Ćwiczenia #2

W ramach tych ćwiczeń pojawią się trzy nowości:

1. Omówione już dość szczegółowo wykorzystanie funkcji wirtualnych.
2. Obiekty, które zachowują się jak funkcje (tzw. obiekty funkcyjne). Popatrzmy na przykład:

   ```
   const range& refRange = bcrange(3, 5);
   cout << refRange(3) << endl;
   ```

   Ewidentnie zmienna `refRange` jest wykorzystywana jak funkcja z jednym argumentem (nasza decyzja).
3. Zastosowanie słowa kluczowego `using` w definicji klasy, do „przekierowania” konstruktorów w klasach pochodnych na konstruktor w klasie bazowej.

Klasa `range` i pochodne

Jak poprzednio będziemy potrzebować klasy reprezentującej przedział obustronnie otwarty i klas pochodnych reprezentujących przedziały domknięte.

Tym razem muszą być spełnione następujące warunki:

1. Obiekty przedziału będę chciał wykorzystać tak, jakby były funkcjami (obiekty funkcyjne).
2. Trzeba napisać jeden operator << wyprowadzenia na strumień wyjściowy (dla klasy bazowej – implementacja moja) i powinien on działać dla wszystkich typów przedziałów.
3. Wszystkie klasy powinny być zdefiniowane w oddzielnym pliku nagłówkowym `range.h`.

Taka skromna hierarchia klas, o ile przejdzie bez błędów testy (plik `zad_m2.cpp`) jest rozwiązaniem drugiego zadania za 2 punkty.

Obiekty funkcyjne

Obiekty funkcyjne przydają się w wielu miejscach, ale szczególnie dobrze widać to przy korzystaniu z algorytmów z biblioteki standardowej. Załóżmy, że naszym zadaniem jest napisanie funkcji, która ma zliczać w kolekcji obiekty, których wartość jest większa niż wartość parametru przekazanego do tej funkcji:

```
int count_greater(const vector<int>& coll, int arg)
{
   return count_if(coll.begin(), coll.end()), ??? );
}
```

Byłoby miło skorzystać z algorytmu zliczania `count_if` gdyby nie problem warunku (predykatu) zliczania `???`. Można napisać tutaj stosowne wyrażenie lambda, ale nie zawsze jesteśmy w stanie to zrobić (np. piszemy w C++98). Rozwiązaniem jest odpowiednia klasa, której obiekt daje się użyć jako predykat jednoargumentowy.

Obiekty funkcyjne

W klasie musi być zdefiniowany operator (), czyli operator wywołania funkcji:

```
class greater_than
{
   int tr_val;
public:
   greater_than(int treshold) : tr_val(treshold) {}
   bool operator()(int val) { return val > tr_val; }
};
```

Najważniejszą zaletą takiego obiektu jest to, że może mieć *stan* – w tym przypadku wartość składowej `tr_val`. Obiekty tej klasy można użyć tak, jakby były funkcjami:

```
   greater_than gt(7);
   for (int i = 1; i < 15; i += 2)
      cout << gt(i);
```

0000111

Obiekty funkcyjne

Mając klasę `greater_than` mogę napisać:

```
int count_greater(const vector<int>& coll, int arg)
{
    greater_than gt(arg);
    return count_if(coll.begin(), coll.end(), gt);
}
// ....
    vector<int> vtst{ 1, 3, 5, 7, 5, 3, 1, -1 };
    cout << count_greater(vtst, 4) << endl;
```

```
3
```

W tym momencie widać już chyba miejsce zastosowania obiektu funkcyjnego w tym zadaniu. Zamiast dość paskudnie wyglądającej funkcji `count_in_range` chciałbym pisać:

```
    count_if(coll.begin(), coll.end(), rng);
```

Obiekty funkcyjne – jak to zrobić?

Implementację takich możliwości w `range` i klasach potomnych można zrobić na wiele sposobów. Ważne, żeby uniknąć pierwszego odruchu (u mnie to jest: świetnie zamieniam `inrange` na `operator()` bo to w końcu tylko zmiana nazwy) i chwilę pomyśleć.

Po tej chwili namysłu wychodzi mi, że wystarczy w klasie bazowej zdefiniować `operator()`, który wywoła stosowną metodę. Co najlepsze, ten operator nie musi być funkcją wirtualną.

Zwróćcie uwagę, że to jest ciągle operator jednoargumentowy; to stan obiektu ma dwa elementy, czyli krańce przedziału.

Jeden operator <<

Warto spojrzeć na elementy wspólne moich implementacjach <<:

```
os << '(' << rng.get_low() << ", " << rng.get_high() << ')';
os << '<' << rng.get_low() << ", " << rng.get_high() << '>';
os << '<' << rng.get_low() << ", " << rng.get_high() << ')';
os << '(' << rng.get_low() << ", " << rng.get_high() << '>';
```

Bardzo podobne!

A gdyby w range były metody `left_paren` zwracająca znak lewego nawiasu przedziału i `right_paren` zwracająca znak prawego nawiasu przedziału? Mógłbym napisać << raz, a dobrze:

```
std::ostream& operator<<(std::ostream& os, const range& rng)
{
   return os << rng.left_paren() << rng.get_low() << ", "
             << rng.get_high() << rng.right_paren();
}
```

Taką implementację widać w `zad_m2.cpp` i przy niej pozostańmy.

Inne użycie `using`

Spójrzmy na przykład Stroustrupa:

```
class Derived : public Base {
public:
   using Base::f;      // lift Base's f into Derived's scope
                       // -- works in C++98
   void f(char);       // provide a new f
   void f(int);        // prefer this f to Base::f(int)

   using Base::Base;   // lift Base constructors Derived's
                       // scope -- C++11 only
   Derived(char);      // provide a new constructor
   Derived(int);       // prefer this constructor to
                       // Base::Base(int)
   // ...
};
```

Zadanie małe nr 2

Zadanie małe nr 2 polega na zaimplementowaniu czterech klas reprezentujących przedziały zgodnie z uwagami w tym pliku.

Proszę wysyłać tylko ten jeden plik `range.h` (oczywiście po uruchomieniu wszystkich testów z `zad_m2.cpp`).

Na początku pliku trzeba umieścić w komentarzu imię, nazwisko i nr albumu autora.

Rozwiązanie należy złożyć w `Moodle` do:

13 marca 2022 23:59.