Instrukcja IV

Wstęp

Na dzisiejszych zajęciach zajmiemy się szablonami (template'ami) klas oraz funkcji (od C++14 istnieją także szablony zmiennych, ale zaznajomienie się z nimi pozostawiamy dla chętnych). Template'y stanowią fundament C++ oraz są głównym powodem, dla którego język ten nie jest tylko "C z klasami" (choć stwierdzenie to można znaleźć w wielu miejscach w sieci). Ich obecność pozwala na pisanie generycznego kodu o maksymalnie szerokiej gamie zastosowań. Przykładem takiego podejścia jest sama biblioteka standardowa (STL - *Standard Template Library*), w której nie znajdziemy prawie żadnych funkcji i klas, lecz *szablony* funkcji i klas. Szablony definiujemy zgodnie z następującą składnią:

```
template< /* lista parametrów */ >
// Tutaj normalna definicja klasy/funkcji/aliasu/obiektu(C++14), wewnątr
```

Dalej możemy korzystać ze zdefiniowego szablonu w następujący sposób:

```
/* nazwa szablonu */ < /* konkretne argumenty zgodne z rodzajem zadeklar
// Powyższa linijka jest nazwą klasy/funkcji/etc., z której możemy korzy
```

Dzięki zastosowaniu template'ów możemy zdefiniować ciało danej klasy/funkcji/etc. tylko raz, a następnie instancjonować dany szablon dla dowolnych (zgodnych z deklaracją) parametrów, w zależności od potrzeby. Podkreślmy, że klasą/funkcją/etc. jest dopiero instancja szablonu, nie sam szablon. Proces instancjonowania template'ów odbywa się w czasie kompilacji, także możemy mieć pewność, że wykorzystanie tej funkcjonalności języka nie pociąga za sobą żadnego kosztu w wydajności programu. Pełne wprowadzenie do tego tematu czytelnik może znaleźć np. tutaj.

Rodzaje parametrów szablonów

Zanim napiszemy pierwszy szablon, powiedzmy, jakie może on w ogóle mieć rodzaje parametrów. Ich pełną listę możemy oczywiście znaleźć w dokumentacji, tutaj ograniczymy się do dwóch najważniejszych: typów oraz parametrów niebędących typami (tłumaczenie z angielskiego jest niestety mało wdzięczne).

Typ jako parametr

Szablony w C++ mogą być sparametryzowanane typami, zgodnie ze składnią

```
template<typename T /* ... */>
/* definicja szablonu */
```

Zamiast `typename` możemy zamiennie użyć `class`, `typename` jest jednak zgodne z powszechną konwencją. Wszędzie, gdzie w definicji danego sparametryzowanego bytu występuje typ, możemy teraz użyć `T`. Możemy zatem użyć `T` m.in. jako:

- typ pola klasy
- typ argumentu funkcji (w tym metody klasy)
- typ zwracany przez funkcję
- argument innego szablonu (template'y możemy dowolnie zagnieżdżać)

Wyobraźmy sobie teraz, że mamy napisać funkcję, która przyjmuje 2 liczby i zwraca ich sumę. Gdyby nie template'y, musielibyśmy pisać osobną funkcję dla `int`ów, `double`'i, `float`ów, `bool`i itd. Teraz wystarczy, że napiszemy jeden szablon, sparametryzowany typem argumentu i odpowiednio go zainstancjonujemy (jak dowiemy się niebawem jawne podanie parametrów szablonu nie będzie konieczne). Przykład ten stanowi jedynie wierzchołek góry lodowej zastosowań szablonów w C++.

NTTP

Parametrem szablonu może być także byt inny niż typ. W języku angielskim mówimy *non-type template parameter*, w dalszej części tego tekstu korzystać będziemy właśnie ze skrótu NTTP. NTTP mogą być:

- typy całkowite (`int`, `char`, `bool`, etc.)
- enumeracje (`enum`)
- referencje lvalue do obiektu lub funkcji (poza zakresem tej instrukcji)
- wskaźniki do obiektu lub funkcji (poza zakresem tej instrukcji)
- `std::nullptr_t` (poza zakresem tej instrukcji)
- od C++20 wymagania NTTP zostały poluzowane, dopuszczane są teraz także typy strukturalne i zmiennoprzecinkowe (zdecydowanie poza zakresem tej instrukcji)

NTTP dają szablonowi dostęp do wartości liczbowych (lub im podobnym) *w czasie kompilacji*. W konsekwencji w pełni legalna jest np. statyczna alokacja pamięci uzależniona od liczby całkowitej będącej parametrem szablonu (`int tab[N]` jest w tym kontekście dopuszczalne).

Szablony klas

Możemy teraz napisać nasz pierwszy szablon klasy. Zacznijmy od rzeczy trywialnych.

Zadanie 1 Napisz szablon klasy `Para`, który trzyma 2 obiekty typu, którym jest sparametryzowany.

Szablonu `Para` możemy teraz użyć wszędzie tam, gdzie potrzebujemy trzymać razem parę obiektów pewnego typu.

Zadanie 2 Dodaj do szablonu klasy `Para` metodę `suma`, która zwraca sumę trzymanych obiektów (użyj operatora +)

Widzimy już pierwszy problem towarzyszący szablonom - aby napisany kod był poprawny, dla typu, którym zainstancjonujemy szablon, musi istnieć zdefiniowany poprawnie operator +. W czasie pisania szablonu nie mamy kontroli nad typem, który zostanie do tego użyty1. Na szczęście wszystkie błędy tego typu zostaną wykryte w czasie kompilacji.

Przećwiczmy także klasy sparametryzowane NTTP.

Zadanie 3 Napisz szablon klasy `TablicaPar` sparametryzowany 1 typem oraz 1 NTTP typu `unsigned int`. Niech przechowuje ona statycznie zaalokowaną tablicę obiektów typu `Para<T>` o długości `N`, gdzie `T` i `N` to parametry klasy `TablicaPar<T, N>`.

Zadanie 4 Przeciąż dla klasy `TablicaPar` operator `[]` tak, aby umożliwić indeksowanie po trzymanej tablicy.

Zadanie 5 Przećwicz działanie napisanych szablonów na typie `double` (np. wypełnij tablicę a następnie policz sumę wszystkich trzymanych par liczb). Teraz wykonaj to samo zadanie dla typu `int`. W ilu miejscach musiałaś/musiałeś zmodyfikować kod?

Specjalizacje szablonów klas

Jeżeli chcemy, aby nasz szablon zachowywał się w szczególny sposób dla jakiejś grupy parametrów, możemy dodać do niego specjalizację. Klasy specjalizujemy wg. następującego schematu:

```
// Definicja
template</* parametry */>
class Klasa { /* ... */ };

// Specjalizacja
template</* lista parametrów specjalizacji, w szczególności może być pusta */
class Klasa</* konkretne typy/wartości/etc. wynikające z parametrów specjaliz
{ /* ... */ };
```

Pokażmy to na konkretnym przykładzie:

```
template <typename T>
struct S {
    void print() { puts("Szablon ogólny"); }
};

template <>
struct S<double> {
    void print() { puts("Specjalizacja dla double"); }
};
```

Zadanie 6 Skopiuj powyższy kod i stwórz w funkcji `main` obiekty typu `S<int>` i `S<double>`. Zweryfikuj, że metoda `print` działa zgodnie z oczekiwaniami.

Nie musimy jednak specjalizować klas dla konkretnych parametrów. Zwróćmy uwagę, że sama specjalizacja także posiada listę parametrów, którą możemy wykorzystać. Na przykład:

```
// Ogólna definicja
template <typename T>
struct S { /* ... */ };

// Specjalizacja dla wskaźników
```

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

// Specjalizacja dla referencji
template <typename T>
struct S<T&> { /* ... */ };
```

Zauważmy też, że nie musimy podawać ogólnej definicji szablonu, wystarczy ogólna *deklaracja*. W takim wypadku, gdy spróbujemy zainstancjonować szablon dla parametrów, które nie pasują do żadnej z jego specjalizacji, nasz program się nie skompiluje. Jest to swego rodzaju sposób nakładania więzów na szablony (choć niezbyt elegancki, *vide* przypis1). Możemy także postąpić odwrotnie: deklarując specjalizację bez definicji "wyłączamy" ją.

Zadanie 7 Zadeklaruj specjalizację dla klasy `TablicaPar`, która "wyłączy" puste tablice (czyli klasy `TablicaPar<T, 0>` dla każdego `T`).

---

1 Nie jest to do końca prawda, istnieją sposoby nakładania więzów na typy, którymi parametryzujemy szablon. Zainteresowane osoby odsyłamy do haseł: SFINAE (dawniej), `if constexpr` (C++17) oraz koncepty (*concepts*, C++20). Techniki te wykraczają jednak poza zakres bieżących zajęć.

Szablony funkcji

Szablony funkcji definiujemy zgodnie z tym samym schematem, co szablony klas. Główną różnicę stanowi możliwość dedukcji typów argumentów2 (opisana niżej). Siłą szablonu funkcji jest fakt, że można wykorzystać jego parametry jako typy argumentów (lub wartości zwracanej). Możemy zatem napisac w jednym miejscu dowolnie skomplikowaną implementację pewnego algorytmu działającego na argumentach nie konkretnego typu, ale całej *rodziny typów*, spełniającej jakieś minimalne założenia tej implementacji. Na przykład, pisząc funkcję

```
template<typename T>
T add(const T& a, const T& b)
{ return a + b; }
```


jesteśmy przy jej pomocy w stanie dodać 2 obiekty każdego typu należacego do rodziny typów, dla których zdefiniowany jest operator `+` zwrcający obiekt tego samego typu co jego argumenty. Działa więc ona równie dobrze dla typu `double`, jak dla typu `Wektor2D` z pierwszego laboratorium. Jest to swego rodzaju statyczny polimorfizm - mamy wspólny interfejs dla różnych klas. Jeżeli spróbujemy zainstancjonować szablon z typem niespełniającym naszych założeń, nasz program się nie skompiluje.

Zadanie 8 Napisz funkcję `iloczyn`, która przyjmuje tablicę typu, którym jest sparametryzowana oraz liczbę całkowitą będącą rozmiarem tablicy. Niech zwraca ona iloczyn elementów tej tablicy, liczony operatorem `*`. Zastanów się, jakie założenia czynisz na temat typu tablicy?

Dedukcja typów argumentów

Napisaną powyżej funkcję możemy zawołać np. w następujący sposób:

```
int tab[]    = {1, 2, 3};
int silnia_3 = iloczyn<int>(tab, 3);
```

W drugiej linijce jawne podanie parametru funkcji `iloczyn` jest niepotrzebne. C++ jest statycznie typowany, a zatem podanie `tab` jako argumentu jednoznacznie determinuje parametr, z jakim ma zostać zainstancjonowany szablon.

Zadanie 9 Napisz wolnostojącą funkcję `sumaPary`, która przyjmuje parę (w znaczeniu szablonu `Para` napisanego wyżej) obiektów typu, którym jest sparametryzowana i zwraca ich sumę (użyj metody `suma`). Stwórz parę liczb całkowitych i policz ich sumę przy użyciu funkcji `sumaPary`. Ile razy musiałaś/musiałeś użyć słowa kluczowego `int`? Dzięki dedukcji typów argumentów odpowiedź powinna wynosić 1!

---

2 Od C++17 istnieje dedukcja parametrów typu obiektu na podstawie typu argumentów jego konstruktora (CTAD), jednak temat ten wykracza poza zakres bieżącego kursu.

Wybrane szablony z STL

W tej części instrukcji pokażemy działanie kilku podstawowych szablonów biblioteki standardowej. Pierwsze 4 dotyczą tzw. *smart pointers*, czyli klas, które pozwalają nam korzystać ze wskaźników w prostszy i bezpieczniejszy sposób: `std::unique_ptr` i `std::shared_ptr` (z nagłówka `memory`). Istnieje także 3. rodzaj smart pointera - `std::weak_ptr` - lecz zaznajomienie się z nim pozostawiamy dla chętnych. Dalej poznamy `std::variant`, `std::get` i `std::visit` (z nagłówka `variant`), które pozwolą nam drastycznie uprościć kod z zajęć dotyczących polimorfizmu. Dodajmy, że celem tego rozdziału nie jest nauczenie czytelnika każdego niuansu omawianych szablonów (po takowe odsyłamy do dokumentacji), tylko przedstawienie ich filozofii i podstaw użytkowania, tak, aby w przyszłości czytelnik wiedział po jakie rozwiązanie sięgnąć w obliczu konkretnego problemu. W tym rozdziale nie zawieramy także zadań dotyczących omawianych szablonów. Zamiast tego, po jego przeczytaniu polecamy przystąpić do wykonywania projektu nr 1, do zaliczenia którego potrzebne będzie wykorzystanie szablonów omówionych poniżej.

std::unique_ptr

Klasa `std::unique_ptr<T>` to smart pointer ("inteligentny wskaźnik") posiadający **wyłączną** własność nad zasobem typu `T` i niszczący ten zasób w swoim destruktorze (zakres życia zasobu jest ograniczony zakresem życia smart pointera). Wypunktujmy najważniejsze cechy tego szablonu:

1. Jeden z konstruktorów `std::unique_ptr<T>` przyjmuje obiekt typu `T*` i zarządza zasobem, na który wskazuje podany wskaźnik. Od C++14 nie korzystamy z tego konstruktora, lecz zamiast tego z funkcji `std::make_unique<T>`.
2. `std::unique_ptr` posiada konstruktor domyślny, który tworzy obiekt, który niczym nie zarządza.
3. `std::unique_ptr` ma usunięty konstruktor kopiujący i kopiujący operator przypisania.
4. `std::unique_ptr` ma dobrze zdefiniowany konstruktor przenoszący i przenoszący operator przypisania. Te dwie metody specjalne "przejmują" zasób, którym zarządzał argument konstruktora/operatora przenoszącego.
5. `std::unique_ptr` posiada zdefiniowane operatory `*` oraz `->`, które działają analogicznie jak dla zwykłego wskaźnika.
6. Destruktor `std::unique_ptr` niszczy zasób, którym dany obiekt zarządza.

Szablon klasy `std::unique_ptr` także posiada specjalizację dla typów będących tablicami (`std::unique_ptr<T[]>`), która reprezentuje wyłączną własność nad *tablicą* obiektów. Działa ona nieco inaczej niż ogólny szablon:

1. `std::unique_ptr<T[]>` nie ma przeciążonych operatorów `*` i `->`. Zamiast nich posiada operator `[]`, który pozwala na indeksowanie po tablicy, którą zarządza.
2. `std::unique_ptr<T[]>` niszczy trzymane zasoby przy użyciu `delete[]`, a nie `delete` (poprawnie usuwa każdy element tablicy).

Wymieniowne powyżej cechy pozwalają nam korzystać z obiektów `std::unique_ptr` dokładnie tak samo, jak z wbudowanych wskaźników, nie musimy się za to martwić o zwalnianie pamięci. Dodatkowo mamy pewność, że nigdy nie wykonamy nieumyślnej kopii zasobu, ani nie spróbujemy odnieść się do zasobu, który został zniszczony. Warto też zaznaczyć, że dynamiczny polimorfizm opisany w instrukcji nr 3 działa w niezmienionej formie dla `std::unique_ptr`! W konsekwencji, jeżeli mamy istenijący kod, w którym korzystamy z wbudowanych wskaźników, to możemy zamienić deklarację wszystkich `T*` na `std::unique_ptr<T>` oraz usunąć wszystkie zawołania operatora `delete` (pod warunkiem, że wbudowane wskaźniki reprezentowały wyłączną własność). Taka operacja pozwoli nam skrócić kod (nie musimy wołać `delete`) oraz zagwarantuje nam jego poprawność (nigdy nie zapomnimy już zwolnić pamięci, próba kopiowania wskaźników teraz kończy się błędem kompilacji). Przyjrzyjmy się, jak może to wyglądać. Rozważmy następujący kod:

```
bool  warunek = sprawdzWarunek();
Baza* wsk_baza;

if (warunek)
    wsk_baza = new Pochodna1{};
else
    wsk_baza = new Pochodna2{};

wsk_baza->metodaWirtualna();
delete wsk_baza;
```

Możemy go przepisać jako:

```
bool                  warunek = sprawdzWarunek();
std::unique_ptr<Baza> wsk_baza; // konstruktor domyślny
```

```
if (warunek)
    wsk_baza = std::unique_ptr<Pochodna1>{new Pochodna1{}};
else
    wsk_baza = std::unique_ptr<Pochodna2>{new Pochodna2{}};

wsk_baza->metodaWirtualna(); // działa dzięki przeciążeniu operatora ->
// nie musimy pamiętać o wołaniu delete, robi to za nas destruktor!
```

W tym przykładzie widzimy, że `std::unique_ptr<KlasaPochodna>` jest konwertowalny na `std::unique_ptr<KlasaBazowa>`.

std::make_unique

W powyższym przykładzie, mało eleganckie mogą wydawać się linijki, w któych tworzymy `std::unique_ptr<PochodnaX>` i przypisujemy je do `wsk_baza`. Szczęśliwie, od standardu C++14, mamy do dyspozycji szablon funkcji `std::make_unique`. `std::make_unique<T>(argumenty...)` konstruuje na stercie obiekt typu `T` przy użyciu podanych argumentów3, a następnie zwraca `std::unique_ptr<T>` do tego obiektu. Efektywnie woła on za nas operator `new`. W konsekwencji, linijkę

```
wsk_baza = std::unique_ptr<Pochodna1>{new Pochodna1{}};
```

możemy zamienić na

```
wsk_baza = std::make_unique<Pochodna1>();
```

co jest niewątpliwie zwięźlejsze i prostsze w zrozumieniu. `std::make_unique` jest jednym z szablonów funkcji, przy użyciu których nie używamy dedukcji typów, lecz zawsze jawnie podajemy parametr szablonu funkcji. Jest to bardzo logiczne - nie jesteśmy w stanie na podstawie typów argumentów stwierdzić typu obiektu, którego konstruktor chcemy zawołać. Wiele klas może mieć konstruktory, które przyjmują dany zestaw typów!

std::shared_ptr

Drugim rodzajem smart pointera, który omówimy w niniejszej instrukcji jest `std::shared_ptr`. Szablon ten reprezentuje wskaźnik do zasobu, który może być współdzielony. Korzystamy z niego podobnie, jak z `std::unique_ptr`, tzn. przy użyciu operatorów `*`, `->` lub `[]`. Różnica polega na tym, że nie ma on usuniętego konstruktora kopiującego i kopiującego operatora przypisania. Te specjalne metody wykonują tzw. płytką kopię, tzn. nowa kopia obiektu typu `std::shared_ptr<T>` wskazuje na *ten sam* zasób `T`, na który wskazywał obiekt kopiowany. Jest to zachowanie identyczne do wbudowanego wskaźnika. "Inteligencja" tego wskaźnika polega na tym, że śledzi on liczbę kopii, która zostanie wykonana i zniszczy zasób dopiero wtedy, gdy zniszczona zostanie ostatnia kopia `std::shared_ptr<T>`, która na niego wskazuje. Ponownie omijamy więc konieczność wołania `delete`! Zobaczmy to na przykładzie następującego programu:

```
#include <iostream>
#include <memory>

// Klasa reprezentująca niekopiowalny zasób
struct NiekopiowalnyZasob
{
    int w;

    NiekopiowalnyZasob(int w_) : w{w_} {}

    // Kopiowanie usunięte
    NiekopiowalnyZasob(const NiekopiowalnyZasob&)            = delete;
    NiekopiowalnyZasob& operator=(const NiekopiowalnyZasob&) = delete;

    // Przenoszenie i destruktor zdefaultowane dla zgodności z Ro5
    NiekopiowalnyZasob(NiekopiowalnyZasob&&)                 = default;
    NiekopiowalnyZasob& operator=(NiekopiowalnyZasob&&)      = default;
    ~NiekopiowalnyZasob()                                    = default;
};

int main()
{
    // Stworzenie zasobu
    std::shared_ptr<NiekopiowalnyZasob> wsk1{new NiekopiowalnyZasob{42}};
    {
        // Kopia WSKAŹNIKA NA zasób
        std::shared_ptr<NiekopiowalnyZasob> wsk2{wsk1};

        std::cout << wsk1->w << '\n';
```

```
        std::cout << wsk2->w << '\n';
        std::cout << "Adres wsk1: "        << &wsk1  << "\nAdres wsk2: "
        std::cout << "Adres zasobu wsk1: " << &*wsk1 << "\nAdres zasobu

    } // Tutaj niszczymy wsk2, ale nie zasób, gdyż wsk1 nadal żyje

} // Tutaj niszczymy wsk1 oraz zasób, gdyż nic już na niego nie wskazuje
```

Kompilując i wykonując powyższy kod (lub podglądając ten link) możemy udowodnić, że `wsk1` i `wsk2` faktycznie wskazują na ten sam obiekt. Dla jasności: w tym kontekście `&*wsk` oznacza wzięcie adresu zasobu, na który wskazuje `wsk`, gdyż `*wsk` zwraca referencję do zasobu (wołamy przeciążony operator `*`), a zatem zawołanie operatora `&` na tej referencji zwróci jego adres. `&wsk` to po prostu adres obiektu `wsk` (wołamy wbudowany operator wzięcia adresu, tak samo jak robiliśmy to w C dla typów wbudowanych)

std::make_shared

`std::make_shared` działa dokładnie analogicznie do `std::make_unique` - konstruuje na stercie obiekt przy pomocy podanych argumentów i zwraca `std::shared_ptr`, który na niego wskazuje. W efekcie pomaga nam ominąć operator `new` (woła go za nas).

Uwagi nt. smart pointerów

Powyżej omówiliśmy 2 typy inteligentnych wskaźników: `std::unique_ptr` reprezentujący wyłączną własność oraz `std::shared_ptr` reprezentujący własność współdzieloną. Jeżeli różnice między nimi nie są w pełni jasne, odsyłamy czytelnika np. do tego nagrania. Poprawne ich wykorzystanie pozwala na wyeliminowanie wycieków pamięci poprzez automatyzację (do pewnego stopnia) zarządzania zasobami. Dzięki pomocniczym funkcjom `std::make_unique` i `std::make_shared` możemy więc sformułować następującą zasadę programowania w C++:

Nigdy nie wołaj bezpośrednio operatorów `new` i `delete`

Znając te narzędzia warto też wiedzieć, kiedy po nie sięgać. Temat ten jest omówiony bardzo dokładnie np. w tym nagraniu (jest to półtoragodzinny wykład, także wymieniamy je jako materiał nadprogramowy). Decydując po jakie rozwiązanie sięgnąć, powinniśmy kierować się następującą hierarchią:

1. Preferujemy zarządzanie zasobami bezpośrednio przez lifetime (zakres istnienia) obiektu, tzn. deklarujemy go bezpośrednio jako zmienną lokalną lub pole klasy.
2. Jeżeli nie jest to możliwe (np. zasób nie mieści się na stosie), tworzymy zasób dynamicznie (`std::make_unique`) i zarządzmy nim przez `std::unique_ptr`.
3. Po `std::shared_ptr` sięgamy dopiero wtedy, gdy `std::unique_ptr` nie jest wystarczający.

Uwaga: `std::unique_ptr` nadal możemy podawać do funkcji przy pomocy referencji. Konieczność korzystania z `std::shared_ptr` objawia się głównie w programach wielowątkowych (zasób współdzielony przez więcej niż jeden wątek, jest automatycznie niszczony gdy wszystkie wątki zakończą pracę) lub w strukturach danych będących grafami (dany wierzchołek może mieć więcej niż jednego rodzica).

std::variant

Cofnijmy się na chwilę do rozważań o dynamicznym polimorfizmie z poprzedniej instrukcji. Celem stosowania kombinacji dziedziczenia i metod wirtualnych była praca z obiektem, którego typ był tak jakby zmienny w czasie wykonania programu. Mając wskaźnik do klasy bazowej, mogliśmy, na podstawie np. wartości wpisanych z klawiatury, decydować na obiekt którego typu pochodnego będzie wskazywał. Rozwiązanie to było jednak obarczone następującymi problemami:

- niepotrzebnie skomplikowany kod
  - konieczność tworzenia abstrakcyjnych klas bazowych
  - pamiętanie o pisaniu słowa `virtual`, szczególnie przy destruktorze
  - design pattern wizytatora jest dość skomplikowany
  - ogólnie rzecz ujmując, sposób, w jaki chcieliśmy przechowywać/używać obiekty klas silnie ingerował w sposób, w jaki implementowaliśmy ich funkcjonalność. W idealnym świecie chcielibyśmy zawrzeć w definicji klasy jedynie to co robi. To, że chcemy trzymać obiekty danej klasy w heterogenicznym kontenerze razem z obiektami innych klas powinno być zmartwieniem kontenera, a nie trzymanych przez niego obiektów.
- konieczność dynamicznej alokacji pamięci
  - koszt w wydajności: sama alokacja jest dość kosztowną operacją
  - koszt w wydajności: dereferencja wskaźnika nie jest darmową operacją (dostęp do obiektu na stercie jest wolniejszy niż dostęp do obiektu na stosie)

– fragmentacja pamięci: dynamiczna alokacja dużej liczby małych obiektów może prowadzić do sytuacji, w której nie mamy dostępnego dużego *ciągłego* obszaru pamięci

Odpowiedzią na te problemy jest dodany w standardzie C++17 szablon `std::variant`. Wprowadza on do XXI wieku koncepcję unii typów, znaną jeszcze z C (choć zapewnie nie z kursu informatyki na wydziale MEiL). Szablon ten wygląda następująco:

```
template <typename T1, typename T2,...>
class variant;
```

Instancja klasy `std::variant<T1, T2,...>` w danym momencie trzyma obiekt dokładnie jednego z typów `T1`, `T2`, itd. Poniżej będziemy nieformalnie odnosić się do tego ciągu typów jako "paczki typów wariantu". Wypunktujmy jego najważniejsze cechy:

- standard gwarantuje, że sama klasa `std::variant` nigdy nie dokonuje dynamicznej alokacji dodatkowej pamięci
- obiekt tej klasy jest rozmiaru największego z typów `T1`, `T2`,... plus pewna (mała) stała wartość (np. w kompilatorze `gcc` jest to 8B)
- posiada konstruktor, który przyjmuje referencję (dobrze zdefiniowany zarówno dla LVR, jak i RVR) do do obiektu klasy należącej do paczki typów wariantu. Możemy więc skonstruować np.

```
std::variant<int, double> v{3.14};
```

ale już nie

```
std::variant<int, float> v{3.14}; // BŁĄD!
```

gdyż wartość `3.14` jest typu `double` (a dokładniej `double&&`), konwersja na `float` nie jest tu dopuszczalna. Jeżeli chcemy jawnie wymusić typ obiektu, który ma trzymać wariant, możemy użyć 5. przeciążenia konstruktora z dokumentacji.

- posiada operator przypisania, który działa analogicznie do konstruktora opisanego powyżej. Np.:

```
std::variant<int double> v;
v = 42;
```

- posiada dobrze zdefiniowane konstruktory kopiujące i przenoszące oraz kopiujące i przenoszące operatory przypisania
- posiada domyślny konstruktor, gdy pierwszy z paczki typów wariantu posiada domyślny konstruktor (wtedy domyślnie konstruuje obiekt `T1`)
- posiada metodę `size_t index()`, która zwraca indeks (liczony od 0) trzymanego obecnie typu z podanej paczki typów wariantu. Np.:

```
std::variant<int double> v1{42};
std::variant<int double> v2{42.};
std::cout << v1.index() << ' ' << v2.index();
```

wydrukuje `0 1`. Z tej metody nie korzystamy jednak zbyt często (po prostu nie ma takiej potrzeby, nie ze względu na jakieś dobre praktyki). - dostęp do obiektu trzymanego przez wariant odbywa się przez `std::get` i `std::visit`, opisane poniżej

std::get4

Mamy dany obiekt typu `std::variant<T1, T2,...> v`, który wiemy, że trzyma w danej chwili obiekt typu `T2`. Możemy uzyskać dostęp do tego obiektu dostęp na 2 różne sposoby:

- za pomocą indeksu

```
T2& wartosc = std::get<1>(v);
```

- za pomocą typu (działa jedynie gdy `T2` występuje w paczce typów wariantu dokładnie raz)

```
T2& wartosc = std::get<T2>(v);
```

Jeżeli `v` nie trzymałby w danej chwili wartości typu `T2`, operacja rzuci wyjątek. O wyjątkach dowiemy się więcej na późniejszym laboratorium, na chwilę obecną powiedzmy jedynie, że próba dostępu do wartości trzymanej przez wariant

przez niepoprawny typ spowoduje zakończenie pracy naszego programu w trybie awaryjnym. Dodajmy też, że `std::get` zwraca *referencję* do trzymanego obiektu, także nie musimy wykonywać jego kopii. Jeżeli chcielibyśmy to zrobić, możemy oczywiście zawołać po prostu:

```
T2 kopia_wartosci = std::get<1>(v); //
```

`std::visit`

Poznana dotychczas funkcjonalność pozwala nam na napisanie wizytatora wariantu (spokojnie, jest to dużo prostsze niż w przypadku wirtualnego polimorfizmu). Jeżeli mamy wariant sparametryzowany paczką `T1, T2,...` i wiemy, że każdy z typów należących do tej paczki ma metodę `drukuj`, możemy napisać następującą funkcję:

```
void drukujWariant(const std::variant<T1, T2,...>& v)
{
    if (v.index() == 0)
        std::get<0>(v).drukuj();
    else if (v.index() == 1)
        std::get<1>(v).drukuj();
    // itd ...
}
```

Funkcja ta jest bardzo konkretnym wizytatorem, który woła metodę `drukuj` obiektu trzymanego przez wariant. Podobnie jak w przypadku wirtualnego polimorfizmu, chcielibyśmy teraz uogólnić ideę wizytowania, tzn. stworzyć uniwersalny mechanizm, przy użyciu którego możliwe jest zawołanie dowolnej zdefiniowanej przez siebie funkcji, która obsłuży w odpowiedni sposób różne możliwe obiekty trzymane przez wariant (spoiler alert: taki mechanizm dostarcza biblioteka standardowa, spróbujemy jednak najpierw stworzyć go sami, aby zrozumieć, jak działa). Tutaj ujawni się esencja wygody (tak, wygody, nie skomplikowania), którą mogą zapewnić nam template'y.

Zanim przejdziemy do przypadku wariantu, zastanówmy się nad zagadnieniem przekazywania funkcji jako argumentów innych funkcji. W języku C służyły do tego wskaźniki do funkcji, które były jednak niewygodne oraz cechowały się dość mało intuicyjną składnią. Aby zobaczyć, jak rozwiązujemy to zagadnienie w C++, pochylmy się nad następującym przykładem. Chcielibyśmy napisać szablon funkcji, która przyjmie argument "wołalny" (ang. *callable*) oraz drugi argument dowolnego typu, a następnie podaje drugi argument do wywołania pierwszego argumentu. Mówiąc prościej, chcielibyśmy przyjąć obiekt funkcjo-podobny oraz jego argument i wywołać tę (tak jakby) funkcję z tym argumentem. Dzięki template'om, możemy w trywialny sposób zapisać taką abstrakcję:

```
template<typename Fun_t, typename Arg_t>
void zawolaj(Fun_t fun, Arg_t arg)
{
    fun(arg);
}
```

Pomijamy rozważania dotyczące przyjmowania argumentów jako referencje i wykonywania kopii, gdyż nie to jest tutaj istotne. Mając taki szablon, możemy teraz napisać:

```
void drukuj(int i) { std::cout << "int: " << i << '\n'; }

int main()
{
    zawolaj(drukuj, 1);
}
```

Dzięki dedukcji typów nie musimy się przejmować, czym jest tak naprawdę `drukuj` podany jako argument do `zawolaj`. Maszyneria template'ów martwi się o to za nas, a my możemy spędzić nasz czas na rzeczach bardziej produktywnych niż przypominanie sobie składni wskaźników do funkcji z języka C (bo to właśnie ta funkcjonalność jest przez nas wykorzystana w powyższym przykładzie). Kłopoty pojawią się, gdy funkcja `drukuj` będzie miała więcej niż jedno przeciążenie. Nie będzie wtedy jednoznaczne, które znich ma zostać podane do funkcji (czytelnik może sprawdzić to samodzielnie). Zamiast tego, możemy podać *obiekt*, który posiada przeciążenia operatora nawiasów okrągłych dla wszystkich potrzebnych typów. Konkretnie:

```
struct Drukarka
{
    void operator()(int i)    { std::cout << "int: " << i << '\n'; }
    void operator()(double d) { std::cout << "double: " << d << '\n'; }
};
```

Teraz możemy zawołać:

```
Drukarka d;
zawolaj(d, 42);
zawolaj(d, 1.);

// Lub zwięźlej:
// zawolaj(Drukarka{}, 42);
// zawolaj(Drukarka{}, 1.);
```

Idea reprezentacji operacji przez obiekty ze zdefiniowanym operatorem `()` (tzw. obiekty funkcyjne lub funktory) zostanie rozwinięta na laboratorium dotyczącym algorytmów STL, powróćmy teraz jednak do wizytacji wariantu.

Wykorzystując opisany wyżej chwyt, możemy napisać szablon ogólnego wizytatora konkretnego wariantu `std::variant<int, double>` (ponownie pomijamy rozważania nt. referencji i kopiowania):

```
template <typename Wizytator_t>
void wizytuj(Wizytator_t wizytator, std::variant<int, double> wariant)
{
    unsigned int index = wariant.index();
    if      (index == 0)
        wizytator(std::get<0>(wariant));
    else if (index == 1)
        wizytator(std::get<1>(wariant));
}
```

Podkreślmy, że próba ominięcia drzewa decyzyjnego skończy się błędem kompilacji

```
wizytator(std::get<wariant.index()>(wariant)); // Błąd!!!
```

gdyż argumenty template'ów muszą zostać określone w czasie kompilacji, a operacja `wariant.index()` jest z natury rzeczy sprawdzana w czasie wykonania programu. Zobaczmy jak możemy wykorzystać ten szablon:

```
std::variant<int, double> v{1.};
wizytuj(Drukarka{}, v);
// wydrukuje "double: 1"

v = 42;
wizytuj(Drukarka{}, v);
// wydrukuje "int: 42"
```

Jeżeli zdefiniujemy inny obiekt funkcyjny, możemy postąpić zgodnie z tym samym schematem! Mamy więc ogólną metodę dostępu do wariantu `std::variant<double, int>`.

Ogólną metodę dostępu do dowolnego wariantu zapewnia nam szablon funkcji `std::visit`. Jest on sparametryzowany nie tylko typem funktora, ale także typem samego wariantu. Dzięki temu możemy w sposób analogiczny do tego zobrazowanego wyżej wizytować obiekt każdej klasy stworzonej przez zainstancjonowanie szablonu `std::variant`. Możemy więc przepisać kod z przykładu jako:

```
std::variant<int, double> v{1.};
std::visit(Drukarka{}, v);
// wydrukuje "double: 1"

v = 42;
std::visit(Drukarka{}, v);
// wydrukuje "int: 42"
```

Ponownie widzimy, że nawet tak skomplikowana funkcjonalność jak `std::visit` (pod maską ma ona dużo meta-programowania) może być przez nas wykorzystana w prosty sposób, a wszystko dzięki dedukcji parametrów z typów argumentów oraz bibliotece standardowej.

Podsumowanie `std::variant`

- `std::variant` daje nam możliwość trzymania różnych typów w jednym obiekcie
- dzięki przyjaznemu interfejsowi możemy nadawać wariantowi wartości w naturalny sposób (operator przypisania, konstruktor)
- dostęp do trzymanego obiektu uzyskujemy używając pomocniczego szablonu funkcji `std::visit`
- dzięki wariantowi możemy w naturalny sposób definiować nasze klasy polimorficzne - omijamy dziedziczenie i słowo `virtual`

Na koniec zobaczmy, jak przepisać wizytator kształtów z poprzedniego laboratorium.

```cpp
#include <iostream>
#include <string>
#include <variant>

// Uproszczona klasa koło
class Kolo
{
public:
    Kolo() : r{0} {}
    Kolo(double r_) : r{r_} {}
    void id() { std::cout << "Jestem kołem o polu " << 3.14 * r * r << '

private:
    double r;
};

// Uproszczona klasa kwadrat
class Kwadrat
{
public:
    Kwadrat() : a{0} {}
    Kwadrat(double a_) : a{a_} {}
    void id() { std::cout << "Jestem kwadratem o polu " << a * a << '\n'

private:
    double a;
};

// Wizytujący funktor, pokażemy jak ominąć jego definicję na lab 6
struct WizytatorKsztaltu
{
    void operator()(Kolo k)    { k.id(); }
    void operator()(Kwadrat k) { k.id(); }
};

int main()
{
    std::variant<Kwadrat, Kolo> v;
    std::string s;
    std::cin >> s;
    double d;
    std::cin >> d;

    if (s == "kwadrat")
        v = Kwadrat{d};
    else if (s == "kolo")
        v = Kolo{d};
    else
    {
        std::cout << "Nie rozpoznano kształtu\n";
        return 1; // wartość inna niż 0 oznacza błąd programu
    }

    std::visit(WizytatorKsztaltu{}, v);
}
```

---

3 Mechanizm, który pozwala definiować szablony dla nieznanej *a priori* liczby parametrów wykracza poza zakres tego kursu. Zainteresowani mogą szukać hasła *variadic templates*.

4 W bibliotece standardowej są co najmniej 3 różne szablony funkcji `std::get`. W tym przypadku mowa o szablonie `std::get(std::variant)`, ale są także `std::get(std::array)` i `std::get(std::tuple)`. Służą one jednak do dostępu do klas, które leżą poza zakresem tego kursu (ze względu na ograniczenia czasowe, nie wysoki stopień skomplikowania `std::tuple` i `std::array`).