C++

2. HÉT – C++1X FEATURE-ÖK

A C++ mögött máig aktív szabványosító testület áll – minden 3-4 évben új szabvány jelenik meg.

Szerencsére a 'gyártók' (akik a fordítókat készítik) a szabványt komolyan is veszik.

AZ ELSŐ NÉHÁNY HÉTBEN A C++11 ÉS AZÓTA MEGJELENT ÚJÍTÁSOKRÓL ESIK SZÓ

Függvények parciális alkalmazása

Minden hatékony programozási paradigma arról szól, hogy tegyük érthetőbbé / intuitívvá a kód felépítését, és könnyítsük meg annak újrafelhasználását (DRY = Don't Repeat Yourself)

Az OOP-ben ezt örökléssel (interfészekkel), virtuális függvényekkel és futásidejű tesztekkel / konverziókkal (dynamic_cast) érhetjük el.

A (generikus) funkcionális programozás paradigma pedig osztályok helyett a függvényeket helyezi előtérbe, és inkább függvények szintjén könnyíti meg a kód újrafelhasználását:

- template-ek értelme: ne kelljen 1 függvényt / osztályt 2x leírni csak azért, mert kicsit más típusokkal vagy értékekkel (mint hiperparaméterekkel) működik
- magasabbrendű függvények értelme: egy műveletláncban egy adott művelet legyen parametrizálható
- parciális alkalmazás értelme: ne kelljen 20 változatot egy függvényből megírni csak azért, mert egy interfész olyan függvényt vár, ami kevesebb argumentumot fogad

Függvények parciális alkalmazása

Példa: Az std::count_if függvény végigmegy egy tartományon (STL konténerekben) és megszámolja, hogy hány elemre teljesül egy egyváltozós bináris függvény (predikátum)

Például: hány ember idősebb 18 évesnél… hány autó régebbi mint 10 éves.

Az std::count_if egyváltozós bináris függvényt vár, nem kétváltozósat. De a nagyobb, mint operator kétváltozós. Az std::greater<T> konstrukció is. Kellene egy nagyobb-mint-X függvény, ami X-szel parametrizálható.

Ilyen esetekre a meglévő tudással (universal references, perfect forwarding) tudunk olyan osztályt készíteni, ami függvényként viselkedik és egy neki átadott függvényt úgy hív meg mindig, hogy annak 2. paramétere fix (régebben volt olyan, hogy std::bind2nd, de kivették a sztenderd könyvtárból)

Függvények parciális alkalmazása

Nézzük, hogyan is működne egy ilyen megoldás!

```
template <typename Function, typename SecondArgType>
class partial_apply_2nd {
    Function function;
    SecondArgType second;
public:
    partial_apply_2nd(Function f, SecondArgType a) : function(f),
second(a) {}
    template <typename FirstArgType>
    decltype(auto)
    operator()(FirstArgType&& first) {
        return function(
        std::forward<FirstArgType>(first),
        second);
    }
};
```

Függvények parciális alkalmazása

... és itt láthatjuk a felhasználás néhány lehetséges módját:

```
partial_apply_2nd<std::function<int(int,int)>, int> somethingPlus5([](int a, int b) -> int
{return a + b; }, 5);

std::cout << somethingPlus5(2) << std::endl;

std::cout << somethingPlus5(5) << std::endl;


partial_apply_2nd<std::function<bool(int, int)>, int> greaterThan18(std::greater<int>(), 18);

std::cout << "19 is greater than 18? " << greaterThan18(19) << std::endl;

std::cout << "17 is greater than 18? " << greaterThan18(17) << std::endl;
```

Függvények parciális alkalmazása

2011 előtt a C++ sztenderd könyvtára pontosan ilyen célra biztosította az std::bind1st és std::bind2nd függvényeket. 2011-től ezek deprecated státuszba kerültek és 2017 óta teljesen eltávolították őket a könyvtárból. Helyettük van viszont a sokkal általánosabb **std::bind()**, amely:

- nemcsak kétargumentumos, hanem akárhány argumentumos függvényekre (sőt, bármilyen *callable* egyedre) alkalmazható
- lehetővé teszi akármelyik (és egyszerre akár több) argumentum "lekötését" bármilyen sorrendben

Legegyszerűbb esetben minden argumentumot leköthetünk – persze ez így már nem parciális alkalmazás:

```
auto bound = std::bind(std::greater<double>(), 6, 24);

bool is_6_gt_24 = bound();

if (!is_6_gt_24) std::cout << "Nooo, dummy, 6 is not greater than 24" << std::endl;

std::function<int()> twoplus3 = std::bind([](int a, int b) -> int {return a + b; }, 2, 3);

std::cout << "2 + 3 = " << twoplus3() << std::endl;
```

Függvények parciális alkalmazása

A valódi parciális alkalmazáshoz std::bind esetében a le nem kötött argumentumokat ún. placeholderekkel kell jelölni. Ezek jelölése _1, _2 és így tovább (attól függően, hogy a keletkező függvény hányadik argumentumát helyettesítjük be pont arra a helyre):

```
auto six_gt_n = std::bind(std::greater<double>(), 6, std::placeholders::_1);

auto seven_lt_n = std::bind(std::greater<double>(), std::placeholders::_1, 7);

if (six_gt_n(5)) { std::cout << "6 is greater than 5" << std::endl; }

if (seven_lt_n(8)) { std::cout << "7 is NOT greater than 8" << std::endl; }
```

Függvények parciális alkalmazása

Ugyanez megcsinálható osztályok tagfüggvényeire is (metódusokra), csak figyelembe kell venni, hogy azoknak van egy implicit első argumentuma, ami a this mutató.

Például vegyük ezt az egyszerű osztályt:

```
class OperationsManager {
    int count = 0;
public:
    int add(int a, int b) {
        count++;
        std::cout << "computing " << a << "+" << b << std
            ::endl;
        return a + b;
    }
    void printStatus () const {
        std::cout << "operations called " << count << "
            times" << std::endl;
    }
};
```

Függvények parciális alkalmazása

Néhány lehetséges felhasználása az add() metódusnak bind segítségével:

```cpp
#include <iostream>
#include <functional>
#include <algorithm>

using namespace std::placeholders;

auto addTo3 = std::bind(&OperationsManager::add, _1, 3, _2);

int main() {
    OperationsManager opm;
    std::cout << addTo3(opm, 5) << std::endl;
    std::cout << opm.add(4,6) << std::endl;
    opm.printStatus();

    std::vector<OperationsManager*> opms { new
        OperationsManager(), new OperationsManager()};
    std::for_each(opms.begin(), opms.end(), std::bind
        (&OperationsManager::add, _1, 5, 3));

    for (auto o : opms) {
        o->printStatus();
        delete o;
    }
    return 0;
}
```

```
> clang++-7 -pthread -std=c++17 -o main main.cpp
> ./main
computing 3+5
8
computing 4+6
10
operations called 2 times
computing 5+3
computing 5+3
operations called 1 times
operations called 1 times
>
```

Függvények parciális alkalmazása

Egy placeholder többször is alkalmazható – ilyenkor természetszerűleg a később átadott argumentum értéke több helyen is szerepel majd a hívásban.

Vegyük észre, hogy a placeholderek fordított alkalmazásával egy függvény argumentumainak sorrendje is felcserélhető.

Természetesen, mivel az std::bind nagyon általános, ezért tartozik hozzá overhead – céltól függően érdemes meggondolni, hogy alkalmazzuk-e.

Szerencsére ami elérhető std::bind-dal, elérhető lambda függvényekkel is. Ehhez az kell, hogy:

- amit változó alapján kötünk le, az legyen capture
- amit érték alapján kötünk le, az legyen benne a lambda törzsében
- a placeholderek legyenek lambda argumentumok

Függvények parciális alkalmazása

Ennek a megközelítésnek az az előnye, hogy a lambda függvényeket a fordító jobban optimalizálni tudja (mivel a nyelv szerves része a lambda függvény és nem template-elt könyvtári függvény)

```
int six = 6;

auto six_gt_n_lambda = [&six](int n) -> bool {return six > n; };

auto seven_lt_n_lambda = [](int n) -> bool {return n > 7; };

if (six_gt_n_lambda(5)) { std::cout << "6 is STILL greater than 5" << std::endl;}

if (seven_lt_n_lambda(8)) { std::cout << "7 is STILL NOT greater than 8" << std::endl;}
```

“Körriolás” (currying)

A “körriolás” szó a saját neologizmusom, egyszer egy vicces pillanatomban találtam ki

A “currying” szó Haskell Curry nevű amerikai matematikus nevéből ered

- A Haskell programozási nyelvet is róla nevezték el
- (Azon gondolkoztam, milyen vicces lenne, ha csinálnának egy Ádám nevű programozási nyelvet, és lenne benne egy olyan művelet is, hogy “csapózás”)

Komolyra fordítva a szót, a currying arról szól, hogy n-bemenetű fv-eket áttranszformálunk egybemenetűre.

- Mindig meg lehet csinálni, hogy a transzformáció által kiadott 1 bemenetű függvény egy olyan másik függvényt ad vissza, ami pedig n-1 bemenetet vár… és ugyanúgy viselkedik, mint az eredeti függvény, amikor az 1. bemenete “le van kötve”

```cpp
#include <iostream>

int add(int x, int y) { return x+y; }
auto addCurried (int x) {
    return [x](int y) -> int { return x + y; };
}

int main() {
    auto addTo5 = addCurried(5);
    std::cout << "5 + 3 = " << addTo5(3) << std::endl;
    return 0;
}
```

```
clang++-7 -pthread -std=c++17 -o main main.cpp
./main
5 + 3 = 8
```

“Körriolás” (currying)

A currying tehát kicsit olyan, mint a bind, azzal a különbséggel, hogy:

- Currying esetén csak sorrendben lehet lekötni az eredeti argumentumokat
- Bindolás esetén minden változót vagy le kell kötni, vagy placeholdert kell hozzá (explicite) megadni. Currying esetén csak annyit teszünk, hogy lekötjük az 1. argumentumot (vagy az első x-et, általános esetben)

De miért tenne ilyet bárki? Mi az értelme?

Tegyük fel, hogy egy interfészhez való hozzáféréskor sok paramétert kell megadnunk. Például ha egy adatbázishoz akarunk lekérdezést intézni, meg kellhet adni:

- Egy “connection handle-t” (a kapcsolatot reprezentáló objektumot, ami tartalmazza pl. a jogosultságunkat kifejező tokent)
- Egy “session handle-t” (ha pl. egy tranzakcióba több lekérdezést szeretnénk tenni, ehhez szükséges lehet egy külön session-t létrehozni)
- Magát a lekérdezést (az ahhoz szükséges táblát, filtert pl.)

“Körriolás” (currying)

Tegyük fel, hogy egy interfészhez való hozzáféréskor sok paramétert kell megadnunk. Például ha egy adatbázishoz akarunk lekérdezést intézni, meg kellhet adni egy connection handle-t, egy session handle-t, a táblát és a filtert.

Vegyük észre, hogy a connection handle-t pl. fárasztó lehet mindig megadni. Mi lenne, ha csinálnánk egy olyan “körriolt” függvényt / metódust, ami referencia szerint leköti a connection handle-t, hogy később már ne kelljen megadnunk?

- Persze ezt önmagában még meg lehetne oldani más OOP módszerrel is, ha mondjuk a ConnectionHandle objektum tenné elérhetővé a query() metódust, ami csak session-t, táblát és filter-t vár
- De a helyzetet bonyolíthatja, amikor már session-t se szeretne a felhasználó külön megadni. Hiszen ha pl. egy read-only adatbázishoz férünk hozzá, igazából felesleges külön session-öket csinálni (a tranzakció mint fogalom okafogyottá válik, hiszen soha semmi nem fogja felülírni az adatbázist)

“Körriolás” (currying)

Ennél továbbmenve, még az is elképzelhető, hogy egy adott user ugyanazzal a kapcsolattal mindig csak ugyanazt a táblát szeretné szűrni, ezért okafogyottá válik a tábla külön megadása minden esetben…

- Tehát már ennél az alap példánál is ott tartunk, hogy a szokásos OOP megközelítéssel 2 külön query metódusra lenne szükség a ConnectionHandle osztályban, és még egy külön query metódusra a Table osztályban is…
- A másik alternatíva, hogy 4 vagy 5 különböző query metódust implementálunk a ConnectionHandle osztályban.
- E helyett viszont létrehozhatunk mindössze 1 query metódust is, ami körriolt módon működik. Jelenlegi tudásunkkal ezt így tudjuk elérni:

```
auto query(const std::string& conn) {
  return [&conn](const std::string& sess) -> auto {
      return [&conn, &sess](const std::string qdata) -> auto {
          return std::string(conn + "/" + sess + ": " + qdata);
      };
  };
}
```

“Körriolás” (currying)

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

auto query(const std::string& conn) {
  return [&conn](const std::string& sess) -> auto {
      return [&conn, &sess](const std::string qdata) -> auto {
          return std::string(conn + "/" + sess + ": " + qdata);
      };
  };
}


int main() {
    std::string connl("localConnection");
    auto remoteQuery = query("remoteConnection");

    std::string sessA("sessionA");
    auto remoteQuerySessionC = remoteQuery("sessionC");

    std::cout << query(connl)(sessA)("something") << std::endl;
    std::cout << remoteQuery("sessB")("anything") << std::endl;
    std::cout << remoteQuerySessionC("nothing") << std::endl;
    return 0;
}
```

```
clang++-7 -pthread -std=c++17 -o main main.cpp
./main
localConnection/sessionA: something
/sessB: anything
/sessionC: nothing
```

Függvény-kompozíció

1986-ban felkérték Donald Knuth-t, hogy egy újságba készítsen egy elegáns programot, ami megszámolja, hogy egy szövegesfájlban melyik szó hányszor fordul elő, és sorrendezi azokat. (*Communications of the ACM folyóirat "Programming Pearls" rovata*)

- Az eredmény egy 10-oldalas Pascal program volt (persze rengeteg commenttel az olvasók részére)

Válaszul Doug McIlroy elkészítette ugyanezt a programot UNIX shell scripteléssel, 6 sorban:

```
tr -cs A-Za-z '\n' |
    tr A-Z a-z |
    sort |
    uniq -c |
    sort -rn |
    sed ${1}q
```

Függvény-kompozíció

Ez a kettő persze nem összevethető, mivel a UNIX shell nyelveket direkt ilyen alkalmazásokra fejlesztették, + egy-egy ilyen parancs (mint ‘sed’) mögött azért összetett algoritmusok állnak

Mindenesetre ez jól példázza a funkcionális és imperatív megközelítés közötti különbségeket

Nézzük meg, hogy C++-ban hogyan lehetne egy hasonlót csinálni.

A terv:

- Készítsünk egy words() nevű függvényt, ami a stringből a szóközöket kivéve visszaad egy listát a szavakkal
- Készítsünk egy count() nevű függvényt, ami összeszámolja, hogy adott szó hányszor szerepel a listában. Az eredmény lehet egy unordered_map, mivel a szavak egyediek kell, hogy legyenek, de nem a szavak szerint akarunk sorrendezni (ezért nem kell map-et használni)
- Készítsünk egy reverse() nevű függvényt, ami egy vektort ad vissza Int-String párokkal
- Végül egy sortVectorOfIntStringPairs() függvénnyel sorrendezzük az eredményt!

Függvény-kompozíció

A feladat megvalósítását meghagyjuk gyakorlati órára.

Mindenesetre a feladat 50 sorban megoldható (majd látni fogjuk). Maga az alkalmazás:

```
int main() {
    VectorOfIntStringPairs voisp =
        sortVectorOfIntStringPairs(
            reverse(
                count(
                    words("this string is a very very very short string this one!")
                )
            )
        );

    for (auto& v : voisp) {
        std::cout << v.first << ": " << v.second << std::endl;
    }
    return 0;
}
```