INFOTRAINING++

profesjonalne szkolenia IT

Zaawansowane programowanie w języku C++

Wydanie 2.0

Infotraining

2019

Spis treści

1 | Semantyka przenoszenia (*Move semantics*)

1.1 Motywacja dla semantyki przenoszenia

- Optymalizacja wydajności:
  - możliwość rozpoznania kiedy mamy do czynienia z obiektem tymczasowym (*temporary object*)
  - możliwość wskazania, że obiekt nie będzie dalej używany - jego czas życia wygasa (*expiring object*)
- Możliwość implementacji obiektów, które nie powinny być kopiowane, ale umożliwiają transfer prawa własności do zasobu:
  - `auto_ptr<T>` w C++98 symulował semantykę przenoszenia za pomocą konstruktora kopiującego i operatora przypisania
  - obiekty kontrolujące zasoby systemowe, które nie mogą być łatwo kopiowane - wątki, pliki, strumienie, itp.

```
void create_and_insert(vector<string>& coll)
{
    string str = "text";

    coll.push_back(str); // insert a copy of str
                         // str is used later

    coll.push_back(str + str); // insert a copy of temporary value
                               // unnecessary copy in C++98

    coll.push_back("text"); // insert a copy of temporary value
                            // unnecessary copy in C++98

    coll.push_back(str);  // insert a copy of str
                          // unnecessary copy in C++98

    // str is no longer used
}
```

1.2 lvalues i rvalues

Aby umożliwić implementację semantyki przenoszenia C++11 wprowadza podział obiektów na:

- **lvalue**
  - obiekt posiada nazwę
  - można pobrać adres obiektu
- **rvalue**
  - nie można pobrać adresu
  - zwykle nienazwany obiekt tymczasowy (np. obiekt zwrócony z funkcji)
  - z obiektów rvalue możemy transferować stan pozostawiając je w poprawnym, ale nieokreślonym stanie

Przykłady:

```
double dx;
double* ptr; // dx and ptr are lvalues

std::string foo(std::string str); // foo and str are lvalues

// foo's return is rvalue
foo("Hello"); // temp string created for call is rvalue

std::vector<int> vec; // vec is lvalue

vi[5] = 0; // vi[5] is lvalue
```

Operacja przenoszenia, która wiąże się ze zmianą stanu, jest niebezpieczna dla obiektów l-value, ponieważ obiekt może zostać użyty po wykonaniu takiej operacji.

Operacje przenoszenia są bezpieczne dla obiektów rvalue.

1.3 Referencje rvalue - *rvalue references*

C++11 wprowadza referencje do r-value - **rvalue references**, które zachowują się podobnie jak klasyczne referencje z C++98 (zwane w C++11 **l-value references**).

- składnia: T&&
- muszą zostać zainicjowane i nie mogą zmienić odniesienia
- służą do identyfikacji operacji, które implementują przenoszenie

Wprowadzenie referencji do rvalue rozszerza reguły wiązania referencji:

- Tak jak w C++98:
  - **l-values** mogą być wiązane do **l-value references**

  - **r-values** mogą być wiązanie do **const l-value references**

- W C++11:
  - **r-values** mogą być wiązane do **r-value references**
  - **l-values** nie mogą być wiązane do **r-value references**

Ważne: Stałe obiekty l-value lub rvalue mogą być wiązane tylko z referencjami do obiektów const (`const T&&` są poprawne składniowo, ale nie mają sensu).

1.4 Implementacja semantyki przenoszenia

Używając **rvalue references** możemy zaimplementować semantykę przenoszenia.

```
template <typename T>
class vector
{
public:
    void push_back(const T& item);  // inserts a copy of item

    void push_back(T&& item);  // moves item into container
};

// ...

void create_and_insert(vector<string>& coll)
{
    string str = "text";

    coll.push_back(str); // insert a copy of str
                         // str is used later

    coll.push_back(str + str); // rvalue binds to push_back(string&&)
                               // temp is moved into container

    coll.push_back("text"); // rvalue binds to push_back(string&&)
                            // tries to move temporary object into container

    coll.push_back(std::move(str));  // tries to move str object into container

    // str is no longer used
}
```

Innym przykładem mało wydajnej implementacji z wykorzystaniem kopiowania jest implementacja `swap` w C++98:

```
template <typename T>
void swap(T& a, T& b)
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
{
    T temp = a;  // copy a to temp
    a = b; // copy b to a
    b = temp; // copy temp to b
} // destroy temp
```

Funkcja `swap()` może zostać wydajniej zaimplementowana w C++11 z wykorzystaniem semantyki przenoszenia - zamiast kopiować przenosimy wewnętrzny stan obiektów (np. wskaźniki do zasobów):

```
#include <utility>

template <typename T>
void swap(T& a, T& b)
{
    T temp {std::move(a)};  // tries to move a to temp
    a = std::move(b); // tries to move b to a
    b = std::move(temp); // tries to move temp to b
} // destroy temp
```

1.5 Semantyka przenoszenia w klasach

Aby zaimplementować semantykę przenoszenia dla klasy należy zapewnić jej:

- konstruktor przenoszący - przyjmujący jako argument **rvalue reference**
- przenoszący operator przypisania - przyjmujący jako argument **rvalue reference**

Ważne: Konstruktor przenoszący i przenoszący operator przypisania są nowymi specjalnymi funkcjami składowymi klas w C++11.

1.5.1 Funkcje specjalne klas w C++11

W C++11 istnieje sześć specjalnych funkcji składowych klasy:

- konstruktor domyślny - `X();`
- destruktor - `~X();`
- konstruktor kopiujący - `X(const X&);`
- kopiujący operator przypisania - `X& operator=(const X&);`
- konstruktor przenoszący - `X(X&&);`
- przenoszący operator przypisania - `X& operator=(X&&);`

Specjalne funkcje mogą być:

- nie zadeklarowane - **not declared**

- niejawnie zadeklarowane – **implicitly declared**
- zadeklarowane przez użytkownika – **user declared**

Specjalne funkcje zdefiniowane jako `= default` są traktowane jako **user declared**.

1.5.2 Domyślna implementacja semantyki przenoszenia w klasach

Klasy domyślnie implementują semantykę przenoszenia.

- Domyślny konstruktor przenoszący przenosi każdą składową klasy.
- Domyślny przenoszący operator przypisania deleguje semantykę przenoszenia do każdej składowej klasy

Konceptualny kod domyślnego konstruktora przenoszącego i przenoszącego operatora przypisania:

```
class X : public Base
{
    Member m_;

    X(X&& x) : Base(static_cast<Base&&>(x)), m_(static_cast<Member&&>(x.m_))
    {}

    X& operator=(X&& x)
    {
        Base::operator=(static_cast<Base&&>(x));
        m_ = static_cast<X&&>(x.m_);

        return *this;
    }
};
```

Przenoszące funkcje specjalne implementowane przez użytkownika:

```
class X : public Base
{
    Member m_;

    X(X&& x) : Base(std::move(x)), m_(std::move(x.m_))
    {
        x.set_to_resourceless_state();
    }

    X& operator=(X&& x)
    {
        Base::operator=(std::move(x));
        m_ = std::move(x.m_);
        x.set_to_resourceless_state();

        return *this;
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    }
};
```

Jeśli klasa nie zapewnia prawidłowej semantyki przenoszenia - w rezultacie wykonania operacji `move()` odbywa się kopiowanie.

1.5.3 Reguła `=default`

Jeżeli jedna z poniższych funkcji specjalnych klasy jest **user declared**

- konstruktor kopiujący
- kopiujący operator przypisania
- destruktor
- jedna z przenoszących funkcji specjalnych

specjalne funkcje przenoszące nie są generowane przez kompilator i operacja przenoszenia jest implementowana poprzez kopiowanie elementu (*fallback to copy*).

Klasa:

```
class Gadget // default copy and move semantics enabled
{

};
```

nie jest równoważna klasie:

```
class Gadget // default move semantics disabled (copy is still allowed)
{
    ~Gadget() = default;
};
```

Aby umożliwić przenoszenie należy zdefiniować (najlepiej) wszystkie funkcje specjalne.

```
class Gadget
{
    Gadget(const Gadget&) = default;
    Gadget& operator=(const Gadget&) = default;
    Gadget(Gadget&&) = default;
    Gadget& operator=(Gadget&&) = default;
    ~Gadget() = default;
};
```

1.5.4 Reguła „Rule of Five”

Jeśli w klasie jest konieczna implementacja jednej z poniższych specjalnych funkcji składowych:

- konstruktora kopiującego

- konstruktora przenoszącego
- kopiującego operatora przypisania
- przenoszącego operatora przypisania
- destruktora

najprawdopodobniej **należy zaimplementować wszystkie**.

Ta regułą stosuje się również do funkcji specjalnych zdefiniowanych jako `default`.

1.6 Implementacja funkcji `std::move()`

Implementacja funkcji `std::move()` dokonuje rzutowania na **rvalue reference** - `T&&`.

```
template <typename T>
typename std::remove_reference<T>::type&& move(T&& obj) noexcept
{
    using ReturnType = std::remove_reference<T>::type&&;
    return static_cast<ReturnType>(obj);
}
```

1.7 Reference collapsing

W procesie tworzenia instancji szablonu następuje często zwijanie referencji (tzw. **reference collapsing**)

Jeśli mamy szablon:

```
template <typename T>
void f(T& item)
{
    // ...
}
```

Jeśli przekażemy jako parametr szablonu `int&`, to tworzona początkowo instancja szablonu wygląda następująco:

```
void f(int& & item);
```

Reguła zwijania referencji powoduje, że `int& &` -> `int&`. W rezultacie instancja szablonu wygląda tak:

```
void f(int& item);
```

W C++11 obowiązują następujące reguły **reference collapsing**

T& &	-> T&
T&& &	-> T&
T& &&	-> T&
T&& &&	-> T&&

1.7.1 Mechanizm dedukcji typów w szablonach

Dla szablonu

```
template <typename T>
void f(T&&)  // non-const rvalue reference
{
    // ...
}
```

typ `T` jest dedukowany w zależności od tego co zostanie przekazane jako argument wywołania funkcji:

- jeśli przekazany zostanie obiekt *l-value* - to parametr szablonu jest referencją l-value - *T&*
- jeśli przekazany zostanie obiekt *r-value* - to parametr szablonu nie jest referencją - *T*

W połączeniu z regułami zwijania referencji:

```
string str;

f(str);  // l-value : f<string&>(string& &&) - > f<string&>(string&)

f(string("Hello")); // rvalue : f<string>(string&&)
```

1.8 Forwarding reference

Referencja rvalue `T&&` użyta w szablonie ma szczególne zastosowanie:

- dla argumentów l-value T&& -> T& - wiąże się z wartościami l-value
- dla argumentów r-value T&& pozostaje T&& - wiąże się z wartościami rvalue

Ponieważ mechanizm dedukcji typów w `auto` jest taki sam jak w szablonach:

```
string get_line(istream& in);

auto&& line = get_line(cin);  // type of line: string&&

string name = "Ola";

auto&& alias = name; // type of alias: string&
```

1.9 Perfect Forwarding

Przeciążanie funkcji w celu optymalizacji wydajności z wykorzystaniem semantyki przenoszenia i referencji rvalue może prowadzić do nadmiernego rozrostu interfejsów:

```
class Gadget;

void have_fun(const Gadget&);
void have_fun(Gadget&); // copy semantics
void have_fun(Gadget&&); // move semantics

void use(const Gadget& g)
{
    have_fun(g); // calls have_fun(const Gadget&)
}

void use(Gadget& g)
{
    have_fun(g); // calls have_fun(Gadget&)
}

void use(Gadget&& g)
{
    have_fun(std::move(g)); // calls have_fun(Gadget&&)
}

int main()
{
    const Gadget cg;
    Gadget g;

    use(cg);  // calls use(const Gadget&) then calls have_fun(const Gadget&)
    use(g);   // calls use(Gadget&) then calls have_fun(Gadget&)
    use(Gadget()); // calls use(Gadget&&) then calls have_fun(Gadget&&)
}
```

Rozwiązaniem jest szablonowa funkcja przyjmująca jako parametr wywołania `T&&` (**forwarding reference**) i przekazująca argument do następnej funkcji z wykorzystaniem funkcji `std::forward()`.

```
class Gadget;

void have_fun(const Gadget&);
void have_fun(Gadget&); // copy semantics
void have_fun(Gadget&&); // move semantics

template <typename Gadget>
void use(Gadget&& g)
{
    have_fun(std::forward<Gadget>(g)); // forwards original type to have_fun()
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
                                    //- rvalue reference if T is Gadget
                                    // without forward<> only have_fun(const Gadget&)
                                    // and have_fun(Gadget&) would get called
}

int main()
{
    const Gadget cg;
    Gadget g;

    use(cg);  // calls use(const Gadget&) then calls have_fun(const Gadget&)
    use(g);   // calls use(Gadget&) then calls have_fun(Gadget&)
    use(Gadget()); // calls use(Gadget&&) then calls have_fun(Gadget&&)
}
```

Funkcja `std::forward()` działa jak warunkowe rzutowanie na `T&&`, gdy dedukowanym parametrem szablonu jest typ nie będący referencją.

1.10 Słowo kluczowe noexcept

Słowo kluczowe `noexcept` może być użyte

- w deklaracji funkcji - aby określić, że funkcja nie może rzucić wyjątku

```
template <typename T>
class vector
{
public:
    iterator begin() noexcept; // it can't throw an exception
};
```

- jako operator - który zwraca `true` jeśli podane jako parametr wyrażenie nie może rzucić wyjątku

```
void swap(Type& x, Type& y) noexcept(noexcept(x.swap(y)))
{
    x.swap(y);
}
```

- Zastępuje specyfikację rzucanych wyjątków z funkcji w C++03
  - nie ma narzutu w czasie wykonania programu
  - jeśli funkcja zadeklarowana jako `noexcept` rzuci wyjątek wywoływana jest funkcja `std::terminate()`
- Umożliwia optymalizację wydajności - np. implementacja `push_back()` w wektorze

2 | Inteligentne wskaźniki - *Smart Pointers*

2.1 Mechanizm RAII

2.1.1 Mechanizm wyjątków a zasoby

Używanie natywnych wskaźników (*raw pointers*) do zarządzania zasobami może powodować wycieki zasobów.

```
void use_resource()
{
    Resource* rsc = new Resource(); // resource acquisition

    rsc->use(); // use() may throw

    may_throw();

    delete rsc; // resource release
}
```

W celu zabezpieczenia przed wyciekiem zasobu możemy użyć konstrukcji `try-catch`:

```
// Nieudolna poprawa
void use_resource()
{
    Resource* rsc = nullptr;
    try
    {
        rsc = new Resource();

        rsc->use(); // Kod, który używa rsc i może rzucić wyjątkiem

        may_throw();
    }
    catch(...) //Przechwytuje wszystkie wyjątki
    {
        delete rsc;
        throw;
    }
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    delete rsc;
}
```

Niestety w rezultacie kod staje się mało czytelny i występuje w nim duplikacja zwalniania zasobu – `delete rsc`.

2.1.2 Zarządzanie zasobami RAII

Resource Acquisition Is Initialization (RAII) - zdobywanie zasobów jest inicjowaniem. Technika łączy przejęcie i zwolnienie zasobu z inicjalizacją zmiennych lokalnych i ich automatyczną destrukcją. Pozyskanie zasobu jest połączone z konstrukcją, a zwolnienie z automatyczną destrukcją obiektu. Ponieważ wywołanie destruktora dla obiektu lokalnego następuje automatycznie, jest zagwarantowane, że zasób zostanie zwolniony od razu, gdy skończy się czas życia zmiennej. Mechanizm ten działa również przy wystąpieniu wyjątku. RAII jest kluczową koncepcją przy pisaniu kodu odpornego na wycieki zasobów.

```
std::mutex mtx;
int state = 0;

void update_state(); // updates state

void unsafe_code()
{
    mtx.lock();

    update_state(); // may throw

    mtx.unlock();
}

class lock_guard
{
    std::mutex& mtx_;
public:
    lock_guard(std::mutex& mtx) : mtx_{mtx}
    {
        mtx_.lock();
    }

    ~lock_guard()
    {
        mtx_.unlock();
    }

    lock_guard(const lock_guard&) = delete;
    lock_guard& operator=(const lock_guard&) = delete;
};
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
// using RAII object
void safe_code()
{
    lock_guard<mutex> lk{mtx};

    update_state(); // may throw
}
```

2.1.3 Kopiowanie obiektów RAII

Klasy implementujące RAII posiadają destruktor, dlatego należy określić sposób zachowania obiektów przy ich kopiowaniu. Możliwe są następujące strategie:

- Całkowite blokowanie kopiowania oraz transferu prawa własności
- Blokowanie kopiowania przy jednoczesnym umożliwieniu transferu prawa własności do zasobu
- Zezwolenie na kopiowanie obiektów - obiekty współdzielą prawo własności z wykorzystaniem licznika referencji

2.2 Inteligentne wskaźniki

Inteligentne wskaźniki umożliwiają wygodne zarządzanie obiektami, które zostały zaalokowane na stercie za pomocą operatora `new`. Przejmują odpowiedzialność za wywołanie destruktora dla zarządzanego obiektu oraz zwolnienie pamięci - poprzez wywołanie operatora `delete`. Inteligentne wskaźniki w C++11 umożliwiają reprezentację prawa własności do zaalokowanego zasobu. Przeciążając operatory `operator*` oraz `operator->` umożliwiają korzystanie z nich w taki sam sposób, jak wskaźników natywnych.

Dostępne implementacje inteligentnych wskaźników w bibliotece standardowej C++ oraz w bibliotece Boost.

Klasa wskaźnika	Kopiowanie	Transfer prawa własności	Licznik referencji
`std::auto_ptr<T>`	o	o	
`std::unique_ptr<T>`		o	
`std::shared_ptr<T>`	o	o	wewnętrzny
`boost::scoped_ptr<T>`			
`boost::shared_ptr<T>`	o	o	wewnętrzny
`boost::intrusive_ptr<T>`	o		zewnętrzny

2.3 Klasa `std::unique_ptr<T>`

Plik nagłówkowy: `<memory>`

Klasa szablonowa `std::unique_ptr` służy do zapewnienia właściwego usuwania przydzielanego dynamicznie obiektu. Implementuje RAII - destruktor inteligentnego wskaźnika usuwa wskazywany obiekt. Wskaźnik `unique_ptr` nie może być ani kopiowany ani przypisywany, może być jednakże przenoszony. Przeniesienie prawa własności odbywa się zgodnie z *move semantics* w C++11 - wymaga dla referencji do l-value jawnego transferu przy pomocy funkcji `std::move()`.

```
void f()
{
    std::unique_ptr<Gadget> my_gadget {new Gadget()};
    // kod, który może wyrzucać wyjątki
    my_gadget->use();


    std::unique_ptr<Gadget> your_gadget = std::move(my_gadget); // explicit move

} // Destruktor klasy unique_ptr wywoła operator delete dla wskaźnika
  //  do kontrolowanej instancji
```

2.3.1 Semantyka przenoszenia dla `unique_ptr`

Obiekt `std::unique_ptr` nie może być kopiowany. Ale dzięki semantyce przenoszenia, może być stosowany tam, gdzie zgodne ze standardem C++03 niekopiowalne obiekty nie mogły działać:

- może być zwracany z funkcji

```
std::unique_ptr<Gadget> create_gadget(int type)
{
    auto gdgt = unique_ptr<Gadget> {new Gadget(arg)};

    return gdgt;
}

auto ptr_gadget = create_gadget(1); // implicit move
ptr_gadget->do_something();
```

W C++14 funkcję `create_gadget()` można zastąpić biblioteczną funkcją `make_unique()`, która tranferuje swoje argumenty wywołania do konstruktora alokowanego na stercie obiektu:

```
auto ptr = make_unique<Gadget>(arg); // C++14
```

- może być przekazywany (przenoszony) do funkcji jako parametr *sink*

```
void sink(unique_ptr<Gadget> gdgt)
{
    gdgt->call_method();
    // sink takes ownership - deletes the object pointed by gdgt
}
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
// ...

sink(move(ptr));   // explicitly moving into sink
sink(create_gadget(2));  // implicit move
```

- może być przechowywany w kontenerach STL (w standardzie C++11)

```
vector<unique_ptr<Gadget>> gadgets;

gadgets.emplace_back(new Gadget());
gadgets.push_back(unique_ptr<Gadget>(new Gadget())); // implicit move to the container
gadgets.push_back(create_gadget(3)); // implicit move to the container

for(const auto& g : gadgets)
    g->do_something();

gadgets.clear(); // elements are automatically destroyed
```

2.3.2 Wskaźniki klas pochodnych

Wskaźnik do klasy pochodnej może zostać przypisany do wskaźnika do klasy bazowej (*upcasting*). Daje to możliwość stosowania polimorfizmu z wykorzystaniem funkcji wirtualnych.

```
Gadget* pb = new SuperGadget();
pb->do_something();
```

Odpowiednik z użyciem `unique_ptr`

```
std::unique_ptr<Gadget> pb = std::make_unique<SuperGadget>();

//explicit conversion - hard to miss it
auto pb = std::unique_ptr<Gadget>{ std::make_unique<SuperGadget>() };
```

2.3.3 Dealokatory

Używając wskaźnika `std::unique_ptr` można zdefiniować własny dealokator, który będzie odpowiedzialny za prawidłowe zwolnienie zasobu. Umożliwia to kontrolę nad zasobami innymi niż obiekty dynamicznie alokowane na stercie lub wymagającymi specjalnej obsługi w fazie destrukcji.

Aby użyć własnego dealokatora należy podać jego typ jako drugi parametr szablonu `std::unique_ptr<T, Dealloc>` oraz przekazać instancję dealokatora jako drugi parametr konstruktora:

```
{
    std::unique_ptr<FILE, int(*)(FILE*)> file{fopen("test.txt"), &fclose};

    read(file.get());
} // fclose() is called for an opened file
```

Ważne: Dealokator dla `unique_ptr` wywoływany jest tylko, jeśli wewnętrzny wskaźnik jest rózny od `nullptr`!

2.3.4 Idiom PIMPL

Wskaźnik `std::unique_ptr` świetnie nadaje się do stosowania tam, gdzie wcześniej stosowane były wskaźniki zwykłe albo obiekty typu `std::auto_ptr` (obecnie mający status *deprecated*), np. do implementacji idiomu PIMPL

PIMPL – Private Implementation:

- minimalizuje zależności na etapie kompilacji
- separuje interfejs od implementacji
- ukrywa implementację

Plik `blob.hpp`:

```
class Blob
{
public:
    // interfejs klasy
    /* . . . */
    ~Blob(); // must be only declared
private:
    class Impl; // deklaracja zapowiadająca

    // implementacja jest ukryta
    std::unique_ptr<Impl> pimpl_; // wskaźnik do implementacji
};
```

Plik `blob.cpp`:

```
class Blob::Impl
{
    // wszystkie składowe prywatne (pola i metody)
    // zmiany implementacji nie wymagają rekompilacji klas obiektów korzystających
    // z instancji Blob
};
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
Blob::Blob() : pimpl_ {new Impl()}
{
    // ustawienie stanu obiektu implementującego
}

Blob::~Blob() = default; // important! after defintion of Blob::Impl()
```

2.3.5 Zastosowanie `std::unique_ptr<T>`

Wskaźniki `std::unique_ptr` należy stosować tam, gdzie:

- w zasięgu obarczonym ryzykiem zgłoszenia wyjątku występuje wskaźnik
- funkcja ma kilka ścieżek wykonania i kilka punktów powrotu
- istnieje tylko jeden obiekt zarządzający czasem życia alokowanego obiektu
- ważna jest odporność na wyjątki

2.4 Klasa `std::unique_ptr<T[]>`

Szablon `std::unique_ptr` stanowi również lepszą alternatywę dla klasycznych tablic przydzielanych dynamicznie. Przejmuje zarządzanie czasem życia takich tablic. Oferuje przeciążony operator indeksowania (`operator[]`) umożliwiający stosowanie naturalnej składni odwołań do elementów tablicy. Destruktor wykorzystuje operator `delete []` aby automatycznie usunąć wskazywaną tablicę.

```
try
{
    std::unique_ptr<Gadget[]> many_gadgets {new Gadget[10]};

    for(int i = 0; i < 10; ++i)
    {
        many_gadgets[i].do_stuff();
        unsafe_use(many_gadgets[i]);
    }
}
catch (...)
{
    std::cout << "Obsługa wyjątku" << std::endl;
}
```

2.5 Wskaźniki ze zliczaniem odniesień

Inteligentne wskaźniki ze zliczaniem odniesień eliminują konieczność kodowania skomplikowanej logiki sterującej czasem życia obiektów współużytkowanych przez

pewną liczbę innych obiektów.

Można podzielić je na:

- **ingerencyjne** (*intrusive*) – wymagają od klas obiektów zarządzanych udostępnienia specjalnych metod lub składowych, za pomocą których realizowane jest zliczanie odwołań
- **nieingerencyjne** (*non-intrusive*) – nie wymagają niczego od obiektu zarządzanego

---

Ważne: Wskaźniki ze zliczaniem odniesień mogą być przechowywane w kontenerach standardowych (np. `vector`, `list`, itp.)

2.6 Klasa `std::shared_ptr`

Plik nagłówkowy: `<memory>`

`std::shared_ptr` jest szablonem nieingerencyjnego wskaźnika zliczającego odniesienia do wskazywanych obiektów.

Działanie:

- konstruktor tworzy licznik odniesień i inicjuje go wartością 1
- konstruktor kopiujący lub operator przypisania inkrementują licznik odniesień
- destruktor zmniejsza licznik odniesień, jeżeli ma on wartość 0, to usuwa obiekt wywołując domyślnie operator `delete`

Przykład 1

```
#include <memory>

using namespace std;

class Gadget { /* implementacja */ };

map<string, shared_ptr<Gadget>> gadgets;

void foo()
{
    shared_ptr<Gadget> p1 {new Gadget(1)};  // reference counter = 1
    {
        shared_ptr<Gadget> p2 = p1; // copying of shared_ptr
        // reference counter == 2

        gadgets.insert(make_pair("mp3 player", p2)); // copying shared_ptr to a std container
        // reference counter == 3

        p2->use();
    }  // destruction of p2 decrements reference counter = 2
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
}  // destruction of p1 decrements reference counter = 1

// ...

gadgets.clear(); // reference counter = 0 - gadget is removed
```

2.6.1 Przydatne metody z interfejsu `shared_ptr<T>`

`T* get() const`
: zwraca przechowywany wskaźnik.

`void reset()`
: zwalnia prawo własności do zarządzanego obiektu.

`template <typename Y*> void reset(Y* ptr)`
: zamienia obiekt zarządzany na obiekt wskazywany przez `ptr`.

`bool unique() const`
: zwraca true, jeśli obiekt shared_ptr, na rzecz którego nastąpiło wywołanie, jest jedynym właścicielem przechowywanego wskaźnika. W pozostałych przypadkach zwraca false

`long use_count() const`
: zwraca wartość licznika odwołań do wskaźnika przechowywanego w obiekcie shared_ptr. Przydatna w diagnostyce.

`void swap(shared_ptr<T>& other)`
: wymiana wskaźników między dwoma obiektami shared_ptr. Wymienia wskaźniki oraz liczniki odwołań.

`explicit operator bool() const`
: umożliwia konwersję do wartości logicznej, np. `if (p && p->is_valid())`

2.6.2 Fabryka `make_shared`

Używanie `shared_ptr` eliminuje konieczność stosowanie operatora `delete`, jednakże nie eliminuje użycia `new`. Można uniknąć używania operatora `new` stosując zamiast tego funkcję pomocniczą `make_shared()`, która pełni rolę fabryki wskaźników `shared_ptr`. Funkcja przekazuje swoje parametry do konstruktora obiektu kontrolowanego przez inteligentny wskaźnik (*perfect forwarding*).

```
std::shared_ptr<std::string> x = std::make_shared<std::string>("hello, world!");
std::cout << *x << std::endl;
```

Stosowanie funkcji `make_shared()` jest wydajniejsze niż konstrukcja `shared_ptr(new std::string)` ponieważ alokowany jest tylko jeden segment pamięci, w którym umieszczany jest wskazywany obiekt oraz blok kontrolny z licznikami odniesień.

2.6.3 Problem tymczasowych obiektów typu shared_ptr

Ważne: Zawsze należy używać nazwanych instancji shared_ptr!

Użycie tymczasowych zmiennych anonimowych typu shared_ptr może powodować wycieki pamięci.

Przykład:

```
void f(shared_ptr<int>, int);
int may_throw();

void ok()
{
    shared_ptr<int> p { new int(2) };
    f(p, may_throw());
}

void bad()
{
    f(shared_ptr<int> {new int(2)}, may_throw() );
}
```

Ponieważ kolejność ewaluacji argumentów funkcji nie jest określona, jest możliwa sytuacja w której wykona się new, następnie funkcja may_throw rzuci wyjątkiem. Obiekt‘‘shared_ptr‘‘ nie został w ogóle skonstruowany, nie wykona się jego destruktor, a tym samym zasób nie zostanie uprzątnięty.

Powyższy problem może zostać również rozwiązany przy użyciu funkcji make_shared().

2.6.4 Dealokatory

Niekiedy pojawia się potrzeba zastosowania wskaźnika shared_ptr do kontroli zasobu takiego typu, że jego zwolnienie nie sprowadza się do prostego wywołania operatora delete. Takie przypadki shared_ptr obsługuje przy pomocy dealokatorów użytkownika.

Dealokator jest:

- obiektem funkcyjnym odpowiedzialnym za zwolnienie zasobu, kiedy liczba referencji spadnie do zera.
- przekazywany jako drugi argument konstruktora shared_ptr.

```
class SocketCloser
{
public:
    void operator()(Socket* s)
    {
        s->close();
    }
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
};

void use_device()
{
    Socket socket;
    std::shared_ptr<Socket> safe_socket(&socket, SocketCloser());

    may_throw(); // może wylecieć wyjątek

    safe_socket->write("Data");
} // zasób Socket został bezpiecznie zwolniony za pomocą metody close()
```

2.6.5 Rzutowania między wskaźnikami `shared_ptr`

Problemy związane z rzutowaniami między wskaźnikami shared_ptr rozwiązują trzy funkcje szablonowe:

`template<class T, class U> shared_ptr<T> static_pointer_cast(shared_ptr<U> const & r)`

- Jeśli `r` jest pusty, zwraca pusty `shared_ptr<T>`
- W innym przypadku `shared_ptr<T>` przechowuje kopię `static_cast<T*>(r.get())` i współdzieli prawo własności z `r`

`template<class T, class U> shared_ptr<T> const_pointer_cast(shared_ptr<U> const &r)`

- Jeśli `r` jest pusty, zwraca pusty `shared_ptr<T>`
- W innym przypadku `shared_ptr<T>` przechowuje kopię `const_cast<T*>(r.get())` i współdzieli prawo własności z `r`

`template<class T, class U> shared_ptr<T> dynamic_pointer_cast(shared_ptr<U> const & r)`

- Jeśli `dynamic_cast<T*>(r.get())` zwraca wartość różną od `nullptr`, zwracany jest obiekt `shared_ptr<T>`, który współdzieli prawo własności do `r`
- W przeciwnym wypadku pusty `shared_ptr<T>`

2.6.6 Zależności cykliczne między wskaźnikami shared_ptr

Problemem dla wskaźników `shared_ptr` są zależności cykliczne, które powodują wycieki zasobów (pamięci).

```
struct Cyclic
{
    shared_ptr<Cyclic> me_;
    /* . . . */
};

void foo()
{
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    shared_ptr<Cyclic> cptr = make_shared<Cyclic>();
    cptr->me_ = cptr;
}  // wyciek pamięci
```

Ważne: Cykle muszą być przerywane za pomocą wskaźników `std::weak_ptr`

2.6.7 Tworzenie wskaźnika `shared_ptr` ze wskaźnika `this`

Niekiedy istnieje konieczność utworzenia inteligentnego wskaźnika `shared_ptr` ze wskaźnika `this`. Oznacza to, że obiekt danej klasy będzie zarządzany za pośrednictwem inteligentnego wskaźnika. W ogólności rozwiązaniem problemu konwersji `this` do `shared_ptr` jest użycie wskaźnika typu `std::weak_ptr`, który jest obserwatorem wskaźników `shared_ptr` (pozwala na podglądanie wskazywanego obiektu bez ingerowania w wartość licznika odwołań). Zdefiniowanie w klasie składowej typu `weak_ptr` i zainicjowanie jej wartością `this` umożliwia późniejsze pozyskiwanie wskaźników `shared_ptr`. Aby nie trzeba było za każdym razem pisać tego samego kodu, można wykorzystać dziedziczenie po pomocniczej klasie `enable_shared_from_this`.

Klasa `enable_shared_from_this<T>` umożliwia obiektowi typu `T`, który jest zarządzany przez instancję `std::shared_ptr<T>` bezpieczne tworzenie dodatkowych wskaźników typu `std::shared_ptr<T>`, które współdzielą prawo własności do zarządzanego obiektu.

Przykład nieprawidłowego generowania instancji `shared_ptr` ze wskaźnika `this`:

```
#include <memory>

void do_stuff(std::shared_ptr<A> p)
{
   // using p
}

// ...

class A
{
public:
   void call_do_stuff()
   {
      do_stuff(std::shared_ptr<A>(this));
   }
};

int main()
{
   auto p = std::make_shared<A>();
   p->call_do_stuff();
}
```

Prawdiłowe tworzenie instancji `std::shared_ptr` ze wskaźnika `this`:

```
#include <memory>

void do_stuff(std::shared_ptr<A> p)
{
   // using p
}

// ...

class A : public std::enable_shared_from_this<A>
{
public:
   void call_do_stuff()
   {
      do_stuff(shared_from_this());
   }
};

int main()
{
   auto p = std::make_shared<A>();
   p->call_do_stuff();
}
```

2.6.8 `std::shared_ptr` – podsumowanie

Wskaźniki `std::shared_ptr` można skutecznie stosować tam, gdzie:

- jest wielu użytkowników obiektu, ale nie ma jednego jawnego właściciela
- trzeba przechowywać wskaźniki w kontenerach biblioteki standardowej
- trzeba przekazywać wskaźniki do i z bibliotek, a nie ma jawnego wyrażenia transferu własności

2.7 Klasa `std::weak_ptr`

Plik nagłówkowy: `<memory>`

Najbardziej znanym problemem, związanym ze wskaźnikami opartymi na zliczaniu odniesień, są odniesienia cykliczne. Występują one w sytuacji, gdy kilka obiektów trzyma wskaźniki do siebie nawzajem, przez co licznik odniesień nie spada do zera i wskazywane obiekty nie są nigdy kasowane. Rozwiązaniem tego problemu jest zastosowanie słabych wskaźników – obiektów `std::weak_ptr`.

Wskaźnik typu `std::weak_ptr` obserwuje wskazywany obiekt, ale nie ma kontroli nad czasem jego życia i nie może zmieniać jego licznika odniesień.

Nie udostępnia operatorów dereferencji. Aby mieć dostęp do wskazywanego obiektu konieczne jest dokonanie konwersji do `std::shared_ptr`.

Gdy wskazywany obiekt został już skasowany konwersja na `std::shared_ptr` daje w wyniku:

- wskaźnik pusty - w przypadku metody `lock()`
- zgłasza wyjątek `std::bad_weak_ptr` - w przypadku konstruktora `shared_ptr<T>(const weak_ptr<T>&)`

2.7.1 Konwersja `weak_ptr` na `shared_ptr`

Przykład 1

```
weak_ptr<Gadget> weakPtr;
shared_ptr<Gadget> ptr(new Gadget());

weakPtr = ptr; // ref counter == 1

ptr.reset();  // ref counter == 0 -> gadget is destroyed

shared_ptr<Gadget> tempPtr = weakPtr.lock();

if (tempPtr)
{
    tempPtr->do_stuff();
}
```

Przykład 2

```
weak_ptr<Gadget> weakPtr;
shared_ptr<Gadget> ptr = make_shared<Gadget>();

weakPtr = ptr; // ref counter == 1

ptr.reset();  // gadget is destroyed

try
{
    shared_ptr<Gadget> anotherPtr(weakPtr);
}
catch(const bad_weak_ptr& e)
{
    cout << e.what() << endl;
}
```

2.7.2 Przechowywyanie `std::weak_ptr` w kontenerach asocjacyjnych

Aby przechować wskaźniki `std::weak_ptr` w kontenerach asocjacyjnych należy klasy `std::owner_less` jako parameteru definiującego sposób porównania wskaźników.

Klasa `std::owner_less` dostarcza implementację umożliwiającą porównanie wskaźników `std::shared_ptr` oraz `std::weak_ptr` na podstawie prawa własności, a nie wartości przechowywanych wskaźników. Dwa wskaźniki są uznane za równoważne tylko wtedy, gdy oba są puste lub oba zarządzają tym samym obiektem (współdzielą blok kontrolny).

```
std::map<std::shared_ptr<Key>, Value, std::owner_less<std::shared_ptr<Key>>> my_map;

std::set<std::weak_ptr<Key>, std::owner_less<std::weak_ptr<Key>>> my_set;
```

2.7.3 Zastosowanie `std::weak_ptr`

Wskaźniki `std::weak_ptr` stosuje się do:

- zrywania cyklicznych zależności dla `shared_ptr`
- współużytkowania zasobu bez przejmowania odpowiedzialności za zarządzanie nim
- eliminowania ryzykownych operacji na wiszących wskaźnikach

3 | Szablony

Szablony implementują koncepcję **programowania generycznego** (*generic programming*) – programowania używającego typów danych jako parametrów.

Szablony zapewniają mechanizm, za pomocą którego funkcje lub klasy mogą być implementowane dla ogólnych typów danych (nawet takich, które jeszcze nie istnieją).

- zastosowanie szablonów klas umożliwia parametryzację kontenerów ze względu na typ elementów, które są w nich przechowywane
- szablony funkcji umożliwiają implementację algorytmów działających w ogólny sposób dla różnych struktur danych

W szczególnych przypadkach implementacja szablonów klas lub funkcji może być wyspecjalizowana dla pewnych typów.

Parametry szablonów mogą być:

- typami
- stałymi typów całkowitych lub wyliczeniowych znanymi w chwili kompilacji
- szablonami

3.1 Szablony funkcji

Szablon funkcji – funkcja, której typy argumentów lub typ zwracanej wartości zostały sparametryzowane.

Składnia definicji szablonu funkcji:

```
template < comma_separated_list_of_parameters >
return_type function_name(list_of_args)
{
    //...
}
```

Przykład definicji szablonu funkcji `max()`:

```
template<typename T>
T max(T a, T b)
{
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    return b < a ? a : b;
}
```

Użycie szablonu funkcji `max()`:

```
int val1 = ::max<int>(4, 9);

double x = 4.50;
double y = 3.14;
double val2 = ::max(x, y);

std::string s1 = "mathematics";
std::string s2 = "math";
std::cout << "max(s1, s2) = " << ::max(s1, s2) << '\n';
```

3.1.1 Dedukcja typu argumentów szablonu

W momencie wywołania funkcji szablonowej `max()` następuje proces dedukcji tych typów argumentów szablonu, które nie zostały podane w ostrych nawiasach <>. Typy argumentów szablonu dedukowane są na podstawie typu argumentów funkcji.

Reguły dedukcji typów dla szablonów funkcji:

- Każdy parametr funkcji może brać udział (lub nie) w procesie dedukcji parametru szablonu
- Wszystkie dedukcje są przeprowadzane niezależnie od siebie
- Na zakończenie procesu dedukcji kompilator sprawdza, czy każdy parametr szablonu został wydedukowany i czy nie ma konfliktów między wydedukowanymi parametrami
- Każdy parametr funkcji, który bierze udział w procesie dedukcji musi ściśle pasować do typu argumentu funkcji - niejawne konwersje są zabronione

```
template<typename T, typename U>
void f(T x, U y);

template<typename T>
void g(T x, T y);

int main()
{
    f(1, 2); // void f(T, U) [T = int, U = int]
    g(1, 2); // void g(T, T) [T = int]
    g(1, 2u); // error: no matching function for call to g(int, unsigned int)
}
```

W przypadku, gdy w procesie dedukcji wykryte zostaną konflikty:

```
short f();

auto val = ::max(f(), 42); // ERROR - no matching function
```

istnieją dwa rozwiązania:

```
// #1
auto val1 = ::max(static_cast<int>(f()), 42); // OK

// #2
auto val2 = ::max<int>(f(), 42); // OK
```

3.1.2 Tworzenie instancji szablonu

Koncepcja szablonów wykracza poza zwykły model kompilacji (konsolidacji). Cały kod szablonu powinien być umieszczony w jednym pliku nagłówkowym. Dołączając następnie zawartość pliku nagłówkowego do kodu aplikacji umożliwiamy generację i kompilację kodu dla konkretnych typów.

Tworzenie **instancji szablonu** - proces, w którym na podstawie szablonu generowany jest kod, który zostanie skompilowany.

Utworzenie instancji szablonu jest możliwe tylko wtedy, gdy dla typu podanego jako argument szablonu zdefiniowane są wszystkie operacje używane przez szablon, np. operatory `<`, `==`, `!=`, wywołania konkretnych metod, itp.

Fazy kompilacji szablonu

Proces kompilacji szablonu przebiega w dwóch fazach:

1. Na etapie definicji szablonu, ale bez tworzenia jego instancji kod jest sprawdzany pod względem poprawności bez uwzględniania parametrów szablonu:
   - wykrywane są błędy składniowe
   - wykrywane jest użycie nieznanych nazw (typów, funkcji, itp.), które nie zależą od parametru szablonu
   - sprawdzane są statyczne asercje, które nie zależą od parametru szablonu
2. Podczas tworzenia instancji szablonu, kod szablonu jest jeszcze raz sprawdzany. W szczególności sprawdzane są części, które zależą od parametrów szablonu.

```
template<typename T>
void foo(T t)
{
    undeclared();   // first-phase compile-time error if undeclared() unknown
    undeclared(t);  // second-phase compile-time error if undeclared(T) unknown
    static_assert(sizeof(int) > 10, "int too small");  // always fails if sizeof(int)<=10
    static_assert(sizeof(T) > 10, "T too small");  // fails if instantiated for T with size <=10
}
```

3.1.3 Specjalizacja funkcji szablonowych

W specjalnych przypadkach istnieje możliwość zdefiniowania funkcji specjalizowanej.

Zdefiniujmy najpierw pierwszy szablon funkcji:

```
template <typename T>
bool is_greater(T a, T b)
{
    return a > b;
}
```

Jeśli wywołamy ten szablon dla literałów znakowych „abc” i „def” utworzona zostanie instancja szablonu, która porówna adresy przechowywane we wskaźnikach a nie tekst. Aby zapewnić prawidłowe porównanie c-łańcuchów musimy dostarczyć specjalizowaną wersję szablonu dla parametru `const char*`:

```
template <>
bool is_greater<const char*>(const char* a, const char* b)
{
    return strcmp(a, b) > 0;
}

is_grater(4, 5); // wywołanie funkcji szablonowej
//...
is_greater("a", "g"); // wywołanie funkcji specjalizowanej dla const char*
```

Ponieważ podawanie typu specjalizowanego w ostrych nawiasach jest redundantne można specjalizację szablonu funkcji zapisać w sposób następujący:

```
template <>
bool is_greater(const char* a, const char* b)
{
    return strcmp(a, b) > 0;
}
```

W praktyce zamiast stosować jawną specjalizację szablonu funkcji można wykorzystać przeciążenie funkcji `is_greater()`:

```
bool is_greater(const char* a, const char* b)
{
    return strcmp(a, b) > 0;
}
```

Połączenie przeciążania szablonów, specjalizacji i przeciążania funkcji

Przykład wykorzystania specjalizacji lub przeciążenia szablonu funkcji:

```
template <typename T> T sqrt(T); // basic template sqrt<T>
template <> float sqrt(float);   // specialization of template sqrt<float>
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
template <typename T> complex<T> sqrt(complex<T>);  // overloaded template sqrt for complex<T>␣
↪arguments
double sqrt(double); // overloaded function sqrt(double)

//...

void f(complex<double> z)
{
   sqrt(2);   // sqrt<int>(int)
   sqrt(2.0); // sqrt(double)
   sqrt(z);   // sqrt<double>(complex<double>)
   sqrt(3.14f); // sqrt<float>(float)
}
```

3.1.4 Przeciążanie szablonów

W programie może obok siebie istnieć mając tę samą nazwę:

- kilka szablonów funkcji – byle produkowały funkcje o odmiennych argumentach,
- funkcje, o argumentach takich, że mogłyby zostać wyprodukowane przez któryś z szablonów (funkcje specjalizowane),
- funkcje o argumentach takich, że nie mógłby ich wyprodukować żaden z istniejących szablonów (zwykłe przeładowanie).

3.1.5 Adres wygenerowanej funkcji

Możliwe jest pobranie adresu funkcji wygenerowanej na podstawie szablonu.

```
template <typename T> void f(T* ptr)
{
   cout << "funkcja szablonowa f(T*)" << endl;
}

void h(void (*pf)(int*))
{
   cout << "h( void (*pf)(int*)" << endl;
}

int main()
{
   h(&f<int>);    // przekazanie adresu funkcji wygenerowanej
                  // na podstawie szablonu
}
```

3.1.6 Parametry szablonu dla wartości zwracanych przez funkcję

W przypadku, gdy funkcja szablonowa ma zwrócić typ inny niż typy argumentów (lub przynajmniej rozważana jest taka możliwość) możemy zastosować następujące rozwiązania:

Parametr szablonu określający zwracany typ

```
template<typename TReturn, typename T1, typename T2>
TReturn max (T1 a, T2 b);
```

Ponieważ nie ma związku pomiędzy typami argumentów a typem zwracanym, wywołując szablon należy określić jawnie typ zwracany.

```
::max<double>(4, 7.2);
```

Dedukcja typu zwracanego z funkcji (C++14)

```
template<typename T1, typename T2>
auto max (T1 a, T2 b)
{
    return b < a ? a : b;
}
```

Użycie klasy cech (*type traits*)

```
#include <type_traits>
template<typename T1, typename T2>
std::common_type_t<T1, T2> max (T1 a, T2 b)
{
    return b < a ? a : b;
}
```

3.1.7 Domyślne parametry szablonu

Definiują parametry szablonu, możemy określić dla nich wartości domyślne. Mogą one odwoływać się do wcześniej zdefiniowanych parametrów szablonu.

```
#include <type_traits>

template<typename T1, typename T2, typename RT = std::common_type_t<T1,T2>>
RT max (T1 a, T2 b)
{
    return b < a ? a : b;
}
```

Wywołując szablon funkcji możemy pominąć argumenty z domyślnymi wartościami:

```
auto val_1 = ::max(1, 3.14); // ::max<int, double, double>
```

lub jawnie podać odpowiedni argument:

```
auto val_2 = ::max<int, short, double>(1, 4);
```

3.2 Szablony klas

Podobnie do funkcji, klasy oraz struktury też mogą być być szablonami sparametryzowanymi typami.

Szablony klas mogą być wykorzystane do implementacji kontenerów, które mogą przechowywać dane typów, które będą definiowane później.

W terminologii obiektowej szablony klas nazywane są *klasami parametryzowanymi*.

W przypadku użycia szablonów klas generowany jest kod tylko dla tych funkcji składowych, które rzeczywiście są wywoływane.

```
template <typename T>
class Vector
{
    size_t size_;
    T* items_;

public:
    explicit Vector(size_t size);
    ~Vector() { delete [] items_; }

    //...

    const T& operator[](size_t index) const
    {
         return items_[index];
    }

    T& operator[](size_t index)
    {
         return items_[index];
    }

    const T& at(size_t index) const;
    T& at(size_t index);

    size_t size() const
    {
        return size_;
    }
};
```

Aby utworzyć zmienne typów szablonowych musimy określić parametry szablonu klasy:

```
Vector<int> integral_numbers(100);
Vector<double> real_numbers(200);
Vector<std::string> words(665);
Vector<Vector<int>> matrix(10);
```

3.2.1 Implementacja funkcji składowych

Definiując funkcję składową szablonu klasy należy określić jej przynależność do szablonu.

```
template <typename T>
Vector<T>::Vector(size_t size)
    : size_{size}, items_{new T[size]}
{
}

template <typename T>
T& Vector<T>::at(size_t index)
{
    if (index >= size_)
        throw std::out_of_range("Vector::operator[]");

    return items_[index];
}

template <typename T>
const T& Vector<T>::at(size_t index) const
{
    if (index >= size_)
        throw std::out_of_range("Vector::operator[]");

    return items_[index];
}
```

3.2.2 Parametry szablonów klas

Każdy parametr szablonu może być:

1. Typem (wbudowanym lub zdefiniowanym przez użytkownika).
2. Stałą znaną w chwili kompilacji (liczby całkowite, wskaźniki i referencje danych statycznych).
3. Innym szablonem.

Parametry szablonów niebędące typami

Można używać parametrów nie będących typami, o ile są to wartości znane na etapie kompilacji.

```
template <class T, size_t N>
struct Array
{
    using value_type = T;

    T items_[N];

    constexpr size_t size()
    {
        return N;
    }
};

Array<int, 1024> buffer;
```

Parametrom szablonu klasy można przypisać argumenty domyślne (od C++11 jest to możliwe również dla szablonów funkcji).

```
template <class T, size_t N = 1024>
struct Array
{
public:
    using value_type = T;

    constexpr size_t size()
    {
        return N;
    }

    T items_[N];
};

template <typename T, typename Container = std::vector<T>>
class Stack
{
public:
    Stack();
    void push(const T& elem);
    void push(T&& elem);
    void pop();
    T& top() const;
private:
    Container items_;
};

// creating objects
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
Stack<int> stack_one; // Stack<int, vector<int>>
Stack<int, std::list<int>> stack_two;
```

Szablony jako parametry szablonów

Jeżeli w kodzie jako parametr ma być użyty inny szablon, to kompilator powinien zostać o tym poinformowany.

```
template <typename T, template<typename, size_t> class Container, size_t N = 1024>
class Stack
{
private:
    Container<T, N> elems_;
public:
    Stack();
    void push(const T& elem);
    void push(T&& elem);
    void pop();
    T& top() const;
};

template <typename T, template<typename, size_t> class Container, size_t N>
Stack<T, Container, N>::Stack()
{
}

// creating objects
Stack<int, Array> stack_three;
Stack<int, Array, 512> stack_four;
```

3.2.3 Specjalizacja szablonów klas

Specjalizacja szablonów klas:

- polega na osobnej implementacji szablonów dla wybranych typów argumentów.
- umożliwia optymalizację implementacji dla wybranych typów lub uniknięcie niepożądanego zachowania na skutek utworzenia instancji szablonu dla określonego typu.

```
template <typename T> class Pair { /* ... */ };     // szablon ogólny
template<typename T> class Pair<T*> { /* ... */ };  // częściowa specjalizacja
template<> class Pair<const char*>{ /* ... */ };    // pełna specjalizacja
```

Pełna specjalizacja szablonu klasy

Deklaracja pełnej specjalizacji wymaga podania:

```
template <>
class Vector<bool>
{
    //...
};
```

Jeśli specjalizujemy szablon klasy, to musimy zapewnić wyspecjalizowaną implementację dla wszystkich funkcji składowych.

Możliwe jest natomiast rozszerzenie interfejsu klasy o dodatkowe składowe (np. `std::vector<bool>` definiuje dodatkowe metody `flip()`)

Specjalizacja częściowa szablonu klasy

Dla szablonów klas (w odróżnieniu od szablonów funkcji) możliwe jest tworzenie częściowych specjalizacji szablonów.

Dla szablonu klasy:

```
template <class T1, class T2>
class MyClass
{
  //...
};
```

możemy utworzyć następujące specjalizacje częściowe:

```
template <typename T>
class MyClass<T, T>
{
  // specjalizacja częściowa: drugim typem jest T
};
```

```
template <typename T>
class MyClass<T, int>
{
    // specjalizacja częściowa: drugim typem jest int
};
```

```
template <typename T1, typename T2>
class MyClass<T1*, T2*>
{
    // oba parametry są wskaźnikami
};
```

Poniższe przykłady pokazują, które wersje szablonu klasy zostaną utworzone:

```
MyClass<int, float> mif;      // uses MyClass<T1,T2>
MyClass<float, float> mff;    // uses MyClass<T,T>
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
MyClass<float, int> mfi;      // uses MyClass<T,int>
MyClass<int*, float*> mp;     // uses MyClass<T1*,T2*>
```

W przypadku, gdy więcej niż jedna specjalizacja pasuje wystarczająco dobrze zgłaszany jest błąd dwuznaczności:

```
MyClass<int, int> me1; // ERROR: matches MyClass<T, T> & MyClass<T, int>

MyClass<int*, int*> me2; // ERROR: matches MyClass<T, T> & MyClass<T1*, T1*>
```

3.2.4 Składowe jako szablony

Składowe klas mogą być szablonami. Dotyczy to:

- wewnętrznych klas pomocniczych,
- funkcji składowych.

```
template <typename T>
class Stack
{
    std::deque<T> items_;
public:
    void push(const T&);
    void pop();
    //...
    // przypisanie stosu o elementach typu T2
    template <typename T2>
    Stack<T>& operator=(const Stack<T2>&);
};

template <typename T>
    template <typename T2>
Stack<T>& Stack<T>::operator=(const Stack<T2>& source)
{
    //...
}
```

3.2.5 Nazwy zależne od typów

Gramatyka języka C++ nie jest niezależna od kontekstu. Aby sparsować np. definicję funkcji, potrzebna jest znajomość kontekstu, w którym funkcja jest definiowana.

Przykład problemu:

```
template <typename T>
auto dependent_name_context1(int x)
{
    auto value = T::A(x);
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    return value;
}
```

Standard C++ rozwiązuje problem przyjmując założenie, że dowolna nazwa, która jest zależna od parametru szablonu odnosi się do zmiennej, funkcji lub obiektu.

```
struct S1
{
    static int A(int v) { return v * v; }
};

auto value2 = dependent_name_context1<S1>(10); // OK - T::A(x) was parsed as a function call
```

Słowo kluczowe `typename` umożliwia określenie, że dany symbol (identyfikator) występujący w kodzie szablonu i zależny od parametru szablonu jest typem, np. typem zagnieżdżonym – zdefiniowanym wewnątrz klasy.

```
struct S2
{
    struct A
    {
        int x;

        A(int x) : x{x}
        {}
    };
};

template <typename T>
auto dependent_name_context2(int x)
{
    auto value = typename T::A(x); // hint for a compiler that A is a type

    return value;
}

auto value = dependent_name_context2<S2>(10); // OK - T::A was parsed as a nested type
```

Przykład użycia słowa kluczowego `typename` dla zagnieżdżonych typów definiowanych w kontenerach:

```
template <class T>
class Vector
{
public:
    using value_type = T;
    using iterator = T*;
    using const_iterator = const T*;
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    // ...
    // rest of implementation
};

template <typename Container>
typename Container::value_type sum(const Container& container)
{
    using result_type = typename Container::value_type;

    result_type result{};

    for(typename Container::const_iterator it = container.begin(); it != container.end(); ++it)
        result += *it;

    return result;
}
```

Podobne problemy dotyczą również nazw zależnych od parametrów szablonu i odwołujących się do zagnieżdżonych definicji innych szablonów:

```
struct S1 { static constexpr int A = 0; }; // S1::A is an object
struct S2 { template<int N> static void A(int) {} }; // S2::A is a function template
struct S3 { template<int N> struct A {}; }; // S3::A is a class template
int x;

template<class T>
void foo()
{
    T::A < 0 > (x); // if T::A is an object, this is a pair of comparisons;
                    // if T::A is a typename, this is a syntax error;
                    // if T::A is a function template, this is a function call;
                    // if T::A is a class or alias template, this is a declaration.
}

foo<S1>(); // OK
```

Aby określić, że dany symbol zależny od parametru szablonu to szablon funkcji piszemy:

```
template <typename T>
voi foo()
{
    T::template A<0>();
}
```

Aby określić, że dany symbol zależny od parametru szablonu to szablon klasy piszemy:

```
template <typename T>
voi foo()
{
    typename T::template A<0>();
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
}
```

3.2.6 Dedukcja argumentów szablonu dla klas

C++17 wprowadza mechanizm dedukcji argumentów szablonu klasy (*Class Template Argument Deduction*). Typy parametrów szablonu klasy mogą być dedukowane na podstawie argumentów przekazanych do konstruktora tworzonego obiektu.

```
template <typename T>
class complex
{
    T re_, img_;
public:
    complex(T re, T img) : re_{re}, img_{img}
    {}
};

auto c1 = complex<int>{5, 2}; // OK - all versions of C++ standard

auto c2 = complex{5, 3}; // OK since C++17 - compiler deduces complex<int>

auto c3 = complex(5.1, 6.5); // OK in C++17 - compiler deduces complex<double>

auto c4 = complex{5, 4.1}; // ERROR - args don't have the same type
auto c5 = complex(5, 4.1); // ERROR - args don't have the same type
```

Ostrzeżenie: Nie można częściowo dedukować argumentów szablonu klasy. Należy wyspecyfikować lub wydedukować wszystkie parametry z wyjątkiem parametrów domyślnych.

Praktyczny przykład dedukcji argumentów szablonu klasy:

```
std::mutex mtx;

std::lock_guard lk{mtx}; // deduces std::lock_guard<std::mutex>
```

Podpowiedzi dedukcyjne (*deduction guides*)

C++17 umożliwia tworzenie podpowiedzi dla kompilatora, jak powinny być dedukowane typy parametrów szablonu klasy na podstawie wywołania odpowiedniego konstruktora.

Daje to możliwość poprawy/modyfikacji domyślnego procesu dedukcji.

Dla szablonu:

```
template <typename T>
class S
{
private:
    T value;
public:
    S(T v) : value(v)
    {}
};
```

Podpowiedź dedukcyjna musi zostać umieszczona w tym samym zakresie (przestrzeni nazw) i może mieć postać:

```
template <typename T> S(T) -> S<T>; // deduction guide
```

gdzie:

- `S<T>` to tzw. typ zalecany (*guided type*)
- nazwa podpowiedzi dedukcyjnej musi być niekwalifikowaną nazwą klasy szablonowej zadeklarowanej wcześniej w tym samym zakresie
- typ zalecany podpowiedzi musi odwoływać się do identyfikatora szablonu (*template-id*), do którego odnosi się podpowiedź

Użycie podpowiedzi:

```
S x{12}; // OK -> S<int> x{12};
S y(12); // OK -> S<int> y(12);
auto z = S{12}; // OK -> auto z = S<int>{12};
S s1(1), s2{2}; // OK -> S<int> s1(1), s2{2};
S s3(42), s4{3.14}; // ERROR
```

W deklaracji `S x{12};` specyfikator `S` jest nazywany symbolem zastępczym dla klasy (*placeholder class type*).

W przypadku użycia symbolu zastępczego dla klasy, nazwa zmiennej musi zostać podana jako następny element składni. W rezultacie poniższa deklaracja jest błędem składniowym:

```
S* p = &x; // ERROR - syntax not permitted
```

Dany szablon klasy może mieć wiele konstruktorów oraz wiele podpowiedzi dedukcyjnych:

```
template <typename T>
struct Data
{
    T value;

    using type1 = T;
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    Data(const T& v)
        : value(v)
    {
    }

    template <typename ItemType>
    Data(initializer_list<ItemType> il)
        : value(il)
    {
    }
};

template <typename T>
Data(T)->Data<T>;

template <typename T>
Data(initializer_list<T>)->Data<vector<T>>;

Data(const char*) -> Data<std::string>;

//...

Data d1("hello"); // OK -> Data<string>

const int tab[10] = {1, 2, 3, 4};
Data d2(tab); // OK -> Data<const int*>

Data d3 = 3; // OK -> Data<int>

Data d4{1, 2, 3, 4}; // OK -> Data<vector<int>>

Data d5 = {1, 2, 3, 4}; // OK -> Data<vector<int>>

Data d6 = {1}; // OK -> Data<vector<int>>

Data d7(d6); // OK - copy by default rule -> Data<vector<int>>

Data d8{d6, d7}; // OK -> Data<vector<Data<vector<int>>>>
```

Podpowiedzi dedukcyjne nie są szablonami funkcji - służą jedynie dedukowaniu argumentów szablonu i nie są wywoływane. W rezultacie nie ma znaczenia czy argumenty w deklaracjach dedukcyjnych są przekazywane przez referencje, czy nie.

```
template <typename T>
struct X
{
    //...
};

template <typename T>
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
struct Y
{
    Y(const X<T>&);
    Y(X<T>&&);
};

template <typename T> Y(X<T>) -> Y<T>; // deduction guide without references
```

W powyższym przykładzie podpowiedź dedukcyjna nie odpowiada dokładnie sygnaturom konstruktorów przeciążonych. Nie ma to znaczenia, ponieważ jedynym celem podpowiedzi jest umożliwienie dedukcji typu, który jest parametrem szablonu. Dopasowanie wywołania przeciążonego konstruktora odbywa się później.

Niejawne podpowiedzi dedukcyjne

Ponieważ często podpowiedź dedukcyjna jest potrzebna dla każdego konstruktora klasy, standard C++17 wprowadza mechanizm **niejawnych podpowiedzi dedukcyjnych** (*implicit deduction guides*). Działa on w następujący sposób:

- Lista parametrów szablonu dla podpowiedzi zawiera listę parametrów z szablonu klasy - w przypadku szablonowego konstruktora klasy kolejnym elementem jest lista parametrów szablonu konstruktora klasy
- Parametry „funkcyjne" podpowiedzi są kopiowane z konstruktora lub konstruktora szablonowego
- Zalecany typ w podpowiedzi jest nazwą szablonu z argumentami, które są parametrami szablonu wziętymi z klasy szablonowej

Dla klasy szablonowej rozważanej powyżej:

```
template <typename T>
class S
{
private:
    T value;
public:
    S(T v) : value(v)
    {}
};
```

niejawna podpowiedź dedukcyjna będzie wyglądać następująco:

```
template <typename T> S(T) -> S<T>; // implicit deduction guide
```

W rezultacie programista nie musi implementować jej jawnie.

Specjalny przypadek dedukcji argumentów klasy szablonowej

Rozważmy następujący przypadek dedukcji:

```
S x{42}; // x has type S<int>

S y{x};
S z(x);
```

W obu przypadkach dedukowany typ zmiennych `y` i `z` to `S<int>`.

W tym przypadku mechanizm dedukcji argumentów klasy szablonowej realizuje kopię obiektu.

- dla deklaracji `S<T> x; S{x}` dedukuje typ: `S<T>{x}` zamiast `S<S<T>>{x}`

W niektórych przypadkach może być to zaskakujące i kontrowersyjne:

```
std::vector v{1, 2, 3}; // vector<int>
std::vector data1{v, v}; // vector<vector<int>>
std::vector data2{v}; // vector<int>!
```

W powyższym kodzie dedukcja argumentów szablonu `vector` zależy od ilości argumentów przekazanych do konstruktora!

Agregaty a dedukcja argumentów

Jeśli szablon klasy jest agregatem, to mechanizm automatycznej dedukcji argumentów szablonu wymaga napisania jawnej podpowiedzi dedukcyjnej.

Bez podpowiedzi dedukcyjnej dedukcja dla agregatów nie działa:

```
template <typename T>
struct Aggregate1
{
    T value;
};

Aggregate1 agg1{8}; // ERROR
Aggregate1 agg2{"eight"}; // ERROR
Aggregate1 agg3 = 3.14; // ERROR
```

Gdy napiszemy dla agregatu podpowiedź, to możemy zacząć korzystać z mechanizmu dedukcji:

```
template <typename T>
struct Aggregate2
{
    T value;
};

template <typename T>
Aggregate2(T) -> Aggregate2<T>;

Aggregate2 agg1{8}; // OK -> Aggregate2<int>
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
Aggregate2 agg2{"eight"}; // OK -> Aggregate2<const char*>
Aggregate2 agg3 = { 3.14 }; // OK -> Aggregate2<double>
```

Podpowiedzi dedukcyjne w bibliotece standardowej

Dla wielu klas szablonowych z biblioteki standardowej dodano podpowiedzi dedukcyjne w celu ułatwienia tworzenia instancji tych klas.

std::pair<T>

Dla pary STL dodana w standardzie podpowiedź to:

```
template<class T1, class T2>
pair(T1, T2) -> pair<T1, T2>;

pair p1(1, 3.14); // -> pair<int, double>

pair p2{3.14f, "text"s}; // -> pair<float, string>

pair p3{3.14f, "text"}; // -> pair<float, const char*>

int tab[3] = { 1, 2, 3 };
pair p4{1, tab}; // -> pair<int, int*>
```

std::tuple<T...>

Szablon `std::tuple` jest traktowany podobnie jak `std::pair`:

```
template<class... UTypes>
tuple(UTypes...) -> tuple<UTypes...>;

template<class T1, class T2>
tuple(pair<T1, T2>) -> tuple<T1, T2>;

//... other deduction guides working with allocators

int x = 10;
const int& cref_x = x;

tuple t1{x, &x, cref_x, "hello", "world"s}; -> tuple<int, int*, int, const char*, string>
```

std::optional<T>

Klasa `std::optional` jest traktowana podobnie do pary i krotki.

```
template<class T> optional(T) -> optional<T>;

optional o1(3); // -> optional<int>
optional o2 = o1; // -> optional<int>
```

Inteligentne wskaźniki

Dedukcja dla argumentów konstruktora będących wskaźnikami jest zablokowana:

```
int* ptr = new int{5};
unique_ptr uptr{ip}; // ERROR - ill-formed (due to array type clash)
```

Wspierana jest dedukcja przy konwersjach:

- z `weak_ptr`/`unique_ptr` do `shared_ptr`:

  ```
  template <class T> shared_ptr(weak_ptr<T>) ->  shared_ptr<T>;
  template <class T, class D> shared_ptr(unique_ptr<T, D>) ->  shared_ptr<T>;
  ```

- z `shared_ptr` do `weak_ptr`

  ```
  template<class T> weak_ptr(shared_ptr<T>) -> weak_ptr<T>;
  ```

```
unique_ptr<int> uptr = make_unique<int>(3);

shared_ptr sptr = move(uptr); -> shared_ptr<int>

weak_ptr wptr = sptr; // -> weak_prt<int>

shared_ptr sptr2{wptr}; // -> shared_ptr<int>
```

std::function

Dozwolone jest dedukowanie sygnatur funkcji dla `std::function`:

```
int add(int x, int y)
{
    return x + y;
}

function f1 = &add;
assert(f1(4, 5) == 9);

function f2 = [](const string& txt) { cout << txt << " from lambda!" << endl; };
f2("Hello");
```

Kontenery i sekwencje

Dla kontenerów standardowych dozwolona jest dedukcja typu kontenera dla konstruktora akceptującego parę iteratorów:

```
vector<int> vec{ 1, 2, 3 };
list lst(vec.begin(), vec.end()); // -> list<int>
```

Dla `std::array` dozwolona jest dedukcja z sekwencji:

```
std::array arr1 { 1, 2, 3 }; // -> std::array<int, 3>
```

3.3 Aliasy szablonów

3.3.1 Aliasy typów

W C++11 deklaracja `using` może zostać użyta do tworzenia bardziej czytelnych aliasów dla typów – zamiennik dla `typedef`.

```
using CarID = int;

using Func = int(*)(double, double);

using DictionaryDesc = std::map<std::string, std::string, std::greater<std::string>>;
```

3.3.2 Aliasy szablonów

Aliasy typów mogą być parametryzowane typem. Można je wykorzystać do tworzenia częściowo związanych typów szablonowych.

```
template <typename T>
using StrKeyMap = std::map<std::string, T>;

StrKeyMap<int> my_map; // std::map<std::string, int>
```

Parametrem aliasu typu szablonowego może być także stała znana w czasie kompilacji:

```
template <std::size_t N>
using StringArray = std::array<std::string, N>;

StringArray<255> arr1;
```

Aliasy szablonów nie mogą być specjalizowane.

```
template <typename T>
using MyAllocList = std::list<T, MyAllocator>;
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
template <typename T>
using MyAllocList = std::list<T*, MyAllocator>; // error
```

Przykład utworzenia aliasu dla szablonu klasy *smart pointer'a*:

```
template <typename Stream>
struct StreamDeleter
{
    void operator()(Stream* os)
    {
        os->close();
        delete os;
    }
}

template <typename Stream>
using StreamPtr = std::unique_ptr<Stream, StreamDeleter<Stream>>;

// ...
{
    StreamPtr<ofstream> p_log(new ofstream("log.log"));
    *p_log << "Log statement";
}
```

Aliasy i typename

Od C++14 biblioteka standardowa używa aliasów dla wszystkich cech typów, które zwracają typ.

```
template <typename T>
using is_void_t = typename is_void<T>::type;
```

W rezultacie kod odwołujący się do cechy:

```
typename is_void<T>::type
```

możemy uprościć do:

```
is_void_t<T>;
```

3.4 Szablony zmiennych

W C++14 zmienne mogą być parametryzowane przy pomocy typu. Takie zmienne nazywamy **zmiennymi szablonowymi** (*variable templates*).

```
template<typename T>
constexpr T pi{3.1415926535897932385};
```

Aby użyć zmiennej szablonowej, należy podać jej typ:

```
std::cout << pi<double> << '\n';
std::cout << pi<float> << '\n';
```

Parametrami zmiennych szablonowych mogą być stałe znane na etapie kompilacji:

```
template<int N>
std::array<int, N> arr{};

int main()
{
    arr<10>[0] = 42; // sets first element of global arr

    for (std::size_t i = 0; i < arr<10>.size(); ++i)
    {
        // uses values set in arr
        std::cout << arr<10>[i] << '\n';
    }
}
```

3.4.1 Zmienne szablonowe a jednostki translacji

Deklaracja zmiennych szablonowych może być używana w innych jednostkach translacji.

- plik – header.hpp

```
template<typename T> T val{};     // zero initialized value
```

- plik – „unit1.cpp”

```
#include "header.hpp"

int main()
{
    val<long> = 42;
    print();
}
```

- plik – „unit2.cpp”

```
void print()
{
    std::cout << val<long> << '\n'; // OK: prints 42
}
```

4 | Klasy cech i wytycznych

4.1 Klasy cech

Klasy cech (traits) reprezentują dodatkowe właściwości parametru szablonu, które mogą być pomocne na etapie implementacji kodu szablonu.

4.1.1 Case Study - Kumulowanie elementów sekwencji

```
template <typename T>
T accumulate(const T* begin, const T* end)
{
    T total = T{};

    while (begin != end)
    {
        total += *begin;
        ++begin;
    }

    return total;
}
```

Problemy:

- określenie typu zmiennej kumulującej
- utworzenie wartości zerowej

Parametryzacja typu zmiennej kumulującej

```
template <typename T>
struct AccumulationTraits;

template <>
struct AccumulationTraits<uint8_t>
{
    typedef unsigned int AccumulatorType;
};
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
template<>
struct AccumulationTraits<float>
{
    typedef double AccumulatorType;
};
```

Szablon `AccumulationTraits` zwany jest szablonem **cechy**, gdyż przechowuje cechę typu, który jest parametrem szablonu.

Algorytm korzystający z klasy cech wygląda następująco:

```
template <typename T>
typename AccumulationTraits<T>::AccumulatorType
accumulate(const T* begin, const T* end)
{
    using AccT = typename AccumulationTraits<T>::AccumulatorType;

    AccT total = T{};

    while (begin != end)
    {
        total += *begin;
        ++begin;
    }

    return total;
}
```

Cechy wartości

Klasy cech mogą również zawierać informację o stałych charakterystycznych dla opisywanego typu. Możemy w klasie cechy zdefiniować wartość zerową dla typu.

```
template<typename T> struct AccumulationTraits;

template<>
struct AccumulationTraits<uint8_t>
{
    using AccumulatorType = int;
    static constexpr AccumulatorType zero = 0;
};

template<>
struct AccumulationTraits<float>
{
    using AccumulatorType = float;
    static constexpr float zero = 0.0f;
};
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
template<>
struct AccumulationTraits<BigInt>
{
    using AccumulationTraits = BigInt;
    inline static BigInt const zero = BigInt{0};  // OK since C++17
};
```

Algorytm korzystający z klasy cech uwzględniającej wartość zerową:

```
template <typename T>
typename AccumulationTraits<T>::AccumulatorType accumulate(const T* begin, const T* end)
{
    using AccT = typename AccumulationTraits<T>::AccumulatorType;

    AccT total = AccumulationTraits<T>::zero; // wykorzystanie cechy

    while (begin != end)
    {
        total += *begin;
        ++begin;
    }

    return total;
}
```

Parametryzowanie cech typów

Parametryzacja cech wymaga dodatkowych parametrów szablonu. Aby stosowanie sparametryzowanych cech typów było wygodne, należy wykorzystać domyślne wartości parametrów szablonu.

```
template <typename T, typename Traits = AccumulationTraits<T>>
typename Traits::AccumulatorType accumulate(const T* begin, const T* end)
{
    using AccT = typename Traits::AccumulatorType;

    AccT total = Traits::zero;

    while (begin != end)
    {
        total += *begin;
        ++begin;
    }

    return total;
}
```

4.2 Klasy wytycznych

Wytyczne (policies) reprezentują konfigurowalne zachowania ogólnych funkcji i typów

Klasa wytycznych – klasa udostępniająca zbiór metod implementujących określony sposób zachowania (algorytm)

```
template
<
   typename T,
   typename AccumulationPolicy = Sum,
   typename Traits = AccumulationTraits<T>
>
typename Traits::AccumulatorType accumulate (const T* begin, const T* end)
{
    using AccT = typename Traits::AccumulatorType;

    AccT total = Traits::zero;

    while (begin != end)
    {
        AccumulationPolicy::accumulate(total, *begin);
        ++begin;
    }

    return total;
}
```

Domyślna klasa wytycznej:

```
struct Sum
{
    template <typename T1, typename T2>
    static void accumulate(T1& total, const T2& value)
    {
        total += value;
    }
};
```

Zmodyfikowana klasa wytycznej:

```
struct Multiply
{
    template <typename T1, typename T2>
    static void accumulate(T1& total, const T2& value)
    {
        total *= value;
    }
};

int main()
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
{
    int data[] = {1,2,3,4,5};
    // wyświetl iloczyn wszystkich wartości
    std::cout << "the product of the integer values is " <<
        accumulate<int, MultiplyPolicy>(begin(data), end(data)) << '\n';
}
```

4.3 Klasy parametryzowane wytycznymi

Konstruowanie klas parametryzowanych wytycznymi polega na składaniu skompliko-wanego zachowania klasy z wielu małych klas (wytycznych).

Każda wytyczna:

- Określa jeden sposób zachowania lub implementacji
- Ustala interfejs dotyczący jednej konkretnej czynności
- Może być implementowana na wiele sposobów

Klasa podstawowa:

```
template <typename T>
class Vector
{
public:
    /* constructors */

    const T& at(size_t index) const;
    void push_back(const T& value);

    /* ... etc. ... */
};
```

Klasa, której zachowanie jest sparametryzowane wytycznymi:

```
template
<
    typename T,
    typename RangeCheckPolicy,
    typename LockingPolicy = NullMutex
>
class Vector : public RangeCheckPolicy
{
    std::vector<T> items_;
    using mutex_type = LockingPolicy;
    mutable mutex_type mtx_;

public:
    using iterator = typename std::vector<T>::iterator;
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    using const_iterator = typename std::vector<T>::const_iterator;

    /* ... etc. ... */
};
```

Przykładowa klasa wytycznej sprawdzającej poprawność zakresu:

```
class ThrowingRangeChecker
{
protected:
    ~ThrowingRangeChecker() = default;

    void check_range(size_t index, size_t size) const
    {
        if (index >= size)
            throw std::out_of_range("Index out of range...");
    }
};
```

Inna implementacja wytycznej kontrolującej zakres indeksów:

```
class LoggingErrorRangeChecker
{
public:
    void set_log_file(std::ostream& log_file)
    {
        log_ = &log_file;
    }

protected:
    ~LoggingErrorRangeChecker() = default;

    void check_range(size_t index, size_t size) const
    {
        if ((index >= size) && (log_ != nullptr))
            *log_ << "Error: Index out of range. Index="
                << index << "; Size=" << size << std::endl;
    }

private:
    std::ostream* log_{};
};
```

Implementacja metody `at()` wektora z uwzględnieniem wytycznych:

```
template
<
    typename T,
    typename RangeCheckPolicy,
    typename LockingPolicy
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
>
const T& Vector<T, RangeCheckPolicy, LockingPolicy>::at(size_t index) const
{
    std::lock_guard<mutex_type> lk{mtx_};

    RangeCheckPolicy::check_range(index, size());

    return (index < items_.size()) ? items_[index] : items_.back();
}
```

Kod klienta:

```
Vector<int, ThrowingRangeChecker, StdLock> vec = {1, 2, 3};
vec.push_back(4);

try
{
    auto value = vec.at(8);
}
catch(const std::out_of_range& e)
{
    std::cout << e.what() << std::endl;
}
```

4.4 Klasa cech iteratorów

Biblioteka standardowa C++ często wykorzystuje technikę cech i wytycznych. Jedną z bardziej przydatnych klas cech w bibliotece standardowej, jest klasa cech iteratorów.

```
template <class Iterator>
struct iterator_traits
{
    typedef typename Iterator::iterator_category iterator_category;
    typedef typename Iterator::value_type value_type;
    typedef typename Iterator::difference_type difference_type;
    typedef typename Iterator::pointer pointer;
    typedef typename Iterator::reference reference;
};

template <class T>
struct iterator_traits<T*>
{
    typedef random_access_iterator_tag iterator_category;
    typedef T value_type;
    typedef ptrdiff_t difference_type;
    typedef T* pointer;
    typedef T& reference;
};
```

Przykład wykorzystania klasy cech iteratorów:

```
#include <iterator>
template <typename Iter>
inline
typename std::iterator_traits<Iter>::value_type accum (Iter start, Iter end)
{
    typedef typename std::iterator_traits<Iter>::value_type VT;
    VT total = VT(); // assume T() actually creates a zero value
    while (start != end)
    {
        total += *start;
        ++start;
    }
    return total;
}
```

4.5 Tag dispatching

Czasami pożądane jest dostarczenie wyspecjalizowanych implementacji dla wybranej funkcji lub klasy w celu poprawy wydajności lub uniknięcia problemów.

Przykładem może być implementacja funkcji `advance_iter()`, która przesuwa iterator `it` o zadaną `n` ilość kroków.

Generyczna implementacja może operować na dowolnym typie iteratora:

```
template<typename InputIterator, typename Distance>
void advance_iter(InputIterator& x, Distance n)
{
    while (n > 0)
    {
        ++x;
        --n;
    }
}
```

Nie jest to implementacja optymalna dla iteratorów o swobodnym dostępie (np. `vector<int>::iterator`).

Optymalizacja funkcji polega na utworzeniu grupy funkcji pomocniczych, które mogą być dopasowane do odpowiedniego rodzaju iteratora za pomocą „taga”, który umożliwia przeciążenie tych funkcji.

```
template<typename Iterator, typename Distance>
void advance_iter_impl(Iterator& x, Distance n, std::input_iterator_tag)
{
    // complexity - O(N)
    while (n > 0)
    {
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
        ++x;
        --n;
    }
}

template<typename Iterator, typename Distance>
void advance_iter_impl(Iterator& x, Distance n, std::random_access_iterator_tag)
{
    // complexity - O(1)
    x += n;
}
```

Funkcja `advance_iter()` po prostu przyjmuje argumenty i przekazuje je do funkcji pomocniczej. Na podstawie „taga" odbywa się dopasowanie odpowiedniej implementacji.

```
template<typename Iterator, typename Distance>
void advance_iter(Iterator& x, Distance n)
{
    advance_iter_impl(x, n,
        typename std::iterator_traits<Iterator>::iterator_category{});
}
```

Klasa cech `std::iterator_traits` umożliwia określenie rodzaju iteratora za pomocą typu `iterator_category`.

Biblioteka standardowa definiuje zbiór typów pełniących tagujących kategorię iteratora:

```
namespace std
{
    struct input_iterator_tag { };
    struct output_iterator_tag { };
    struct forward_iterator_tag : public input_iterator_tag { };
    struct bidirectional_iterator_tag : public forward_iterator_tag { };
    struct random_access_iterator_tag : public bidirectional_iterator_tag { };
}
```

5 | Biblioteka klas cech typów - <type_traits>

Biblioteka standardowa zawiera zestaw szablonów klas umożliwiających cechowanie typów na etapie kompilacji. Dzięki cechom typów (**type traits**) na etapie kompilacji:

- wybierana jest optymalna (wydajna) implementacja danej funkcjonalności (z wykorzystaniem SFINAE).
- przeprowadzana jest statyczna asercja umożliwiająca wczesne wychwycenie błędów

5.1 Meta-funkcje

Technika cechowania wykorzystuje meta-funkcje, najczęściej implementowane jako szablony klas przyjmujące jako parametr cechowany typ i zwracające:

- `type_trait<T>::type` – nowy typ będący efektem transformacji (np. usunięcia modyfikatora `const`)
- `type_trait<T>::value` – stałą statyczną wartość `constexpr` (najczęściej typu `bool`)

5.1.1 Meta-funkcje zwracające wartość

Przykład implementacji meta-funkcji zwracającej stałą wartość ewaluowaną na etapie kompilacji:

```
template<typename T>
struct TypeSize
{
    static constexpr std::size_t value = sizeof(T);
};
```

Wykorzystanie meta-funkcji:

```
std::cout << TypeSize<int>::value << std::endl;
```

5.1.2 Meta-funkcje zwracające typy

Przy implementacji meta-funkcji często stosujemy specjalizację szablonów:

```
// generic implementation for containers
template<typename C>
struct ElementT
{
    using type = typename C::value_type
;

//partial specialization for arrays of known bounds
template<typename T, std::size_t N>
struct ElementT<T[N]>
{
    using type = T;
};

//partial specialization for arrays of unknown bounds
template<typename T>
struct ElementT<T[]>
{
    using type = T;
};
```

Meta-funkcje mogą być wykorzystane w innych szablonach:

```
template <typename Container>
void process(const Container& c)
{
    typename ElementT<Container>::type item{};

    //...
}
```

5.2 Stałe całkowite - *integral constants*

Szablon `std::integral_constant` pozwala opakować w w postaci meta-funkcji stałą zdefiniowaną na etapie kompilacji.

```
template<class T, T v>
struct integral_constant
{
    static constexpr T value = v;
    typedef T value_type;
    typedef integral_constant type; // using injected-class-name
    constexpr operator value_type() const noexcept { return value; }
    constexpr value_type operator()() const noexcept { return value; } //since c++14
};
```

Wykorzystanie meta-funkcji `integral_constant` wygląda następująco:

```
integral_constant<int, 5>::value;
```

```
(const int) 5
```

Biblioteka standardowa definiuje pomocniczy alias dla stałej typu `bool`:

```
template <bool v>
using bool_constant = integral_constant<bool, v>;
```

Zdefiniowane są również dwa typowe przypadki takich stałych:

```
using true_type = bool_constant<true>; // integral_constant<bool, true>
using false_type = bool_constant<false>; // integral_constant<bool, false>
```

5.3 Klasy cech typów

5.3.1 Klasy cech transformujące typy

Często w trakcie pisania kodu szablonowego zachodzi potrzeba transformacji typu określonego parametru szablonu (np. wymagane jest usunięcie lub dodanie referencji, modyfikatora `const` lub `volatile`, itp.).

W takim przypadku możemy posłużyć się klasą cechy transformującej (implementowaną jako meta-funkcja):

```
template <typename T>
struct remove_reference
{
    using type = T;
};

template <typename T>
struct remove_reference<T&>
{
    using type = T;
};

template <typename T>
struct remove_reference<T&&>
{
    using type = T;
};
```

Od C++14 do klas cech dodane są odpowiednie aliasy szablonów, które umożliwiają uniknięcie konieczności deklaracji `typename` przed zagnieżdżonym typem `type`:

```
template <typename T>
using remove_reference_t = typename remove_reference<T>::type;
```

Użycie cech transformujących wygląda następująco:

```
auto closure = [](auto&& item) {
    remove_reference_t<decltype(item)> item_cpy = item;
};
```

Cechy transformujące typy w bibliotece standardowej

Biblioteka standardowa w nagłówku `<type_traits>` definiuje zbiór klas cech transformujących:

Cecha	Rezultat `::value`
`remove_reference`	usuwa referencję z typu (`int& -> int`)
`add_lvalue_reference`	dodaje lvalue referencję (`double -> double&`)
`add_rvalue_reference`	dodaje rvalue referencję (`double -> double&&`)
`remove_pointer`	usuwa wskaźnik z typu (`int* -> int`)
`add_pointer`	dodaje wskaźnik (`int -> int*`)
`remove_const`	usuwa modyfikator `const` (`const int& -> int&`)
`remove_volatile`	usuwa modyfikator `volatile` (`volatile int -> int`)
`remove_cv`	usuwa modyfikatory `const` i `volatile`
`add_const`	dodaje modyfikator `const` (`int -> const int`)
`add_volatile`	dodaje modyfikator `volatile` (`double -> volatile double`)

Cecha `std::decay`

Przydatną cechą jest zdefiniowana w bibliotece standardowej cecha `std::decay`.

Dokonuje ona transformacji odpowiadającej następującym przekształceniom:

- usuwane są referencje
- usuwane są modyfikatory `const` lub `volatile`
- tablice konwertowane są do wskaźników
- funkcje konwertowane są do wskaźników do funkcji

```
template <typename T, typename U>
void check_decay()
{
    static_assert(std::is_same_v<std::decay_t<T>, U>);
}

check_decay<int&, int>();
check_decay<const int&, int>();
check_decay<int&&, int>();
check_decay<int(int), int(*)(int)>();
check_decay<int[20], int*>();
check_decay<const int[20], const int*>();
```

5.3.2 Klasy cech - predykaty

Implementacja cech typów, które pełnią rolę predykatów, polega zwykle na zdefiniowaniu ogólnego szablonu dziedziczącego po `false_type` (dla typów nie posiadających określonej cechy). Kolejnym krokiem jest dostarczenie wersji specjalizowanej szablonu dla typów z cechą, która dziedziczy po typie `true_type`.

Specjalizacja szablonu może być całkowita:

```
template <typename T>
struct IsVoid : std::false_type
{};

template <>
struct IsVoid<void> : std::true_type
{};
```

```
IsVoid<int>::value;
```

```
(const bool) false
```

```
IsVoid<void>::value;
```

```
(const bool) true
```

lub częściowa:

```
template <typename T>
struct IsPointer : std::false_type{};

template <typename T>
struct IsPointer<T*> : std::true_type{};
```

```
IsPointer<int>::value;
```

```
(const bool) false
```

```
IsPointer<const int*>::value;
```

```
(const bool) true
```

5.3.3 Standardowe cechy typów - predykaty

Biblioteka standardowa definiuje szeroki zbiór meta-funkcji, które umożliwiają odpytanie na etapie kompilacji, czy dany typ posiada odpowiednie cechy.

Cechy podstawowe

Cecha podstawowa	Rezultat `::value`
`is_array<T>`	`true` jeśli `T` jest typem tablicowym
`is_class<T>`	`true` jeśli `T` jest klasą
`is_enum<T>`	`true` jeśli `T` jest typem wyliczeniowym
`is_floating_point<T>`	`true` jeśli `T` jest typem zmiennoprzecinkowym
`is_function<T>`	`true` jeśli `T` jest funkcją
`is_integral<T>`	`true` jeśli `T` jest typem całkowitym
`is_member_object_pointer<T>`	`true` jeśli `T` jest wskaźnikiem do składowej
`is_member_function_pointer<T>`	`true` jeśli `T` jest wskaźnikiem do funkcji składowej
`is_pointer<T>`	`true` jeśli `T` jest typem wskaźnikowym (ale nie wskaźnikiem do składowej)
`is_lvalue_reference<T>`	`true` jeśli `T` jest referencją do l-value
`is_rvalue_reference<T>`	`true` jeśli `T` jest referencją do r-value
`is_union<T>`	`true` jeśli `T` jest unią (bez wsparcia kompilatora zawsze zwraca `false`
`is_void<T>`	`true` jeśli `T` jest typu void
`is_null_pointer<T>`	`true` jeśli `T` jest typu `std::nullptr_t`

Cechy kompozytowe

Cechy kompozytowe są kompozycją najczęściej kilku cech podstawowych.

Cecha grupowana	Rezultat `::value`
`is_arithmetic<T>`	`is_integral<T>::value || is_floating_point<T>::value`
`is_fundamental<T>`	`is_arithmetic<T>::value || is_void<T>::value || is_null_pointer<T>::value`
`is_compound<T>`	`!is_fundamental<T>::value`
`is_object<T>`	`is_scalar<T>::value || is_array<T>::value || is_union<T>::value || is_class<T>::value`
`is_reference<T>`	`is_lvalue_reference<T> || is_rvalue_reference<T>`
`is_member_pointer<T>`	`is_member_object_pointer<T> || is_member_function_pointer<T>`
`is_scalar<T>`	`is_arithmetic<T>::value || is_enum<T>::value || is_null_pointer<T>::value is_pointer<T>::value || is_member_pointer<T>::value`

Właściwości typów

Standard używa terminu **właściwość typu** w celu zdefiniowania cechy opisującej wybrane atrybuty typu.

Wybrane właściwości typu:

Właściwość typu	Rezultat `::value`
`is_const<T>`	`true` jeśli `T` jest typem `const`
`is_volatile<T>`	`true` jeśli `T` jest typem ulotnym
`is_polymorphic<T>`	`true` jeśli `T` posiada przynajmniej jedną funkcję wirtualną
`is_trivial<T>`	`true` jeśli `T` jest typem trywialnym
`is_trivially_copyable<T>`	`true` jeśli `T` jest trywialnie kopiowalne
`is_standard_layout<T>`	`true` jeśli `T` jest typem o standardowym layout'cie
`is_pod<T>`	`true` jeśli `T` jest typem POD
`is_abstract<T>`	`true` jeśli `T` jest typem abstrakcyjnym
`is_unsigned<T>`	`true` jeśli `T` jest typem całkowitym bez znaku lub typem wyliczeniowym zdefiniowanym przy pomocy typu `unsigned`
`is_signed<T>`	`true` jeśli `T` jest typem całkowitym ze znakiem lub typem wyliczeniowym zdefiniowanym przy pomocy typu `signed`

5.3.4 Cechy typów i statyczne asercje

Jednym z podstawowych zastosowań cech typów jest wykorzystanie ich do statycznych asercji w kodzie. Takie asercje nie pozwalają wygenerować błędnego kodu i jednocześnie podnoszą czytelność komunikatów o błędach w szablonach.

Przykład:

```
template <class T>
void swap(T& a, T& b)
{
    static_assert(std::is_copy_constructible<T>::value,
                  "Swap requires copying");

    static_assert(noexcept(std::is_nothrow_move_constructible<T>::value
                           && std::is_nothrow_move_assignable<T>::value),
                   "Swap may throw");

    auto c = b;
    b = a;
    a = c;
}
```

6 | Tag dispatching

Czasami pożądane jest dostarczenie wyspecjalizowanych implementacji dla wybranej funkcji lub klasy w celu poprawy wydajności lub uniknięcia problemów.

Przykładem może być implementacja funkcji `advance_iter()`, która przesuwa iterator `it` o zadaną `n` ilość kroków.

Generyczna implementacja może operować na dowolnym typie iteratora:

```
template<typename InputIterator, typename Distance>
void advance_iter(InputIterator& x, Distance n)
{
    while (n > 0)
    {
        ++x;
        --n;
    }
}
```

Nie jest to implementacja optymalna dla iteratorów o swobodnym dostępie (np. `vector<int>::iterator`).

Optymalizacja funkcji polega na utworzeniu grupy funkcji pomocniczych, które mogą być dopasowane do odpowiedniego rodzaju iteratora za pomocą „taga", który umożliwia przeciążenie tych funkcji.

```
template<typename Iterator, typename Distance>
void advance_iter_impl(Iterator& x, Distance n, std::input_iterator_tag)
{
    // complexity - O(N)
    while (n > 0)
    {
        ++x;
        --n;
    }
}

template<typename Iterator, typename Distance>
void advance_iter_impl(Iterator& x, Distance n, std::random_access_iterator_tag)
{
    // complexity - O(1)
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    x += n;
}
```

Funkcja `advance_iter()` po prostu przyjmuje argumenty i przekazuje je do funkcji pomocniczej. Na podstawie „taga" odbywa się dopasowanie odpowiedniej implementacji.

```
template<typename Iterator, typename Distance>
void advance_iter(Iterator& x, Distance n)
{
    advance_iter_impl(x, n,
        typename std::iterator_traits<Iterator>::iterator_category{});
}
```

Klasa cech `std::iterator_traits` umożliwia określenie rodzaju iteratora za pomocą typu `iterator_category`.

Biblioteka standardowa definiuje zbiór typów pełniących tagujących kategorię iteratora:

```
namespace std
{
    struct input_iterator_tag { };
    struct output_iterator_tag { };
    struct forward_iterator_tag : public input_iterator_tag { };
    struct bidirectional_iterator_tag : public forward_iterator_tag { };
    struct random_access_iterator_tag : public bidirectional_iterator_tag { };
}
```

7 | SFINAE - Substitution Failure Is Not An Error - enable_if

Jednym z podstawowych narzędzi w meta-programowaniu w C++ jest szablon **enable_if** wykorzystujący mechanizm **SFINAE**.

Motywacją dla stosowania `enable_if` jest chęć dostarczenia dla klientów uniwersalnego interfejsu, bez zmuszania ich do podejmowania decyzji o wyborze określonych implementacji. Decyzje takie są podejmowane przez implementorów bibliotek. Dobór optymalnych implementacji w zależności od typów danych dokonywany jest na etapie kompilacji.

Chcemy uniknąć kodu, który często wygląda tak:

```
// Don't even dare to pass an array of complex obejcts to this function!!!
template <typename T>
T* copy_it(T const* src, size_t n)
{
    // impl
}
```

7.1 SFINAE

Substitition Failure Is Not An Error (SFINAE) jest mechanizmem wykorzystywanym przez kompilator w trakcie dedukcji typów dla argumentów szablonu. Szablony, dla których nie udaje się podstawić określonego typu jako typu argumentu szablonu są ignorowane i nie powodują błędów kompilacji.

W praktyce jeśli dedukcja typów argumentów dla szablonu funkcji znajdzie przynajmniej jedno dopasowanie, błędne podstawienia są usuwane z listy funkcji kandydujących do wywołania (*overload resolution*) i nie zgłaszają błędów kompilacji.

Załóżmy następującą implementację szablonów funkcji:

```
template <typename T> void foo(T arg)
{}

template<typename T> void foo(T* arg)
{}
```

Wywołanie `foo(42)` powoduje utworzenie instancji szablonu funkcji `foo<int>(int)`. Ponieważ kompilatorowi nie udaje się podstawić prawidłowo typu `int` dla drugiej implementacji działa SFINAE - nieudane podstawienie nie generuje błędów kompilacji a implementacja znika z listy funkcji kandydujących do wywołania. Jeśli brakowałoby pierwszej implementacji funkcji (tej prawidłowo dopasowanej) kompilator zgłosiłby błąd.

7.2 Szablon `enable_if`

Rozważmy dwie implementacje funkcji z identycznymi interfejsami, ale różnymi wymaganiami odnośnie typu danych:

```
template <typename T>
void do_stuff(T t) // for small objects
{
    //~~~
}

template <typename T>
void do_stuff(T const& t) // for large objects
{
    //~~~
}
```

Sygnatury tych funkcji są zbyt podobne, aby można było zastosować klasyczne przeciążenie funkcji. Rozwiązaniem tego problemu jest zastosowanie szablonu `enable_if` i mechanizmu SFINAE.

Typowa implementacja szablonu `enable_if` wygląda następująco:

```
template <bool Condition, typename T = void>
struct enable_if
{
    using type = T;
};

template <typename T>
struct enable_if<false, T>
{};
```

Szablon ogólny jest sparametryzowany wartością logiczną (zwykle wyrażenie logiczne, które jest ewaluowane na etapie kompilacji) oraz typem (domyślnie `void`). Wewnątrz struktury definiowany jest typ `type`. W wersji specjalizowanej dla wartości logicznej `false` takiej definicji typu brakuje.

- Jeśli wyrażenie `Condition` przekazane jako pierwszy argument szablonu ewaluowane jest do wartości `true`:
  - `enable_if<Condition>` ma składową `type`
  - i `enable_if<Condition>::type` odnosi się do tego typu
- Jeśli wyrażenie `Condition` ewaluowane jest do `false`:

- `enable_if<Condition>` nie ma składowej `type`
- i `enable_if<Condition>::type` jest nieprawidłowym odwołaniem
- w rezultacie podstawienie nie udaje się (działa SFINAE)

W celu ułatwienia korzystania z `enable_if` C++14 definiuje poniższy alias szablonu:

```
template <bool Condition, typename T = void>
using enable_if_t = typename enable_if<Condition, T>::type;
```

Wykorzystanie szablonu `enable_if` do problemu wyboru implementacji wygląda następująco:

```
#include <iostream>

template <typename T>
typename enable_if<(sizeof(T) < 8)>::type do_stuff(T t) // for small objects
{
    std::cout << "do_stuff(Small Object)\n";
}

template <typename T>
enable_if_t<(sizeof(T) > 8)> do_stuff(T const& t) // for large objects
{
    std::cout << "do_stuff(Large Object)\n";
}
```

```
#include <string>

do_stuff('A'); // OK, small object
do_stuff(std::string("abc")); // OK, large object
```

```
do_stuff(Small Object)
do_stuff(Large Object)
```

W zależności od rozmiaru typu wydedukowanego w procesie tworzenia instancji szablonu jest tworzona i później wywoływana odpowiednia instancja szablonu funkcji `do_stuff()`.

Próba utworzenia instancji dla typu `double` zgłaszany jest błąd kompilacji:

```
do_stuff(3.14);
```

```
error: no matching function for call to do_stuff
    note: candidate template ignored: disabled by enable_if [with T = double]
```

Kompilator nie jest w stanie wygenerować żadnej instancji szablonu dla typu o rozmiarze 8 bajtów.

Mechanizm SFINAE zapewnia dwufazowe dopasowanie funkcji do wywołania (*two-phase overload resolution*):

1. W fazie pierwszej `enable_if` i SFINAE eliminują funkcje kandydujące, dla których nie można zrealizować podstawienia
2. W fazie drugiej dopasowana może zostać tylko jedna funkcja z grupy funkcji kandydujących do wywołania

7.3 Cechy typów i `enable_if`

Często jako w wyrażeniu logicznym, będącym pierwszym argumentem szablonu `enable_if`, wykorzystywane są cechy typów:

```
#include <type_traits>

template <typename T>
std::enable_if_t<std::is_pod<T>::value> store_item(T const& item)
{
    // do something when T is POD
}

template <typename T>
std::enable_if_t<!std::is_pod<T>::value> store_item(T const& item)
{
    // do something when T is not POD
}
```

SFINAE może zostać użyte również do wprowadzenia ograniczeń jeśli chodzi i zakres typów, dla których realizowane jest tworzenie instancji szablonu:

```
struct Shape
{
    //...
};

struct Rectangle : Shape
{
    //...
};
```

```
template <typename T>
enable_if_t<std::is_base_of<Shape, T>::value> do_stuff_with_shape(T const& shape)
{
    //...
}
```

7.4 Domyślne argumenty szablonów funkcji

W C++11 argumenty szablonów funkcji mogą przyjmować wartości domyślne.

Pozwala to na czytelniejszą dla programisty implementację SFINAE z `enable_if`:

```
template <
    typename T,
    typename = std::enable_if_t<std::is_base_of<Shape, T>::value>
>
void do_other_stuff_with_shape(T const& shape)
{
    //...
}
```

Aby podnieść jeszcze bardziej czytelność kodu możemy zdefiniować pomocniczy alias:

```
template <typename T>
using IsShape = std::enable_if_t<std::is_base_of<Shape, T>::value>;
```

i wykorzystać go do implementacji szablonu funkcji z ograniczeniem SFINAE:

```
template <
    typename T,
    typename = IsShape<T>
>
void do_other_stuff_with_shape_alt(T const& shape)
{
    //...
}
```

7.5 Ograniczenia w szablonach klas

SFINAE oraz `enable_if` mogą również zostać użyte dla szablonów klas.

Możemy na przykład ograniczyć możliwość tworzenia instancji szablonów dla typów zmiennoprzecinkowych:

```
template <typename T>
using FloatingPoint = std::enable_if_t<std::is_floating_point<T>::value>;

template <
    typename T,
    typename = FloatingPoint<T>
>
class Data
{
    //...
};
```

```
Data<double> d1;
```

```
(Data<double> &) @0x7faca2165022
```

```
Data<int> d2;
```

```
error: no type named type in std::enable_if<false, void>
```

7.6 SFINAE i przeciążone konstruktory

SFINAE może rozwiązać problemy związane z przeciążonymi konstruktorami klas i ich czasami zaskakującym zachowaniem.

Załóżmy, że implementujemy klasę `Heap`, która potrzebuje dwóch wersji konstruktorów:

```
#include <iostream>

template <typename T>
class Heap
{
public:
    Heap(size_t n, T const& v)
    {
        std::cout << "Heap(size_t, T)\n";
    }

    template <typename InIt>
    Heap(InIt start, InIt end)  // range init using pair of input iterators
    {
        std::cout << "Heap(InIt, InIt)" << std::endl;
    }
};
```

Obecność konstruktora definiowanego jako szablon może dać zaskakujący rezultat:

```
Heap<int> h(5, 0); // range init constructor called
```

```
Heap(InIt, InIt)
```

Powyższe kod spowoduje wywołanie konstruktora przyjmującego jako argumenty parę iteratorów, ponieważ ta wersja konstruktora gwarantuje lepsze dopasowanie argumentów.

Możemy uniknąć takiego zachowania wprowadzając ograniczenie dla konstruktora wykorzystującego iteratory:

```
#include <iterator>

template <typename It>
using Category = typename std::iterator_traits<It>::iterator_category;

template <typename It>
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
using InputIterator = std::is_base_of<std::input_iterator_tag, Category<It>>;

template <typename It>
using IsInputIterator_t = std::enable_if_t<InputIterator<It>::value>;
```

```
namespace Sfinae
{
    template <typename T>
    class Heap
    {
    public:
        Heap(size_t n, T const& v)
        {
            std::cout << "Heap(size_t, T)\n";
        }

        template <
            typename InIt,
            typename = IsInputIterator_t<InIt>
        >
        Heap(InIt start, InIt end)  // range init using pair of input iterators
        {
            std::cout << "Heap(InIt, InIt)" << std::endl;
        }
    };
}
```

Teraz przy wywołaniu

```
Sfinae::Heap<int>(5, 0);
```

```
Heap(size_t, T)
```

konstruktor z iteratorami znika z funkcji kandydujących do wywołania.

Natomiast, gdy przekażemy konstruktorowi zakres zdefiniowany przez iteratory, to odpowiednia wersja konstruktora jest instancjonowana i później wywołana.

```
auto range = { 1, 2, 3 };

Sfinae::Heap<int>(range.begin(), range.end());
```

```
Heap(InIt, InIt)
```

8 | Variadic templates

W C++11 szablony mogą akceptować dowolną ilość (również zero) parametrów. Jest to możliwe dzięki użyciu specjalnego grupowego parametru szablonu tzw. *parameter pack*, który reprezentuje wiele lub zero parametrów szablonu.

8.1 Parameter pack

Parameter pack może być

- grupą parametrów szablonu

```
template<typename... Ts> // template parameter pack
class tuple
{
    //...
};

tuple<int, double, string&> t1; // 3 arguments: int, double, string&
tuple<> empty_tuple; // 0 arguments
```

- grupą argumentów funkcji szablonowej

```
template <typename T, typename... Args>
shared_ptr<T> make_shared(Args&&... params)
{
    //...
}

auto sptr = make_shared<Gadget>(10); // Args as template param: int
                                     // Args as function param: int&&
```

8.2 Rozpakowanie paczki parametrów

Podstawową operacją wykonywaną na grupie parametrów szablonu jest rozpakowanie jej za pomocą operatora ... (tzw. *pack expansion*).

```
template <typaname... Ts>  // template  parameter pack
struct X
{
    tuple<Ts...> data;  // pack expansion
};
```

Rozpakowanie paczki parametrów (*pack expansion*) może zostać zrealizowane przy pomocy wzorca zakończonego elipsą ...:

```
template <typaname... Ts>  // template  parameter pack
struct XPtrs
{
    tuple<Ts const*...> ptrs;  // pack expansion
};

XPtrs<int, string, double> ptrs; // contains tuple<int const*, string const*, double const*>
```

Najczęstszym przypadkiem użycia wzorca przy rozpakowaniu paczki parametrów jest implementacja **perfect forwarding'u**:

```
template <typename... Args>
void make_call(Args&&... params)
{
    callable(forward<Args>(params)...);
}
```

8.3 Idiom Head/Tail

Praktyczne użycie **variadic templates** wykorzystuje często idiom **Head/Tail** (znany również **First/Rest**).

Idiom ten polega na zdefiniowaniu wersji szablonu akceptującego dwa parametry: - pierwszego `Head` - drugiego `Tail` w postaci paczki parametrów

W implementacji wykorzystany jest parametr (lub argument) typu `Head`, po czym rekursywnie wywołana jest implementacja dla rozpakowanej paczki parametrów typu `Tail`.

Dla szablonów klas idiom wykorzystuje specjalizację częściową i szczegółową (do przerwania rekurencji):

```
template <typename... Types>
struct Count;

template <typename Head, typename... Tail>
struct Count<Head, Tail...>
{
    constexpr static int value = 1 + Count<Tail...>::value; // expansion pack
};
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
template <>
struct Count<>
{
    constexpr static int value = 0;
};

//...
static_assert(Count<int, double, string&>::value == 3, "must be 3");
```

W przypadku szablonów funkcji rekurencja może być przerwana przez dostarczenie odpowiednio przeciążonej funkcji. Zostanie ona w odpowiednim momencie rozwijania rekurencji wywołana.

```
void print()
{}

template <typename T, typename... Tail>
void print(const T& arg1, const Tail&... params)
{
    cout << arg1 << endl;
    print(params...); // function parameter pack expansion
}
```

8.4 Operator `sizeof...`

Operator `sizeof...` umożliwia odczytanie na etapie kompilacji ilości parametrów w grupie.

```
template <typename... Types>
struct VerifyCount
{
    static_assert(Count<Types...>::value == sizeof...(Types),
                  "Error in counting number of parameters");
};
```

8.5 Forwardowanie wywołań funkcji

Variadic templates są niezwykle przydatne do forwardowania wywołań funkcji.

```
template <typename T, typename... Args>
std::unique_ptr<T> make_unique(Args&&... params)
{
    return std::unique_ptr<T>(new T(forward<Args>(params)...);
}
```

8.6 Ograniczenia paczek parametrów

Klasa szablonowa może mieć tylko jedną paczkę parametrów i musi ona zostać umieszczona na końcu listy parametrów szablonu:

```
template <size_t... Indexes, typename... Ts>  // error
class Error;
```

Można obejść to ograniczenie w następujący sposób:

```
template <size_t... Indexes> struct IndexSequence {};

template <typename Indexes, typename Ts...>
class Ok;

Ok<IndexSequence<1, 2, 3>, int, char, double> ok;
```

Funkcje szablonowe mogą mieć więcej paczek parametrów:

```
template <int... Factors, typename... Ts>
void scale_and_print(Ts const&... args)
{
    print(ints * args...);
}

scale_and_print<1, 2, 3>(3.14, 2, 3.0f);  // calls print(1 * 3.14, 2 * 2, 3 * 3.0)
```

Uwaga! Wszystkie paczki w tym samym wyrażeniu rozpakowującym muszą mieć taki sam rozmiar.

```
scale_and_print<1, 2>(3.14, 2, 3.0f);  // error
```

8.7 „Nietypowe” paczki parametrów

- Podobnie jak w przypadku innych parametrów szablonów, paczka parametrów nie musi być paczką typów, lecz może być paczką stałych znanych na etapie kompilacji

```
template <size_t... Values>
struct MaxValue;  // primary template declaration

template <size_t First, size_t... Rest>
struct MaxValue<First, Rest...>
{
    static constexpr size_t rvalue = MaxValue<Rest...>::value;
    static constexpr size_t value = (First < rvalue) ? rvalue : First;
};

template <size_t Last>
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
struct MaxValue<Last>
{
    static constexpr size_t value = Last;  // termination of recursive expansion
};
```

```
static_assert(MaxValue<1, 5345, 3, 453, 645, 13>::value == 5345, "Error");
```

8.8 Variadic Mixins

Variadic templates mogą być skutecznie wykorzystane do implementacji klas **mixin**

```
#include <vector>
#include <string>

template <typename... Mixins>
class X : public Mixins...
{
public:
    X(Mixins&&... mixins) : Mixins(mixins)...
    {}
};
```

Klasa taka może zostać wykorzystana później w następujący sposób:

```
X<std::vector<int>, std::string> x({ 1, 2, 3 }, "text");

x.std::string::size();
```

```
(unsigned long) 4
```

```
x.std::vector<int>::size();
```

```
(unsigned long) 3
```

8.9 Curiously-Recurring Template Parameter (CRTP)

Ciekawym sposobem użycia **variadic templates** jest implementacja znanego idiomu **CRTP**.

Zaimplementujmy najpierw klasę licznika obiektów:

```
template <typename T>
class Counter
{
public:
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    Counter() { ++counter_; }
    ~Counter() { --counter_; }
    static size_t count() { return counter_; }
private:
    static size_t counter_;
};

template <typename T> size_t Counter<T>::counter_;
```

Klasa `Counter` może zostać wielokrotnie użyta do implementacji zliczania obiektów za pomocą idiomu **CRTP**. Dziedzicząc typ `T` po `Counter<T>` wstrzykujemy do `T` implementację zliczania obiektów.

```
class Thing : public Counter<Thing> // OK
{
};
```

```
Thing thing1, thing2;
{
    Thing thing3;
}
Thing::count();
```

```
(unsigned long) 2
```

Innym praktycznym zastosowaniem **CRTP** jest implementacja operatorów porównań dla klas.

- Operatory `==` oraz `!=` mogą zostać zaimplementowane za pomocą funkcji pomocniczej `equal_to()`.
- Operatory `<`, `<=`, `>`, `>=` mogą zostać zaimplementowane za pomocą funkcji `equal_to()`, `less_than()` oraz `greater_than()`
- Klasy szablonowe `Eq<T>` oraz `Rel<T>` implementują generyczne operatory porównań wykorzystujące wyżej wymienione funkcje pomocnicze (operatory te są zadeklarowane jako zaprzyjaźnione – `friend`)

```
template <typename T>
class Eq
{
    friend bool operator==(T const& a, T const& b) { return a.equal_to(b); }
    friend bool operator!=(T const& a, T const& b) { return !a.equal_to(b); }
};

template <typename T>
class Rel
{
    friend bool operator<(T const& a, T const& b) { return a.less_than(b); }
    friend bool operator<=(T const& a, T const& b) { return !b.less_than(a); }
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    friend bool operator>(T const& a, T const& b) { return b.less_than(a); }
    friend bool operator>=(T const& a, T const& b) { return !a.less_than(b); }
};
```

```
struct AnotherThing : public Eq<AnotherThing>
{
    int value;

    AnotherThing(int value) : value{value}
    {}

    bool equal_to(AnotherThing const& other) const
    {
        return value == other.value;
    }
};
```

```
AnotherThing at1{1};
AnotherThing at2{2};

(at1 != at2);
```

```
(bool) true
```

```
(at1 == at2);
```

```
(bool) false
```

Użycie szablonu jako parametru szablonu upraszcza stosowanie **CRTP**

```
template <template <typename> class CRTP>
struct OtherThing : public CRTP<OtherThing<CRTP>>
{
    int value;

    OtherThing(int value = 0) : value{value}
    {}

    bool equal_to(OtherThing const& other) const
    {
        return value == other.value;
    }

    bool less_than(OtherThing const& other) const
    {
        return value < other.value;
    }
};
```

```
OtherThing<Counter> counted_thing;
OtherThing<Eq> equality_comparable_thing;
OtherThing<Rel> relational_thing;
```

Jeżeli chcemy użyć wielu implementacji **CRTP** (np. `Thing<Counter, Eq>`) dla jednej klasy musimy wykorzystać **variadic templates**:

```
template <template <typename> class... CRTPs>
struct SuperThing : public CRTPs<SuperThing<CRTPs...>>...
{
    int value;

    SuperThing(int value = 0) : value{value}
    {}

    bool equal_to(SuperThing const& other) const
    {
        return value == other.value;
    }

    bool less_than(SuperThing const& other) const
    {
        return value < other.value;
    }
};
```

```
SuperThing<Counter, Eq, Rel> a_thing{10};
SuperThing<Counter, Eq, Rel> b_thing{20};

a_thing.count();

std::cout << (a_thing < b_thing) << std::endl;
```

W implementacji klasy szablonowej `SuperThing`:

- `SuperThing` jest nazwą szablonu
- `CRTPs` jest szablonową paczką parametrów szablonu (*template template parameter pack*)
- `SuperThing<CRTPs...>` jest nazwą typu
- `CRTPs<SuperThing<CRTPS...>>` jest wzorcem paczki parametrów
- `CRTPs<SuperThing<CRTPS...>>...` jest rozwinięciem wzorca
- `public CRTPs<SuperThing<CRTPs...>>...` jest listą klas bazowych

9 | Krotki w C++

9.1 Krotki – motywacja

Chcemy napisać funkcję zwracającą trzy wartości statystyk dla wektora liczb całkowitych:

- wartość minimalną
- wartość maksymalną
- wartość średnią

Problem – funkcja może zwrócić tylko jedną wartość

Rozwiązanie 1 – przekazanie parametrów przez referencję

```
void calculate_stats(const vector<int>& data, int& min, int& max, double& avg)
{
    min = *min_element(data.begin(), data.end());
    max = *max_element(data.begin(), data.end());
    avg = accumulate(data.begin(), data.end(), 0.0) / data.size();
}
```

Rozwiązanie 2 – użycie `std::tuple`

```
std::tuple<int, int, double> calculate_stats(const vector<int>& data)
{
    auto min = *min_element(data.begin(), data.end());
    auto max = *max_element(data.begin(), data.end());
    auto avg = accumulate(data.begin(), data.end(), 0.0) / data.size();

    return std::tuple<int, int, double>(min, max, avg);
}


// Przykładowe użycie
vector<int> data = { 5, 1, 35, 321, 23, 5, 9, 88, 44, 324 };

auto stats = calculate_stats(data);

cout << "Min: " << get<0>(stats)
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    << "; Max: " << get<1>(stats)
    << "; Avg: " << get<2>(stats) << endl;
```

9.2 Konstruowanie krotek

Konstrukcja obiektu typu `std::tuple` wymaga deklaracji typu i (opcjonalnego) udostępnienia listy wartości początkowych, zgodnych co do typu z typami poszczególnych elementów montowanej krotki.

Konstruktor domyślny krotki inicjalizuje wartościowo każdy element (*value initialized*).

```
tuple<int, double, string> triple(42, 3.14, "Test krotki");
tuple<short, string> another;  // default value for every element
```

Funkcja `make_tuple(wart1, wart2, ...)` tworzy krotkę, dedukując typy elementów na podstawie typów argumentów wywołania.

```
tuple<int, double> get_values()
{
    return std::make_tuple(3, 5.56);
}
```

9.3 Krotki z referencjami

Funkcja `make_tuple()` określa domyślnie typy elementów jako modyfikowalne i nierefe-rencyjne.

```
void foo(const A& a, B& b)
{
    //...
    make_tuple(a, b); // tworzy tuple<A, B>
    //...
}
```

Aby elementy krotki były referencjami należy skorzystać z wrapperów `std::ref()` oraz `std::cref()`

```
A a; B b; const A ca = a;

make_tuple(cref(a), b); // tworzy tuple<const A&, B>

make_tuple(ref(a), b); // tworzy tuple<A&, B>

make_tuple(ref(a), cref(b)); // tworzy tuple<A&, const B&>

make_tuple(cref(ca)); // tworzy tuple<const A&>
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
make_tuple(ref(ca)); // tworzy tuple<const A&>
```

9.4 Odwołania do elementów krotek

Elementy krotki są dostępne przy pomocy zewnętrznej funkcji `get<Index>()` zwracającej referencję do odpowiedniego elementu krotki.

```
double d = 2.7; A a;

tuple<int, double&, const A&> t(1, d, a);
const tuple<int, double&, const A&> ct = t;

int i = get<0>(t);
int j = get<0>(ct); // ok
get<0>(t) = 5; // ok
get<0>(ct) = 5; // błąd! nie można przypisać do const
double e = get<1>(t); // ok
get<1>(t) = 3.14; // ok
get<2>(t) = A(); // błąd! nie można przypisać do const
A aa = get<3>(t); // błąd! indeks poza zakresem ...
++get<0>(t); // ok, można używać jak zmiennej
```

9.5 Przypisywanie i kopiowanie krotek

Krotki można przypisywać i kopiować (wywołując konstruktor kopiujący) pod warunkiem, że dla odpowiednich typów krotki źródłowej i docelowej dostępne są wymagane konwersje.

```
class A {};
class B : public A {};
struct C { C(); C(const B&); };
struct D { operator C() const; };
...
tuple<char, B*, B, D> t;
...
tuple<int, A*, C, C> a(t); // ok
a = t; // ok
```

9.6 Porównywanie krotek

Krotki redukują operatory ==, !=, <, >, <= i >= do odpowiadających im operatorów typów przechowywanych w krotce.

Równość pomiędzy krotkami a i b jest zdefiniowana przez:

- `a == b` jeśli dla każdego `i`: ‘‘a[i] == b[i] ‘‘
- `a != b` jeśli istnieje `i`: `a[i] != b[i]`

Operatory <, >, <= oraz >= implementują porównywanie leksykograficzne.

```
tuple<string, int, A> t1("same?", 2, A());
tuple<string, long, A> t2("same?", 2, A());

t1 == t2; // true
```

9.7 Wiązanie zmiennych w krotki

Funkcja szablonowa `std::tie` umożliwia wiązanie samodzielnych zmiennych w krotki. Wszystkie elementy krotki utworzonej przez `std::tie` są modyfikowalnymi referencjami.

```
vector<int> data = { 5, 1, 35, 321, 23, 5, 9, 88, 44, 324 };

int min, max;
double avg;

tie(min, max, avg) = calculate_stats(data);

cout << "Min: " << min << "; Max: " << max << "; Avg: " << avg << endl;
```

Wyjście:

```
1 324 85.5
```

Mechanizm wiązania działa również z obiektami szablonu `std::pair<T>`.

Algorytm `std::minmax_element()` zwraca parę iteratorów wskazujących na wystąpienia najmniejszej i największej wartości w podanym zakresie. Zwracaną parę można przypisać do zmiennych za pomocą funkcji `std::tie()`

```
std::vector<int>::iterator min_it, max_it;
tie(min_it, max_it) = std::minmax_element(data.begin(), data.end());
```

Obiekt `std::ignore` umożliwia ignorowanie elementu krotki przy operacji wiązania zmiennych

```
int min, max;

tie(min, max, ignore) = calculate_stats(data);
```

9.8 Krotki – podsumowanie

- Krotki umożliwiają łatwe zwracanie wielu wartości z funkcji i wiązanie ich z samodzielnymi zmiennymi

- Upraszczają implementację operatorów porównań dla klas użytkownika

10 | Sekwencje indeksów

Sekwencje indeksów (**index sequences**)

- są wykorzystywane do meta-programowania z wykorzystaniem szablonów
- są używane w połączeniu z *variadic templates* (w szczególności z możliwością rozwijania paczek parametrów)
- umożliwiają prostszą implementację meta-algorytmów

10.1 Wybieranie elementów z krotki

Chcemy napisać szablon funkcji, który umożliwi selekcję elementów krotki w następujący sposób:

```
auto existing = make_tuple(0, 1.0, '2', "three");
auto selected = select<1, 0, 3, 1, 2, 0>(existing);
```

Aby tego dokonać będziemy chcieli wywołać operację `get<I>` dla każdego indeksu z listy parametrów szablonu. Argumentem wywołania funkcji będzie krotka.

```
#include <tuple>

template <size_t... Indexes, typename... Ts>
auto select(std::tuple<Ts...> const& tpl)
{
    return std::make_tuple(std::get<Indexes>(tpl)...);
}
```

W implementacji wykorzystaliśmy wzorzec `get<Indexes>(tpl)`. Wzorzec ten rozwinięty jako `get<Indexes>(tpl)...` skutkuje przekazaniem listy elementów do funkcji `std::make_tuple()`.

```
auto existing = std::make_tuple(0, 1.0, '2', "three");
auto selected = select<1, 3>(existing);

std::get<1>(selected);
```

```
"three"
```

10.2 Sekwencje indeksów w C++14

Sekwencje indeksów są często wykorzystywane w procesie rozwijania paczek parametrów szablonu.

C++14 dostarcza zestaw gotowych narzędzi do generowania własnych sekwencji indeksów:

```
template <typename T, T...> struct integer_sequence;

template<size_t... I>
using index_sequence = integer_sequence<int, I...>;

template <size_t N>
using make_index_sequence = make_integer_sequence<size_t, N>;
```

10.2.1 Uproszczona implementacja sekwencji indeksów w C++11

```
template <size_t... Seq>
struct IndexSequence
{
    typedef std::index_sequence<Seq...> type;
    typedef size_t value_type;

    static constexpr size_t size() noexcept
    {
        return sizeof...(Seq);
    }
};

template <size_t N, size_t I, typename Seq>
struct MakeSequence;

template <size_t N, size_t... Seq>
struct MakeSequence<N, N, IndexSequence<Seq...>>  // ends recursion at N
    : IndexSequence<Seq...>
{};

template <size_t N, size_t I, size_t... Seq>
struct MakeSequence<N, I, IndexSequence<Seq...>>
    : MakeSequence<N, I+1, IndexSequence<Seq..., I>>
{};

template <size_t N>
using MakeIndexSequence = typename MakeSequence<N, 0, IndexSequence<>>::type;
```

```
#include <type_traits>

std::is_same<std::index_sequence<0, 1, 2, 3, 4>, MakeIndexSequence<5>>::value;
```

```
(const bool) true
```

10.3 Zastosowanie sekwencji indeksów

Typowe zastosowanie sekwencji indeksów polega na rozpakowaniu odpowiedniego wzorca dla **variadic templates**. Algorytm postępowania zwykle wygląda następująco:

1. Utworzenie odpowiedniej sekwencji.

```
using I = std::make_index_sequence<N>; // [0; N)
```

2. Utworzenie obiektu dla którego chcemy wykorzystać sekwencję indeksów

```
std::array<int , N * 2> a;
```

3. Ekstrakcja sekwencji indeksów z `index_sequence` i aplikacja ich we wzorcu **variadic template**. Proces ekstrakcji może wykorzystywać:

   - mechanizm dedukcji typów dla argumentów szablonu:

```
#include <array>
#include <algorithm>
#include <iostream>

namespace Impl1
{
    template <typename T, size_t N, size_t... I>
    auto select_evens_impl(std::array<T, N> const& a, std::index_sequence<I...>)
    {
        return std::array<T, sizeof...(I)>{ a[I * 2]...};
    }

    template <typename T, size_t N>
    auto select_evens(std::array<T, N> const& a)
    {
        using Indexes = std::make_index_sequence<N/2>;
        return select_evens_impl(a, Indexes{});
    }
}
```

```
std::array<int, 11> a = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
auto even_1 = Impl1::select_evens(a);

for(const auto& item : even_1)
    std::cout << item << " ";
```

```
0 2 4 6 8
```

   - lub mechanizm częściowej specjalizacji szablonów

```
#include <array>
#include <algorithm>
#include <iostream>

namespace Impl2
{
    template <typename T, size_t N, typename Seq>
    struct SelectEvens;

    template <typename T, size_t N, size_t... I>
    struct SelectEvens<T, N, std::index_sequence<I...>>
    {
        static auto apply(std::array<T, N> const& a)
        {
            return std::array<T, sizeof...(I)>{ a[I * 2]...};
        }
    };


    template <typename T, size_t N>
    auto select_evens(std::array<T, N> const& a)
    {
        using Indexes = std::make_index_sequence<N/2>;

        return SelectEvens<T, N, Indexes>::apply(a);
    }
}
```

```
auto even_2 = Impl2::select_evens(a);

for(const auto& item : even_2)
    std::cout << item << " ";
```

```
0 2 4 6 8
```

10.4 Meta-programowanie z użyciem krotek

10.4.1 Implementacja foreach dla krotki

Chcemy napisać funkcję umożliwiającą wywołanie funkcji, obiektu funkcyjnego lub lambdy dla każdego elementu krotki.

Przykład użycia mógłby wyglądać następująco:

```
auto printer = [](auto const& item) { std::cout << "value: " << item << std::endl; };
auto tpl = std::make_tuple(1, 3.14, 'a', "text"s);

tuple_apply(printer, tpl);
```

Implementacja wymaga zastosowania sekwencji indeksów.

```
#include <utility>
#include <tuple>
#include <string>
#include <iostream>

template <typename F, typename T, size_t... I>
void apply_tuple_impl(F f, T const& t, std::index_sequence<I...>)
{
    std::initializer_list<bool> { (f(std::get<I>(t)), false)... };
}

template <typename F, typename... Ts>
inline void tuple_apply(F f, std::tuple<Ts...> const& t)
{
    using Indexes = std::make_index_sequence<sizeof...(Ts)>;

    apply_tuple_impl(f, t, Indexes{});
}
```

Implementacja wykorzystuje idiom z `std::initializer_list`, który gwarantuje prawidłową kolejność wywołania funkcji.

```
using namespace std::literals;

auto printer = [](auto const& item) {
    std::cout << "value: " << item << std::endl;
};

auto tpl = std::make_tuple(1, 3.14, 'a', "text"s);

tuple_apply(printer, tpl);
```

```
value: 1
value: 3.14
value: a
value: text
```

Powyższy kod jest uproszczoną implementacją funkcji `std::apply()` wprowadzonej w C++14:

```
<template <typename F, typename Tuple, size_t... I>
auto apply_impl(F&& f, Tuple&& t, std::index_sequence<I...>)
{
    return std::forward<F>(f)(std::get<I>(std::forward<Tuple>(t))...);
}

template <typename F, class Tuple>
auto apply(F&& f, Tuple&& t)
{
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    using Indexes = std::make_index_sequence<std::tuple_size<Tuple>::value>;

    return std::apply_impl(std::forward<F>(f), std::forward<Tuple>(t), Indexes{});
}
```

10.4.2 Konwersja tablicy do krotki

Załóżmy, że mamy tablicę zdefiniowaną jako:

```
std::array<std::string, 10> a = { "one", "two", "three" };
```

Chcemy skonwertować ją do krotki przy pomocy następującej funkcji:

```
auto tpl = array2tuple(a); // tpl is a tuple<std::string, std::string, etc.>
```

Implementacja funkcji `array2tuple()` polega na: * utworzeniu sekwencji indeksów dla każdego elementu tablicy * użyciu sekwencji do rozwinięcia wzorca **variadic template** w wywołaniu `make_tuple()`

```
template <typename Array, size_t... I>
inline constexpr auto array2tuple_impl(Array const& a, std::index_sequence<I...>)
{
    return std::make_tuple(a[I]...);
}

template<typename T, size_t N, typename Indexes = std::make_index_sequence<N>>
inline constexpr auto array2tuple(std::array<T, N> const& a)
{
    return array2tuple_impl(a, Indexes{});
}
```

```
std::array<int, 3> arr = { 1, 2, 3 };
array2tuple(arr);
```

```
(std::tuple<int, int, int>) { 1, 2, 3 }
```

10.4.3 Konwersja krotki do tablicy

```
#include <utility>
#include <tuple>
#include <iostream>

template <typename... Ts, size_t... I>
inline auto tuple2array_impl(std::tuple<Ts...> const& t, std::index_sequence<I...>)
{
    using TT = std::common_type_t<Ts...>;
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    return std::array<TT, sizeof...(Ts)> { static_cast<TT>(std::get<I>(t))...};
}

template <typename... Ts, typename Indexes = std::make_index_sequence<sizeof...(Ts)>>
inline auto tuple2array(std::tuple<Ts...> const& t)
{
    return tuple2array_impl(t, Indexes{});
}
```

```
auto t4 = std::make_tuple(1, -1L, 3.14);
auto a3 = tuple2array(t4);

for(const auto& item : a3)
    std::cout << item << " ";
```

```
1 -1 3.14
```

10.5 Operacje na sekwencjach indeksów

10.5.1 Przesunięcie indeksów

Zaimplementujmy operację przesunięcia wartości indeksów o zadaną wartość (poprzez dodanie tej wartości do każdego elementu sekwencji).

```
#include <utility>

template <size_t Shift, typename Seq>
struct ShiftSequence;

template <size_t Shift, size_t... Seq>
struct ShiftSequence<Shift, std::index_sequence<Seq...>>
{
    using type = std::index_sequence<(Shift + Seq)...>;
};
```

```
#include <type_traits>

using OriginalSeq = std::make_index_sequence<5>;
using ShiftedSeq = ShiftSequence<5, OriginalSeq>::type;

static_assert(std::is_same<ShiftedSeq, std::index_sequence<5, 6, 7, 8, 9>>::value, "Error");
```

Zgodnie z konwencją wprowadzoną w C++14 tworzymy dodatkowo alias zakończony `_t`, który upraszcza wywoływanie metafunkcji transformujących typy.

```
template <size_t Shift, typename Seq>
using ShiftSequence_t = typename ShiftSequence<Shift, Seq>::type;

using OriginalSeq = std::make_index_sequence<5>;
using ShiftedSeq = ShiftSequence_t<5, OriginalSeq>;

static_assert(std::is_same<ShiftedSeq, std::index_sequence<5, 6, 7, 8, 9>>::value, "Error");
```

10.5.2 Transformacja indeksów

Możemy zaimplementować meta-algorytm będący odpowiednikiem `std::transform()` z biblioteki STL.

Jako meta-funkcji transformującej na etapie kompilacji możemy użyć szablonowego parametru szablonu (*template template parametr*).

```
template <typename Seq, template <size_t> class Op>
struct Transform;

template <template <size_t> class Op, size_t... I>
struct Transform<std::index_sequence<I...>, Op>
{
    using type = std::index_sequence<Op<I>::value...>;
};

template <typename Seq, template <size_t> class Op>
using Transform_t = typename Transform<Seq, Op>::type;
```

Użycie meta-algorytmu `Transform_t` wygląda następująco:

```
template <size_t N>
struct Square
{
    static constexpr size_t value = N * N;
};


using OriginalSeq = std::make_index_sequence<5>;
using ShiftedByOneSeq = ShiftSequence_t<1, OriginalSeq>;
using NewSeq = Transform_t<ShiftedByOneSeq, Square>;

static_assert(std::is_same<NewSeq, std::index_sequence<1, 4, 9, 16, 25>>::value, "Error");
```

10.5.3 Generowanie sekwencji

Meta-algorytm będą odpowiednikiem STL-owego `generate_n()` może być zaimplementowany w następujący sposób:

```
template <size_t N, template <size_t> class Gen>
struct Generate_n
{
    using type = Transform_t<std::make_index_sequence<N>, Gen>;
};

template <size_t N, template <size_t> class Gen>
using Generate_n_t = typename Generate_n<N, Gen>::type;

using Squares = Generate_n_t<5, Square>;
static_assert(std::is_same<Squares, std::index_sequence<0, 1, 4, 9, 16>>::value, "Error");
```

10.5.4 Modyfikowanie indeksów za pomocą rekursji

Odwracanie sekwencji

Niektóre z meta-algorytmów wymagają rekurencyjnej implementacji z wykorzystaniem idiomu **Head/Tail**.

Zaimplementujmy algorytm odwracający sekwencję indeksów:

```
template <typename Seq>
struct Reverse;

template <typename Seq>
using Reverse_t = typename Reverse<Seq>::type;

template <typename Seq, size_t Item>
struct PushBack;

template <typename Seq, size_t Item>
using PushBack_t = typename PushBack<Seq, Item>::type;

template <size_t... I, size_t Item>
struct PushBack<std::index_sequence<I...>, Item>
{
    using type = std::index_sequence<I..., Item>;
};

template <size_t First, size_t... Rest>
struct Reverse<std::index_sequence<First, Rest...>>
{
    using Tail = std::index_sequence<Rest...>;
    using Rtail = Reverse_t<Tail>;
    using type = PushBack_t<Rtail, First>;
};

template <>
struct Reverse<std::index_sequence<>>
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
{
    using type = std::index_sequence<>;
};
```

```
using ReversedSquares = Reverse_t<Squares>;
static_assert(std::is_same<ReversedSquares, std::index_sequence<16, 9, 4, 1, 0>>::value, "Error
↪");
```

Usuwanie elementów sekwencji

Zaimplementujmy meta-algorytm usuwający z sekwencji indeksów indeksy spełniające predykat przekazany za pomocą meta-funkcji:

```
template <typename Seq, template <size_t> class Pred>
struct RemoveIf;

template <typename Seq, template <size_t> class Pred>
using RemoveIf_t = typename RemoveIf<Seq, Pred>::type;

template <typename Seq, size_t Item>
struct PushFront;

template <typename Seq, size_t Item>
using PushFront_t = typename PushFront<Seq, Item>::type;

template <size_t... I, size_t Item>
struct PushFront<std::index_sequence<I...>, Item>
{
    using type = std::index_sequence<Item, I...>;
};

template <template <size_t> class Pred, size_t First, size_t... Rest>
struct RemoveIf<std::index_sequence<First, Rest...>, Pred>
{
    using Tail = RemoveIf_t<std::index_sequence<Rest...>, Pred>;
    using type = std::conditional_t<Pred<First>::value, Tail, PushFront_t<Tail, First>>;
};

template <template <size_t> class Pred>
struct RemoveIf<std::index_sequence<>, Pred>
{
    using type = std::index_sequence<>;
};
```

```
template <size_t I>
struct IsOdd
{
    static constexpr bool value = I % 2;
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
};
```

```
using Indexes = std::make_index_sequence<10>;
using Evens = RemoveIf_t<Indexes, IsOdd>;

static_assert(std::is_same<Evens, std::index_sequence<0, 2, 4, 6, 8>>::value, "Error");
```

10.6 Rozwijanie wielu paczek parametrów

Szablony klas mają dość istotne ograniczenie w postaci możliwości zdefiniowania tylko jednej paczki argumentów na liście argumentów szablonu. Rozwiązanie tego typu ograniczenia polega na zastosowaniu zagnieżdżonych, częściowo specjalizowanych szablonów w celu rozpakowania kolejnych paczek argumentów. Każdy poziom zagnieżdżenia rozpakowuje pojedynczą paczkę argumentów. Na najniższym poziomie zagnieżdżenia dostępna jest zawartość wszystkich paczek.

Implementacja podana poniżej ilustruje tę technikę dla sekwencji indeksów, ale łatwo może być uogólniona dla dowolnych paczek argumentów.

```
#include <utility>

template <typename Seq1, typename Seq2, size_t... Seq3>
struct Merge3
{
    template <typename> struct Do1;

    template <size_t... DoSeq1>
    struct Do1<std::index_sequence<DoSeq1...>>
    {
        // we have Seq1 & Seq3 here

        template <typename> struct Do2;

        template <size_t... DoSeq2>
        struct Do2<std::index_sequence<DoSeq2...>>
        {
            // we have Seq1, Seq2 & Seq3 here
            using type = std::index_sequence<DoSeq1..., DoSeq2..., Seq3...>;
        };

        using type = typename Do2<Seq2>::type;
    };

    using type = typename Do1<Seq1>::type;
};
```

Łączenie ze sobą sekwencji indeksów może zostać zaimplementowane za pomocą techniki zagnieżdżonego rozpakowywania paczek:

```
template <typename... IndexSequences>
struct Concatenate;

template <typename... IndexSequences>
using Concatenate_t = typename Concatenate<IndexSequences...>::type;

template <typename FirstSeq, typename... RestSeq>
struct Concatenate<FirstSeq, RestSeq...>
{
    template <typename HeadIndexes>
    struct First;

    template <size_t... HeadIndexes>
    struct First<std::index_sequence<HeadIndexes...>>
    {
        // recurse to concatenate tail sequences...
        using RestType = Concatenate_t<RestSeq...>;

        template <typename TailIndexes> struct Rest;
        template <size_t... TailIndexes>
        struct Rest<std::index_sequence<TailIndexes...>>
        {
            // extract merged tail, prepend first
            using type = std::index_sequence<HeadIndexes..., TailIndexes...>;
        };

        // ... using pass it up
        using type = typename Rest<RestType>::type;

    };

    using type = typename First<FirstSeq>::type;
};

template <>
struct Concatenate<>
{
    using type = std::index_sequence<>;
};
```

Wykorzystanie meta-algorytmu `Concatenate_t`:

```
using IncreasingSeq = std::make_index_sequence<10>;
using DecreasingSeq = Reverse_t<IncreasingSeq>;

using Doppler = Concatenate_t<IncreasingSeq, DecreasingSeq>;
```

11 | Uogólnione stałe wyrażenia - constexpr

C++11 wprowadza dwa znaczenia dla „stałej”:

- `constexpr` – stała ewaluowana na etapie kompilacji
- `const` – stała, której wartość nie może ulec zmianie

Stałe wyrażenie (**constant expression**) jest wyrażeniem ewaluowanym przez kompilator na etapie kompilacji. Nie może zawierać wartości, które nie są znane na etapie kompilacji i nie może mieć efektów ubocznych.

Jeśli wyrażenie inicjalizujące dla `constexpr` nie będzie mogło być wyliczone na etapie kompilacji kompilator zgłosi błąd:

```
int x1 = 7;  // variable
constexpr int x2 = 7;  // constant at compile-time

constexpr int x3 = x1; // error: initializer is not a contant expression

constexpr auto x4 = x2; // Ok
```

W wyrażeniu `contexpr` można użyć:

- Wartości typów całkowitych, zmiennoprzecinkowych oraz wyliczeniowych
- Operatorów nie modyfikujących stanu (np. +, ? i [] ale nie = lub ++)
- Funkcji `constexpr`
- Typów literalnych
- Stałych `const` zainicjowanych stałym wyrażeniem

11.1 Stałe wartości constexpr

W C++11 `constexpr` przed definicją zmiennej definiuje ją jako stałą, która musi zostać zainicjowana wyrażeniem stałym.

Stała `const` w odróżnieniu od stałej `constexpr` nie musi być zainicjowana wyrażeniem stałym.

```
constexpr int x = 7;
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
constexpr auto prefix = "Data";

constexpr double pi = 3.1415;

constexpr double pi_2 = pi / 2;
```

```
(const double) 1.570750
```

11.2 Funkcje constexpr

W C++11 funkcje mogą zostać zadeklarowane jako `constexpr` jeśli spełniają dwa wymagania:

- Ciało funkcji zawiera tylko jedną instrukcję `return` zwracającą wartość, która nie jest typu `void`
- Typ wartości zwracanej oraz typy parametrów powinny być typami dozwolonymi dla wyrażeń `constexpr`

C++14 znacznie poluzowuje wymagania stawiane przed funkcjami `constexpr`. Funkcją `constexpr` może zostać dowolna funkcja o ile:

- nie jest wirtualna
- typ wartości zwracanej oraz typy parametrów są typami literalnymi
- zmienne użyte wewnątrz funkcji są zmiennymi typów literalnych
- nie zawiera instrukcji `asm`, `goto`, etykiet oraz bloków `try-catch`
- zmienne użyte wewnątrz funkcji nie są statyczne oraz nie są `thread-local`
- zmienne użyte wewnątrz funkcji są zainicjowane

Funkcje ‘‘constexpr‘‘ nie mogą mieć żadnych efektów ubocznych. Zapisywanie stanu do nielokalnych zmiennych jest błędem kompilacji.

Przykład rekurencyjnej funkcji `constexpr`:

```
constexpr int factorial(int n)
{
    return (n == 0) ? 1 : n * factorial(n-1);
}
```

Funkcja `constexpr` może zostać użyta w kontekście, w którym wymagana jest stała ewaluowana na etapie kompilacji (np. rozmiar tablicy natywnej lub stała będąca parametrem szablonu):

```
#include <array>

const int size = 2;
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
int arr1[factorial(1)];
int arr2[factorial(size)];
std::array<int, factorial(3)> arr3;
```

```
template <typename T, size_t N>
constexpr size_t size_of_array(T(&)[N])
{
   return N;
}

int arr4[factorial(size_of_array(arr2))];
```

```
(int [2]) { 0, 0 }
```

11.2.1 Instrukcje warunkowe w funkcjach `constexpr`

Pominięty blok kodu w instrukcji warunkowej nie jest ewaluowany na etapie kompilacji.

```
constexpr int low = 0;
constexpr int high = 99;
```

```
#include <stdexcept>

constexpr int check(int i)
{
    return (low <= i && i < high) ? i : throw std::out_of_range("range error");
}
```

```
constexpr int val0 = check(50);  // Ok

constexpr int val2 = check(200);  // Error
```

11.3 Typy literalne

C++11 wprowadza pojęcie **typu literalnego** (*literal type*), który może być użyty w stałym wyrażeniu `constexpr`:

Typem literalnym jest:

- Typ arytmetyczny (całkowity, zmiennoprzecinkowy, znakowy lub logiczny)
- Typ referencyjny do typu literalnego (np: `int&`, `double&`)
- Tablica typów literalnych
- Klasa, która:
  - ma trywialny destruktor (może być `default`)

- wszystkie niestatyczne składowe i typy bazowe są typami literalnymi
- jest agregatem lub ma przynajmniej jeden konstruktor `contexpr`, który nie jest konstruktorem kopiującym lub przenoszącym (konstruktor musi mieć pustą implementację, ale umożliwia inicjalizację składowych na liście inicjalizującej)

```
class Complex
{
    double real_, imaginary_;
public:
    constexpr Complex(const double& real, const double& imaginary)
        : real_ {real}, imaginary_ {imaginary}
    {}

    constexpr double real() const { return real_; };
    constexpr double imaginary() const { return imaginary_; }
};
```

```
constexpr Complex c1 {1, 2};
```

11.4 Przykłady zastosowań wyrażeń i funkcji stałych (constexpr)

11.4.1 Operacje na polach bitowych

Interesującym zastosowaniem funkcji `constexpr` jest implementacja operatorów bitowych dla wyliczeń.

```
namespace Constexpr
{
    enum class Bitmask { b0 = 0x1, b1 = 0x2, b2 = 0x4 };

    constexpr Bitmask operator|(Bitmask left, Bitmask right)
    {
        return Bitmask( static_cast<int>(left) | static_cast<int>(right) );
    }
}
```

Umożliwia to, zastosowanie czytelnych wyrażeń bitowych np. w etykietach instrukcji `switch`:

```
#include <iostream>

using namespace std;
using namespace Constexpr;

Bitmask b = Bitmask::b0 | Bitmask::b1;

switch (b)
{
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    case Bitmask::b0 | Bitmask::b1:
        cout << "b0 | b1 - " << static_cast<int>(b) << endl;
        break;
    default:
        cout << "Other value...";
}
```

```
b0 | b1 - 3
```