C++
Gyakorlat jegyzet
12. óra.

A jegyzetet Umann Kristóf készítette Horváth Gábor gyakorlatán. (2017. január 5.)

1. Beugró kérdések

A következő kérdések egy régebbi, kiszivárgott C++ beugróból lettek beemelve. Így értelemszerűen ezekkel a kérdésekkel az olvasó nem fog semmiképpen se találkozni. Alaposabb magyarázatot nem fűztem a megoldásokhoz, ugyanis azok a jegyzet alapos megértése után triviálisak.

1.) Hány byte-on tárol a C++ egy short int-et?

a) 1
b) implementációfüggő
c) 8
d) 2

Válasz: Implementációfüggő.

2.) Melyik reláció **hamis** az alábbiak közül?

a) `sizeof(char) == sizeof(signed char)`
b) `sizeof(short) <= sizeof(long int)`
c) `sizeof(bool) == sizeof(char)`
d) `sizeof(float) <= sizeof(long double)`

Válasz: `bool == char`

3.) Mennyi a 012 konstans értéke?

a) 12
b) 0.12
c) 18
d) 10

Válasz: 10

4.) Melyik nem preprocesszor direktíva?

a) `#define`
b) `#else`
c) `#elif`
d) `#elseif`

Válasz: `#elseif`

5.) Melyik kulcsszó **nem** a tárolási osztályt specifikálja egy deklarációban ill. definícióban?

a) `static`
b) `auto`
c) `register`
d) `public`

Válasz: `public`, ugyanis a register azt jelenti, hogy a változót a `register`be tárolja, azonban azt hogy milyen változót hol érdemes tárolni, jobban tudja a fordító, így nem érdemes kiírni. Az `auto` azt jelenti hogy a változó stacken legyen, ez régebben implicit módon mindehol ott volt, de mostmár nem, sőt, c++11ben mást is jelent.

6.) Az `X::f()` függvényhívás során mit ír ki a program?

```
int i = 1;
namespace X
{
    int i = 2;
    void f()
    {
        int a = i;
        int i = a + X::i + ::i;
        std::cout << i << std::endl;
    }
}
```

a) 1

b) semmit, fordítási hiba keletkezik

c) 5

d) 4

Válasz: 5

7.) Melyik igaz az alábbiak közül?

```
struct X
{
    X(int i = 0) {}
};
```

a) A fenti struct-nak van default konstruktora.

b) A fenti struct-nak csak default konstruktora van.

c) A fenti struct-nak nincs default konstruktora.

d) A fenti struct-nak nincs copy konstruktora.

Válasz: Az első az igaz, továbbá sok egyéb dolgot is generál még.

8.) Az alábbi példában a `Foo f(10);` konstruktor hívása után mennyi lesz `f.x` értéke?

```
struct Foo
{
    int x, y;
    Foo(int i):y(i),x(y++) {}
};
```

a) 11

b) 0

c) nem definiált

d) 10

Válasz: Nem definiált.

9.) Melyik típusnak van push_front tagfüggvénye?

a) `std::set`

b) `std::list`

c) `std::stack`

d) `std::vector`

Válasz: `std::list`

10.) Mi lesz az `a` változó értéke a függvényhívás után?

```
int a = 1, b = 2;
void f(int x, int y)
{
    int t = x;
    x = y;
    y = t;
}
f(a,b);
```

a) 1

b) 2

c) nem definiált

d) semmi, fordítási hiba keletkezik

Válasz: 1

11.) Melyik állítás igaz egy konstans objektum esetében?

a) Az objektumnak csak private adattagja lehet.

b) Az objektumnak csak azok a tagfüggvényei hívhatóak meg, amelyek nem módosítják az adattagjait.

c) Az objektumnak csak a konstans tagfüggvényei hívhatóak meg.

d) Az objektum csak default konstruktorral hozható létre.

Válasz: Az objektumnak csak a konstans tagfüggvényei hívhatók meg.

12.) Melyik állítás igaz az alábbiak közül?

a) A `dynamic_cast` soha nem dob kivételt.

b) A `dynamic_cast` használatához nem lehet statikus adattagja az osztálynak.

c) A `dynamic_cast` használatához polimorf osztályokra van szükség.

d) A `dynamic_cast` fordítás idejű típuskonverziót végez.

Válasz: Polimorf osztályra van szükség a használatához.

13.) Mi a paraméterdedukció?

a) Az az eljárás, amikor referencia-szerinti paraméterátadásra cseréljük az érték-szerintit.

b) Az az eljárás, amikor a fordítóprogram levezeti a template paramétereket a függvényhívásból.

c) Az az eljárás, amikor linker feloldja a külső függvényhívások paramétereit.

d) Az az eljárás, amikor default paraméterekkel látjuk el a függvény paramétereket.

Válasz: Az az eljárás, amikor a fordítóprogram levezeti a template paramétereket a függvényhívásból.

14.) Az alábbiak közül melyik függvényhívással lehet ekvivalens az alábbi (csillaggal jelölt) operátorhívás?

```
class Matrix
{
    // ...
};
Matrix a,b;
a + b; // (*)
```

a) `a.operator+(a,b);`

b) `b.operator+(a);`

c) `operator+(a,b);`

d) `Matrix.operator+(a,b);`

Válasz: A harmadik a jó, mert lehet egy függvényen kívüli függvényhívás a csillagozott rész.

15.) Melyik állítás igaz az alábbiak közül?

a) Egy `int* const` típusú pointer mérete 8 byte.

b) A `sizeof(int) == sizeof(int* const)` reláció mindig igaz.

c) Egy `int* const` típusú pointer nem változtathatja meg a mutatott értéket.

d) Egy `int* const` típusú pointer mutathat változóra.

Válasz: Utolsó.

2. Mintaviszga megoldás

2.1. Az írásbeli vizsga menete

A vizsga írásbeli részénél a feladat egy header fájl elkészítése, mellyel egy előre megadott cpp fájl helyesen fordul. Amennyiben a program szabványos, azaz lefordul és nem okoz nem definiált viselkedést, kiírja a vizsgázó osztályzatát.
A cpp fájl leellenőrzi hogy a header fájlban leírtak helyen oldják meg a feladatot, azonban lehetnek szélsőséges esetek, melyek nem vizsgál. Így a sikeres futás után **sem** biztos, hogy a header fájl a kiírt jegyet megéri.
A kiírt jegy emelett nem „íródik jóvá" azonnal: a vizsgázónak meg kell védenie a munkáját, azaz bizonyítania kell hogy a leírtakat, és a félév során tanultakat valóban érti. Amennyiben erre képtelen, úgy a jegye romlik, szélsőséget esetben elégtelenre is akár.
A header fájl csak is kizárólag a C++98-as szabványnak megfelelő kódot tartalmazhat, ellentkező esetben csak akkor, ha a vizsgázó később meg tudja mutatni, hogy 98-as szabvány szerint mi lenne a megoldás.

2.2. Feladatleírás

A feladat az, hogy írjunk egy osztályt, mely egy billentyűzetet szimulál. Megfigyelhető a lenti C++ kód alapján, hogy `line_editor` lesz a konténer neve, valamint hogy 2 template paraméterrel rendelkezik. Az felhasználónak képesnek kell lennie, arra, hogy a billentyűleütéseit kiirattassa majd a konzolra, sőt, akár arra is, hogy egy adott szövegnek ne csak végére, hanem elejére is írhasson.

lemain.cpp

```
#include <iostream>
#include "lineedit.h"
#include <string>
#include <algorithm>
#include <iterator>
#include "lineedit.h"
#include <list>
#include <vector>

const int max = 1000;

int main()
{
    int your_mark = 1;

    //2-es

    line_editor<std::vector<int>, int> lev;
    for( int i = 0; i < max; ++i )
    {
        lev.press( i );
        lev.home();
        lev.press( i );
    }
```

```
    std::vector<int> v = lev.enter();

    line_editor<std::list<double>, double> lel;
    lel.press( 4.8 );
    lel.home();
    lel.press( 1.1 );
    std::list<double> c = lel.enter();

    line_editor<std::string, char> les;
    les.press( 'W' );
    les.press( 'o' );
    les.press( 'r' );
    les.press( 'l' );
    les.press( 'd' );
    les.home();
    les.press( 'H' );
    les.press( 'e' );
    les.press( 'l' );
    les.press( 'l' );
    les.press( 'o' );

    std::string s = les.enter();
    if ( "HelloWorld" == s && "" == les.enter() && 2.2 > *(c.begin()) &&
    2 * max == v.size() && max - 1 == v[ 0 ] )
    {
        your_mark = c.size();
    }

    //3-as

    les.press( 'H' );
    les.press( 'e' );
    les.press( 'l' );
    les.press( 'l' );
    les.press( 'o' );
    les.home();
    les.insert();
    les.press( 'H' );
    les.press( 'a' );
    s = les.enter();


    for( int i = 0; i < max; ++i )
    {
        lev.press( 2 );
    }
    lev.home();
    lev.insert();
    for( int i = 0; i < max; ++i )
    {
        lev.press( 1 );
    }
    v = lev.enter();

    lel.press( 7.9 );
    lel.press( 1.2 );
    lel.home();
    lel.insert();
```

```
    lel.press( 1.5 );
    lel.insert();
    lel.press( 3.7 );
    c = lel.enter();

    if ( 1.7 > c.front() && 1.3 < c.front() && v[ max / 2 ] == v[ max / 5 ]
       &&
    1.4 > c.back() && 1U * max == v.size() && "Hallo" == s && 1 == v[ max /
       4 ] )
    {
        your_mark = c.size();
    }

    //4-es

    line_editor<std::vector<int> > llev;
    llev.press( 3.3 );
    llev.press( 1.1 );
    llev.home();
    llev.del();
    std::vector<int> lv = llev.enter();

    line_editor<std::string> lles;
    lles.press( 'J' );
    lles.press( 'a' );
    lles.press( 'v' );
    lles.press( 'a' );
    lles.backspace();
    lles.backspace();
    std::string f = lles.enter();

    if ( "Ja" == f && lv[ 0 ] < 1.3 )
    {
        your_mark += lv.size();
    }

    //5-os

    line_editor<std::list<int> > lle;
    lle.press( 3 );
    lle.home();
    lle.insert();

    llev.press( 8 );
    llev.press( 2 );
    llev.home();

    lle.swap( llev );
    lle.press( 1 );
    std::list<int> cl = lle.enter();
    v = llev.enter();
    if ( 1 == cl.front() && cl.size() > v.size() && 3 == v[ 0 ] )
    {
        your_mark += v.size();
    }


    std::cout << "Your mark is " << your_mark;
```

```
    std::endl( std::cout );
}
```

2.2.1. Megjegyzés. Én most kiszedtem a kommenteket, de minden blokk (tehát pl. a 2-es és 3as közötti) ki van kommentezve.

2.3. 1-es

Ahhoz hogy a program egyáltalán lefordúljon, és kiírja hogy `Your mark is 1`, létre kell hoznunk egy header fájlt. Fent látható, hogy ennek milyen nevűnek kell lennie: `lineedit.h`. Az is megfigyelhető, hogy ez a header kapásból kétszer van beillesztve, így (ahogy minden header fájlnál is szokás) írjunk egy header guardot.

```
#ifndef LINE_EDIT_H
#define LINE_EDIT_H

#endif
```

Így már van egy 1esünk.

2.4. 2-es

A ketteshez már létre kell hoznunk magát az osztályt, és ahogy korábban megállapítottuk, 2 template paraméter kell ehhez: az első egy konténer, amiben a felhasználó egy adott sor tartalmát fogja visszakapni, a második pedig a tárolandó típus.

```
template <typename Cont, typename CharT>
class line_editor
{

};
```

A 2-eshez az alábbi függvényeket kell majd megírnunk:

- `press`: A `CharT` típsú kapott paramétert eltárolja.
- `home`: Az adott sor elejére ugrik (nem vár paramétert).
- `enter`: Új sort kezd, és a meglevőt visszaadja egy `Cont` típusú objektumban eltárolva (nem vár paramétert).

Kéne egy konténert választani, melyben eltároljuk a karaktereket. Itt logikus választás az `std::list`, mert gyakran kell majd szúrnunk a sor közepébe.

2.4.1. Megjegyzés. Használhatnánk `std::vector`-t is, sőt bármit, melyet védéskor kellőképpen meg tudunk indokolni.

```
#include <list>

template <typename Cont, typename CharT>
class line_editor
{
    std::list<CharT> line;
};
```

A következő kérdés, hogyan reprezentáljuk a kurzort, mely jelzi hova szúrjon majd be a `press` függvény. Erre praktikus megoldás lehet egy iterátor használata.

```
template <typename Cont, typename CharT>
class line_editor
{
    std::list<CharT> line;
    typename std::list<CharT>::iterator cursor;
};
```

Felmerül az is, kell-e konstruktor, destruktor, copy konstruktor és értékadó operátor?
Ezt az elsőt nem ártana megírnunk: azonban `line`-t és `cursor`-t mivel inicializáljuk? Megállapítható, hogy az `std::list` default konstruktora számunkra megfelelő, így azzal külön foglalkoznunk nem kell. `cursor`-t állítsuk a sor végére.
Mivel mi dinamikusan nem foglaltunk le memóriát, és csak reguláris típusokat tárolunk, így a többi 3 függvényre nincs szükségünk.

```
template <typename Cont, typename CharT>
class line_editor
{
    std::list<CharT> line;
    typename std::list<CharT>::iterator cursor;
public:
    line_editor() : cursor(line.end()) {}

    void press(CharT c)
    {
        //...
    }
    void home()
    {
        //...
    }
    Cont enter()
    {
        //...
    }
};
```

2.4.2. Megjegyzés. Megfigyelhető, hogy `press` érték szerint veszi át a paraméterét – nem kötelező így, de bárhogyan is tesszük, meg kell indokolni. Ebben az esetben lehet rá számítani, hogy a felhasználó csak primitív típusokat akar majd tárolni, legalábbis nem túl komplikáltakat, így az érték szerinti átvétel itt várhatóan hatékonyabb lesz.

A fentiek közül a `home` függvény talán a legegyszerűbb: állítsuk `cursor`-t a sor elejére!

```
void home()
{
    cursor = line.begin();
}
```

Térjünk `press`-re: mivel az iterátor könnyen invalidálódhat ha új elemet szúrunk be, az `insert` az `std::list`-nél visszaad egy iterátort az új elemre, így érdemes a `cursor`-t erre beállítani, és egyel előrébb vinni, hogy a következő beszúrás is az új elem után következzen be.

```
void press(CharT c)
{
    cursor = line.insert(cursor, c);
    cursor++;
}
```

Az `enter`-nél vissza kell adnunk egy, a template paraméterként megadott konténer típususával megyező kontért, mert tartalmazza az eddig begépelt elemeket, majd törli a listát. Mivel minden STL konténer rendelkezik olyan konstruktorral, mely egy iterátor párt vár, és ez alapján a két iterátor közti elemeket be tudja szúrni, így ezt bátran írhatjuk:

```
Cont enter()
{
    Cont ret(line.begin(), line.end());
    line.clear();
    cursor = line.end();
    return ret;
}
```

Ezzel a kettesünk is készen van.

2.5. 3-as

Szükségünk lesz egy `insert` tegfüggvényre, mely ki-be kapcsolja az insert billentyűt (amennyiben ez nem ismerős, az insert meghívásának hatására nem a kurzor után kell majd beszúrni elemeket, hanem a kurzor utáni elemeket kell majd felülírni).
Ehhez szükségünk lesz egy logikai változóra is, melyet alapértelmezetten állítsunk hamisra, és módosítanunk kell majd `press`-t is.

```
template <typename Cont, typename CharT>
class line_editor
{
    //...
    bool isInsert;
public:
    line_editor() : cursor(line.end()), isInsert(false) {}

    void insert()
    {
        isInsert = !isInsert;
    }
    void press(CharT c)
    {
        if(isInsert && cursor != line.end())
            *cursor = c;
        else
            cursor = line.insert(cursor, c);
        cursor++;
    }
    //...
};
```

2.6. 4-es

Megfigyelhetjük, hogy a 4-eshez már a `line_edit` osztályt csak 1 template paraméterrel is tudnunk kell példányosítani. Ez azt jelenti, hogy a második template paraméternek kell egy alapértelmezett behelyettesítést biztonsítani, melynek meg kell egyeznie a `Cont` által tárolt típussal.

```
template <typename Cont, typename CharT = typename Cont::value_type>
class line_editor
{
    //...
};
```

2.6.1. Megjegyzés. A fenti kettős `typename` talán összezavaró lehet – a másodikra a dependent scope miatt van szükség.

2.6.2. Megjegyzés. Mivel internetet lehet használmi, nem kell megijedni, ha nem tudjuk ezt hogyan kéne egyből megcsinálni. Előfordulhat olyan, hogy olyan dologgal találkozunk vizsgán szembe, amit se előadáson, se gyakorlaton nem mondtak el: ez azért van ,mert ennek a tárgynak célja az, hogy ezekkel is meg tudjunk küzdeni.

Ezen kívül két új tagfüggvényt is meg kéne írnunk:

1. `backspace`: Az `cursor` mögötti elemet törli ki.
2. `del`: A `cursor` előtti elemet törli.

```
template <typename Cont, typename CharT = typename Cont::value_type>
class line_editor
```

```
{
    //...
public:
    void backspace()
    {
        cursor--;
        del();
    }
    void del()
    {
        cursor = line.erase(cursor);
    }
    //...
};
```

2.7. 5-ös

Meg kell írnunk egy swap függvényt, mely 2, különböző template paraméterrel rendelkező `line_editor`-t megcserél.

```
//...
#include <algorithm>

class line_editor
{
    //...
public:
    //...
    template <class T, class U>
    void swap(line_editor<T,U> &le)
    {
        std::swap(line, le.line);
        std::swap(cursor, le.cursor);
        std::swap(isInsert, le.isInsert);
    }
};
```

Azonban fordítási hibát kapunk, hisz egy különböző template pareméterekkel rendelkező `line_editor` külön típusnak számít, így gondoskodunk kell arról is, hogy minden `line_editor` barát legyen.

```
class line_editor
{
    //...
public:
    //...
    template <class T, class U>
    friend class line_editor;
};
```

Így az ötöst is elértük.