OOP vizsgakérdések kidolgozás

Vári Gergő

2025. november 17.

Tartalomjegyzék

1 Ismertesse a C++ nyelvben alkalmazott bővítéseket az egyszerű adattípusok terén, valamint a konzol ki/bemenet megvalósításait!

1.1 C++ bővítések az egyszerű adattípusok terén

A C++ a C nyelv típusrendszerére épül, de szigorúbb típusellenőrzést vezet be, és számos új elemmel bővíti azt a biztonságosabb és kényelmesebb programozás érdekében.

- **Logikai típus (`bool`):**
  - A C-vel ellentétben (ahol az egész számok 0/nem-0 értéke jelentette a logikát) a C++ bevezeti az önálló `bool` típust.
  - Lehetséges értékei: `true` (igaz) és `false` (hamis).
  - Memóriaigénye: implementációfüggő, de általában 1 bájt.
- **Referencia típus (`&`):**
  - Ez egy már létező változó "álneve" (alias).
  - **Jellemzői:** Deklaráláskor azonnal inicializálni kell, és később nem állítható át más változóra.
  - **Előnye:** Lehetővé teszi a cím szerinti paraméterátadást a mutatók (pointerek) bonyolult szintaxisa nélkül (pl. `void fv(int &x)`).
- **Inicializálási módok bővülése:**
  - **Konstruktor szintaxis:** `int a(5);` – úgy néz ki, mint egy függvényhívás.
  - **Egységes inicializálás (C++11):** `int a{5};` – ez a legbiztonságosabb forma (megakadályozza a szűkítő konverziót).
- **Típuslevezetés (`auto`):**
  - A fordító a kezdőérték alapján automatikusan határozza meg a változó típusát.
  - Példa: `auto x = 5.5;` (a típus `double` lesz).
  - Kötelező inicializálni a használatakor.
- **`nullptr`:**
  - A típusbiztonság érdekében bevezették a `nullptr` kulcsszót a NULL makró (ami sima 0) helyett, így elkerülhetők a pointerek és egészek keveredéséből adódó hibák.

1.2 Konzol Ki- és Bemenet (I/O) megvalósítása

A C++ az objektumorientált `iostream` könyvtárat használja a `cstdio` (printf/scanf) helyett/mellett. Az I/O műveletek adatfolyamokként (streamek) valósulnak meg.

- **Könyvtár és Névtér:**
  - Header fájl: `#include <iostream>`
  - Minden elem az `std` névtérben található (pl. `std::cout`).
- **Alapvető objektumok (Streamek):**
  - `cin`: Szabványos bemenet (billentyűzet).
  - `cout`: Szabványos kimenet (képernyő/konzol).
  - `cerr`: Szabványos hibakimenet (pufferelés nélküli).
  - `clog`: Szabványos naplózó kimenet (pufferelt).
- **Operátorok:**
  - **Beszúró operátor (<<):** Adatot küld a kimeneti folyamra. Láncolható.
    Példa: `cout << "Ertek:  " << x;`
  - **Kiemelő operátor (>>):** Adatot olvas a bemeneti folyamról egy változóba. Automatikusan kezeli a típusokat és figyelmen kívül hagyja a szóközöket (whitespace).
    Példa: `cin >> x;`
- **Manipulátorok (Formázás):**
  - A kimenet formázását végző speciális elemek.
  - `std::endl`: Új sor beszúrása és a puffer ürítése (flush).
  - További formázások az `<iomanip>` headerben: `setw()` (mezőszélesség), `setprecision()` (tizedesjegyek).
- **Előnyök a C (printf) megoldással szemben:**
  - **Típusbiztonság:** Nem kellenek formátumkódok (pl. `%d`), a fordító felismeri a típusokat.
  - **Bővíthetőség:** Saját osztályokra/típusokra is túlterhelhetők a `<<` és `>>` operátorok.

2 Ismertesse a csak C++ nyelvben alkalmazható típuskonverziót, valamint a header fájlok használatánál alkalmazható egyszerűsítést!

2.1 C++ specifikus típuskonverziók (Casting)

A C-stílusú `(tipus)ertek` konverzió helyett a C++ négy speciális operátort vezetett be. Ezek célja a típusbiztonság növelése és a szándék egyértelmű jelzése a kódban (könnyebb kereshetőség).

- `static_cast<T>(kif)` – A "logikus" konverzió
  - **Felhasználás:** Kompatibilis típusok között (pl. `int` → `float`, `enum` → `int`, pointer upcast).
  - **Ellenőrzés:** Fordítási időben (Compile-time).
  - **Biztonság:** Ha a típusok logikailag nem konvertálhatók, a fordító hibát jelez.
- `dynamic_cast<T>(kif)` – Az "objektumorientált" konverzió
  - **Felhasználás:** Polimorf osztályoknál (van `virtual` függvénye) öröklődési fában lefelé (downcast).
  - **Ellenőrzés:** Futási időben (Runtime check).
  - **Biztonság:** Ellenőrzi, hogy az objektum ténylegesen az-e, aminek hisszük.
    - Pointer esetén: sikertelenségnél `nullptr`-t ad.
    - Referencia esetén: sikertelenségnél `std::bad_cast` kivételt dob.
- `const_cast<T>(kif)` – A "szabálytalanító"
  - **Felhasználás:** A `const` vagy `volatile` minősítő levétele.
  - **Cél:** Főleg régebbi (legacy) kódok illesztéséhez, ahol egy függvény nem módosít, de lemaradt a paraméteréből a `const`.
- `reinterpret_cast<T>(kif)` – A "brutális" átértelmezés
  - **Felhasználás:** Alacsony szintű bit-manipuláció (pl. pointer konvertálása `int`-té, vagy két teljesen független pointer típus között).
  - **Jellemző:** Nincs ellenőrzés, a biteket változatlanul hagyja, csak másképp értelmezi. Veszélyes és nem hordozható (platformfüggő).

2.2 Header fájlok használatának egyszerűsítése

A nagy projektek fordítási idejének csökkentése és a körkörös függőségek (circular dependency) elkerüléséhez.

- **Include Guard modernizálása (`#pragma once`)**
  - **Probléma:** Egy header fájl többszörös beillesztése fordítási hibát okoz.
  - **Hagyományos megoldás:** `#ifndef LABEL`, `#define LABEL`, `#endif`.
  - **Egyszerűsítés:** A fájl legelső sorába írt `#pragma once`.
  - **Előny:** Nem kell egyedi makróneveket kitalálni, rövidebb, tisztább a kód, a fordító optimalizálhatja.
- **Elődeklaráció (Forward Declaration)**
  - **Módszer:** A header fájlban nem `#include "Osztaly.h"`-t használunk, hanem csak kiírjuk: `class Osztaly;`.
  - **Feltétele:** Csak akkor működik, ha az adott headerben csak pointert (`Osztaly*`) vagy referenciát (`Osztaly&`) használunk az adott típusból (nem példányosítjuk, nem érjük el a mezőit).
  - **Előny:**
    - Drasztikusan csökken a fordítási idő (kevesebb fájlt kell feldolgozni).
    - Megszünteti a körkörös hivatkozásokat (A include B, B include A).

3 Ismertesse a referencia típusú változók deklarációját, használatát, és a referencia adattagot tartalmazó osztályok konstruktorának megvalósítását! Írjon példát a referencia típus használatára függvényben!

3.1 Referencia változók (Alias)

A referencia egy már létező objektum vagy változó **másodlagos neve** (álneve). A C++ nyelvben a pointerek biztonságosabb és kényelmesebb alternatívájaként szolgál számos esetben.

- **Deklaráció:** A típus után tett `&` jellel.
- **Kötelező inicializálás:** Deklaráláskor azonnal hozzá kell kötni egy létező változóhoz.
- **Rögzítettség:** Az inicializálás után **nem irányítható át** más változóra.
- **Nincs NULL érték:** A referenciának mindig érvényes memóriaterületre kell mutatnia.

```
int x = 10;
int& ref = x;  // 'ref' mostantól 'x' álneve

ref = 20;      // Ez valójában 'x'-et módosítja 20-ra
// int& hibas; // HIBA: nincs inicializálva!
```

3.2 Referencia adattagok és a konstruktor

Ha egy osztály referencia típusú adattagot tartalmaz, speciális szabályok vonatkoznak a konstruktorra, mivel a referenciát létrehozáskor inicializálni kell.

- **Probléma:** A konstruktor törzsének futásakor az adattagok már létrejöttek (memória lefoglalva), így ott már késő lenne értéket adni a referenciának (az már értékadás lenne, nem inicializálás).
- **Megoldás:** Kötelező a **taginicializáló lista** (Member Initializer List) használata.

```
class AdatTarolo {
    int& referenciaAdat; // Referencia adattag

public:
    // Konstruktor
    AdatTarolo(int& kulsoValtozo)
        : referenciaAdat(kulsoValtozo) // Taginicializáló lista
    {
        // A törzsben (itt) már hiba lenne értéket adni neki először.
    }
};
```

3.3 Referencia használata függvényekben

A referenciák leggyakoribb felhasználása a függvényparamétereknél történik (*Pass-by-Reference*).

- **Hatékonyság:** Nem készül másolat az objektumról (nagy adatstruktúráknál kritikus).
- **Módosíthatóság:** A függvény képes megváltoztatni a hívó fél eredeti változóját (kimenő paraméter).
- **Konstans referencia:** Ha a másolást el akarjuk kerülni, de a módosítást tiltani szeretnénk: `const Type&`.

```
// Két szám cseréje (referenciák nélkül nem működne másolás miatt)
void csere(int& a, int& b) {
    int temp = a;
    a = b; // Az eredeti változót írja felül
    b = temp;
}

int main() {
    int x = 5, y = 10;
    csere(x, y);
    // Itt: x = 10, y = 5
    return 0;
}
```