TİP DÖNÜŞÜMLERİ

Tip Dönüşümü Nedir?

Tip dönüşümü, bir veri tipini anlamını kaybetmeyecek şekilde başka uyumlu tip dönüştürmek anlamına gelir.

```
#include <iostream>
using namespace std;

int main() {
    int i = 10;
    char c = 'A';
    cout << (int)c << endl; // c karakterinin tamsayı kodu ekrana yazıldı: 65

    int sum = i + c; /* c karakteri char veri tipinden int veritipine
                        dönüştürüldü ve i değişkeni ile toplandı ardından
                        atama işleci ile sum değişkenine atandı */
    cout << sum; //75
}
```

Üstü Kapalı Tip Dönüşümleri

C++ dilinde üstü kapalı veri tipi dönüşümü (implicit type casting), zorlama (coercion) ya da standart tip dönüşümü (standart type casting) olarak bilinir. Otomatik tip dönüşümü olarak da adlandırılan ve programcının açık talimatlar vermesine gerek kalmadan derleyicinin bir veri tipini otomatik olarak başka bir uyumlu veri tipine dönüştürmesi işlemidir. Şu durumlarda gerçekleşir;

- Dönüşüm farklı veri tiplerinin değerleri üzerinde gerçekleştirilir.
- Farklı bir veri tipi bekleyen bir fonksiyona bir argüman geçirmeniz halinde gerçekleşir.
- Bir veri tipinin değerini başka bir veri tipinin değişkenine atama durumunda gerçekleşir.

Otomatik tip dönüşümde bellekte az yer kaplayan veri tipinden tanımlanmış değişkenler, bellekte çok yer kaplayan değişkenlere atanmaya (widening casting) çalışıldığında otomatik olarak çok yer kaplayanın veri tipine dönüştürülür:

```
bool→ char→ short int→ int→ unsigned int→ long→ unsigned→ long long→ float→ double→ long
double
```

Örnek;

```
//...
char c;   //1 bayt
int i;    //4 bayt
long l;   //8 bayt
float f;  //4 bayt
double d; //8 bayt
c=65;
i=c; //int veri tipine atanmadan önce; char, int veritipine dönüştürülür.
l=i; //long veri tipine atanmadan önce; int, long veritipine dönüştürülür.
f=12.50;
d=f; //double veri tipine atanmadan önce; float, double veritipine dönüştürülür.
//...
```

Ayrıca bellekte çok yer kaplayan veri tipinden tanımlanmış değişkenler, bellekte az yer kaplayan değişkenlere atanmaya (narrowing casting) çalışıldığında, tip dönüşümü otomatik olarak yapılır ancak veri kaybı söz konusu olur.

```
//...
char c;
```

```
int i;
float f;
i=4095; //i=0x00000FFF -> 0x00,0x00,0x0F,0xFF olarak 4 bayt
c=i;    /* c=0xFF -> c değişkenine int veritipinden char veritipine en
           anlamsız baytı atanır.*/

f=12.50;
/*
Kayan noktalı bir sayıdan tamsayıya atama yapılıyor ise tam kısmı tamsayıya çevrilir ve
sonrasında atama yapılır.
*/
i=f;    //i=12
//...
```

Bilinçli Tip Dönüşümü

Bazen belli işlemleri daha kontrollü yapmak amacıyla programcı yapılan işlemde işlenen veri tipini bir başka ver tipine kasıtlı olarak değiştirir. Bu tür dönüşümüne bilinçli tür dönüşümü (explicit type casting) denir ve tekli tip dönüştürme işleci (unary cast operator) ile yapılır.

Diğer tekli işleçler gibi ifadenin (expression) önünde kullanılır ve parantez içerisinde dönüştürülmek istenen veri tipi yazılarak `int j=(int) 12.8/4;` şeklinde kodlanır. Bu işleç, diğer `+`, `-`, `!`, `--` (pre-decrement), `++` (pre-increment) gibi işleçlerle aynı önceliklidir. Bu dönüşümler C dilinden gelen bilinçli dönüştürmedir. Aşağıda buna ilişkin bir örnek program verilmiştir;

```
#include<stdio.h>
int main() {
   float x = 9.9;
   int y = x+3.3;
   int z= (int)(x)+3.3; // x int veritipine dönüştürülmüş ve ardından toplama yapılmıştır.

   printf("x: %f\n",x);  // x: 9.900000
   printf("y: %d\n",y);  // y: 13
   printf("z: %d\n",z);  // z: 12
}
```

C++ dilinde ise bilinçli tip dönüştürme yukarıdakinin aksine aşağıdaki gibi yapılır;

1. Statik dönüşüm (static cast): Derleme zamanı (compile time) tip dönüşümleri için kullanılır.

```
#include <iostream>
using namespace std;

int main() {
    double x = 1.2;
    int sum = static_cast<int>(x + 1); // 2
    cout << sum;
}
```

2. Dinamik dönüşüm (dynamic cast): Çok biçimlilik (polymorphism) ve kalıtımda (inheritance) çalışma zamanı (run time) tip dönüşümü için kullanılır.

```
#include <iostream>
using namespace std;

class Taban {
public:
    virtual void fonksiyon() {
        cout<< "Taban sınıf fonksiyonu..." << endl;
    }
};
class Turemis : public Taban {
```

```
};

int main() {
    Taban* b = new Turemis();
    Turemis* d = dynamic_cast<Turemis*>(b);
    if (d != NULL) {
        cout << "Taban* Turemis* olarak dönüştürülebilir. d çalışır." <<endl;
        d->fonksiyon();
    }
    else
        cout << "Taban* Turemis* olarak DÖNÜŞTÜRÜLEMEZ. d çalışmaz." << endl;
}
/* Program çalıştığında:
Taban* Turemis* olarak dönüştürülebilir. d çalışır.
Taban sınıf fonksiyonu...

...Program finished with exit code 0
*/
```

3. Sabit dönüşüm (const cast): `const` veya `volatile` niteleyicilerini kaldırır veya ekler.

```
#include <iostream>
using namespace std;
class Kisi
{
private:
    int yas;
public:
    Kisi(int pYas):yas(pYas) {
    }

    // yaş alanını değiştiren bir const yasDegistir fonksiyonu
    void yasDegistir() const{
        ( const_cast<Kisi*> (this) )->yas = 30;
    }

    int getYas()  { return yas; }
};
int fonksiyon(int* ptr) {
    return (*ptr + 100);
}
int main(void) {
    Kisi ilhan(50);
    cout << "Eski Yaş: " << ilhan.getYas() << endl;
    ilhan.yasDegistir();
    cout << "Yeni Yaş: " << ilhan.getYas() << endl;

    const int val = 500;
    const int *ptr = &val;
    int *ptr1 = const_cast<int *>(ptr);
    cout << fonksiyon(ptr1);
}
/*
Eski Yaş: 50
Yeni Yaş: 30
600

...Program finished with exit code 0
*/
```

4. Yeniden yorumlama dönüşümü (reinterpret cast): Dönüştürme öncesi ve sonrası veri tipleri farklı olsa bile, bir veri tipindeki göstericiyi başka bir veri tipindeki göstericiye dönüştürmek için kullanılır. Gösterici tipinin ve göstericinin işaret ettiği verinin aynı olup olmadığı kontrol edilmez.

```
#include <iostream>
using namespace std;
int main() {
    int* p = new int(65);
    char* ch = reinterpret_cast<char*>(p);
    cout << *p << endl;
    cout << *ch << endl;
    cout << p << endl;
    cout << ch << endl;
}
/* Program Çalıştığında:
65
A
0x58af464ee2b0
A

...Program finished with exit code 0
*/
```

Bir tamsayının hangi 4 bayt bellek alanından oluştuğu *Değişken Tanımlama* başlığında anlatılmıştı. Aşağıda, tanımlanan bir tamsayının hangi baytlardan oluştuğunu gösteren bilinçli tip dönüşümlerini içeren program örneği verilmiştir;

```
#include <iostream>
#include <iomanip> // setbase(16): yazdıdılacak bir sonaki değişkenin 16 tabanında yazdırır
using namespace std;
int main(){
    int i=0x10203040;
    char* p= reinterpret_cast<char*>(&i);
    /* int gösteren adres, char içeriyor diye (char*) ifadesiyle
       bilinçli tip dönüşümü (imlplicit cast) yapılıyor. */
    cout << "i:" << setbase(10) << i << "-" <<setbase (16) << i << endl;
    cout << "1.bayt:" << setbase(10) << static_cast<int>(*p)<< "-"
                      << setbase (16) << static_cast<int>(*p) << endl;
    /* Yukarıdaki takimatta static_cast<int>(*p) ifedesiyle p ile gösterilen karakterin
       tamsayı olarak konsola yazdırılması için  bilinçli tip dönüşümü (imlplicit cast)
       yapılıyor. */
    cout << "2.bayt:" << setbase(10) << static_cast<int>(*(p+1)) << "-"
                      << setbase (16) << static_cast<int>(*(p+1)) << endl;
    cout << "3.bayt:" << setbase(10) << static_cast<int>(*(p+2)) << "-"
                      << setbase (16) << static_cast<int>(*(p+2)) << endl;
    cout << "4.bayt:" << setbase(10) << static_cast<int>(*(p+3)) << "-"
                      << setbase (16) << static_cast<int>(*(p+3)) << endl;
}
/* Program çalıştırıldığında:
i:270544960-10203040
1.bayt:64-40
2.bayt:48-30
3.bayt:32-20
4.bayt:16-10

...Program finished with exit code 0
*/
```

Tip dönüşümünün üstünlükleri;

- İşlemlerde Esneklik: Farklı veri tiplerini içeren işlemleri gerçekleştirmede esneklik sağlar. Programcının verileri bir tipten diğerine açıkça dönüştürmesini sağlayarak karışık tip aritmetiğini ve diğer işlemleri kolaylaştırır.
- Uyumluluk: Farklı veri tipleri arasında uyumluluğun sağlanmasına yardımcı olur. Programcının verileri belirli bir bağlamda kullanmadan önce uyumlu bir tipe dönüştürmesini sağlayarak veri uyumsuzluğu hatalarını önler.
- Hassasiyet Kontrolü: Hassasiyet kontrolünün kritik olduğu durumlarda, programcının özellikle sayısal işlemlerde veri tipleri arasında dönüşüm yaparak istenen hassasiyeti açıkça belirtmesini sağlar.
- Açıklık: Tip dönüşümü, programcının veri tipini değiştirme niyetini belirterek kodu daha açık hale getirir. Bu, kod okunabilirliğini artırabilir ve işlenen veri tipiyle ilgili kafa karışıklığını azaltabilir.

Tip dönüşümünün zayıf yönleri;

- Hassasiyet Kaybı: Tip dönüştürmenin en büyük dezavantajlarından biri hassasiyet kaybı potansiyelidir. Örneğin, kayan noktalı bir sayıyı tam sayıya dönüştürürken kesirli kısım kesilir ve bu da bilgi kaybına yol açar.
- Çalışma Zamanı Yükü: Bilinçli tip dönüştürme genellikle program yürütme sırasında dönüştürmenin gerçekleştirilmesi gerektiğinden çalışma zamanı yüküne neden olur. Bu ek işlem, özellikle tip dönüştürmenin sık olduğu durumlarda performansı etkileyebilir.
- Derleyici Uyarıları ve Hataları: Yanlış veya güvenli olmayan tip dönüştürme, derleyici uyarılarına veya hatalarına yol açabilir. Örneğin, uyumsuz tipler arasında dönüştürme yapmaya çalışmak veya geçersiz dönüştürme sözdizimi kullanmak, hata ayıklaması zor olabilecek sorunlara yol açabilir.
- Tanımsız Davranış Potansiyeli: Bazı durumlarda, tip dönüştürme, özellikle uyumsuz tipler arasında dönüştürme yaparken veya dönüştürülen değer hedef tipin aralığının dışında olduğunda tanımsız davranışa yol açabilir. Bu, programda öngörülemez durumlara neden olabilir.
- Kod Bakım Zorlukları: Bilinçli tip dönüşümüne kodun bakımı ve anlaşılması, özellikle karmaşık veya iç içe ifadelerde gerçekleştirildiğinde, daha zor hale gelebilir. Bu, artan kod karmaşıklığına ve azalan okunabilirliğe yol açabilir.
- Taşınabilirlik Endişeleri: Bilinçli tip dönüşümüne farklı platformlar veya derleyiciler arasında daha az taşınabilir olabilir. C'de tip dönüşümünün davranışı değişebilir ve belirli veri tipi boyutları hakkındaki varsayımlar tüm ortamlarda geçerli olmayabilir.

Tip Çıkarımı

Tip çıkarımı (type inference), bir ifadenin veri tipinin otomatik olarak belirlenmesi anlamına gelir. C++ dilinin 2011 sürümünden sonra, `auto` ve `decltype` gibi anahtar sözcükler bu amaçla dile dahil edilmiştir. Böylece bir programcının tür çıkarımını derleyicinin kendisine bırakmasına olanak sağlıyor. Tür çıkarımı yetenekleriyle, derleyicinin zaten bildiği şeyleri yazmak gerekmez.

Tüm tipler yalnızca derleyici aşamasında belirlendiği için, derleme için gereken süre biraz artar ancak programın çalışma süresini etkilemez.

C++ dilinde `auto` anahtar sözcüğü (auto keyword), bildirilen değişkenin tipinin ilk değer verme işleminden otomatik olarak çıkarır. Fonksiyonlar söz konusu olduğunda, dönüş tipleri `auto` ise bu, çalışma zamanında (run time) dönüş tipi belirlenir.

```
#include <iostream>
using namespace std;
int main() {
    // auto a; hata verir. Çünkü ilk değer verilmediğinden veri tipi belirlenemez!
    auto x = 4;
    auto y = 3.37;
    auto z = 3.37f;
    auto c = 'a';
    auto ptr = &x;
    auto pptr = &ptr; //pointer to a pointer
```

```
    cout << typeid(x).name() << endl  //i
         << typeid(y).name() << endl  //d
         << typeid(z).name() << endl  //f
         << typeid(c).name() << endl  //c
         << typeid(ptr).name() << endl  //pi
         << typeid(pptr).name() << endl; // ppi
}
```

C++ dilinde `auto` anahtar sözcüğü belirli bir tipe sahip bir değişken bildirimine olanak tanırken, `decltype` değişkenden tip çıkarmanıza olanak tanır; dolayısıyla `decltype`, geçirilen ifadenin türünü değerlendiren bir tip işlecidir (decltype operator).

```
int fun1() { return 10; }
char fun2() { return 'g'; }
int main() {
    decltype(fun1()) x;
    decltype(fun2()) y;

    cout << typeid(x).name() << endl; //i
    cout << typeid(y).name() << endl; //c

    char a = 65;
    decltype(a) b = a + 5;

    cout << typeid(b).name() << endl; //c
}
```

Lamda İfadeleri

Bir lamda ifadesi (lambda expression), basit fonksiyon nesneleri oluşturmak için özlü bir yol sağlar. 'Lamda' adı, *Alonzo Church* tarafından 1930'larda mantık ve hesaplanabilme hakkındaki soruları araştırmak için icat edilen matematiksel bir form olan lamda hesabından gelir. Lamda hesabı, fonksiyonel bir programlama dili olan LISP'in temelini oluşturmuştur.

Bir lamda ifadesi genellikle çağrılabilir bir nesne alan fonksiyonlara argüman olarak kullanılır. Lamda ifadesi, yalnızca argüman olarak geçirildiğinde kullanılacak olan gövdesi tanımlı bir fonksiyon oluşturmaktan daha basit olabilir. Bu gibi durumlarda, lamda ifadeleri genellikle tercih edilir çünkü işlev nesnelerini satır içi (inline) tanımlamaya izin verirler.

Bir lamda tipik olarak üç bölümden oluşur; bir yakalama listesi `[]`, isteğe bağlı bir parametre listesi `()` ve bir gövde `{}`, bunların hepsi boş olabilir;

`[]` yakalama listesidir. Varsayılan olarak, çevreleyen kapsamın değişkenlerine bir lamda tarafından erişilemez. Bir değişkeni yakalamak, onu lamda içinde, bir kopya veya bir referans olarak erişilebilir hale getirir. Yakalanan değişkenler lamdanın bir parçası haline gelir; fonksiyon argümanlarının aksine, lamda çağrılırken geçirilmeleri gerekmez;

```
int a = 0;
auto f = []() { return a*9; }; // Hata: 'a' erişilemez
auto f = [a]() { return a*9; }; // 'a' değerinden yakalanı
auto f = [&a]() { return a++; }; // 'a' referansından yakalanır
auto call_b = f(); // Lamda fonksiyonu çağrılır
```

`()` parametre listesidir ve normal fonksiyonlardakiyle hemen hemen aynıdır. Lamda hiçbir argüman almazsa, bu parantezler atlanabilir (lamdayı değiştirilebilir olarak bildirmeniz gerekmediği sürece). Bu iki lamda eşdeğerdir;

```
auto call_foo = [x](){ x.foo(); };
auto call_foo2 = [x]{ x.foo(); };
```

Parametre listesi gerçek tipler yerine yer tutucu tipi olan `auto` tipini kullanabilir. Bunu yaparak, bu argüman bir fonksiyon şablonunun şablon parametresi gibi davranır. Aşağıdaki lamdalar, genel kodda bir vektörü sıralamak istediğinizde eşdeğerdir;

```
auto sort_cpp11 = [](std::vector<T>::const_reference lhs, std::vector<T>::const_reference rhs)
                    { return lhs < rhs; };
auto sort_cpp14 = [](const auto &lhs, const auto &rhs) { return lhs < rhs; };
```

`{}` normal fonksiyonlardakiyle aynı olan gövdedir. Bir lamda ifadesinin sonuç nesnesi, diğer fonksiyon nesnelerinde olduğu gibi, `()` işleci kullanılarak yapılır:

```
int multiplier = 5;
auto timesFive = [multiplier](int a) { return a * multiplier; };
std::out << timesFive(2); // 10
multiplier = 15;
std::out << timesFive(2); // 2*5 = 10
```

Varsayılan olarak, bir lamda ifadesinin dönüş tipi türetilir. Aşağıdaki durumda dönüş tipi `bool` olur.

```
[](){ return true; };
```

Aşağıdaki sözdizimini kullanarak dönüş türünü elle da belirtebilirsiniz:

```
[]() -> bool { return true; };
```

Lamda ifadeleriyle basit bir C++ programı aşağıdaki bibi yazılabilir;

```
#include <iostream>
#include <iomanip>
using namespace std;

auto main() -> int
{
    int const   satir  = 3;
    int const   sutun  =4;
    int const   matris[satir][sutun] =
    {
        { 1, 2, 3, 4},
        { 5, 6, 7, 8},
        { 9,10,11,12}
    };
    for( int y = 0; y < satir; ++y )
    {
        for( int x = 0; x < sutun; ++x )
        {
            cout << setw( 4 ) << matris[y][x];
        }
        cout << endl;
    }
}
 /*Program Çıktısı:
   1   2   3   4
   5   6   7   8
   9  10  11  12
 */
```

Lamda İfadeleriyle Yakalanan Nesneler

Lamdadaki değerle yakalanan nesneler varsayılan olarak değişmezdir (immutable). Bunun nedeni, oluşturulan kapatma nesnesinin işleci olan ()'i varsayılan olarak `const` olmasıdır.

```
auto func = [c = 0](){++c; std::cout << c;}; /* ++c lamda'nın durumunu değiştirmeye çalıştığı
için derlenemez.*/
```

Değiştirilmeye, `mutable` anahtar sözcüğü kullanılarak izin verilebilir; bu, yakın nesnenin işleci olan ()'i sabit olmayan hale getirir:

```
auto func = [c = 0]() mutable {++c; std::cout << c;};
```

Dönüş tipiyle birlikte kullanıldığında, `mutable` ondan önce gelir;

```
auto func = [c = 0]() mutable -> int {++c; std::cout << c; return c;};
```

Dönüş tipi belirlenirken bazı durumlar göz önüne alınmalıdır;

```
auto l = [](int value) {
    return value > 10; // Returns bool
};
auto l = [](int value) {
    if (value < 10) {
        return 1; // Hata: lamda dönüş tipini belirleyemedi int?

    } else {
        return 1.5; // Hata: lamda dönüş tipini belirleyemedi double?
    }
};
auto l2 = [](int value) -> double {// Dönüş tipi 'double'
    if (value < 10) {
        return 1;
    } else {
        return 1.5;
    }
};
```

explicit Sıfatı

C++ dilinde explicit anahtar kelimesi, tiplerin üstü kapalı olarak dönüştürülmemesi için yapıcıları (constructor) nitelendirmek için kullanılır. Aşağıdaki kodu göz önüne alalım;

```
#include <iostream>
using namespace std;
class Karmasik {
private:
    double gercek;
    double sanal;
public:
    Karmasik(double r = 0.0, double i = 0.0) :
           gercek(r), sanal(i)
    {
    }
    bool operator == (Karmasik c) //Bir başka karşamış sayıya eşit mi?
    {
        return (gercek == c.gercek && sanal == c.sanal);
    }
};
int main()
{
    Karmasik k(3.0, 0.0);

    if (k == 3.0) cout << "Aynı";
    else cout << "Farklı";
}
/*Program Çıktısı:
Aynı
*/
```

Aşırı yüklenen eşitlik işlecinde kontrol ifadesindeki `&&` işlecinin solundaki ifade doğru ise sağdakine bakılmaz. Bu nedenle program çıktısı `Aynı` olacaktır. Bunu önlemek için explicit anahtara kelimesi yapıcı önünde kullanılır.

```
    explicit Karmasik(double r = 0.0, double i = 0.0) :
           gercek(r), sanal(i)
    {
    }
```

Bu durumda program derlenince aşağıdaki derleme hatası alınır;

```
no match for ‘operator==’ (operand types are ‘Karmasik’ and ‘double’)
```

Hala double değerlerini `Karmasik` tipe dönüştürebiliriz, ancak şimdi açıkça tip dönüşümü yapmamız gerekiyor;

```
    if (k == (Karmasik) 3.0) cout << "Aynı";
    else cout << "Farklı";
```