C++

6. 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

Új típusok C++17-ben

A C++17-es szabvány több új típust is bevezetett. Most az alábbiak kerülnek bemutatásra:

- **std::optional**: olyan típus, aminek vagy van értéke, vagy nincs (üres)
- **std::variant**: olyan típus, amely egy adott típushalmazból bármilyen típusú értéket tartalmazhat
- **std::any**: olyan típus, amely bármilyen típusú értéket tartalmazhat

Nézzük meg ezeket egyenként, részletesebben!

std::optional motivációja

Hogyan kezeljük, ha egy változónak nem mindig van értelmes értéke az élettartalma során?

- Mutató esetén nullptr, de csak emiatt nem érdemes heap-re tenni az adatot
- Felveszünk mellé egy bool változót, ami jelzi, hogy érvényes-e
  - Plusz kód, plusz hibalehetőség

És mit csináljunk, ha egy függvény nem mindig ad vissza értéket?

- Ismét megoldható nullptr visszaadásával, de akkor más esetben is mutatót kell visszaadni
- Visszaadhat egy std::pair<T, bool>-t, ahol a bool jelzi az érvényességet, de ez nem túl elegáns és T konstruktálását igényli
- Visszaadhat egy hibakódot, ami nem érvényes adat, pl. index helyett –1-et
  - Nem mindig lehet ilyen értéket megfeleltetni, az eredményt konstruktálni kell, dokumentálni kell, és vigyázni, hogy ellenőrizzék a spec. értéket
- Dobhat kivételt, ha nem tud értéket visszaadni
  - Így már nem kell konstruktálni a visszaadandó értéket, de ismét a dokumentációra kell támaszkodni, nincs kikényszerítve az ellenőrzés

Erre jó az <optional> headerben található std::optional<T> template típus, amely vagy tartalmaz egy T értéket, vagy nem

- Memóriahasználata nem igényel heap allokációt – ugyanúgy a stacken foglal helyet a T típusú érték, mint egy mezei lokális változó (viszont definiálja a * és -> operátorokat az érték eléréséhez)

std::optional

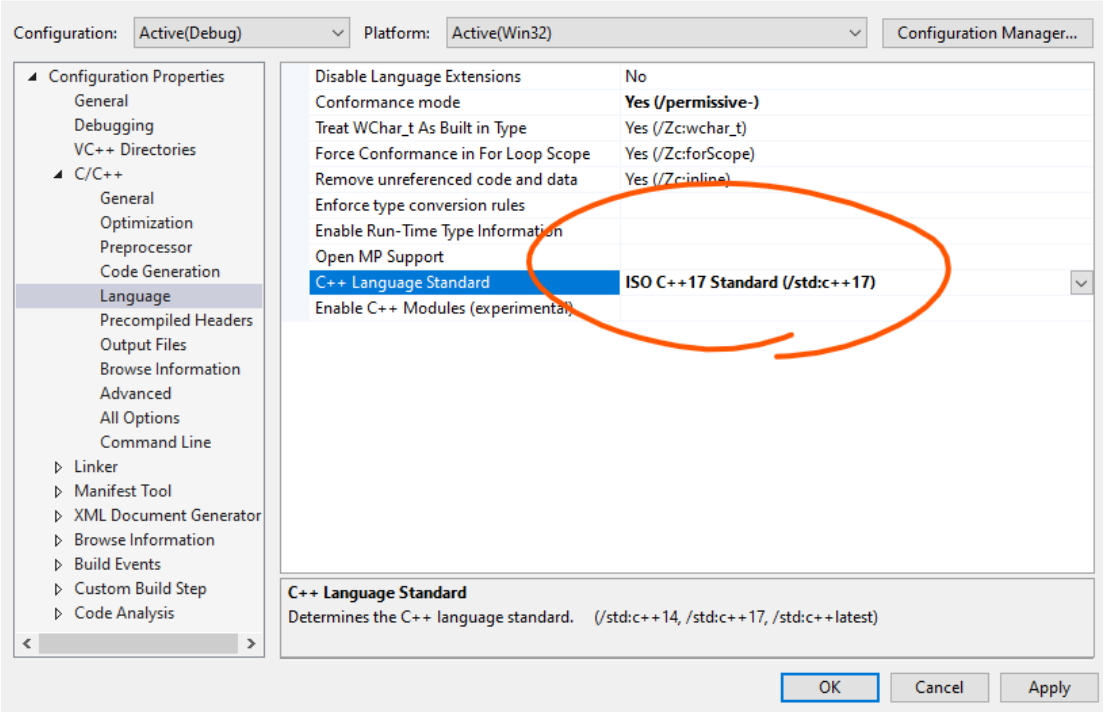
Ha az std::optional<T> objektum értékére tesztelünk, pontosan akkor kapunk vissza igaz értéket, ha az objektum tartalmaz értéket

```
#include<optional>

class A {

  int x;

public:

  A(int a) : x(a) {}

  void print() {

    std::cout << "my value is " << x << std::endl; }

};
```

```
int main() {

  A a1{ 5 };

  std::optional<A> perhapsa1;

  std::optional<A> definitelya1{ a1 };

  if (!perhapsa1) { std::cout << "perhapsa1 has no value" << std::endl; }

  if (definitelya1) { // definitelya1.has_value() is jo

    definitelya1.value().print();

    (*definitelya1).print(); // ez a 2 is mukodik...

    definitelya1->print();

  }
```

std::optional

Ahhoz, hogy mindez Visual Studio-ban működjön, a projekt properties oldalán biztosítsuk, hogy a fordító a C++17-es sztenderd szerinti:



std::variant

A <variant> headerben található std::variant típusú változó létrehozásakor template paraméterként felsorolásszerűen adhatjuk meg, hogy az adott változó milyen típusú értékeket tartalmazhat.

A **union** adatszerkezet már a C nyelvben is létezett. Ez egy olyan adatszerkezet, amely több különböző típusú adattagból mindig csak az egyiket tartalmazza. Az adatszerkezet mérete a legnagyobb adattag mérete, így bármelyik típus belefér.

Az std::variant a unionnak egy korszerűbb változata, mivel nyilván is tartja, hogy épp milyen típusú értéket tárol (míg union esetében ezt a programozónak kell megtennie). Ha egy std::variant-ból nem a megfelelő típusú értéket próbáljuk kiolvasni, kivétel dobódik vagy nullptr-t kapunk; míg union esetében nem definiált viselkedést kapunk.

Az std::variant típus megadása így néz ki:

```
std::variant<A, int, double, std::string> var;
```

std::variant

Hasonlít az öröklődés nyújtotta polimorfizmusra, ahol a hivatkozott típus helyett annak altípusai is behelyettesíthetők. Itt viszont az objektumban tárolható adattípusok lehetnek egymástól függetlenek.

- Az std::variant objektum nem tartalmazhat referenciákat (az std::optional sem)
- A default konstruktora az első alternatív típus default konstruktorát hívja (ha nincs, a variant-nek se lesz)
- A tárolt értéket az std::get<Type> vagy std::get<typeIndex> függvénnyel kérhetjük le
  - Ha nem olyan típust kérünk, mint ami tárolva van, std::bad_variant_access kivételt dob
  - Nincs implicit konverzió, hiába lehetne konvertálni a tárolt típust a kért típusra
  - Az std::get_if hasonlóan működik, csak mutatót ad vissza az adatra, vagy rossz típus esetén nullptr-t
- A tárolt típus indexét az index() tagfüggvénnyel lehet lekérdezni
- Ritka esetben (mert pl. értékadás közben kivétel jön létre) az std::variant hibás állapotba kerül, amit ellenőrizhetünk a valueless_by_exception() tagfüggvénnyel

std::variant

```
class A {

  int x;

public:

  A(int a) : x(a) {}

  void print() { std::cout << "my value is " << x
<< std::endl; }

};

int main() {

  std::variant<double, int, A, std::string> var; //
A nem lehet elol mert nincs default konstruktora!

  double x = std::get<0>(var);

  double x2 = std::get<double>(var); // ez is mukodik

  std::cout << "initial value of variant is " << x <<
std::endl; // 0, implicit default inicializacioval

  try {

    A y = std::get<2>(var);

    y.print();

  } catch (std::bad_variant_access& e) {

    std::cout << e.what() << std::endl;

  }
```

std::variant

Általában különbözőképpen akarjuk kezelni a különböző típusú adatokat. Ahelyett, hogy mindig elágaznánk a variant-ben tárolt típus alapján, készíthetünk egy Visitor objektumot, ami a megfelelő típusokra definiálja az operator() metódust. Az std::visit() segédfüggvénynek átadva a Visitort és a variant változót, az meghívja a Visitor megfelelő típusú operator()-át.

```
struct Visitor {
  void operator()(int n) {
    std::cout << n << " is an integer\n";
  }
  void operator()(float f) {
    std::cout << f << " is a float\n";
  }
};
```

```
int main()
{
  std::variant<int,float> number = 42;
  Visitor v;
  std::visit(v, number);
  number = 3.14f;
  std::visit(v, number);
}
```

std::any

Az <any> headerben található std::any – ahogy a név is sugallja – bármilyen típust tud tárolni, vagy lehet üres is.

A tárolt típusnak kell, hogy legyen másoló konstruktora.

Az std::variant-tel ellentétben heap-en tárolja az adatot.

Amennyiben nem üres, az std::any_cast függvénnyel megpróbálhatjuk értékét egy konkrét típusra kasztolni.

- Amennyiben a konverzió nem lehetséges, std::bad_any_cast típusú kivételt dob.
- Ahogy az std::variant-nél, itt sincs implicit konverzió.

std::any

```
std::any x;
if (!x.has_value()) {
  std::cout << "x has no value yet\n";
} // !x nem jo
x = std::string("haha");
x = 55;
```

```
if (x.has_value()) {
  try {
    std::string s = std::any_cast<std::string>(x);
    std::cout << s << '\n';
  } catch (const std::bad_any_cast& e) {
    std::cout << e.what() << '\n';
  }
}
```

typedef helyett using alias

A típusoknak új nevet (aliast) lehet adni, így nem kell mindig kiírni a hosszú, sok template paraméteres típusnevet. Sokszor az auto is segít, de meg vannak a hátrányai is.

A typedef a C nyelvből lett átvéve:

- typedef std::unique_ptr<std::unordered_map<std::string, std::string>> **UPtrMapSS**;

C++11-től alias deklaráció a using kulcsszóval:

- using **UPtrMapSS** = std::unique_ptr<std::unordered_map<std::string, std::string>>;
- Érthetőbb szintaxis
- Csak a scope-ján belül van hatása

Lehet alias template-et is deklarálni vele:

- template<typename T>
  using MyAllocList = std::list<T, MyAlloc<T>>;
  MyAllocList<int> numbers;

constexpr

Ez a kissé zavaros nevű kulcsszó a C++11-es szabvánnyal jelent meg, azóta pedig átesett pár ráncfelvarráson

Olyan kifejezést jelöl, ami nem csak const, de értéke fordítási időben ismert

- Ez lehetővé tesz néhány fordítói optimalizálást
- Kiváltja a nem típusos #define makrókat és bonyolult makrófüggvényeket
- Bizonyos helyeken csak fordítási időben ismert értékek adhatók meg, pl. tömb vagy std::array mérete

```
int n = 42;
//constexpr int bad = n; // hiba: n értéke fordításkor nem ismert
//constexpr int unknown; // hiba: kötelező értéket megadni
constexpr int size = 10;
//size = 3; // hiba: nem változhat a constexpr objektum értéke
std::array<int, size> arr;
//std::array<int, n> badarr; // hiba: a tömbméret fordításkor nem ismert
```

constexpr függvények

A makrófüggvényeket felváltó constexpr fv. olyan fv., ami fordítási időben is kiértékelhető, ha a paraméterek már ismertek (constexpr objektumok vagy literálok)

- Ha normál változót adunk át, akkor normál függvényként működik
- Ha fordítási időben hívódik, akkor constexpr értéket ad vissza

C++11-ben csak nagyon egyszerű fv. lehetett constexpr, ami csak egy return utasításból állt

- Az újabb szabványok lazítottak ezen, már lokális változó is létrehozható a függvényben
- Nem hívhat non-constexpr függvényt
- Ha fordítási időben kerül kiértékelésre, akkor nem férhet hozzá runtime (pl. globális) változókhoz

```
constexpr int square(int n) {
    return n * n;
}
std::array<int, square(4)> arr;
```

consteval, std::is_constant_evaluated

Mivel a constexpr függvényben nem lehetett tudni, hogy éppen fordítási időben történik-e a kiértékelés, a C++20 szabvány bevezetett néhány segédeszközt

A **consteval** függvényt (más néven immediate function) kötelező fordítási időben kiértékelni

- A constexpr helyett a consteval kulcsszót kell megadni

constexpr függvényben az **std::is_constant_evaluated** pontosan akkor lesz igaz, amikor fordítási időben történik a fv. kiértékelése, így ez alapján másképp viselkedhet a fv. fordítási és futási időben

- Attól, hogy a hívás helyén fordítási időben ismertek a paraméterei, még nem biztos, hogy fordításkor kerül kiértékelésre, de ha az eredményt constexpr objektumba mentjük, akkor igen (itt egy részletes magyarázat)

A standard library fejlesztése olyan irányba halad, hogy mindennek legyen constexpr változata, így akár std::vector-t is lehessen fordítási időben használni

- Érdekes videó: C++Now 2017 előadás egy constexpr JSON parser készítéséről (constexpr vector, string, map)

std::is_constant_evaluated példa

```
int sum = 1;

constexpr int update(int x) {
    return sum += x; // cannot be evaluated at compile-time! error in g++, warning in clang++
}

constexpr int compute(int n) {
    if (std::is_constant_evaluated())
        return n <= 1 ? 1 : n * compute(n - 1);
    else
        return sum += n;
}

int main() {
    int n = update(2); // no problem, it's a run-time call
    //constexpr int c = update(5); // error: attempt to access run-time storage
    constexpr int a = compute(5); // = 5! = 120
    int b = compute(4); // = sum + 4 = 3 + 4 = 7
}
```

LiteralType

Definiálhatunk saját típust, ami fordítási időben is példányosítható, akár egy literál (innen a név)

Kell, hogy legyen constexpr konstruktora és destruktora

Fordítási időben csak a constexpr tagfüggvények lesznek hívhatók

```
constexpr int absol(int x) {  // abs(int) is not constexpr :(
    return x < 0 ? -x : x;
}
struct Coord {
    int x, y;
    constexpr Coord(int x = 0, int y = 0) : x(x) , y(y) { }
    constexpr int manhattanDist(Coord c) const {
        return absol(x - c.x) + absol(y - c.y);
    }
};
int main() {
    constexpr Coord start, finish(7,7);
    std::array<Coord, start.manhattanDist(finish)> steps;
}
```