

C++: A TAVALYI GYŐZTES KATEGÓRIÁJA

5. forduló

RENDELKEZÉSRE ÁLLÓ IDŐ:

10:00

Ismertető a feladathoz

Ha a feladatok szövege máshogy nem rendelkezik, a kérdések a C++20 szabványra vonatkoznak.

Felhasznált idő: 02:01/10:00

Elért pontszám: 0/31

1. feladat 0/5 pont

Mi a sizeof("abcde") kifejezés értéke?

Válasz

- ☐ 4
- ☐ 5
- ☒ 6
- ☐ 8
- ☐ Fordítási hiba

Magyarázat

"The type of an unprefix string literal is `const char[N]`, where N is the size of the string in code units of the execution narrow encoding, including the null terminator."

https://en.cppreference.com/w/cpp/language/string_literal

"abcde" típusa tehát `const char[6]`.

2. feladat 0/5 pont

Mi történik, ha az alábbi módon deklarált függvény mégis exceptiont dob?

```
void doSomething() noexcept;
```

Válasz

- ☐ Nem történhet ilyesmi, a fordító nem enged olyan kódot lefordulni, ami ehhez vezethetne.
- ☐ Meghívódik az `std::terminate` függvény, ami a program azonnali leállításához vezet.
- ☐ Meghívódik az `std::unexpected` függvény, ami alapértelmezésben az `std::terminate`-en keresztül a program azonnali leállításához vezet, de ez konfigurálható (`std::unexpected_handler`).
- ☐ Undefined behaviour

Magyarázat

A `noexcept`-tel deklarált függvények igenis dobhatnak exceptiont, ilyenkor az `std::terminate` hívódik meg. Az `std::unexpected_handler` a régi *dynamic exception specification*-re volt igaz (`throw()` szintaxis, ami már C++20-ben nem létezik).

Akkor mi értelme van a `noexcept`-nek? A válasz röviden az, hogy az `std::vector` átméretezésekor problémákat okozhat az, ha a `move constructor` exceptiont dobhat, mert ilyenkor az `std::vector::resize` nem tudná egyszerűen rollbackelni a műveletet. ld. https://en.cppreference.com/w/cpp/utility/move_if_noexcept. Ezért volt szükség a `noexcept` bevezetésére.

De miért nem ellenőrzi a fordító, hogy egy `noexcept` függvény ténylegesen dob-e, vagy sem? Lényegében azért, hogy lehessenek olyan `move constructor`ok, amelyek *valid* objektumparaméter esetén nem dobnak, egyébként elméletben dobhatnak (pl. legacy third party kódba hívunk bele), de nem szeretnénk ezért feláldozni az `std::vector` által `noexcept` esetben biztosított optimalizációt.

3. feladat 0/7 pont

Miért nem fordul le az alábbi kód? (Több helyes válasz is van, mindegyiket be kell jelölni!)

```
#include <iostream>

class A {
    virtual void print() const = 0;
};

void A::print() const {
    std::cout << 12 << std::endl;
}

int main() {
```

```
A a;  
a.print();  
return 0;  
}
```

Válaszok

- ☐ Pure virtual tagfüggvénynek nem lehet definíciója.
- ☒ **A** absztrakt osztály, így nem példányosítható.
- ☐ **A**-nak nem definiáltunk konstruktort.
- ☐ **A** destruktora nem virtuális.
- ☒ **A::print** nem publikus.

Magyarázat

A gyakorlatban ritka, de pure virtual tagfüggvénynek is lehet definíciója. Ilyenkor a függvényt statikus módon (pl. `a.A::print()`) meghívhatjuk, de dinamikus módon (`a.print()`) nem érhető el, mert a ténylegesen megkonstruált leszármazott objektumban muszáj lenne felüldefiniálnunk.

Nem definiáltunk konstruktort, de a fordító automatikusan generált egyet, amit a leszármazott objektumok konstruktora tudna meghívni.

https://en.cppreference.com/w/cpp/language/abstract_class

4. feladat 0/7 pont

Mi az a, b, c ill. d változók típusa?

```
int main() {  
    int x(1);  
    auto a = x;  
    auto& b = x;  
    auto&& c = x;  
    auto d{x};  
    return 0;  
}
```

Válasz

- ☐ int, int&, int&&, std::initializer_list<int>
- ☐ int, int&, int&, int
- ☐ int, int&, int&&, int

- ☐ `int, int&, int&, std::initializer_list<int>`
- ☐ `int, const int&, const int&&, const int`

Magyarázat

a és b típusa nyilvánvalóan `int` ill. `int&`.

Az `"auto&&"` neve: *forwarding* avagy *univerzális referencia*. Ezt a fogalmat T&& típusú template paraméterekre szoktuk használni, de az `auto` (*majdnem*) ugyanazt a dedukciós algoritmust használja, mint a template paraméterek. Az `"auto&&"` módon definiált változó típusa lvalue vagy rvalue referencia, attól függően, hogy a jobb oldal lvalue vagy sem. Tehát c típusa `int&` lesz.

d esetében a dedukált típus definíció szerint `int` lesz (*direct-list-initialization*). Akkor lenne `std::initializer_list<int>`, ha lenne egy egyenlőségjel a d és a { között (*copy-list-initialization*).

Ld. még:

https://en.cppreference.com/w/cpp/language/reference#Forwarding_references

https://en.cppreference.com/w/cpp/language/template_argument_deduction#Other_contexts

5. feladat 0/7 pont

Mi kerülhet az alábbi kódrészletben a (*) helyére?

```
auto [a, b] = (*){1, 2};
```

Válaszok

- ☒ `std::pair<int, int>`
- ☒ `std::tuple<int, int>`
- ☐ `std::vector<int>`
- ☒ `std::array<int, 2>`
- ☐ Egyik sem

Magyarázat

Nagy vonalakban az mondható el a kódban szereplő *strukturált kötésről*, hogy akkor működik, ha a jobb oldalon szereplő kifejezés típusa egy olyan adatstruktúra, mely fordítási időben ismert számú elemet tartalmaz. Ilyen pl. a tömb, az `std::array`, az `std::pair`, és az `std::tuple`. Az `std::vector` nem ilyen.

https://en.cppreference.com/w/cpp/language/structured_binding

[Legfontosabb tudnivalók](#)

[Kapcsolat](#)

[Versenyszabályzat](#)

[Adatvédelem](#)

© 2022 Human Priority Kft.

KÉSZÍTETTE

Megjelenés

 Világos 