Referans Semantiği

Referans semantiği gösterici *(pointer)* semantiğine alternatif olan ancak C++ dilinin diğer araçlarına daha uygun bir yapıdır. `C++` programcılarına verilen ilk tavsiyelerden biri şöyledir:

> "Referans kullanabildiğin her yerde referans yalnızca zorunlu olduğun yerlerde gösterici `pointer` kullan."

Eski `C++`'ta yalnızca tek bir referans kategorisi vardı. `C++11` ile birlikte dile "sağ taraf referansı" `(R value reference)` olarak isimlendirilen ikinci bir referans kategorisi eklendi. Sağ taraf referansları, C++ dilinde daha önce doğrudan mümkün olmayan `taşıma semantiği`'ni `(move semantics)` ve "mükemmel gönderim"i `(perfect forwarding)` gerçekleştirmek üzere dile eklenmiştir.

Modern C++ referansları ikiye ayırıyor:

- `L value reference` (Sol taraf referansları)
- `R value reference` (Sağ taraf referansları)

Biz şimdilik yalnızca sol taraf referanslarını inceleyeceğiz. Sağ taraf referansları çok önemli bir konu olmakla birlikte ayrı bir dersimizin konusu olacak. Aksi belirtilmedikçe bundan sonra (şimdilik) referans dendiğinde sol taraf referansı anlaşılmalıdır.

referans nedir?

Refererans bir nesnenin yerine geçen bir isim olarak düşünülebilir. Öyle ki biz referans olan ismi kullandığımızda aslında o ismin bağlandığı (yerine geçtiği) nesneyi kullanmış oluyoruz. Referanslar & atomu `(declarator)` ile tanımlanır. `T` bir tür olmak üzere

```
T &r = nesne;
```

`r`, "nesne"nin yerine geçen bir referanstır.

referanslara ilk değer vermek `(initialize)` zorunludur.

```
int x = 10;
int &r = x;
```

İlk değer verilmeden bir referansın tanımlanması kesinlikle geçersizdir.

```
int main()
{
 int &r;  //geçersiz
}
```

Bir referansın aynı türden bir nesneye bağlanması zorunludur.

```
void func
{
 double dval = 10.5;
 int &r = dval; //geçersiz
}
```

Tanımlanan bir referans bir nesnenin yerine geçer. Artık söz konusu referans isim, kendi kapamı `(scope)` içinde hep aynı nesnenin yerine geçecektir. Tanımlanmış bir referans bir başka nesnenin yerine geçemez yani referansın daha sonra bir başka nesneye bağlanması mümkün değildir:

```
#include <iostream>

int main()
{
 int a = 10;
 int b = 20;

 int &r = a;
 r = b;

 std::cout << "a = " << a << "\n";
}
```

Yukarıdaki `main` işlevinde tanımlanan `r` referansı, `a` değişkeninin yerine geçiyor. Daha sonra yapılan

```
r = b;
```

ataması, `r` referansının `b` değişkeninin yerine geçtiği anlamına gelmiyor. Bu atamanın anlamı şudur: `r` referansının bağlandığı nesneye `b` değişkeninin değeri atanmaktadır. Referanslar kendisi `const` olan göstericilere `(const pointer)` benzetilebilir. Yukarıdaki kodun bir referans ile değil de bir gösterici değişken kullanılarak yazıldığını düşünelim:

```
#include <iostream>

int main()
{
 int a = 10;
 int b = 20;
 int *const ptr = &a;
 *ptr = b;
 std::cout << "a = " << a << "\n";

 ptr = &b; //geçersiz
}
```

Yukarıdaki örnekte `ptr`, kendisi `const` olan bir gösterici değişkendir `(const pointer)`. `ptr` değişkeninin tanımından sonra, `ptr`'ye artık başka bir nesnenin adresi atanamaz. Aşağıdaki kodu derleyip çalıştırınız:

```
#include <iostream>

int main()
{
 int x = 10;
 int &r = x;
 int y = 45;

 ++r;

 std::cout << "x = " << x << "\n";
 r = y;  //x = y;
 cout << "x = " << x << "\n";

 std::int *ptr = &r;
 *ptr = 8754;
 std::cout << "x = " << x << "\n";
}
```

Şüphesiz birden fazla referans aynı nesneye bağlanabilir:

```
#include <iostream>

int main()
{
 int x = 10;
 int& r1 = x;
 int& r2 = x;
 int& r3 = x;
 int& r4 = x;

 ++r1, ++r2, ++r3, ++r4;

 std::cout << "x = " << x << "\n";
}
```

`C++`'ta referansın yerine geçen referans `reference to reference` yoktur. Aşağıdaki kodu inceleyin:

```
#include <iostream>

int main()
{
 int x = 10;

 int &r1 = x;
 int &r2 = r1; //int &r2 = x;

 ++r1;
 r2 *= 5;
 std::cout << "x = " << x << "\n";
}
```

Herhangi türden bir nesneye bir referans bağlanabilir. Referansın bağlandığı nesne şüphesiz bir gösterici de `(pointer)` olabilir.

```
#include <iostream>


int main()
{
 int x = 10;
 int y = 50;

 int *p = &x;
 int* &r = p;

 std::cout << *r << endl;
 r = &y;
 std::cout << *r << endl;
}
```

Bir yapı türünden (ileride bir sınıf türünden) nesneye de referans bağlanabilir:

```
struct Point {
 int x, y;
};

void clear_point(Point &p)
{
 p.x = 0;
 p.y = 0;
}

#include <iostream>

int main()
{
 Point p = { 12, 67 };
 std::cout << "p.x = " << p.x << " p.y = " << p.y << "\n";
 clear_point(p);
 std::cout << "p.x = " << p.x << " p.y = " << p.y << "\n";
}
```

Bir sol taraf referansı herhangi bir sol taraf değeri ifadesine bağlanabilir:

```
#include <iostream>

int main()
{
 int x = 10;
 int *ptr = &x;
 int &r = *ptr;
 ++r;
 std::cout << "x = " << x << "\n";
}
```

Bir başka örnek:

```
#include <iostream>

int main()
{
 int a[4][4] = { {0, 0, 0, 0}, {1, 7, 9, 3} , {0, 0, 0, 0} , {0, 0, 0, 0} };
 int &r = a[1][2];

 std::cout << r << "\n";
}
```

Bir sol taraf referansının bir sağ taraf ifadesine bağlanması geçersizdir. Aşağıdaki bildirimlere bakınız:

```
int foo();

int main()
{
 int x = 10;
 int &r1 = x; //gecerli
 int &r2 = 20; //gecersiz, sağ taraf degeri
 int &r3 = x + 5; //gecersiz, sağ taraf degeri
 int &r4 = foo(); //gecersiz, sağ taraf degeri
}
```

referansların const olarak bildirilmesi `(const referanslar)`

Bir referans `const` anahtar sözcüğü ile tanımlanabilir. `const` anahtar sözcüğü `"&"` atomundan önce yazılır. Böyle referanslara `const referans` diyeceğiz. `T` bir tür olmak üzere

```
const T &r = x;
```

yazmak ile

```
T const &r = x;
```

yazmak aynı anlamdadır. Bu şekilde tanımlanmış bir referansın bağlandığı (yerine geçtiği) nesne, bu referans yoluyla değiştirilemez. Örneğin:

```
int main()
{
 int a = 10;
 const int &r = a;
 r = 20; // geçersiz
}
```

`main` işlevi içinde yer alan

```
r = 20
```

ifadesi geçerli değildir. Atama aslında `a` değişkenine yapılır. `const` referans kullanılarak, referansın bağlandığı nesne değiştirilemez.Referans isim, salt okuma erişimli olarak işlemlere sokulabilir. Yukarıdaki program parçasının göstericilerle oluşturulan eşdeğer `C` karşılığı şöyle olabilir:

```
int main()
{
 int x = 10;
 const int *const ptr = &x;
 *ptr = 20; //Geçersiz
}
```

Burada `ptr`, hem kendisi `const` hem de gösterdiği nesne `const` olan bir gösterici değişkendir. `ptr`'nin kendisine yapılan atamalar geçersiz olduğu gibi `ptr`'nin gösterdiği nesneye yapılan atamalar da geçersizdir. Peki `const` anahtar sözcüğü `'&'` atomundan sonra, referansın isminden önce kullanabilir mi? Gösterici isminden önce `'*'` atomundan sonra `const` anahtar sözcüğü kullanıldığında, bu durum göstericinin kendisinin `const` olduğu anlamına geliyordu.

```
int x;
int *const ptr = &x;
int &const r = x;
```

Yukarıdaki kodda *r* isimli referansın tanımı doğru değildir. Çünkü referanslar zaten tanımları gereği yalnızca belirli bir nesneye bağlanmak üzere oluşturulur. Yani bir referans, zaten bir nesne ile ilk değerini aldıktan sonra artık başka bir nesnenin yerine geçemez. Referanslar bu anlamlarıyla zaten kendileri const nesnedir.Dolayısıyla, *const* anahtar sözcüğünün yukarıdaki biçimde kullanılmasına gereksizdir.

`const` bir nesne adresinin, ancak gösterdiği nesne `const` olan bir göstericiye atanabileceğini anımsayın. `T` bir tür olmak üzere `const T *` türünden `T *` türüne örtülü `(implicit)` tür dönüşümü `(type conversion)` yoktur.

```
const int x = 10;
int *p1 = &x; //geçersiz
const int *p2 = &x; //geçerli
```

Benzer şekilde, `const` olmayan bir sol taraf referansı `const` bir nesneye bağlanamaz. `const` bir nesnenin yerine ancak `const` bir referans geçebilir.

```
const int x = 10;
int &r1 = x; //geçersiz!
const int &r2 = x; //geçerli
```

Aşağıdaki örneği inceleyin:

```
struct Data {
 int x, y, z;
};

int main()
{
 Data mydata = { 1, 2, 3 };
 const Data &rd = mydata;

 rd.x = 10;  //gecersiz

 int x = 10;
 const int &r = x;

 int a = r; //geçerli
 r = 34; //gecersiz
 ++r; //gecersiz
}
```

`const` bir sol taraf referansı bir sağ taraf değeri ifadesine `(R value expression)` bağlanabilir.

```
void func()
{
 int &r = 10; // geçersiz!
 const int &r = 10; // geçerli
 //...
}
```

Bu durumda derleyici önce geçici bir nesneyi `(temporary object)` sağ taraf değeri ifadesi ile oluşturur. Aslında referansı geçici nesneye bağlayan bir kod üretir. Yukarıdaki kodda `r` referansının tanımı için derleyicinin aşağıdaki gibi bir kod ürettiğini düşünebilirsiniz:

```
const int temp = 10; // geçici nesne 10 değeri ile oluşturuluyor.
const int &r = temp; //r referansı derleyicinin oluşturduğu nesneye bağlanıyor.
```

`const` bir sol taraf referansına başka türden bir nesne ile (eğer geçerli bir tür dönüşümü var ise) ilk değer verilmesi geçerlidir: Derleyicilerin hemen hepsi bu durumu şüpheyle ile karşılayarak bir uyarı mesajı verirler.

```
void func()
{
 double dval = 10.5;
 const int &r = dval; // geçerli ama muhtemelen yanlış
 //...
}
```

Bu durumda yine aslında derleyici bir geçici nesne oluşturur. `const` referansın bağlandığı aslında derleyicinin oluşturduğu geçici nesnedir. Derleyicinin aşağıdaki gibi bir kod ürettiğini düşünebilirsiniz:

```
int main()
{
 double d = 10.5;
 const int temp = (int)d;
 const int &r = temp;
}
```

`const` referanslar neden kullanılır?

`const` nesne göstericileri `(pointer to const object)` ne amaçla kullanılıyorlarsa, `const` referanslar da aynı amaçla kullanılır. `const` referanslar yoluyla bunların bağlandıkları nesneler değiştirilemez. Böyle referanslar, yalnızca okuma amaçlı `(access - get)` nesnelerin yerine geçerler.

referanslar ve fonksiyonların parametre değişkenlerinin referans olması

Referansların en fazla kullanıldığı yer referansla çağrı `(call by reference)` fonksiyon yapısıdır. Bir fonksiyonun parametresi/parametreleri referans yapılır ve fonksiyon nesnenin kendisi ile çağrılır. Aşağıda `int` türden iki nesneyi takas edecek `swap_c` isimli fonksiyon gösterici semantiği ile yazılıyor:

```
void swap_c(int *p1, int *p2)
{
 int temp = *p1;
 *p1 = *p2;
 *p2 = temp;
}

#include <iostream>

int main()
{
 int x = 10, y = 34;

 swap_c(&x, &y);

 std::cout << "x = " << x << "\n";
 std::cout << "y = " << y << "\n";
}
```

Bu kez `int` türden iki nesneyi takas edecek fonksiyon referans semantiği ile yazılıyor:

```
void swap_r(int &r1, int &r2)
{
 int temp = r1;
 r1 = r2;
 r2 = temp;
}

#include <iostream>

int main()
{
 int x = 10, y = 34;

 swap_r(x, y);

 std::cout << "x = " << x << "\n";
 std::cout << "y = " << y << "\n";
}
```

Aşağıdaki gibi bir çağrıya bakarak "değerle çağrı" ya da "referansla çağrı" olduğunu anlayamayız. Bunu anlamak için fonksiyonun bildirimini ve tanımını görmemiz gerekir. (Tabi fonksiyonun isimlendirilmesi bir ipucu verebilir)

```
int main()
{
 int x = 10;

 func(x);
}
```

Aşağıda `foo` işlevi "değerle çağrı" biçiminde, `func` işlevi ise referansla çağrı biçiminde tanımlanıyor.

```
#include <iostream>

void foo(int a)
{
 a = 9999;
}

void func(int &a)
{
 a = 76;
}

int main()
{
 int x = 10;

 foo(x); //call by value
 func(x); //call by reference
}
```

`C++` dilinin semantik yapısı büyük ölçüde const doğruluğuna `(const correctness)` dayanır. `const` olması gereken her varlık `const` olmalıdır. `C`'den aşağıdaki iki fonksiyonun farklı iki arayüzü temsil ettiğini hatırlayalım. `T` bir tür olmak üzere

```
void getter(const T *ptr);
void setter(T *ptr);
```

- `getter` isimli işlev `T` türünden bir nesneye salt okuma `(read/get/access)` amaçlı erişim talep etmektedir. Böyle fonksiyonlara İngilizcede `getter/accessor/get function` gibi terimler yakıştırıldığını hatırlayalım. `const T*` biçiminde tanımlanan parametrelere `"input paramete"` de denmektedir.
- Ancak `setter` isimli işlev `T` türünden bir nesneye bir değer aktarma onu değiştirme `(write/set/mutate)` amaçlı erişim talep etmektedir. Yine böyle fonksiyonlara İngilizcede `setter/mutator/set function` gibi terimler yakıştırıldığını hatırlayalım. `T*` biçiminde tanımlanan parametrelere `"output parameter"` de denmektedir.

Arayüzdeki bu farklılık referans semantiği için de geçerlidir:

```
void getter(const T &ptr);
void setter(T &ptr);
```

Bir fonksiyonun geri dönüş türü bir referans türü olabilir. Geri dönüş türü bir referans türü olan bir fonksiyon bir nesnenin kendisini döndürmektedir. Böyle bir fonksiyona yapılan çağrı ifadesi sol taraf değeridir `(L value expression)`. Aşağıdaki koda bakınız:

```
#include <iostream>

int g = 10;

int &func()
{
 //...
 return g;
}

int main()
{
 std::"cout << "g = " << g << "\n";
 func() = 76;
 std::cout << "g = " << g << "\n";
 ++func();
 std::cout << "g = " << g << "\n";
}
```

Bir fonsiyonun referans semantiği ile aldığı nesneyi yine referans semantiği ile deri döndürmesi tipik bir durumdur:

```
T bir tür olmak üzere

T& foo(T &r)
{
 //...
 return r;
}
```

biçiminde tanımlanan bir fonksiyon aldığı nesneyi döndürmektedir.

```
int &func(int &r)
{
 //...
 r = 777;

 return r;
}
```

Adres döndüren bir fonksiyonun otomatik ömürlü bir nesne adresi döndürmesi tanımsız davranıştır `(undefined behavior)`. Benzer şekilde referans döndüren bir işlevin de otomatik ömürlü nesneye bağlanmış bir referans döndürmesi tanımsız davranıştır. Sentaks hatası olmayan bu durum derleyicilerin hemen hepsinin lojik kontrolüne takılır ve derleyiciler bir uyarı mesajı verirler.

Referans döndüren işlevler

- Statik ömürlü bir nesneyi `(global bie değişkeni ya da statik bir yerel değişkeni)`
- Dinamik ömürlü bir nesneyi
- Kendisini çağıran koddan yine referans semantiği ile aldığı bir nesneyi döndürebilir. Ancak dinamik ömürlü bir nesnenin döndürülmesi durumunda daha çok gösterici semantiği tercih edilmektedir.

Bir fonksiyonun geri dönüş değeri `const` referans da olabilir:

```
const T &foo();
```

`foo` işlevi bir nesnenin kendisini döndürmekle birlikte onun yalnızca okuma `(read/access/get)` amaçlı kullanımını şart koşmaktadır. Bu durumun gösterici semantiğindeki karşılığının

```
const T* foo();
```

olduğunu düşünebilirsiniz.

referanslar ile göstericilerin karşılaştırılması

Her durumda olmasa da birçok durumda referans semantiği ile pointer semantiği birbirlerinin yerine kullanılabilir. Ancak gösterici semantiği ile referans semantiğinin birebir eşdeğer olduğunu söyleyemeyiz.

Bir referans isim her zaman aynı nesnenin yerine geçmek zorundadır. Bir referans ilk değerini aldıktan sonra bir başka nesnenin yerine geçemez. Yani referanslar kendisi const olan göstericilere `(const pointer - top level const)` benzetilebilir. Oysa bir gösterici değişken *const* olmak zorunda değildir. Bir gösterici değişkene başka bir nesnenin adresi atanabilir.

```
 int x = 10;
 int y = 20;
 int &r = x;
 r = y;
```

işlemleri yapıldığında, `r` referansı `x`'in yerine geçmiş olmaz, `x` değişkenine `y` değişkeninin değeri atanmaktadır.

`C`'de diziler fonksiyonlara adres semantiği ile gönderilebilir. Bir dizi üzerinde işlem yapan fonksiyon tipik olarak dizinin adresini boyutunu alır. Belirli türde dizi üzerinde işlem yapan bir fonksiyonun parametresi referans olabilir mi? Diziler fonksiyonlara referans semantiği ile gönderilebilir mi? Hayır! Referanslarla bu iş göstericilerle olduğu gibi yapılamaz. Ancak, örneğin `10` elemanlı `int` türden bir diziyi gösteren gösterici olduğu gibi `10` elemanlı `int` türden bir dizinin yerine geçecek bir referans da tanımlanabilir. Aşağıdaki kodu inceleyin:

```
#include <iostream>

void display(int(&r)[10])
{
 int k;

 for (k = 0; k < 10; ++k)
  std::cout << r[k] << " ";
 std::cout << "\n";
}

int main()
{
 int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
 int b[5] = {1, 2, 3, 4, 5};

 display(a);
 display(b); //geçersiz
}
```

Yukarıdaki kodda yer alan `display` isimli fonksiyona yalnızca `10` elemanlı bir `int` dizi gönderilebilir. Örneğin `display` fonksiyonuna `b` dizisinin gönderilmesi bir sentaks hatasıdır.

- Elemanları gösterici olan bir dizi `(pointer array)` olabilir ama elemanları referans olan bir dizi olamaz.

```
#include <iostream>

int x = 1;
int y = 1;
int z = 1;
int t = 1;

int main()
{
 int *p[] = { &x, &y, &z, &t };

 for (int i = 0; i < 4; ++i) {
  ++*p[i];
 }

 std::cout << x << y << z << t << "\n";
}
```

- Bir göstericiyi gösteren bir gösterici `(pointer to pointer)` olabilir ama bir referansın yerine geçen referans olamaz. Bir gösterici değişkenin adresini bir başka göstericide tutabiliriz.

```
void func()
{
 int x = 10;
 inr y = 20;
 int* ptr = &x;
 int** p = &ptr;
 *p = &y;
 **p = 999;
 //...
}
```

Bu durumun referans semantiğinde doğrudan bir karşılığı yoktur. Ancak şüphesiz bir göstericinin yerine geçen bir referans olabilir. Aşağıdaki kodu inceleyin:

```
#include <iostream>

int main()
{
 int x = 10;
 int *ptr = &x;
 int*& r = ptr;

 *r = 20;
 std::cout << "x = " << x << "\n";
 int a[] = { 10, 20, 30, 40, 50};
 r = a; //ptr = a;
 ++r;   //++ptr;
 ++*r;  //++*ptr;

 std::cout << "a[1] = " << a[1] << "\n";
}
```

`NULL` adresi ve referanslar

C'de değeri `NULL` adresi olan bir gösterici değişkenin hiçbir nesneyi göstermeyen bir gösterici değişken olduğunu hatırlayalım. `NULL` adresinin `C`'de ne kadar yaygın bir biçimde kullanıldığını biliyorsunuz. Bu arada `C++11` standartları ile artık modern C++'ta `NULL pointer` olarak `nullptr` sabiti kullanılmaktadır. `(ileride ayrıntılı`

`olarak göreceğiz).` `nullptr` `C++`'ta bir anahtar sözcüktür ve türü `nullptr_t` olan bir sabittir `(constant)`.

- C'de bir fonksiyonun bir adres döndürmesi durumunda başarısızlık belirtmek amacıyla `NULL` adresi döndürmesi çok yaygın bir konvensiyondur. Örneğin standart C fonksiyonu olan *fopen* bir dosyayı açamaz ise `NULL` adresi döndürür. Standart C fonksiyonu `malloc` başarısız olduğunda `NULL pointer` döndürür.
- C Dilinde bir veri yapısında arama yapan fonsiyonlar aranan değeri buamadıklarında tipik olarak `NULL` gösterici döndürürler. Örneğin standart bir C işlevi olan `strchr` işlevinde bir yazı içinde bir karakter arar. Aranan karakteri yazıda bulursa bulduğu yerin adresini bulamazsa `NULL` adresi döndürür.
- Çağıran koddan bir nesne adresi isteyen bir fonksiyon kendisine `NULL` adresi gönderilmesini çağıran koda bir seçenek olarak verebilir. Örneğin standart `C` fonksiyonu olan time kendisine `NULL` adresi gönderilirse takvim zamanı değerini yani `time_t` türünden değeri bir nesneye yazmaz yalnızca geri dönüş değeri olarak üretir.
- Yine `C`'de `NULL` adresi bir gösterici için bir bayrak değeri olarak kullanılabilir. Örneğin bir kontrol deyiminde bir göstericinin değerinin `NULL` adresi olup olmamasına göre farklı işler yapılabilir.

Bu tür temaların hiçbirinde referans semantiği kullanılamaz. Hiçbir nesnenin yerine geçmeyen bir referans tanımlanamaz. `NULL` referans diye bir kavram yoktur. Bir referans tanımlandığı zaman bir nesnenin yerine geçmelidir.

`C++`'ta referans semantiğinin bulunması göstericilere olan tüm ihtiyacı ortadan kaldırmamıştır. Ancak `C++` dilinin `C` dilinde olmayan birçok aracıyla uyumlu olması için gösterici yerine referans kullanımı tercih edilir.