C++
Gyakorlat jegyzet
7. óra

A jegyzetet Umann Kristóf készítette Horváth Gábor gyakorlata alapján. (2018. április 30.)

0.1. Header fájlra és fordításra egységre szétbontás

Ha egy darab header fájlban tárolnánk mindent, számos problémába ütköznénk. Ha több fordítási egységbe illesztenénk be a headert, fordítási idejű hibát kapnánk, hogy számos függvényt többször próbáltunk definiálni (sértenénk az ODR-t). Erre megoldás lehet, hogy a definíciókat és deklarációkat elválasztjuk: az osztályban lévő függvények deklarációit hagyjuk meg a header fájlban, és a definíciókat egy külön fordítási egységbe tegyük!

0.1.1. Megjegyzés. Feltűnhet majd, hogy pár függvénydefiníció bent maradt. Erre később lesz magyarázat.

list.hpp:

```
#ifndef LIST_H
#define LIST_H

#include <iosfwd>

class List;

class Iterator
{
public:
    explicit Iterator(List *p) : p(p) {}
    bool operator==(Iterator other) const { return p == other.p; }
    bool operator!=(Iterator other) const { return !(*this == other); }
    Iterator operator++();
    int& operator*() const;
private:
    friend class ConstIterator;
    List *p;
};

class ConstIterator
{
public:
    ConstIterator(Iterator it) : p(it.p) {}
    explicit ConstIterator(const List *p) : p(p) {}
    bool operator==(ConstIterator other) const { return p == other.p; }
    bool operator!=(ConstIterator other) const { return !(*this == other); }
    ConstIterator operator++();
    int operator*() const;
private:
    const List *p;
};

class List
{
public:
    explicit List(int data_, List *next = 0) : data(data_), next(next) {}
    ~List() { delete next; }
    List(const List &other);
```

```
    List& operator=(const List &other);
    void add(int data);
    Iterator begin() { return Iterator(this); }
    ConstIterator begin() const { return ConstIterator(this); }
    Iterator end() { return Iterator(0); }
    ConstIterator end() const { return ConstIterator(0); }
private:
    friend Iterator;
    friend ConstIterator;
    int data;
    List *next;
};

#endif
```

list.cpp:

```
#include <iostream>

#include "list.hpp"
#include <iostream>

List::List(const List &other) : data(other.data), next(0)
{
    if (other.next != 0)
    {
        next = new List(*other.next);
    }
}

List& List::operator=(const List &other)
{
    if (this == &other)
        return *this;
    delete next;
    data = other.data;
    if (other.next)
    {
        next = new List(*other.next);
    }
    else
    {
        next = 0;
    }
    return *this;
}

void List::add(int data)
{
    if (next == 0)
    {
        next = new List(data);
    }
    else
    {
        next->add(data);
    }
}
```

```
Iterator Iterator::operator++()
{
    p = p->next;
    return *this;
}

int& Iterator::operator*() const
{
    return p->data;
}

ConstIterator ConstIterator::operator++()
{
    p = p->next;
    return *this;
}

int ConstIterator::operator*() const
{
    return p->data;
}
```

main.cpp:

```
#include <iostream>
#include "list.hpp"

void print(const List &l)
{
    for(ConstIterator it = l.begin(); it != begin(); ++it)
    {
        std::cout << *i << ' ';
    }
    std::cout << std::endl;
}

int main()
{
    List head(5);
    head.add(8);
    head.add(10);
    head.add(8);
}
```

Ez a szétválasztás sok egyéb előnnyel is jár: a `List`-hez tartozó információk sokkal kisebb helyen elférnek. Azonban ahogy a fenti megjegyzés is felhívta rá a figyelmet, a list.hpp továbbá is tartalmaz definíciókat! Ennek ellenére azt tapasztaljuk, hogyha több fordítási egységbe illesztjük be a headert, még akkor sem kapunk fordítási idejű hibát. Ennek a magyarázatához tegyünk egy kisebb kitérőt.

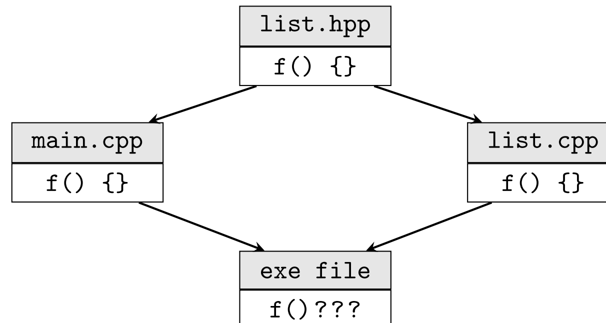
0.2. Inline függvények

Tekintsük azt a példát, amikor a `void f() {}` függvényt is beillesztjük a headerbe: ha több fordítási egységet fordítanánk egyszerre, melybe ez a header be van illesztve, linkelési hibát kapnánk, mert `f` többször lesz definiálva. Ez azonban megkerülhető az `inline` kulcsszó használatával, segítségével ugyanis kiküszöbölhető a linker hiba: minden azonos nevű, visszatérési értékű, és paraméter listájú inline-ként definiált függvény definícióval együtt beilleszthető több különböző fordítási egységbe, és nem fog fordítási hibát okozni.

Ez úgy oldható meg, hogy a fordító a linkelés folyamán a definíciók közül egyet tetszőlegesen kiválasztást. Az osztályon belül kifejtett függvények implicit inline-ok, így sose okozhatnak fordítási hibát.

```
inline void f() {}
```

Az ábra jól demonstrálja a problémát. `f()` egy un. *strong reference*-el jön létre ha nem inline, így a linker



1. ábra. A `main.cpp`-ben vagy a `list.cpp`-ben lévő definíciója szerepeljen `f`-nek a futtatható fájlban?

hibát dob ha több fordítási egységben definiálva van. Ha azonban inline-ként adjuk meg, akkor *weak reference*-ként értelmezi, a meglevő definíciók közül tetszőlegesen kerül egy kiválasztásra. Ez nyilván azt is jelenti, hogy minden ilyen függvény definíciójának meg kell egyeznie, hisz kellemetlen meglepetés érhet minket, ha különböző definíciók közül olyat választ a fordító, melyre nem számítanánk (és ez egyben nem definiált viselkedés is).

0.2.1. Megjegyzés. A legtöbb fordítónál lehet egy LTO (*link time optimalization*) funkciót bekapcsolni, mely a linkelésnél optimalizál, többek között ott végzi el az inlineolást.

0.2.2. Megjegyzés. Az inline függvények hajlamosak erősen megnövelni a bináris kódot, így az erőltetett használatuk nem javallott.

0.2.3. Megjegyzés. Az inline kulcsszó egy javaslat a fordítónak, de nem parancs. Nem inline függvények lehetnek inline-ok, és inlineként definiált függvények lehet mégsem lesznek azok.

Azok a tagfüggvények, melyek nem az osztály törzsében vannak definiálva, nem lesznek inline-ok, ezért volt az, hogy mielőtt szétszedtük a listánkat header fájlra és fordítási egységre, linkelési hibát kaptunk (`Iterator` és `ConstIterator` pár tagfüggvénye külön volt véve).

Cseréljük le a print függvényt:

```
std::ostream& operator<<(std::ostream &os, const List &l)
{
    for(ConstIterator it = l.begin(); it != l.end(); ++it)
    {
        os << *it;
    }
    return os;
}
```

Sajnos ismét fordítási hibát kapunk, hisz a fordító nem tudja mi az az `ostream`, hisz az ismerős `iostream` könyvtár nincs include-olva. Ilyenkor azonban érdemes inkább az `iosfwd` headert beilleszteni az `iostream` helyett, mert ez minden beolvasással és kiíratással kapcsolatos osztály/függvénynek csak a deklarációját tartalmazza, és így csökken a fordítási egység mérete (azonban a cpp fájlban muszáj `iostream`-et használni, hogy a definíciók meglegyenek).

0.2.4. Megjegyzés. Ha szeretnénk egy `std::ostream` típusú objektumot használni, szükségünk lenne az `iostream` könyvtárra, hisz a fordítónak tudnia kéne, mekkora az `std::ostream` mérete, és ehhez szüksége van a teljes osztálydefinícióra. Azonban a referencia vagy pointer típusoknál erre nincs szükség, amíg nem akarunk egy tagfüggvényüket meghívni.

1. Névterek

A kódunkkal kapcsolatban felmerülhet egy másik probléma is: nagyon sok hasznos nevet elhasználtunk, pl. több `Iterator` nevű osztályt nem hozhatunk létre (az un. globális névtérbe vagy *global namespace*-be kerültek), különben a névütközés áldozatai leszünk. Pedig várhatóan nem csak ennek az egy konténernek szeretnék iterátort írni.

Megoldás lehet, hogyha inline class-t hozunk létre, azaz az iterátor teljes deklarációját beillesztjük a `List`-be, így csak a `List` lát rá `Iterator`-ra követlenül, mindenhol máshol úgy kell hivatkozni rá, hogy `List::Iterator`.

```
class List
{
public:
    class Iterator
    {
        //...
    };
    //...
};
```

Azonban szerencsésebb, ha az iterátorainkat egy névtérbe (*namesspace*) rakjuk.

```
namespace detail
{
    class Iterator
    {
        //...
    };
    class ConstIterator
    {
        //...
    };
}
```

Így az `Iterator` és `ConstIterator` osztályra a későbbiekben csak úgy hivatkozhatunk, ha megmondjuk, mely névtérből származnak. Ugyanezzel a módszerrel, ha létrehozunk pl. egy `Vector` nevű osztályt, annak is írhatunk egy `Iterator` nevű osztályt, amit pl. egy `VectorDetail` névtérbe tehetünk.

```
detail::Iterator it;
detail::ConstIterator cit;
```

A névterek segíthetnek abban, hogy logikai egységebre rendezzük a kódunkat. Az egyik legnagyobb ilyen egység az `std` névtér, mely tartalmaz minden függvényt, változót, stb, ami a standard részét alkotja.

Lehet névtereket egymásba is ágyazni, erre lehet példa a C++11-es `chrono` könyvtár, mely az `std` névteren belül számos dolgot a `chrono` alnévtérben tárol.

1.1. Argument Dependent Lookup

A szokásos Hello World programban nem feltétlenül kell használni a `using namespace std;` sort. Azonban megállapítható, hogy a `<<` operátor globális, `std` névtérbeli, mégse kellett elé `std` névtérre hivatkozás (azaz *explicit namespace resolution*). Ez az úgynevezett ADL-nek (*Argument Dependent Lookup*) köszönhető, melyet a fordító alkalmaz: az adott függvényt abban a névtérben keresi először, ahol az argumentumai megtalálhatóak.

Példaképp, a Hello World programban a `cout` változó az `std` névtér tagja, így a fordító először az `std` névtérben fogja keresni a `<<` operátort.

2. Typedef

A `typedef` kulcsszó szinonimák létrehozására használatos, és ha ügyesen használjuk, ki lehet használni valamennyi előnyét.

Hozzunk létre egy osztályt, melynek adatokat kell tárolnia. Tegyük fel, hogy egyenlőre nem fontos számunkra az, hogy milyen konténerben tároljuk az adatokat az osztályon belül, és a példa kedvéért követeljük meg ettől a leendő konténertől hogy rendelkezzen `push_back` tagfüggvénnyel. Az `std::vector` konténerrel ez így nézhetne ki:

```
class StoreIntData
{
private:
    std::vector<int> data;
};
```

Írjunk egy tagfüggvényt, mellyel két `StoreIntData` típus tárolt adatait össze tudjuk fűzni.

```
class StoreIntData
{
public:
    std::vector<int> merge(StoreIntData &other)
    {
        std::vector<int> ret;
        for(int i = 0; i<data.size(); i++) ret.push_back(data[i]);
        for(int i = 0; i<other.size(); i++) ret.push_back(other.data[i]);
        return ret;
    }
private:
    std::vector<int> data;
};
```

2.0.1. Megjegyzés. Később látunk majd példát egy ennél jóval elegánsabb megoldásra is.

Tegyük fel, hogy az implementáció egy pontján úgy döntünk, mégse `std::vector`-ban, hanem `std::deque`-ban szeretnénk tárolni (erről a konténerről részletesen szó lesz később, egyenlőre legyen elég annyi, hogy az `std::vector`-hoz hasonló). Ekkor rákényszerülünk arra, hogy minden helyen, ahova `std::vector`-t írtunk, módosítanunk kelljen, és a kódismétlés áldozatai lettünk. Ehelyett használjunk egy `typedef`-et!

```
class StoreIntData
{
public:
    typedef std::deque<int> Container;

    Container merge(StoreIntData &other)
    {
        Container ret;
        for(int i = 0; i<data.size(); i++) ret.push_back(data[i]);
        for(int i = 0; i<other.size(); i++) ret.push_back(other.data[i]);
        return ret;
    }
private:
    Container data;
};
```

Így ha módosítanunk kell a konténerünk típusát, elég a typedefet átírni.

Visszatérve a láncolt listánkra, a módosítás után `List` nem tudja, mi az az `Iterator`, hisz az egy `datail` nevű névtérben van, ezért vagy minden `Iterator`-t lecserélünk `detail::Iterator`-ra, vagy pedig létrehozunk egy szinonimát.

```
class List
{
public:
    typedef detail::Iterator Iterator;
    typedef detail::ConstIterator ConstIterator;
```

```
    //...
};
```

A typedef segítségével viszont még egy dolgot nyertünk: mivel ezen a szinonimák publikusak, így az osztályon kívül is tudunk rájuk hivatkozni így:

```
List::Iterator it = head.begin();
```