9. fejezet

Hibakezelés

Haladó Alkalmazott Programozás
ELTE, 2019

- std::optional

std::optional

Az std::optional egy olyan osztály template, ami vagy tárol egy adott típusú értéket, vagy nem (azaz üres).

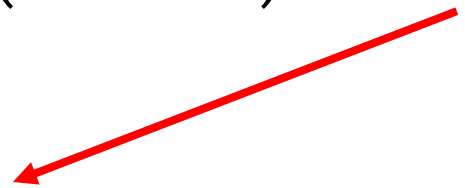
Hasznos, ha azt akarjuk kifejezni, hogy egy függvény általában egy T típusú értéket ad vissza, de néha hiba történhet, és akkor nincs visszatérési érték.

std::optional

```cpp
#include <optional>
#include <cmath>

std::optional<double> correct_sqrt(double x)
{
  if(!std::isfinite(x) || x < 0.0)
  {
    return std::optional<double>{};
  }
  else
  {
    return std::optional<double>{ std::sqrt(x) };
  }
}
```

std::optional

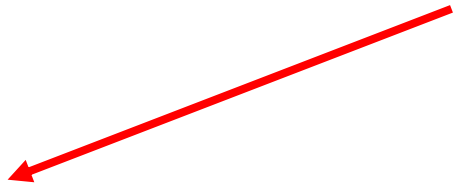
```
#include <optional>
#include <cmath>

std::optional<double> correct_sqrt(double x)
{
  if(std::isfinite(x) && x >= 0.0)
  {
    return std::optional<double>{ std::sqrt(x) };
  }
  else
  {
    return std::optional<double>{};
  }
}
```



Ha nem végtelen,
nem NaN,
és nem negatív, akkor
értelmes a gyökvonás

std::optional

```
#include <optional>
#include <cmath>

std::optional<double> correct_sqrt(double x)
{
  if(std::isfinite(x) && x >= 0.0)
  {
    return std::optional<double>{ std::sqrt(x) };
  }
  else
  {
    return std::optional<double>{};
  }
}
```



Egyébként üres értéket adunk vissza

std::optional

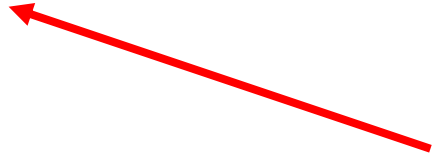
```
#include <optional>
#include <cmath>

std::optional<double> correct_sqrt(double x)
{
  if(std::isfinite(x) && x >= 0.0)
  {
    return std::sqrt(x);
  }
  else
  {
    return std::nullopt;
  }
}
```

Kicsit rövidebben
kifejezve ugyan ez...

std::optional

Hasznos és hatékony lekérdezés az optional-ból a következő:

```
auto f(std::optional<double> const& o)
{
  return o.value_or(0.0);
}
```



Ha van benne érték, visszaadja azt, ha üres volt, akkor azt az értéket adja, amit itt megadunk neki!

- std::variant

std::variant

Az std::variant kb. az optional, több esettel, mindig pontosan egyet tárol a felsorolt template paraméterek közül.

std::variant

Példa:

Mi lenne szabatosan kifejezve az év elejei másodfokú egyenlet megoldó visszatérési típusa?

std::variant

Példa:

Mi lenne szabatosan kifejezve az év elejei másodfokú egyenlet megoldó visszatérési típusa?

```
struct NoSolution{};

std::variant<NoSolution, double, std::pair<double, double>>
```

std::variant

Példa:

Mi lenne szabatosan kifejezve az év elejei másodfokú egyenlet megoldó visszatérési típusa?

```
struct NoSolution{};
```

A variantban nem lehet `void`!
Ezért a „nincs megoldás” reprezentálásához bevezetünk egy üres struct-ot.

```
std::variant<NoSolution, double, std::pair<double, double>>
```

std::variant

Példa:

Mi lenne szabatosan kifejezve az év elejei másodfokú egyenlet megoldó visszatérési típusa?

```
struct NoSolution{};
```

Ha 1 megoldás van, akkor egy double van a variantban

```
std::variant<NoSolution, double, std::pair<double, double>>
```

std::variant

Példa:

Mi lenne szabatosan kifejezve az év elejei másodfokú egyenlet megoldó visszatérési típusa?

```
struct NoSolution{};
```

Ha 2 megoldás van, akkor egy std::pair, amiben két double van

```
std::variant<NoSolution, double, std::pair<double, double>>
```

std::variant


```
#include <cmath>
#include <variant>
struct NoSolution{};

std::variant<NoSolution, double, std::pair<double, double>>

solve(double a, double b, double c)
{
  if( double sqb = b*b, discr = sqb-4.0*a*c; discr < 0.0 ){ return NoSolution{}; }
  else if(discr == 0.0){ return -b / (2.0*a); }
  else
  {
    double sqrt_discr = std::sqrt(discr);
    return std::make_pair( (-b + sqrt_discr)/(2.0*a),
                           (-b - sqrt_discr)/(2.0*a) );
  }
}
```

std::variant

Hogyan vesszük ki a különböző esetekben tárolt különböző értékeket a variant-ból?

std::variant

Ha egy lehetőség kell, és különbözőek a tárolt típusok akkor get_if:

```
std::variant<NoSolution, double, std::pair<double, double>> v;

if(double* p = std::get_if<double>(&v))
{
  std::cout << "Az egyetlen megoldás: " << *p << '\n';
}
else
{
  std::cout << "Vagy nincs megoldás, vagy kettő van" << '\n';
}
```

std::variant

Ha egy lehetőség kell, és index alapján szeretnénk akkor is get_if:

```
std::variant<NoSolution, double, std::pair<double, double>> v;

if(double* p = std::get_if<1>(&v))
{
  std::cout << "Az egyetlen megoldás: " << *p << '\n';
}
else
{
  std::cout << "Vagy nincs megoldás, vagy kettő van" << '\n';
}
```

std::variant

Lekérdezések:

```
std::variant<NoSolution, double, std::pair<double, double>> v;

size_t idx = v.index();

bool two_sol =
          std::holds_alternative<std::pair<double, double>>(v);
```

std::variant

Lekérdezések:

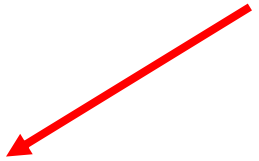
```
std::variant<NoSolution, double, std::pair<double, double>> v;

size_t idx = v.index();

bool two_sol =
          std::holds_alternative<std::pair<double, double>>(v);
```

0-tól számolt indexe a tárolt esetnek

std::variant

Lekérdezések:

```
std::variant<NoSolution, double, std::pair<double, double>> v;

size_t idx = v.index();

bool two_sol =
          std::holds_alternative<std::pair<double, double>>(v);
```



Adott típust éppen tárol-e?
(Ha egyediek a típusok!)

std::variant

Minden eset lekezelése: std::visit

```
std::visit( [](auto arg){ ... }, std::variant<...> );
```

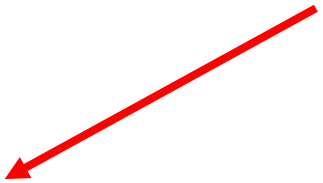
A lambdának minden esetet tudnia kell fogadnia, ezért ez template függvény, vagy generikus lambda kell legyen.

std::variant

```
std::variant<NoSolution, double, std::pair<double, double>> v;

std::visit( [](auto x)
{
  using T = decltype(x);

  if constexpr (std::is_same_v<T, NoSolution>)
  {
    std::cout << "Nem volt megoldas\n";
  }
  else if constexpr (std::is_same_v<T, double>)
  {
    std::cout << "Egy megoldas volt: " << x << "\n";
  }
  else if constexpr (std::is_same_v<T, std::pair<double, double>>)
  {
    std::cout << "Ket megoldas volt: " << x.first << " es " << x.second << "\n";
  } }, v);
```

std::variant

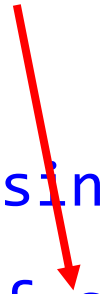
Névhez kötjük x típusát:
A decltype egy kifejezés típusát adja vissza,
a using bevezet egy típus szinonimát.



```
using T = decltype(x);

if constexpr (std::is_same_v<T, NoSolution>)
{
  std::cout << "Nem volt megoldas\n";
}
```

std::variant

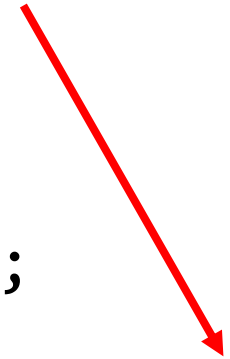
Esetszétválasztást típusokon csak fordítás időben ismert feltétel alapján tudunk tenni. Ahhoz, hogy ez egyszerű legyen az if-ágaihoz ki kell írni a constexpr kulcsszót, ez azt jelenti, hogy az ottani feltételt ki kell tudni értékelni fordításidőben (nem függhet futás idejű értéktől!)



```
using T = decltype(x);

if constexpr (std::is_same_v<T, NoSolution>)
{
  std::cout << "Nem volt megoldas\n";
}
```

std::variant

Az `std::is_same_v` egy template, ami egy fordítás idejű igaz/hamis értéket ad vissza, ha a két típus argumentuma ugyan az a típus, hamisat, ha nem egyeznek



```
using T = decltype(x);
if constexpr (std::is_same_v<T, NoSolution>)
{
  std::cout << "Nem volt megoldas\n";
}
```

std::variant

```
std::variant<NoSolution, double, std::pair<double, double>> v;

std::visit( [](auto x)
{
  using T = decltype(x);
  if constexpr (std::is_same_v<T, NoSolution>)
  {
    std::cout << "Nem volt megoldas\n";
  }
  else if constexpr (std::is_same_v<T, double>)
  {
    std::cout << "Egy megoldas volt: " << x << "\n";
  }
  else if constexpr (std::is_same_v<T, std::pair<double, double>>)
  {
    std::cout << "Ket megoldas volt: " << x.first << " es " << x.second << "\n";
  } }, v);
```

Az esetszétválasztásnál meg se nézi a fordító azt az ágat, aminek épp nem egyezik a típusa a bejövővel!

Ezért különbözhetnek teljesen!

Az `optional` és a `variant` előnye, hogy NEM használnak dinamikus memóriát.

A tárolt objektumok/esetek simán „bennük” vannak elhelyezve, ezért minimális költségük van.

A variant mérete kb. a legnagyobb méretű eset + egy egész szám.

- Kivételkezelés

Kivételkezelés

Kivételnek (exception) nevezzük azt a program futása során keletkező állapotot (általában hibát), amelyet nem a szokványos vezérlési utakon (control flow) akarunk / tudunk kezelni.

A kivételkezelés egy alternatív vezérlési út, amikor a végrehajtás visszafejti a hívási láncot és lehetőséget biztosít arra, hogy a kivételes helyzetet a hívó függvények valamelyike egy erre a célra megadott kezelő részen feldolgozza.

Kivételkezelés

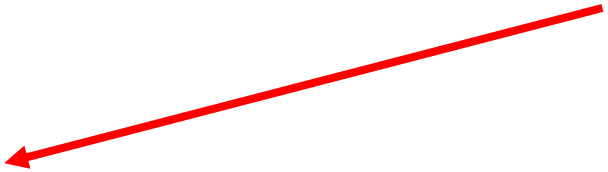
C++-ban a minimum, amit ismerni kell a következő három kulcsszó:

```
try
{
  throw ...;
}
catch( ... )
{

}
```

Kivételkezelés

C++-ban a minimum, amit ismerni kell a következő három kulcsszó:

```
try
{
   throw ...;
}
catch( ... )
{

}
```



A `throw` kulcsszó váltja ki a kivételt és hozza létre a kivétel objektum példányát.

Bármilyen objektumot használhatunk erre a célra, bármilyen információt belepakolhatunk.

Kivételkezelés

C++-ban a minimum, amit ismerni kell a következő három kulcsszó:

```
try
{
  throw ...;
}
catch( ... )
{

}
```

A `throw` által létrehozott objektum addig utazik felfelé a hívási listán (call stack), amíg egy `try` blokk nem veszi körbe valahol.

Kivételkezelés

C++-ban a minimum, amit ismerni kell a következő három kulcsszó:

```
try
{
  throw ...;
}
catch ( ... )
{

}
```

Az első befoglaló `try` blokk után jövő `catch`-blokkokat sorra nézi a rendszer és kiválasztja azt, amelyiknek a típusa egyezik a kivétel típusával.

Az ebben foglaltakat végrehajtja.

Kivételkezelés

C++-ban a minimum, amit ismerni kell a következő három kulcsszó:

```
try
{
  throw ...;
}
catch ( ... )
{

}
```

Ha a main-ben sem kezelte le a kivételt egyetlen try-catch blokk sem, akkor a program leállításra kerül.

Általában kapunk egy értesítést az operációs rendszertől, hogy kezeletlen kivétel történt (unhandled exception)

Kivételkezelés

```
#include <vector>
#include <iostream>

struct no_first{};

int first( std::vector<int> const& v )
{
  if(v.size() == 0){ throw no_first{}; }
  else             { return v[0]; }
}

int main()
{
  try{ return first( { } ); }
  catch(no_first const& e)
  {
    std::cout << "No first element in vector\n";
  }
}
```

Kivételkezelés

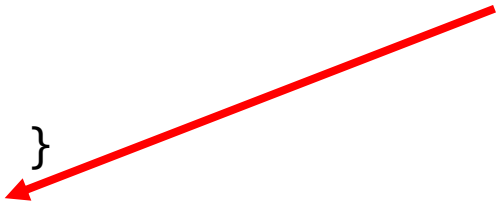

```
#include <vector>
#include <iostream>

struct no_first{};

int first( std::vector<int> const& v )
{
  if(v.size() == 0){ throw no_first{}; }
  else             { return v[0]; }
}

int main()
{
  try{ return first( { } ); }
  catch(no_first const& e)
  {
    std::cout << "No first element in vector\n";
  }
}
```

Ha üres a vector, akkor kivételt váltunk ki

Ez most meg is történik,
mert üres vector-t hozunk létre.

Kivételkezelés

```
#include <vector>
#include <iostream>

struct no_first{};

int first( std::vector<int> const& v )
{
  if(v.size() == 0){ throw no_first{}; }
  else             { return v[0]; }
}

int main()
{
  try{ return first( { } ); }
  catch(no_first const& e)
  {
    std::cout << "No first element in vector\n";
  }
}
```



A catch blokk elkapja a kivételt, és kiír egy üzenetet.

Kivételkezelés

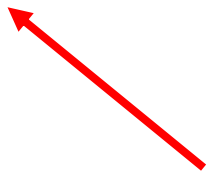

```
#include <vector>
#include <iostream>


int first( std::vector<int> const& v )
{
  return v.at(0);
}


int main()
{
  try{ return first( { } ); }
  catch(std::out_of_range const& e)
  {
    std::cout << e.what() << "\n";
  }
}
```

A vector at metódusa is kivételt vált ki túlindexelésnél.

De másik fajta kivétel típust dob.

Kivételkezelés


```
catch(std::out_of_range const& e)
{
  std::cout << e.what() << "\n";
}
```



Az exception objektumoknál szokás egy what() függvényt definiálni, ami vissza ad egy magyarázatot tartalmazó string-et, hogy mi történt.

Kivételkezelés

A kivétel kezelés nagyon sok mindent csinál a háttérben, többek között garantálja, hogy a throw előtt létrehozott objektumok a létrehozásuk fordított sorrendjében felszabadításra kerülnek, és sok mást még.

Ez viszont egy nagyon költséges folyamat, ezért teljesítmény kritikus kódban kerülendő a használata.

- Stream hibakezelés

Stream hibakezelés

A stream-ek használata közben előforduló hibákat kezelhetjük kivételként is, ha ezt beállítjuk a stream objektumon:

```
try
{
  std::ifstream in("in.txt");
  in.exceptions(std::ifstream::failbit);
}
catch (std::ios_base::failure& fail)
{
  std::cout << fail.what() << "\n";
}
```

Stream hibakezelés

A stream-ek használata közben előforduló hibákat kezelhetjük kivételként is, ha ezt beállítjuk a stream objektumon:

Itt mondjuk meg, hogy ezen a streamen milyen hibákról akarunk exception formájában értesülni

```
try
{
  std::ifstream in("in.txt");
  in.exceptions(std::ifstream::failbit);
}
catch (std::ios_base::failure& fail)
{
  std::cout << fail.what() << "\n";
}
```