A programozás alapjai 2.

Hivatalos laborsegédlet hallgatók számára.

Alapprobléma - függvények

Tegyük fel, hogy írtunk egy rendező függvényt int típusra, mely átvesz egy tömböt és a tömb elemszámát paraméterként. Legyen ez a függvény a következő:

```
void sort(int* array, int count) {
  for (int i = 0; i < count-1; i++) {
    int min = i;
    for (int j = i + 1; j < count; j++) {
      if (array[j] < array[min])
      min = j;
    }
    if (min != i) {
      int c = array[i];
      array[i] = array[min];
      array[min] = c;
    }
  }
}
```

Aztán szükségünk van egy double tömböt rendező függvényre. Meg egy string tömböt rendező függvényre. Meg egy saját Computer típusú elemekből álló tömböt rendező függvényt. És ez így mehetne tovább nagyon sok típusra.

Meg lehetne oldani úgy a problémát, hogy lemásoljuk a megírt függvényt, kicseréljük a típusokat a megfelelőre (az elemszám azért marad int), és kész.

Na ez az elfogadhatatlan megoldás, mivel pont a duplikáció esete, amit nagyon nem szeretünk, így minél inkább el szeretnénk kerülni.

Függvénytemplate

A tamplate-ek a generikus programozás alapjai, lehetővé teszik, hogy típusoktól független függvényeket írjunk. Egyszer kell megírni a függvényt, mégis több típussal használható. A template paraméterek lehetővé teszik, hogy a használandó típus(oka)t is átadjuk paraméterként.

A függvény törzsétől függően a felhasznált típusnak bizonyos kritériumoknak meg kell felelnie. Például ha a függvényünkben cout stream-re kiírjuk az átvett paramétert, csak akkor működik a függvény, ha a paraméter típusára meg van írva az operator<<.

A fentebb említett függvényünk a következőképp írható meg függvény template-ként:

```
template <class T>
void sort(T* array, int count) {
  for (int i = 0; i < count-1; i++) {
    int min = i;
    for (int j = i + 1; j < count; j++) {
      if (array[j] < array[min])
        min = j;
    }
    if (min != i) {
      T c = array[i];
      array[i] = array[min];
      array[min] = c;
    }
  }
}
```

Ebben az esetben csak akkor használható a függvény T típusú tömbbel, ha a T típusra meg van írva az operator< és az operator=.

Template paraméter

A rendezőfüggvény templateparamétere T. Ez a template paraméter egy típust reprezentál, ami még nem lett definiálva, de a függvényben használhatjuk úgy, mint egy szokványos típust.

Amikor egy adott típussal meghívjuk a függvényt, a compiler automatikusan generál egy függvényt (ha még nem tette meg korábban), ahol a használt típus már nem a template paraméter, hanem a hívásban szereplő paraméter típusa. A template-ek fordítási időben fejtődnek ki azokra a típusokra, amelyekkel használjuk, a többivel nem. Csak fordítási időben derülnek ki a hibák.

Pl. ha a rendező függvényt mi meghívjuk double típusú tömbbel, létrehoz egy olyan függvénypéldányt, ahol minden T típus helyén double áll.

Függvénytemplate használata

Vezessünk be egy új függvényt, ami kiírja két, paraméterként átvett változó négyzetösszegét:

```
template <class T>
void squareSum (const T a, const T b) {
   cout << a * a + b * b << endl;
}
```

Implicit példányosítás

Függvénytemplate-eket használhatunk úgy, mint egy egyszerű, szokványos függvényeket.

Pl.:

```
squareSum(2, 3);
squareSum(1.0, 2.0);
```

Mivel mindkét paraméter ugyanolyan típusú a különböző hívásokkor (az elsőben int, a másodikban double), a fordító automatikusan ki tudja találni, hogy mi a template paraméter, milyen típussal kell példányosítania a függvényt.

Explicit példányosítás

Választhatóan

Az előző két példában mutatott esetben kiírhatnánk konkrétan, hogy milyen típussal hívjuk a függvényt a következő módon:

```
squareSum<int>(2, 3);
squareSum<double>(1.5, 2.5);
```

Ezt nem kötelező megtenni ezekben az esetekben.
Ezek a függvények már teljesen olyanok, mint a szokványos függvények.

Kötelezően

Bizonyos esetekben azonban muszáj konkrét típust megadni, aminek fő oka, hogy a template függvény nem végez semmilyen konverziót. Például nem tudja végrehajtani a fordító a `squareSum(2.5, 3);` hívást, mivel a két paraméterének típusa eltérő, és a függvény mindkét paraméternek egy közös, T típust vár. Nem tudja kitalálni, mi legyen ebben az esetben.

Ekkor két lehetőségünk van. Az első, hogy az egyik paramétert átkonvertálom híváskor a másik típusára, hogy ki tudja találni a fordító a típust. A második, hogy explicit megadjuk, milyen típusú paramétereket várjon, így ha mégsem olyat kapna, megpróbálja átkonvertálni a paramétereket erre (ha nem sikerül, hibát kapunk). Így a két lehetséges megoldás:

```
squareSum(2.5, double(3));
squareSum<double>(2.5, 3);
```

Ha a paraméterek konstansok helyett változók lennének, ugyanezek a szabályok lennének érvényesek.

Alapprobléma - osztályok

Nagyjából ugyanaz a probléma, mint korábban, csak az osztály tagjainak kell tudnia használni több típust.

Például szeretnénk írni egy Stack osztályt, amiben dinamikus tömbként tárolhatunk ugyanolyan típusú elemeket. Viszont szeretnénk külön csak int, double, string stb. típusú elemeket tároló Stack-et. Ekkor írhatnánk külön-külön osztályt pl. StackInt, StackDouble, StackString stb. néven, de ez nagy pazarlás lenne. Mindegyiknél külön-külön az összes függvényt megírni minek? Csak időpazarlás, karbantarthatatlan és csúnya.

Parametrizált osztályok

Erre a problémára adnak megoldást a template osztályok. Pl.:

```
template <class T>
class Stack{
  T* pData;
  int elementNum;
public:
  int push(T newElement); // visszatérési érték=0 ha sikeres; -1, ha nem
  //…
};
```

Ekkor a pData dinamikus tömb által tárolandó elemek típusát a Stack osztály példányosításakor adhatjuk meg, de az osztály definícióját úgy írhatjuk meg, hogy ezt nem tudjuk.

Amennyiben egy függvény definícióját az osztályon kívül írjuk meg, minden definíció elé ki kell írni a template-et. Pl.:

```
template <class T>
int Stack<T>::push(const T& newElement) {..}
```

Osztálytemplate használata

A Stack osztály használata pl.:

```
Stack<int> integers;
Stack<double> doubles;
Stack<Stack<int>> complicated;
```

A felparaméterezett osztály már egy közönséges osztály, bárhol használható, ahol egyszerű osztály is.

Default template argumentum

Template paraméternek megadható default érték, pl. ha azt szeretnénk, hogy alapértelmezetten int-eket tároló Stack jöjjön létre: `template <class T = int>`

Ekkor, ha int típusú elemeket tároló Stack-et szeretnénk, létrehozhatjuk ilyen módon is:

Stack<> s;

Mi lehet template paraméter

Template paramétek lehetnek a következők:

- típus
  - int, double, string
  - felparaméterezett template osztály: pl. `Stack<Stack<int>> s;` jó, ekkor a paraméter `Stack<int>`
- típusos konstans
  - konstans
  - konstans változó
- template osztály
  - nem felparaméterezett template osztály: pl. `Vector<Stack> s;`
  - ehhez a Vector osztály deklarációjának a következő template-tel kell rendelkeznie:

```
template <template <class V> class T>
class Vector {
  T a;
  V b;
};
```

Tagfüggvény template

Akár template osztályról, akár egyszerű osztályról van szó, a tagfüggvényeknek lehetnek külön templateparaméterei. Template osztálynál ezt az indokolhatja különösebben, ha az osztály template paramétertől különböző template paramétert szeretnénk átvenni.

Pl.:

```
template <class T>
class MyClass {
  //…
  template <class V>
  V myFunc(V param);
};
```

Annyi verzió generálódik belőle automatikusan, ahányféle különböző típussal használjuk.

Öröklés template osztályból

Csak osztályból lehet örököltetni. Egy template osztály addig template osztály, amíg van nem rögzített template paramétere. Amint minden templateparaméter rögzítve lett, a template osztályból általános osztály lesz.

Pl:

```
template <class T, class U>
class Base {...}

class Derived: public Base<int, V> {...}
```

Template vs. Öröklés

Ha ugyanazt a viselkedést szeretnénk leírni több típusra, használjunk template-et.

Ha felüldefiniálható viselkedésre van szükségünk, használjunk öröklést.

template<class T> vs template<typename T>

Technikailag a két kulcsszó ugyanazt jelenti. Azért létezik mégis két kulcsszó rá, mert az ősidőkben Stroustroup jónak látta, hogy egy szóhoz több jelentést rendelünk (static-nél még rosszabb a helyzet).

Van azonban pár speciális eset, amikor különböznek, mint például:

```
template<template<class> typename MyTemplate, class Bar> class Foo { };    //
nem fut le
template<template<class>    class MyTemplate, class Bar> class Foo { };    //
lefut
```

Amikor template templateket készítesz, a class kulcsszót kell használnod.

Template osztály dekompozíció

Template osztályt nem lehet két fájlra bontani olyan módon, hogy az osztály deklarációja egy header, a definíciója pedig egy .cpp fájlban van, a fordítás sikertelen lesz. Ennek oka a következő: A fordítási folyamat második lépésében a compiler a template osztályt szeretné lefordítani egyszerű osztállyá minden olyan értékre, mellyel használták. Pl. ha a Stack template osztályt használjuk int és double típusokkal, akkor két osztályt generálna le. Amikor egy helyen lát egy ilyen példányosítást, a beinclude-olt header fájlból (muszáj include-olva lennie, különben nem lehetne használni) látja, hogy milyen tagváltozói és tagfüggvényei vannak.

El szeretné készíteni az osztályhoz tartozó memóriastruktúrát, de mivel csak a header fájlt látja, fogalma sem lesz róla, hogy a függvényeket hogyan kell elkészíteni. Mivel a .cpp fájlt ő nem ismeri (fájlonként fordít, a fájlok közti kapcsolatokat a következő lépésben a linker végzi el). A 2. lépésben még nem keletkezik hiba, a fordító feltételezi, hogy valahol már léteznek a függvények a szükséges típusokkal (a példában double és int típussal). Viszont amikor a template osztályhoz tartozó .cpp fájlt fordítja (függetlenül az előző fájl fordításától), mivel nincs megadva típus a template osztálynak, nem tudja a függvényekből létrehozni a típusos függvényeket (ebben a fájlban senki nem mondja meg neki, mik kellenének).

A probléma a linkeléssel van. Mivel a compiler nem tudta példányosítani a függvényeket a megfelelő típusokkal, a linker nem talál a generált típusos osztályok függvényeinek definíciót és linkage error keletkezik.

Megoldás: Tegyük egy fájlba az egy osztályhoz tartozó deklarációkat és definíciókat, így az include-oláskor a függvények definíciója is elérhetővé válik. Ettől még nem kell az osztályon belül, inline megírni a függvények törzsét, lehet az osztálydeklaráció után.