C++: A TAVALYI GYŐZTES KATEGÓRIÁJA

6. 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: 02:00/10:00 **Elért pontszám:** 0/29

1. feladat 0/5 pont

Mit ír ki az alábbi program?

```
#include <iostream>
#include <string>

struct StringReferenceWrapper {
  std::string& string;
};

int main() {
  std::string s1("alma");
  std::string s2("korte");
  StringReferenceWrapper ref1{s1};
  StringReferenceWrapper ref2{s2};
  ref1 = ref2;
  std::cout << s1 << ", " << s2 << std::endl;
  return 0;
}
```

Válasz

- ◯ A program nem fordul le

- ◯ alma, alma
- ◯ alma, korte
- ◯ korte, alma
- ◯ korte, korte
- ◯ Undefined behaviour

Magyarázat

Ha egy objektum referencia típusú mezőt tartalmaz, alapértelmezésben az operator= nincs definiálva:

"A defaulted copy assignment operator for class T is defined as *deleted* if any of the following is true: [...] T has a non-static data member of a reference type"

(https://en.cppreference.com/w/cpp/language/copy_assignment)

Ennek az az oka, amire egyébként ez a feladat is igyekszik rávilágítani, hogy nem igazán lenne egy ilyen operátornak értelme. Mert ugye mit kellene az operátornak csinálnia? Mutasson innentől a referencia mező a másik stringre? Ez nem jó, mert a referenciák fixek. Vagy másolódjon át a referencia által mutatott objektum tartalma? Ez is sántít, hiszen gyakori, hogy az objektumok egyfajta szülőobjektumra mutató referenciát tartalmaznak, és a gyerekobjektum másolásakor nem szeretnénk a szülőt módosítani.

Ha támogatni akarjuk a copy szemantikát (és a mutatott objektum helyett a mutatót akarjuk másolni), akkor az std::reference_wrapper-t érdemes használnunk: https://en.cppreference.com/w/cpp/utility/functional/reference_wrapper. A pointer is egy megoldás, de nem ideális, mivel pointernél a null is egy valid érték, referenciánál meg nem.

2. feladat 0/5 pont

Hány példánya él a Link osztálynak a (*)-gal jelölt ponton?

```
#include <memory>

struct Link {
  std::shared_ptr<Link> next;
};

void makeChain() {
  const std::shared_ptr<Link> first = std::make_shared<Link>();
  const std::shared_ptr<Link> last = std::make_shared<Link>();
  first->next = last;
  last->next = first;
}

int main() {
  makeChain();
```

```
    // (*)
    return 0;
}
```

Válasz

- ○ 0
- ○ 1
- ◉ 2
- ○ 4
- ○ Implementációtól függ

Magyarázat

Az std::shared_ptr egy referenciaszámlálós smart pointer. A fenti kódban a körbehivatkozás miatt mindkét példányra van élő referencia, ezért nem semmisülnek meg.

3. feladat 0/6 pont

Mely állítások igazak?

Válaszok

- ☑ Ha végigiterálunk egy std::set<int>-en, növekvő sorrendben fogjuk megkapni az elemeket.
- ☐ Az std::vector::resize() növelni és csökkenteni is tudja a vektor által ténylegesen lefoglalt memória méretét.
- ☑ Egy std::string a belsejében is tartalmazhat '\0' karaktereket.
- ☑ Az std::span<char> támogatja az elemek módosítását a [] operátoron keresztül.
- ☐ Az std::string_view támogatja az elemek módosítását a [] operátoron keresztül.

Magyarázat

Ezek közül talán csak az std::vectoros szorul magyarázatra. A resize sosem szabadít fel memóriát, ennek röviden az az oka, hogy bár nyernénk vele egy kis memóriát, de sebességet veszítenénk (át kellene mozgatni az elemeket), ill. az iterátorok is invalidálódnának. Nem világos, hogy mindez megéri-e vagy sem.

4. feladat 0/6 pont

Melyik C++ idiómára látunk példát az alábbi (sematikus) kódrészletben?

```
// include third-party libraries
// ...

template <class Handle>
class HandleManager {
public:
  HandleManager() = default;
  explicit HandleManager(Handle handle)
    : handle_{handle} {}
  ~HandleManager() {
    DeleteObject(handle_);
  }

  HandleManager(const HandleManager &) = delete;
  HandleManager(HandleManager &&other) noexcept
    : handle_{other.handle_} {
    other.handle_ = Handle{};
  }

  HandleManager& operator=(const HandleManager &) = delete;
  HandleManager& operator=(HandleManager &&other) noexcept {
    using std::swap;
    swap(handle_, other.handle_);
    return *this;
  }

  [[nodiscard]] Handle handle() const {
    return handle_;
  }

private:
  Handle handle_{};
};

int main() {
  const HandleManager handleManager{CreateBrush(...)};
  // use the brush
  // ...
  return 0;
}
```

Válaszok

- [x] RAII
- [x] Rule of Five

- [ ] Pointer to implementation
- [ ] CRTP
- [ ] Schwarz counter

Magyarázat

RAII: akkor semmisül meg az objektum által , amikor kimegy a scope-ból

Rule of Five: ha a destruktor, copy constructor, copy assignment operator, move constructor, move assignment operator közül egyet definiálunk, akkor definiáljuk (vagy explicite töröljük) az összeset. Ugyanis ebben az esetben gyanús, hogy az objektum egy erőforrást kezel, és ilyen esetben az előre definiált verziók nem működnek.

5. feladat 0/7 pont

Melyek a helyes main() definíciók a C++20 szabvány szerint?

Válaszok

- [ ] void main(void) {}
- [x] int main() { return 0; }
- [x] int main() {}
- [ ] auto main() { return 1; }
- [x] int main(int kutya, char* cica[]) { return 2; }
- [ ] int main(int argc, char** argv) = delete;

Magyarázat

C++-ban muszáj, hogy a visszatérési érték típusa int legyen (auto sem lehet). Így pl. a "void main(void)" C-ben helyes, de C++-ban nem. A return utasítás a main-ből hiányozhat, ilyenkor automatikusan 0 lesz a visszatérési érték. Az esetleges argc és argv argumentumok neve tetszőleges lehet. A main() esetében a delete nem megengedett.