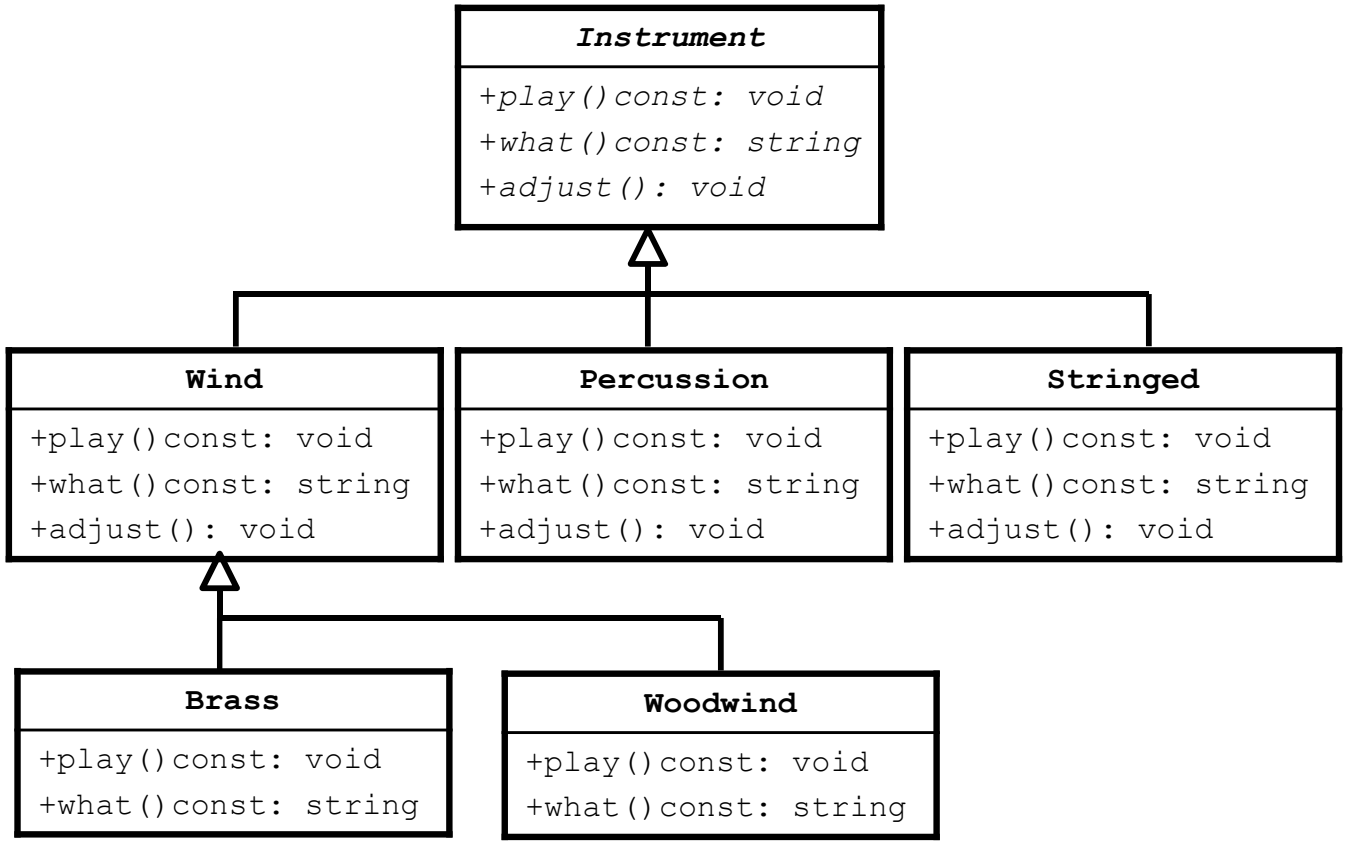
Soyut Taban Sınıfı (Abstract Base Class) ve Saf Sanal Fonksiyonlar (Pure Virtual Functions)

- Saf sanal fonksiyon (pure virtual function), başında **virtual** anahtar kelimesi bulunan ve tanımı verilmeden sıfıra eşitlenen fonksiyondur. (= **0.**)
  - *Örneğin, Instrument sınıfında,* `virtual void tune()=0;` *şeklinde tanımlanır.*
- Eğer birisi, soyut sınıftan (abstract class) bir nesne yaratmaya kalkarsa, derleyici hata verir.
- Bu, aynı zamanda belli bir tasarıma zorlamak için araçtır.

```
#include <iostream>
#include <string>
using namespace std;
class Instrument{
public:
    virtual void play() const=0;
    virtual string what() const=0;
    virtual void adjust()=0;
};
class Wind : public Instrument {
public:
     void play() const{
           cout << "Wind::play" << endl;
     }
     string what() const{
          return "Wind";
     }
     void adjust(){
          // Assume this will modify
          // the object
     }
};
```

Soyut Taban Sınıfı (Abstract Base Class) ve Saf Sanal Fonksiyonlar (Pure Virtual Functions)

- **Instrument,** sınıfı türetilen bütün sınıflar için ortak bir ara yüz oluşturmaktadır.



Çok işlevde Yapıcı Fonksiyonlar (virtual functions & constructors)

- Virtual fonksiyon içeren bir nesne yaratılırken, uygun VTABLE tablosunu gösteren VPTR başlatılmalıdır.
- Bu işlem, virtual fonksiyona çağrı olasılığı oluşmadan yapılmalıdır.
- Tahmin edilebileceği gibi, yapıcı fonksiyon nesnenin varolmasında görev yürütür. Bu nedenle de, VPTR oluşturulması yapıcı fonksiyonun görevidir.
- Derleyici, VPTR başlatacak kodu, yapıcı fonksiyonun başına gizli olarak ekler.
- Eğer sınıf virtual fonksiyona sahip ise, sentezlenen yapıcı fonksiyon uygun VPTR başlangıç kodunu içerir.
- **Böylece, bir kurucu fonksiyon virtual olarak tanımlanamaz.(A constructor cannot be virtual).**

Çok işlevde Yıkıcı Fonksiyonlar (Destructors and virtual destructors)

- **virtual** kelimesi, yapıcı fonksiyonlar(constructors) ile kullanılmaz. Ancak, yıkıcı fonksiyonlar (destructors) virtual olabilir, hatta olmalıdır..

```
// Behavior of virtual vs. non-virtual destructor
#include <iostream>
using namespace std;
class Base1 {
public:
        ~Base1() { cout << "~Base1()\n"; }
};
class Derived1 : public Base1 {
public:
        ~Derived1() { cout << "~Derived1()\n"; }
};
class Base2 {
public:
        virtual ~Base2() { cout << "~Base2()\n"; }
};
class Derived2 : public Base2 {
public:
        ~Derived2() { cout << "~Derived2()\n"; }
};
```

Çok işlevde Yıkıcı Fonksiyonlar (Destructors and virtual destructors)

```
int main() {
    Base1* bp = new Derived1; // Upcast
    delete bp;
    Base2* b2p = new Derived2; // Upcast
    delete b2p;
}
```

```
Output:
    ~Base1()
    ~Derived2()
    ~Base2()
```

Saf Sanal Yıkıcı Fonksiyonlar (Pure virtual destructors)

- Standart C++ içinde, saf sanal yıkıcı fonksiyon (pure virtual destructors) kullanımı mümkündür. Ancak, ilave bir kısıt vardır: saf sanal yıkıcı fonksiyonun (pure virtual destructor) kodlamasını vermelisiniz..

```
// Pure virtual destructors
// seem to behave strangely
class AbstractBase {
public:
        virtual ~AbstractBase() = 0;
};

AbstractBase::~AbstractBase() {}

class Derived : public AbstractBase {};

// No overriding of destructor necessary?
int main() {
        Derived d;
}
```

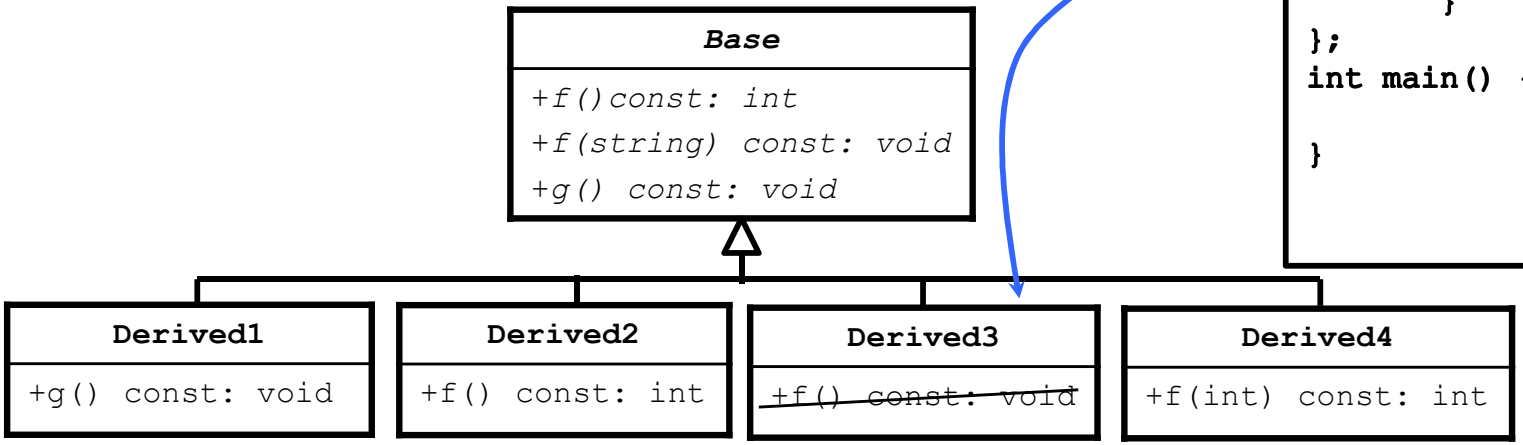
Yükleme ve Yerini Alma (Overloading & overriding)

- Yükleme ve yerine alma işleminde, sadece dönüş tipinin değişmesi kabul edilmez.

```
#include <iostream>
#include <string>
using namespace std;
class Base {
public:
     virtual int f() const {
           cout << "Base::f()\n";
           return 1;
      }
      virtual void f(string) const {}
      virtual void g() const {}
};
class Derived1 : public Base {
public:
      void g() const {}
};
```

```
class Derived2 : public Base {
public:
      // Overriding a virtual function:
      int f() const {
            cout << "Derived2::f()\n";
            return 2;
      }
};
class Derived3 : public Base {
public:
      // Cannot change return type:
      //! void f() const{
      //       cout<< "Derived3::f()\n";}
};
class Derived4 : public Base {
public:
      // Change argument list:
      int f(int) const {
            cout << "Derived4::f()\n";
            return 4;
       }
};
int main() {

}
```

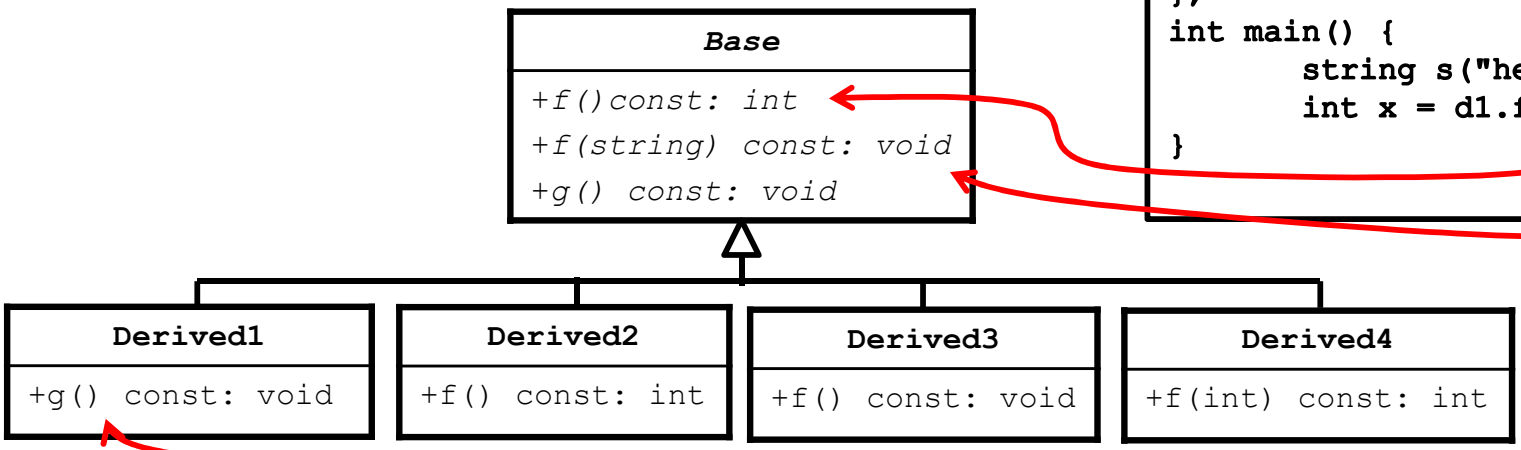


Yükleme ve Yerini Alma (Overloading & overriding)

- Miras (inheritance yapısı), Base sınıfındaki f fonksiyonları ile Derived1 sınıfındaki g fonksiyonuna (overriding) çağrı olur.

```
#include <iostream>
#include <string>
using namespace std;
class Base {
public:
     virtual int f() const {
           cout << "Base::f()\n";
           return 1;
      }
      virtual void f(string) const {}
      virtual void g() const {}
};
class Derived1 : public Base {
public:
      void g() const {}
};
```

```
class Derived2 : public Base {
public:
      // Overriding a virtual function:
      int f() const {
            cout << "Derived2::f()\n";
            return 2;
      }
};
class Derived3 : public Base {
public:
      // Cannot change return type:
      //! void f() const{
      //       cout<< "Derived3::f()\n";}
};
class Derived4 : public Base {
public:
      // Change argument list:
      int f(int) const {
            cout << "Derived4::f()\n";
            return 4;
       }
};
int main() {
       string s("hello"); Derived1 d1;
       int x = d1.f(); d1.f(s); d1.g();
}
```

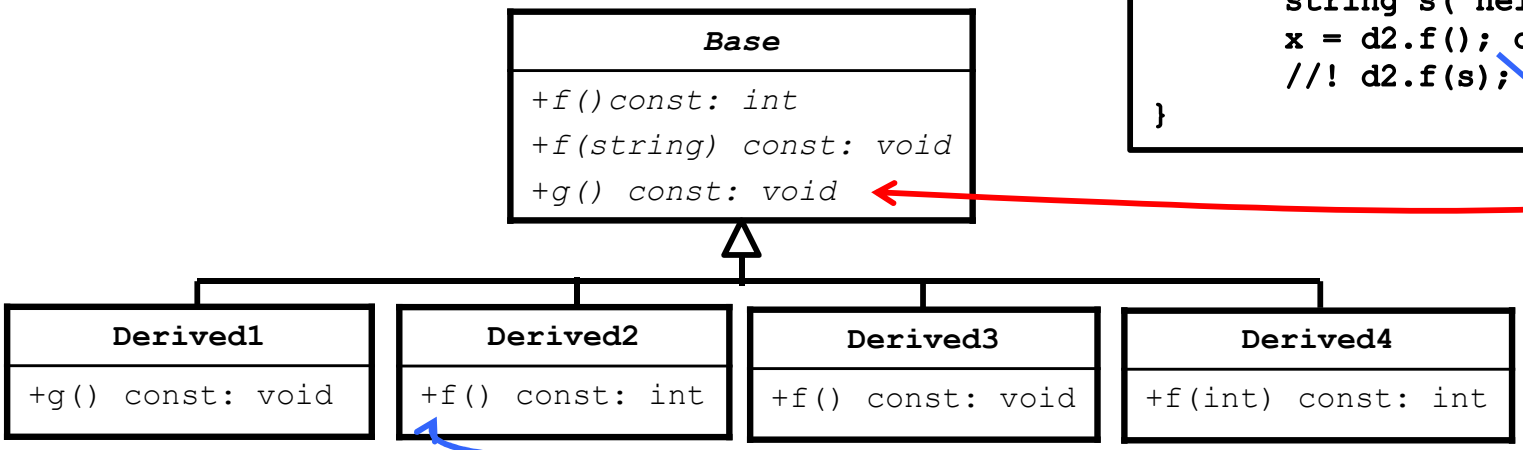


Yükleme ve Yerini Alma (Overloading & overriding)

- Miras (inheritance yapısı), Derived2 sınıfındaki f fonksiyonu (overriding) ile Base sınıfındaki g fonksiyonuna çağrı olur.

```
#include <iostream>
#include <string>
using namespace std;
class Base {
public:
     virtual int f() const {
           cout << "Base::f()\n";
           return 1;
      }
      virtual void f(string) const {}
      virtual void g() const {}
};
class Derived1 : public Base {
public:
      void g() const {}
};
```

```
class Derived2 : public Base {
public:
      // Overriding a virtual function:
      int f() const {
            cout << "Derived2::f()\n";
            return 2;
      }
};
class Derived3 : public Base {
public:
      // Cannot change return type:
      //! void f() const{
      //       cout<< "Derived3::f()\n";}
};
class Derived4 : public Base {
public:
      // Change argument list:
      int f(int) const {
            cout << "Derived4::f()\n";
            return 4;
       }
};
int main() {
       string s("hello"); Derived2 d2;
       x = d2.f(); d2.g();
       //! d2.f(s); // string version hidden
}
```

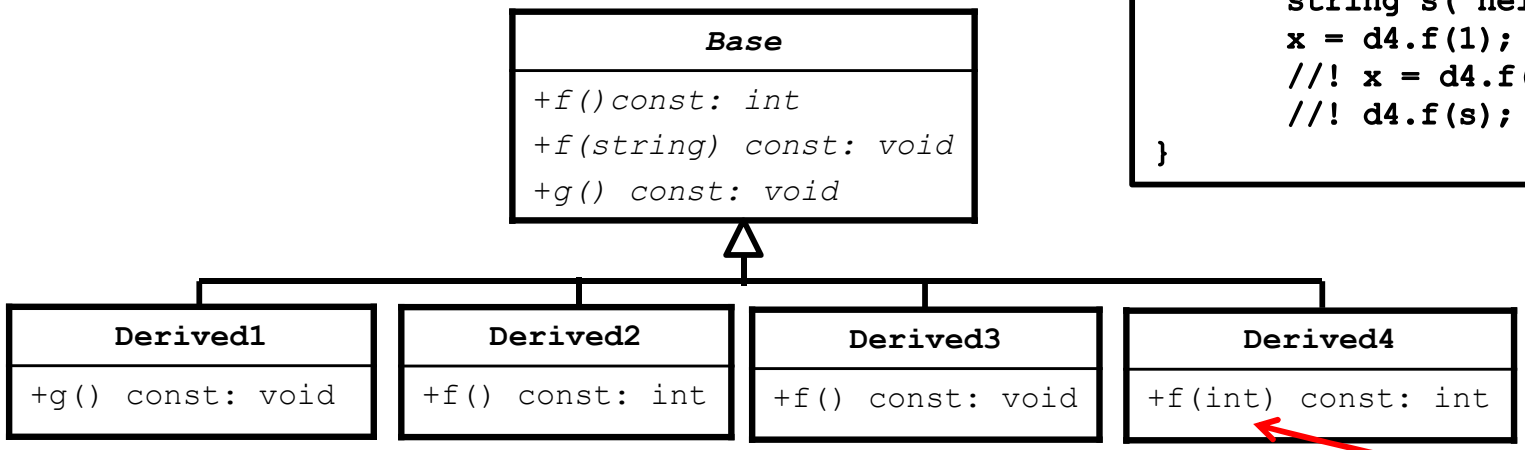


Yükleme ve Yerini Alma (Overloading & overriding)

- Miras (inheritance yapısı), Derived4 sınıfındaki f fonksiyonu (overriding) ile Base sınıfındaki g fonksiyonuna çağrı olur.

```
#include <iostream>
#include <string>
using namespace std;
class Base {
public:
     virtual int f() const {
           cout << "Base::f()\n";
           return 1;
      }
      virtual void f(string) const {}
      virtual void g() const {}
};
class Derived1 : public Base {
public:
      void g() const {}
};
```

```
class Derived2 : public Base {
public:
      // Overriding a virtual function:
      int f() const {
            cout << "Derived2::f()\n";
            return 2;
      }
};
class Derived3 : public Base {
public:
      // Cannot change return type:
      //! void f() const{
      //       cout<< "Derived3::f()\n";}
};
class Derived4 : public Base {
public:
      // Change argument list:
      int f(int) const {
            cout << "Derived4::f()\n";
            return 4;
       }
};
int main() {
       string s("hello"); Derived4 d4;
       x = d4.f(1);
       //! x = d4.f(); // f() version hidden
       //! d4.f(s); // string version hidden
}
```

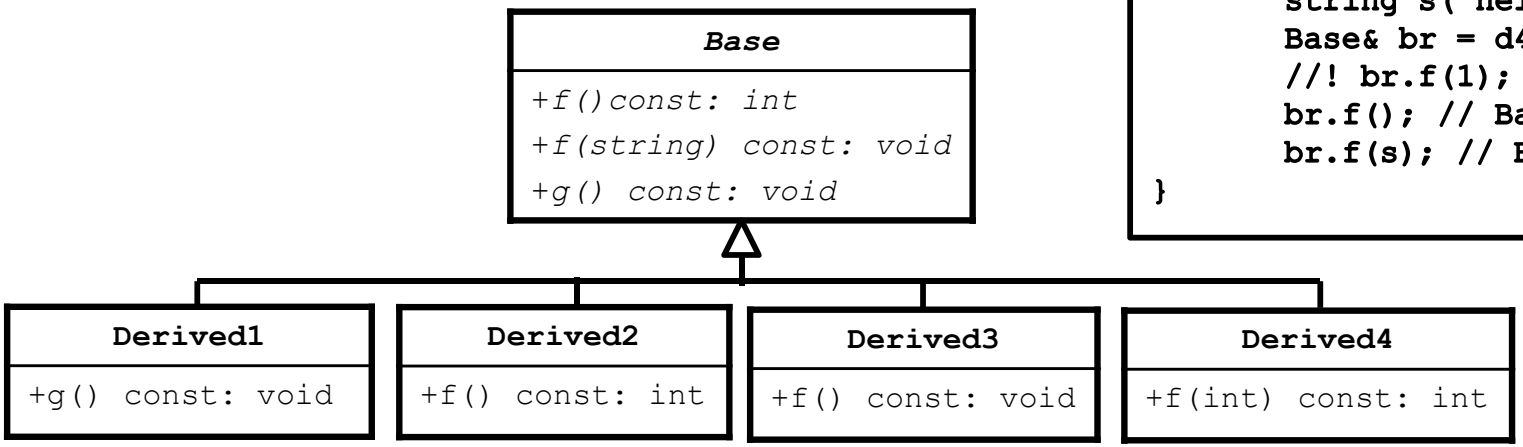


Yükleme ve Yerini Alma (Overloading & overriding)

- Çok işlev (polymorphic yapı), Base sınıfındaki f fonksiyonları ile Base sınıfındaki g fonksiyonuna çağrı olur.

```
#include <iostream>
#include <string>
using namespace std;
class Base {
public:
     virtual int f() const {
           cout << "Base::f()\n";
           return 1;
      }
      virtual void f(string) const {}
      virtual void g() const {}
};
class Derived1 : public Base {
public:
      void g() const {}
};
```

```
class Derived2 : public Base {
public:
      // Overriding a virtual function:
      int f() const {
            cout << "Derived2::f()\n";
            return 2;
      }
};
class Derived3 : public Base {
public:
      // Cannot change return type:
      //! void f() const{
      //       cout<< "Derived3::f()\n";}
};
class Derived4 : public Base {
public:
      // Change argument list:
      int f(int) const {
            cout << "Derived4::f()\n";
            return 4;
       }
};
int main() {
       string s("hello"); Derived4 d4;
       Base& br = d4; // Upcast
       //! br.f(1); // Derived version unavailable
       br.f(); // Base version available
       br.f(s); // Base version abailable
}
```



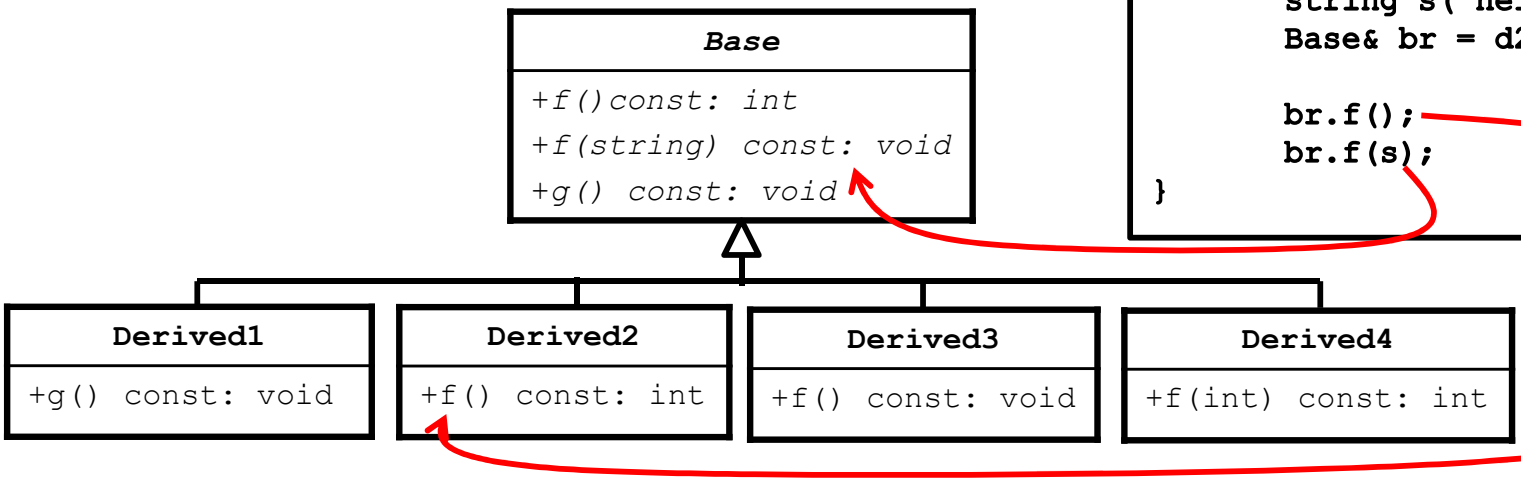
Yükleme ve Yerini Alma (Overloading & overriding)

- Çok işlev (polymorphic yapı), Base sınıfındaki f(string) ve g fonksiyonları, Derived2 sınıfındaki f() fonksiyonuna çağrı olur.

```
#include <iostream>
#include <string>
using namespace std;
class Base {
public:
     virtual int f() const {
           cout << "Base::f()\n";
           return 1;
      }
      virtual void f(string) const {}
      virtual void g() const {}
};
class Derived1 : public Base {
public:
      void g() const {}
};
```

```
class Derived2 : public Base {
public:
      // Overriding a virtual function:
      int f() const {
            cout << "Derived2::f()\n";
            return 2;
      }
};
class Derived3 : public Base {
public:
      // Cannot change return type:
      //! void f() const{
      //       cout<< "Derived3::f()\n";}
};
class Derived4 : public Base {
public:
      // Change argument list:
      int f(int) const {
            cout << "Derived4::f()\n";
            return 4;
       }
};
int main() {
       string s("hello"); Derived2 d2;
       Base& br = d2; // Upcast

       br.f();
       br.f(s);
}
```



Kaynaklar

- T.C. Lethbridge and R. Laganiere, Object-Oriented Software Engineering - Practical software development using UML and Java, McGraw Hill, Second Edition, 2005.
- H.M.Deitel and P.J.Deitel, C++ How To Program, 9E, Pearson Press, 2014.
- B. Stroustrup, The C++ Programming Language, 3rd Edition, Special Edition, Addison Wesley, 2000.
- Dr. Feza Buzluca, Ders Notları.
- Ç. Turhan ve F.C. Serçe, C++ Dersi: Nesne Tabanlı Programlama, 2nci Baskı, 2014.