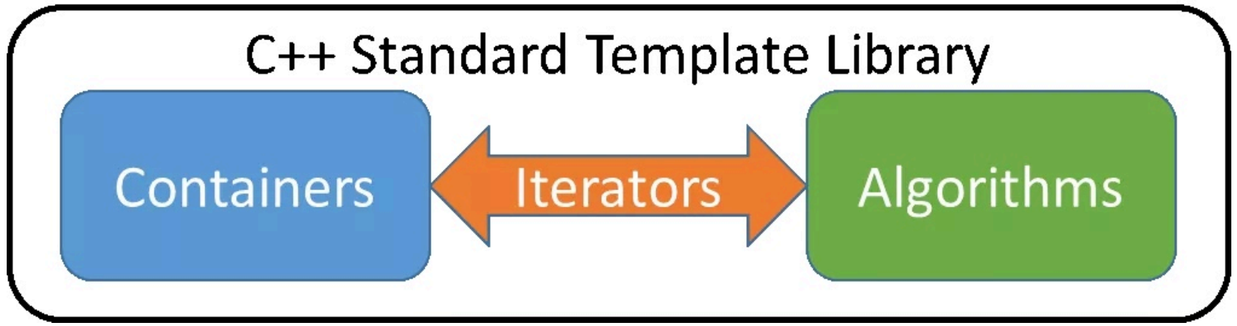
Objektinis Programavimas

Objektiškai orientuotas dizainas.

Bendrinis programavimas

Iš čia kylama į žvaigždes



Turinys

Motyvacija (1)

— Vietoje atskirų funkcijų kiekvienam tipui:

```
int max(int left, int right) {
    return (left > right) ? left : right;
}

double max(double left, double right) {
    return (left > y) ? left : right;
}
```

— Sukuriame **vieną** šabloninę (angl. template) funkciją panaudojant typename arba class raktažodį:

```
template<typename T> // ekvivalentu: template<class T>
T max(T left, T right) {
    return (left > right) ? left : right;
}
```

Motyvacija (2)

— Kadangi T tipo objektai gali būti dideli, todėl efektyviau būtų naudoti objektų nuorodas &:

```
template<class T>
T max(const T& left, const T& right) {
    return left > right ? left : right;
}

// viskas taip pat, tik return tipas T&
template<typename T> // `typename T` vietoje `class T`
const T& max(const T& x, const T& y) { // grąžina T&
    return (x > y) ? x : y;
}
```

Motyvacija (3)

```cpp
int main() {
    // surask max iš dviejų `int`
    int i1 = max<int>(1, 2);
    std::cout << i1 << '\n'; // grąžina 2

    // galima ir be `<int>`:
    int i2 = max(3, 4);      // kviečia: max(int, int)
    std::cout << i2 << '\n'; // grąžina 4

    // surask max iš dviejų `duoble`
    double d = max(3.14, 2.72); // kviečia: max(double, double)
    std::cout << d << '\n'; // grąžina 3.14

    // surask max iš dviejų `char`
    char ch = max('r', '7'); // kviečia: max(char, char)
    std::cout << ch << '\n'; // grąžina 'r'
}
```

Šabloninės funkcijos (1)

— Šabloninės funkcijos (angl. **template functions**) turi formą:

```
template<class parametro-tipas [, class parametro-tipas]... >
return-tipas funkcijos-vardas (parametrų-sąrašas)
```

— Kiekvienas iš šių parametro-tipas nebūtinai turi būti panaudotas funkcijos viduje.

— Tačiau kiekvienas iš parametro-tipas būtinai turi būti panaudotas funkcijos parametrų-sąrašas, kad kompiliatorius galėtų nustatyti kiekvieno iš parametrų tikruosius **tipus**.

Šabloninės funkcijos (2)

— Jeigu tipas(-i) nepasirodo parameterų-sąrašas, tuomet tas tipas(-i) **privalo** būti perduotas po funkcijos vardo (norint negauti error'o), kad kompiliatorius jį(-uos) gebėtų nustatyti:

```
template<class T> T noriuNulio() {
    return 0;
}
```

— Čia turime šabloninę funkciją noriuNulio() su vienu šabloniniu parametro-tipu T, kuris naudojamas return tipo nustatymui. Naudojant šią funkciją, mes privalome pateikti grąžinamo kintamojo tipą tiksliai (angl. **explicitly**) po funkcijos vardo bet prieš parametrų sąrašo skliaustus:

```
double x1 = noriuNulio();          // Ką gausime?
double x2 = noriuNulio<double>(); // Ką dabar gausime?
```

Šabloninės funkcijos (3)

— Raktažodis typename taip pat privalo būti naudojamas deklaracijose, kuriose apibrėžiamas kintamojo tipos, priklausantis nuo šabloninių parametrų tipų:

```
typename T::size_type vardas;
```

deklaruoja vardas kintamąjį, kurio tipas yra size_type ir jis turi būti apibrėžtas kaip kintamųjų tipas viduje T.

Šabloninės klasės (1)

— Šabloninės klasės kuriamos pagal tą pačią logiką, kaip ir šabloninės funkcijos:

```
template <class parametro-tipas [, class parametro-tipas]... >
class klases-pavadinimas { ... } ;
```

— sukuriame šabloninė klasė klases-pavadinimas, kurios kintamųjų tipai priklauso nuo **parametro-tipas** reikšmių.

Šabloninės klasės (2)

```
// Klasės viduje, šabloninei klasei nebūtina nurodyti jos patametro-tipo:
template <class T>
class Test {
public:
    Test& operator=(const Test&); // Nereikia: Test<T>
};
```

— Tačiau už klasės ribų klasės vardas turi būti susietas su parametro tipu:

```
template <class T>
Test<T>& Test<T>::operator=(const Test&) { ... }
```

— Tikruosius vardus vartotojai nurodo kurdami objektus: Test<int> sukuriame Test versiją, kurios **parametro-tipas** yra int.

Iteratoriai (1)

— Vienas iš didžiausių C++ standartinės bibliotekos bendrinio dizaino laimėjimų yra tai, kad realizuoti algoritmai nepriklauso nuo duomenų tipų, panaudojant **iteratorius**!

— Dar daugiau, algoritmai realizuoti atsižvelgiant į tai, kokios operacijos gali būti atliekamos su tam tikro tipo iteratoriais.

— Pvz. vieniems algoritmams reikia tikrinti elementus visada tik paeiliui, todėl jiems nereikalingi iteratoriai, kurie užtikrintų elementų pasiekiamumą bet kuria tvarka.

Iteratoriai (2)

- C++ apibrėžia 5-as iteratorių kategorijas:
  - **Input iterator**'iai: Vienkryptis nuoseklus/iteracinis elementų pasiekiamumas, tik **input** (elementų skaitymas)
  - **Output iterator**'iai: Vienkryptis nuoseklus/iteracinis elementų pasiekiamumas, tik **output** (elementų įrašymas)
  - **Forward iterator**'iai: Vienkryptis nuoseklus/iteracinis elementų pasiekiamumas, **input** ir **output**
  - **Bidirectional iterator**'iai: Dvikryptis nuoseklus/iteracinis elementų pasiekimas, **input** ir **output**
  - **Random-access iterator**'iai: Efektyvus elementų pasiekiamumas bet kokia tvarka, **input** ir **output**

Iteratoriai (3)

Input iteratoriaus pvz.

Tarp C++ bibliotekos funkcijų, elementus nuosekliai pasiekia pvz. find():

```
template <class  In, class X>
In find(In begin, In end, const X& x) {
    while (begin != end && *begin != x)
        ++begin; // pasiekiame elementus nuosekliai
    return begin;
}
```

— Kai kreipemės `find(begin, end, x)`, rezultatas yra pirmasis iteratorius it intervale [begin, end) toks, kad *it == x, arba end jeigu toks iteratorius neegzistuoja.

— **Input iteratoriai** turi palaikyti (gali ir daugiau) šias operacijas: ++, ==, !=, *, ->

Iteratoriai (4)

Output iteratoriaus pvz.

— Įvesties (**input**) iteratoriai gali būti naudojami tik sekos elementams skaityti.

— Akivaizdu, kad yra kontekstų, kuriuose norėtume naudoti iteratorius ir įrašyti sekos elementus. Panagrinėkime copy funkciją:

```
template<class In, class Out>
Out copy(In begin, In end, Out dest) { // pirmi 2 iter. iš kur kopijuoti, 3-as į kur
    while (begin != end)
        *dest++ = *begin++;
    return dest;
}
```

— Reikalavimai kaip **Input iteratoriams** + "write-once" reikalavimas.

Klausimai?!