C++

12. HÉT – TEMPLATE-EKRŐL HALADÓKNAK

A JEGYZETET KÉSZÍTETTE:

CSAPÓ ÁDÁM BALÁZS ÉS ŐSZ OLIVÉR

Ismétlés: template

Egy template-ből (sablonból) legenerálható egy osztály, struktúra vagy függvény definíciója

- A template argumentumok behelyettesítődnek a template-be
  - Lehet default értéket adni nekik
- A fordító generálja le a definíciót, amikor használatra kerül

Template specializáció

- Egyes template argumentumokhoz meg lehet adni más definíciót
- Osztályoknál és struktúráknál lehetséges részleges specializáció is

```
template <typename Key, typename Value>
class KeyValuePair {};

template <>
class KeyValuePair<int, std::string> {};

template <typename Key>
class KeyValuePair<Key, std::string> {};
```

Template metaprogramming

Mivel a template-ek fordításkor kerülnek legenerálásra, a constexpr és consteval megjelenése előtt a template-ek segítségével lehetett fordítási időben kiértékelődő függvényeket írni.

```
template <unsigned N>
struct factorial {
    static unsigned value = N * factorial<N - 1>::value;
};

template <>
struct factorial<0> {
    static unsigned value = 1;
};
```

Néhány újítás

Template alias (C++11)

Már volt róla szó a constrexpr előadáson

```
template<typename T> using StrMap = std::unordered_map<T, std::string>;
```

Variable template (C++14)

Egy paraméterben adott típusú változót definiál, pl. a számunkra szükséges pontosságú számtípusból

```
template<typename T>
constexpr T pi = T(3.14159265358979323846264383);
float fpi = pi<float>;
```

Variadic template (C++11)

Variadic függvények mintájára: tetszőleges számú argumentum megadható

Variadic function

C nyelvből átvett működés

A paraméterek típusa és száma nem kerül ellenőrzésre, ezért jobb inkább variadic template-et használni

```
int function(int mandatory, int optional = 3, ...);
```

A . . . jelzi, hogy további paraméterek adhatók meg

- A paraméterlista végén kell lennie
- Legalább 1 kötelező vagy opcionális paraméternek kell lennie előtte

A további paraméterek kiértékeléséhez makrók vannak definiálva az stdarg.h-ban (C++-ban <cstdarg>)

- **va_list**: egy típus a paraméterlista deklarálásához (C++-ban std::va_list)
- **va_start(args, lastarg)**: elindítja a bejárást a ... előtti utolsó paramétertől
- **va_arg(args, argtype)**: lekérdezi a következő paramétert a megadott típusú értékként
- **va_end(args)**: paraméterlista bejárása utáni takarítás
- **va_copy(dest, src)**: lemásolja a paraméterlistát (ha több iterátorra van szükségünk)

Variadic function példa

```
#include <cstdarg>

double average(int count, ...) {
    double sum = 0;
    std::va_list args;
    va_start(args, count);
    for (int i = 0; i < count; ++i) {
        double num = va_arg(args, double);
        sum += num;
    }
    va_end(args);
    return sum / count;
}
```

Variadic template

Tetszőleges számú template argumentum (akár 0)

Osztály template: `template<typename... Values> class tuple;`

Függvény template (parameter pack kiértékelése rekurzióval):

```
template<typename T>
std::string join(const std::string&, const T& lastparam) {
    return std::to_string(lastparam);
}

template<typename T, typename... Params>
std::string join(const std::string& sep, const T& first, Params... p) {
    return std::to_string(first) + sep + join(sep, p...);
}

int main() {
    std::cout << join(", ", 1, 2, 3.14) << '\n';
}
```

sizeof… operátor

A parameter pack mérete (hossza) lekérdezhető a sizeof... operátorral

```
template<typename T>
double avg(T number) {
    return number;
}

template<typename T, typename... Numbers>
double avg(T number, Numbers... rest) {
    int count = 1 + sizeof...(Numbers);
    double sumOfRest = avg(rest...) * (count - 1);
    return (number + sumOfRest) / count;
}

int main() {
    std::cout << avg(1, 2, 3, 4) << '\n';
}
```

Parameter pack bejárása inicializáló listával

A kapcsos zárójeles inicializáló listának megadható egy parameter pack, így rekurzió nélkül is bejárhatók

```
template<typename T, typename... Numbers>
double avg(Numbers... args) {
    std::vector<T> numbers = { args... };  // all must be convertible to T
    T sum = 0;
    for (auto x : numbers)
        sum += x;
    return (double)sum / sizeof...(Numbers);  // or / numbers.size()
}

int main() {
    std::cout << avg<int>(1, 2, 3, 4) << '\n';
}
```

SFINAE

Substitution Failure Is Not An Error

Több template definíció is alkalmas lehet a behelyettesítésre

- Ha egy template-be helyettesítés nem hajtható végre, nem áll le hibával a fordító, hanem kipróbálja a következő lehetséges template-et
- A legjobb egyezéstől a legáltalánosabbig halad, és az első helyes behelyettesítést fogadja el
- Ha egyik helyettesítés sem lehetséges, akkor áll le hibával

Ezt a funkciót kihasználhatjuk arra, hogy egy típusról megállapítsunk bizonyos tulajdonságokat, ez a type introspection

- Ennek tudatában más definíciót adhatunk egy template részeinek, anélkül, hogy specializálnunk kellene a template-et konkrét típusokra
  - A C++20-ban megjelent concepts további támogatást nyújt a template argumentumok tulajdonságainak kezelésére, de ezt most nem tárgyaljuk

SFINAE type introspection

```cpp
template<typename T>
class has_iter {
    typedef char yes[1];  // type of size 1
    typedef char no[2];   // type of size 2

    template<typename C>
    static yes& test(typename C::iterator*);  // typename needed!

    template<typename>
    static no& test(...);
public:
    static const bool value = sizeof(test<T>(nullptr)) == sizeof(yes);
};

int main() {
    std::cout << has_iter<std::vector<int>>::value << '\n';
    std::cout << has_iter<int>::value << '\n';
}
```

type_traits

C++11-től kezdődően jelent meg a <type_traits> header, majd a későbbi verziók tovább bővítették

Egyszerűbbé teszi a type instrospectiont

```
#include <type_traits>

template <typename T, typename = void>
struct has_iter : std::false_type {};

template <typename T>
struct has_iter<T, std::void_t<typename T::iterator>> : std::true_type {};

int main() {
    std::cout << has_iter<std::vector<int>>::value << '\n';
    std::cout << has_iter<int>::value << '\n';
}
```

std::is_detected

C++17 óta ezzel még egyszerűbb, de MSVC nem támogatja, GCC-ben experimental

```cpp
#include <iostream>
#include <vector>

#include <experimental/type_traits>

template <typename T>
using has_iter_t = typename T::iterator;  // template alias

int main() {
    std::cout << std::experimental::is_detected<has_iter_t, std::vector<int>>::value << '\n';
    std::cout << std::experimental::is_detected<has_iter_t, int>::value << '\n';
}
```

std::enable_if és std::conditional

```
template< bool B, class T = void >
struct enable_if;
```

Egy struktúra template, aminek első argumentuma egy logikai érték, ami ha igaz, akkor az `enable_if::type` belső típus a T típusra van definiálva, különben definiálatlan

- Az `std::enable_if_t<B,T>` egy alias az `std::enable_if_t<B,T>::type` -ra

```
template< bool B, class T, class F >
struct conditional;
```

Hasonló, de hamis B esetén a belső típus F-re lesz definiálva

```cpp
using namespace std;
struct Joiner {
    template<typename StringLike,
        enable_if_t<is_convertible<StringLike, string>::value, bool> = true> // substitution failure if StringLike is not convertible to string
    // string converter version
    static string join(const string&, const StringLike& lastparam) {
        return string(lastparam);
    }

    template<typename T,
        enable_if_t<!is_convertible<T, string>::value, bool> = true> // substitution failure if T is convertible to string
    // to_string version
    static string join(const string&, const T& lastparam) {
        return to_string(lastparam);
    }

    template<typename StringLike, typename... Params,
        enable_if_t<is_convertible<StringLike, string>::value, bool> = true>
    // string converter version
    static string join(const string& sep, const StringLike& first, Params... p) {
        return string(first) + sep + join(sep, p...);
    }

    template<typename T, typename... Params,
        enable_if_t<!is_convertible<T, string>::value, bool> = true>
    // to_string version
    static string join(const string& sep, const T& first, Params... p) {
        return to_string(first) + sep + join(sep, p...);
    }
};

int main() {
    cout << Joiner::join(", ", 1, "two", 3, "four") << '\n';
}
```

CRTP

Curiously Recurring Template Pattern

Leszármaztatunk egy osztály template-ből, úgy, hogy a leszármaztatott osztály lesz a template argumentum

```
template <class T>
class Base
{
    // methods within Base can cast the `this` pointer to T*, to access members of Derived
};
class Derived : public Base<Derived>
{
    // ...
};
```

```
template<typename specific_animal>
struct animal {
    void who() { static_cast<specific_animal*>(this)->who(); }
};

struct dog : animal<dog> {
    void who() { std::cout << "dog\n"; }
};

struct cat : animal<cat> {
    void who() { std::cout << "cat\n"; }
};

template<typename specific_animal>
void who_am_i(animal<specific_animal>& animal) {
    animal.who();
}

int main() {
    cat c;
    who_am_i(c);
    dog d;
    who_am_i(d);
}
```