Nesne yönelimli programlamanın temel ilkelerinden birisi, belki de en önemlisi `"encapsulation"` `(kapsülleme)`. Bu ilke verilerin `(data)` ve bu veriler üzerinde işlem yapacak işlevlerin `(methods)` birlikte tutulması olarak tanımlanabilir. `C++` gibi dillerde kapsüllemeye büyük ölçüde destek olan sınıfın `private` bölümü üzerinde uygulanan erişim kontrolü. Sınıf nesnesinin kullandığı verileri ve sınıf nesnesinin verdiği hizmetlere ilişkin kodları sınıfın `private` bölümüne koyarak bu öğeleri gizlemiş `(data hiding)` hizmet alan kodlardan yalıtmış oluyoruz. Yani arayüz `(interface)` ile kodu `(implementation)` birbirinden ayırıyoruz. Ancak öyle durumlar var ki sınıfın `private` bölümüne sınıfın kendi kodlarının dışında bazı kodların erişebilmesi en iyi çözüm haline gelebiliyor. Şüphesiz, bir sınıfın `private` bölümünü başka kodların erişimine açmak, kodların birbirine bağlılığını `(coupling)` arttırıyor. Ancak ne yapalım, bazen buna katlanmak zorunda kalıyoruz.

`C++` dilinde bir sınıf bir global işleve, başka bir sınıfın bir üye işlevine ya da başka bir sınıfın tüm işlevlerine bir arkadaşlık `(friend)` bildirimi ile, kendi `private` bölümüne erişim hakkı verebiliyor. Arkadaşlık `C++` dilinde müşteri kodlardan ziyade çoğunlukla sınıfın `public` arayüzünde yer alan kendi global işlevlerine ve içsel sınıflarına verildiğinden veri gizleme ilkesi ihlal edilmiş olmuyor. Bir başka sınıfın, sınıfımızın yalnızca seçilmiş belirli `private` öğelerine erişebilmesini istiyor olalım. Bunun `C++` dilinde doğrudan bir yolu yok. `C++`'ta `friend` bildirimiyle, yalnızca sınıfın belirli `private` öğelerine erişim hakkı verilemiyor. Eğer bir sınıfa arkadaşlık verirseniz sınıfınızın tüm private öğelerini o sınıfın erişimine açmış oluyorsunuz. Yani ya hep ya hiç.

`friend`'lik iki sınıf arasındaki bağlantıyı `(coupling)` arttırıyor. Kodların birbirine bağlılığını azaltmak için sınıfın `private` öğelerinin tümünü erişime açmak yerine yalnızca bu ihtiyaca konu `private` öğeleri başka kod ya da kodların erişimine açmak daha iyi bir çözüm. Aşağıdaki gibi bir sınıfımız olsun:

```
class OurClass {
public:
 //public arayüz
private:
 int mx1, mx2;
 void pfunc1();
 void pfunc2();
 void pfunc3();
 //...
};
```

`OtherClass` isimli bir sınıfa, sınıfımızın yalnızca private mx1 veri öğesine ve private pfunc1 isimli üye işlevine erişim olanağı sağlamak istiyoruz. `OtherClass` sınıfı şöyle olsun:

```
class OurClass;

class OtherClass{
public:
 void foo(OurClass& r);
 //...diğer public öğeler
};
```

Amacımızı gerçekleştirmenin birden fazla yolu olsa da en sık kullanılan yöntemlerden biri İngilizcede `Attorney Client` olarak isimlendirilen idiyomu kullanmak. İdiyom basitçe şöyle: `OtherClass` sınıfına doğrudan arkadaşlık vermek yerine -ki o zaman tüm private öğelerimizi `OtherClass` sınıfının erişimine açmış olurduk- kendi oluşturacağımız bir yardımcı sınıfa arkadaşlık vereceğiz. Arkadaşlık vereceğimiz sınıfın ismi `Helper` olsun. Yardımcı `Helper` sınıfı yalnızca static private işlevlere sahip olacak ve o da `OtherClass` sınıfına arkadaşlık verecek:

```
class OurClass {
public:
 //public arayüz
private:
 int mx1, mx2;
 void pfunc1();
 void pfunc2();
 void pfunc3();
 friend class Helper;
 //...
};


class Helper {
 static void pfunc1(OurClass& r)
 {
  r.pfunc1();
 }

 static int get_mx(const OurClass& r)
 {
  return r.mx1;
 }
 friend class OtherClass;
};
```

Artık, `OtherClass` sınıfı `OurClass` sınıfının `private` `pfunc1` işlevine ve `mx` veri öğesine `Helper` sınıfının `static private` işlevlerine çağrı yaparak erişebilir:

```
class OtherClass {
public:
 void foo(OurClass& r)
 {
  Helper::pfunc1(r);
  auto a{Helper::get_mx(r)};
  ///
 }
};
```

`Helper` sınıfının amacı gerçekleştirmek için sunduğu `private` işlevlerin `inline` olarak tanımlandığını görüyorsunuz. `OtherClass` sınıfının `private` öğelerine erişim için `Helper` sınıfının `private` işlevlerine yaptığı çağrılar ciddi bir ek maliyet getirmeyecek.

Biraz da idiyoma verilen popüler ada değinelim: Avukat Müvekkil idiyomu. Buradaki yapı bir avukat ile müvekkili arasındaki ilişkiye benzemiyor mu? Bir avukat, temsil ettiği müvekkilinin tüm sırlarını biliyor olabilir ancak bu

sırların sadece bir kısmını başkalarıyla paylaşır. Örneğimizdeki `Helper` sınıfı avukat, `OurClass` sınıfı da müvekkil. Nasıl farklı işlerimiz için birden fazla avukat tutabiliyorsak, farklı farklı sınıflara sınıfımızın `private` bölümünün farklı kesimlerine erişim sağlamak için birden fazla yardımcı sınıf da oluşturabiliriz, değil mi?

Arkadaşlık bildirimleri taban sınıflardan kalıtım yoluyla türemiş sınıflara geçmiyor. `Base` sınıfı `AttorneyBase` sınıfına arkadaşlık vermiş olsun. Bu durumda `AttorneyBase` sınıfı `Base` sınıfının `private` bölümüne erişebilir. `Der` sınıfı da `Base` sınıfından `public` kalıtımı yoluyla elde edilmiş olsun. Bu durumda `AttorneyBase` sınıfı `Der` sınıfının `private` bölümüne erişemez. Ancak taban sınıf tarafından arkadaşlık verilen bir sınıf, taban sınıfın `private` bölümünde bildirilen bir sanal işlevi `(virtual function)` taban sınıf göstericisi ya da referansı ile çağırdığında, türemiş sınıfın ezen `(override)` işlevinin çalıştırılmasına bir engel yok. Aşağıdaki kodu inceleyin:

```
#include <iostream>

class Base {
 virtual void vfunc()
 {
  std::cout << "Base::vfunc" << "\n";
 }

 friend class BaseAttorney;
};


class Der : public Base
{
 virtual void vfunc() override
 {
  std::cout << "Der::vfunc" << "\n";
 }
};

class BaseAttorney
{
 static void sfunc(Base& bref)
 {
  bref.vfunc();
 }

 friend class Fclass;
};

class Fclass
{
public:
 void foo()
 {
  Base base_object;
  BaseAttorney::sfunc(base_object);  //Base::vfunc
  Der der_object;
  BaseAttorney::sfunc(der_object);  //Der::vfunc
 }
};

int main()
{
 Fclass f;

 f.foo();
}
```