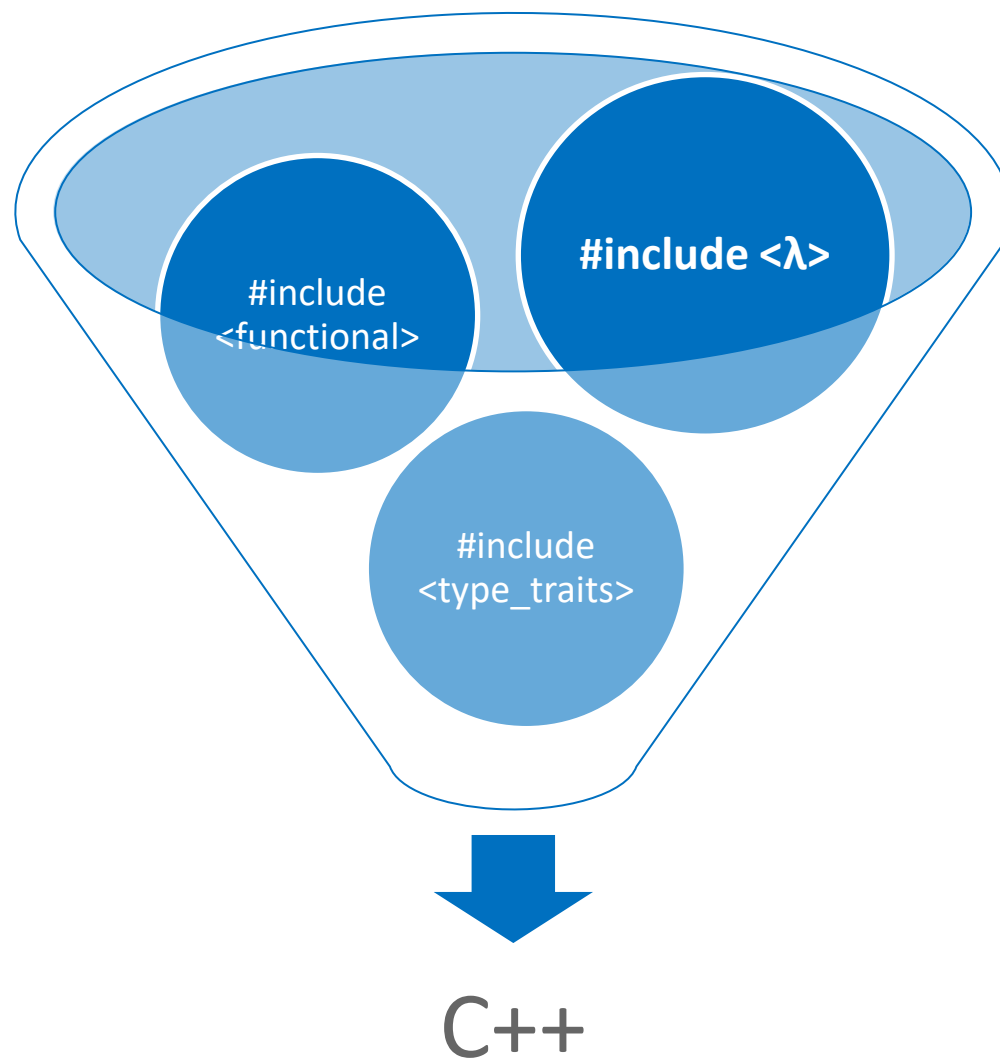


Funkcionális programozás C++-ban

Kik...

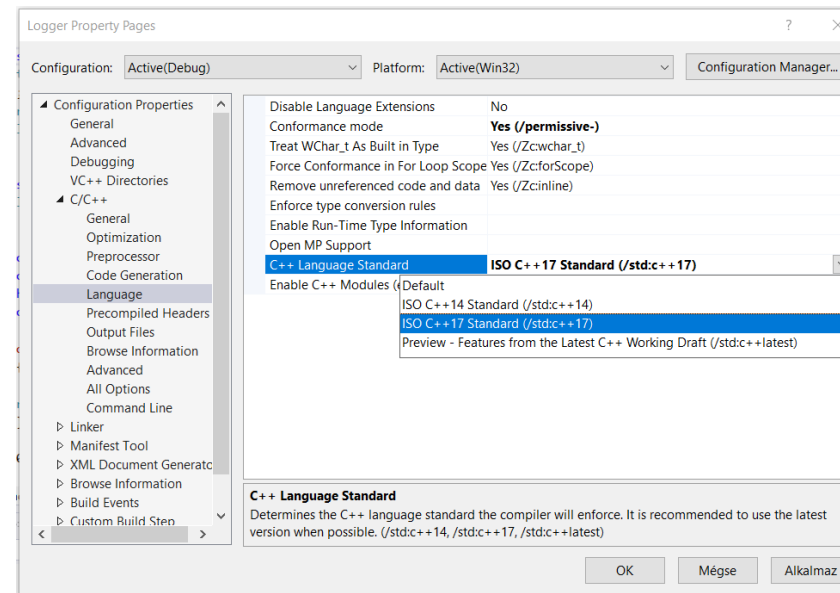
- **Varga Pál**
 - pvara@tmit.bme.hu
 - IB 220
- **Németh Gábor**
 - nemethgab@tmit.bme.hu
 - nemeth.gabor@vik.bme.hu

Mit...



Hogy...

- ▶ előadás + labor
 - ▶ keverve
 - ▶ diák + kódok a GitHub-on elérhetőek
 - ▶ ?
- ▶ C++17, C++20
 - ▶ gcc, clang
 - ▶ -std=c++17
 - ▶ Visual Studio
 - ▶ /std:c++17
 - ▶ on-line fordítók
 - ▶ Compiler Explorer
 - ▶ Coliru
 - ▶ ...



Számonkérés formája

- ▶ 1 db ZH
 - ▶ Helyszínen/Moodle
- ▶ 1 db ellenőrző mérés
 - ▶ Helyszínen/Moodle

- ▶ **opcionális** feladatok
 - ▶ 5 db elméleti
 - ▶ 3 db gyakorlati



- ▶ megajánlott jeles: legjobb öt
- ▶ 5-5 db megajánlott 5-ös, illetve 4-es az elméleti részből
- ▶ pontok beszámítása (30 %)

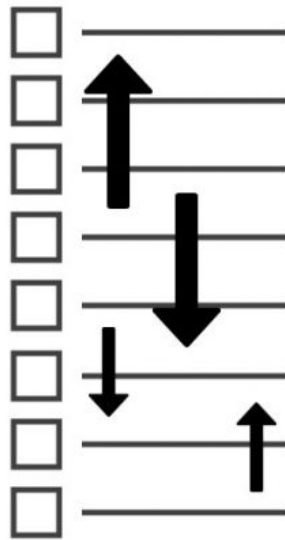
Funkcionális programozás

Bevezetés

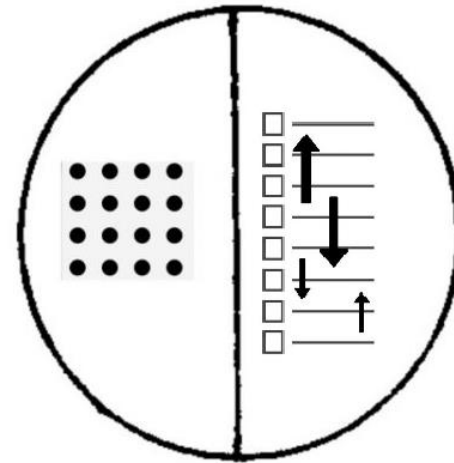
Programozási **paradigmák**

- ▶ imperatív

- ▶ függvények
- ▶ állapot



objektum orientált



- ▶ deklaratív

- ▶ logikai
- ▶ funkcionális

- ▶ 1950-es évek óta
- ▶ adat mozgatása függvényről függvényre..., transzformációk sorozata
- ▶ állapot a gonosz; egy függvény meghívása azonos paraméterekkel mindig azonos eredményt ad

Süssünk sütit!

▶ Imperatív

- ▶ Melegítsük elő a sütőt 175 °C fokra.
- ▶ Vajazzuk ki a tepsit.
- ▶ Válasszuk szét a tojások sárgáját és fehérjét.
- ▶ ...
- ▶ Süssük 30 percig, majd vegyük ki a sütőből.

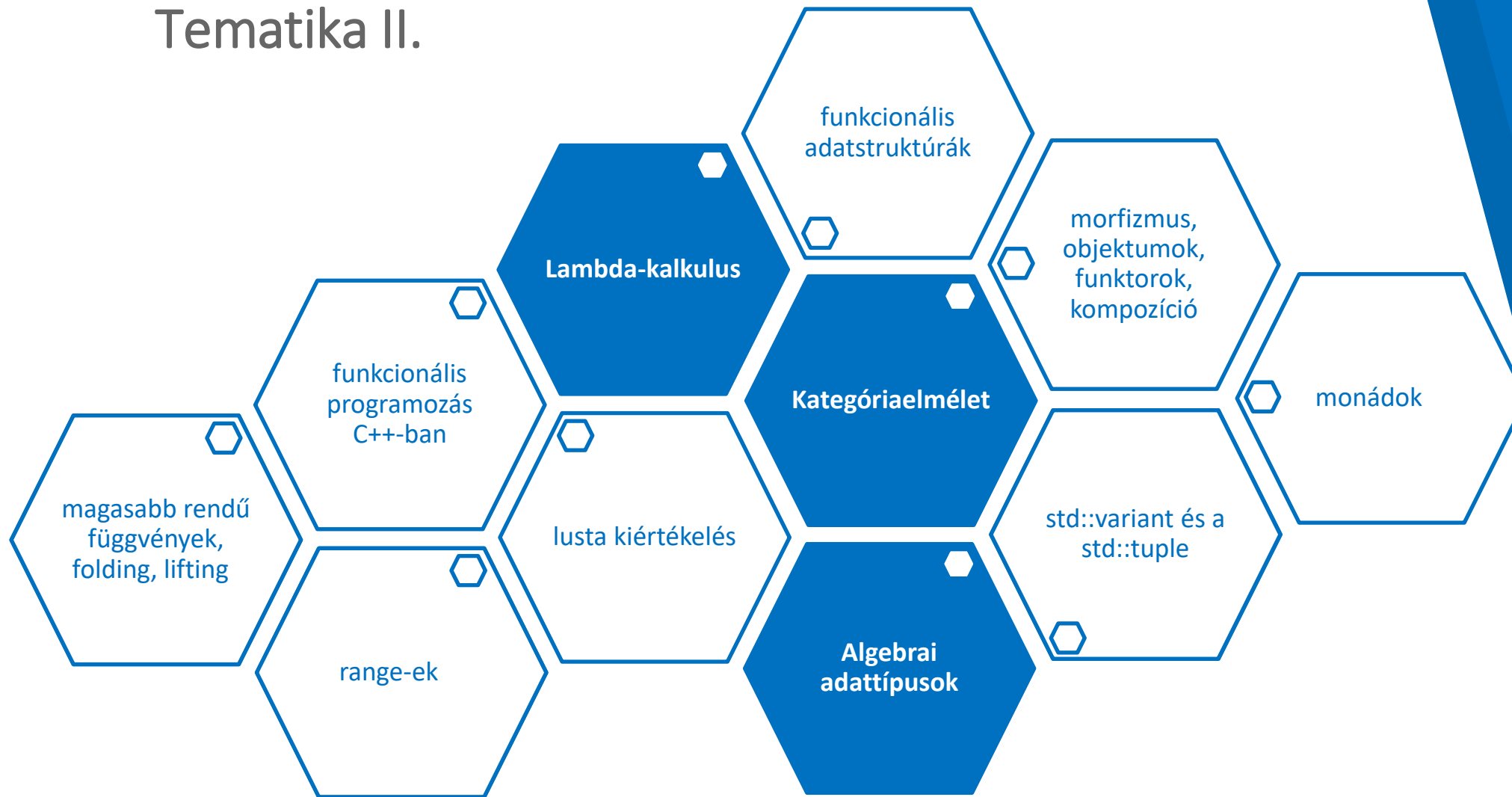
▶ Funkcionális

- ▶ A süti egy olyan előkészített sütemény, amely előmelegített sütőben készült 30 percig.
- ▶ Az előmelegített sütő 175 °C-ra volt melegítve.
- ▶ Az előkészített sütemény olyan...
- ▶ ...

Tematika I.

- ▶ 1. hét
 - ▶ A **mozgatás szemantika** és a **tökéletes továbbítás**. Az lvalue és rvalue referenciák. **Sablonok**. Copy elision, RVO.
- ▶ 2. hét
 - ▶ **Variadikus sablonok**. Sablonparaméterek feldolgozása fordítási idejű rekurzióval. Fold kifejezések (fold expression). **Fordítási idejű döntés**: constexpr if, sablonok specializációja, tag dispatching, SFINAE.

Tematika II.

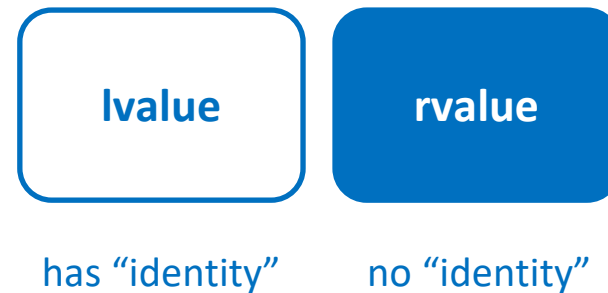


Mozgatás szemantika

Érték kategóriák

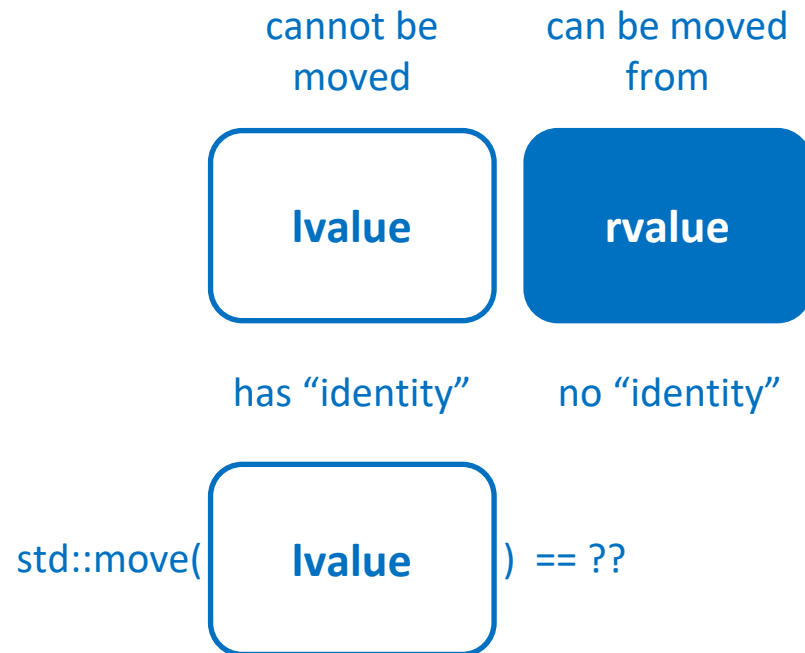
Érték kategóriák

- ▶ CLP
 - ▶ **lvalue**, **rvalue**
 - ▶ **l** = értékadás bal oldala
 - ▶ **r** = értékadás jobb oldala
 - ▶ Christopher Strachey
- ▶ C
 - ▶ **lvalue**, minden más
 - ▶ **l** = lokátor, objektumot azonosít
 - ▶ Dennies Ritchie
- ▶ C++98
 - ▶ **lvalue**, **rvalue**
 - ▶ csak konstans referencia köthet **rvalue**-hoz



C++11: A probléma

- ▶ mozgás szemantika
 - ▶ közbenső, de érvényes állapot
 - ▶ **rvalue** referencia



lvalue

value

has object
has identity

"long" lifetime

not movable
polymorphic (ST != DT)

can be cv-qual

Avalue

dvalue

Dmvalue

has object
has identity

expiring

movable
polymorphic

cv-qual

tuvalue

Bvalue

F Cvalue

class
has object
cv-qual
svalue

G lvalue

non class
no object
not cv-qual

Ervalue

does not have identity

may have object

short lifetime

not movable
not polymorphic

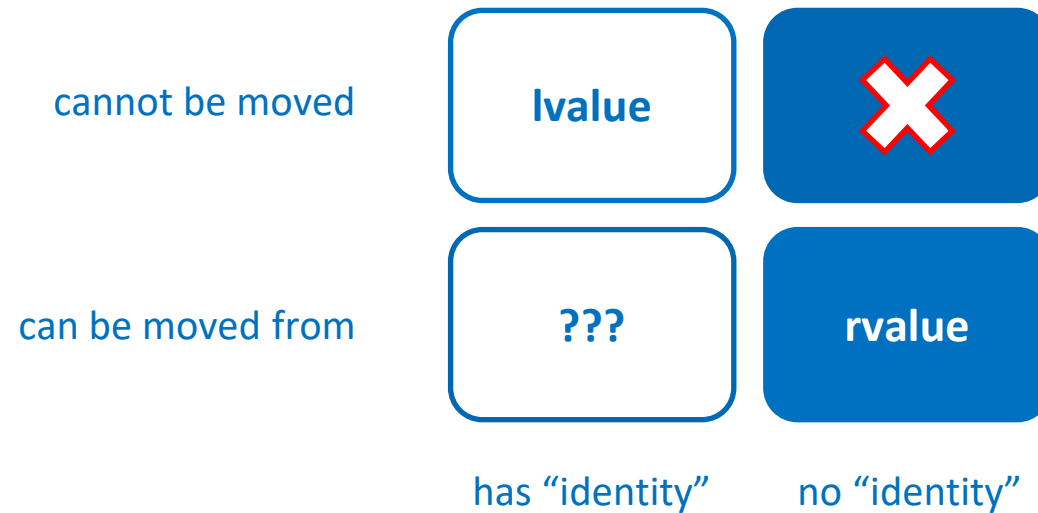
non-class not cv-qual

new beginning

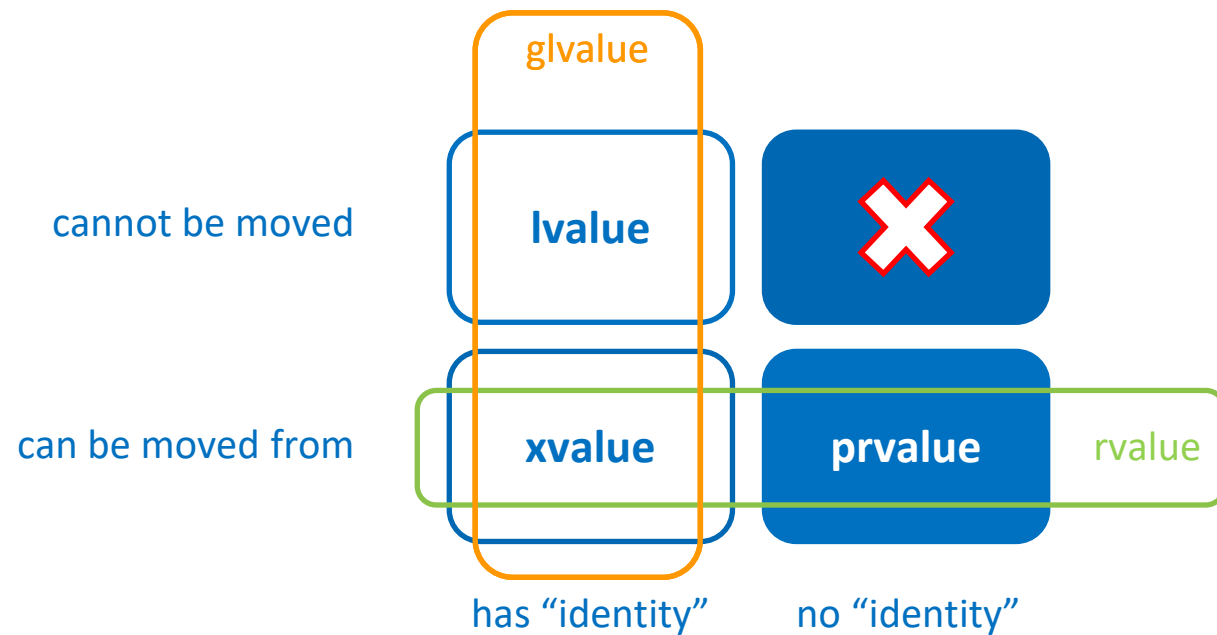
near end

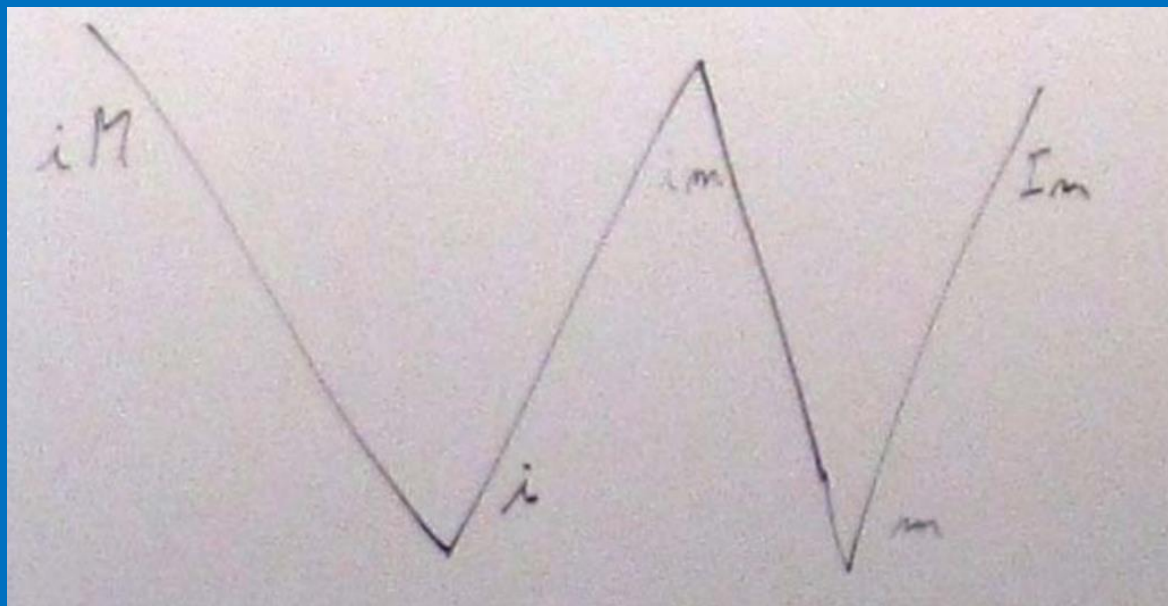
C++11: A megoldás I.

- ▶ mozgatus és az identitás ortogonális tulajdonságok



C++11: A megoldás II.

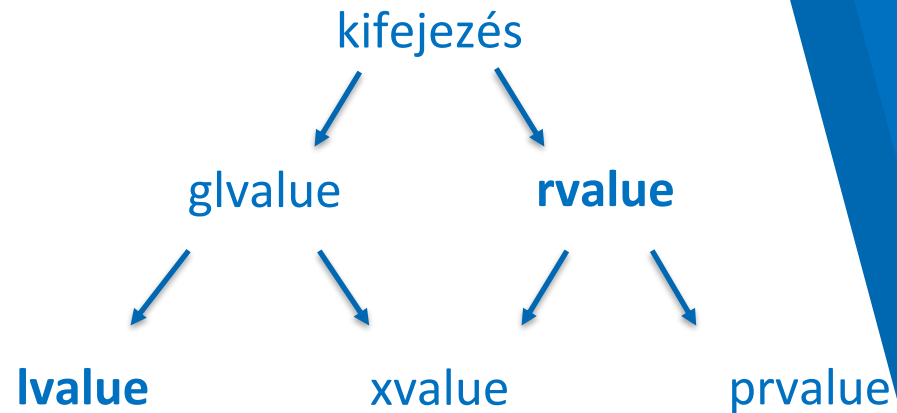




C++11: Érték kategóriák I. - glvalue

- ▶ glvalue

- ▶ konkrét objektumra hivatkozik
 - ▶ egy kifejezés referenciát (T&) ad vissza
 - ▶ egy változóra hivatkozik annak nevével
 - ▶ egy objektum rvalue referenciája (T&&)



```
int& foo(){  
    static int i=0;  
    return ++i;  
}
```

```
int a;
```

```
struct Bar{ int m; };
```

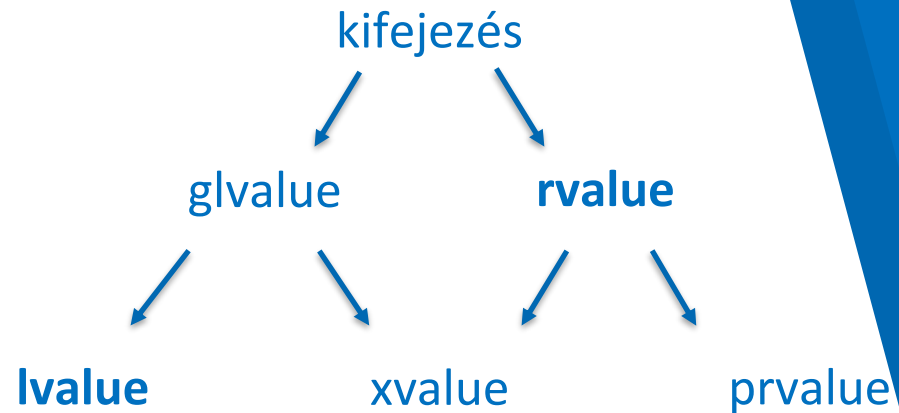
```
Bar bar;
```

```
...  
foo()           // statikus i-re hivatkozik, int& a visszatérési érték  
...  
a  
std::move(a)  
...  
bar  
bar.m
```

C++11: Érték kategóriák II. - xvalue

▸ xvalue

- glvalue
- erőforrásai újrahasználhatóak
 - közel van élettartamának a végéhez

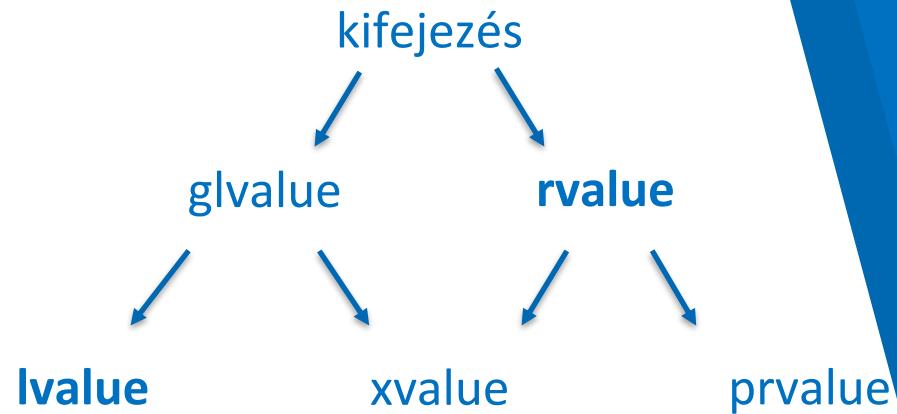


```
struct Foo{/* definition */};
```

```
int main() {  
    Foo a;  
    std::move(a);           // xvalue  
    static_cast<Foo&&>(a);   // xvalue (explicit cast)  
  
    Foo arr[10] = {};  
    std::move(arr)[0];      // xvalue  
}
```

C++11: Érték kategóriák III. - prvalue

- ▶ prvalue
 - ▷ “pure” rvalues.
 - ▷ ideiglenes változók (C++17 előtt)

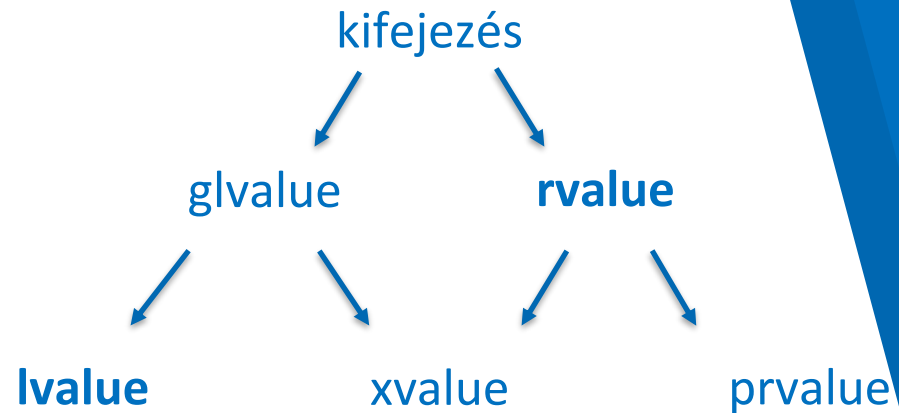


- ▶ C++17
 - ▷ kiértékelése egy másik objektumot inicializál
 - ▷ egy operátor operandusát számolja ki
 - ▷ `T var = T();` // C++17 nincs mozgatus!!

C++11: Érték kategóriák IV.

- ▶ ++a
- ▶ a++
- ▶ "Hello World!"
- ▶ a = b
- ▶ a += b
- ▶ a == b
- ▶ [](int x){ return x * x; }
- ▶ &a

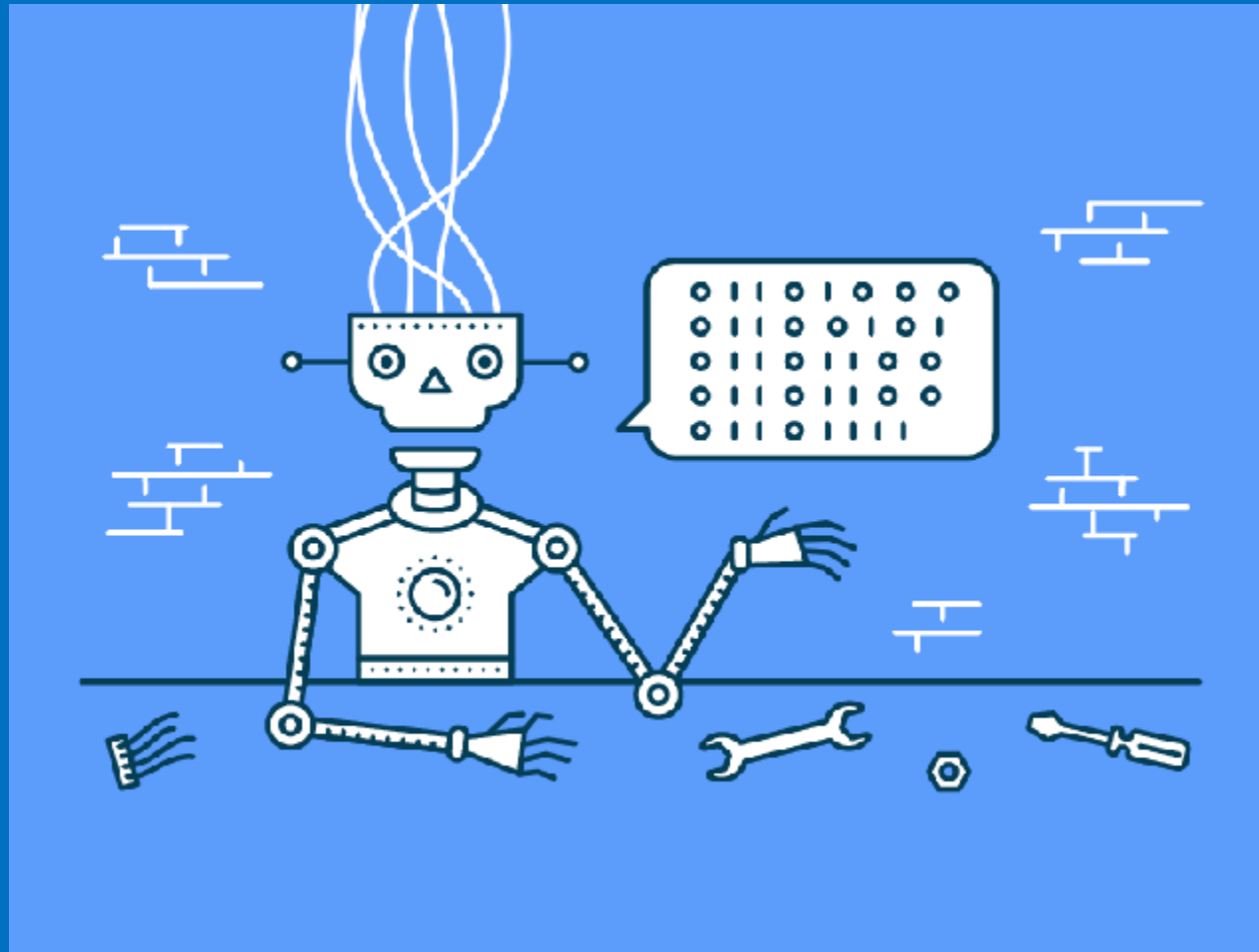
- ▶ lvalue
- ▶ prvalue
- ▶ lvalue
- ▶ lvalue (beépített op.)
- ▶ lvalue
- ▶ prvalue (beépített op.)
- ▶ prvalue
- ▶ prvalue (beépített op.)

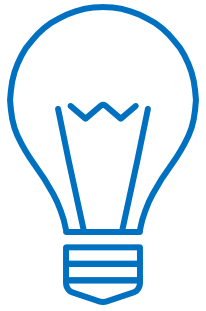


Mozgatás szemantika

Osztályok,
konstruktorok

ex_0: move ctor





```
#include <utility>
```

```
std::exchange
```

```
std::swap
```


The compiler supplies...

If you write...

	None	dtor	Copy-ctor	Copy-op=	Move-ctor	Move-op=
dtor	✓	♦	✓	✓	✓	✓
Copy-ctor	✓	✓	♦	✓	✗	✗
Copy-op=	✓	✓	✓	♦	✗	✗
Move-ctor	✓	✗	Overload resolution will result in copying		♦	✗
Move-op=	✓	✗			✗	♦

Copy operations
are independent...

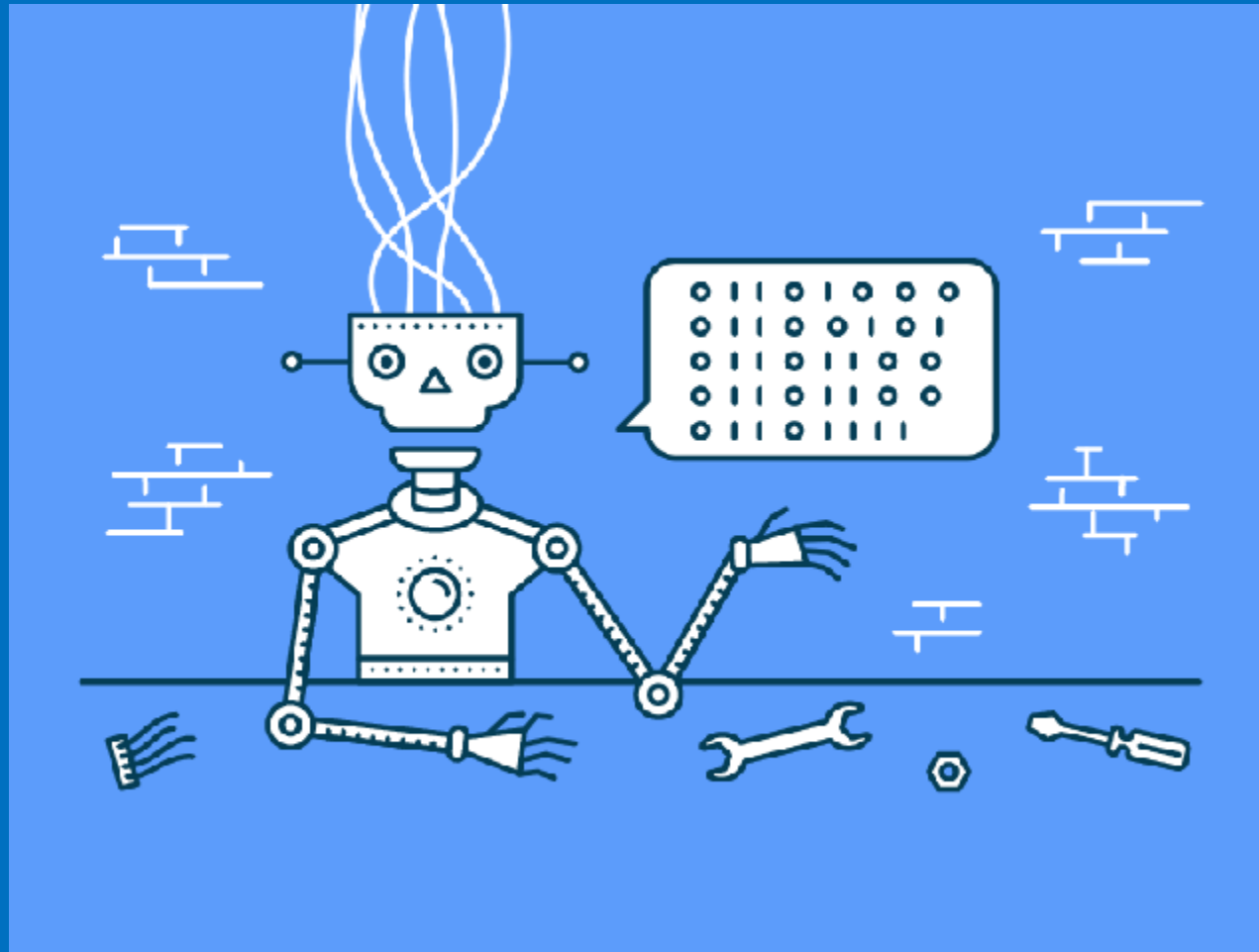
Move operations
are not.

Copy elision

Copy elision

- ▶ a másoló-, illetve mozgó konstruktor hívásának az elhagyása
- ▶ mellékhatás eltűnhet
- ▶ copy elision
 - ▶ kötelező
 - ▶ nem kell másoló- és mozgó konstruktor
 - ▶ megengedett

ex_1: html builder





```
T f() {  
    return T{ };  
}
```

```
f();
```

```
T x = T{ T{ f() } };
```

RVO és NRVO

- ▶ RVO
 - ▶ kötelező
 - ▶ `return T{};`
- ▶ NRVO
 - ▶ visszatérési érték típusa (cv-kvalifikációt ignorálva) ugyanaz, mint a visszaadott automatikus változónak
 - ▶ gyakorlatilag minden implementáció használja
 - ▶ pl. `std::vector<T>` visszaadása

Sablonok

Függvénytípusok,
osztálysablonok,
dedukció

Sablonok

- ▶ a sablonok közvetlenül támogatják a **generikus** programozást
 - ▶ a típusok paraméterek


```
template<typename T> auto my_fun(const T &value) {  
    ...  
}
```

```
template<typename T> class MyClass {  
    private:  
        T data;  
    public:  
        ...  
}
```


Típusok dedukciója

- ▶ `template<typename T> void f(ParamType param);`

- ▶ `f(expr);`



T
const T&
const T*
T&&

- ▶ A fordítás során két típust kell kitalálni: T és ParamType típusát

Típusok dedukciója: ParamType referencia

```
int x = 42;  
const int cx = x;  
const int &rx = x;
```

} feledkezzünk meg az esetleges referenciáról és illesszük

```
template<typename T> void fun(T &param);
```

```
fun(x);           // T = int  
fun(cx);          // T = const int  
fun(rx);          // T = const int
```

```
template<typename T> void fun(const T &param);
```

```
fun(x);           // T = int  
fun(cx);          // T = int  
fun(rx);          // T = int
```

Típusok dedukciója: ParamType mutató

```
int x = 42;  
const int *px = &x;
```

```
template<typename T> void fun(T *param);
```

```
fun(x);           // T = int  
fun(px);          // T = const int
```

```
template<typename T> void fun(const T *param);
```

```
fun(x);           // T = int  
fun(px);          // T = int
```

Típusok dedukciója: ParamType forwarding referencia I.

```
int x = 42;  
const int cx = x;  
const int& rx = x;
```

```
template<typename T> void fun(T &&param);
```

fun(x);	// T = ??
fun(cx);	// T = ??
fun(rx);	// T = ??
fun(42);	// T = ??

Típusok dedukciója: ParamType forwarding referencia II.

► `template<typename T> void f(T&& param);`

`int& param`

`T&& = int&`

`T = int&`

`T&& = int&&& = int&`

Típusok dedukciója: ParamType forwarding referencia I.

```
int x = 42;  
const int cx = x;  
const int& rx = x;
```

```
template<typename T> void fun(T &&param);
```

fun(x);	// T = int&
fun(cx);	// T = const int&
fun(rx);	// T = const int&
fun(42);	// T = int

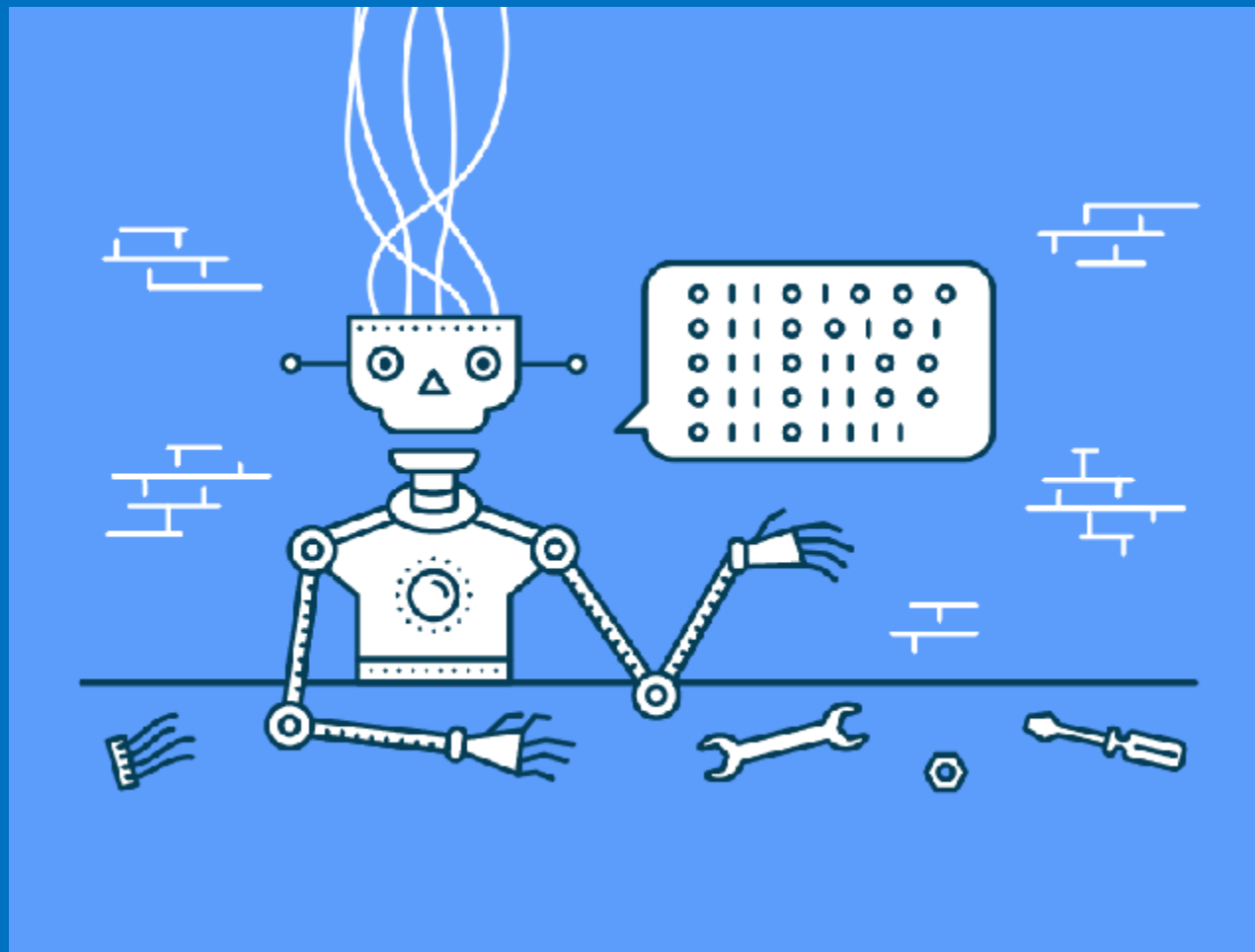
Típusok dedukciója: ParamType érték

```
int x = 42;  
const int cx = x;  
const int& rx = x;
```

```
template<typename T> void fun(T param);
```

fun(x);	// T = int
fun(cx);	// T = int
fun(rx);	// T = int
fun(42);	// T = int

ex_2: type deduction





Type & Value detector

Működési elv: nincs definíció a típushoz, ezért ha példányosítjuk egy template-ben, fordítási hibát fog okozni, de úgy, hogy a compiler diagnosztikában egyértelműen kiolvasható a típusból a keresett paraméter típus vagy érték

```
template<typename T>  
struct TD;
```

```
TD<U> t;
```

```
template<auto V>  
struct VD;
```

```
VD<N> t;
```

Tökéletes továbbítás

A std::forward

lvalue vagy rvalue referencia?

```
void print(int &i) {  
    ...  
}
```

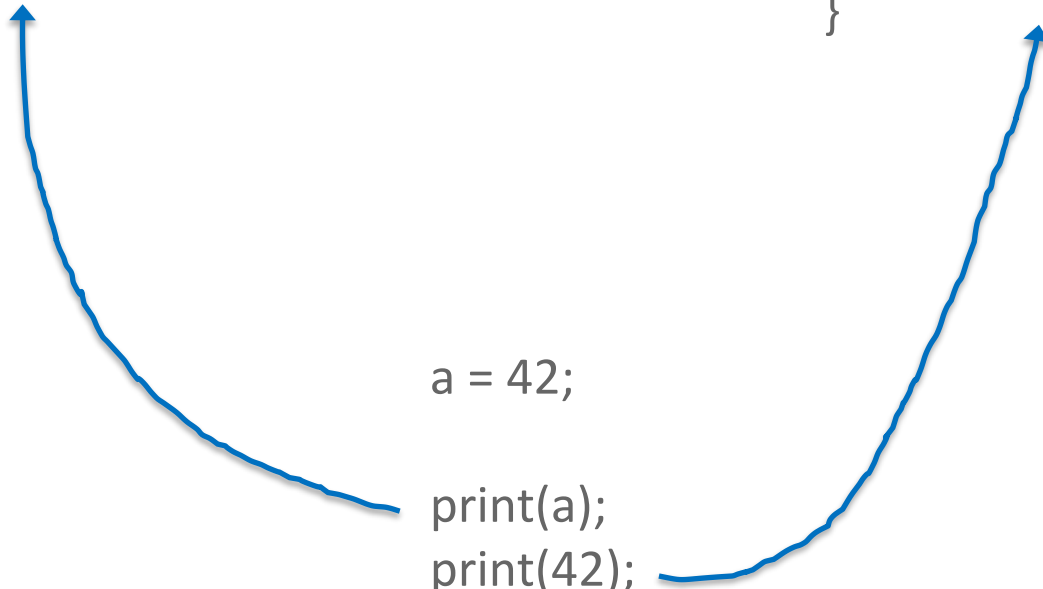
```
void print(int &&i) {  
    ...  
}
```

a = 42;

print(a);

print(42);

print(std::move(a));



Tökéletes továbbítás I.

```
void print(int &i) {  
    ...  
}
```

???

```
template<typename T> void proxy(T &&i) {  
    print(i);  
}
```

```
void print(int &&i) {  
    ...  
}
```

a = 42;

proxy(a);
proxy(42);

proxy(std::move(a));

Forwarding referencia

```
void print(int &i) {  
    ...  
}
```

```
void print(int &&i) {  
    ...  
}
```

```
template<typename T> void proxy(T &&i) {  
  
    print(std::forward<T>(i));  
  
}
```

std::move
std::forward

static_cast<std::remove_reference<T>::type&&>
static_cast<T&&>

Tökéletes továbbítás II.

```
void print(int &i) {  
    std::cout << "int&";  
}
```

```
template<typename T>  
void f(T &&p){ print(p); }
```

```
template<typename T>  
void g(T &&p){ print(std::move(p)); }
```

```
template<typename T>  
void h(T &&p){ print(std::forward<T>(p)); }
```

```
void print(int &&i) {  
    std::cout << "int&&";  
}
```

```
int a = 20;
```

```
f(a); f(20);  
g(a); g(20);  
h(a); h(20);
```

Köszönöm a figyelmet!

Folytatjuk...