



C++

1. HÉT – C++1X FEATURE-ÖK

A C++ mögött máig aktív szabványosító testület áll – minden 3-4 évben új szabvány jelenik meg.

Szerencsére a ‘gyártók’ (akik a fordítókat készítik) a szabványt komolyan is veszik.

AZ ELSŐ NÉHÁNY HÉTBEN A C++11 ÉS AZÓTA MEGJELENT
ÚJÍTÁSOKRÓL ESIK SZÓ

Uniform inicializáció

Alap probléma: eltérő típusok (beépített típusok, objektumok, gyűjtemények) inicializálására más szintaxis.

Másik gond: implicit szűkítés (implicit narrowing)

- A lenti példában y változó értéke valójában 55 lesz.

```
int y = 55.6;
```

Uniform inicializáció

Az uniform inicializáció a C++11 óta létezik. Lényege, hogy mindezt most már egységesen meg lehet oldani:

```
int x{5}; // inicializált beépített típus
```

```
int x2{3}; // inicializált beépített típus
```

```
int a[]{ 1,2,3,4 }; // inicializált gyujtemeny
```

```
std::vector<int> myvec{ 3, 4, 5 }; // inicializált STL gyujtemeny
```

```
MyClass mc{}; // default konstruktor
```

```
MyClass mc2{5}; // konstruktor 1 parameterrel
```

```
MyClass mc4{mc2}; // copy konstruktor
```

```
int y2{ 55.6 }; // fordito ezt nem engedi at - halleluia
```

```
class MyClass {  
    private:  
        int val;  
    public:  
        MyClass() { val = 0; }  
        MyClass(int v) : val(v) {}  
};  
  
// ...
```

final kulcsszó

Ez is a C++11-ben jelent meg először.

A C++ OOP aspektusának lényegi része, hogy:

- minden osztály kiterjeszthető (származtathatunk belőle), illetve
- virtuális függvényeket a leszármazott osztályok (akármilyen távoliak is legyenek) felüldefiniálhatják (és a leszármazott típusú változóra mutató szülő típusú pointer esetén a leszármazott osztály azonos metódusa kerül meghívásra).

Ha meg akarjuk tiltani, hogy egy adott osztályból származtatni lehessen, vagy hogy egy virtuális függvényt az adott osztályból leszármazó további osztályok felüldefiniálhassák, használjuk a **final** kulcsszót.

Final kulcsszó a deklarációban használható.

final kulcsszó

```
class Base {};
```

```
class Child final : public Base {}; // Grandchild osztály már nem lehet..
```

```
class Grandchild : public Child {}; // a 'final' class type cannot be used as a base class
```

final kulcsszó

```
class Base2 {  
public:  
    virtual void print() { std::cout <<  
        "Base2" << std::endl; }  
};  
  
class Child2 : public Base2 {  
    virtual void print() final { std::cout  
        << "Child2" << std::endl; }  
};
```

```
class Grandchild2 : public  
Child2 {  
    // cannot override 'final'  
    function "Child2::print"  
  
    virtual void print() {  
        std::cout << "Grandchild2" <<  
        std::endl; }  
};
```

final kulcsszó

```
class Base2 {  
public:  
    virtual void print() { std::cout <<  
        "Base2" << std::endl; }  
};  
  
class Child2 : public Base2 {  
    virtual void print() final { std::cout  
        << "Child2" << std::endl; }  
};
```

```
class Grandchild2 : public  
Child2 {  
    // cannot override 'final'  
    function "Child2::print"  
  
    virtual void print() {  
        std::cout << "Grandchild2" <<  
        std::endl; }  
};
```


Foreach iterálás, auto

A C++-ban többféle iterálási mód létezik konténerek fölött.

A legalapabb (és legfárasztóbb, és legkevésbé biztonságos) megoldás, ha a konténer hosszát vesszük egy for-ciklus alapjául:

```
int myArray[] { 4,6,7,8 };  
for (int inx = 0; inx < 4; inx++) {  
    myArray[inx]++;  
    std::cout << myArray[inx] << std::endl;  
}
```

Foreach iterálás, auto

Az STL konténerek kontextusában a C++ sztenderd könyvtár definiál egy iterator nevű absztrakciót, ami általános eszközt ad az iterálásra. Ez pár oktávval jobb megoldás:

```
std::array<int,4> myStdArray{ 4,6,7,8 };

for (std::array<int,4>::iterator it = myStdArray.begin(); it != myStdArray.end(); it++) {

    *it = *it + 1;

    std::cout << *it << std::endl;

}

// tenyleg modositotta az ertekeket:

std::cout << myStdArray[0] << ", " << myStdArray[1] << ", " << myStdArray[2] << ", " <<
myStdArray[3] << std::endl;
```

Foreach iterálás, auto

Az auto típusnév mint “joker” (= a fordító megmondja a típust, ha egyértelmű) hasznos ilyenkor – sokat egyszerűsít az életünkön:

```
std::array<int, 4> myStdArray{ 4,6,7,8 };  
  
for (auto it = myStdArray.begin(); it != myStdArray.end(); it++) {  
    *it = *it + 1;  
  
    std::cout << *it << std::endl;  
  
}  
  
// tenyleg modositotta az ertekeket:  
  
std::cout << myStdArray[0] << ", " << myStdArray[1] << ", " << myStdArray[2] << ", " <<  
myStdArray[3] << std::endl;
```

Foreach iterálás, auto

De a **legegyszerűbb** megoldás, ha a foreach iterálást alkalmazzuk (akár auto-val együtt).

Fontos tudni, hogy alapesetben másolatot kapunk az értékekről:

```
std::array<int, 4> myStdArray{ 4,6,7,8 };  
for (auto item : myStdArray) { // lehetne int item is...  
    item++;  
    std::cout << item << std::endl;  
}  
  
// itt viszont NEM modositotta az ertekeket:  
std::cout << myStdArray[0] << ", " << myStdArray[1] << ", " << myStdArray[2] << ", " << myStdArray[3] << std::endl;
```

Foreach iterálás, auto

Ezen segíthetünk úgy, hogy referenciát kérünk az elemekre (az auto kombinálható a const és & módosítókkal is):

```
std::array<int, 4> myStdArray{ 4,6,7,8 };  
for (auto& item : myStdArray) { // lehetne int item is...  
    item++;  
    std::cout << item << std::endl;  
}  
  
// így már oke:  
std::cout << myStdArray[0] << ", " << myStdArray[1] << ", " << myStdArray[2] << ", " << myStdArray[3] <<  
std::endl;
```

Reifikált (tárgyasult) függvények

Korábban már találkoztunk (vagy nem?) a függvény pointer és functor (itt: function object) koncepcióval is – legyen ez itt egy rövid ismételés.

Tetszőleges függvény szignatúrájából lehet azonos típusú függvényre hivatkozó pointert csinálni, ha a függvény neve elé csillagot teszünk és a névvel együtt zárójelezzük (zárójel nélkül a visszatérési típushoz tartozna a csillag) – itt pl. fp az nem egy függvény szignatúra, hanem egy függvény pointer változó neve:

```
int xfunc( int(*fp)(int), int b ) {  
    return fp(b);  
}  
  
int func1(int x) {  
    return x * 2;  
}
```

```
int func2(int x) {  
    return x * 3;  
}  
  
int main()  
{  
    std::cout << xfunc(func1, 2) << ", " << xfunc(func2, 2) << std::endl;
```

Reifikált függvények

Az így létrehozott függvény pointer **nevet és típust is meg tud határozni**.

Az előző xfunc() függvény egy fp nevű függvény pointert várt argumentumként

De ha typedef-fel használjuk, akkor egy olyan nevű típust lehet a függvény pointerből csinálni:

```
typedef int(*function_ptr)(int);

int xfunc(function_ptr fp, int b) {
    return fp(b);
}

int func1(int x) {
    return x * 2;
}
```

```
int func2(int x) {
    return x * 3;
}

int main()
{
    std::cout << xfunc(func1, 2) << ", " << xfunc(func2, 2) << std::endl;
```

Reifikált függvények

Amikor legacy kóddal dolgozunk (pl. régi C-könyvtárakkal), előfordulhat, hogy egy API kifejezetten függvény pointert vár. Ha mégis bonyolultabb funkciót megvalósító objektumot adnánk át az API-nak, megtehetjük, hogy egy korszerűbb osztályt adunk át ami függvény pointerként **is** tud viselkedni – ehhez az **operator function_ptr()** operátort kell megvalósítani (ha a függvény pointer típusa az, hogy function_ptr:

```
typedef int(*function_ptr)(int);  
  
int myfunc(int x) { return x * 3; }  
  
class ConvertibleToFunctionPointer {  
    int num_times_called;  
  
public:  
    ConvertibleToFunctionPointer() :  
        num_times_called{0} {}
```

```
operator function_ptr() {  
    num_times_called++;  
    std::cout << "called " << num_times_called  
        << " times!" << std::endl;  
    return myfunc;  
}  
};
```


Reifikált függvények

Megjegyezzük, hogy az “operator xyz()” szignatúrájú metódusok (ahol xyz egy konkrét típus neve, és a metódus visszatérési értéke), éppen a statikus konverziót teszik lehetővé.

- Ha pl. egy osztályban van egy ilyen metódus, hogy “operator int()”, akkor az egy int típusal kell, hogy visszatérjen, és alkalmas lesz arra, hogy az adott osztály példányából int típusra konvertáljon.

```
int main() {  
    ConvertibleToFunctionPointer ctfp;  
    std::cout << function_ptr(ctfp)(5) << std::endl;  
    std::cout << static_cast<function_ptr>(ctfp)(6);  
}
```

```
called 1 times!  
15  
called 2 times!  
18  
C:\Users\Adame\
```

Reifikált függvények

Manapság persze inkább functorokat (function objecteket, azaz függvényként meghívható objektumokat) szokás használni.

Ilyenre láttunk már példát és látunk is majd még bőven. Álljon itt egy rövid példa:

```
class AddTwo {  
    public:  
        int operator()(int input) { return input + 2; }  
};
```

auto

Az auto-t hivatalosan úgy is hívják, hogy ‘placeholder type specifier’ – hiszen pont ez a lényege

Változók esetén az auto azt jelenti, hogy a típust a változó inicializációja alapján automatikusan meghatározza a fordító. Ez a felhasználás a C++11 óta létezik.

Függvények esetén a C++14 óta a visszatérés típusa is lehet auto. Ilyenkor a return statement határozza meg, hogy a fordító milyen típust tulajdonít a visszatéréshez.

A C++17 óta pedig az ún. ‘non-type template parameter’-ek, vagyis a nem típust, hanem értéket meghatározó template paraméterek is kikövetkeztethetők auto-val. Például az előző példában az `std::array` mérete (13-as fólia) auto-val helyettesíthető lett volna (legalábbis olyan fordító esetében, ami a C++17 vagy későbbi szabványt valósít meg).

decltype – kis kitérővel (referenciák)

A következő téma a decltype operátor, ami változók és kifejezések típusának lekérdezését teszi lehetővé. De mielőtt ebbe belemennénk, át kell beszélnünk, hogy milyen referencia típusok vannak a C++-ban (ez ismétlés a múltkori óráról).

A C++-ban függvény meghívásakor az adott függvény másolatot kap az argumentumként átadott változókból -> ha módosítja az értéküket, a módosítás csak lokális érvényű lesz

Ez alól kivételt képeznek a pointerek és referenciák: ezek **hivatkozások** egy változóra. A hivatkozás másolata is egy hivatkozás, ezért kivételesek a pointerek és referenciák (nem azért, mert nem ugyanúgy értékként kerülnének átadásra).

decltype – kis kitérővel (referenciák)

A C++ nyelv 2011 óta 3-féle referenciát különböztet meg:

- Lvalue (bal oldali) referencia: hivatkozás T típusra T& jelöléssel
- Const (konstans) referencia: hivatkozás T típusra const T& jelöléssel – ilyenkor az adott érték csak olvasható. Mégis hasznos, mert – különösen ha nagy objektumról van szó – jó dolog, ha nem kell a stack-en másolgatni
- Rvalue (jobb oldali) referencia: hivatkozás T típusra T&& jelöléssel. Az rvalue referencia azt mondja a fordítónak, hogy az adott referencia egy temporary (vagy expiring) változó értékére hivatkozik, amely változóra később nem lesz szükség. Miért jó ez?
 - Mint láthattuk, a változó által birtokolt erőforrások memóriaterülete másolás nélkül átadható egy másik változónak
 - Úgy érdemes felfogni, hogy van egy memóriarekeszem, ami tárol egy értéket. Ha tudom, hogy az adott változót már nem fogja a program használni, az értéket átadhatom egy másik változónak anélkül, hogy le kellene másolnom.

decltype – kis kitérővel (referenciák)

Például: vessük össze az alábbi két swap() függvényt. Itt az a különbség, hogy std::move(x) egy jobb oldali referenciát csinál x változóból

```
void swap1(std::string& a, std::string& b) {  
    std::string temp{ a };  
    a = b;  
    b = temp;  
}
```

```
void swap2(std::string& a, std::string& b) {  
    std::string temp{ std::move(b) };  
    b = std::move(a);  
    a = std::move(temp);  
}
```

Ezért, míg a bal oldalon temp változó új dinamikus memóriát foglal le, a jobb oldalon egyszerűen felhasználja b korábbi dinamikus memória területét. Az új b pedig felhasználja a korábbi dinamikus memória területét. Az új a pedig felhasználja temp (vagyis b korábbi) dinamikus memória területét.

decltype – kis kitérővel (referenciák)

Foglaljuk kicsit most össze más szempontból is, mit tudunk a referenciákról!

A referenciákat mindig inicializálni kell – hiszen nem változók hanem valamilyen (bal oldali vagy temporary) értékre hivatkoznak!

Bal oldali referenciát csak bal oldali értékkel (tehát pl. egy változó nevével) inicializálhatunk

Konstans referenciát inicializálhatunk bal oldali értékkel és jobb oldali értékkel is

Jobb oldali referenciát csak jobb oldali értékkel inicializálhatunk (pl. egy konstans literállal) – de persze, mint láttuk az `std::move()` segítségével jelezhetjük a fordítónak, hogy egy adott változót kezelhet ellopható jobb oldali értéként is.

A referenciákban közös, hogy mindegyik egyfajta hivatkozás – a másolást megspóroljuk velük. De a jobb oldali referencia kifejezetten a mozgatót támogatja, a bal és konstans referencia pedig a költséghatékony átadását értékeknek (úgy hogy írhatjuk, vagy úgy hogy csak olvashatjuk).

decltype – kis kitérővel (referenciák)

Végezetül, fontos arról szót ejteni, hogy mi történik, ha egy referencia referenciáját vesszük! (ez a későbbiek miatt lesz fontos)

Ilyenkor eredményül csak akkor kapunk jobb oldali referenciát, ha jobb oldali referencia jobb oldali referenciáját vesszük!

Minden más esetben sima, bal oldali referenciát kapunk (tehát egy bal oldali referencia bármilyen referenciája bal oldali marad, és egy jobb oldali referencia bal oldali referenciáját is bal oldali referenciaként értelmez a fordító. Ezeket a szabályokat úgy nevezik, hogy **“reference collapsing rules”**, és mindezt akkor lesz fontos értenünk, amikor olyan függvényeket írunk, melyek jobb oldali referenciát várnak, de ha bal oldali referenciát adunk nekik át akkor a reference collapse miatt bal oldali referenciaként is kezelhetik a bemenő típust. (ez majd a perfect forwarding témakörnél jön elő).

decltype

A generikus programozásnál hasznos, ha egy változó típusát, vagy egy kifejezés által visszaadott típust le tudunk kérdezni.

A C++11 óta a decltype kulcsszó (függvény) éppen ezt teszi lehetővé. **A decltype tehát egy függvény, ami egy változót vagy kifejezést vár és annak a deklarált típusát adja vissza.**

- *Ez kicsit hasonlít az auto kulcsszóra, de más, mivel az auto-t meg egy éppen létrehozni kívánt változó típusának kikövetkeztetésére használjuk (éppen a deklaráció közben!)*

Generikus programozásnál sokszor nagyon nehéz (vagy lehetetlen előre) megmondani, hogy mi az a pontos típus, amire szükségünk van (ld. példa később).

A decltype történetéhez hozzátartozik, hogy már a '90-es években sokféle fordító sokféle változatát megvalósította az ún. 'typeof' operátornak. 2002-ben javasolta Stroustrup, hogy legyen ennek egy sztenderd változata, és ő javasolta a decltype nevet (declared type alapján mozaikszó)

Fontos tudni: decltype eredménye általában véve nem értékeli ki az operandust, csak szintaktikailag nézi a forráskódot.

decltype

Tehát, informálisan azt lehet mondani, hogy ‘decltype(x)’-et a fordító úgy értékeli ki, hogy:

- A.) Ha x egy változóra (lokális vagy névtér scope-ban levő változóra, vagy statikus változóra) utaló kifejezés, akkor decltype a hivatkozott változó deklarált típusát adja vissza
- B.) Ha x nem egy “változóra utaló *id-expression* kifejezés”, helyzettől függően bal vagy jobb oldali referenciát ad vissza.

```
auto& returnRef(int& p) {  
    return p;  
}  
  
struct MyStruct {  
    int d;  
    MyStruct(int d) : d(d) {}  
};
```

```
int a = 10;  
  
decltype(a) b; // a int lesz  
  
std::cout << "decltype(a) is " << typeid(b).name() << std::endl;  
  
decltype(returnRef(a)) c = b; // muszaj neki erteket adni! mert referencia  
  
MyStruct* ms = new MyStruct(5);  
  
decltype(ms->d) e1; // int  
  
decltype((ms->d)) e2; // itt is muszaj erteket adni! Nem id-expression  
kifejezes
```

decltype

De mikor elengedhetetlenül hasznos a decltype?

Hasznos (és haladó) programozási megközelítés a **funkcionális programozás (FP)**. Az egyik legalapvetőbb fogalom az FP-ben a magasabbrendű függvények fogalma. Ezek olyan függvények, amik maguk is függvényt vagy függvényeket kapnak argumentumként. Előfordulhat, hogy egy magasabbrendű függvénynek vissza kell adnia egy argumentumként kapott függvény kimenetét. De milyen típusú lesz a kimenet?

Tulajdonképpen itt egy olyan mechanizmusra van szükségünk, ami a visszatérési típus decltype-ját adja vissza. A visszatérési típus viszont kifejezhető az auto kulcsszó segítségével!

-> használjunk tehát decltype(auto)-t, mint visszatérési típust!

decltype(auto)

Amikor egy függvény változtatás nélkül visszaadja egy argumentumként kapott függvény kimenetét, azt úgy nevezzük, hogy ‘perfect forwarding’.

Ilyenkor a visszatérés típusa: `decltype(auto)` – megj. Ez a példa még nem perfect forwarding, ld. később

```
template <typename Function, typename Object>
decltype(auto) applyFnToObject(Function func, Object obj) {
    return func(obj);
}
```

Decltype(auto)

A függvény ún. functorral (objektum, ami úgy viselkedik, mint egy függvény) meghívható:

```
class AddTwo {  
public:  
    int operator()(int input) { return input + 2; }  
};  
int main()  
{  
    std::cout << applyFnToObject<AddTwo, int>(AddTwo(), 5) << std::endl;  
    std::cout << applyFnToObject<AddTwo, int>(AddTwo(), 6) << std::endl;  
}
```

Decltype(auto)

Ugyanakkor: ha az `AddTwo()` kifejezetten jobb oldali értéket vár, és jobb oldali referenciát adunk át az `applyFnToObject()` függvénynek, a program le se fordul, mert `applyToFnObject` fv. az `Object` típusú `obj` nevű argumentumból lokális változót csinál (érték szerinti másolás!) ami már nem jobb oldali referencia lesz!

- Az, hogy jobb oldali referenciát adunk át az `applyFnToObject()` fv-nek, az csak azt jelenti, hogy miután az értéket átadjuk a függvénynek, a függvényen kívül már nem tervezzük használni!

Ezért mondtuk, hogy az előző példa még nem perfect forwarding-ot valósít meg.

Az alábbi példában látszik, hogy a függvény meghívásakor lép fel a fordítási hiba (5. sor)!

Decltype(auto) – még nem perfect forwarding

```
1  #include <iostream>
2
3  template <typename Function, typename Object>
4  decltype(auto) applyFnToObject(Function func, Object obj) {
5      return func(obj);
6  }
7
8  class AddTwo {
9  public:
10     int value;
11     AddTwo() { value = 0; }
12     int operator()(int input) {
13         value = input + 2;
14         return value;
15     }
16 };
17
18 class AddTwoProblematic {
19 public:
20     int value;
21     AddTwoProblematic() { value = 0; }
22     int operator()(int&& input) {
23         value = input + 2;
24         return value;
25     }
26 };
27
28 int main() {
29     std::cout << applyFnToObject(AddTwo(), 5) << std::endl;
30     int x = 5;
31     std::cout << applyFnToObject(AddTwoProblematic(), std
        ::move(x)) << std::endl;
32     return 0;
33 }
```

```
clang++-7 -pthread -std=c++17 -o main main.cpp
main.cpp:5:10: error: no matching function for call to object of
      type 'AddTwoProblematic'
      return func(obj);
             ^~~~~
main.cpp:31:18: note: in instantiation of function template
      specialization 'applyFnToObject<AddTwoProblematic, int>'
      requested here
      std::cout << applyFnToObject(AddTwoProblematic(), st...
                     ^
main.cpp:22:7: note: candidate function not viable: no known
      conversion from 'int' to 'int &&' for 1st argument
      int operator()(int&& input) {
          ^
1 error generated.
compiler exit status 1
```

Decltype(auto)

Ezen a problémán lokális értelemben segíteni lehetne úgy, hogy `std::move`-ot alkalmazunk az argumentumra – (persze ez nagy általánosságban nem megoldás, mert ha `applyFnToObject`-nek nem jobb oldali értéket adtunk át, és `func` meghívása után még fel akarnánk a változót használni, `func` meghívásakor már (részben, dinamikus memória szempontjából) kihúzhatnánk a memóriát a változó alól):

```
template <typename Function, typename Object>
decltype(auto) applyFnToObject(Function func, Object obj) {
    return func(std::move(obj));
}
```

(ebben az esetben ez nem játszik, de általánosságban ez akkor sem megoldás...)

Decltype(auto)

A teljes megoldás abból áll, hogy:

- A.) az `applyFnToObject()` függvény az objektumra rvalue referenciát vár
- B.) az argumentum továbbításához az `std::forward()` függvényt használjuk

```
template <typename Function, typename Object>
decltype(auto) applyFnToObject(Function func, Object&& obj) {
    return func(std::forward<Object>(obj));
}
```

Ezt a megoldást úgy is nevezik, hogy ‘**universal referencing**’
(univerzális referenciálás)

De miért működik ez? Egyrészt a speciális template feloldási szabályok miatt, amik `Object&&` esetén érvényesülnek.

Ha a függvénynek jobb oldali értéket adunk át, `Object` típusa a reference collapse szabályai miatt jobb oldali ref. lesz. Ha bal oldali értéket, akkor ugyanezért sima bal oldali referencia lesz.

Magyarán, az `obj` változó az `applyFnToObject` függvényen belül mindig vagy bal oldali, vagy jobb oldali, de mindenképpen referencia lesz.

`std::forward()` pedig azt a célt szolgálja, hogy ezt a referencia típust pontosan ugyanolyan referenciaként adjuk át a `func` függvénynek,

Decltype(auto)

Ez a példa mutatja, hogy az átadott érték tényleg lehet bal oldali referencia:

```
1  #include <iostream>
2
3  void otherFunc(int x) { // itt nem referenciat varunk...
4      x = x*1000;;
5  }
6
7  void otherFuncB(int& x) {
8      x = x+1;
9  }
10
11  template <typename Object>
12  void addSomeNumber(Object&& x) {
13      x += 1;
14      otherFunc(std::forward<Object>(x));
15      otherFuncB(std::forward<Object>(x));
16  }
17
18  int main() {
19      int a = 1;
20      addSomeNumber(a);
21      std::cout << a << std::endl;
22      return 0;
23  }
```

```
> clang++-7 -pthread -std=c++17 -o main main.cpp
> ./main
3
> []
```

Decltype(auto)

Itt pedig láthatjuk a perfect forwardingot működés közben:

```
1  #include <iostream>
2
3  template <typename Function, typename Object>
4  decltype(auto) applyFnToObject(Function& func, Object&& obj) {
5      return func(std::forward<Object>(obj));
6  }
7
8  class AddTwo {
9      int value;
10 public:
11     AddTwo() : value{0} {}
12     int operator()(int input) {
13         value = input + 2;
14         return value;
15     }
16 };
17
18 class AddTwoProblematic {
19     int value;
20 public:
21     AddTwoProblematic() : value{0} {}
22     int operator()(int&& input) {
23         value = input + 2;
24         return value;
25     }
26 };
27
28 int main() {
29     int x = 5;
30     AddTwo at;
31     AddTwoProblematic atp;
32     std::cout << applyFnToObject(at, x) << std::endl;
33     std::cout << applyFnToObject(atp, std::move(x)) << std::endl;
34
35     return 0;
36 }
```

```
> clang++-7 -pthread -std=c++17 -o main main.cpp
> ./main
7
7
> []
```

Miért decltype(auto) és nem auto?

Mondtuk, hogy visszatérési érték esetében az auto kulcsszó is használható. De az auto önmagában sosem viszi át a const és referencia tulajdonságokat.

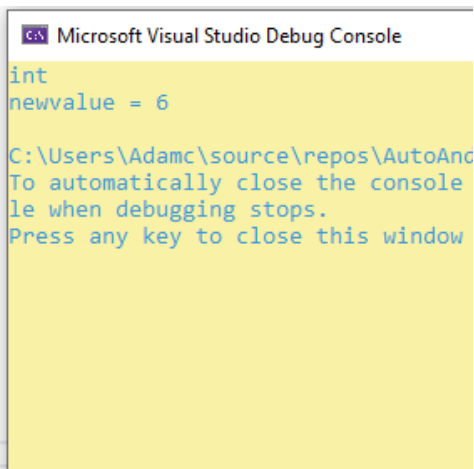
Ha nem azt írom, hogy 'auto&', vagy 'const auto&', hanem csak azt, hogy 'auto' akkor az eredményül kapott típus sohasem lesz sem const, sem referencia. Éppen ez teszi lehetővé, hogy auto esetén is irányítani tudjuk, hogy a változók "másolódnak" vagy "hivatkoznak" és hogy módosíthatóak-e.

```
#include <iostream>

template <typename Object>
decltype(auto) increment(Object&& toIncrement) {
    std::cout << typeid(toIncrement).name() << std::endl;
    toIncrement++;
    return toIncrement;
}

int main() {
    int value = 5;
    int& newvalue = increment(value);
    std::cout << "newvalue = " << newvalue << std::endl;

    return 0;
}
```



```
Microsoft Visual Studio Debug Console

int
newvalue = 6

C:\Users\Adamc\source\repos\AutoAnd
To automatically close the console
le when debugging stops.
Press any key to close this window
```

Mint a korábbi példánál, Object típusa int& lesz!

Végül newvalue csak egy referencia lesz value-ra!

Miért decltype(auto) és nem auto?

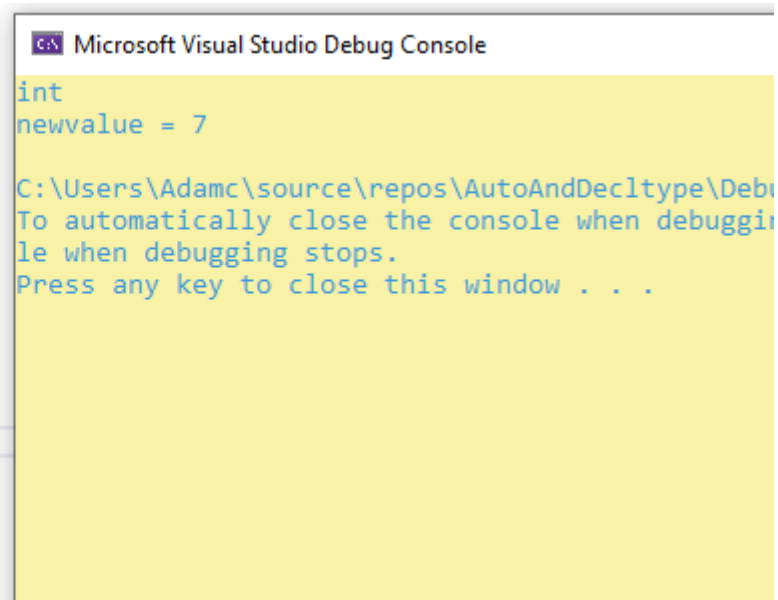
Hogy ez így van, azt az alábbi példa igazolja:

```
#include <iostream>

template <typename Object>
decltype(auto) increment(Object&& toIncrement) {
    std::cout << typeid(toIncrement).name() << std::endl;
    toIncrement++;
    return toIncrement;
}

int main() {
    int value = 5;
    int& newvalue = increment(value);
    value = 7;
    std::cout << "newvalue = " << newvalue << std::endl;

    return 0;
}
```



The screenshot shows the Microsoft Visual Studio Debug Console window. The title bar reads "Microsoft Visual Studio Debug Console". The console output is as follows:

```
int
newvalue = 7

C:\Users\Adamc\source\repos\AutoAndDecltype\Debug
To automatically close the console when debugging
le when debugging stops.
Press any key to close this window . . .
```

Miért decltype(auto) és nem auto?

Ha viszont simán auto-t használunk, a program le se fordul. Object itt is bal oldali referencia lenne ugyan... de az auto nem fog a visszatérési értékből referenciát csinálni – kimásolja, mint egy értéket. És bal oldali referenciát nem lehet jobb oldali értékkel inicializálni!

```
4  #include <iostream>
5
6  template <typename Object>
7  auto increment(Object&& toIncrement) {
8      std::cout << typeid(toIncrement).name() << std::endl;
9      toIncrement++;
10     return toIncrement;
11 }
12
13
14 int main() {
15     int value = 5;
16     int& newvalue = increment(value);
17     std::cout << "newvalue = " << newvalue << std::endl;
18     return 0;
19 }
20
21
```

00% 1 0 0 Messages Build + IntelliSense Ln: 11

Error List

Entire Solution 2 Errors 0 Warnings 0 Messages Build + IntelliSense Search Er

	Code	Description	Project	File	Line
	E0461	initial value of reference to non-const must be an lvalue	AutoAndDecltype	AutoAndDecltype.cpp	16
	C2440	'initializing': cannot convert from 'int' to 'int &'	AutoAndDecltype	AutoAndDecltype.cpp	16

Miért `decltype(auto)` és nem `auto`?

Röviden tehát: az `auto` nem a változó (`toIncrement`) deklarált típust veszi figyelembe, hanem “valami hasonló” típust ad vissza (referencia és `const` levágásával).