**ARÇELİK C++ ANAHTAR NOTLAR  
Eğitmen: Kaan ASLAN**

**Kurs Başlangıcı: 19/10/2018**

**Giriş**

- C++ C’yi kapsayan bir programalama dilidir. C++’ın C’den farkı genel olarak fazlalık biçimindedir. Yani C’de olan her şey C++’ta da vardır. Ancak C++’ın daha fazla özellikleri vardır. (Ancak aslında bu bir özet anlatımdır. Çünkü C++’ın C’den farklı olduğu noktalar da vardır. Yani aslında bir C programının C++ derleyicisi ile derlendiğinde sorun çıkartacak noktaları da vardır.)

C++’ın C’den fazlalıkları ve farklılıkları iki gruba ayrılarak incelenebilir:

1) C++’ın C’den NYPT (Nesne Yönelimli Programlama Tekniği) İle Doğrudan İlgisi Olmayan Fazlalıkları ve Farklılıkları.

2) C++’ın C’den NYPT(Nesne Yönelimli Programlama Tekniği) İle Doğrudan İlgisi Olan Fazlalıkları ve Farklılıkları

**C++’ın C’den NYPT İle Doğrudan İlgisi Olmayan Fazlalıkları ve Farklılıkları**

Burada her özellik bir anahtar cümleyşle başlatılacaktır.

**1) C++'ta // ile satır sonuna kadar yorumlama yapılabilmektedir:** // ile satır sonuna kadar yorumlama klasik C'de (C90) yoktu. Ancak pek çok C derleyicisi bir eklenti (extension) biçiminde bunu destekliyordu. Zaten daha sonra C99 ile birlikte bu özellik C'ye de sokulmuştur. Ancak başından beri C++'ta // ile satır sonuna kadar yorumlama vardır.

**2) C++’ta Yerel Değişkenler Blokların Başlarında Bildirilmek Zorunda Değildir:** C90’da yerel değişkenler blokların başlarında bildirilmek zorundaydı. C99’da bu zorunluluk kaldırılmıştır. C++’ta da yerel değişkenler blokların herhangi bir yerinde bildirilebilir. Yerel değişkenler blok içerisinde bildirildiği noktadan blok sonuna kadarki bölgede kullanılabilirler.

**3) C++’ta default int Kuralı Kaldırılmıştır**: Bildirimlerde tür belirten sözcükler (type specifiers) bulunmak zorundadır. Örneğin:

foo()

{

//...

}

Bu bildirim C’de geçerlidir, ancak C++’ta geçersizdir.

**4) C++’ta Çağrılan Bir Fonksiyonun Daha Yukarıda Bildiriminin Bulunması (Fonksiyon Tanımlamasının ya da Prototipinin) Zorunludur:** Anımsanacağı gibi C’de çağrılma noktasına kadar çağrılan fonksiyonun bir bildirimi yapılmamışsa derleyici o fonksiyonun geri dönüş değeri int olarak bildirildiğini varsaymaktadır. Halbuki C++’ta çağrılma noktasına kadar fonksiyonun ya kendisinin tanımlanmış olması ya da prototip bildiriminin yapılmış olması gerekir. Örneğin aşağıdaki kod C’de başarılı olarak derleneceği halde C++’ta error oluşturur:

#include <stdio.h>

int main()

{

double result;

result = sqrt(10);

printf("%f\n", result);

return 0;

}

**5) C++’ta for Döngüsünün Birinci Kısmında ve Diğer Deyimlerin Parantezleri İçerisinde Bildirim Yapılabilir:** Örneğin aşağıdaki bildirim C++’ta geçerlidir, ancak C90’da (klasik C) geçersizidir:

#include <stdio.h>

int main()

{

for (int i = 0; i < 10; ++i)

printf("%d\n", i);

return 0;

}

Zaten C99’a da bu özellik eklenmiştir. Ayrıca Java gibi C# gibi dillerde for döngüsünün birinci kısmında bildiirm yapma özelliği başından beri vardır. for döngüsünün birinci kısmında bildirilen değişkenlerin faaliyet alanları for döngüsü ile sınırlıdır. Bunlar döngünün dışında kullanılamazlar. Örneğin:

#include <stdio.h>

int main()

{

for (int i = 0; i < 10; ++i) {

//..

}

i = 100; // error!

return 0;

}

Başka bir deyişle:

for (bildirim; ifade; ifade) {

//...

}

işleminin eşdeğeri şöyledir:

{

bildirim;

for (; ifade; ifade) {

//...

}

}

Örneğin:

#include <stdio.h>

int main()

{

for (int i = 0; i < 10; ++i) {

// i'yi kullanırsak bu for döngüsünün i'si anlaşılır

for (int i = 0; i < 10; ++i) {

// i'yi kullanırsak bu for döngüsünün i'si anlaşılır

}

}

// Burada artıık hiçbir i'yi kullanamayız

return 0;

}

for döngüsünün birinci kısmında bildirilen değişkene ilkdeğer vermememe geçerli fakat anlamsızdır.

C++’ta ayrıca deyimlerin paranztezleri içerisinde de bildirimler yapılabilmektedir. Bu özellik C99’da yoktur. Örneğin:

while (int \*pi = (int \*)malloc(sizeof(int))) {

//...

}

**6) C++’ta İsmine “Referans” Denilen Göstericilere Benzer Başka Bir Tür de Vardır:** C++’ta referanslar & atomu ile bildirilirler. Referans bildirilirken ilkdeğer verilmesi zorunludur. Örneğin:

int a;

int &r = a; // geçerli

int &k; // error! ilkdeğer verilmemiş

Referanslar aslında göstericiler gibi adres tutarlar. Referansa verilen ilkdeğerdeki nesnesnin adresi otomatik olarak referansın içerisine yerleştirilmektedir. Örneğin:

int a;

int &r = a;

Burada r’nin içerisine a’nın adresi yerleştirilmektedir.

Her referansın eşdeğer gösterici karşılığı oluşturulabilir. Yani referansla yapılmak istenen şey eşdeğer olarak göstericilerle de yapılabilir. Örneğin:

int a;

int &r = a;

Bu bildirimlerin eşdeğer gösterici karşılığı şöyledir:

int a;

int \*r = &a;

Bir referansa adres bildirim sırasında ilkdeğer olarak verilen nesne yoluyla yerleştirilir sonra da referansın içerisindeki adres bir daha değiştirilemez. Bir referans ilkdeğer verildikten sonra kullanıldığında artık referansın içerisindeki adreste bulunan nesne anlaşılır. Örneğin:

int a;

int &r = a;

r = 10;

Burada artık 10 değeri r’nin adresini tuttuğu nesneye (yani a’ya) yerleştirilmektedir. Bu işlemin eşdeğer gösterici karşılığı şöyledir:

int a;

int \*r = &a;

\*r = 10;

Örneğin:

#include <stdio.h>

int main()

{

int a = 10;

int &r = a;

r = 20;

printf("%d\n", a); // 20

return 0;

}

C++'ta referanslar “güvenli göstericiler” oluşturmak için dile eklenmiştir. Yani bir referansa bildirim sırasında bir adres yerleştirilr. Sonra bir daha bu adres değiştirilemez.

Örneğin:

int a = 10;

int &r = a;

int &k = r;

Burada k’da da artık a’nın adresi bulunmaktadır. Bunun eşdeğer gösterici karşılığı şöyledir:

int a = 10;

int \*r = &a;

int \*k = &\*r;

Bir referans aynı türden bir nesne ile ilkdeğer verilerek bildirilmek zorundadır. Örneğin:

long a;

int &r = a; // geçersiz (error)

Referansa verilen ilkdeğer nesne belirtmek zorundadır. Örneğin sabitler nesne belirtmemektedir:

int &r = 10; // geçersiz (error), 10 nesne değil, adresi yok!

Özetle bir referans aynı türden bir nesneyle ilkdeğer verilerek tanımlanmak zorundadır.

C ve C++ semantiğine göre parametreli bir fonksiyonun çağrılması aynı zamanda fonksiyonun parametre değişkenine ilkdeğer verme işlemidir.

Bir fonksiyonun parametre değişkeni bir referans türünden olabilir. Bu durumda biz o fonksiyonu aynı türden bir nesneyi argüman yaparak çağırmak zorundayız. Bu durumda da nesnenin adresi parametre değişkeni olan referansa derleyici tarafından atanacaktır. Böylece fonksiyon içerisinde biz referansı kullandığımızda artık o argüman olarak belirtilen nesne anlamına gelecektir. Örneğin:

#include <stdio.h>

void Foo(int &r)

{

r = 100;

}

int main()

{

int a = 10;

Foo(a); // int &r = a

printf("%d\n", a);

return 0;

}

Bu işlemin eşdeğer gösterici karşılığı şöyle oluşturulabilir:

#include <stdio.h>

void Foo(int \*r)

{

\*r = 100;

}

int main()

{

int a = 10;

Foo(&a); // int \*r = &a

printf("%d\n", a);

return 0;

}

İki değişken içerisindeki değerleri yer değiştiren Swap isimli fonksiyon referanslar kullanılarak şöyle yazılabilir:

#include <stdio.h>

void Swap(int &a, int &b)

{

int temp = a;

a = b;

b = temp;

}

int main()

{

int x = 10, y = 20;

Swap(x, y);

printf("x = %d, y = %d\n", x, y);

return 0;

}

Bu işlemin eşdeğer gösterici karşılığı da şöyledir:

#include <stdio.h>

void Swap(int \*a, int \*b)

{

int temp = \*a;

\*a = \*b;

\*b = temp;

}

int main()

{

int x = 10, y = 20;

Swap(&x, &y);

printf("x = %d, y = %d\n", x, y);

return 0;

}

Yapılar da benzer biçimde referans yoluyla fonksiyonlara aktarılabilir. Tabii bu durumda fonksiyonun parametre değişkeni ilgili yapı türünden referans olur. Biz de fonksiyon içerisinde nokta operatörüyle yapı elemanlarına erişebiliriz. Örneğin:

#include <stdio.h>

struct PERSON {

char name[32];

int no;

};

void DispPerson(struct PERSON &r)

{

printf("%s, %d\n", r.name, r.no);

}

int main()

{

struct PERSON per = { "Kaan Aslan", 123 };

DispPerson(per);

return 0;

}

Kodun eşdeğer gösterici karşılığı da şöyledir:

#include <stdio.h>

struct PERSON {

char name[32];

int no;

};

void DispPerson(struct PERSON \*r)

{

printf("%s, %d\n", (\*r).name, (\*r).no); // printf("%s, %d\n", r->name, r->no);

}

int main()

{

struct PERSON per = { "Kaan Aslan", 123 };

DispPerson(&per);

return 0;

}

Referanslarla göstericiler arasındaki benzerlikler ve farklılıklar şöyledir:

1) Referansların içerisine adresler bildirim sırasında bir kez yerleştirilir, bir daha da onların içerisindeki adresler değiştirilemez. Halbuki göstericler içerisindeki adresleri istediğimiz zaman değiştirebiliriz. Bu özellik referansların kimi durumlarda daha güvenli kullanımına olanak sağlamaktadır. C++’ta her zaman eğer mümkünse referans kullanımı tercih edilmelidir.

2) Referanslar dizi gibi veri yapılarının adres artırımlarıyla dolaşılmasında kullanılamazlar. Dolayısıyla bu tür durumlarda yine gösterici kullanmak gerekir. Örneğin gösterici bir dizinin ilk elemanını gösteriyor durumda olabilir. O gösterici artırıldıkça dizinin sonraki elemanlarına erişilebilir. Halbuki referanslar tekil nesneleri göstermek için daha çok kullanılmaktadır (Tabii dizi referansları da söz konusu olabilir.)

3) Referansların adresteki nesneye erişme durumlarında kullanımları daha kolaydır. Bunlarda \* operatörü zaten adeta default olarak kullanılıyor gibidir.

Şimdi referans kullanımına ilişkin bazı ayrıntılar üzerinde duralım:

Bir referans const olabilir. Bu durum referansın gösterdiği (refere ettiği) nesnenin değiştirilemeyeceği anlamına gelir. Örneğin:

#include <stdio.h>

int main()

{

int a = 10;

const int &r = a;

printf("%d\n", r);

r = 20; // geçersiz (error)

a = 30; // geçerli

return 0;

}

const refernsların eşdeğer gösterici karşılıkları gösterdiği yer const olan const gösterilerle oluşturulabilir. Örneğin:

#include <stdio.h>

int main()

{

int a = 10;

const int \*r = &a;

printf("%d\n", \*r);

\*r = 20; // geçersiz (error)

a = 30; // geçerli

return 0;

}

Özellikle const referanslar fonksiyon parametresi olarak sıkça kullanılmaktadır. Bu durum adresi alınan nensnenin değiştirilemeyeceği anlamına gelir. Örneğin:

void Foo(const int &r)

{

//...

}

Burada fonksiyona ya da prototipe bakan kişi fonksiyonun adresi ile aldığı nesneyi değiştirmeyeceği sonucunu çıkartır. Bu da kodun daha iyi anlaşılmasına yol açar. Aynı zamanda bu durum derleyici için bazı optimizasyonaların yapılmasına da olanak sağlayabilmektedir. O halde fonksiyonun parametresi bir referans ise ve fonksiyon adresi ile aldığı nesneyi değiştirmeyecekse parametrinin const referans yapılması iyi bir tekniktir.

Mademki referansların içerisindeki adresler ilkdeğer verildikten sonra bir daha değiştirilemiyorlar o halde referanslar kendisi const olan const göstericilere benzetilebilir. Örneğin:

int a = 10;

int &r = a;

Bu işlemin eşdeğer gösterici karşılığı şöyle de belirtilebilir:

int a = 10;

int \* const r = &a;

Böylece:

int a = 10;

const int &r = a;

Kodunun da eşdeğer gösterici karşılığı şöyle ifade edilebilir:

int a = 10;

const int \*const r = &a;

Normal olarak bir referans aynı türden bir nesneyle ilkdeğer verilerek tanımlanır. Ancak eğer referans const ise referans farklı türden bir nesneyle ilkdeğer verilerek de tanımlanabilir. Örneğin:

long a;

const int &r = a; // geçerli

Ancak bu durumda referansın const olmasının zorunlu tutulduğuna dikkat ediniz. Böylesi durumlarda aslında derleyici önce referans ile aynı türden geçici bir nesne (temporary object) oluşturur. İlkdeğer olarak verilen nesnenin değerini bu geçici nesneye atar ve bu geçici nesnenin adresini referansa yerleştirir. Yani örneğin:

long a;

const int &r =a;

işleminin eşdeğeri aslında şöyledir:

long a;

int temp = a;

const int &r = temp;

Aslında bir referans bir sabitle de ilkdeğer verilerek yaratılabilir. Bu durumda referansın yine const olması zorunludur. const bir referans bu biçimde bir sabit ile ilkdeğer verilerek tanımlandığında önce geçici bir nesne yaratılır. Geçici nesnenin adresi referansa yerleştirilir. Örneğin:

const int &r = 10;

Burada 10 değeri önce derleyici tarafından yaratılan bir geçici nesneye yerleştirilecek sonra onun adresi r’ye atanacaktır.

C++’a C++11 standartlarıyla birlikte “sağ taraf değeri alan referans (R-Value Reference)” denilen referanslar da eklenmiştir. Bu tür referanslar && atomuyla bildirilir. Sağ taraf değeri alan referanslar doğrudan nesne belirtmeyen sabitler ve ifadelerle ilkdeğer verilerek tanımlanabilmektedir. Örneğin:

#include <stdio.h>

int main()

{

int &&r = 10; // sağ taraf değeri alan referans

printf("%d\n", r);

r = 20;

printf("%d\n", r);

return 0;

}

Sağ taraf değeri alan referanslara nesne belirten ifadelerle ilkdeğer verilemez. Örneğin:

int a = 10;

int &&r = a; // geçersiz (error)

Ya da örneğin:

long a = 10;

int &&r = a; // geçersiz (error)

Ancak otomatik dönüştürme ile sağ taraf değeri alan referansın türüne dönüştürülebilen sağ taraf değeri türleriyle bu referanslara ilkdeğer verilebilir. Örneğin:

int &&r = 12.4; // geçerli

Sağ taraf değeri alan referanslara yine nesne belirtmeyen ifadelerle ilkdeğer vcerildiğinde önce aynı türden geçici bir nesne yaratılıp bu geçici nesnenin adresi referansa atanmaktadır. Ancak bu referans yoluyla artık bu geçici nesne değiştirilebilmektedir. Bu referanslar C++’a “move semantics” denilen özellik yüzünden eklenmiştir. Sağ taraf değeri alan referanslar const da olabilirler. Örneğin:

const int &&r = 12.4; // geçerli

Fonksiyonların geri dönüş değerleri de referans olabilir. Çünkü aslında return işlemi geri dönüş değeri türünden geçici bir nesnenin yaratılarak ilkdeğer verilmesi işlemidir.

#include <stdio.h>

int g\_a = 10;

int &Foo() // int &temp = g\_a

{

return g\_a;

}

int main()

{

Foo() = 20; // burada aslında g\_a nesnesine değer atanmaktadır

printf("%d\n", g\_a); // 20

return 0;

}

Bu işlemin eşdeğer gösteri karşılığı şöyle ifade edilebilir:

#include <stdio.h>

int g\_a = 10;

int \*Foo() // int &temp = g\_a

{

return &g\_a;

}

int main()

{

\*Foo() = 20; // burada aslında g\_a nesnesine değer atanmaktadır

printf("%d\n", g\_a); // 20

return 0;

}

**7) C++’ta Dinamik Bellek Tahsisatı İçin new ve delete İsimli İki Operatör Vardır:** Bilindiği gibi C'de dinamik bellke yönetimi için malloc, calloc, realloc ve free fonksiyonları kullanılmaktadır. Ancak C++'ta dinamik bellek yönetimi için new ve delete isimli iki operatör bulundurulmştur.

new operatörünün genel biçimi şöyledir:

1) new <tür>

2) new <tür><[<uzunluk ifadesi>]

Birinci genel biçimde new operatörünün yanına tür belirten bir sözcük yazılır. Örneğin:

new int

new double

new Sample

Bu durumda o türden bir elemanlık yer heap’te tahsis edilir ve new operatörü tahsis edilen o alanın adresini bize verir. Örneğin:

#include <stdio.h>

int main()

{

int \*pi;

pi = new int;

\*pi = 10;

printf("%d\n", \*pi);

return 0;

}

new operatörünün bize verdiği adres tahsis edilen tür türündendir. (Halbuki malloc fonksiyonu bize void \*) türünden adres vermektedir.

İkinci genel biçimde new operatörünü önce bir tür, sonra da köşeli parantez içerisinde bir uzunluk ifadesi izler. Bu durumda o türden o uzunluk kadar ardışıl eleman heap’te tahsis edilir ve bize new operatörü o alanın başlangıç adresini verir. Örneğin:

#include <stdio.h>

int main()

{

int \*pi;

pi = new int[10];

for (int i = 0; i < 10; ++i)

pi[i] = i \* 10;

for (int i = 0; i < 10; ++i)

printf("%d ", pi[i]);

printf("\n");

return 0;

}

new operatörü aslında tahsisatı programın çalışma zamanı sırasında “operator new” isimli bir fonksiyonu çağırarak yapmaktadır. operator new fonksiyonu da default durumda C'nin malloc fonksiyonunu çağırmaktadır.

new ile tahsis edilen dinamik alan delete operatörüyle yok edilebilir. delete operatörünün de iki genel biçimi vardır:

1) delete <adres>

2) delete [ ] <adres>

Eğer new işlemi köşeli parantezsiz yapılmışsa delete işleminin de köşeli parantezsiz biçimi kullanılır. Eğer new işlemi köşeli parantezli biçimle yapılmışsa delete işlemi de yine köşeli parantezli olarak gerçekleştirilir. Örneğin:

#include <stdio.h>

int main()

{

int \*pi;

pi = new int[10];

for (int i = 0; i < 10; ++i)

pi[i] = i \* 10;

for (int i = 0; i < 10; ++i)

printf("%d ", pi[i]);

printf("\n");

delete [] pi;

return 0;

}

Burada köşeli parantezin için boş bırakıldığına dikkat ediniz. Örneğin:

#include <stdio.h>

#include <string.h>

char \*GetName()

{

char s[64];

char \*name;

printf("Adi Soyadi:");

gets\_s(s, 64);

name = new char[strlen(s) + 1];

strcpy(name, s);

return name;

}

int main()

{

char \*name;

name = GetName();

puts(name);

delete[] name;

return 0;

}

new operatörü başarısız olduğunda NULL adres vermez. Exception fırlatır. Dolayısıyla programcı bu operatörün ürettiği değeri test etmez. Exception konusu ileride ele alınacaktır.

new ve delete operatörlerine ilişkin bazı ayrıntılar vadır. Fakat bunlar burada ele alınmayacaktır.

**8) C++’ta Farklı Parametrik Yapılara İlişkin Aynı İsimli Birden Fazla Fonksiyon Bulunabilir:** Bu özelliğe İngilizce “function overloading” denilmektedir. Aslında bu özellik neredeyse tüm nesne yönelimli programlama dillerinde vardır.

C’de aynı isimli birden fazla fonksiyon tanımlanamaz. Ancak C++’ta eğer fonksiyonların parametrik yapıları farklıysa bu mümkünüdür. Parametrik yapının farklı olması demek parametrelerin sayıca ya da türce farklı olması demektir. Örneğin:

#include <stdio.h>

void Foo()

{

//...

}

void Foo(int a)

{

//...

}

void Foo(long a)

{

//...

}

void Foo(int a, int b)

{

//...

}

void Foo(int a, long b)

{

//...

}

int main()

{

return 0;

}

Parametre değişkenlerinin isimlerinin bir önemi yoktur. Önemli olan onların türleridir. Fonksiyonların geri dönüş değerlerinin farklı olması da bu anlamda bir farklılık oluşturmamaktadır. Örneğin :

void Foo(int a)

{

//...

}

int Foo(int b) // error! Parametreler aynı türden

{

//...

}

Aynı isimli bir fonksiyon çağrıldığında bunlardan hangisinin çağrılmış olduğunun tespit edilmesi sürecine “overload resolution” denilmektedir.

Overload resolution işleminin bazı ayrıntıları vardır. Fakat özet olarak süreç şöyle ifade edilebilir: Derleyici çağrılma ifadesindeki argümanların türlerine bakar. Bu türlere uygun parametre değişkeni olan fonksiyonun çağrıldığını kabul eder. Örneğin:

#include <stdio.h>

void Print()

{

printf("void\n");

}

void Print(int a)

{

printf("int: %d\n", a);

}

void Print(long a)

{

printf("long: %ld\n", a);

}

void Print(double a)

{

printf("double: %f\n", a);

}

void Print(const char \*str)

{

printf("const char \*: %s\n", str);

}

int main()

{

int a = 123;

long b = 456;

double c = 12.34;

Print(a); // int

Print(b); // long

Print(c); // double

return 0;

}

Pekiyi ya böyle bir fonksiyon yoksa ne olacaktır? İşte bu noktada bazı ayrıntılar devreye girer.

Overload resolution süreci üç aşamada yürütülmektedir Önce aday fonksiyonlar (candidate functions) seçilir. Sonra bu daylar arasından uygun olanlar (viable functions) alınır. Sonra da onların arasından en uygun olan (the most viable) fonksiyon seçilir.

Çağrılması ifadesindeki isimle aynı olan aynı faaliyet alanındaki tüm fonksiyonlar aday fonksiyonlardır. Çağrılma ifadesindeki argüman sayısıyla aynı sayıda parametre değişkenine sahip ve her argümandan parametre değişkenine otomatik dönüştürmenin mümkün olduğu aday fonksiyonlar uygun (viable) fonksiyonlardır. Uygunlar arasından en uygun olan şöyle tespit edilir: En uygun fonksiyon tüm argüman parametre dönüştürmesi kalite bakımından diğer uygun olanlardan ya daha iyi ya da daha kötü olmayan fonksiyondur. Argüman parametre dönüştümeleri arasında kalite farklılıkları vardır. Kalite farklılkları özet olarak şöyle ifade edilebilir:

1) T1 türünden T2 türüne ve T1 türünden T3 türüne dönüştürmeler söz konusu olduğunda T2 ya da T3 türünün hangisi T1 ile aynıysa o daha iyidir. Yani özdeş olan iyidir. Örneğin:

int -> long

int -> int // daha iyi

2) int türüne yükseltme ile dönüştürme (integral promotion) diğer dönüştürmelerden daha iyidir. Örneğin:

char -> int // daha iyi

char -> double

3) Gerçek sayı türleri arasındaki dönüştürmeler, tamsayı türleri arasındaki dönüştürmelerden daha iyidir. Örneğin float -> double, float->int dönüştürmesinde float->double dönüştürmesi daha iyidir.

4) Diğer standart dönüştürmeler (yani diğer nümerik tür dönüştürmeleri arasında bir kalite farkı yoktur)

int -> long

int -> double

Burada bu dönüştürmeler arasında fark yoktur.

5) Standart dönüştürmeler kullanıcı tanımlı dönüştürmelerden daha iyidir.

İşte en uygun fonksiyon tüm argüman parametre dönüştürmesi diğerlerinden daha iyi olan ya da daha kötü olmayan fonksiyondur. Böyle bir fonksiyon hiç yoksa ya da birden fazla varsa çağrım başarız olur. Örneğin:

#include <stdio.h>

void Print(int a, long b) // #1

{

printf("int, long\n");

}

void Print(int a, double b) // #2

{

printf("int, double\n");

}

void Print(double a, int b) // #3

{

printf("double, int\n");

}

void Print(int a, int b) // #4

{

printf("int, int\n");

}

void Print(int a, const char \*str) // #5

{

printf("int, const char \*\n");

}

void Print() // #6

{

printf("void\n");

}

int main()

{

Print(100L, 1.2); // int , double

return 0;

}

Örnekte Print fonksiyonu long ve double argümanlarla çağrılmıştır. 1, 2, 3, 4, 5, ve 6 numaralı fonksiyonlar aday fonksiyonlardır. Ancak 1, 2, 3, 4 numaralı fonksiyonlar uygun fonksiyonlardır. En uygun fonksiyon 2 numaralı fonksiyondur. Çünkü bu fonksiyonun birinci argüman-parametre dönüştürmesi diğer uygun olanlarla yarışa sokulduğunda onlardan daha kötü değildir. İkinci argüman-parametre dönüştürmesi ise diğer uygun olanlardan daha iyidir. Şimfi fonksiyonu şyle çağpırmış olalım:

Print(1.2, 1.3); // error

Burada fonksiyon double, double argümanlarla çağrılmıştır. İkinci fonksiyonun birinci argüman parametre dönüştürmesi üçüncüden daha kötüdür. Üçüncü fonksiyonun da ikinci argüman parametre dönüştürmesi ikinciden daha kötüdür. Yani tüm argüman parametre dönüştürmesi diğerlerinden daha kötü olmayan ya da daha iyi olan tek bir fonksiyon yoktur. Çağrım error ile sonuçlanır. Genellikle bu tür error’lere “ambiguity” denilmektedir.

Benzer işlemleri yapan fonksiyonlara aynı isimlerin verilmesi toplamda kodun daha ele alınmasını kolaylaştırmaktadır.

**10) C++’ta Fonksiyonun Parametre Değişkenlerine Default Değerler Verilebilir:** C’de böyle bir özellik yoktur. C++’ta çağrımı kolaylaştırmak için çağırma ifadesinde bazı argümanlar yazılmazsa onların yerine bazı default değerlerin yazılmış olduğu varsayabilmektedir. Bu nedenle bu özelliğe “default argüman (default argument)” da denilmektedir. Örneğin:

#include <stdio.h>

void Foo(int a = 10, int b = 20)

{

printf("a = %d, %d\n", a, b);

}

int main()

{

Foo(1, 2);

Foo(1); // Foo(1, 20)

Foo(); // Foo(10, 20)

return 0;

}

Burada Foo iki parametreli bir fonksiyondur. Bu fonksiyona her zaman iki argüman gönderilmektedir. Ancak biz çağrım sırasında bunları belirtmeyebiliriz. Bu durumda o default argüman değerinin kullanıldığı varsayılır. Yani yukarıdaki örnekte Foo(1) çağrısı derleyici için aslında Foo(1, 20) çağrısıyla eşdeğerdir. Foo() çağrısı da Foo(10, 20) çağrısı ile eşdeğerdir.

Pekiyi böyle bir özelliğe neden gereksinim duyulmuştur? İşte bazen fonksiyonların bazı parametreleri çoğunlukla belli argümanlarla çağrılır. Bu durumlarda bu argümanların tekrar tekrar çağrımda belirtilmesinin önüne geçilmiş olunmaktadır. Örneğin bir int sayıyı ekrana yazdıran Print fonksiyonun ikinci parametresi sayıyı kaçlık sistemde yazdıracağını belirtiyor olsun. Burada genellikle 10’luk sistem kullanılacağından kolaylık olsun diye bu parametre için default 10 değeri argüman olarak veriebilir:

#include <stdio.h>

#include <stdlib.h>

void Print(int a, int base = 10)

{

char buf[64];

itoa(a, buf, base);

printf("%s\n", buf);

}

int main()

{

Print(123);

Print(123, 8);

Print(123, 2);

return 0;

}

Örneğin:

#include <stdio.h>

void PutMessage(const char \*msg = "Ok")

{

printf("%s\n", msg);

}

int main()

{

PutMessage("Error!");

PutMessage();

return 0;

}

Default değer alan parametrelerin parametre listesinin sonunda bulunması zorunludur. Örneğin aşağıdaki tanımlama geçerli değildir:

void Foo(int a = 10, int b, int c = 20) // error!

{

//...

}

Ancak şu geçerlidir:

void Foo(int b, int a = 10, int c = 20) // geçerli

{

//...

}

Başka bir deyişle bir parametre değişkeni default değer almışsa onun sağında bulunan bütün parametre değişkenlerinin de default değer alması gerekir.

Default değer alan fonksiyonlar için prototip yazılmışsa default değerlerin ya prototipte ya da tanımlama sırasında belirtilmesi gerekir. Her iki yerde de belirtilmesi hata oluşturur. Tabii tavsiye edilen prototipte belirtilip tanımlama sırasında belirtilmemesidir. Örneğin:

#include <stdio.h>

void Foo(int a, int b = 10, int c = 20);

int main()

{

Foo(100);

return 0;

}

//...

void Foo(int a, int b = 10, int c = 20) // geçersiz (error)

{

//...

}

Fakat örneğin:

#include <stdio.h>

void Foo(int a, int b = 10, int c = 20);

int main()

{

Foo(100);

return 0;

}

//...

void Foo(int a, int b, int c) // geçerli

{

//...

}

Default argüman almış fonksiyonların çağrılması sırasında “overload resolution” işleminde iki anlamlılık (ambiguity) hataları ortaya çıkabilir. Örneğin:

#include <stdio.h>

void Foo(int a);

void Foo(int a, int b = 0);

void Foo(int a)

{

//...

}

void Foo(int a, int b)

{

//...

}

int main()

{

Foo(10, 20); // geçerli iki parametreli olan çağrılır

Foo(10); // geçersiz! ambiguity!

return 0;

}

Tabii bu örnekte ikinci Foo fonksiyonunun default argüman almasının bir anlamı kalmamıştır.

Eğer bir fonksiyon %70 oranında o default değerlerle çağrılıyorsa fonksiyonun default argüman alması uygun olur. Aksi taktirde kodun anlamlandırlması konusunda yanlış fikirler edinilebilir.

Bazen parametre değişkenine verilen default değer gerçek değer olmayabilir. O parametrenin default argümanla belirlemek için bu değer kullanılır. Gerçek değer birtakım işlemler sonucunda elde ediliyor olabilir. Örneğin:

#include <stdio.h>

#include <time.h>

void DispDate(int day = 0, int month = 0, int year = 0);

void DispDate(int day, int month, int year)

{

time\_t t;

struct tm \*ptm;

time(&t);

ptm = localtime(&t);

if (day == 0)

day = ptm->tm\_mday;

if (month == 0)

month = ptm->tm\_mon + 1;

if (year == 0)

year = 1900 + ptm->tm\_year;

printf("%02d/%02d/%04d\n", day, month, year);

}

int main()

{

DispDate(26, 1);

DispDate(26);

DispDate();

return 0;

}

Default argümana verilen ilkdeğerin sabit ifadesi olması gerekmez. Örneğin faaliyet alanındaki başka değişkenler bunlara ilkdeğer verme işleminde kullanılabilir. Örneğin:

#include <stdio.h>

#include <math.h>

void Foo(double x = sqrt(10));

int main()

{

Foo(20);

Foo();

return 0;

}

void Foo(double x)

{

printf("%f\n", x);

}

**10) C++’ta Yapı, Sınıf, Birlik ve Enum Türlerinden bildirimlerde struct, class, union ve enum Anahtar Sözcükleri Hiç Belirtilmeyebilir:** C’de yapı türleri belirtilirken struct anahtar sözcüğü ile yapı ismi birlikte kullanılmak zorundadır. Halbuki C++’ta yalnızca yapı ve sınıf isimleri tür belirtmek için kullanılabilir. Örneğin:

#include <stdio.h>

struct Complex {

double real;

double imag;

};

int main()

{

Complex z = { 3, 2 }; // struct Complex = {3, 2};

printf("%f + %fi\n", z.real, z.imag);

return 0;

}

**11) C++’ta bool İsimli Bir Tür Vardır, Ayrıca true ve false Anahtar Sözcükleri de Dile Eklenmiştir:** Bilindiği gibi C90’da doğru-yanlış bilgilerini tutabilecek mantıksal bir tür yoktur. Halbuki C++’a bu amaçla bool isimli bir tür eklenmiştir. (C99’a da \_Bool ismiyle bir mantıksal tür eklendi) Ayrıca C++’a bool türünden iki sabit daha eklenmiş durumdadır: true ve false. Standartlara göre bool türü için ayrılacak alan tamsayı türlerinden herhangi birinin uzunluğu kadar olabilir. Pek çok derleyici default olarak bunun için 1 byte yer ayırmaktadır. bool türü aritmetik işlemlere sokulabilir. Bu durumda işlem öncesinde bool türü otomatik olarak int türüne dönüştürülmektedir. Eğer bool türünün belirttiği değer true ise 1, false ise 0 olarak. Örneğin:

#include <stdio.h>

int main()

{

bool b;

int a;

b = true;

a = b + 5; // 6

printf("%d\n", a);

return 0;

}

Yine skaler türlerle bool türü arasında otomatik dönüştürme vardır. Yani biz bir tamsayı ya da gerçek sayı değerini bool türüne atayabiliriz ya da bool türünden bir değeri tamsayı ya da gerçek sayı türüne atayabiliriz. Farklı türden bir değer bool türüne atanırken sıfır dışı değerler true olarak sıfır değeri false olarak bool türüne dönüştürülmektedir.

Örneğin:

#include <stdio.h>

int main()

{

bool b;

int a;

b = 10.2;

a = b + 1;

printf("%d, %d\n", a, b); // 2, 1

return 0;

}

Ayrıca C++’ta if gibi, while ve for gibi deyimlerin karar ifadelerinin bool türden olacağı belirtilmiştir. Tabii bool türüne otomatik dönüştürme söz konusu olduğu için bunun C ile uyumu bozan bir tarafı yoktur.

**12) C++ C’nin Standart Kütüphanesine Sahiptir Ancak C++’ın Template Tabanlı Ayrı Bir Sınıf Kütüphanesi de Vardır:** Biz C++’ta C’nin standart kütüphane fonksiyonlarını aynı biçimde kullanabiliriz. Ancak bu fonksiyonşların prototiplerinin bulunduğu başlık dosyalarının isimleri değiştirilmiştir. Orijinal C başlık dosyası <X.H> biçimindeyse artık bunun C++’taki ismi <CX> olmuştur. Örneğin:

<stdio.h> yerine <cstdio>

<stdlib.h> yerine <cstdlib>

<math.h> yerine <cmath>

gibi. Fakat C++ C ile uyumu bozmamak için C’nin standart başlık dosyalarını da aynı biçimde desteklemektedir. Biz kursumuzda artık C++’a özgü bu yeni başlık dosyalarının isimlerini kullanacağız.

Örneğin C++’ta ekran, klavye ve genel olarak dosya işlemlerini yapmak için <iostream> isimli bir kütüphane kullanılmaktadır. Bu <iostream> kütüphanesinin nasıl çalıştığı ileride ele alınacaktır. Burada ekrana birşeyler bastırmak için cout isimli nesne kullanılmaktadır. cout nesnesi ostream isimli bir sınıf türündendir. Bu sınıfın da << operatör fonksiyonu yazdırma işlemini yapar. Örneğin:

#include <iostream>

using namespace std;

int main()

{

int a = 10, b = 20;

cout << "This is a test\n";

cout << "a = " << a << ", " << "b = " << b << "\n";

return 0;

}

Örnekteki using direktifi std isim alanına erişmek için kullanılmıştır. Bu konuda ileride ele alınacaktır. <iostream> kütüphanesinde klavyeden okumalar için cin nesnesi kullanılmaktadır. Bu nesne istream sınıfı türündendir. Bu sınıfın >> operatör fonksiyonu okuma işlemini yapar:

#include <iostream>

using namespace std;

int main()

{

int a, b;

cout << "a değerini giriniz:";

cin >> a;

cout << "b değerini giriniz:";

cin >> b;

cout << "a = " << a << ", " << "b = " << b << "\n";

return 0;

}

Biz kurusumuzda artık alıştırma amaçlı <iostream> kütüphanesini de kullanacağız.

**13) C++’ta enum Türleri Ayrı Bir Tür Olarak Değerlendirilmektedir:** Anımsanacağı gibi C’de enum türleri aslında arka planda int türü gibi davranmaktadır. Dolayısıyla enum türünden değişkenler ve enum sabitleri de sanki int türdenmiş gibi ele alınmaktadır. Halbuki C++’ta bunlar ayrı bir tür olarak ele alınırlar. Dolayısıyla örneğin “overload” işleminde aynı isimli farklı enum parametreli fonksiyonlar bir arada bulunabilmektedir. C++’ta artık enum türünden nesnelere farklı türden bir değer (örneğin int değer) doğrudan atanamaz. Böyle bir otomatik dönüştürme yoktur. Fakat tür dönüştürme operatörü ile bu atama yapılabilir. Ayrıca bir enum türünün enum sabitleri (enumerators) ilgili enum türündendir. Bu durumda C++’ta bir enum türünden nesneye o enum türünün enum sabitleri doğrudan atanabilir. Örneğin:

#include <iostream>

using namespace std;

enum Color {Red, Green, Blue};

enum Gender { Male, Female};

int main()

{

Color c;

c = 2; // error!

c = (Color)2; // geçerli

c = Blue; // geçerli

c = Male; // error!

return 0;

}

Ancak C++’ta enum türlerinden tamsayı ve diğer türlere otomatik dönüştürme vardır. (Yani örneğin biz bir tamsayıyı doğrudan enum türüne atayamayız ancak enum türünü tamsayı türüne doğrudan atayabiliriz) Örneğin:

#include <iostream>

using namespace std;

enum Color {Red, Green, Blue};

int main()

{

Color c = Blue;

int a;

a = c + 1;

cout << a << endl;

return 0;

}

C++11 ile birlikte C++'a tıpkı Java ve C#'taki gibi faaliyet alanı enum ile sınırlandırımış enum türleri de eklendi. Bu yeni türlerine "scoped enumeration" denilmektedir. Faaliyet alanı sınırlandırılımış enum türleri enum class ya da enum struct anahtar sözcüğü ile bildirilmektedir. Bu ebun türlerinin enum sabitlerine erişebilmek için :: operatörü kullanılmaktadır. Örneğin:

#include <iostream>

using namespace std;

enum class Color {

Red, Green, Blue

};

int main(void)

{

Color c;

c = Color::Blue;

cout << (int)c << endl;

return 0;

}

**14) C++’ta inline Fonksiyonlar Vardır:** inline fonksiyon kavramı C99’a da eklenmiştir. Ancak C99 ile C++ arasında inline fonksiyon bağlamında bazı küçük farklılıklar vardır. inline fonksiyonlar adeta makro gibi davranan fonksiyonlardır. Derleyici bir inline fonksiyonun çağrımıyla karşılaştığında o fonksiyon çağrısı için CALL makine komutu üretmek yerine o fonksiyonunun içini alarak koda insert eder. Örneğin:

#include <iostream>

using namespace std;

inline int Square(int a)

{

return a \* a;

}

int main()

{

int result;

result = Square(10); /\* result = 10 \* 10; \*/

printf("%d\n", result);

return 0;

}

Burada derleyici Square fonksiyonu çağrıldığında adeta onun iç kodunu alarak çağrılan yere yerleştirir. Ancak inline anahtar sözcüğü bir emir belirtmez. Yani derleyici fonksiyonu inline açabilir ya da açmayabilir. Eğer derleyici fonksiyonu inline açmazsa ona normal birer fonksiyon muamelesi yapar. Örneğin derleyici inline fonksiyonun içi çok karışıksa, özyinelemeli çağırma içeriyorsa ya da fonksiyon zaten çok uzun miktarda döngülere sahipse onu inline olarak açmak istemeyebilir. Çünkü bu durumda elde edilecek kazanç kodu büyütmenin maliyetini karşılamayacaktır. Çünkü her inline çağrım, kodu büyütme eğilimi gösterir. Programcı birkaç satırlık basit fonksiyonları inline olarak bildirebilir. Ayrıca pek çok derleyicide optimizasyon seçenekleri kapalıysa inline açım yapmamaktadır. Default durumda derleyicilerin pek çoğunda optimizasyon seçenekleri kapalı durumdadır.

inline fonksiyonların derleyici tarafından derleme işlemi sırasında görülmesi gerekir. Aksi takdirde derleyici inline açılım yapamaz. Bu nedenle bu fonksiyonların başlık dosyalarına yerleştirilmesi uygun olur. inline fonksiyonlar static anahtar sözcüğü kullanılmamışsa external linkage’a sahiptir. Yani derleyiciler inline fonksiyonları “başka modül tarafından çağrılabilir” fikriyle object koda yazarlar.

Derleyiciler eğer fonksiyonu inline olarak açmazlarsa ona normal fonksiyon muamelesi yaparlar. Yani onu yine CALL makine komutuyla fonksiyon gibi çağırırlar. Bu durumda derleyicinin herhangi bir uyarı vermesi gerekmemektedir.

**C++’ın C’den Nesne Yönelimli Programlama Tekniği İle Doğrudan İlişkisi Olan Fazlalıkları ve Farklılıkları**

NYPT’nin en önemli yapı taşları sınıflardır. Bu nedenle kursun önemli bir bölümü sınıfların kullanımına ayrılmıştır. Tabii diğer bazı önemli konular da ele alınacaktır.

**Sınıflara Giriş**

Sınıflar, içerisinde verilerin (data) ve fonksiyonların bulunduğu veri yapılarıdır. Bunlar C’deki yapılara benzetilebilirler. Ancak onlardan en önemli farkları sınıfların fonksiyon da bulundurmasıdır. Sınıflar class anahtar sözcüğü kullanılarak aşağıdaki gibi bildirilirler:

class <sınıf ismi> {

[eleman bildirimleri]

};

Bir sınıf üç bölümden oluşmaktadır: public, protected ve private. Bir bölüm bölüm belirten anahtar sözcük ve ‘:’ atomu ile başlatılır. Diğer bir bölüm belirten anahtar sözcüğe kadar devam eder. Örneğin:

class Sample {

public:

// public bölüm

protected:

// protected bölüm

private:

// private bölüm

};

Bölüm belirten anahtar sözcükler sınıf içerisinde birden fazla kez kullanılabilirler. Bunların sırasının da hiçbir önemi yoktur. Örneğin:

class Sample {

public:

// public bölüm

private:

// private bölüm

public:

// public bölüm

private:

// private bölüm

};

Bir sınıf bildiriminde bu üç bölümün de bulunması gerekmez. Sınıf bildiriminde hiçbir bölüm belirtilmeden bildirime başlanabilir. Bu durumda default private bölüm anlaşılır. Örneğin:

class Sample {

// private bölüm

public:

// public bölüm

};

Sınıflardaki public, protected ve private anahtar sözcüklerine erişim belirleyicileri (access specifiers) denilmektedir.

C++’ta sınıf ile yapı tamamen aynı veri yapısını belirtir. Başka bir deyişle aslında yapılar da birer sınıftır. Ancak sınıflarla yapılar arasında küçük bir farklılık vardır: Sınıflarda default bölüm private iken yapılarda public biçimdedir. Örneğin:

class Sample {

// private bölüm

public:

// public bölüm

};

struct Sample {

// public bölüm

private:

// private bölüm

};

Kursumuzda ağırlıklı olarak sınıfsal temsil class bildirimi ile yapılacaktır. Ancak sınıflar için söylenen her şey yapılar için de geçerlidir.

Sınıf bildirimi içerisinde hem veri bildirimi (data’lar) hem de fonksiyon bildirimi yapılabilir. Sınıf içerisinde bildirilen elemanlara veri elemanlara (data members), fonksiyonlara ise “üye fonksiyonlar (member functions)” denilmektedir. Üye fonskiyonların yalnızca prototipleri sınıf bildirimi içerisinde bulundurulabilir. Kendileri dılarıda tanımlanabilir. Ya da üye fonksiyonlar doğrudan sınıf bildirimi içerisinde de tanımlanabilirler. Sınıf bildirimi içerisinde tanımlanan fonksiyonlar default olarak inline biçimdedir. Eğer fonksiyonun yalnızca prototipi sınıf bildirimi içerisinde bulundurulup kendisi dışarıda tanımlanacaksa (ki genellikle böyle yapılır) bu tanımlamanın genel biçimi şöyledir:

<geri dönüş değerinin türü> <sınıf ismi> :: <fonksiyon ismi> ([parametre bildirimi])

{

//...

}

Örneğin:

class Sample {

public:

int a;

int b;

void Foo(int x);

void Bar();

};

void Sample::Foo(int x)

{

//...

}

void Sample::Bar()

{

//...

}

:: operatörüne “çözünürlük operatörü (resolution operator)” denilmektedir. Yukarıda da belirtildiği gibi üye fonksiyonların tanımlaması sınıf bildirimi içerisinde de yapılabilirdi:

class Sample {

public:

int a;

int b;

void Foo(int x)

{

//...

}

void Bar()

{

//...

}

};

Tanımlaması sınıf bildiriminin içerisinde yapılan üye fonksiyonlar aynı zamanda inline fonksiyon kabul edilirler.

O halde C++’ta iki tür fonksiyon vardır:

1) Hiçbir sınıfın içerisinde olmayan fonksiyonlar (bunlara “global fonksiyonlar” denir)

2) Sınıfların içerisinde bulunan fonksiyonlar (bunlara “sınıfın üye fonksiyonlar” denir)

Sınıfların bölümlerinin (public, protected, private) ne anlam ifade ettiği daha ileride ele alınacaktır. Bu konu ele alınana kadar biz tüm bildirimleri sınıfın public bölümünde yapacağız.

**Sınıf Türünden Nesnelerin Tanımlanması**

Sınıflar da tıpkı C’deki yapılar gibi aynı zamanda bir tür belirtirler. Biz ilgili sınıf türünden nesneler tanımlayabiliriz. Örneğin:

#include <iostream>

using namespace std;

class Sample {

public:

int a;

int b;

void Foo();

void Bar();

};

void Sample::Foo()

{

cout << "Sample::Foo\n";

}

void Sample::Bar()

{

cout << "Sample::Bar\n";

}

int main()

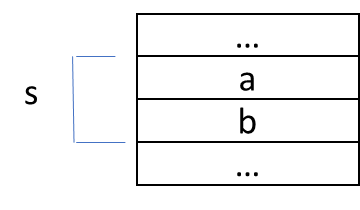
{

Sample s; // s Sample sınıfı türünden

return 0;

}

Bir sınıf türünden nesne tanımlandığında o nesne için sınıfın toplam veri elemanları kadar yer ayrılır. Sınıfın üye fonksiyonları nesnenin içerisinde bulunmazlar. Aslında üye fonksiyonlar yalnızca mantıksal bakımdan sınıfla ilişkilidir. Yukarıdaki örnekte Sample sınıfının int türden iki veri elemanı vardır: a ve b. Bunlar bellekte ardışıl bir blok oluşturacak biçimde nesnenin içerisinde bulunurlar.



Görüldüğü s bileşik bir nesnedir. İçerisinde a ve b parçaları vardır. Standartlara göre sınıfın iki erişim belirleyici anahtar sözcüğünün arasındaki elemanlar ilk yazılan düşük adreste bulunacak biçimde ardışıldır. Ancak bölümler arasındaki ardışıllık hakkında bir belirlemede bulunulmamıştır. Fakat ne olursa olsun derleyicilerin hemen hepsi ilk yazılan eleman düşük adreste olacak biçimde tüm elemanları yukarıdan aşağıya doğru bölümlere bakılmaksızın ardışıl dizmektedir. Tabii hizalamadan (alignment) dolayı elemanlar arasında kontrollü boşluklar bırakılabilmektedir.

Bir sınıf nesnesinin sizeof operatörü ile uzunluğunu alabiliriz. Yine onun toplam byte uzunluğunu elde etmiş oluruz.

Sınıfın veri elemanlarına tıpkı C’deki yapılarda olduğu gibi nokta operatörüyle erişilir. Örneğin:

Sample s;

s.a = 10;

s.b = 20;

**Üye Fonksiyonların Çağrılması**

Bir üye fonksiyon o sınıf türünden bir nesneyle nokta operatörü kullanılarak çağrılır. Örneğin:

#include <iostream>

using namespace std;

class Sample {

public:

int a;

int b;

void Foo();

void Bar();

};

void Sample::Foo()

{

cout << "Sample::Foo\n";

}

void Sample::Bar()

{

cout << "Sample::Bar\n";

}

void Foo()

{

cout << "Global Foo\n";

}

void Bar()

{

cout << "Global Bar\n";

}

int main()

{

Sample s;

Foo(); // Global Foo

Bar(); // Global Bar

s.Foo(); // Sample sınıfındaki Foo

s.Bar(); // Sample sınıfındaki Bar

return 0;

}

Sınıfların üye fonksiyonları da “overload” edilebilir. Yani bir sınıfta aynı isimli farklı parametrik yapılara sahip birden fazla fonksiyon bulunabilir. Şüphesiz farklı sınıflarda aynı isimli ve aynı parametrik yapıya sahip fonksiyonlar bulunabilir. Bunlar biribirlerine karışmazlar. Çünkü farklı sınıftadırlar.

**Sınıfın Elemanlarının Faaliyet Alanları**

C++’ta bir sınıfın içerisinde bildirilen elemanlar sınıf faaliyet alanına (class scope) sahiptir ve sınıfın tüm üye fonksiyonlarından ortak biçimde kullanılabilmektedir. Bir üye fonksiyon içerisinde sınıfın bir veri elemanı kullanıldığında bu veri elemanı o üye fonksiyon hangi nesneyle çağrılmışsa onun elemanıdır. Örneğin:

#include <iostream>

using namespace std;

class Sample {

public:

int a;

int b;

void Set();

void Disp();

};

void Sample::Set()

{

a = 10;

b = 20;

}

void Sample::Disp()

{

cout << a << endl;

cout << b << endl;

}

int main()

{

Sample s;

s.Set(); // Set'in içerisindeki a ve b s'nin a ve b elemanlarıdır

s.Disp(); // Disp'in içerisindeki a ve b s'nin a ve b elemanlarıdır

return 0;

}

Örneğin:

#include <iostream>

using namespace std;

class Sample {

public:

int a;

int b;

void Set(int x, int y);

void Disp();

};

void Sample::Set(int x, int y)

{

a = x;

b = y;

}

void Sample::Disp()

{

cout << a << endl;

cout << b << endl;

}

int main()

{

Sample s, k;

s.Set(10, 20); // Set'in içerisindeki a ve b s'nin a ve b elemanlarıdır

k.Set(30, 40); // Set'in içerisindeki a ve b k'nın a ve b elemanlarıdır

s.Disp(); // Disp'in içerisindeki a ve b s'nin a ve b elemanlarıdır

k.Disp(); // Disp'in içerisindeki a ve b k'nın a ve b elemanlarıdır

return 0;

}

Örneğin:

#include <iostream>

using namespace std;

struct Complex {

double real;

double imag;

void Set(double r, double i);

void Disp();

};

void Complex::Set(double r, double i)

{

real = r;

imag = i;

}

void Complex::Disp()

{

cout << real << "+" << imag << "i" << endl;

}

int main()

{

Complex z;

z.Set(2, 3);

z.Disp();

return 0;

}

**Kavramlar ve Sınıflar**

Dilde özel isimler dışındaki tüm sözcükler birer kavram belirtir. Kavramlar nesnelerin ortak özelliğinden hareketle türetilmişlerdir. Örneğin doğada Ağaç diye bir nesne yoktur. Ağaç bir kavramdır. Doğada birbirine benzeyen birtakım nesnelere ağaç denilmiştir. İnsan ile hayvan arasındaki en önemli farklılık kavram oluşturma ve bunun sonucu olarak da dili kullanma olgusudur.

İşte sınıflar da aslında kavramlara karşılık gelir. Bir Ağaç sınıfı bildirdiğimizde o bellekte yer kaplamaz. Yalnızca Ağaç kavramını betimler. Ağaç sınıfı türünden nesneler yarattığımızda bunlar gerçek ağaçlar olur. Sınıf türünden nesnelere İngilizce “instance (örnek)” de denilmektedir. Bu anlamda örnek o kavram türünden somut bir nesneyi belirtir.

Sınıfın veri elemanları kavramın özelliklerini oluşturur. Üye fonksiyonları ise işlevlerini oluşturmaktadır. Örneğin bir Doktor sınıfı yazacak olalım. Doktorun ismi, uzmanlık alanı, görev yeri vs. hep Doktor sınıfının veri elemanlarında saklanır. Doktor sınıfının üye fonksiyonları doktora iş yaptırır. Örneğin AmeliyataGir fonksiyonu doktora ameliyat yaptırır. HastaBak fonksiyonu ona hasta baktırır.

Bir proje nesne yönelimli olarak modellenirken önce projedeki tüm kavramlar birer sınıfla temsil edilir. Tabii, sınıflar tümden birbirlerinden bağımsız değildir. Sınıflar arasında birtakım ilişkiler de vardır. Bu ilişkiler de betimlendikten sonra bu sınıf türünden nesneler oluşturarak proeje gerçekleştirilir. Örneğin bir hastane otomasyonu yazacak olalım. Önce hastanedik tüm kavramlar birer sınıfla temsil edilir:

- Hastane

- Doktor

- Hemşire

- Hasta

- Hasta Bakıcı

- İlaç

- Yönetim

- Muhasebe

- Laboratuar

- Biyomedikal Aygıtlar

...

Örneğin bu hastanede 100 tane doktor çalışıyorsa bizim 100 tane Doktor nesnesi yaratıp onun özelliklirini (yani veri elemanlarını) set etmeliyiz.

**Sınıfların Başlangıç Fonksiyonları (Constructors)**

Bir sınıf nesnesi yaratıldığında derleyici tarafından otomatik olarak çağrılan üye fonksiyona “başlangıç fonksiyonu” ya da “yapıcı fonksiyon”, İngilizcesiyle “constructor” denilmektedir. Sınıfın sınıf ismiyle aynı isimli üye fonksiyonları başlangıç fonksiyonlarıdır. Başlangıç fonksiyonları da “overload” edilebilirler. Yani sınıfın farklı parametrik yapılara ilişkin birden fazla başlangıç fonksiyonu olabilir. Örneğin:

#include <iostream>

using namespace std;

class Sample {

public:

int a;

int b;

Sample(); // constructor

Sample(int x, int y); // constructor

void Foo();

void Bar();

};

Sample::Sample()

{

a = 0;

b = 0;

}

Sample::Sample(int x, int y)

{

a = x;

b = y;

}

void Sample::Foo()

{

//...

}

void Sample::Bar()

{

//i..

}

int main()

{

//...

return 0;

}

Başlangıç fonksiyonlarının geri dönüş değerleri diye bir kavramları yoktur. Bunlarda geri dönüş değeri türü yerine birşey yazılmaz. Yazılırsa hata oluşur.

Sınıfın parametresiz başlangıç fonksiyonuna “default başlangıç fonksiyonu (default constructor)” denilmektedir. Eğer nesne tanımlanırken nesne isminden sonra hiç parantez açılmazsa “default başlangıç fonksiyonu” çağrılır. Eğer parantez açılıp bir argüman listesi yazılırsa o argüman listesine uygun olan fonksiyon çağrılır. Örneğin:

Sample s; // default constructor

Sample k(10, 20); // int int parametreli constructor

Örneğin:

#include <iostream>

using namespace std;

class Sample {

public:

int a;

int b;

Sample(); // constructor

Sample(int x, int y); // constructor

void Disp();

};

Sample::Sample()

{

a = 0;

b = 0;

}

Sample::Sample(int x, int y)

{

a = x;

b = y;

}

void Sample::Disp()

{

cout << "a = " << a << ", " << "b = " << b << endl;

}

int main()

{

Sample s; // default constructor

Sample k(10, 20); // int int parametreli constructor

s.Disp(); // a = 0, b = 0

k.Disp(); // a = 10, b = 20

return 0;

}

Default başlangıç fonksiyonuyla nesne yaratırken isimden sonra parantez açılmadığına dikkat ediniz. Aşağıdaki bildirim fonksiyon prototipi anlamına gelmektedir:

Sample s();

Başlangıç fonksiyonlarının iki amacı vardır:

1) Sınıfın veri elemanlarına birtakım ilkdeğerleri vermek.

2) Nesne kullanılmadan önce gereken birtakım ilk işlemleri yapmak

Yerel bir sınıf nesnesi yaratıldığında ya da dinamik olarak new operatörüyle tahsis edildiğinde sınıfın veri elemanlarının içerisinde rastgele değerler vardır. Ancak global sınıf nesneleri tanımlandığında içerisinde default olarak 0 değeri bulunur. İşte biz bazen nesneyi tanımladığımız zaman bazı veri elemanlarında bazı değerlerin bulunmasını isteyebiliriz. Bazen nesne kullanılmadan önce birtakım işlemlerin de yapılması gerekebilmektedir. Örneğin bir dosyasının açılması, bir seri port edilmesi gibi. İşte bu işlemler de sınıfın başlangıç fonksiyonunda yapılmaktadır. Örneğin:

#include <iostream>

using namespace std;

class Date {

public:

int day, month, year;

Date();

Date(int d, int m, int y);

void Disp();

};

Date::Date()

{

day = 1;

month = 1;

year = 1900;

}

Date::Date(int d, int m, int y)

{

day = d;

month = m;

year = y;

}

void Date::Disp()

{

cout << day << "/" << month << "/" << year << endl;

}

int main()

{

Date d;

Date k(31, 1, 2017);

d.Disp();

k.Disp();

return 0;

}

C++ standartlarına göre eğer programcı bir sınıf için hiçbir başlangıç fonksiyonu yazmamışsa default başlangıç fonksiyonu derleyici tarafından içi boş olarak yazılır.

Yerel sınıf nesneleri için başlangıç fonksiyonları akış nesnenin tanımlandığı (yaratıldığı) noktaya geldiğinde çağrılır. Global sınıf nesneleri için başlangıç fonksiyonları ise akış daha main fonksiyonuna girmeden yukarıdan aşağıya doğru tanımlanma sırasına göre çağrılır.

Örneğin:

#include <iostream>

using namespace std;

class Sample {

public:

int a;

Sample(int x)

{

cout << "consrtuctor: " << x << endl;

}

};

Sample x(10);

int main()

{

cout << "main begins..." << endl;

Sample z(50);

return 0;

}

Sample y(20);

**Sınıfların Bitiş Fonksiyonları (Destructors)**

Bir sınıf nesnesi yok edilmeden hemen önce derleyici tarafından otomatik olarak çağrılan üye fonksiyona “bitiş fonksiyonu (destrtuctor)” denilmektedir. Bitiş fonksiyonlarının isimleri sınıf ismi ile aynıdır ancak başında ona yapışık olarak bir ~ karakteri bulunur. Örneğin:

#include <iostream>

using namespace std;

class Sample {

public:

int a;

int b;

Sample(); // default constructor

~Sample(); // destructor

void Foo();

};

Sample::Sample()

{

cout << "constructor\n";

}

Sample::~Sample()

{

cout << "destructor\n";

}

void Samp le::Foo()

{

cout << "Sample::Foo\n";

}

int main()

{

cout << "begins..\n";

Sample s; // constructor

s.Foo();

cout << "ends\n";

return 0;

}

Sınıfın bitiş fonksiyonları overload edilemez. Sınıfın tek bir bitiş fonksiyonu vardır. Bitiş fonksiyonlarının parametresi olamaz. Yerel sınıf nesneleri için bitiş fonksiyonları akış onların tanımlandığı bloktan çıkarken, global nesneler için bitiş fonksiyonları ise main fonksiyonu bittikten sonra çağrılır.

C++’ta ileride ele alınacak konuları da kapsayacak biçimde önemli bir kural vardır: Başlangıç fonksiyonlarıyla bitiş fonksiyonları birbirlerine göre her zaman ters sırada çağrılırlar. Örneğin:

{

Sample s, k;

//...

Sample z;

//...

}

Burada başlangıç fonksiyonlarının çağrılma sırası s, k, z biçimindedir. Bloktan çıkılırken bitiş fonksiyonları ters sırada yani z, k, s sırasında çağrılır. Örneğin:

Sample x, y;

int main()

{

Sample a, b;

//...

}

Burada başlangıç fonksiyonları x, y, a, b sırasında bitiş fonksiyonları da b, a, y, x sırasıyla çağrılır.

Bir sınıf için hiç bitiş fonksiyonu yazmamışsak derleycici bitiş fonksiyonunu içi boş biçimde bizim için yazar. Şüphesiz bizim her zaman bitiş fonksiyonu yazmamıza gerek yoktur. Ancak başlangıç fonksiyonlarında yapılan birtakım işlerin geri alınması gerektiği durumlarda bitiş fonksiyonlarından faydalanılabilir. Örneğin:

- Başlangıç fonksiyonunda bir dosya açılmıştır. Bitiş fonksiyonunda dosya kapatılabilir.

- Başlangıç fonksiyonunda new ile dinamik bir tahsisat yapılmıştır. Bitiş fonksiyonunda bu alan delete edilebilir.

Örneğin:

#include <cstdio>

#include <cstdlib>

#include <iostream>

using namespace std;

class File {

public:

File();

~File();

bool Open(const char \*path, const char \*mode);

void Type();

FILE \*f;

};

File::File()

{

f = NULL;

}

File::~File()

{

if (f != NULL)

fclose(f);

}

bool File::Open(const char \*path, const char \*mode)

{

f = fopen(path, mode);

return f != NULL;

}

void File::Type()

{

fseek(f, 0, SEEK\_SET);

int ch;

while ((ch = fgetc(f)) != EOF)

putchar(ch);

}

int main()

{

File f;

if (!f.Open("Sample.cpp", "r")) {

cerr << "cannot open file!\n";

exit(EXIT\_FAILURE);

}

f.Type();

return 0;

}

**Sınıfların Veri Elemanlarının İsimlendirilmesi**

Bir üye fonksiyona bakıldığında bir değişkenin sınıfın veri elemanı mı yoksa yerel bir değişken mi olduğunun kolay belirlenmesi kodun anlamlandırılmasını kolaylaştırır. Bu nedenle C++ programcıları sınıfın veri elemanlarını birtakım öneklerle ya da soneklerle isimlendirirler. Örneğin m\_ ya da d\_ önekleri bu anlamda çok kullanılmaktadır. Biz de kursumuzda bundan sonra okunabilirliği artırmak için sınıfın veri elemanlarını m\_ öneki ile başlayarak isimlendireceğiz.

Benzer biçimde pek çok programcı global değişkenler için de g\_ önekini kullanmaktadır. Ancak yerel ve parametre değişkenleri içinm herhangi bir önek kullanılmamaktadır.

**Sınıflarda Temel Erişim Kuralları**

Sınıflardaki temel erişim kuralları üç maddede özetlenebilir:

1) Sınıfın public bölümüne her yerden erişilebilir. Yani sınıfın dışından sınıfın public bölümündeki elemanlara o sınıf türünden bir nesne ya da gösterici yoluyla ya da sınıf ismi yoluyla erişebiliriz.

2) Sınıfın private bölümüne yalnızca o sınıfın içerisinden doğrudan erişilebilir. Sınıfın dışındaki bir fonksiyondan bu elemanlara erişemeyiz.

3) Sınıfların protected bölümüne o sınıfıntan ya da türemiş sınıflardan doğrudan erişilebilir. Ancak sınıfın dışından erişilemez. (protected bölüm türetme olduğunda anlam kazanmaktadır. Türetme olmadıktan sonra protected bölümün private bölümden bir farkı yoktur)

Yukarıdaki kuralları şöyle özetleyebiliriz:

1) Sınıfın dışından (yani sınıfın üye fonksiyonu olmayan bir fonksiyondan) nokta ya da ok operatörünü kullanrak bir sınıf nesnesi, referansı ya da göstericisi yoluyla sınıfın yalnızca public bölümüne erişilebilir. Benzer biçimde sınıf ismi ile sınııfn ancak public bölümündeki static elemanlarına erişebiliriz.

2) Sınıfın üye fonksiyonları doğrudan sınıfın her bölümüne erişebilir.

3) Türemiş sınıflar taban sınıfın public ve protected bölümlerine doğrudan erişebilirler. Ancak private bölümüne erişemezler. Türetme konusu ileride ele alınacaktır.

**NesneYönelimli Programlama Tekniği Nedir?**

NYPT’nin tek bir cümleyle tanımını yapmak zordur. Ancak yapmak gerekirse “sınıflar kullanarak program yazma tekniğidir” denilebilir. NYPT aslında bir grup anahtar kavramın toplamından oluşmaktadır. Bu kavramlar da aslında biribirlerinden bağımsız değildirler, iç içe geçmiştirler. Ancak bu anahtar kavramların hepsinin ortak amacı kodun algılanabilirliğini artırmaktır. Zaten bu teknik bilgisayarların donanımsal olarak güçlenmesiyle yazılımların satır sayıslarının artmasının yarattığı algısal karmaşıklı gidermek için ortaya çokmıştır.

**Kapsülleme (Encapsulation)**

Kapsülleme NYPT en önemli anahtar kavramlarından biridir. Bu kavram bir olgunun ya da kavramın bir sınıf ile temsil edilmesi, ancak onun dışarıyı ilgilendiren kısımlarının public bölüme yerleştirilerek açılması, iç işleyişe ilişkin kısımlarının ise private bölüme yerleştirilerek dışarıdan gizlenmesi anlamına gelir.

Bir sınıf için iki bakış açısı vardır: Sınıfı yazanların bakış açısı ve sınıfı kullananların bakış açısı. Sınıfı yazanlar sınıfın her bölümünü bilmek zorundadırlar. Ancak sınıfı kullananların yalnızca public bölümü (türetme söz konusu olduğunda protected bölümü de) bilmesi yeterlidir. Benzer biçimde bir sınıf kullanıcılar için dokümante edilecekse onun private bölümünün dokümante edilmesine hiç gerek yoktur. Yalnızca public bölümü (türetme söz konusuysa protected bölümü de) dokümante edilebilir.

Örneğin Thing isimli bir sınıf olsun. Bu sınıfın da dışarıdan kullanılabilecek DoSomethingImportant isimli bir üye fonksiyonu olsun. Bu fonksiyon işin belirli kısımlarını yapan Foo, Bar, Tar fonksiyonlarını çağırıyor olsun. O halde biz yalnızca DoSomethingImportant fonksiyonunu public bölüme yerleştirebiliriz. Diğer fonksiyonları sınıfın private bölümünde gizleyebiliriz.

#include <iostream>

using namespace std;

class Thing {

public:

void DoSomethingImportant();

private:

void Foo();

void Bar();

void Tar();

};

void Thing::DoSomethingImportant()

{

//...

Foo();

//...

Bar();

//...

Tar();

//...

}

int main()

{

Thing t;

t.DoSomethingImportant();

return 0;

}

**Sınıfın Veri Elemanlarının private Bölümde Gizlenmesi (Data Hiding)**

Sınıfın veri elemanlarıçoğu kez iç işleyişe ilişkindir. Bu nedenle sınıfın rpivate bölümünde gizlenir. Bu prensibe “veri elemanlarının gizlennmesi (data hiding)” denilmektedir. Ve bu prensip NYPT’nin anahtar kavramlarından biridir. Biz de bundan sonra veri elemanlarını private bölüme yerleştireceğiz.

Veri elemanları private bölüme yerleştirildiğinde artık onlara dışarıdan erişilemez. Onlara erişmek için public bölümde Get/Set fonksiyonları kullanılır. Yani onların içerisindeki değer GetXXX biçiminde bir fonksiyonla alınabilir, onlara değr SetXXX biçiminde bir fonksiyonla yerleitirilebilir. Örneğin:

#include <iostream>

using namespace std;

class Date {

public:

Date();

Date(int day, int month, int year);

void Disp();

int GetDay() { return m\_day; }

void SetDay(int day) { m\_day = day; }

int GetMonth() { return m\_month; }

void SetMonth(int month) { m\_month = month; }

int GetYear() { return m\_year; }

void SetYear(int year) { m\_year = year; }

private:

int m\_day, m\_month, m\_year;

};

Date::Date()

{

m\_day = 1;

m\_month = 1;

m\_year = 1900;

}

Date::Date(int day, int month, int year)

{

m\_day = day;

m\_month = month;

m\_year = year;

}

void Date::Disp()

{

cout << m\_day << "/" << m\_month << "/" << m\_year << endl;

}

int main()

{

Date date(3, 5, 2002);

cout << date.GetDay() << endl;

date.SetDay(6);

date.Disp();

return 0;

}

Pekiyi neden biz veri elemanlarını private bölüme yerleştirerek onlara public Get ve Set fonksiyonlarıyla erişmeye çalışıyoruz? Bunun birkaç nedeni vardır:

1) Deneyimler sınıfın veri elemanlarının isim ve tür bakımından çok sık değiştirildiğini göstermektedir. Örneğin yukarıdaki Date sınıfı pek çok projede kullanıldıktan sonra tarih bilgisi yazısal olarak tutulacak biçimde değiştirilmek istenebilir:

class Date {

public:

//...

private:

char m\_date[11]; // ”dd/mm/yyyy”

};

İşte böyle bir değişiklik yapıldığında eğer veri elemanları public bölümde bildirilmişse onları doğrudan kullanan önceden yazılmış kodlar geçersiz kalır. Fakat veri elemanları private bölüme yerleştirilip onlara public Get ve Set fonksiyonlarıyla erişilirse böyle bir değişiklik yapıldığında public fonksiyonların (Get ve Set fonksiyonları da dahil) prototipi aynı kalmak üzere onların içi değiştirilerek eski kodların geçerli kalması sağlanabilir. Örneğin:

#include <cstdio>

#include <iostream>

#include <cstring>

using namespace std;

class Date {

public:

Date();

Date(int day, int month, int year);

void Disp();

int GetDay() { return atoi(m\_date); }

void SetDay(int day)

{

sprintf(m\_date, "%02d", day); m\_date[2] = '/';

}

int GetMonth() { return atoi(m\_date + 3); }

void SetMonth(int month)

{

sprintf(m\_date + 3, "%02d", month); m\_date[5] = '/';

}

int GetYear() { return atoi(m\_date + 6); }

void SetYear(int year)

{

sprintf(m\_date + 6, "%04d", year);

}

private:

char m\_date[11];

};

Date::Date()

{

strcpy(m\_date, "01/01/1900");

}

Date::Date(int day, int month, int year)

{

sprintf(m\_date, "%02d/%02d/%04d", day, month, year);

}

void Date::Disp()

{

cout << m\_date << endl;

}

int main()

{

Date date(3, 5, 2002);

cout << date.GetDay() << endl;

date.SetDay(6);

date.Disp();

return 0;

}

O halde veri elemanlarını private bölüme yerleştirmenin birinci nedeni veri elemanlarının isim veya tür bakımından değişmesi durumunda onları kullanan kodların bundan etkilenmemesini sağlamaktır.

2) Eğer veri elemanlarını public bölüme yerleştirirsek onlara herhangi bir değer atanabilir. Halbuki bazen bvelli veri elemanlarının belli sınıurlarda değer alması gerekmektedir. Örneğin Date sınııfnda m\_day veri elemanına 60 gibi değeri atamak geçerli olmamlıdır. Ancak m\_day veri elemanı public bölümdeyse bunu kimse denetleyemez. Halbuki biz veri elemanlarını private bölüme yerleştirip onlara set fonksiyonu ile erişimi zorlarsak bu set fonksiyonlarının içerisinde sınır kontrolleri (validation) yapabiliriz.

3) Bazen veri elemanlrının arasında bir ilişki olabilir. Yani onlrdan bir tanesi set edildiğinde diğerlerinin ona uygun biçimde yeniden set edilmeleri gerekebilir. İşte veri elemanları public bölüme yerleştirilirse bu sorumluluk sınıf kullananlara bırakılır ki bu durumda kullanım zor olur. Halbuki veri elemanları private bölüme yerleştirilip onlara get ve set fonksiyonlarıyla erişilirse bu değişiklik bu fonksiyonların içerisinde gizlice yapılabilir.

4) Bazen bir veri elemanının get ya da set edildiği durumda başka birtakım işlemlerin yapılması gerekebilmektedir. Örneğin bazı bilgiler log dosylarına yazılabilir. Birtakım başka birimler üzerinde işlemler yapılabilir. İşte veri elemanları private bölüme yerleştirilip bunlara get vet fonksiyonlarıyla erişim zorlanıra bu get ve set fonksiyonlarının içerisinde söz konusu bu işlemler gizlice yapılabilir.

Fakat yine de veri yukarıdaki gerekçelerden hiçbiri geçerli olmayabilir. Bu durunda veri elemanları doğurdan public bölüme yerleştirilebilir. Ancak %90 veri elemanlarının private bölüme yerleştirilmesi daha iyi teknik olmaktadır.

**const Üye Fonksiyonlar**

C++’ta üye fonksiyonlar const olabilmektedir. Ancak global fonksiyonlar ve static üye fonksiyonlar (ileride görülecek) const olamazlar. const üye fonksiyonlar fonksiyonun parametre parantezinden sonra const anahtar sözcüğü ile belirtilirler. Örneğin:

class Sample {

public:

int Foo(int a) const;

//...

};

int Sample::Foo(int a) const

{

//...

}

const üye fonksiyonlarda const anahtar sözcüğü hem prototip bildiriminde hem de tanımlama sırasında belirtilmek zorundadır.

const üye fonksiyonlar içerisinde sınıfın veri elemanları kullanılabilir. Ancak onların değerleri değiştirilemez. Bir fonksiyon veri elemanlarını değiştirmiyorsa onun const yapılması iyi bir tekniktir. Çünkü const üye fonksiyonlar hem okunabilirliği artırır hem de derleyicinin bazı optimiazsyonları yapabilmesine zemin hazırlar. Örneğin sınıfların private veri elemanlarını get eden fonksiyonları, ekrana birtakım bilgiler yazdıran fonksiyonları vs. const olma eğilimindedir.

const bir üye fonksiyon dolaylı olarak da sınıfın veri elemanlarını değiştiremez. Böylece const üye fonksiyonlar sınıfın const olmayan üye fonksiyonlarını çağıramazlar (tabii bunun tersinde bir sorun yoktur. Yani const olmayan bir üye fonksiyon sınıfın const bir üye fonksiyonunu çağırabilir). Örneğin:

#include <iostream>

using namespace std;

class Sample {

public:

Sample();

void Foo() const;

void Bar();

private:

int m\_a;

int m\_b;

};

Sample::Sample()

{

m\_a = m\_b = 0;

}

void Sample::Foo() const

{

Bar(); // error!

}

void Sample::Bar()

{

m\_a = 10;

m\_b = 20;

}

int main()

{

Sample s;

//

s.Foo();

//

return 0;

}

Ayrıca const bir sınıf nesnesi ya da göstericisi yoluyla sınıfın ancak const üye fonksiyonalrı çağrılabilir. C++’ta sınıfların başlangıç ve bitiş fonksiyonları const üye fonksiyon yapılamazlar.

**const Sınıf Nesneleri**

Bir sınıf nesnesi const olabilir. Örneğin:

const Sample s;

const Sample k(10, 20);

const sınıf nesnesinin veri elemanları başlangıç ve bitiş fonksiyonlarının dışında herhangi bir biçimde değiştirilemezler. const bir sınıf nesnesi ya da göstericisi ile sınıfın const olmayan üye fonksiyonları çağrılamaz. Yalnızca const üye fonksiyonları çağrılabilir.

Veri elemanlarını değiştirmeyen fonksiyonların const yapılması iyi bir tekniktir. Aksi halde bunlar const nesneyle çağrılamazlar.

**Sınıflar Türünden Adresler ve Göstericiler**

Bir sınıf nesnesinin adresi alınabilir. Bu durumda tüm nesnenin başlangıç adresi elde edilir. Çünkü sınıf nesnelerinin içerisindeki veri elemanları (her ne kadar standartlar biraz esnek bırakmışsa da) ardışıl bir biçimde dizilmektedir.

Bir sınıf nesnesinin adresini aldığımızda onu ilgili sınıf türünden göstericilere atarız. Artık o gösterici o sınıf nesnesinin adresini tutuyor durumda olur. Örneğin Sample bir sınıf olsun:

Sample s;

Sample \*ps;

ps = &s;

ps bir sınıf türünden gösterici olmak üzere ps gösterisicisinin gösterdiği yerdeki nesnenin m\_a veri elemanına ps->m\_a ifadesiyle erişebiliriz. Tabii bu ifadenin aslında (\*ps).m\_a ile eşdeğer olduğunu biliyorsunuz. Benzer biçimde ps göstericisinin gösterdiği nesneyle ilgili sınıfın Foo üye fonksiyonunu ps->Foo() biçiminde çağırabiliriz. ps->Foo() ifadesinde Foo üye fonksiyonu içerisinde kullanılan veri elemanları ps göstericisinin gösterdiği yerdeki nesnenin veri elemanıdır.Dolayısıyla bu ifade de aslında (\*ps).Foo() ifadesiyle eşdeğerdir. Örneğin:

#include <iostream>

using namespace std;

class Sample {

public:

Sample(int a, int b)

{

m\_a = a;

m\_b = b;

}

void Disp() const;

private:

int m\_a;

int m\_b;

};

void Sample::Disp() const

{

cout << m\_a << ", " << m\_b << endl;

}

int main()

{

Sample s(10, 20);

Sample \*ps;

ps = &s;

ps->Disp();

(\*ps).Disp();

return 0;

}

**Sınıflar Türünden Referanslar**

Sınıf türünden referanslar da bildirilebilir. Tabii bildirim sırasında bu referanslara aynı sınıf türünden bir nesneyle ilkdeğer vermek gerekir. Örneğin:

int main()

{

Sample s(10, 20);

Sample &rs = s;

rs.Disp();

return 0;

}

Burada elemana erişimin ve üye fonksiyon çağırmanın nokta operatörüyle yapıldığına dikkat ediniz. Çünkü rs’yi bildirimden sonra kullandığımızda aslında rs’nin gösterdiği yerdeki nesnenin kendisi anlaşılmaktadır.

Tabii sınıf türünden göstericiler ve referanslar const da olabilirler. Bu durumda bu gösterici ya da referanslarla biz sınıfın ancak const üye fonksiyonlarını çağırabiliriz. Örneğin:

int main()

{

Sample s(10, 20);

const Sample &rs = s;

rs.Disp(); // geçerli, çünkü Disp const bir üye fonksiyon

return 0;

}

**Sınıf Nesnelerinin Fonksiyonlara Parametre Yoluyla Aktarılması**

Sınıf nesneleri de tıpkı C’deki yapılar gibi bileşik nesnelerdir. Onların fonksiyonlara aktarılması kopyalama yoluyla (call by value) ya da adres yoluyla (call by reference) yapılabilmektedir. Tabii adres yoluyla aktarım hemen her zaman tercih edilir. Çünkü işlem maliyeti düşüktür. Adres yoluyla aktarım C++’ta göstericilerle ya da referanslarla yapılabilir. Her iki biçim de aynı etkinliktedir. Programcı duruma göre bunlardan birini tercih eder.

Gösterici yoluyla aktarımda fonksiyonun parametre değişkeni sınıf türünden gösterici olur. Fonksiyon da aynı sınıf türünden bir nesnenin adresiyle çağrılır. Fonksiyon içerisinde biz sınıfın elemanlarını ok operatörüyle erişiriz. Örneğin:

#include <iostream>

using namespace std;

class Sample {

public:

Sample(int a, int b)

{

m\_a = a;

m\_b = b;

}

void Disp() const;

int getA() const { return m\_a; }

int getB() const { return m\_b; }

void SetA(int a) { m\_a = a; }

void SetB(int b) { m\_b = b; }

private:

int m\_a;

int m\_b;

};

void Sample::Disp() const

{

cout << m\_a << ", " << m\_b << endl;

}

void Foo(Sample \*ps)

{

ps->Disp();

ps->SetA(30);

ps->SetB(40);

}

int main()

{

Sample s(10, 20);

Foo(&s);

s.Disp(); // 30, 40

return 0;

}

Şimdi aynı işlemi referans yoluyla yapalım. Bu durumda fonksiyonun parametresi sınıf türünden referans olur. Biz de fonksiyonu aynı türden bir sınıf nesnesinin kendisiyle çağırırız. Fonksiyon içerisinde sınıfın elemanlarına nokta operatörüyle erişiriz:

#include <iostream>

using namespace std;

class Sample {

public:

Sample(int a, int b)

{

m\_a = a;

m\_b = b;

}

void Disp() const;

int getA() const { return m\_a; }

int getB() const { return m\_b; }

void SetA(int a) { m\_a = a; }

void SetB(int b) { m\_b = b; }

private:

int m\_a;

int m\_b;

};

void Sample::Disp() const

{

cout << m\_a << ", " << m\_b << endl;

}

void Foo(Sample &s)

{

s.Disp();

s.SetA(30);

s.SetB(40);

}

int main()

{

Sample s(10, 20);

Foo(s);

s.Disp(); // 30, 40

return 0;

}

Tabii bu tür durumlarda göstericiler ve referanslar const da olabilir. Özellikle sınıfın elemanlarında değişiklik yapma iddiası olamayan fonksiyon parametrelerinin const gösterici ya da referans yapılması iyi bir tekniktir.

Sınıflar Türünden Dinamik Bellek Tahsisatları

Sınıflar türünden new operatörüyle dinamik tahsisatlar yapılabilir. Bunlar daha sonra programcı tarafından uygun yerlerde delete operatörüyle yok edilmelidir. Pekiyi nesnelerin dinamik yaratılmasının ne faydası vardır? Bezen sınıf türünden nesnelerin belli bir noktada yaratılıp belli bir bölgede ve sürede kalıcı olmasını sieyebiliriz. new operatörüyle dinamik olarak yaratılan nesneler delete yapılana kadar (ya da program sonlanana kadar) bellekte kalmaktadır. Örneğin:

//…

{

//…

Sample s;

//…

} 🡪 nesne bu noktadaayok edilir

//…

Ancak örneğin:

Sample \*g\_ps;

//…

{

//…

g\_ps = new Sample();

//…

} 🡪 artık bloktan çıkıldığında da nesne yaşıyor durumda olacaktır

//…

new işleminin sınıf lar için genel biçimi şöyledir:

1) new <sınıf ismi>

2) new <sınıf ismi>([argüman listesi])

Örneğin:

Sample \*s;

s = new Sample; // birinci genel biçim

s = new Sample(10, 20); // ikinci genel biçim

new ile nesne yaratıldığında yine başlangıç fonksiyonu (constructor) çağrılmaktadır. Eğersınıf isminden sonra parantez kullanılmazsa nesne için default başlangıç fonksiyonu kullanılırsa argüman listesine uygun başlangıç fonksiyonu çağrılır. Aşağıdaki iki biçim arasında fark var mıdır?

s = new Sample;

s = new Sample();

Her iki çağrımda da sınıfın default başlangıç fonksiyonu çağrılır. Bu iki ifade arasında küçük bir farklılık vardır. Ancak şimdilik bu farklılık üzerinde önemsiz olduğu için durulmayacaktır.

new ile yaratılan dinamik nesne delete operatörü ile yok edilebilir. Örneğin:

delete s;

delete operatörü de sınıfın bitiş fonksiyonunu (destructor) çağırmaktadır. Örneğin:

#include <iostream>

using namespace std;

class Sample {

public:

Sample(int a, int b)

{

m\_a = a;

m\_b = b;

}

~Sample();

void Disp() const;

int getA() const { return m\_a; }

int getB() const { return m\_b; }

void SetA(int a) { m\_a = a; }

void SetB(int b) { m\_b = b; }

private:

int m\_a;

int m\_b;

};

void Sample::Disp() const

{

cout << m\_a << ", " << m\_b << endl;

}

Sample::~Sample()

{

cout << "destructor\n";

}

void Foo(Sample &s)

{

s.Disp();

s.SetA(30);

s.SetB(40);

}

int main()

{

Sample \*ps = new Sample(10, 20);

ps->Disp();

delete ps;

return 0;

}

Sınıf türünden diziler de söz konusu olabilir. Bu durumda dizi elemanları için baştan sona tek tek default başlangıç fonksiyonu çağrılmaktadır. Başka bir başlangıç fonksiyonunun çağrılması söz konusu olmaz. Dizinin faaliyet aalanı bittiğinde ters sırada her eleman için destructor çağrılmaktadır. Örneğin:

int main()

{

Sample ss[10]; // 10 eleman için de default constructor çağrılır

//...

return 0; // 10 eleman için de ters sırada destructor çaağrılır

}

Sınıf türünden dizilere C++’ta fazlaca gereksinim duyulmamaktadır. Benzer biçimde sınıf türünden dinamik diziler de oluşturulabilir. Örneğin:

int main()

{

Sample \*ps;

ps = new Sample[10]; // her eleman için default constructor çağrılır

//...

delete[] ps; // her eleman için destructor çağrılır

return 0;

}

Burada delete işleminde köşeli parantezlerin kullanıldığına dikkat ediniz.

**Sınıfların Dosya Organizasyonu**

Biz şimdiye kadar sınıfları her şeyiyle birlikte bir .cpp dosyası içerisine yazdık. Halbuki düzenli çalışma için genellikle programcılar bir sınıfı iki dosya bçiminde organize ederler. Sınıfın ismi X olmak üzere X.h ya da X.hpp dosyalarının içerisine sınıfın yalnızca bildirimi yerleştirilir. X.cpp içerisinde de üye fonksiyonların tanımlamaları bulunur. Böylece sınıfı kullanacak kaynak dosya bu X.h ya da X.hpp dosyasını include eder. Sınıfın kodları kütüphanelerden ya da X.cpp dosyasından link aşamasına dahiledilir. Başlık dosyasına include koruması yerleştirilmelidir. Örneğin:

#ifndef X\_HPP\_

#define X\_HPP\_

// Dosya içeriği

#endif

Örneğin:

// Complex.hpp

#ifndef COMPLEX\_HPP\_

#define COMPLEX\_HPP\_

class Complex {

public:

Complex();

Complex(double real, double imag);

void Disp() const;

private:

double m\_real;

double m\_imag;

};

#endif

// Complex.cpp

#include <iostream>

#include "Complex.hpp"

using namespace std;

Complex::Complex()

{}

Complex::Complex(double real, double imag)

{

m\_real = real;

m\_imag = imag;

}

void Complex::Disp() const

{

cout << m\_real << "+" << m\_imag << "i" << endl;

}

// Sample.cpp

#include <iostream>

#include "Complex.hpp"

using namespace std;

int main()

{

Complex z(10, 2);

z.Disp();

return 0;

}

Tabii bazen birden fazla sınıf için tek bir .hpp dosyası ve tek bir .cpp dosyası oluşturulabilir. Eğer sınıflar kütüphanelere yerleştirilmişse programcı yine başlık dosyalarını include ederek bu sınıfları kullanır.

**Sınıflar Arasındaki İlişkiler**

NYPT’de bir proje modellenirken projedeki tüm kavramlar sınıflarla temsil edilir. Sonra o sınıflar türünden nesneler yaratılır ve program bu nesneler kullanılarak yazılır. Fakat sınıflar arasında da birtakım ilişkiler vardır. Sınıflar arasındaki ilişkiler dört biçimde ifade edilebilir.

**İçerme İlişkisi (Composition):** Bir sınıf türünden nesne başka bir sınıf türünden nesnenin bir parçasını oluşturuyorsa bu iki sınıf arasında içerme ilişkisi vardır. Örneğin Araba sınıfı türünden bir araba nesnesi Motor sınıfı türünden bir Motor nesnesini içermektedir. İçerme ilişkisinin sağlanması gereken iki özelliği vardır:

1) İçerilen nesne tek bir nesne tarafından içerilmelidir.

2) İçerilen nesneyle içeren nesnenin ömürleri aynı olmalıdır.

Tabii her ilişkinin istisnaları olabilir. Burada tipik durumalra bakmak gerekir. İnsan sınıfı ile Karaciğer sınıfı arasında, Satranç tahtası ile Kare’ler arasında böyle bir ilişki vardır. Ancak Hastana ile Doktor sınıfları arasında içerme ilişkisi yoktur. Oda sınıfı ile Duvar sınıfı arasında da içerme ilişkisi yoktur. İçerme ilişkisine İngilizce “has a” ilişkisi de denilmektedir.

UML sınıf diyagraamlarında içerme ilişkisi içeren sınıf tarafında içi dolu bir baklavacıkla gösterilir. İçerme ilişkisi 1’e 1 olabileceği gibi 1’e 2, 1’e 3 ya da genel olarak 1’e N biçiminde olabilir. Örneğin İnsan sınıfı türünden bir nesne 2 tane Böbrek sınıfı türünden nesneyi içerebilir.

**2) Toplaşma İlişkisi (Aggregation):** Bu ilişki biçiminde bir sınıf nesnesi başka bir sınıf nesnesine bünyesine katarak kullanmaktadır. Ancak kullanılan nesne birden fazla nesne tarafından kullanılıyor olabilir. Ayrıca kullanan nesneyle kullanılan nesnenin ömürleri de aynı olmak zorunda değildir. Örneğin Hastane sınıf ile Doktor sınıfı arasında, Çiftlik sınıfı ile Koyun sınıfı arasında, Oda ile Duvar sınıfı arasında toplaşma ilişkisi vardır. Toplaşma ilişkisi de 1’e 1 olabileceği gibi 1’e N olabilir.

Toplaşma ilişkiisne İngilizece “holds a” ilişkisi de denilmektedir. UML sınıf diyagramlarında toplaşma ilişkisi nesneyi tutan sınıfta içi boş bir baklavacıkla gösterilir.

**3) Türetme İlişkisi (Inheritance):** Elimizde mevcut bir A sınıfı olsun. Biz onu bozmadan ona birşeyler eklemek isteyelim. İşte bir sınıfa ona dokunmadan ekleme yapma sürecine “türetme (derivation)” denilmektedir. Bu işlemde işlevi genişletilmek istenen asıl sınıfa “taban sınıf (base class)”, yeni özellikler eklenen sınıfa da “türemiş sınıf (derived class)” denir. Tüm bu kavrama da “kalıtım (inheritance)” denilmektedir. Türetme bir sınıfı genişletme anlamına gelir. Örneğin A sınıfının foo ve bar isimli iki fonksiyonu olsun. Biz bu sınıftan bir B sınıfı türetelim. Bu B sınıfına tar fonksiyonunu ekleyelim. Türemiş sınıf tamamen taban sınıf gibi de davranabilmektedir. Ancak fazlalıkkları vardır.

Türetme ilişkisine İngilizce “is a” ilişkisi de denir. UML sınıf diyagramlarında türetme ilişkisi türemiş sınıftan taban sınıfa açekilen içi boş bir okla gösterilmektedir.

Türetmenin en önemli amacı tekrarı engellemektir. Birden fazla sınıf aynı elemanlara sahipse tekrarı engellemek için ortak elemanlar bir taban sınıfta toplanır ve diğer sınıflar bu sınıftan türetilir.

Türetme ilişkisi için şu örnekler verilebilir:

- Bir işletmede çalışanların bütün ortak özellikleri Employee isimli bir sınıfta toplanabilir. Daha sonra bu sınıftan Worker, SalesPerson, Manager gibi sınıflar türetilebilir.

- Örneğin bir kargo şirketi taşıtları sınıflarla temsil ediyor olsun. Tüm taşıtların ortak birtakım özellikleri vardır. Böylece tüm taşıtların ortak özellikleri Vehicle isimli bir sınıfla temsil edilebilir. Bu sınıftan diğer taşıt sınıfları türetilebilir:

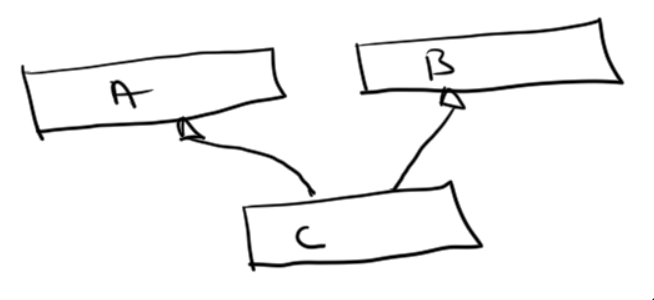
- Örneğin bir satranç programında satranç taşları sınıflarla temsil edilebilir. Tüm taşların ortak bazı özellikleri vardır. Örneğin rengi, hangi karede olduğu gbi. Bu bilgiler Figure isimli bir sınıfla temsil edilebilir. Diğer sınıflar bu sınıftan türetilebilir.

- Bir tetris oyununda tüm şekillerin ortak özellikleri vardır. Bu ortak özellikler Shape isimli bir sınıfta toplanabilir. Diğer sınıflar bu Shape sınıfından türetilebilirler.

Bir sınıfın tüm taban sınıflarına o sınıfın taban sınıfları (base classes) denir. Bir sınıfın hemen yukarısındaki taban sınıflara o sınıfın “doğrudan taban sınıfları (direct base classes)” denilmektedir. Bir sınıfın doğrudan taban sınıflarının taban sınıflarına ise “dolaylı taban sınıfları (indirect base classes)” denir. Örneğin:

Burada D’nin taban sınıfları C, B ve A’dır. D’nin doğrudan taban sınıfı C’dir, dolaylı taban sınıfları B ve A’dır.

Bir sanıfın biren fazla tban sınıfı olması durumuna çoklu türetme (multiple inheritance) denilmektedir. C++’ta çoklu türetme vardır. Ancak C# ve Java’da yoktur.



**4) Çağrışım İlişkisi (Association):** Bu ilişki biçiminde bir sınıf bir sınıfı kullanmaktadır. Ancak bu kullanma bünyesine katarak değil yüzeyseldir (yani örneğin tek bir fonksiyonda kullanma gibi). Örneğin Hastane ile Reklam sınıfı arasında, Taksi ile Müşteri arasında böyle bir ilişki vardır. Çağrışım ilişkisi kullanıla n sınıftan kullanan sınıfa çekilen ince bir okla temsil edilmektedir.

**C++’ta İçerme İlişkisinin (Composition) Gerçekleştirilmesi**

C++’ta içerme ilişkisini gerçekleştirmek için iki yöntem kullanılabilir:

1) İçeren sınıf içerilen sınıf türünden private bölümde bir veri elemanına sahiptir. Böylece içeren nesne yaratıldığında içerilen nesne de yaratılmış olur. Örneğin:

class Engine {

//...

};

class Automobile {

//...

private:

Engine m\_engine;

//...

};

Burada içerilen nesne (m\_engine) yalnızca tek bir nesne tarafından içerilmektedir. Automobile nesnesi yaratıldığında Engine nesnesi de yaratılmış olur. Automobile nesnesi yok edildiğinde m\_engine nesnesi de yok edilecektir.

2) İçeren sınıfının elemanı içerilen sınıf türünden bir göstericidir. İçeren sınıfın başlangıç fonksiyonunda içerilen nesne new operatörüyle yaratılır. İçeren nesnenin bitiş fonksiyonunda da nesne delete operatörüyle yok edilir. Örneğin:

class Engine {

//...

};

class Automobile {

public:

Automobile();

~Automobile();

//...

private:

Engine \*m\_engine;

};

Automobile::Automobile()

{

m\_engine = new Engine();

//...

}

Automobile::~Automobile()

{

delete m\_engine;

//...

}

İçeren sınıf türünden bir nesne yaratıldığında içeren sınıf için başlangıç fonksiyonu çağrılacaktır. Ancak içeren sınıfın başlangıç fonksiyonu içerilen sınıf nesnesi için o sınıfın başlangıç fonksiyonunu çağırır. Pekiyi bu çağırma tam olarak ne zaman yapılmaktadır? Derleyici içeren sınıfın başlangıç fonksiyonunun ana bloğunun başında gizli yerleştirdiği çağırma kodu yoluyla içerilen nesneler için başlangıç fonklsiyonlarını çağırır. Yani işlevsel olarak önce içerilen nesneler için sonra içeren nesneler için başlangıç fonksiyonları çalıştırılmış olur. Örneğin:

Automobile::Automobile()

{

----> Bu noktada m\_engine nesnesi için başlangıç fonksiyonu çağrılmış durumda

//...

}

Örneğin:

#include <iostream>

#include <cstdio>

using namespace std;

class A {

public:

A();

//...

};

class B {

public:

B();

//...

private:

A m\_a;

//...

};

A::A()

{

cout << "A constructor\n";

}

B::B()

{

cout << "B constructor\n";

}

int main()

{

B b;

return 0;

}

Pekiyi içerilen nesneye ilişkin sınıfın birden fazla başlangıç fonksiyonu varsa hangisi çağrılır? İşte elemana sahip sınıfın başlangıç fonksiyonlarının kapanış parantezinden sonra önce bir ‘:’ atomu sonra eleman listesi yazılırsa ve eleman isimlerinden sonra parantezler açılıp argüman listesi girilirse o argüman listesine uygun başlangıç fonksiyonları çağrılır. Örneğin:

#include <iostream>

#include <cstdio>

using namespace std;

class A {

public:

A();

A(int a);

//...

private:

int m\_a;

};

class B {

public:

B();

//...

private:

A m\_a;

//...

};

A::A()

{

cout << "A default constructor\n";

m\_a = 0;

}

A::A(int a)

{

cout << "A int constructor\n";

m\_a = a;

}

B::B() : m\_a(100) // m\_a için int parametreli constructor'ı çağır

{

cout << "B constructor\n";

}

int main()

{

B b;

return 0;

}

Bu sentaksa MIL (Member Initialization Syntax) denilmektedir. MIL sentaksının genel biçimine dikkat ediniz:

B::B([argüman listesi]) : m\_a([argüman listesi]) [, diğer eleman olan nesneler]

{

//...

}

MIL sentaksı yalnızca başlangıç fonksiyonlarında kullanılmaktadır. Eğer başka sınıf türünden eleman için MIL sentaksında bir belirleme yapılmadıysa bu durumda bu eleman için kendi sınıfıın default başlangıç fonksiyonu çağrılır. Başka bir deyişle:

B::B()

{

//...

}

ile,

B::B(...) : m\_a()

{

//...

}

eşdeğerdir. MIL sentaksında eleman için aregüman girildiğinde elemana ilişkin sınıfın tüm başlangıç fonksiyonları aday fonksiyonlardır. Bunlar arasından daha önce görmüş olduğumuz “overload resolution” kuralına göre en uygun olanlar ve en uygun fonksiyon seçilir.

Pekiyi sınıfın birden fazla başka sınıf türünden veri elemanı varsa bunlar hangi sırada çağrılırlar. Örneğin:

class D {

public:

D();

//...

private:

C m\_c;

B m\_b;

A m\_a;

};

D::D() : m\_a(), m\_b(10), m\_c(20)

{

//...

}

Elemanlar için başlangıç fonksiyonlarının çağrılma sırası konusunda MIL sentaksındaki sıranın hiçbir önemi yoktur. Her zaman bildirimdeki sıra dikkate alınır. Dolayısıyla yukarıdaki örnekte önce m\_c için, sonra, m\_b için sonra da m\_a için başlangıç fonksiyonları çağrılacaktır.

MIL sentaksı yalnızca bir sınıfın başka sınıf türünden veri elemanları için kullanılmaz. Aynı zamanda temel türden elemanları için de kullanılabilir. Örneğin:

#include <iostream>

#include <cstdio>

using namespace std;

class A {

public:

A();

A(int a);

//...

private:

int m\_a;

};

class B {

public:

B();

//...

private:

A m\_a;

int m\_b;

//...

};

A::A()

{

cout << "A default constructor\n";

m\_a = 0;

}

A::A(int a)

{

cout << "A int constructor\n";

m\_a = a;

}

B::B() : m\_a(100), m\_b(200)

{

cout << "B constructor\n";

}

int main()

{

B b;

return 0;

}

Tabii temel türden elemanlara ilkdeğerlerin MIL sentaksında verilmesinin özel bir etkisi yoktur. Bunlar için atama operatörüyle blok içerisinde de değer verilebilir. Ancak sıralama yine bildirimdeki sıraya göredir.

Pekiyi içerme ilişkisinde bitiş fonksiyonları nasıl çağrılmaktadır? C++’ta her zaman başlangıç fonksiyonlarıyla bitiş fonksiyonlarının çağrılma sırası terstir. Eleman için bitiş fonksiyonları elemana sahip sınıfın bitiş fonksiyonunun ana bloğunun sonunda derleyicinin yerleştirdiği gizli bir çağırma kodu yoluyla çağrılmaktadır. Sınıfın birden fazla elemanı varsa bunlar için bitiş fonksiyonları yine ters sırada çağrılacaktır. Böylece önce elemana sahip sınıf için bitriş fonksiyonu çağrılmış olur sonra eleman için bitiş fonksiyonları çağrılmış olur.

MIL sentaksında elemana sahip sınıfın başlangıç fonksiyonun parametreleri elemana ilişkin sınıf nesnesinin argümanı olarak kullanılabilir. Örneğin:

B::B(int a, int b) : m\_a(a), m\_b(b)

{

//...

}

**C++’ta Toplaşma (Aggregation) İlişkisinin Gerçekleştirilmesi**

C++’ta toplaşma ilişkisi tipik olarak şöyle gerçekleştirilir: Elemanı kullanan sınıfın elemana ilişkin sınıf türünden bir gösterici veri elemanı olur. Sınıfın ayrıca bu elemanı get/set yapan üye fonksiyonları da vardır. Tabii toplaşma ilişkisi 1’e çok ise bu durumda kullanan sınıfta bir container tutulması söz konusu olabilir. Örneğin:

#include <iostream>

#include <cstdio>

#include <cstring>

using namespace std;

class A {

public:

A(const char \*name);

void DoImportant();

//...

private:

char m\_name[32];

};

class B {

public:

B();

A \*getA() const { return m\_a; }

void setA(A \*a) { m\_a = a; }

void DoSomething();

//...

private:

A \*m\_a;

//...

};

A::A(const char \*name)

{

strcpy(m\_name, name);

}

void A::DoImportant()

{

cout << '\'' << m\_name << '\'' << " is doing an important thing" << endl;

}

B::B()

{

//...

}

void B::DoSomething()

{

m\_a->DoImportant();

}

int main()

{

B b;

B otherb;

//...

A a("Ali");

//...

b.setA(&a);

//...

otherb.setA(&a);

//...

b.DoSomething();

otherb.DoSomething();

return 0;

}

**C++’ta Türetme İlişkisinin (Inheritance) Gerçekleştirilmesi**

Bu konu ayrıntılı olduğu için ileride ayrı bir başlık halinde ele alınacaktır.

**C++’ta Çağrışım İlişkisinin (Association) Gerçekleştirilmesi**

Çağrışım ilişkisi yüzeysel bir ilişkidir. Genellikle bu ilişkide sınıfın belli bir üye fonksiyonu parametresiyle aldığı bir sınıf neesnesini kullanır. Örneğin:

class Client {

//...

};

class Taxi {

public:

//...

void TakeClient(const Client &client);

//...

};

**String Sınıfları**

Nesne Yönelimli Programlama Dillerinin kütüphanelerinde yazısal işlemleri yapan string sınıfları bulunur. Bu string sınıflarının görevi bir yazıyı alarak onun üzerinde çeşitli işlemleri yapmaktır. C++’ta da std isim alanında string isimli (gerçek ismi basic\_string) bir sınıf bulunmaktadır. Bu sınıf <string> başlık dosyası içerisinde bildirilmiştir. Dolayısıyla bu başlık dosyasının include edilmesi gerekir. C++’ın string sınıfı oldukça geniş özelliklere sahiptir. Biz henüz bazı konuları görmediğimiz için burada ondan yüzeysel olarak bahsedeceğiz.

Bir string nesnesi sınıfın başlangıç fonksiyonuyla oluşturulabilir. cout ile doğrudan ekrana yazdırılabilir:

#include <iostream>

#include <string>

using namespace std;

int main()

{

string s("This is a test");

cout << s << endl;

return 0;

}

Belli bir karakterden n tane olacak biçimde de bir string nesnesi oluşturulabilir:

#include <iostream>

#include <string>

using namespace std;

int main()

{

string s(20, 'x');

cout << s << endl;

return 0;

}

Bir yazının ilk n karakterinden bir yazı oluşturabiliriz:

#include <iostream>

#include <string>

using namespace std;

int main()

{

string s("This is test", 10);

cout << s << endl; // this is a te

return 0;

}

Nesnenin tuttuğu yazının uzunluğu size ya da length üye fonksiyonuyla elde edilebilir:

#include <iostream>

#include <string>

using namespace std;

int main()

{

string s("This is test");

cout << s.size() << endl; // 12

return 0;

}

string sınıfının köşeli parantez operatör fonksiyonu bize ilgili indeksteki karakteri vermektedir. Operatör fonksiyonları ileride ele alınacaktır. Örneğin:

#include <iostream>

#include <string>

using namespace std;

int main()

{

string s("This is a test");

for (int i = 0; i < s.size(); ++i)

cout << s[i] << endl;

return 0;

}

Bir string nesnesindeki yazının sonuna başka bir yazı eklemek için append üye fonksiyonu kullanılır. Örneğin:

#include <iostream>

#include <string>

using namespace std;

int main()

{

string s("This is a test");

cout << s << endl;

s.append(". Yes this is a test...");

cout << s << endl;

return 0;

}

string nesnesi içerisindeki yazının belli bir indeksine başka bir yazıyı insert edebiliriz. Örneğin:

#include <iostream>

#include <string>

using namespace std;

int main()

{

string s("anra");

cout << s << endl; // anra

s.insert(2, "ka");

cout << s << endl; // ankara

return 0;

}

Örneğin:

#include <iostream>

#include <string>

using namespace std;

int main()

{

string s("anra");

cout << s << endl; // anra

s.insert(2, "askana", 2, 2);

cout << s << endl; // ankara

return 0;

}

Bir yazının belli bir indeksinden itibaren n tane karakteri erase üye fonksiyonlarıyla silinebilir. Örneğin:

#include <iostream>

#include <string>

using namespace std;

int main()

{

string s("ankara");

cout << s << endl; // ankara

s.erase(2, 2);

cout << s << endl; // anra

return 0;

}

string sınıfının replace fonksiyonları belli bir indeksten itibaren yazının n karakterini silip onun yerine başka bir yazıyı insert eder. Örneğin:

#include <iostream>

#include <string>

using namespace std;

int main()

{

string s("ankara");

cout << s << endl; // ankara

s.replace(2, 2, "istanbul");

cout << s << endl; // anistanbulra

return 0;

}

C++’ın standart string sınıfından C tarzı sonu ‘\0’ ile biten bir yazı adresi elde etmek için c\_str fonksiyonu kullanılır. Elimizde bir string nesnesi varsa fakat bizden char \* ya da const char \* isteniyorsa bu fonksiyonu kullanmalıyız. Örneğin:

#include <iostream>

#include <cstdio>

#include <string>

using namespace std;

int main()

{

string s("ankara");

puts(s.c\_str());

return 0;

}

string sınıfının find fonksiyonları yazı içerisinde bir karakteri ararlar. Bulurlarsa onun indeksi ile geri dönerler. Karakter bulunamazsa fonksiyonlar string::npos değerine geri dönerler. Örneğin:

#include <iostream>

#include <string>

using namespace std;

int main()

{

string s("ankara");

size\_t index;

index = s.find('k');

if (index == string::npos)

cout << "character cannot be found!" << endl;

else

cout << "character found at " << index << " index" << endl;

return 0;

}

find fonksiyonuyla yazının içerisinde başka bir yazıyı da arayabiliriz. Örneğin:

#include <iostream>

#include <string>

using namespace std;

int main()

{

string s("bence ankara Turkiyen'in en sakin sehridir");

size\_t index;

index = s.find("ankara");

if (index == string::npos)

cout << "string cannot be found!" << endl;

else

cout << "string found at " << index << " index" << endl;

return 0;

}

find fonksiyonlarıyla arama belli bir indeksten başlanarak da yapılabilir. Örneğin:

#include <iostream>

#include <string>

using namespace std;

int main()

{

string s("bence ankara Turkiyen'in en sakin sehridir");

size\_t index;

index = s.find("ankara", 2);

if (index == string::npos)

cout << "character cannot be found!" << endl;

else

cout << "character found at " << index << " index" << endl;

return 0;

}

Örneğin:

#include <iostream>

#include <string>

using namespace std;

int main()

{

string s("ankara ankara guzel ankara. Seni gormek ister her bahti kara");

size\_t index = 0, n = 0;

while ((index = s.find("ankara", index)) != string::npos) {

++n;

++index;

}

cout << "found " << n << " times" << endl;

return 0;

}

string sınıfının rfind fonksiyonları find fonksiyonları gibidir. Ancak ilk bulunan yerin indeskini değil son bulunan yerin indeksini verirler.

string sınıfının substr fonksiyonu yazının belli bir kısmını alarak yeni bir string nesnesi olarak verir. Örneğin:

#include <iostream>

#include <string>

using namespace std;

int main()

{

string s("ankara");

string k;

k = s.substr(2, 3);

cout << k << endl; // kar

return 0;

}

string sınıfının + operatör fonksiyonu sayesinde biz iki string nesnesini + operatörüyle işlemee sokabiliriz. Bunun sonucu olarak yeni bir string nesnesi elde edilir. Bu string nesnesi iki nesnedeki yazıların birleştirilmesinden oluşmuştur. Örneğin:

#include <iostream>

#include <string>

using namespace std;

int main()

{

string s("ankara");

string k("izmir");

string m;

m = s + k;

cout << m << endl;

return 0;

}

Operatör fonksiyonları sonraki konularda ele alınacaktır.

Daha sonraki konularda görülecek olmakla birlikte string sınıfının kullanımı ile ilgili birkaç noktanın daha üzerinde duracağız. Biz bir string nesnesine başka bir string nesnesini ya da const char \* ya da char \* türünden bir adresi atayabiliriz. Bu durumda atanan string atanılan yazının kopyasını tutar. Örneğin:

#include <iostream>

#include <string>

using namespace std;

int main()

{

string s;

s = "ankara";

cout << s << endl;

return 0;

}

Yine C++’taki kullanıcı tanımlı tür dönüştürme kuralına göre biz bir const string nesnesi türünden referansı const char \* ya da char \* türünden bir adresle ilkdeğer vererek bildirebiliriz. Bu durumda yeni bir string nesnesi yaratılır ve o referans bu string nesnesinin adresini tutar hale gelir. Örneğin:

#include <iostream>

#include <string>

using namespace std;

int main()

{

const string &s = "ankara";

cout << s << endl;

return 0;

}

Böylece eğer bir fonksiyonun parametresi const string & ise biz o fonksiyonu iki tırnak ifadesi ile ya da char \*, const char \* türünden adreslerle çağırabiliriz. Örneğin:

#include <iostream>

#include <string>

using namespace std;

void Foo(const string &s)

{

cout << s << endl;

}

int main()

{

Foo("ali"); // geçerli

string s("veli");

Foo(s); // geçerli

return 0;

}

**C++’ta İsim Alanları (Namespace)**

Büyük projelerde programcı pek çok kütüphaneyi bir arada kullanabilmektedir. Böylesi durumlarda global alanda isim çakışmaları oluşabilir. Örneğin A ve B kurumlarının kütüphanelerini birlikte kullanıyor olalım. Bu iki kurum tesadüfen aynı isimli (Örneğin List) bir sınıf bildirmiş olsunlar. İşte bu durum derleme aşamasında error ile sonuçlanacaktır. İsim alanları ismi niteliklendirerek (yani adeta bir soyad gibi) onları diğerlerinden ayrıştırmak için kullanılmaktadır. Global bölge isim alanlarından oluşur. Bir isim alanı bildiriminin genel biçimi şöyledir:

namespace <isim>

{

//...

}

Bir isim alanının içerisindeki isme, isim alanı ismi ve değişken ismi :: operatörüyle birleştirilerek erişilir. Örneğin A::List, B::List gibi. Her kurumun kendi kütüphanelerinin kendine özgü bir isim alanında bulundurması iyi bir tekniktir. Örneğin Arcelik şirketinin kodlarının aşağıdaki gibi bir isim alanında bulundurulması C++ için iyi bir teknikitr:

namespace Arcelik

{

//...

}

Örneğin:

#include <iostream>

using namespace std;

namespace CSD

{

class Sample

{

//...

};

void Foo()

{

cout << "I am Foo" << endl;

}

}

int main()

{

CSD::Sample s;

CSD::Foo();

return 0;

}

İsim alanları iç içe bildirilebilir. Bu durumda dışarıdan içteki isim alanına erişebilmek için birden fazla :: operatörü kullanmak gerekir. Örneğin:

#include <iostream>

using namespace std;

namespace CSD

{

namespace Util

{

class Sample

{

//...

};

void Foo()

{

cout << "I am Foo" << endl;

}

}

}

int main()

{

CSD::Util::Sample s;

CSD::Util::Foo();

return 0;

}

Aslında isim alanlarının dışındaki global bölge de C++’ta bir isim alanı belirtir. Buraya “global isim alanı (global namespace)” denir. Global isim alanındaki isimlere tek operandlı :: operatörüyle erişilir. Örneğin:

#include <iostream>

using namespace std;

namespace CSD

{

namespace Util

{

class Sample

{

//...

};

void Foo()

{

cout << "I am Foo" << endl;

}

}

}

void Foo(void)

{

//...

}

int main()

{

CSD::Util::Sample s;

CSD::Util::Foo();

::Foo(); // Global Foo

return 0;

}

Bir isim alanı içerisindeki sınıfın üye fonksiyonun tanımlaması o sınıfı kapsayan isim alanlarının herhangi birinde yapılabilir. Ancak iç bir isim alanında yapılamaz. Örneğin:

#include <iostream>

using namespace std;

namespace CSD

{

namespace Util

{

class Sample

{

public:

void Foo();

void Bar();

void Tar();

//...

};

void Sample::Foo()

{

//...

}

}

void Util::Sample::Bar()

{

//...

}

}

void CSD::Util::Sample::Tar()

{

//...

}

int main()

{

CSD::Util::Sample s;

return 0;

}

Aynı isim alanı içerisinde aynı isimli birden fazla sınıf bildirilemez. Ancak farklı parametrik yapılara ilişkin birden fazla aynı isimli fonksiyon bildirilebilir. Tabii farklı isim alanlarının içerisinde tamamen aynı isimli ve parametrik yapıya sahip fonksiyonlar da bildirilebilirler. Örneğin:

#include <iostream>

using namespace std;

namespace CSD

{

namespace Util

{

void Foo()

{

//...

}

}

void Foo()

{

//...

}

}

int main()

{

CSD::Foo();

CSD::Util::Foo();

return 0;

}

Örneğin:

#include <iostream>

using namespace std;

namespace CSD

{

namespace Util

{

void Foo(int a)

{

cout << "Foo(int)" << endl;

}

void Foo(long a)

{

cout << "Foo(long)" << endl;

}

}

void Foo()

{

//...

}

}

int main()

{

CSD::Util::Foo(10);

CSD::Util::Foo(10L);

return 0;

}

C++’ın bütün sınıfları ve global fonksiyonları std isim alanında bildirilmiştir. Örneğin biz string sınıfını şöyle kullanmalıyız:

std::string s;

Bir isim alanının ikinci kez bildirilmesi error oluşturmaz. Ekleme yapma anlamına gelir. Örneğin:

#include <iostream>

using namespace std;

namespace CSD

{

void Foo()

{

//...

}

}

//...

namespace CSD // geçerli

{

void Bar()

{

//...

}

}

int main()

{

CSD::Foo();

CSD::Bar();

return 0;

}

Programcının std isim alanına bu biçimde ekleme yapması uygun değildir.

**using namspace Direktifi**

using namespace direktifi niteliklendirmeyi azaltmak için kullanılmaktadır. Bu direktifin genel biçimi şöyledir:

**using namespace <isim alanı ismi>;**

Bu direktif isim alanlarına, yerel bloklara yerleştirilebilir. Ancak sınıf bildirimleri içerisine yerleştirilemez.

Direktifte iki isim alanı söz konusudur: Direktifin yerleştirildiği isim alanı ve direktifte belirtilen isim alanı. Direktif şöyle bir etki gösterir: Direktifte belirtilen isim alanı ile direktifin yerleştirildidği isim alanını kapsayan en dar isim alanı bulunur. Sonra niteliksiz kullanılmış olan isimler bu isim alanında da aranır. (Ya da başka bir deyişle direktifte belirtilen isim alanı içerisindeki öğeler bu en dar isim alanına enjekte edilmektedir) bu en dar isim alanındaki Ancak bu süreç direktifin yerleştirildiği faaliyet alanında yürütülmektedir. Örneğin:

#include <iostream>

namespace CSD

{

void Foo()

{

//...

}

}

using namespace CSD;

int main()

{

Foo(); // geçerli

return 0;

}

Burada direktifin yerleştirildiği isim alanı global isim alanıdır. Direktifte belirtilen isim alanı da CSD isim alanıdır. İkisini de kapsayan en dar isim alanı CSD isim alanıdır. İşte Foo ismi artık bu isim alanında da aranacaktır. Bizim çağrıyı yaparken CSD:::Foo() ifadesini kullanmamıza gerek kalmaz. Tabii ilgili isim birden fazla yerde varsa error oluşur. Örneğin:

#include <iostream>

namespace CSD

{

void Foo()

{

//...

}

}

using namespace CSD;

void Foo()

{

//...

}

int main()

{

Foo(); // error

return 0;

}

Burada Foo’nun global Foo mu yoksa CSD isim alanındaki Foo mu olduğu anlaşaılamaz. Artık gerçekten niteliklendirme gerekir:

#include <iostream>

namespace CSD

{

void Foo()

{

//...

}

}

using namespace CSD;

void Foo()

{

//...

}

int main()

{

::Foo(); // global Foo

CSD::Foo(); // CSD::Foo

return 0;

}

Tabii bu örnekte using namespace direktifinin bir etkisi kalmamış durumdadır.

Overload resolution sırasında hem using direktifinin yerleştirildidği faaliyet alanındaki hem de direktifte belirtilen faaliyet alanındaki aynı isimli fonksiyonlar aday fonksiyonlardır. Bunların arasından uygun ve en uygun fonksiyon bulunur. Örneğin:

#include <iostream>

namespace CSD

{

void Foo()

{

//...

}

}

using namespace CSD;

void Foo(int a)

{

//...

}

int main()

{

Foo(); // CSD::Foo

return 0;

}

Birdfen fazla isim alanına using direktifi uygulandığında da benzer bir sorun oluşabilir. Örneğin:

#include <iostream>

namespace CSD

{

void Foo()

{

//...

}

}

using namespace CSD;

void Foo(int a)

{

//...

}

int main()

{

Foo(); // CSD::Foo

return 0;

}

#include <iostream>

namespace CSD

{

void Foo()

{

//...

}

}

namespace Audio

{

void Foo()

{

//...

}

void Foo(int a)

{

//...

}

}

using namespace CSD;

using namespace Audio;

int main()

{

Foo(); // error

Foo(10); // geçerli

return 0;

}

C++’ın standart sınıfları ve fonksiyonları std isim alanı içerisinde olduğu için bunlara kolay erişmek amacıyla C++ programcıları genellikle standart başlık dosyalarından sonra global isim alanına using namespace std direktifini yerleştirirler. Örneğin:

#include <iostream>

#include <string>

int main()

{

std::string s("Ali Serce");

std::cout << s << std::endl;

return 0;

}

Bu işlem şöyle kısaltılabilirdi:

#include <iostream>

#include <string>

using namespace std;

int main()

{

string s("Ali Serce");

cout << s << endl;

return 0;

}

Ancak using namespace direktiflerinin başlık dosyalarına yerleştirilmesi iyi bir teknik değildir. Çünkü bu durumda o dosyayı include eden programcılar kendi isteklerinin dışında derleyicinin başka isim alanlarına bakması durumuyla karşılaşırlar. Örneğin:

#ifndef PERSON\_HPP\_

#define PERSON\_HPP\_

#include <string>

class Person

{

public:

Person();

Person(const std::string &name, int no);

~Person();

std::string getName() const { return m\_name; }

int getNo() const { return m\_no; }

void Disp() const;

private:

std::string m\_name;

int m\_no;

};

#endif

using namespace direktifi geçişlidir. Örneğin:

#include <iostream>

#include <string>

using namespace std;

namespace A

{

class Sample

{

//...

};

}

namespace B

{

using namespace A;

//...

}

using namespace B;

int main()

{

Sample s; // geçerli

return 0;

}

**Sınıflar Türünden Geçici Nesnelerin Yaratılması (Temporary Object)**

X bir sınıf ismi olmak üzere X(...) ifadesi C++’ta “X sınıfı türünden geçici nesneyi girilen argümanlara uygun başlangıç fonksiyonuyla yarat” anlamına gelir. Bu biçimde yaratılan geçici nesneler geçici nesnenin kullanıldığı ifadenin sonunda derleyici tarafından bitiş fonksiyonları çağrılarak yok edilirler. Bir ifade içerisinde bu biçimde birden fazla geçici nesne yaratılmışsa bunların bitiş fonksiyonları ters sırada çağrılır.

**Türetme İşlemleri**

C++’ta türetme işleminin genel biçimi şöyledir:

class <türemiş sınıf ismi> : [türetme biçimi] <taban sınıf ismi> {

//...

};

Türetme biçimi public, protected ya da private olabilir. Ancak en çok kullanılan türetme public türetmesidir. Türetme biçimi bleirtilmemişse sınıflar için private yapılar için public türetmesi söz konusudur. (Örneğin Java ve C#’ta türetme biçimi yoktur. Oradaki türetme zaten default olarak public türetmesidir.) Örneğin:

class A {

};

class B : public A {

//...

};

Türetme işlemlerinde taban sınıfın elemanları sanki türemiş sınıfın elemanlarıymış gibi işlem görür. Örneğin:

#include <iostream>

using namespace std;

class A {

public:

void Foo();

void Bar();

int m\_x;

int m\_y;

};

class B : public A {

public:

void Tar();

int m\_z;

int m\_k;

};

void A::Foo()

{

cout << "A::Foo" << endl;

}

void A::Bar()

{

cout << "A::Bar" << endl;

}

void B::Tar()

{

cout << "B::Tar" << endl;

}

int main()

{

B b;

cout << sizeof(b) << endl;

b.Foo();

b.Bar();

b.Tar();

return 0;

}

Türemiş sınıf türünden bir nesne tanımlandığında bu nesne hem türemiş sınıfın veri elemanlarını hem de taban sınıfın veri elemanlarını tutar. Örneğin:

Türemiş sınıf nesnesi içerisinde taban ve türemiş sınıfın veri elemanları ardışıl blok halinde bulunurlar. Her ne kadar standartlara göre taban sınıfın veri elemanları mı yoksa türemiş sınıfın veri elemanaları mı daha önce geleceği derleyicileri yazanlara bırakılmışsa da hemen her zaman derleyiciler taban sınıf veri elemanları düşük adreste olacak biçimde yerleşimi yaparlar.

Türemiş sınıftan yeniden sınıf türetildiğinde yeni türetilen sınıf tüm taban sınıfların veri elemanlarını tutacaktır. Örneğin:

Bir sınıf birden fazla sınıfın taban sınıfı durumunda olabilir. Bu gayet normaldir. Örneğin:

Burada C sınıfı türünden bir nesne ile biz hem B’nin hem de A’nın elemanlarını kullanabiliriz. Ancak D sınıfı türünden bir nesneyle yalnızca D ve A’nın elemanlarını kullanabiliriz:

#include <iostream>

using namespace std;

class A {

public:

void Foo();

void Bar();

int m\_x;

int m\_y;

};

class B : public A {

public:

void Tar();

int m\_z;

int m\_k;

};

class C : public B {

public:

void Car();

int m\_k;

int m\_l;

};

class D : public A {

void Mar();

};

void A::Foo()

{

cout << "A::Foo" << endl;

}

void A::Bar()

{

cout << "A::Bar" << endl;

}

void B::Tar()

{

cout << "B::Tar" << endl;

}

void C::Car()

{

cout << "C::Car" << endl;

}

void D::Mar()

{

cout << "D::Mar" << endl;

}

int main()

{

C c;

c.Foo();

c.Bar();

c.Tar();

c.Car();

D d;

d.Foo();

d.Bar();

return 0;

}

Bir sınıfın birden fazla taban sınıfa sahip olması durumuna “çoklu türetme (multiple inheritance)” denilmektedir. Çoklu türetme C# ve Java’da yoktur. Örneğin Hava Taşıtları bir sınıfla Deniz Taşıtları başka bir sınıfla temsil edilmiş olabilir. Havada ve Denizde Giden taşıtlar bu iki sınıftan çoklu türetilerek oluşturulabilirler.

**Türemiş Sınıflarda Erişim Kuralları**

Türemiş sınıflarda erişim kuralları türetme biçimine bağlı olarak değişmektedir.

**public Türetmesi Durumunda Erişim Kuralları**

Bu türetme biçiminde taban sınıfın public bölümü türemiş sınıfın public bölümüymüş gibi, taban sınıfın protected bölümü türemiş sınıfın protected bölümüymüş gibi işlem görür. Taban sınıfın private bölümü tamamen korunmuştur. Türemiş sınıf tarafından erişilemez.

Bu türetme biçiminden üç sonuç çıkartılabilir:

1) Dışarıdan (yani sınıfın dışındaki bir fonksiyondan) Türemiş sınıf nesnesi, referansı ya da göstericisi yoluyla taban ve türemiş sınıfın yalnızca public bölümlerine erişilebilir.

2) Sınıfın içerisinden hem taban sınıfın hem de türemiş sınıfın public ve protected bölümlerine erişilebilir.

3) Taban sınıfın private bölümü tamamen korunmuştur. Türemiş sınıf tarafından erişilemez.

**protected Türetmesi Durumunda Erişim Kuralları**

Bu türetme biçiminde taban sınııfn public ve protected bölümleri türemiş sınıfın protected bölümüymüş gibi işlem görür. Taban sınıfın private bölümü tamamen korunmuştur. Türemiş sınıf tarafından erişilemez.

protected türetmesinden çıkan sonuçlar şunlardır:

1) Türemiş sınıf nesnesi ya da göstericisi yoluyla dışarıdan (türemiş sınıfın dışından) yalnızca türemiş sınıfın public bölümüne erişilebilir. Taban sınıfın public bölümüne erişilemez.

2) Sınıfın içerisinden hem taban sınıfın hem de türemiş sınıfın public ve protected bölümlerine erişilebilir.

3) Taban sınıfın private bölümü tamamen korunmuştur. Türemiş sınıf tarafından erişilemez.

**private Türetmesi Durumunda Erişim Kuralları**

Bu türetme biçiminde taban sınıfın public ve protected bölümleri türemiş sınıfın private bölümü gibi ele alınmaktadır.

private türetmesinden çıkan sonuçlar protected türetmesinden çıkan sonuçlarla aynıdır. Ancak bir dizi türetme yapıldığında arada farklılık oluşur. Örneğin:

Burada B sınıfı protected türetmesiyle A’dan türetildiğinde biz C’nin üye fonksiyonları içerisinde A’nın protected elemanlarını kullanabiliriz. Ancak B sınıfı A’dan private türetmesiyle türetildiğinde artık biz C sınıfının üye fonksiyonları içerisinde A sınıfının protected elemanalarını kullanamayız.

Bir dizi türetme yapıldığında aynı kuralların yukarıdan aşağıya doğru uygulandığına dikkat ediniz. Örneğin:

Burada biz D sınıfı türünden bir nesne ya da gösterici yoluyla dışarıdan A’nın, B2nin C’nin ve D’nin public bölümlerine erişebiliriz. Benzer biçimde D sınıfının bir üye fonksiyonu içerisinde biz A’nın B’nin, C’nin public ve protected bölümlerine doğrudan erişebiliriz. Ancak A’nın, B’nin ve C’nin private bmlümlerine hiçbir biçimde erişemeyiz.

**protected Bölümün Anlamı**

Sınıfın protected bölümü türemiş sınıf tarafında erişilen bölümdür. public bölüme her sınıf tarafından erişilebilir. private bölüme ise yalnızca kendi sınıf erişebilmektedir. O halde biz bir sınıf tasarlarken eğer birtakım elemanların sınıfımızdan türetme yapan sınıf tarafından erişilmesini istiyorsak onları protected bölüme yerleştirmeliyiz. Sınıfın en korunmuş bölümü private bölümdür. En az korunmuş bölümü (hiç korunmamış bölümü) public bölümdür. protected bölüm ikisinin arasında düşünülebilir.

**Türetme Durumunda Başlangıç ve Bitiş Fonksiyonlarının Çağrılması**

Türetme durumunda türemiş sınıf türünden bir nesne yaratıldığında türemiş sınıfın başlangıç fonksiyonu çağrılır. Türemiş sınıfın başlangıç fonksiyonu yalnızca kendi sınıfının private bölümüne erişebilmektedir. Taban sınıfın veri elemanları muhtemelen o sınıfın private bölümünde olacağından türemiş sınıfın başlangıç fonksiyonu taban sınıfın veri elemanlarına değer atayamaz. Pekiyi nesnenin taban sınıf kısmı nasıl ilkdeğer alacaktır? İşte türemiş sınıfın başlangıç fonksiyonun ana bloğunun başında derleyici tarafından yerleştirilmiş gizli bir çağırma kodu yoluyla taban sınıfın başlangıç fonksiyonu çağrılmaktadır. Taban sınıfın hangi başlangıç fonksiyonun çağrılacağı MIL sentaksıyla belirtilir. Örneğin B sınıfı A sınıfından türetilmiş olsun:

B::B(...) : A(...)

{

//...

}

Yani işlevsel olarak önce taban sınıfın başlangıç fonksiyonu sonra türemiş sınıfın başlangıç fonksiyonu çalıştırılmış olur. MIL sentaksında yalnızca sınıfın doğrudan taban sınıfı için belirleme yapılmaktadır. Bir dizi türetme söz konusu olduğunda her sınıf kendi doğrudan taban sınıfının başlangıç fonksiyonunu çağıracağından çağırma işlemi yukarıdan aşağıya doğru olur. Eğer taban sınıf ismi MIL sentaksında hiç belirtilmezse bu durumda taban sınıf için onun default başlangıç fonksiyonu çağrılır. Yani örneğin:

B::B(...)

{

//...

}

İle,

B::B(...) : A()

{

//...

}

aynı anlamdadır.

Çoklu türetmede başlangıç fonksiyonlarının çağrılma sırası yine MIL sentaksındaki sıraya göre değil, bildirimdeki sıraya göre yapılmaktadır. Örneğin:

class C : public A, public B {

//...

};

//...

C::C(...) : B(..), A(..)

{

//...

}

Burada önce A sınıfının başlangıç fonksiyonu sonra B sınıfının başlangıç fonksiyonu çağrılacaktır.

Sınıfın hem bir sınıftan türetildiğini hem de başka sınıf türünden veri elemanları olduğunu düşünelim. Bu durumda önce her zaman taban sınıfların başlangıç fonksiyonları bildirimdeki sıraya göre çağrılır. Sonra sınıfın başka sınıf türünden veri elemanları için başlangıç fonksiyonları bildirimdeki sıraya göre çağrılır. Örneğin yukarıdaki C sınıfı şöyle bildirilmiş olsun:

class C : public A, public B {

//...

private:

D m\_d;

E m\_e;

};

//...

C::C(..) : B(...), m\_e(...), m\_d(...), A(...)

{

//...

}

Burada önce A sınıfı için, sonra B sınıfı için sonra m\_d için sonra da m\_e için başlangıç fonksiyonları çağrılacaktır.

Anımsanacağı gibi C++’ta her zaman başlangıç fonksiyonlarıyla bitiş fonksiyonları ters sırada çağrılmaktadır. İşte türemiş sınıfın bitiş fonksiyonunun ana bloğunun sonunda derleyici tarafından yerleştirilen gizli bir çağırma kodu yoluyla önce elemanlar için sonra taban sınıflar için ters sırada bitiş fonksiyonları çağrılacaktır. Yukarıdaki örnekte önce önce m\_e için sonra m\_d için sonra da sırasıyla B ve A sınıfları için bitiş fonksiyonları C sınıfın bitiş fonksiyonun ana bloğunun sonunda çağrılmaktadır.

Örneğin:

// Date.hpp

#pragma once

class Date

{

public:

Date();

Date(int day, int month, int year) :

m\_day(day), m\_month(month), m\_year(year)

{}

~Date();

int Day() const { return m\_day; }

void setDay(int day) { m\_day = day; }

int Month() const { return m\_month; }

void setMonth(int month) { m\_month = month; }

int Year() const { return m\_year; }

void setYear(int year) { m\_year = year; }

void Disp() const;

private:

int m\_day, m\_month, m\_year;

};

//Date.cpp

#include <iostream>

#include "Date.hpp"

using namespace std;

Date::Date()

{

}

Date::~Date()

{

}

void Date::Disp() const

{

cout << m\_day << "/" << m\_month << "/" << m\_year << endl;

}

// Employee.hpp

#pragma once

#include <string>

#include "Date.hpp"

class Employee

{

public:

Employee();

Employee(const std::string &name, const std::string department,

const Date &bdate) : m\_name(name), m\_department(department),

m\_bdate(bdate)

{}

~Employee();

std::string Name() { return m\_name; }

void SetName(const std::string &name) { m\_name = name; }

std::string Department() { return m\_department; }

void SetDepartment(const std::string &department) { m\_department = department; }

Date BDate() { return m\_bdate; }

void SetBDate(const Date &bdate) { m\_bdate = bdate; }

void Disp() const;

private:

std::string m\_name;

std::string m\_department;

Date m\_bdate;

};

// Employee.cpp

#include <iostream>

#include "Employee.hpp"

using namespace std;

Employee::Employee()

{

}

Employee::~Employee()

{

}

void Employee::Disp() const

{

cout << m\_name << ", " << m\_department << ", " <<

m\_bdate.Day() << "/" << m\_bdate.Month() << "/" << m\_bdate.Year() << endl;

}

// Worker.hpp

#pragma once

#include <string>

#include "Employee.hpp"

class Worker : public Employee

{

public:

Worker();

Worker(int weekHours, int hourWage) : m\_weekHours(weekHours), m\_hourWage(m\_hourWage)

{}

Worker(std::string name, std::string department, const Date &bdate, int weekHours, int hourWage)

: Employee(name, department, bdate), m\_weekHours(weekHours), m\_hourWage(hourWage)

{}

int WeekHours() const { return m\_weekHours; }

void SetWeekHours(int weekHours) { m\_weekHours = weekHours; }

int HourWage() const { return m\_hourWage; }

void SetHourWage(int hourWage) { m\_hourWage = hourWage; }

void Disp() const;

~Worker();

private:

int m\_weekHours;

int m\_hourWage;

};

// Worker.cpp

#include <iostream>

#include "Worker.hpp"

using namespace std;

Worker::Worker()

{

}

Worker::~Worker()

{

}

void Worker::Disp() const

{

Employee::Disp();

cout << m\_weekHours << ", " << m\_hourWage << endl;

}

// Main.cpp

#include <iostream>

#include "Employee.hpp"

#include "Worker.hpp"

using namespace std;

int main()

{

Worker w("Sacit Apaydin", "Uretim", Date(3, 5, 1970), 40, 30);

w.Disp();

return 0;

}

Bir dizi türetme durumunda taban sınıfların çok fazla set edilecek elemanları olabilir. Bunların hepsinin başlangıç fonksiyonlarında set edilmesi iyi bir teknik değildir. Başlangıç fonksiyonlarında en önemli elemanlar set edilebilir. Diğerleri nesne yaratıldıktan sonra ayrıca set fonksiyonlarıyla set edilebilir. Örneğin:

// Main.cpp

#include <iostream>

#include "Employee.hpp"

#include "Worker.hpp"

using namespace std;

int main()

{

Worker w;

w.SetName("Sacit Apaydin");

w.SetDepartment("Uretim");

w.SetBDate(Date(3, 5, 1970));

w.SetWeekHours(40);

w.SetHourWage(30);

w.Disp();

return 0;

}

**Sınıfın static Üye Fonksiyonları**

C++’ta sınıfın üye fonksiyonları static olabilir ya da olmayabilir. Biz şimdiye kadar static olmayan üye fonksiyonları kullandık. Bir üye fonksiyonu static yapmak için prototip bildiriminde fonksiyon bildiriminin önüne static anahtar sözcüğü getirilir. static anahtar sözcüğü tanımlama sırasında (eğer tanımlama sınıfın dışında yapılıyorsa) kullanılmaz. Örneğin:

class Sample {

public:

Sample()

{}

Sample(int a, int b);

void Foo(); // static olmayan üye fonksiyon

static void Bar(); // static üye fonksiyon

private:

int m\_a;

int m\_b;

};

Sample::Sample(int a, int b)

{

m\_a = a;

m\_b = b;

}

void Sample::Foo()

{

//...

}

void Sample::Bar()

{

//...

}

Sınıfın static üye fonksiyonları sınıfın static olmayan elemanlarını (yani static olmayan fonksiyonlarını ve veri elemanlarını) doğrudan kullanamaz. Sınıfın static üye fonksiyonları adeta global fonksiyon olmaya adaydır. Ancak mantıksal bakımından o sınıfla ilgili olduğu için sınıfın içine yerleştirilmiştir. Örneğin Date sınıfının içerisinde verilen bir değerinm artık yıl olup olmadığını veren bir fonksiyon olsun. Bu fonksiyon sınıfın aslında hiçbir elemanını kullanmaz. Bu durumda fonksiyonun static yapılması uygun olur. Örneğin:

class Date {

public:

Date()

{}

Date(int day, int month, int year) : m\_day(day), m\_month(month), m\_year(year)

{}

static bool IsLeap(int year);

//...

private:

int m\_day;

int m\_month;

int m\_year;

};

Sınıfın static üye fonksiyonları sınıfın static veri elemanlarını ve static üye fonksiyonlarını kullanabilir.

Sınıfın static üye fonksiyonları bir nesne ile ya da bir gösterici ile çağrılabilir. Ancak static üye fonksiyonlar sınıfın veri elemanlarını kullanamadığına göre bu çağrım yanlış anlaşılmalara yol açabilmektedir. Static üye fonksiyonlar eğer public bölümdeyse sınıf ismi ve çözünürlük operatörüyle de kullanılabilirler. Örneğin:

int main()

{

Sample s;

s.Bar(); // geçerli, fakat kötü teknik

Sample::Bar(); // geçerli iyi teknik

return 0;

}

**Sınıfın static Veri Elemanları**

Tıpkı üye fonksiyonlarda olduğu gibi veri elemanları da static olabilir ya da olmayabilir. Biz şimdiye kadar static olmayan veri elemanlarını gördük. Bir veri elemanını static yapmak için bildirimde static anahtar sözcüğü kullanılır. Ancak static veri elemanları ayrıca sınıfın dışında da sınıf ismi belirtilerek tanımlanmalıdır. Örneğin:

class Sample {

public:

Sample()

{}

Sample(int a, int b);

void Foo(); // static olmayan üye fonksiyon

static void Bar(); // static üye fonksiyon

private:

int m\_a;

int m\_b;

static int m\_c;

};

int Sample::m\_c; // Sınıfın static veri elemanının dışarıda tanımlanması

Sample::Sample(int a, int b)

{

m\_a = a;

m\_b = b;

}

void Sample::Foo()

{

//...

}

void Sample::Bar()

{

//...

}

Sınıfın static veri elemanları nesne içerisinde yer kaplamaz. Bunlardan yalnızca toplamda tek bir kopya vardır ve o sınıf türünden nesne yaratılsa da yaratılmasa da bu kopya kullanılabilir.

Sınıfın static veri elemanları da bir nesne ya da gösterici yoluyla dışarıdan kullanılabilirler. Ancak static veri elemanları nesnenin içerisinde yer kaplamadığı için bu kullanım iyi bir teknik değildir. Sınıfın static elemanları eğer public bölümdeyse sınıf ismi ve çözürlük operatörüyle kullanılabilir. Örneğin:

int main()

{

Sample::m\_c = 10;

cout << Sample::m\_c << endl;

return 0;

}

**Static’lik Bakımından Sınıfta Kim Kimi Kullanabilir?**

Sınıfın static fonksiyonları doğrudan kendi sınıflarının static fonksiyonlarını çağırabilir ve static veri elemanlarını kullanabilir. Ancak static olmayan fonksiyonları çağırmaz ve static olmayan veri elemanlarını kullanamaz.

Sınıfın static olmayan fonksiyonları sınıfın hem static olmayan fonksiyonlarını hem de static fonksiyonlarını doğrudan çağırabilir ve hem static olmayan veri elemanlarını hem de static olan veri elemanlarını doğrudan kullanabilir.

Özetle sınıfın static olmayan fonksiyonları sınıfın hem static elemanlarını hem de static olamayan elemanlarını doğrudan kullanabilmektedir. Ancak static fonksiyonları sınıfın yalnızca static elemanlarını doğrudan kullanabilmektedir.

**Üye Fonksiyonlar ve Veri Elemanları Ne zaman static Olmalı Ne Zaman Olmamalı?**

Bir sınıf nesnesi aslında sınıfın static olmayan veri elemanlarının toplamından oluşmaktadır. Eğer biz bir veri elemanının her nesne için ayrıca onun bir parçası olmasını istiyorsak (ki genellikle isteriz) bu durumda o veri elemanı static yapılmamalıdır. Örneğin her ağacın yaprakları vardır. Her tarihin gün, ay, yıl bileşeni vardır. Her insanın karaciğeri vardır. Ancak eğer ilgili elemandan toplamda tek bir kopya olmasını istiyorsak bu durumda veri elemanını static yapmalıyız.

Bir üye fonksiyon sınıfın static olmayan bir elemanını doğrudan kullanıyorsa zaten onu static yapamayız. Ancak kullanmıyorsa teorik olarak onu static de yapabiliriz static olmayan da yapabiliriz. Ancak sınıfın hiçbir static olmayan elemanını kullanmayan fonksiyonların static yapılması doğru tekniktir. Çünkü aksi halde o fonksiyon gerekmediği halde bir nesneyle ya da gösterici ile çağrılmak zorunda olur.

Bir teorem olarak şunu söyleyebiliriz: İyi teknik uygulanmışsa bir static olmayan fonksiyon eninde sonunda sınıfın static olmayan bir veri elemanını doğrudan ya da dolaylı olarak kullanıyor olacaktır. Aksi takdirde bu fonksiyon sınıf nesnesini toplamda kullanmaış olacağı için zaten onun onun static yapılması gerekmektedir.

**this Göstericisi**

Aslında üye fonksiyon kavramı yapay bir kavramdır. Derleyici aslında sınıfın static olmayan üye fonksiyonlarını global fonksiyonlarmış gibi derlemektedir. Pekiyi bu üye fonksiyonlar sınıfın veri elemanlarına nasıl erişirler? Örneğin:

class Sample {

public:

void set(int a, int b);

void disp();

private:

int m\_a;

int m\_b;

};

void Sample::set(int a, int b)

{

m\_a = a;

m\_b = b;

}

void Sample::disp()

{

cout << m\_a << “, “ << m\_b << endl;

}

//...

Sample s;

s.set(10, 20);

s.disp();

İşte derleyici aslında buradaki Set ve Disp fonksiyonlarını sanki global fonksiyonlarmış gibi oluşturur. Sınıf da aslında C’deki struct gibidir. Derleyici fonksiyonlara bu sınıf nesnelerinin adreslerini gizlice bir parametre yoluyla yoluyla geçirmektedir. Fonksiyonların sınıfın veri elemanlarını kullanması aslında bu gizlice geçirilen parametre yoluyla yapılır. Örneğin:

void Sample\_Set(Sample \*this, int a, int b)

{

this->m\_a = a;

this->m\_b = b;

}

void Sample\_Disp(Sample \*this)

{

cout << this->m\_a << “, “ << this->m\_b << endl;

}

Özetle C++’ta aslında üye fonksiyonlar global fonksiyonlar gibidir. Onlara gizlice çağrımda kullanılan nesnelerin adresleri geçirilir. Fonksiyon da sınıfn veri elemanlarına bu gizli parametre yoluyla erişir. Bu gizli parametre açıkça programcı tarafından this anahtar sözcüğü ile kullanılabilir.

this anahtar sözcüğü yalnızca static olmayan üye fonksiyonlar içerisinde kullanılabilir. this anahtar sözcüğü static olmayan üye fonksiyonun çağrılmasında kullanılan nesnenin adresini belirtir. Bu durumda örneğin sınıfın static olmayan bir üye fonksiyonu içerisinde sınıfın m\_a isimli bir veri elemanına doğrudan m\_a ile erişmekle this->m\_a ile erişmek arasında bir farklılık yoktur. Zaten doğrudan m\_a ile erişimde derleyici bu gizlice geçirilen this göstericisini kullanmaktadır.

Sınıfın static üye fonksiyonları aslında nesne ile çağrılmadığı için ve bunlar sınıfın static olmayan elemanlarına erişemediği için bunlara this göstericisi geçirilmez. Dolayısıyla biz sınıfın static olmayan üye fonksiyonlarında this anahtar sözcüğünü kullanamayız.

Static olmayan üye fonksiyon içerisinde \*this ifadesi o üye fonksiyonun çağrılmasında kullanılan nesnenin kendisini belirtir.

Pekiyi this göstericisinin türü nedir? this hangi sınıfın static olmayan üye fonksiyonunda kullanılmışsa o türden bir göstericidir. const üye fonksiyonlarda this göstericisi de gösterdiği yer const olan const bir göstericidir.

**Operatör Fonksiyonları (Operator Overloading)**

Operatör fonksiyonları sınıf türünden nesneleri sanki temel türlerden nesnelermiş gibi aritmetik, karşılaştırma gibi işlemlere sokabilmemizi sağlarlar. Aslında operatör fonksiyonları yalnızca okunabilirliği artırmaktadır. Yani biz operatör fonksiyonlarıyla yapmaya çalıştığımız şeyleri normal fonksiyonlarla (örneğin static fonksiyonlarla) da yapabiliriz. Örneğin:

// Complex.hpp

#pragma once

class Complex

{

public:

Complex();

Complex(double real, double imag = 0)

: m\_real(real), m\_imag(imag)

{}

void Disp() const;

static Complex Add(const Complex &a, const Complex &b);

~Complex();

private:

double m\_real;

double m\_imag;

};

// Complex.cpp

#include <iostream>

#include "Complex.hpp"

using namespace std;

Complex::Complex()

{

}

Complex::~Complex()

{

}

void Complex::Disp() const

{

if (m\_real != 0)

cout << m\_real;

if (m\_imag > 0 && m\_real != 0)

cout << "+";

if (m\_imag != 0) {

if (m\_imag != 1 && m\_imag != -1)

cout << m\_imag;

else if (m\_imag == -1)

cout << "-";

cout << "i";

}

if (m\_real == 0 && m\_imag == 0)

cout << "0";

cout << endl;

}

Complex Complex::Add(const Complex &a, const Complex &b)

{

Complex result;

result.m\_real = a.m\_real + b.m\_real;

result.m\_imag = a.m\_imag + b.m\_imag;

return result;

}

#include <iostream>

#include "Complex.hpp"

using namespace std;

int main()

{

Complex x(3, 2), y(3, 5), z;

z = Complex::Add(x, y);

z.Disp();

return 0;

}

C++’ta operatör fonksiyonları static olmayan üye fonksiyon biçiminde ya da global fonksiyon biçiminde yazılabilir. Bir operatör fonksiyonun geri dönüş değeri herhangi bir türden olabilir. Operatör fonksiyonlarının isimleri operator anahtar sözcüğü ile operatör sembolünden oluşmaktadır. Parametre sayıları ile ilgili şu kısıtlar vardır:

|  |  |  |
| --- | --- | --- |
| Operatörün Türü | static Olmayan Üye Fonksiyon | Global Fonksiyon |
| tek operandlı | 0 | 1 |
| iki operandlı | 1 | 2 |

Örneğin:

// Complex.cpp

#include <iostream>

#include "Complex.hpp"

using namespace std;

Complex::Complex()

{

}

Complex::~Complex()

{

}

void Complex::Disp() const

{

if (m\_real != 0)

cout << m\_real;

if (m\_imag > 0 && m\_real != 0)

cout << "+";

if (m\_imag != 0) {

if (m\_imag != 1 && m\_imag != -1)

cout << m\_imag;

else if (m\_imag == -1)

cout << "-";

cout << "i";

}

if (m\_real == 0 && m\_imag == 0)

cout << "0";

cout << endl;

}

Complex Complex::operator +(const Complex &z) const

{

Complex result;

result.m\_real = m\_real +z.m\_real;

result.m\_imag = m\_imag + z.m\_imag;

return result;

}

a ve b iki sınıf türünden nesne olmak üzere bunlar a + b gibi bir ifadede kullanıldıklarında derleyici bu işlemi yapabilecek sınıf içerisinde ve global düzeyde operatör fonksiyonları araştırır. Eğer bulursa operandları bu fonksiyona argüman olarak geçirerek fonksiyonu çağırır. Fonksiyonun geri dönüş değeri operatörün ürettiği değer olarak kullanılır.

Bir operatör fonksiyonu aslında normal fonksiyon çağırma sentaksıyla da çağrılabilir. Örneğin iki operandlı bir üye operatör fonksiyonu söz konusu olsun:

a op b

işleminin eşdeğeri,

a .operator op(b)

biçimindedir. Tek operandlı bir üye fonksiyonun kullanıldığını düşünelim:

op a ya da a op

işleminin eşdeğeri,

a.operator op()

biçimindedir.

Şimdi derleyicinin bir operatör ile karşılaştığında ne yaptığına bakalım. Derleyici önce operatörün operandlarının türlerine bakar. Operandların ikisi de (ya da tek operandlı operatörlerde teki) temel türlere ilişkinse derleyici hiç operatör fonksiyonu araştırmadan işlemi yapar. Ancak operandlardan en az biri bir sınıf türündense derleyici o işlemi yapabilecek o sınıflar içerisinde ve global düzeyde operatör fonksiyonları araştırır. Operandları o fonksiyona argüman olarak geçirerek o fonksiyonu çağrır. Fonksiyonun geri dönüş değerini de operatörün ürettiği değer olarak alır. Örneğin:

// Complex.hpp

#pragma once

class Complex

{

public:

Complex();

Complex(double real, double imag = 0)

: m\_real(real), m\_imag(imag)

{}

void Disp() const;

Complex operator +(const Complex &z) const;

Complex operator -(const Complex &z) const;

Complex operator \*(const Complex &z) const;

~Complex();

private:

double m\_real;

double m\_imag;

};

// Complex.cpp

#include <iostream>

#include "Complex.hpp"

using namespace std;

Complex::Complex()

{

}

Complex::~Complex()

{

}

void Complex::Disp() const

{

if (m\_real != 0)

cout << m\_real;

if (m\_imag > 0 && m\_real != 0)

cout << "+";

if (m\_imag != 0) {

if (m\_imag != 1 && m\_imag != -1)

cout << m\_imag;

else if (m\_imag == -1)

cout << "-";

cout << "i";

}

if (m\_real == 0 && m\_imag == 0)

cout << "0";

cout << endl;

}

Complex Complex::operator +(const Complex &z) const

{

Complex result;

result.m\_real = m\_real +z.m\_real;

result.m\_imag = m\_imag + z.m\_imag;

return result;

}

Complex Complex::operator -(const Complex &z) const

{

Complex result;

result.m\_real = m\_real - z.m\_real;

result.m\_imag = m\_imag - z.m\_imag;

return result;

}

Complex Complex::operator \*(const Complex &z) const

{

Complex result;

result.m\_real = m\_real \* z.m\_real - m\_imag \* z.m\_imag;

result.m\_imag = m\_real \* z.m\_imag + m\_imag \* z.m\_real;

return result;

}

Karşılaştırma operatörlerine ilişkin operatör fonksiyonlarının parametreleri ve geri dönüş değerleri herhangi bir türden olabilir. Tabii bunların geri dönüş değerlerinin bool türdenolması en normal durumdur. Örneğin:

// Rational.hpp

#pragma once

class Rational

{

public:

Rational();

Rational(int a, int b);

void Disp() const;

~Rational();

Rational operator +(const Rational &r) const;

Rational operator -(const Rational &r) const;

Rational operator \*(const Rational &r) const;

Rational operator /(const Rational &r) const;

bool operator >(const Rational &r) const;

bool operator <(const Rational &r) const;

bool operator >=(const Rational &r) const;

bool operator <=(const Rational &r) const;

bool operator ==(const Rational &r) const;

bool operator !=(const Rational &r) const;

private:

void reduce();

static int gcd(int a, int b);

private:

int m\_a;

int m\_b;

};

// Rational.cpp

#include <iostream>

#include "Rational.hpp"

using namespace std;

Rational::Rational()

{

m\_a = 0;

m\_b = 1;

}

Rational::Rational(int a, int b)

{

m\_a = a;

m\_b = b;

reduce();

}

Rational Rational::operator +(const Rational &r) const

{

Rational result;

result.m\_a = m\_a \* r.m\_b + r.m\_a \* m\_b;

result.m\_b = m\_b \* r.m\_b;

result.reduce();

return result;

}

Rational Rational::operator -(const Rational &r) const

{

Rational result;

result.m\_a = m\_a \* r.m\_b - r.m\_a \* m\_b;

result.m\_b = m\_b \* r.m\_b;

result.reduce();

return result;

}

Rational Rational::operator \*(const Rational &r) const

{

Rational result;

result.m\_a = m\_a \* r.m\_a;

result.m\_b = m\_b \* r.m\_b;

result.reduce();

return result;

}

Rational Rational::operator /(const Rational &r) const

{

Rational result;

result.m\_a = m\_a \* r.m\_b;

result.m\_b = m\_b \* r.m\_a;

result.reduce();

return result;

}

bool Rational::operator >(const Rational &r) const

{

return (double)m\_a / m\_b > (double)r.m\_a / r.m\_b;

}

bool Rational::operator <(const Rational &r) const

{

return (double)m\_a / m\_b < (double)r.m\_a / r.m\_b;

}

bool Rational::operator >=(const Rational &r) const

{

return (double)m\_a / m\_b >= (double)r.m\_a / r.m\_b;

}

bool Rational::operator <=(const Rational &r) const

{

return (double)m\_a / m\_b <= (double)r.m\_a / r.m\_b;

}

bool Rational::operator ==(const Rational &r) const

{

return (double)m\_a / m\_b == (double)r.m\_a / r.m\_b;

}

bool Rational::operator !=(const Rational &r) const

{

return (double)m\_a / m\_b != (double)r.m\_a / r.m\_b;

}

Rational::~Rational()

{

}

void Rational::reduce()

{

int gcdVal = gcd(m\_a, m\_b);

m\_a /= gcdVal;

m\_b /= gcdVal;

if (m\_b < 0) {

m\_a = -m\_a;

m\_b = -m\_b;

}

}

int Rational::gcd(int a, int b)

{

if (a == 0 && b == 0)

return -1;

a = a < 0 ? -a : a;

b = b < 0 ? -b : b;

int r;

while (b) {

r = a % b;

a = b;

b = r;

}

return a;

}

void Rational::Disp() const

{

if (m\_a == 0)

cout << "0";

else {

cout << m\_a;

if (m\_b != 1)

cout << "/" << m\_b;

}

cout << endl;

}

++ ve -- operatörlerine ilişkin operatör fonksiyonları yazılırken bu operatörlerin önek ve sonek biçimleri ayrı ayrı yazılır. Sonek biçimlerinde boş bir int parametre kullanılmalıdır.

**Global Operatör Fonksiyonları**

Global operatör fonksiyonlarının parametre sayıları üye operatör fonksiyonlarından bir fazladır. Yani eğer operatör sembolü tek operandlı bir operatöre ilişkinse global operatör fonksiyonunun 1 parametresi iki operandlı operatöre ilişkinse 2 parametresi bulunmak zorundadır.

Global operatör fonksiyonları temel türlere (sınıf olmayan türlere) ilişkin operatör fonksiyonlarının yazılması sırasında mecburen gerekebilmektedir. Örneğin soldaki operandı temel türlere ilişkin olan sağdaki operandı sınıf türlerine ilişkin olan operatör fonksiyonları üye operatör fonksiyonu olarak yazılamamaktadır. Bunlar mecburen global operatör fonksiyonu olarak yazılırlar. Ayrıca bildirimi bize ait olmayan sınıflar için operatör fonksiyonlarını mecburen global düzeyde yazmak zorunda kalırız. Global operatör fonksiyonları herhangi bir isim alanında bulunabilmektedir. Örneğin:

// rational.hpp

#ifndef RATIONAL\_HPP

#define RATIONAL\_HPP

class Rational

{

public:

Rational()

{

m\_a = 0;

m\_b = 1;

}

Rational(int a, int b = 1);

Rational operator +(const Rational &r) const;

Rational operator -(const Rational &r) const;

Rational operator \*(const Rational &r) const;

Rational operator /(const Rational &r) const;

Rational operator +(int a) const;

Rational operator -(int a) const;

Rational operator \*(int a) const;

Rational operator /(int a) const;

bool operator >(const Rational &r) const;

bool operator <(const Rational &r) const;

bool operator >=(const Rational &r) const;

bool operator <=(const Rational &r) const;

bool operator ==(const Rational &r) const;

bool operator !=(const Rational &r) const;

bool operator >(int a) const;

bool operator <(int a) const;

bool operator >=(int a) const;

bool operator <=(int a) const;

bool operator ==(int a) const;

bool operator !=(int a) const;

Rational &operator ++();

const Rational operator ++(int);

int denom() const { return m\_b; }

int num() const { return m\_a; }

void disp() const;

private:

static int gcd(int a, int b);

void simplify();

private:

int m\_a;

int m\_b;

};

Rational operator +(int a, const Rational &r);

Rational operator -(int a, const Rational &r);

Rational operator \*(int a, const Rational &r);

Rational operator /(int a, const Rational &r);

#endif // RATIONAL\_HPP

// rational.cpp

#include <iostream>

#include "rational.hpp"

using namespace std;

Rational::Rational(int a, int b) : m\_a(a), m\_b(b)

{

simplify();

}

void Rational::disp() const

{

if (m\_a == 0)

cout << "0";

else {

cout << m\_a;

if (m\_b != 1)

cout << "/" << m\_b;

}

cout << endl;

}

int Rational::gcd(int a, int b)

{

if (a == 0 && b == 0)

return -1;

a = a < 0 ? -a : a;

b = b < 0 ? -b : b;

int r;

while (b) {

r = a % b;

a = b;

b = r;

}

return a;

}

void Rational::simplify()

{

int divider;

divider = gcd(m\_a, m\_b);

m\_a /= divider;

m\_b /= divider;

}

Rational Rational::operator +(const Rational &r) const

{

Rational result;

result.m\_a = m\_a \* r.m\_b + m\_b \* r.m\_a;

result.m\_b = m\_b \* r.m\_b;

result.simplify();

return result;

}

Rational Rational::operator -(const Rational &r) const

{

Rational result;

result.m\_a = m\_a \* r.m\_b - m\_b \* r.m\_a;

result.m\_b = m\_b \* r.m\_b;

result.simplify();

return result;

}

Rational Rational::operator \*(const Rational &r) const

{

Rational result;

result.m\_a = m\_a \* r.m\_a;

result.m\_b = m\_b \* r.m\_b;

result.simplify();

return result;

}

Rational Rational::operator /(const Rational &r) const

{

Rational result;

result.m\_a = m\_a \* r.m\_b;

result.m\_b = m\_b \* r.m\_a;

result.simplify();

return result;

}

Rational Rational::operator +(int a) const

{

Rational result;

result.m\_a = m\_a + a \* m\_b;

result.m\_b = m\_b;

result.simplify();

return result;

}

Rational Rational::operator -(int a) const

{

Rational result;

result.m\_a = m\_a - a \* m\_b;

result.m\_b = m\_b;

result.simplify();

return result;

}

Rational Rational::operator \*(int a) const

{

Rational result;

result.m\_a = m\_a \* a;

result.m\_b = m\_b;

result.simplify();

return result;

}

Rational Rational::operator /(int a) const

{

Rational result;

result.m\_a = m\_a;

result.m\_b = m\_b \* a;

result.simplify();

return result;

}

bool Rational::operator >(const Rational &r) const

{

return (double)m\_a / m\_b > (double)r.m\_a / r.m\_b;

}

bool Rational::operator <(const Rational &r) const

{

return (double)m\_a / m\_b < (double)r.m\_a / r.m\_b;

}

bool Rational::operator >=(const Rational &r) const

{

return (double)m\_a / m\_b >= (double)r.m\_a / r.m\_b;

}

bool Rational::operator <=(const Rational &r) const

{

return (double)m\_a / m\_b <= (double)r.m\_a / r.m\_b;

}

bool Rational::operator ==(const Rational &r) const

{

return (double)m\_a / m\_b == (double)r.m\_a / r.m\_b;

}

bool Rational::operator !=(const Rational &r) const

{

return (double)m\_a / m\_b != (double)r.m\_a / r.m\_b;

}

bool Rational::operator >(int a) const

{

return (double)m\_a / m\_b > a;

}

bool Rational::operator<(int a) const

{

return (double)m\_a / m\_b < a;

}

bool Rational::operator >=(int a) const

{

return (double)m\_a / m\_b >= a;

}

bool Rational::operator <=(int a) const

{

return (double)m\_a / m\_b <= a;

}

bool Rational::operator ==(int a) const

{

return (double)m\_a / m\_b == a;

}

bool Rational::operator !=(int a) const

{

return (double)m\_a / m\_b == a;

}

Rational &Rational::operator ++()

{

m\_a = m\_a + m\_b;

simplify();

return \*this;

}

const Rational Rational::operator ++(int)

{

Rational result;

result = \*this;

m\_a = m\_a + m\_b;

simplify();

return result;

}

Rational operator +(int a, const Rational &r)

{

return r + a;

}

Rational operator -(int a, const Rational &r)

{

return r - a;

}

Rational operator \*(int a, const Rational &r)

{

return r \* a;

}

Rational operator /(int a, const Rational &r)

{

return r / a;

}

**Atama Operatör Fonksiyonlarının Yazımı**

C++’ta aynı türden iki sınıf nesnesini birbirlerine atayabiliriz. Aslında bu atama işlemi sırasında sınıfın atama operatör fonksiyonu (operator = fonksiyonu) devreye girmektedir. Sınıfın aynı sınıf türünden iki nesneyi atamada kullanılan operatör fonksiyonuna “kopya atama operatör fonksiyonu (copy assignment operator function) denilmektedir. İşte biz böyle bir operatör fonksiyonunu yazmamışsak derleyici bizim için onu sınıfın karşılık veri elemanlarını birbirlerine atayacak biçimde yazar.

Bazen derleyicinin kendisinin yazdığı kopya atama operatör fonksiyonu bizim işimizi görmeyebilir. Onun bizim tarafımızdan yazılması gerekebilir. Örneğin içerik kopyalamasının yapılması gerektiği durumda kopya atama operatör fonksiyonu programcı tarafından içerik kopyalaması yapacak biçimde yazılmalıdır. Örneğin aşaüıdaki gibi bir String sınıfı söz konusu olsun:

class String {

public:

String()

{

m\_str = new char[1];

m\_len = 1;

\*m\_str = '\0';

}

String(const char \*str)

{

m\_len = strlen(str);

m\_str = new char[m\_len + 1];

strcpy(m\_str, str);

}

void Disp() const

{

cout << m\_str << endl;

}

~String()

{

delete[] m\_str;

}

private:

char \*m\_str;

int m\_len;

};

Burada biz iki String nesnesini birbirine atadığımızda derleyicinin kendisinin yazdığı kopya atama operatör fonksiyonu devreye girer ve veri elemanlarını karşılıklı olarak birbirlerine atar. Böylece m\_str içerisindeki adres karşı tarafa atanmış olur. İki String nesnesi aynı yazıyı gösterir hale gelir. Bunlardan biri yok edildiğinde bitiş fonksiyonu o alanı sileceği için diğerinin gördüğü alan da silinmiş olacaktır. Örneğin aşağıdaki kod programın çökmesine yol açabilir:

int main()

{

String s("this is a test");

{

String k;

k = s;

k.Disp();

}

s.Disp();

return 0;

}

İşte bu tür durumlarda bizim içerik kopyalaması yapacak biçimde kopya atama operatör fonksiyonunu kendimizin yazması gerekir. Atama sırasında göstericilerin gösterdiği nesnenlerin kopyalarından çıkartılır ve artık göstericiler kendi nesnelerini gösterir hale gelir. Örneğin:

// App.cpp

#include <iostream>

#include <cstring>

#include "Complex.hpp"

#include "Rational.hpp"

using namespace std;

class String {

public:

String()

{

m\_str = new char[1];

m\_len = 1;

\*m\_str = '\0';

}

String(const char \*str)

{

m\_len = strlen(str);

m\_str = new char[m\_len + 1];

strcpy(m\_str, str);

}

String &operator =(const String &r)

{

if (&r == this)

return \*this;

delete[] m\_str;

m\_str = new char[r.m\_len + 1];

strcpy(m\_str, r.m\_str);

m\_len = r.m\_len;

return \*this;

}

void Disp() const

{

cout << m\_str << endl;

}

~String()

{

delete[] m\_str;

}

private:

char \*m\_str;

int m\_len;

};

int main()

{

String s("this is a test");

{

String k;

k = s;

k.Disp();

}

s.Disp();

return 0;

}

Kopya atama operatör fonksiyonun yazımına dikkat ediniz:

String &String::operator =(const String &r)

{

if (&r == this)

return \*this;

delete[] m\_str;

m\_str = new char[r.m\_len + 1];

strcpy(m\_str, r.m\_str);

m\_len = r.m\_len;

return \*this;

}

Gerçekten de standart string sınııfndaki kopya atama operatör fonksiyonu yukarıdakiyle benzer biçimde yazılmıştır.

Tabii her sınıf için kopya atama operatör fonksiyonunun programcı tafaından yazılması gerekmez. Özellikle sınıfın veri elemanın gösterici olduğu durumlarda buna gereksinim olmaktadır.

**cout ve cin Nesneleri Ne Anlam İfade Etmektedir?**

Ekraana birşeyler yazmak için cout, klavyeden okuma yapmak için ise cin nesneleri kullanılır. Aslında cout ostream, cin de istream isimli sınıf türünden global nesnelerdir. ostream sınıfının << operatör fonksiyonu bir öteleme yapmak yerine ekrana (stdout dosyasına) yazdırma yapar. Benzer biçimde istream sınıfının >> operatör fonksiyonu da klavyeden okuma yapmaktadır. Bu operatör fonksiyonları overload edilmiştir. Örneğin:

class ostream {

public:

ostream &operator << (int );

ostream &operator << (long );

ostream &operator << (const char \* );

//...

};

Biz de kendi sınıf nesnelerimizi cout nesnesi ile yazdıracak duruma getirebiliriz. Bunun için global operatör fonksiyonu yazmak gerekir. Birinci parametresi ostream & olan, ikinci parametesi bizim sınıfımız türünden olan. Örneğin:

ostream &operator<< (ostream &os, const Rational &r)

{

if (r.m\_a == 0)

os << "0";

else {

os << r.m\_a;

if (r.m\_b != 1)

os << "/" << r.m\_b;

}

os << endl;

return os;

}

**Operataör Fonksiyonlarının Bazı Ayrıntıları**

Aslında C++’ta operatör fonksiyonları çok ayrıntılı bir konudur. Kursumuzun süresi dikkate alındığında burada yalnızca temellerini görmüş olduk. Konunun pek çok ayrıntısı da vardır. Örneğin:

- new ve delete operatörlerine ilişkin operatör fonksiyonları yazılabilir.

- Fonksiyon çağırma ve index operatörüne ([] operatörüne) ilişkin operatör fonksiyonları yazılabilir.

- Bazı operatörlerin (örneğin nokta operatörü gibi, koşul operatörü gibi) operatör fonksiyonları yazılamamaktadır.

- Bazı operatörlerin operatör fonksiyonları (örneğin atama operatörünün) yalnızca üye fonksiyon olarak yazılabilir.

**Türemiş Sınıflarda Adres Dönüştürmeleri**

C++’ta void göstericiler dışında farklı türden adresler birbirlerine atanamazlar. Ancak istisna olarak türemiş sınıf nesnesinin adresi taban sınıf türünden bir gösrericiye atanabilmektedir. Örneğin B sınıfı A sınıfından türetilmiş olsun:

Biz B sınıfı türünden (türemiş sınıf) bir nesnenin adresini A sınıfı türünden bir göstericiye atayabiliriz:

class A {

public:

//...

int m\_x;

int m\_y;

};

class B : public A {

public:

//...

int m\_z;

int m\_k;

};

int main()

{

B b;

A \*pA;

pA = &b; // geçerli

//...

return 0;

}

Türemiş sınıf nesnesinin adresini taban sınıf türünden bir göstericiye atadığımızda bu gösterici artık bağımsız bir taban sınıf nesnesini göstermez. Türemiş sınıf nesnesinin taban sınıf kısmını gösterir:

Türemiş sınıf nesnesinin adresinin taban sınıf türünden göstericiye atanması durumunda bu taban sınıf göstericisi bağımsız bir taban sınıf nesnesini değil türemiş sınıf nesnesinin taban sınıf kısmını gösteriyor durumda olur. Örneğin:

// App.cpp

#include <iostream>

using namespace std;

class A {

public:

//...

int m\_x;

int m\_y;

};

class B : public A {

public:

//...

int m\_z;

int m\_k;

};

int main()

{

B b;

b.m\_x = 10;

b.m\_y = 20;

b.m\_z = 30;

b.m\_k = 40;

A \*pA;

pA = &b; // geçerli

cout << pA->m\_x << ", " << pA->m\_y << endl; // 10, 20

//...

return 0;

}

Türemiş sınıf nesnesinin adresi taaban sınıf türünden gösteriye atanabilir ancak bunun tersi yapılamaz. Yani taban sınıf nesnesinin adresi türemiş sınıf göstericisine atanamaz. Eğer atanabilseydi olmayan elemanlara erişim gibi potansiyel bir tehlike oluşurdu:

Anımsanacağı gibi aslında referanslar da bir çeşit göstericidir. Dolayısıyla C++’ta taban sınıf türünden bir referans türemiş sınıf türünden bir nesneyle ilkdeğer verilerek tanımlanabilir. Bu da bir çeşit türemişten tabana adres atamasıdır. Örneğin:

int main()

{

B b;

b.m\_x = 10;

b.m\_y = 20;

b.m\_z = 30;

b.m\_k = 40;

A &r = b;

cout << r.m\_x << ", " << r.m\_y << endl;

//...

return 0;

}

Böylece örneğin biz taban sınıf türünden bir gösterici ya da referansa sahip bir fonksiyonu türemiş sınıf türünden bir nesnenin adresiyle ya da kendisiyle çağırabiliriz. Örneğin:

void Foo(A \*pA)

{

//...

}

//...

B b;

Foo(&b);

ya da örneğin:

void Foo(A &r)

{

//...

}

//...

B b;

Foo(b);

Bir dizi türetme söz konusu olduğunda herhangi bir türemiş sınıf nesnesinin adresi herhangi bir taban sınıf göstericisine (ya da referansına) atanabilir. Örneğin:

D d;

B \*pB;

pB = &d;

Burada pB göstericisi yoluyla biz D’nin B parçasına erişiriz. D’nin B parçası aslında kendi içerisinde A’yı da içermektedir. O halde biz pB göstericisi ile D’nin A ve B parçöalarına erişebiliriz.

**Türemiş Sınıf Nesne Adresinin Taban Sınıf Türünden Göstericiye (ya da Referansa) Atanmasının Anlamı**

Bir türetme şemasında taban sınıfa ilişkin gösterici ya da referans parametresine sahip bir fonksiyon artık genel işlemleri yapan bir fonksiyondur. Biz böyle bir fonksiyonu herhangi bir türemiş sınıf nesnesinin adresiyle çağırabiliriz. Örneğin aşağıdaki gibi bir türetme eşeması olsun:

Burada çalışanların temel bilgileri Employee sınıfında saklanmaktadır. Şimdi aşağıdaki gibi bir fonksiyon yazmak isteyelim:

void DispIdentity(Employee \*e)

{

//...

}

Burada DispIdentity fonksiyonu her türden sınıfla çağrılabilir. Çünkü bütün nesnelerin bir Employee kısmı vardır.

**Sınıf Türünden Göstericilerin ve Referansların Statik ve Dinamik Türleri**

Gösterici ya da referans olmayan nesnelerin static ve dinamik türleri yoktur. Onların yalnıcz türleri vardır. Benzer biçimde sınıf türünden olmayan göstericilerin ve referansların da static ve dinamik türleri yoktur. Onların da yalnızca türleri vardır. Özetle statik ve dinamik tür kavramı yalnızca sınıf türünden gösterici ve referanslar için kullanılmaktadır.

Bir sınıf türünden gösterici ya da referansın statik türü bildirimde belirtilen türüdür. Dinamik türü ise o gösterici ya da referans bir nesnenin bir kısmını gösteriyorsa onun gösterdiği yerdeki nesnenin bütününün türüne o gösterici ya da referansın dinamik türü denilmektedir. Örneğin:

Burada pA göstericisisnin statik türü A’dır. Ancak dinamik türü B’dir. Çünkü pA göstericisi A türünden bir gösterici olarak bildirilmiştir (statik tür) ancak bu pA göstericisi bağımsız bir A nesnesini değil B nesnesinin A kısmını göstermektedir. Yani bu pA göstericisi bütünü B olan bir nesneyi göstermektedir. Örneğin:

Burada pA’!nın statik türü A’dır. Ancak dinamik türü C’dir. Örneğin yukarıdaki türetme şema söz konusu olsun:

C c;

B \*pB;

A \*pA;

pB = &c;

pA = pB;

Burada pB göstericisinin statik türü B, dinamik türü C’dir. Benzer biçimde pA göstericisinin statik türü A, dinamik türü C’dir. Statik tür bildirimle belirtilen türdür. Hiç değişmez ancak dinamik tür değişebilir. Örneğin:

A a;

B b;

C c;

pA = &a; // pA’nın dinamik türü A

//...

pA = &b; // artık pA’nın dinamik türü B

//...

pA = &c; // artık pA’nın dinamik türü C

Örneğin:

void Foo(A &r)

{

//...

}

//...

A a;

B b;

C c;

Foo(a);

Foo(b);

Foo(c);

Burada Foo her çağrıldığında r referansının dinamik türü değiebilmektedir.

Görüldüğü gibi dinamik tür göstericisinin ya da referansın gösterdiği yerle ilgilidir.

**Çokbiçimlilik (Polymorphism)**

Çokbiçimlilik NYPT’nin anahtar kavramlarından biridir. Bir dilin nesne yönelimli olabilmesi için onda sınıf ve türetme olgusunun dışında çokbiçimliliğin de olması gerekir. Eğer bir dilde sınıf varsa, türetme varsa ancak çokbiçimlilik yoksa böyle dillere nesne tabanlı (object base) diller denilmektedir.

Çokbiçimlilik biyolojiden aktarılmış bir terimdir. Biyolojide çokbiçimlilik canlıların doku ve organlarının evrim süreci içerisinde onların yaşam koşullarına bağlı olarak temel işlevleri aynı kalmak üzere farklılık göstermesine denilmektedir. Örneğin pek çok canlıda kulak vardır. Fakat bu kulaklar duyma bakımından farklılıklar gösterirler.

Yazılımda çokbiçimlilik üç biçimde tanımlanabilir:

1) Biyolojik Tanım: Çok biçimlilik taban sınıfın belli bir fonksiyonunun türemiş sınıflar tarafından temel işlevi aynı kalmak üzere onlara özgü bir biçimde tanımlanmasıdır.

2) Yazılım Mühendisliği Tanımı: Çokbiçimlilik türden bağımsız kod parçalarının yazılabilmesi için kullanılan bir tekniktir.

3) Aşağı Seviyeli Tanım: Çokbiçimlilik önceden yazılmış kodların sonradan yazılacak kodları çağırabilmesi özelliğidir.

C++’ta çokbiçimlilik sanal fonksiyonlarla (virtua l functions) gerçekleştirilir. Bir fonksiyonu sanal yapabilmek için onun prototip bildiriminde virtual anahtar sözcüğünün kullanulması gerekir. virtual anahtar sözcüğü yalnızca prototip bildiriminde kullanılır tanımlama sırasında kullanılmaz. Örneğin :

#include <iostream>

using namespace std;

class A {

public:

void Foo(); // normal fonksiyon

virtual void Bar(int a); // sanal fonksiyon

//...

};

void A::Foo()

{

cout << "A::Foo" << endl;

}

void A::Bar(int a)

{

cout << "A::Bar" << endl;

}

Bir sanal fonksiyon türemiş sınıfta aynı isimle, aynı geri dönüş değeri ve parametrik yapıyla (parametre isimleri farklı olabilir) tanımlanırsa bu işleme “taban sınıftaki sanal fonksiyonun türemiş sınıfta override edilmesi” denilmektedir. Bu durumda -türemiş sınıftaki fonksiyonun prototip bildiriminde virtual anahtar sözcüğü yazılmasa bile fonksiyon sanal olur (ynai virtaul anahtar sözcüğü override işleminde belirtilebilir ya da belirtilmeyebilir). Örneğin:

#include <iostream>

using namespace std;

class A {

public:

void Foo(); // normal fonksiyon

virtual void Bar(int a); // sanal fonksiyon

//...

};

class B : public A {

public:

void Tar(); // normal fonksiyon

void Bar(int a); // override etme işlemi default virtual

//...

};

void A::Foo()

{

cout << "A::Foo" << endl;

}

void A::Bar(int a)

{

cout << "A::Bar" << endl;

}

void B::Tar()

{

cout << "B::Tar" << endl;

}

void B::Bar(int a)

{

cout << "B::Bar" << endl;

}

C++’ta override işleminde dikkat edilecek noktalar şunlardır:

- Bir fonksiyonun override edilebilmesi için onun sanal (virtual) olması gerekir. Sanal olmayan bir fonksiyonu türemiş sınıfta aynı geri dönüş değeri, aynı parametrik yapı ve aynı isimli bildirmek override etme anlamına gelmez.

- Override işlemi sırasında fonksiyonun taban sınıftaki fonksiyonla aynı isimli, aynı türden geri dönüş değerine sahip ve aynı parametrik yapıya sahip olması gerekir.

- C++’ta taban sınıftaki sanal fonksiyon türemiş sınııfn başka bir bölümde override edilebilir. (Örneğin Java ve C#’ta bu yasaktır)

Türemiş sınıftan da sınıf türetip o sanal fonksiyonu yeniden override edebiliriz. Örneğin:

#include <iostream>

using namespace std;

class A {

public:

void Foo(); // normal fonksiyon

virtual void Bar(int a); // sanal fonksiyon

//...

};

class B : public A {

public:

void Tar(); // normal fonksiyon

void Bar(int a); // override etme işlemi default virtual

//...

};

class C : public B {

public:

void Zar();

void Bar(int a); // override etme işlemi default virtual

//...-

};

void C::Zar()

{

cout << "C::Zar" << endl;

}

void C::Bar(int a)

{

cout << "C::Bar" << endl;

}

void A::Foo()

{

cout << "A::Foo" << endl;

}

void A::Bar(int a)

{

cout << "A::Bar" << endl;

}

void B::Tar()

{

cout << "B::Tar" << endl;

}

void B::Bar(int a)

{

cout << "B::Bar" << endl;

}

int main()

{

return 0;

}

**Çokbiçimli Mekanizma**

Bir sınıf türünden gösterici ya da referansla sınıfın bir üye fonksiyonu çağrılmış olsun. Derleyici çağrılan fonksiyonu gösterici yta da referansın statik türü ne ilişkin sınıfın faaliyet alanında arar. Bulursa onun sanal olup olmadığına bakar. eğer sanal değilse bulduğu fonksiyonu çağırır. Eğer sanalsa o gösterici ya da referansın dinamik türüne ilişkin sınıfta override edilmiş olan fonksiyonu çağırır. Örneğin:

#include <iostream>

using namespace std;

class A {

public:

void Foo(); // normal fonksiyon

virtual void Bar(int a); // sanal fonksiyon

//...

};

class B : public A {

public:

void Tar(); // normal fonksiyon

void Bar(int a); // override etme işlemi default virtual

//...

};

class C : public B {

public:

void Zar();

void Bar(int a); // override etme işlemi default virtual

//...

};

void C::Zar()

{

cout << "C::Zar" << endl;

}

void C::Bar(int a)

{

cout << "C::Bar" << endl;

}

void A::Foo()

{

cout << "A::Foo" << endl;

}

void A::Bar(int a)

{

cout << "A::Bar" << endl;

}

void B::Tar()

{

cout << "B::Tar" << endl;

}

void B::Bar(int a)

{

cout << "B::Bar" << endl;

}

int main()

{

C c;

A \*pA;

pA = &c;

pA->Bar(0); // C::Bar çağrılır

return 0;

}

Örneğin:

#include <iostream>

using namespace std;

class A {

public:

void Foo(); // normal fonksiyon

virtual void Bar(int a); // sanal fonksiyon

//...

};

class B : public A {

public:

void Foo(); // normal fonksiyon, override etme değil

void Bar(int a); // override etme işlemi default virtual

//...

};

class C : public B {

public:

void Foo(); // normal fonksiyon override değil

void Bar(int a); // override etme işlemi default virtual

//...

};

void A::Foo()

{

cout << "A::Foo" << endl;

}

void A::Bar(int a)

{

cout << "A::Bar" << endl;

}

void B::Foo()

{

cout << "B::Foo" << endl;

}

void B::Bar(int a)

{

cout << "B::Bar" << endl;

}

void C::Foo()

{

cout << "C::Foo" << endl;

}

void C::Bar(int a)

{

cout << "C::Bar" << endl;

}

int main()

{

C c;

A \*pA;

pA = &c;

pA->Bar(0); // C::Bar çağrılır

pA->Foo(); // A::Foo çağrılır, çünkü Foo sanal değilş

return 0;

}

Örneğin:

#include <iostream>

using namespace std;

class A {

public:

virtual void Bar(int a); // sanal fonksiyon

//...

};

class B : public A {

public:

void Bar(int a); // override etme işlemi default virtual

//...

};

class C : public B {

public:

void Bar(int a); // override etme işlemi default virtual

//...

};

void A::Bar(int a)

{

cout << "A::Bar" << endl;

}

void B::Bar(int a)

{

cout << "B::Bar" << endl;

}

void C::Bar(int a)

{

cout << "C::Bar" << endl;

}

void Test(A \*pA)

{

pA->Bar(0);

}

int main()

{

A a;

B b;

C c;

Test(&a);

Test(&b);

Test(&c);

return 0;

}

**Çokbiçimli Mekanizmanın Faydaları**

Çokbiçimlilik türden bağımsız kod parçalarının yazılması için bir araçtır. Bir projede değişebilecek yerlerle değişmeyecek yerler tespit edilir. Değişebilecek öğelere doğrudan değil sanal fonksiyonlarla çokbiçimli olarak erişilir. Böylece kod türden bağımısız biçimde yazılmış olur. Örneğin topla oynanan bir oyun programında top değişebilen bir öğe olabilir. Oyun kimi zaman normal topla, kimi zaman zıplayan topla kimi zaman patlak topla oynanabilir. Topun gitmesi çokbiçimli bir eylemdir. Yani her top gider ancak gitmesi kendine göredir. Bu durumda bu programda biz topa türden bağımsız olarak bir taban sınıf yoluyla erişiriz.

Top sınıfının Git isimli sanal fonksiyonun diğer türemiş sınıflarda override edildiğini düşünelim. Şimdi oyun şöyle yazılabilir:

Top \*top = new NormalTop();

//...

top->Git();

//...

top->Git();

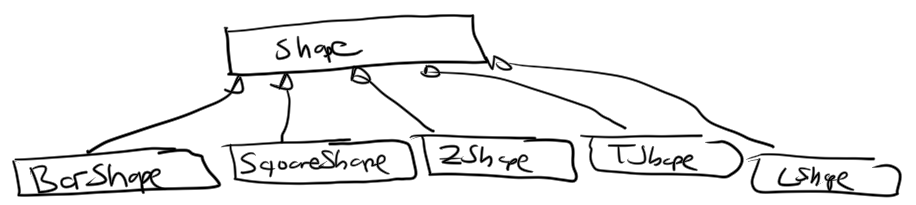
Burada top göstericisinin dinamik türü NormalTop’tur. O halde top normal bir top gibi gidecektir. Biz buradaki topu PatlakTop yapmak için kodun yalnızca şu bölümünü değiştiririz:

Top \*top = new PatlakTop();

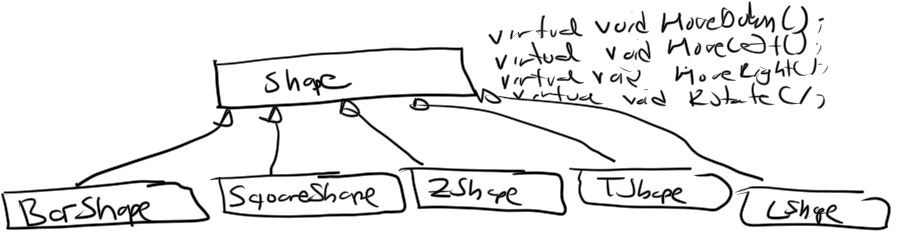
Aslında bu değişikliği bile yapmadan ekleme yöntemiyle de bu işlem yapılabilir (Factory class pattern).

Pekiyi göstericinin ya da referansın dinamik türüne ilişkin sınıfta ilgili sanal fonksiyon override edilmemişse ne olur? İşte bu durumda yukarıya doğru dinamik türe ilişkin sınıfın sanal fonksiyonu override edilmiş ilk sınıfın fonksiyonu çağrılır. Örneğin C sınıfında sanal fonksiyon override edilmediyse B sınıfındaki override edilmiş olan çağrılır. B sınıfında da ilgili fonksiyon override edilmediyse A sınıfındaki çağrılır. Buradan da anlaşılacağı gibi taban sınıftaki sanal fonksiyonun türemiş sınıfta override edilmesi zorunlu değildir.

Bir Tetris oyunu yazacak olalım. Oyundaki şekillerin birtakım ortak elemanları vardır. Tüm bu ortak elemanlar taban bir Shape sınıfında toplanabilir. Diğer şekil sınıfları bu sınıftan türetilerek oluşturulabilir.



Şekillerin düşmesi, sola, sağa ötelenmesi ve döndürülmesi çokbiçimli eylemlerdir. Yani her şekil düşer, sola ve sağa gider ancak bunu kendine göre farklı biçimde yapar. İşte Shape sınıfında bu eylemler sanal fonksiyonlarla bildirilebilir. Sonra türemiş sınıflarda override işlemi yapılabilir.



Şimdi düşen şekilden bağımsız kod parçaları oluşturulabilir. Örneğin:

Shape \*shape = GetRandomShape();

//...

shape->MoveDown();

//...

shape->Rotate();

//...

shape->MoveLeft();

//...

NYPT ideal kullanıldığında daha önceden yazılmış kodların değiştirilmesine gereksinim duyulmamalıdır. Her zaman yeni özellikler “ekleme yapılarak” sağlanmalıdır. Böylece programın mevcut hali sağlamsa ve ekledikten sonra sorun çıkıyorsa eklenen öğeden şüphelenilmelidir. Bu da programın testini kolaylaştırır.

**Saf Sanal Fonksiyonlar (Pure Virtual Functions) ve Soyut (Abstract) Sınıflar**

Bazı çokbiçimli uygulamalarda taban sınıftaki sanal fonksiyonlar aslında hiç çağrılmamaktadır. Örneğin yukarıdaki top örneğindeki Top sınıfının Git fonksiyonu aslında hiç çağrılmaz. Tetris sörneğindeki Shape sınıfınında sanal fonksiyonları aslında hiç çağrılamamaktadır. Bu sınıflar türden bağımsız işlem yapılmasını sağlamak için tepede bulunmaktadır ve ortak bazı elemanları da tutmaktadır. İşte bu tür durumlarda tepediki taban sınıfın sanal fonksiyonlarını içi boş olarak yazmak yerine hiç yazmayabiliriz. Bunun için ilgili sanal fonksiyonu “saf sanal (pure virtual)” biçimde bildirmeliyiz. Saf sanal bildirim = 0 sentaksı ile yapılmaktadır. Örneğin:

class Top {

public:

virtual void Git() = 0;

//...

};

C++’ta saf sanal fonksiyonların tanımlamaları (gövdeleri) yapılmak zorunda değildir. Ancak bunların tanımlamaları yapılsa da sorun oluşmaz. Tek bir saf sanal fonksiyona sahip olan sınıfa “soyut sınıf (abstract class)” denilmektedir. Soyut sınıflar türünden nesneler yaratılmaz. Yaratılmak istenirse hata derleme aşamasında oluşur. Tabii soyut sınıflar türünden göstericiler ve referanslar bildirilebilir. Örneğin:

#include <iostream>

using namespace std;

class A {

public:

virtual void *foo*() = 0;

void bar();

//...

private:

int m\_a;

};

void A::bar()

{

cout << "A::bar" << endl;

}

int main()

{

A a; // error

A \*pa; // geçerli

//...

return 0;

}

Soyut bir sınıftan türetme yapıldığında türemiş sınıfın taban sınıfın tüm saf sanal fonksiyonlarını override etmesi gerekir. Aksi taktirde türemiş sınıf da soyut olur. Türemiş sınıf türünden de nesneler yaratamayız. Soyut sınıflar veri elemanlarına ve normal üye fonksiyonlara sahip olabilirler. Bunların başlangıç fonksiyonları olabilir. (Tabii bu başlangıç fonksiyonları ancak türemiş sınıf tarafından çağrılabilecektir. Bu durumda onların protected bölümde olması daha anlamlıdır.)

**C++’ta Şablon (Template) Kullanımı**

Bazen aslında aynı işi yapan fakat türleri farklı olan fonksiyonlar yazmamız gerekebilmektedir. Örneğin bir diziyi sıraya dizen sort isimli fonksiyon hangi türden diziyi sıraya dizecektir?

void sort(int \*pi, int size);

Burada fonksiyonun birinci parametresi dizinin başlangıç adresini, ikinci parametresi onun uzuluğunu almaktadır. Bu fonksiyon yalnızca int türden dizileri sıraya dizebilir. Eğer örneğin biz long türden dizileri sıraya dizmek istiyorsak içi aynı olmak üzere yeniden bir sort fonksiyonu yazmamız gerekir:

void sort(long \*pl, int size);

İşte şablonlar sayesinde böyle bir fonksiyon şablon (template) olarak yazılır. Sonra derleyici farklı türler için bu şablondan açılım yapar.

C++’ta fonksiyonlar şablon olarak yazılabileceği gibi sınflar da şablon olarak yazılabilmektedir. Bu nedenle şablonlar konusu “fonksiyon şablonları (function template)” ve “sınıf şablonları (class template)” olmak üzere iki ayrılır.

**Fonksiyon Şablonları**

Fonksiyon şablonlarının genel biçimi şöyledir:

template <<şablon parametreleri>>

<fonksiyon tanımlaması>

Şablon parametrelerinde tür olarak class ya da eşdeğer biçimde typename anhatar sözcüğü kullanılır. Bu ilgili parametrenin herhangi bir türden olabileceği anlamına gelir. Örneğin:

template <class T>

void sort(T \*p, int size)

{

//...

}

Bir fonksiyon şablonu normal bir fonksiyon gibi çağrılır. Ancak şablon parametrelerinin tespiti iki biçimde yapılabilmektedir:

1) Açıkça belirtilerek

2) Otomatik olarak

Açıkça belirtme işleminde fonksiyonu çağırırken açısal parantezler içerisinde türler tek tek belirtilir. Örneğin:

#include <iostream>

using namespace std;

template <class T>

T GetMax(const T \*pt, int size)

{

T max = pt[0];

for (int i = 1; i < size; ++i)

if (max < pt[i])

max = pt[i];

return max;

}

int main()

{

int a[10] = { 12, 34, 23, 11, 56, 34, 42, 33, 65, 24 };

double b[5] = { 23.4, 34.7, 19.7, 34.2, 11.8 };

int iresult;

double dresult;

iresult = GetMax<int>(a, 10);

cout << iresult << endl;

dresult = GetMax<double>(b, 5);

cout << dresult << endl;

return 0;

}

Bir fonksiyon şablonu ya da sınıf şablonu sentaks bakımından herhangi bir aşamada kontrol edilebilir. Bazı kontroller daha hiç o şablon kullanılmadan yapılabilir. Bazıları da açım aşamasında yani şablon kullanılırken yapılabilir. Örneğin Microsoft C++ derleyicileri kontrollerin büyük kısmını yalnızca eğer o şablon kullanılmışsa (yani açım (instantiation) aşamasında) yapmaktadır.

Bazı koşullar sağlanıyorsa bir fonksiyon şablonunun şablon parametrelerinin türleri otomatik de tespit edilebilir (automatic template argument deduction). Örneğin:

#include <iostream>

using namespace std;

template <class T>

T GetMax(const T \*pt, int size)

{

T max = pt[0];

for (int i = 1; i < size; ++i)

if (max < pt[i])

max = pt[i];

return max;

}

int main()

{

int a[10] = { 12, 34, 23, 11, 56, 34, 42, 33, 65, 24 };

double b[5] = { 23.4, 34.7, 19.7, 34.2, 11.8 };

int iresult;

double dresult;

iresult = GetMax(a, 10);

cout << iresult << endl;

dresult = GetMax(b, 5);

cout << dresult << endl;

return 0;

}

Burada GetMax(a, 10) çağrısında derleyici a’nın int \* türünden olduğunu görünce T’nin de int olduğunu çıkarsayabilecektir.

#include <iostream>

using namespace std;

template <typename T, typename K>

void Foo(T a, K b)

{

cout << "a = " << a << ", " << b << endl;

}

int main()

{

Foo(10, 20); // T = int, K = int

Foo(12.3, 4); // T = double, K = int

Foo("ali", 123); // T = const char \*, K = int

return 0;

}

Eğer şablon parametrelerinin hepsi fonksiyonun parametrelerinde geçmiyorsa bu durumda otomatik tür tespiti yapılamaz. Örneğin:

#include <iostream>

using namespace std;

template <typename T, typename K>

K Foo(T a)

{

//...

return 0;

}

int main()

{

Foo(10.2); // error: T otomatik tespit edilir ancak K edilemez

Foo<double, int>(10.2); // geçerli T = double, K = int

return 0;

}

Şablon parametrelerinden bazıları açıkça verilebilir, bazılarının derleyici tarafından tespit edilmesi sağlanabilir. Örneğin:

#include <iostream>

using namespace std;

template <typename T, typename K>

T Foo(K a)

{

//...

return 0;

}

int main()

{

Foo<int>(12.4); // T = int, K = double

return 0;

}

Burada T türü otomatik tespit edilemeyeceğinden dolayı açıkça verilmiştir. Otomatik tür tespitinin tek yolu ilgili şablon parametresinin fonksiyonun parametrelerinde kullanılıyor olmasıdır.

Otomatik tür tespitinin başarısız olmasının bir nedeni de fonksiyon argümanlarındaki tutarsızlıktır. Örneğin:

#include <iostream>

using namespace std;

template <class T>

T Add(T a, T b)

{

return a + b;

}

int main()

{

double result;

result = Add(10, 20.3); // error T = int mi, double mı?

cout << result << endl;

return 0;

}

Bu durumda şablon parametresinin türü açıkça belirtilebilir. Örneğin:

#include <iostream>

using namespace std;

template <class T>

T Add(T a, T b)

{

return a + b;

}

int main()

{

double result;

result = Add<double>(10, 20.3); // T = double

cout << result << endl;

return 0;

}

Şablon parametresi açıkça belirtilse de belirtilmese de açım sırasında türden kaynaklanan semantik hatalar söz konusu olabilir. Örneğin:

Add("ali", "veli");

Bu çağrıda T şablon parametresi tespit edilir ancak iki adres toplanamayacağı için derleme sırasında sorun oluşur. Tabii burada açım string türünden olsaydı, string sınıfının + operataör fonksiyonu olduğu için derleme sorunsuz gerçekleşirdi:

#include <iostream>

#include <string>

using namespace std;

template <class T>

T Add(T a, T b)

{

return a + b;

}

int main()

{

string result;

result = Add(string("ali"), string("veli"));

cout << result << endl;

return 0;

}

Bir fonksiyon şablonu ya da sınıf şablonu derlenerek kütüphanelere yerleştirilemez. Çünkü bunları derleyicinin derleme aşamasında görmesi gerekir. Şablonların başlık dosyalarına yerleştirilip include edilmeleri çok karşılaşılan bir durumdur. C++’ın standart kütüphanesinde de şablonların kodları başlık dosyalarındadır.

C++’ın standart kütüphanesinde (buna STL de denilmektedir) pek çok global fonksiyon şablonu vardır. Bunların büyük bölümü <algorithm> dosyası içerisindedir. C++’ın dizi parametresi alan global fonksiyon şablonlarından diziler “iterator” denilen sisteme uyum sağlanması için ilk elemanının adresi ve sondan bir sonraki elemanın adresi ile alınmaktadır. Örneğin:

#include <iostream>

#include <algorithm>

using namespace std;

int main()

{

int a[5] = { 3, 1, 203, 23, 16 };

sort(a, a + 5);

for (int i = 0; i < 5; ++i)

cout << a[i] << " ";

cout << endl;

return 0;

}

Örneğin:

#include <iostream>

#include <algorithm>

#include <string>

using namespace std;

int main()

{

string names[] = { "Ali", "Veli", "Selami", "Ayse", "Fatma" };

sort(names, names + 5);

for (int i = 0; i < 5; ++i)

cout << names[i] << " ";

cout << endl;

return 0;

}

sort fonksiyonu default olarak karşılaştırmayı operator < fonksiyonuyla yapmaktadır. Yani biz de kendi sınıfımızı bu fonksiyonla sıraya dizmek istersek kendi sınıfımız için bir operator < fonksiyonu bulundurmamız gerekir.

fill fonksiyonu bir diziyi belli bir değerle doldurur. Örneğin:

#include <iostream>

#include <algorithm>

using namespace std;

int main()

{

int a[10];

fill(a, a + 10, 1);

for (int i = 0; i < 10; ++i)

cout << a[i] << " ";

cout << endl;

return 0;

}

reverse fonksiyonu bir diziyi ters yüz etmek için kullanılır. Örneğin:

#include <iostream>

#include <algorithm>

using namespace std;

int main()

{

int a[5] = { 1, 2, 3, 4, 5 };

reverse(a, a + 5);

for (int i = 0; i < 5; ++i)

cout << a[i] << " ";

cout << endl;

return 0;

}

**Sınıf Şablonları (class templates)**

Bir sınıfın tamamı (yani tüm üye fonksiyonları) şablon olarak yazılabilir. Sınıf şablonlarının genel biçimi şöyledir:

template <class <template parametreleri>>

class <isim> {

//...

};

Yine class anahtar sözcüğü yerine typename anahtar sözcüğü de kullanılabilir.

Bir sınıf şablonu kullanılırken kesinlikle sınıf isminden sonra açısal parantezler içerisinde şablon parametrelerinin türleri belirtilmek zorundadır. Sınıf şablonlarında otomatik tür tespiti yoktur. Örneğin:

#include <iostream>

using namespace std;

template <typename T>

class Sample {

public:

Sample(T a) : m\_a(a)

{}

T getA() const { return m\_a; }

private:

T m\_a;

};

int main()

{

Sample<int> s(10);

cout << s.getA() << endl;

return 0;

}

Derleyici her farklı şablon açılımı için sınıf şablonundan bir kopya oluşturur. Bu işleme C++ standratlarında “template class instantiations” denilmektedir.

C++’ın standart kütüphanesinde pek çok hazır sınıf şablonu vardır. Örneğin vector isimli sınıf şablonu dinamik büyütülen bir diziyi temsil eder. Biz bir vektöre eleman eklediğimizde vektör elemanları kendi içerisinde tahsis ettiği dinamik bir dizide saklar. Gerektiğinde o dinamik diziyi büyütür. Ancak programcı bu sıkıcı işle uğraşmaz. vector sınıfının şablon parametresi vector içerisinde saklanacak elemanların türünü belirtir. Sınıfın push\_back fonksiyonu sona eleman eleman eklemek için kullanılır. Sınıfın [] operatör fonksiyonu belli indeksteki elemana erişmektedir. Sınıfın size üye fonksiyonu o anda vector’de ne kadar eleman olduğunu bize verir. Örneğin:

#include <iostream>

#include <vector>

using namespace std;

int main()

{

vector<int> v;

for (int i = 0; i < 100; ++i)

v.push\_back(i \* i);

for (int i = 0; i < v.size(); ++i)

cout << v[i] << " ";

cout << endl;

return 0;

}

Örneğin:

#include <iostream>

#include <string>

#include <vector>

using namespace std;

int main()

{

vector<string> v;

v.push\_back("Ali");

v.push\_back("Veli");

v.push\_back("Selami");

v.push\_back("Ayse");

v.push\_back("Fatma");

for (int i = 0; i < v.size(); ++i)

cout << v[i] << endl;

return 0;

}

Bir şablon parametresi default değer alabilir. Bu durumda biz o parametreyi belirtmezsek o default tür kabul edilir. Örneğin:

#include <iostream>

using namespace std;

template <typename T, typename K = int>

class Sample {

public:

Sample(T a, K b) : m\_a(a), m\_b(b)

{}

T getA() const { return m\_a; }

K getB() const { return m\_b; }

void Disp() const

{

cout << m\_a << ", " << m\_b << endl;

}

private:

T m\_a;

K m\_b;

};

int main()

{

Sample<double> s(10.2, 20);

s.Disp();

return 0;

}

Burada açım yapılırken tek bir şablon parametresi için belirlemede bulunulmuştur. K şablon parametresi için tür belirtilmediğinden default int olarak alınır.

Bir sınıf şablonunun üye fonksiyonu sınıfın dışarısında da tanımlanabilir. Ancak sınıf şablonlarının üye fonksiyonları da şablon fonksiyonlar gibidir. Dolayısıyla onların tanımlanmasında yine template anahtar sözcüğü kullanılır. Sınıf isimlerinden sonra da yine template parametrelerinin açısal parantezler içerisinde belirtilmesi gerekir. Örneğin:

#include <iostream>

using namespace std;

template <typename T, typename K = int>

class Sample {

public:

Sample(T a, K b);

T getA() const;

K getB() const;

void Disp() const;

private:

T m\_a;

K m\_b;

};

template <typename T, typename K>

Sample<T, K>::Sample(T a, K b)

{

m\_a = a;

m\_b = b;

}

template <typename T, typename K>

T Sample<T, K>::getA() const

{

return m\_a;

}

template <typename T, typename K>

K Sample<T, K>::getB() const

{

return m\_b;

}

template <typename T, typename K>

void Sample<T, K>::Disp() const

{

cout << m\_a << ", " << m\_b << endl;

}

int main()

{

Sample<double, double> s(10.2, 20);

s.Disp();

return 0;

}

Normal bir sınıf şablon bir sınıftan türetilebilir. Ancak bu durumda türetmes sırasında açım tür açıkça belirtilmelidir. Örneğin:

template <class T>

class A {

//...

};

class B : public A<int> {

//...

};

Burada B sınıfı A sınıfının int açılımından türetilmiştir. Şablon bir sınıf şablon bir sınıftan da türetilebilir. Örneğin:

#include <iostream>

using namespace std;

template <class T>

class A {

//...

};

template <class T>

class B : public A<T> {

//...

};

int main()

{

B <int> b; // B<int> sınıfı A<int> sınıfından türetilmiştir

//...

return 0;

}

Şablon bir sınıf normal bir sınıftan da türetilebilir. Örneğin:

class A {

//...

};

template <class T>

class B : public A {

//...

};

Burada B’nin her türden açılımı A’dan türetilmiştir.

Bir sınıf şablonunun her türden açılımı farklı bir tür belirtir. Örneğin Sample<int> türü ile Sample<long> türü farklı türlerdir. Örneğin:

Sample<int> a;

Sample<long> b;

a = b; // error!

Bir sınıf şablonunda static veri elemanı olabilir. Sınıfın her farklı türden açılımı için bu veri elemanının ayrı kopyası vardır. Örneğin:

template <class T>

class Sample {

//...

static int m\_a;

};

template <class T>

int Sample<T>::m\_a;

Burada örneğin Sample<int>::m\_a ile Sample<long>::m\_a ayrı nesnelerdir.

**Exception İşlemleri**

Exception (istisna) terimi yazılımda “ortaya çıkan beklenmedik hatalı durumlar” anlamına gelmektedir. Bu hatalar sistemin o anki durumundan kaynaklanabilir ya da programcının yapmış olduğu hatalardan kaynaklanabilir. C Programlama Dilinde hatalar ağırlıklı olarak fonksiyonların geri dönüş değerlerinin kontrol edilmesiyle ele alınırlar. Örneğin:

FILE \*f;

if ((f = fopen(“test.dat”, “r”)) == NULL) {

fprintf(stderr, “cannot open file!..\n”);

exit(EXIT\_FAILURE);

}

Her fonksiyonda başarının kontrol edilmesi programı çok kontrollü hale getirebilmektedir. Örneğin dosyaya her yapılan write işleminin kontrol edilmesi programcı için yük oluşturabilmektedir. Bu tür durumlarda programcılar sarma fonksiyonlardan (wrapper functions) faydalanabilmektedir. Örneğin write fonksiyonu çağıran bir fonksiyon yazılıp bunun içerisnde kontrolün yapılması ve başarısızlık durumunda mesaj verilerek programın sonlandırılması gibi.

Fonksiyonların geri dönüş değerlerinin kontrol edilmesi yönteminin bir dezavantajı da iç içe fonksiyon çağrılarında gözükmektedir. Dip bir fonksiyonda başarısızlık oluduğu zaman en yukarıya kadar onları çağıran fonksiyonların başarısızlıkla geri döne döne dışarı çıkması gerekmektedir. Bu da kodu biraz yüklü göstermektedir.

İşte exception mekanizması bu dezavantajları ortadan kaldırmak için düşünülmüştür. Exception mekanizması sayesinde bir hata oluştuğunda ele alım işlemi belli bir noktada yapılabilmektedir. Ancak exception mekanizması doğal bir mekanizma değildir. Derleyici bunu sağlamak için koda programcının yazdıklarının dışında birtakım kodlar eklemektedir. Exception mekanizmasının programın çalışmasını göreli olarak yavaşlattığı da söylenebilir. Bazı C++ programcıları bu nedenle exception mekanizmasını hiç kullanmazlar. Oysa Java gibi C# gibi dillerde exception mekanizması çok yoğun olarak kullanılmaktadır. Bu dillerin kütüphanelerindeki fonksiyonlar genel olarak hata koduyla geri dönmezler. Sorun oluştuğunda exception fırlatırlar.

C++’ta exception mekanizması try, catch ve throw anahtar sözcükleriyle kurulur. try anahtar sözcüğünü bir blok izlemek zorundadır (tek deyim olsa da blok zorunludur). Buna try bloğu denilmektedir. try bloğu tek başına bulunamaz. try bloğunu bir ya da birden fazla catch bloğu izlemek zorundadır. try bloğu ile catch blokları arasında ya da catch bloklarının arasında başka deyimler bulundurulamaz. catch bloğu tek bir catch parametresi alır. Parametrenin yazlnızca türü yazılabilir ya da hem türü hem de değişken ismi yazılabilir. Örneğin:

try {

//...

}

catch (int a) {

//...

}

catch (long a) {

//...

}

catch (double a) {

//...

}

catch bloklarının türleri farklı olmak zorundadır. Programın akışı catch bloğuna geldiğinde artık exception kontrolü uygulanmaya başlamıştır. Yani akış bir throw deyimini gördüğünde doğrudan bir goto işlemi gibi uygun catch bloğuna dallanır. Exception’ı oluşturan deyim throw deyimidir. throw deyiminin genel biçimi şöyledir:

throw [ifade];

throw işlemi gerçekleştiğinde akış bir goto işlemi gibi throw ifadesinin türüne uygun catch bloğuna aktarılır. Ve uygun catch bloğu çalıştırılır. Sonra diğer catch blokları atlanarak akış catch bloklarının sonundan devam eder. Yani adeta catch blokları switch deyiminin break’li case bölümleri gibidir. Örneğin:

#include <iostream>

using namespace std;

void Bar(int a)

{

cout << "Bar begins..\n";

if (a < 0)

throw 100; // int türden

cout << "Bar ends..\n";

}

void Foo(int a)

{

cout << "Foo begins..\n";

Bar(a);

cout << "Foo ends..\n";

}

int main()

{

try {

Foo(-2);

}

catch (int a) {

cout << "int exception caught...\n";

}

catch (long a) {

cout << "long exception caught...\n";

}

catch (double a) {

cout << "double exception caught...\n";

}

cout << "ends..\n";

return 0;

}

throw işleminde throw edilen tür ve değer söz konusudur. Throw edilen tür akışın hangi catch bloğuna geçirileceği ile ilgilidir. throw edilen değer ise catch bloğunun parametresine aktarılır.

Eğer akış try bloğuna girdikten sonra hiç throw işlemi oluşmazsa bu durumda akış try bloğundan normal olarak çıkar. Diğer catch blokları atlanır ve akış catch bloklarının sonundan devam eder. Yani catch blokları “exception oluşursa” kullanılmaktadır.

Eğer throw işlemi oluştuğunda bunu yakalayacak tamamen aynı türden bir catch bloğu yoksa bu durumda derleyici tarafından std::terminate fonksiyonu çağrılır. Bu fonksiyon da std::abort fonksiyonunu çağırmaktadır. abort fonksiyonu da programı sonlandırır. Yani özetle “ele alınmayan exceptionlar (unhandled exceptions)” programın çökmesine yol açmaktadır.

Genellikle int, long, double gibi türlerele throw işlemi yerine sınıflarla throw etme tercih edilmektedir. Örneğin:

#include <iostream>

#include <string>

using namespace std;

class MyException {

public:

MyException(const std::string &errMsg, int errNo) : m\_errMsg(errMsg), m\_errNo(errNo)

{}

friend ostream &operator <<(ostream &os, const MyException &me);

private:

std::string m\_errMsg;

int m\_errNo;

};

ostream &operator <<(ostream &os, const MyException &me)

{

os << me.m\_errMsg << " (" << me.m\_errNo << ")";

return os;

}

void Bar(int a)

{

cout << "Bar begins..\n";

if (a < 0)

throw MyException("Value must not be negative", 123);

cout << "Bar ends..\n";

}

void Foo(int a)

{

cout << "Foo begins..\n";

Bar(a);

cout << "Foo ends..\n";

}

int main()

{

try {

Foo(-2);

}

catch (const MyException &me) {

cout << me << endl;

}

cout << "ends..\n";

return 0;

}

Bir exception oluştuğunda bir sınıf nesnesi yaratılıp onun içerisine o hatayla ilgili gerekli bilgiler yerleştirildikten sonra o nesneyle throw işlemi yapılabilir. Böylece exception’ı yakalayan programcı o nesnenin içerisinden ilgili bilgileri alabilir.

C++’ta bir exception oluştuğunda akış try bloğuna girdikten sonra yaratılmış tüm yerel sınıf nesneleri için ters sırada bitiş fonksiyonları otomatik olarak çağrılmaktadır. Bu sürece C++ terminolojiisinde “Stack unwinding” denilmektedir. Örneğin:

#include <iostream>

#include <string>

using namespace std;

class MyException {

public:

MyException(const std::string &errMsg, int errNo) : m\_errMsg(errMsg), m\_errNo(errNo)

{}

friend ostream &operator <<(ostream &os, const MyException &me);

private:

std::string m\_errMsg;

int m\_errNo;

};

ostream &operator <<(ostream &os, const MyException &me)

{

os << me.m\_errMsg << " (" << me.m\_errNo << ")";

return os;

}

class Sample {

public:

Sample(const std::string &str) : m\_str(str) {}

~Sample()

{

cout << "destructor: " << m\_str << endl;

}

private:

std::string m\_str;

};

void Bar(int a)

{

Sample s("test-2");

if (a < 0)

throw MyException("Value must not be negative", 123);

cout << "Bar ends..\n";

}

void Foo(int a)

{

Sample s("test-1");

Bar(a);

}

int main()

{

try {

Foo(-2);

}

catch (const MyException &me) {

cout << me << endl;

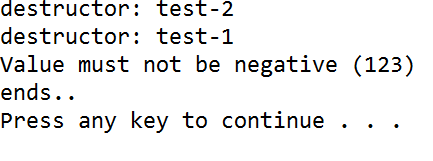
}

cout << "ends..\n";

return 0;

}

Burada görüldüğü gibi exception fırlatıldığında iki sınıf nesnesi yaratılmış durumdadır. throw işlemi yapıldığında bunlar için ters sırada bitiş fonksiyonları çağrılacakır. Yukarıdaki kodun ekran çıktısı şöyle olacaktır:



Burada önemli bir nokta vardır: Akış throw deyimine geldiğinde yalnızca o zamana kadar yaratılmış olan yerel sınıf nesneleri için bitiş fonksiyonları çağrılır. Dinamik yaratılan nesneler için bitiş fonksiyonları çağrılmamaktadır. Bu tür dinamik tahsisatların otomatik geri alınabilmesi için auto\_ptr, shared\_ptr gibi sınıfların kullanılması gerekmektedir.

Programın akışı birden fazla kez try bloğuna girmiş olabilir. Bu durumda bir throw oluştuğunda son girilen try bloğundan dışa doğru try bloklarının catch blokları sırasıyla taranmaktadır. Uygun catch bloğu bulunduğunda akış oraya devredilir. Örneğin:

#include <iostream>

#include <string>

using namespace std;

class MyException {

public:

MyException(const std::string &errMsg, int errNo) : m\_errMsg(errMsg), m\_errNo(errNo)

{}

friend ostream &operator <<(ostream &os, const MyException &me);

private:

std::string m\_errMsg;

int m\_errNo;

};

ostream &operator <<(ostream &os, const MyException &me)

{

os << me.m\_errMsg << " (" << me.m\_errNo << ")";

return os;

}

class Sample {

public:

Sample(const std::string &str) : m\_str(str) {}

~Sample()

{

cout << "destructor: " << m\_str << endl;

}

private:

std::string m\_str;

};

void Bar(int a)

{

Sample s("test-2");

try {

if (a < 0)

throw MyException("Value must not be negative", 123);

cout << "Bar ends..\n";

}

catch (int a) {

cout << "int exception caught: " << a << endl;

}

}

void Foo(int a)

{

Sample s("test-1");

Bar(a);

}

int main()

{

try {

Foo(-2);

}

catch (const MyException &me) {

cout << me << endl;

}

cout << "ends..\n";

return 0;

}

Burada throw işlemi yapıldığında oluşan exception’ın türü MyException olduğu için bunu içteki try bloğunun catch blokları yakalayamayacaktır. Dış try bloğunun catch bloğu yakalayabilecektir. Bir exception iç try bloğunun catch bloğu tarafından yakalanırsa artık dış try blokları için exception oluşmamış gibi işlem devam eder. Örneğin:

#include <iostream>

#include <string>

using namespace std;

class MyException {

public:

MyException(const std::string &errMsg, int errNo) : m\_errMsg(errMsg), m\_errNo(errNo)

{}

friend ostream &operator <<(ostream &os, const MyException &me);

private:

std::string m\_errMsg;

int m\_errNo;

};

ostream &operator <<(ostream &os, const MyException &me)

{

os << me.m\_errMsg << " (" << me.m\_errNo << ")";

return os;

}

class Sample {

public:

Sample(const std::string &str) : m\_str(str) {}

~Sample()

{

cout << "destructor: " << m\_str << endl;

}

private:

std::string m\_str;

};

void Bar(int a)

{

Sample s("test-2");

try {

if (a < 0)

throw 123;

cout << "Bar ends..\n";

}

catch (int a) {

cout << "int exception caught: " << a << endl;

}

}

void Foo(int a)

{

Sample s("test-1");

Bar(a);

}

int main()

{

try {

Foo(-2);

}

catch (const MyException &me) {

cout << me << endl;

}

cout << "ends..\n";

return 0;

}

C++’ta std isim alanında pek çok standart exception sınıfı da vardır. Eğer programcı isterse kendi exception sınıflarını yazmak yerine zaten var olan bu sınıfları kullanabilir ya da bu sınıflardan türetme yaparak kendi exception sınıflarını oluşturabilir.

C++’ın standart exception sınıflarının en tepesinde exception isimli bir sınıf bulunur. Diğer exception sınıfları bu sınıftan türetilmiştir. Pek çok framework’ün kendine özgü exception sınıfları da bulunabilmektedir. std::exception sınıfı bizden başlangıç fonksiyonuyla bir hata yazısı alır. Sonra biz de bu yazaıyı exception’ı yakalayınca what fonksiyonuyla geri alabiliriz. Örneğin:

exception sınıfından türetilmiş olan pek çok exception sınıfları vardır. exception sınıfından türetilmiş olan en önemli iki sınıf logic\_error ve runtime\_error sınıflarıdır. logic\_error daha çok fonksiyonlardaki çeşitli mantıksal hatalara işaret etmek için runtime\_error ise çalışma zamanındaki hatalara işaret etmek için düşünülmüştür. Aslında logic\_error ve runtime\_error sınıflarında da çeşitli sınıflar türetilmiş durumdadır. Örneğin biz bir fonksiyon yazarken bir argümanı beğenmiyorsak logic\_error sınıfından türetilmiş olan invalid\_argument sınıfı ile throw yapabiliriz. Örneğin:

#include <iostream>

#include <string>

#include <exception>

using namespace std;

void Bar(int a)

{

if (a < 0)

throw invalid\_argument("value must not be negative");

}

void Foo(int a)

{

Bar(a);

}

int main()

{

try {

Foo(-2);

}

catch (const invalid\_argument &e) {

cout << e.what() << endl;

}

return 0;

}

logic\_error sınıfından türetilmiş olan diğer bir sınıf da out\_of\_range isimli exception sınıfıdır. Bu exception özellikle belli bir aralıkta bir değer bekleyen fonksiyonlarda eğer bu aralığın dışında değer girilmişse throw edilsin diye oluşturulmuştur. Diğer exception sınıfları için C++’ın dokümanlarına başvurulabilir.

C++’ta türemiş sınıf türünden exception’lar taban sınıf türünden catch bloklarıyla yakalanabilirler. Örneğin çağırdığımız fonksiyon invalid\_argument nesnesiyle throw etmişse biz bunu exception parametreli catch bloğu ile yakalayabiliriz:

#include <iostream>

#include <string>

#include <exception>

using namespace std;

void Bar(int a)

{

if (a < 0)

throw invalid\_argument("value must not be negative");

}

void Foo(int a)

{

Bar(a);

}

int main()

{

try {

Foo(-2);

}

catch (const exception &e) {

cout << e.what() << endl;

}

return 0;

}

Böylece bir türetme şeması içerisinde pek çok farklı sınıf türünden throw yapılmış olsa bile biz bunu tek bir taban sınıf türünden catch bloğu ile yakalayabiliriz. Taban sınıf ve türemiş sınıf catch blokları birlikte bulundurulacaksa türemiş sınıf catch bloğunun daha yukarıda bukundurulması zorunludur. Çünkü exception oluştıuğunda catch bloklarına yukarıdan aşağıya doğru bakılmaktadır. (Eğer taban sınıf catch bloğu daha yukarıda bulundurulabilseydi hepsini o kapardı). Örneğin:

#include <iostream>

#include <string>

#include <exception>

using namespace std;

void Bar(int a)

{

if (a < 0)

throw invalid\_argument("value must not be negative");

}

void Foo(int a)

{

Bar(a);

}

int main()

{

try {

Foo(-2);

}

catch (const invalid\_argument &ia) {

cout << "invalid argument: " << ia.what() << endl;

}

catch (const exception &e) {

cout << e.what() << endl;

}

return 0;

}