4

SZABLONY

Gdy coś się nie udaje, mówimy,
że to był tylko eksperyment.
Robert Penn Warren

Nieuchronnie, wielkimi krokami, zbliżamy się do końca kursu C++. Przed tobą jeszcze tylko jedno, ostatnie i arcyważne zagadnienie: tytułowe szablony.

Ten element języka, jak chyba żaden inny, wzbudza wśród wielu programistów różne niezdrowe emocje i kontrowersje; porównać je można tylko z reakcjami na preprocesor. Nie są to aczkolwiek reakcje skrajnie negatywne: przeciwnie, szablony powszechnie uważa się za jeden z największych atutów języka C++.
Problemem jest jednak to, iż obecne ich możliwości (mimo że już teraz ogromne) są niezadowalające dla biegłych programistów. Dlatego też właśnie szablony są tą częścią C++, która najszybciej podlega ewolucji. Trzeba jednak uświadomić sobie, że od odgórnie narzuconego pomysłu Komitetu Standaryzacyjnego do implementacji stosownej funkcji w kompilatorach wiedzie bardzo daleka droga. Skutek jest taki, że na palcach jednej ręki można policzyć kompilatory, które w pełni odpowiadają tym zaleceniom i oferuje szablony całkowicie zgodne ze standardem. Jest to zadziwiające, zważywszy że sama idea szablonów liczy już sobie kilkanaście (!) lat.

Mam jednak także pocieszającą wiadomość. Otóż można kręcić nosem i narzekać, że kompilator, którego używamy, nie jest w pełni „na czasie", lecz dla większości programistów nie będzie to miało wielkiego znaczenia. Oczywiście, najlepiej jest używać zawsze najnowszych wersji narzędzi programistycznych; nie oznacza to wszakże, że starsze ich wersje nie nadają się do niczego.
Skoro już o tym mówię, to przydałoby się wspomnieć, jak wygląda obsługa szablonów w naszym ulubionym kompilatorze, czyli Visual C++. I tu czeka nas raczej miła niespodzianka. Przede wszystkim warto wiedzieć, że jego aktualna wersja, zawarta w pakiecie Microsoft Visual Studio .NET 2003, jest absolutnie zgodna z aktualnym standardem języka C++ - naturalnie, także pod względem obsługi szablonów. Jeżeli natomiast chodzi o starszą wersję Visual Studio .NET (nazywaną teraz często .NET 2001), to tutaj sprawa także przedstawia się nie najgorzej. W codziennym, ani nawet nieco bardziej egzotycznym programowaniu nie odczujemy bowiem żadnego niedostatku w obsłudze szablonów przez ten kompilator.
Niestety, podobnie dobrych wiadomości nie mam dla użytkowników Visual C++ 6. To leciwe już środowisko może szybko okazać się niewystarczające. Warto więc zaopatrzyć w jego nowszą wersję.

W każdym jednak przypadku, niezależnie od posiadanego kompilatora, znajomość szablonów jest niezbędna. Wpisały się one w praktykę programistyczną na tyle silnie, że obecnie mało który program może się bez nich obejść. Poza tym przekonasz się wkrótce na własnej skórze, że stosowanie szablonów zdecydowanie ułatwia typowe czynności koderskie i sprawia, że tworzony kod staje się znacznie bardziej uniwersalny i elastyczny. Najlepszym przykładem tego jest Biblioteka Standardowa języka C++, z której fragmentów miałeś już okazję korzystać.
Zabierzmy się zatem do poznawania szablonów - na pewno tego nie pożałujesz :D

Podstawy

Na początek przedstawię ci, czym w ogóle są szablony i pokaże kilka przykładów na ich zastosowanie. Bardziej zaawansowanymi zagadnieniami zajmiemy się bowiem w następnym podrozdziale. Na razie czas na krótkie wprowadzenie.

Idea szablonów

Mógłbym teraz podwinąć rękami, poprosić cię o uwagę i kawałek po kawałku wyjaśniać, czym są te całe szablony. Na to również przyjdzie pora, ale najpierw lepiej chyba odkryć, do czego mogą nam się te dziwne twory przydać. Dzięki temu może łatwiej przyjdzie ci ich zrozumienie, a potem znajdowanie dlań zastosowań i wreszcie… polubienie ich! Tak, szablony naprawdę można polubić - za robotę, której nam oszczedzają; nam: ciężko przecież pracującym programistom ;-)
Zobacz zatem, jakie fundamentalne problemy pomogą ci niedługo rozwiązywać te nieocenione konstrukcje…

Ścisłość C++ powodem bólu głowy

Pewnie słyszałeś już wcześniej, że C++ jest językiem o ścisłej kontroli typów. Znaczy to, że typy danych pełnią w nim duże znaczenie i że zawsze istnieje wyraźne rozgraniczenie pomiędzy nimi.
Jednocześnie wiele mechanizmów tego języka służy, paradoksalnie, właśnie zatarciu granic pomiędzy typami danych. Wystarczy przypomnieć chociażby niejawne konwersje, które pozwalają dokonywać „w locie” zamiany z jednego typu na drugi, w sposób niezauważalny. Ponadto klasy w C++ są skonstruowane tak, aby w razie potrzeby mogły niemal doskonale imitować typy wbudowane.

Mimo to, ściśły podział informacji na liczby, napisy, struktury itd. może być często sporą przeszkodą…

Dwa typowe problemy

Kłopoty zaczynają się, gdy chcemy napisać kod, który powinien działać w odniesieniu do kilku możliwych typów danych. Z grubsza można tu rozdzielić dwie sytuacje: gdy próbujmy napisać uniwersalną funkcję i gdy podobną próbę czynimy przy definiowaniu klasy.

Problem 1: te same funkcje dla różnych typów

Tradycyjnym, wręcz klasycznym przykładem tego pierwszego problemu jest funkcja wyznaczająca większa liczbę spośród dwóch podanych. Prawdopodobnie z takiej funkcji będziesz często skorzystał, więc kiedyś możesz ją zdefiniować np. jako:

```
int max(int nLiczba1, int nLiczba2)
{
   return (nLiczba1 > nLiczba2 ? nLiczba1 : nLiczba2);
}
```

Taka funkcja działa dobrze dla liczb całkowitych, ale już całkiem nie radzi sobie z liczbami typu `float` czy `double`, bo zarówno wynik, jak i parametry są zaokrąglane do jedności. Dla zdefiniowanych przez nas typów danych jest zaś zupełnie nieprzydatna, co chyba zresztą całkowicie zrozumiałe.

Naturalnie, możemy sobie dodać inne, przeciążone wersje funkcji - jak chociażby taką:

```
double max(double fLiczba1, double fLiczba2)
{
```

```
    return (fLiczba1 > fLiczba2 ? fLiczba1 : fLiczba2);
}
```

Takich wersji musiałoby być jednak bardzo wiele: za każdym kolejnym typem, dla którego chcielibyśmy stosować `max()`, musiałaby iść odrębna funkcja. Ich definiowanie byłoby uciążliwe i nudne, a podczas wykonywania tej nużącej czynności trudno byłoby nie zwątpić, czy jest to aby na pewno słuszne rozwiązanie…

Problem 2: klasy operujące na dowolnych typach danych

Innym problemem są klasy, które z jakichś względów muszą być elastyczne i operować na danych dowolnego typu. Koronnym przykładem są pojemniki, jak np. tablice dynamiczne, podobne do naszej klasy `CIntArray`. Jak wiemy, ma ona sporą wadę: przy jej pomocy nie można bowiem zarządzać tablicą elementów innego typu niż `int`. Chcąc to osiągnąć, należałoby napisać nową klasę - zapewne bardzo podobną do wspomnianej. Tę samą pracę trzebaby wykonać dla każdego następnego typu elementów…

To na pewno nie jest dobre wyjście!

Możliwe rozwiązania

„Ale jakie mamy wyjście?”, spytasz pewnie. Cóż, można sobie jakoś radzić…

Wykorzystanie preprocesora

Ogólną funkcję `max()` (i podobne) możemy zasymulować przy użyciu parametryzowanych makr:

```
#define MAX(a,b)      ((a) > (b) ? (a) : (b))
```

Sądzę jednak, że pamiętasz wady takich makrodefinicji. Nawiasy wokół `a` i `b` likwidują wprawdzie problem pierwszeństwa operatorów, ale nie zabezpieczą przed podwójnym obliczaniem wyrażeń. Wiesz przecież, że preprocesor działa na kodzie tak jak na tekście, zatem np. wyrażenie w rodzaju:

```
MAX(10, rand())
```

nie zwróci nam wcale liczby pseudolosowej równej co najmniej `10`. Zostanie ono bowiem rozwinięte do:

```
((10) > (rand()) ? 10 : (rand()))
```

Funkcja `rand()` będzie więc obliczana dwukrotnie, z każdym razem dając oczywiście inny wynik - bo takie jest jej przeznaczenie. Makro `MAX()` nie będzie więc zawsze działało poprawnie.

Używanie ogólnych typów

Jeszcze mniej oczywisty jest sposób na zaimplementowanie ogólnej klasy, np. tablicy przechowującej dowolny typ elementów. Tutaj aczkolwiek także istnieje pewne rozwiązanie: można użyć ogólnego wskaźnika, tworząc tablicę elementów typu `void*`:

```
class CPtrArray
{
   private:
         // tablica i jej rozmiar
         void** m_ppvTablica;
         unsigned m_uRozmiar;

   // itd. (metody i przeciążone operatory)
```

```
};
```

Będziemy musieli się jednak zmagać z niedogodnościami wskaźników `void*` - przede wszystkim z utratą informacji o rzeczywistym typie danych:

```
CPtrArray Tablica(5);

// alokacja pamięci dla elementu (!)
Tablica[2] = new int;

// przypisanie - nieszczególnie ładne...
*(static_cast<int*>(Tablica[2])) = 10;
```

Każdorazowe rzutowanie na właściwy typ elementów (tutaj `int`) na pewno nie będzie należało do przyjeności. Poza tym trzeba będzie pamiętać o zwolnieniu pamięci zaalokowanej dla poszczególnych elementów. W przypadku małych obiektów, jak liczby, nie ma to żadnego sensu…

Zatem nie! To na pewno nie jest zadowalające wyjście!

Szablony jako rozwiązanie

W porządku, dosyć tych bezowocnych poszukiwań. Myślę, że domyślasz się, iż to szablony są tym rozwiązaniem, którego poszukujemy. Zatem nie tracąc więcej czasu, znajdźmy je wreszcie :)

Kod niezależny od typu

Wróćmy wpierw do prób napisania funkcji `max()`. Patrząc na jej dwie wersje, dla typów `int` i `double`, możemy łatwo zauważyć, że różnią się one bardzo niewiele. Właściwie to można stwierdzić, że po prostu drugi z wariantów ma wpisane `double` tam, gdzie w pierwszym widnieje typ `int`.
Gdybyśmy więc chcieli napisać ogólny wzorzec dla funkcji `max()`, wyglądałby on tak:

```
typ max(typ Parametr1, typ Parametr2)
{
   return (Parametr > Parametr2 ? Parametr1 : Parametr2);
}
```

No dobrze, możemy sobie pisać takie wzorce, ale co nam z tego? Nie znamy przecież żadnego sposobu, aby przekazać go kompilatorowi do wykorzystania… Czy na pewno?…

Kompilator to potrafi

Ależ nie! Możemy ten wzorzec - ten **szablon** (ang. *template*) - wpisać do kodu, tworząc ogólną funkcję `max()`. Trzeba to jedynie zrobić w odpowiedni sposób - tak, aby kompilator wiedział, z czym ma do czynienia. Zobaczmy więc, jak można tego dokonać.

Składnia szablonu

A zatem: chcąc zdefiniować wzorzec funkcji `max()`, musimy napisać go w ten oto sposób sposób:

```
template <typename TYP>    TYP max(TYP Parametr1, TYP Parametr2)
{
   return (Parametr > Parametr2 ? Parametr1 : Parametr2);
}
```

Dopóki nie wyjaśnimy sobie dokładnie kwestii umieszczania szablonów w plikach źródłowych, zapamiętaj, aby wpisywać je **w całości w plikach nagłówkowych**.

W ten sposób tworzymy **szablon funkcji** (ang. *function template*) Zobaczmy, co się na niego składa.

Zauważyłeś zapewne najpierw zupełnie nową część nagłówka funkcji:

```
template <typename TYP>
```

Jest ona obowiązkowa dla każdego rodzaju szablonów, nie tylko funkcji. Słowo kluczowe `template` ('szablon') mówi bowiem kompilatorowi, że nie ma tu do czynienia ze zwykłym kodem, lecz właśnie z szablonem.
Dalej następuje, ujęta w nawiasy ostre, **lista parametrów szablonu**. W tym przypadku mamy tylko jeden taki parametr: słowo `typename` ('nazwa typu') informuje, że jest nim typ. Okazuje się bowiem, że parametrami szablonu mogą być także „normalne" wartości, podobne do argumentów funkcji - nimi też się zajmiemy, ale później. Na razie mamy tu jeden parametr szablonu będący typem o jakże opisowej nazwie `TYP`.

Potem przychodzi już normalna definicja funkcji - z jedną drobną różnicą. Jak widać, używamy w niej nazwy `TYP` zamiast właściwego typu danych (czyli `int`, `double`, itd.). Stosujemy go jednak w tych samych miejscach, czyli jako typ wartości zwracanej oraz typ obu przyjmowanych parametrów funkcji.
Treść szablonu odpowiada więc wzorcowi z poprzedniego akapitu. Różnica jest jednak taka, że o ile tamten „kod" był niezrozumiały dla kompilatora, o tyle ten szablon jest jak najbardziej poprawny i, co najważniejsze, działa zgodnie z oczekiwaniami. Nasza funkcja `max()` potrafi już bowiem operować na dowolnym typie argumentów:

```
int      nMax = max(-1, 2);          // TYP = int
unsigned uMax = max(10u, 65u);       // TYP = unsigned
float    fMax = max(-12.4, 67);      // TYP = double (!)
```

Najciekawsze jest to, iż to funkcja na podstawie swych argumentów „sama zgaduje", jaki typ danych ma być wstawiony w miejsce symbolicznej nazwy `TYP`. To właśnie jedna z zalet szablonów funkcji: używamy ich zwykle tak samo, jak normalnych funkcji, a jednocześnie zyskujemy zadziwiającą uniwersalność.

Popatrzmy jeszcze na ogólną składnię szablonu w C++:

```
template <parametry_szablonu> kod
```

Jak wspomniałem, słówko template jest tu obowiązkowe, bo dzięki nim niemu kompilator wie, że ma do czynienia z szablonem. *parametry_szablonu* to najczęściej symboliczne oznaczenia nieznanych z góry typów danych; oznaczenia te są wykorzystywane w następującym dalej *kodzie*.
Na temat obu tych kluczowych części szablonu powiemy sobie jeszcze mnóstwo rzeczy.

Co może być szablonem

Wpierw ustalmy, do jakiego rodzaju kodu w C++ możemy „doczepić" frazę `template<...>`, czyniąc ją szablonem. Generalnie mamy dwa rodzaje szablonów:

- szablony funkcji - są to więc taki funkcje, które mogą działać w odniesieniu do dowolnego typu danych. Zazwyczaj kompilator potrafi bezbłędnie ustalić, jaki typ jest właściwy w konkretnym wywołaniu (por. przykład zastosowania szablonu `max()` z poprzedniego punktu)

- szablony klas - czyli klasy, potrafiące operować na danych dowolnego typu. W tym przypadku musimy zwykle podać ten właściwy typ; zobaczymy to wszystko nieco dalej

Wkrótce aczkolwiek okazało się, że bardzo pożądane są także inne rodzaje szablonów - głównie po to, aby ułatwić pracę z szablonami klas. My jednak zajmiemy się zwłaszcza tymi dwoma rodzajami szablonów. Wpierw więc poznasz nieco bliżej szablony funkcji, a potem zobaczysz także szablony klas.

Szablony funkcji

Szablon funkcji możemy wyobrazić sobie jako:
- ogólny algorytm, który działa poprawnie dla danych różnego typu
- zespół funkcji, zawierającą odrębne wersje funkcji dla poszczególnych typów

Oba te podejścia są całkiem słuszne, aczkolwiek jedno z nich bardziej odpowiada rzeczywistości. Otóż:

Szablon funkcji reprezentuje zestaw (rodzinę) funkcji, działających dla dowolnej liczby typów danych.

Zasada stojąca za szablonami jest taka, że kompilator sam dokonuje po prostu tego, co mógłby zrobić programista, nudząc się przy tym niezmiernie. Na podstawie szablonu funkcji generowane są więc jej konkretne egzemplarze (specjalizacje, będące przeciążonymi funkcjami), operujące już na rzeczywistych typach danych. Potem są one wywoływane w trakcie działania programu.

Proces ten nazywamy **konkretyzacją** (ang. *instantiation*) i zachodzi on dla wszelkiego rodzaju szablonów. Zanim aczkolwiek może do niego dojść, szablno trzeba zdefiniować. Zobaczmy więc, jak definiuje się szablony funkcji.

Definiowanie szablonu funkcji

Definicja szblonu funkcji nie różni się zbytnio od zwykłej definicji funkcji. Ot, po prostu jeden typ (lub więcej) nie są w niej podane *explicité*, lecz wnioskowane z wywołania funkcji szablonowej. Niemniej, temu wszystkiemu trzeba się przyjrzeć bliżej.

Podstawowa definicja szablonu funkcji

Oto jeden z prostszych chyba przykładów szablonu funkcji - wartość bezwzględna:

```
template <typename TYP>    TYP Abs(TYP Liczba)
{
   return (Liczba >= 0 ? Liczba : -Liczba);
}
```

Posiada takiego szablonu ma tę niezaprzeczalną zaletę, że bez dodatkowego wysiłku możemy posługiwać się tą funkcją dla liczb dowolnego typu: `int`, `float`, `double`, itd. Co najważniejsze, w wyniku otrzymamy wartość tego samego typu, co podany parametr, zatem nie musimy posługiwać się rzutowaniem - co byłoby konieczne w przypadku zdefiniowania zwykłej funkcji dla najbardziej „pojemnego” typu `double`.
Dlaczego tak jest? Oczywiście dlatego, iż symboliczne oznaczenie `TYP` (czyli **parametr szablonu**) występuje zarówno jako typ wartości zwracanej, jak i typ parametru funkcji. W konkretnych egzemplarzach funkcji w obu miejscach wystąpi więc ten sam typ, np. `int`.

Stosowalność definicji

Można zapytać: „Czy powyższy szablon może działać tylko dla wbudowanych typów liczbowych? Czy poradziłby sobie np. z wyznaczeniem wartości bezwzględnej z liczby wymiernej, czyli obiektu zdefiniowanej ongiś klasy `CRational`?…”

Aby zdecydować o tym i o podobnych sprawach, musimy odpowiedzieć na inne pytanie:

> Czy to, co robimy w treści szablonu funkcji, da się wykonać po podstawieniu żądanego typu w miejsce parametru szablonu?

U nas więc typ danych, występujący na razie pod oznaczeniem `TYP`, musi udostępniać:
- operator porównania `>=`, pozwalający na konfrontację obiektu z zerem
- operator negacji `-`, służący tutaj do uzyskania liczby przeciwnej do danej
- publiczny konstruktor kopiujący, umożliwiający zwrot wyniku funkcji

Pod wszystkie te wymagania podpadają rzecz jasna wbudowane typy liczbowe. Jeśli zaś wyposażylibyśmy klasę `CRational` we dwa wspomniane operatory, to także jej obiekty mogłyby być argumentami funkcji `Abs()`! Wynika stąd, że:

Szablon funkcji może być stosowany dla **tych typów danych**, dla których **poprawne są wszystkie operacje**, dokonywane na **obiektach tychże typów** w **treści szablonu**.

Łatwo można więc stwierdzić, że np. dla typu `std::string` ten szablon byłby niedozwolony. Klasa `std::string` nie udostępnia bowiem operatora negacji, ani też nie pozwala na porównywanie swych obiektów z liczbami całkowitymi.

Parametr szablonu użyty w ciele funkcji

Trudno zauważyć to na pierwszy rzut oka, ale przedstawiony wyżej szablon ma jeden dość poważny zgrzyt. Mianowicie, wymusza on na podanym mu typie danych, aby pozwalał na porównywanie go z typem `int`. Do takiego typu należy bowiem newralgiczne `0`.

Nie jest to zbyt dobre i lepiej, żeby funkcja nie korzystała z takiego rozwiązania. Interpretacja zera w różnych typach liczbowych może być bowiem całkiem odmienna od zakładanej przez nas.
Lepiej więc, żeby punkt zerowy mógł być ustalony przez **domyślny konstruktor**. Wówczas szablon będzie wyglądał tak - zmiana jest niewielka:

```
template <typename TYP>    TYP Abs(TYP Liczba)
{
   return (Liczba >= TYP() ? Liczba : -Liczba);
}
```

Teraz będzie on jednak działał poprawnie dla każdego sensownego typu danych liczbowych.

„Chwileczkę”, rzekniesz. „A co z typami podstawowymi? Przecież one nie mają konstruktorów!” Faktycznie, słuszna uwaga. Taką uwagę poczynił pewnie swego czasu któryś z twórców C++, gdyż zaowacowała ona wprowadzeniem do języka tzw. **inicjalizacji zerowej**. Jest to bardzo prosta rzecz: otóż typy wbudowane (jak int czy bool) zostały wyposażone w swego rodzaju „konstruktory”. Nie są to prawdziwe funkcje składowe, jak w przypadku klas, lecz po prostu możliwość użycia tej samej składni jawnego wywołania domyślnego konstruktora. Wygląda ona tak:

```
typ()
```

i dla klas nie jest, jak sądzę, żadną niespodzianką. To samo jednak możemy uczynić także w stosunku do podstawowych typów danych. W C++ są więc całkowicie poprawne wyrażenia typu `int()`, `float()`, `bool()` czy `unsigned()`. Co ważniejsze w wyniku dają one **zero odpowiedniego typu** - czyli działają tak, jakbyśmy napisali (odpowiednio): `0`, `0.0f`, `false` i `0u`.

Inicjalizacja zerowa gwarantuje więc współpracę naszego szablonu z typami podstawowymi, ponieważ wyrażenie `TYP()` da w każdym przypadku potrzebny nam tutaj „obiekt zerowy". Nieważne, czy będzie chodziło o typ podstawowy C++, czy też klasę zdefiniowaną przez programistę.

Parametr szablonu i parametr funkcji

Mówiąc o szablonach funkcji, można się nieco zagubić w znaczeniu słowa 'parametr'. Mamy mianowicie aż dwa rodzaje parametrów:

- parametry funkcji - czyli te znane nam już od dawna, bo występuje one w każdej niemal funkcji. Każdy taki parametr ma swój typ i nazwę
- parametry szablonu poznaliśmy w tym rozdziale. W przypadku szablonów funkcji mogą to być wyłacznie nazwy typów. Parametry szablonu stosujemy więc w nagłówku i w ciele funkcji tak, jak gdyby były to nazwy typów, np. `float` czy `VECTOR2D`

To naturalne, że oba te rodzaje parametrów są ze sobą ściśle związane. Popatrzmy choćby na nagłówek funkcji `max()`:

```
template <typename TYP>    TYP max(TYP Parametr1, TYP Parametr2)
```

Parametry tej funkcji to `Parametr1` i `Parametr2`. Obydwa należą one do typu oznaczonego po prostu jako `TYP`. Ów `TYP` mógłby być klasą, aliasem zdefiniowanym poprzez `typedef`, wyliczeniem `enum`, itd. Tutaj jednak `TYP` jest **parametrem szablonu**: deklarujemy go w nawiasach ostrych po słowie `template` przy pomocy `typename`. Fakt, że `TYP` parametrów funkcji jest parametrem szablonu ma dalekosiężne i dobroczynne konsekwencje. Powoduje to mianowicie, iż może on być wydedukowany z argumentów wywołania funkcji:

```
// (było już dość przykładów wywoływania max(), więc jeden wystarczy :D)
std::cout << max(42, 69);
```

Nie musimy w powyższej linijce wyraźnie określać, że szablon `max()` ma być tu użyty do wygenerowania funkcji pracującej na argumentach typu `int`. Ten typ zostanie po prostu „wzięty" z argumentów wywołania (które są typu `int` właśnie). To jedna z wielkich zalet szablonów funkcji.

Możliwe jest aczkolwiek jawne określenie typu, czyli parametru szablonu. O tym powiemy sobie w następnym paragrafie.

Kilka parametrów szablonu

Dotąd widzieliśmy jednoparametrowe szablony funkcji, ale nie jest to kres możliwości szablonów. Tak naprawdę bowiem mogą mieć one dowolną liczbę parametrów. Oto na przykład inny wariant funkcji `max()`:

```
template <typename TYP1, typename TYP2>
   TYP1 max(TYP1 Parametr1, TYP2 Parametr2)
{
   return (Parametr > Parametr2 ? Parametr1 : Parametr2);
}
```

Podobnie jak parametry funkcji, parametry szablonu zawarte w nawiasach ostrych także o oddzielamy przecinkami. Może ich być dowolna ilość; tutaj mamy dwa parametry szablonu, które bezpośrednio przedkładają się na dwa parametry funkcji. Nowa wersja funkcji `max()` potrafi więc porównywać wartości różnych typów - o ile oczywiście istnieje odpowiedni operator `>`.
Oto przykład wykorzystania tego szablonu:

```
int       nMax = max(-18, 42u);   // TYP1 = int, TYP2 = unsigned
float     fMax = max(9.5f, 34);   // TYP1 = float, TYP2 = int
          fMax = max(6.78, 80);   // TYP1 = double, TYP2 = int
```

W ostatnim wywołaniu wartością zwróconą przez `max()` będzie `80.0` typu `double`. Jej przypisanie do mniej pojemnego typu `float` spowoduje zapewne ostrzeżenie kompilatora.

Jak widać, argumenty funkcji nie muszą być tu konwertowane do wspólnego typu, jak to się działo przy jednoparametrowym szablonie. W sumie jednak między oboma szablonami nie ma wielkiej róznicy funkcjonalnej; podałem tu jedynie przykład na to, że szablon funkcji może mieć więcej parametrów niż jeden.

Z powyższym szablonem jest jednak pewien dość istotny kłopot. Chodzi mianowicie o typ wartości zwracanej. Wpisałem w nim wprawdzie `TYP1`, ale to nie ma żadnego uzasadnienia, gdyż równie dobry (a raczej niedobry) były `TYP2`.
Problemem jest to, iż na etapie kompilacji nie wiemy rzecz jasna, jakie wartości zostaną przekazane do funkcji. Nie wiemy wobec tego, jaki powinien być typ wartości zwracanej. W takiej sytuacji należałoby użyć typu ogólniejszego, bardziej pojemnego: dla `int` i `float` byłby to zatem `float`, i tak dalej (przypomnij sobie z poprzedniego rozdziału, kiedy jakiś typ jest ogólniejszy od drugiego). Niestety, ponieważ z samego założenia szablonów funkcji nie wiemy, dla jakich faktycznych typów będzie on użyty, nie możemy nijak określić, który z tej dwójki będzie pojemniejszy. W zasadzie więc nie wiemy, jaki powinien być typ wartości zwracanej!
Rozsądne rozwiązanie tego problemu nie leży niestety w zakresie możliwości programisty. Potrzebny jest tutaj jakiś nowy mechanizm jezyka; zwykle mówi się w tym kontekście o operatorze `typeof` (‘typ czegoś’). Miałby on zwracać nazwę typu z podanego mu (stałego) wyrażenia. Nazwa ta mogłaby być potem użyta tak, jak każda inna nazwa typu - a więc na przykład w charakterze rodzaju wartości zwracanej przez funkcję.
Obecnie istnieją kompilatory, które oferują operator `typeof`, ale oficjalny standard C++ póki co nic o nim nie mówi.

Specjalizacja szablonu funkcji

Podstawowy szablon funkcji definiuje nam ogólną rodzinę funkcji, której członkowie (specjalizacje) dla każdego typu (parametru szablonu) zachowują się tak samo. Nasza funkcja `max()` będzie więc zwracały większą liczbę niezależnie od tego, czy typem jest liczby będzie `double` czy `int`.

Powiesz: „I bardzo dobrze! O to nam przecież chodzi.” No tak, ale jest pewien szkopuł. Dla pewnych typów danych algorytm wyznaczania większej wartości może być nieodpowiedni. Uogólniając sprawę, można zkonkludować, że niekiedy potrzebna nam jest specjalna wersja szablonu funkcji, która dla jakiegoś konkretnego typu (parametru szablonu) będzie się zachowywała inaczej niż dla reszty.
Wtedy właśnie musimy sami zdefiniować ową konkretną **specjalizację szablonu funkcji**. Tym zajmiemy się w niniejszym paragrafie.

Wyjątkowy przypadek

Twoja nauka C++ opiera się między innymi na serii narzuconych przypuszczeń, zatem teraz przypuśćmy, że chcemy rozszerzyć nieco funkcjonalność szablonu funkcji `max()`. Załóżmy mianowicie, że chcemy uczynić ją władną do współpracy nie tylko z liczbami, ale też z taką oto klasą wektora:

```
#include <cmath>

struct VECTOR2
{
   // współrzędne tegoż wektora
   double x, y;

   //---------------------------------------------------------------------

   // metoda licząca długość wektora
   double Dlugosc() const  { return sqrt(x * x + y * y); }

   // (reszta jest średnio potrzebna, zatem pomijamy)
};
```

Naturalnie, możnaby wyposażyć ją w odpowiedni `operator>()`. My jednak chcemy zdefiniować specjalizowaną wersję szablonu funkcji `max()`. Czynimy to w taki oto sposób:

```
template<> VECTOR2 max(VECTOR2 vWektor1, VECTOR2 vWektor2)
{
   // porównujemy długości wektorów; w przypadku równości zwracamy 1-szy
   return (vWektor1.Dlugosc() >= vWektor2.Dlugosc() ?
           vWektor1 : vWektor2);
}
```

Właściwie to można powiedzieć, że funkcja ta nie różni się prawie niczym od normalnej funkcji `max()` (nieszablonowej). Dlatego też ważne jest opatrzenie jej frazą `template<>` (z pustymi nawiasami ostrymi), bo dzięki temu kompilator może uznać nasza definicję za **specjalizację szablonu funkcji `max()`**.
Co do nagłówka funkcji, to jest to ten sam naglówek, co w oryginalnym szablonie - z tą tylko różnicą, że `TYP` zostało zamienione na nazwę rzeczywistego typu, czyli `VECTOR2`. Ze względu na tą jednoznaczność specjalizacja nie wymaga żadnych dalszych zabiegów. W sumie jednak można (i zaleca się) bezpośrednie podanie typu, dla którego specjalizujemy szablon:

```
template<> VECTOR2 max<VECTOR2>(VECTOR2 vWektor1, VECTOR2 vWektor2)
```

Dziwną frazę `max<VECTOR2>` można tu z powodzeniem traktować jako nazwę funkcji - specjalizacji szablonu `max()` dla typu `VECTOR2`. W takiej zresztą roli poznamy podobne konstrukcje, gdy zajmiemy się dokładniej użyciem funkcji szablonowych.

Ciekawostka: specjalizacja częściowa szablonu funkcji

Jak każda Ciekawostka, także i ta nie jest przeznaczona dla początkujących, a już na pewno nie podczas pierwszego kontaktu z tekstem.

Poprzednio specjalizowaliśmy funkcję dla ściśle określonego typu danych. Teoretycznie możnaby jednak zrobić coś innego: napisać specjalną jej wersję dla pewnego **rodzaju typów**.

„No, teraz to już przesadzasz!”, możesz tak odpowiedzieć. To jednak może mieć sens; wyobraźmy sobie, że przy pomocy `max()` spróbujemy porównać dwa wskaźniki. Co

otrzymamy w wyniku takiego porównania?… Naturalnie, dostaniemy ten wskaźnik, którego adres jest mniejszy.
Zapytam wprost: i co nam z tego? Lepiej chyba byłoby, aby porównanie dokonywane było raczej na obiektach, do których te wskaźniki się odnoszą. Wtedy mielibyśmy bardziej sensowny wynik i np. z dwóch wskaźników typu `int*` dostalibyśmy ten, który odnosi się do większej liczby.

Takie działanie szablonu funkcji `max()` w odniesieniu do wskaźników - przy zachowaniu jego normalnego działania dla pozostałych typów danych - nie jest możliwe do osiągnięcia przy pomocy zwykłej specjalizacji, zaprezentowanej w poprzednim punkcie. Trzebaby bowiem zdefiniować osobne wersje dla wszystkich typów wskaźników (`int*`, `CRational*`, `float*`, …), jakich chcielibyśmy używać. Całkowicie przekreśla to sens szablonów, które przecież opierają się właśnie na tym, że to sam kompilator generuje ich wyspecjalizowane wersje w zależności od potrzeb.
Tutaj trzeba by użyć mechanizmu **specjalizacji częściowej**, znanego bardziej z szablonów klas. Oznacza on ni mniej, ni więcej, jak tylko zdefiniowanie innej wersji szablonu dla całej grupy typów (parametrów szablonu). W tym przypadku ta grupą są typy wskaźnikowe, a szablon funkcji `max()` wyglądałby dla nich tak:

```
template <typename TYP>
   TYP* max<TYP*>(TYP* pWskaznik1, TYP* pWskaznik2)
{
   return (*pWskaznik1 > *pWskaznik2 ? pWskaznik1 : pWskaznik2);
}
```

Nazwa specjalizowanej funkcji, czyli `max<TYP*>`, gdzie `TYP` jest parametrem szablonu, wskazuje jednoznacznie, iż chodzi nam o wersję funkcji przeznaczoną dla wskaźników. Naturalnie, typ wartości zwracanej i parametrów funkcji musi być również taki sam.

Kiedy zostanie użyty ten bardziej wyspecjalizowany szablon?… Otóż wtedy, gdy jako parametry funkcji `max()` zostaną przekazane jakieś wskaźniki, np.:

```
int nLiczba1 = 10, nLiczba2 = 98;
int* pnLiczba1 = &nLiczba1;
int* pnLiczba2 = &nLiczba2;

std::cout << *(max(pnLiczba1, pnLiczba2));   // szablon max<TYP*>(),
                                             // gdzie TYP = int
```

W tym więc przypadku wyświetlaną liczbą będzie zawsze `98`, bo liczyć się będą tutaj faktyczne wartości, a nie rozmieszczenie zmiennych w pamięci (a więc nie adresy, na które pokazują wskaźniki).

Częściowe specjalizacje szablonów funkcji nie wyglądają może na zbytnio skomplikowane. Może cię jednak zaskoczyć to, iż to jeden z najbardziej zaawansowanych aspektów szablonów - tak bardzo, że póki co Standard C++ o nim nie wspomina (!), a tylko nieliczne kompilatory obsługują go. Póki co jest to więc bardzo rzadko używana technika i dlatego na razie należy ją traktować jako ciekawostkę.

Wywoływanie funkcji szablonowej

Skoro już mniej więcej wiemy, jak można definiować szablony funkcji, nauczmy się teraz z nich korzystać. Zważywszy, że już to robiliśmy, nie powinno to sprawiać żadnych trudności.

Zastanówmy się jednak, co dzieje się w momencie wywołania funkcji szablonowej. Oto przykład takiego wywołania:

```
max(12, 56)
```

`max()` jest tu szablonem funkcji, którego parametr (typ) jest stosowany w charakterze typu obu parametrów funkcji, jak również zwracanej przez nią wartość. Nie podajemy jednak tego typu dosłownie; to właśnie wielka zaleta szablonów funkcji, gdyż właściwy typ - parametr szablonu, tutaj `int` - może być wydedukowany z jej wywołania. O tym, jak to się dzieje, mówi następny akapit.

Aby jednak zrozumieć istotę szablonów funkcji, musimy choć z grubsza wiedzieć, jak kompilator traktuje takie wywołania jak powyższe. Generalnie nie jest trudne. Jak wspomniałem wcześniej, szablony w C++ są implementowane w ten sposób, iż podczas kompilacji tworzony jest ich właściwy („nieszablonowy") kod dla każdego typu, dla którego używamy danego szablonu. Proces ten nazywamy **konkretyzacją** (ang. *instantiation*) a poszczególne egzemplarze szablonów - **specjalizacjami** (ang. *specialization* albo *instance*).
Tak więc kompilator musi sobie wytworzyć odpowiednie specjalizacje, które będą wykorzystywane w miejscach użycia szablonu. W przykładzie powyżej szablon funkcji `max()` posłuży do wygenerowania jej konkretnej wersji: funkcji `max()` dla parametru szablonu równego `int`. Dopiero ta konkretna wersja - specjalizacja - będzie skompilowana w normalny sposób, do normalnego kodu maszynowego. W ten sposób zarówno funkcje, jak też klasy szablonowe zachowują niemal wszystkie cechy zwykłych funkcji i klas.

To, jak szablon funkcji zostanie skonkretyzowany w danym przypadku, zależy wyłącznie od sposobu jego użycia w kodzie. Przyjrzyjmy się więc sposobom na wywoływanie funkcji szablonowych.

Jawne określenie typu

Zwykle używając szablonów funkcji pozwalamy kompilatorowi na samodzielne wydedukowanie typu, dla którego ma on być skonkretyzowany. Zdarza się jednak, że chcemy go sami wyraźnie określić. To również jest możliwe.

Wywoływanie konkretnej wersji funkcji szablonowej

Możemy więc zażyczyć sobie, aby funkcja `max()` działała w danym przypadku, powiedzmy, na liczbach typu `unsigned` - mimo że typem jej argumentów będzie `int`:

```
unsigned uMax = max<unsigned>(45, 3);  // 45 i 3 to liczby typu int
```

Składnia `max<unsigned>` pozwala nam podać żądany typ. Ściślej mówiąc, **w nawiasach ostrych podajemy parametry szablonu** (w odróżnieniu od parametrów funkcji, podanych jak zwykle w nawiasach okrągłych). Tutaj jest to jeden parametr, będący typem; nadajemy mu „wartość" `unsigned`, czyli typu liczb bez znaku.

Takie wywołanie powoduje, że nie jest już przeprowadza żadna dedukacja typu argumentów funkcji. Kompilator nie zważa już na nie, lecz oczekuje, że będą one zgadzały się z typem pdoanym jawnie - parametrem szablonu. W tym więc przypadku liczby muszą pasować do typu `unsigned` i oczywiście pasują do niego (są dodatnie), choć ich właściwy typ to `int`. Nie gra on jednak żadnej roli, gdyż sami odgórnie narzuciliśmy tutaj parametr szablonu.

Użycie wskaźnika na funkcję szablonową

`max<unsigned>` występuje tutaj w miejscu, gdzie zwykle pojawia się nazwa funkcji w przypadku normalnych procedur. To nie przypadek: możemy tę frazę traktować właśnie jako **nazwę funkcji** - konkretnej już funkcji, a nie jej szablonu.

Nie jest to żadne pustosłowie, bowiem ma to konkretne konsekwencje. Nazwa `max<unsigned>` działa mianowicie tak samo, jak każda inna nazwa funkcji. W szczególności, możemy jej użyć do pobrania adresu funkcji szablonowej:

```
unsigned (*pfnUIntMax)(unsigned, unsigned) = max<unsigned>;
```

Zauważ różnicę: nie możemy pobrać adresu szablonu (czyli `max`), bo ten **nie istnieje w pamięci** podczas działania programu. Jest on tylko instrukcją dla kompilatora (podobnie jak makra są instrukcjami dla preprocesora), mówiącą mu, jak ma wygenerować prawdziwe, specjalizowane funkcje. `max<unsigned>` jest taką właśnie wyspecjalizowaną funkcją i ona już istnieje w pamięci, bowiem jest kompilowana do kodu maszynowego tak, jak normalna funkcja. Możemy zatem pobrać jej adres.

Dedukcja typu na podstawie argumentów funkcji

Jawne podawanie parametrów szablonu funkcji jest generalnie nieczęsto stosowane. Zdecydowanie największą zaletą tych szablonów jest to, iż potrafią same wykryć typ argumentów funkcji i na tej podstawie dopasować odpowiedni parametr szablonu. Spójrzmy, jak to się odbywa.

Jak to działa

A zatem, skąd kompilator wie, dla jakich parametrów ma skonkretyzować szablon funkcji?… Innymi słowy, skąd bierze on właściwy typ dla funkcji szablonowej? Cóż, nie jest to bardzo skomplikowane:

Parametry szablonu funkcji są **dedukowane** w oparciu o **parametry jej wywołania** oraz niejawne konwersje.

Prześledźmy to na przykładzie wywołania szablonu funkcji:

```
template <typename TYP> TYP max(TYP Parametr1, TYP Parametr2);
```

w kilku formach:

```
max(67, 76)                              // 1
max(5.6, 6.5f)                           // 2
max(8.7f, 9.0f)                          // 3
max("Hello", std::string("world"))       // 4
```

Pierwszy przykład jest jak sądze prosty. Obie liczby są tu typu `int`, zatem użytą tu funkcją `max<int>`. Nie ma żadnych watpliwości.
Dalej jest ciekawiej. Parametry drugiego wywołania funkcji są typu `double` i `float`. Mamy jednak jeden parametr szablonu (`TYP`), który musi przyjąć tą samą „wartość” w wywołaniu funkcji. Co zatem zrobi kompilator? Wykorzysta on to, że między `float` i `double` istnieje niejawna konwersja i wybierze typ `double` jako ogólniejszy. Użytym wariantem będzie więc `max<double>`.
Kolejny przykład… to nic nowego :) Oba argumenty są tu typu `float` (skutek przyrostka `f`), zatem wykorzystaną funkcją będzie `max<float>`.
Ostatnia, czwarta linijka jest zdecydowanie najciekawsza. Napisy `"Hello"` i `"world"` mają z pewnością ten sam typ - `const char[]`. Niemniej, drugi parametr jest typu `std::string`, bowiem jawnie tworzymy obiekt tej klasy przy użyciu konstruktora. Wobec takiego obrotu sprawy kompilator musi pogodzić go z `const char[]`. Robi to, ponieważ

istnieje niejawna konwersja łancucha typu C na `std::string`. Szablon funkcji zostanie więc skonkretyzowany do `max<std::string>`[124].

Ogólny wniosek z tych przykładów jest taki, że jeśli jeden parametr szablonu musi być dopasowany na podstawie kilku różnych typów parametrów funkcji, to kompilator próbuje zastosować niejawne konwersje celem sprowadzenia ich do jakiegoś jednego typu ogólnego. Dopiero jeżeli ta próba się nie powiedzie, sygnalizowany jest błąd.

W zasadzie to trzeba powiedzieć: „jeżeli ta próba się nie powiedzie i nie ma żadnych innych możliwych dopasowań". Możliwe bowiem, że istnieją inne szablony, których parametry pozwalają na problematyczne dopasowanie. Przykładowo, wywołanie `max(18, "tekst")` nie mogłoby być dopasowane do jednoparametrowego szablonu `max()`, ale bez problemu przypasowane zostałoby do szablonu dwuparametrowego `max()`, podanego jakiś czas temu (i poniżej). Ten dopuszczałby przecież różne typy argumentów.
Reguła mówiąca, iż pierwsze niepowodzenie dopasowywania parametrów szablonu nie jest błędem, funkcjonuje pod skrótem SFINAE (ang. *Substitution Failure Is Not An Error* - porażka podstawiania nie jest błędem).

Dedukcja przy wykorzystaniu kilku parametrów szablonu

Proces dedukcji zaczyna nabierać rumieńców, gdy mamy do czynienia z szablonem o większej liczbie parametrów niż jeden. Przypomnijmy sobie szablon funkcji `max()` z dwoma parametrami (deklarację tylko, bo definicja jest chyba oczywista):

```
template <typename TYP1, typename TYP2>
   TYP1 max(TYP1 Parametr1, TYP2 Parametr2);
```

Tutaj wszystko jest nawet znacznie prostsze niż poprzednio. Dzięki temu, że każdy parametr funkcji ma swój własny typ (parametr szablonu), kompilator ma ułatwione zadanie. Nie musi już brać pod uwagę żadnych niejawnych konwersji.

Z powyższym szablonem związanym jest jednak pewien problem. Nie bardzo wiadomo, jaki ma być typ zwracany tej funkcji. Może to być zarówno `TYP1`, jak i `TYP2` - zależy po prostu, która z wartości zwycięży w teście porównawczym. Tego jego nie sposób ustalić w czasie kompilacji; można jednak dodać typ oddawany do parametrów szablonu:

```
template <typename TYP1, typename TYP2, typename ZWROT>
   ZWROT max(TYP1 Parametr1, TYP2 Parametr2);
```

Próba wywołania tej funkcji w zwykłej formie zakończy się jednak błędem - a to dlatego, że ten nowy, trzeci parametr nie może zostać wydedukowany przez kompilator! Mówiłem przecież, że dedukcja dokonywana jest **wyłącznie na podstawie parametrów funkcji**. Wartość zwracana się zatem nie liczy.
„Hmm, to nie jest aż taki problem", odpowiesz może. „Ten jeden parametr mogę przecież podać; wpisze tam po prostu typ ogólniejszy spośród dwóch poprzedzających". Tak się jednak nie da! Nie możemy podać do szablonu ostatniego parametru, gdyż wpierw musielibyśmy podać dwa poprzedzające go:

```
max<int, float, float>(17, 67f);
```

To chyba żadna niespodzianka: analogicznie jest z parametrami funkcji. W ten sposób tracimy jednak wszystkie wspaniałości automatycznej dedukcji parametrów szablonu.

---

[124] Porównywanie dwóch napisów może się wydawać dziwne, ale jest ono poprawne. Klasa `std::string` posiada operator `>`, dokonujący porównania tekstów pod względem ich długości oraz przechowywanych weń znaków (ich kolejności alfabetycznej).

Istnieje aczkolwiek sposób na to. Należy przesunąć parametr `ZWROT` na początek listy parametrów szablonu:

```
template <typename ZWROT, typename TYP1, typename TYP2>
   ZWROT max(TYP1 Parametr1, TYP2 Parametr2);
```

Teraz pozostałe dwa typy mogą być odgadnięte z parametrów funkcji. Tego szablonu `max()` będziemy więc mogli używać, podając tylko typ wartości zwracanej:

```
max<float>(17, 67f);
```

Wynika stąd prosty wniosek:

Dedukcja parametrów szablonu następuje **od końca** (od prawej strony). Te parametry, które mogą być wzięte z wywołania funkcji, powinny zatem znajdować się na końcu listy.

Szablony klas

Szablony funkcji mogą przedstawiać się wcale zachęcająco, jednak o wiele większą zaletą C++ są szablony klas. Ponownie, możemy je traktować jako:

- swego rodzaju ogólne klasy (zwane czasem metaklasami), definiujące zachowanie się obiektów w odniesieniu do dowolnych typów danych
- zespół klas, delegujących odrębne klasy do obsługi różnych typów

Po raz kolejny też to drugie podejście jest bardziej poprawne.

Szablon klasy reprezentuje zestaw (rodzinę) klas, mogących współpracować z różnymi typami danych.

Konieczność istnienia szablonów klas bezpośrednio wynika z faktu, że C++ jest językiem zorientowanym obiektowo. Do potrzeb programowania strukturalnego z pewnością wystarczyłyby szablony funkcji; kiedy jednak chcemy w pełni korzystać z dobrodziejstw OOPu i cieszyć się elastycznością szablonów, naturalnym jest użycie szablonów klas. Z bardziej praktycznego punktu widzenia szablony klas są znacznie przydatniejsze i częściej stosowane niż szablony funkcji. Typowym ich zastosowaniem są klasy pojemnikowe, czyli znane i lubiane struktury danych - a one obok algorytmów, są według klasyków informatyki podstawowymi składnikami programów. Niemniej przez lata istnienia szablony klas dorobiły się także wielu całkiem niespodziewanych zastosowań.

Szablony klas intensywnie wykorzystuje Biblioteka Standardowa języka C++, a także niezwykle popularna biblioteka Boost.

Niezależnie od tego, czy twój kontakt z tymi rodzajami szablonów będzie się ograniczał wyłącznie do pojemników w rodzaju wektorów lub kolejek, czy też wymyślisz dla nich znacznie więcej zastosowań, powinieneś dobrze poznać ten element języka C++. I te temu właśnie służy niniejsza sekcja.

Definicja szablonu klas

Wpierw więc zajmiemy się definiowaniem szablonu klasy. Popatrzmy sobie najpierw na prosty przykład szablonu, będący rozszerzeniem klasy `CIntArray`, przewijającej się przez kilka poprzednich rozdziałów. Dalej zajmiemy się też bardziej zaawansowanymi aspektami definicji szablonów klas.

Prosty przykład tablicy

W rozdziale o wskaźnikach pokazałem ci prostą klasę dynamicznej tablicy `int`-ów - `CIntArray`. Wtedy interesowała nas dynamiczna alokacja pamięci, więc nie przeszkadzał nam fakt nieporęczności tejże klasy. Miała ona bowiem dwa mankamenty: nie pozwalała na użycie nawiasów kwadratowych `[]` celem dostępu do elementów tablicy, no i potrafiła przechowywać wyłącznie liczby typu `int`.
Obiecałem jednocześnie, że w swoim czasie pozbędziemy się obu tych niedogodności. Miałeś się już okazję przekonać, że nie rzucam słów na wiatr, bowiem nauczyliśmy już naszą klasę poprawnie reagować na operator `[]`. Zapewne domyślasz się, że teraz usuniemy drugi z mankamentów i wyposażymy ją w możliwość przechowywania elementów dowolnego typu. Jak nietrudno zgadnąć, będzie to wymagało uczynienia jej szablonem klasy.

Zanim przystąpimy do dzieła, spójrzmy na aktualną wersję naszej klasy:

```
class CIntArray
{
   // domyślny rozmiar tablicy
   static const unsigned DOMYSLNY_ROZMIAR = 5;

   private:
         // wskaźnik na właściwą tablicę oraz jej rozmiar
         int* m_pnTablica;
         unsigned m_uRozmiar;

   public:
         // konstruktory
         explicit CIntArray(unsigned uRozmiar = DOMYSLNY_ROZMIAR)
                           : m_uRozmiar(uRozmiar);
                             m_pnTablica(new int [m_uRozmiar])      { }
         CIntArray(const CIntArray&);

         // destruktor
         ~CIntArray()      { delete[] m_pnTablica; }

         //-------------------------------------------------------------

         // pobieranie i ustawianie elementów tablicy
         int Pobierz(unsigned uIndeks) const
               { if (uIndeks < m_uRozmiar)   return m_pnTablica[uIndeks];
                 else                        return 0;                   }
         bool Ustaw(unsigned uIndeks, int nWartosc)
               { if (uIndeks >= m_uRozmiar) return false;
                 m_pnTablica[uIndeks] = uWartosc;
                 return true;                                    }

         // inne
         unsigned Rozmiar() const      { return m_uRozmiar; }

         //-------------------------------------------------------------

         // operator indeksowania
         int& operator[](unsigned uIndeks)
               { return m_pnTablica[uIndeks]; }

         // operator przypisania (dłuższy, więc nie w definicji)
         CIntArray& operator=(const CIntArray&);
};
```

Przeróbmy ją zatem na szablon.

Definiujemy szablon

Jak więc zdefiniować szablon klasy w C++? Patrząc na ogólną składnię szablonu można by nawet domyślić się tego, lecz spójrzmy na poniższy - pusty na razie - przykład:

```
template <typename TYP> class TArray
{
   // ...
};
```

Jest to szkielet definicji szablonu klasy `TArray`, czyli tablicy dynamicznej na elementy dowolnego typu[125]. Widać tu znane już części: przede wszystkim, fraza `template <typename TYP>` identyfikuje konstrukcję jako szablon i deklaruje parametry tegoż szablonu. Tutaj mamy jeden parametr - będzie nim rzecz jasna typ elementów tablicy.

Dalej mamy właściwie zwykłą definicję klasy i w zasadzie jedyną dobrze widoczną różnicą jest to, że wewnątrz niej możemy użyć nazwy `TYP` - parametru szablonu. U nas będzie on pełnić identyczną rolę jak `int` w `CIntArray`, zatem pełna wersja szablonu `TArray` będzie wyglądała następująco:

```
template <typename TYP> class TArray
{
   // domyślny rozmiar tablicy
   static const unsigned DOMYSLNY_ROZMIAR = 5;

   private:
         // wskaźnik na właściwą tablicę oraz jej rozmiar
         TYP* m_pTablica;
         unsigned m_uRozmiar;

   public:
         // konstruktory
         explicit TArray(unsigned uRozmiar = DOMYSLNY_ROZMIAR)
                        : m_uRozmiar(uRozmiar),
                          m_pTablica(new TYP [m_uRozmiar])   { }
         TArray(const TArray&);

         // destruktor
         ~TArray()   { delete[] m_pTablica; }

         //-------------------------------------------------------------

         // pobieranie i ustawianie elementów tablicy
         TYP Pobierz(unsigned uIndeks) const
               { if (uIndeks < m_uRozmiar)   return m_pTablica[uIndeks];
                 else                        return TYP();              }
         bool Ustaw(unsigned uIndeks, TYP Wartosc)
               { if (uIndeks >= m_uRozmiar) return false;
                 m_pTablica[uIndeks] = Wartosc;
                 return true;                                  }

         // inne
         unsigned Rozmiar() const      { return m_uRozmiar; }

         //-------------------------------------------------------------
```

[125] Litera `T` w nazwie `TArray` to skrót od *template*, czyli 'szablon'.

```
        // operator indeksowania
        TYP& operator[](unsigned uIndeks)
              { return m_pTablica[uIndeks]; }

        // operator przypisania (dłuższy, więc nie w definicji)
        TArray& operator=(const TArray&);
};
```

Możesz być nawet zaskoczony, że było to takie proste. Faktycznie, uczynienie klasy `CIntArray` szablonem ograniczało się do zastąpienia nazwy `int`, użytej jako typ elementów tablicy, nazwą parametru szablonu - `TYP`. Pamiętaj jednak, że nigdy nie powinno się bezmyślnie dokonywać takiego zastępowania; `int` mógł być przecież choćby typem licznika pętli `for` (`for (int i = ...)`) i w takiej sytuacji zastąpienie go przez parametr szablonu nie miałoby żadnego sensu. Nie zapominaj więc, że jak zwykle podczas programowania należy myśleć nad tym, co robimy.

Naturalnie, gdy już opanujesz szablony klas (co, jak sądzę, stanie się niedługo), dojdziesz do wniosku, że wygodniej jest od razu definiować właściwy szablon niż wychodzić od „specjalizowanej" klasy i czynić ją ogólną.

Implementacja metod poza definicją

Szablon jest już prawie gotowy. Musimy jeszcze dodać do niego implementacje dwóch metod: konstruktora kopiującego i operatora przypisania - ze względu na ich długość lepiej będzie, jeśli znajdą się poza definicją. W przypadku zwykłych klas było to jak najbardziej możliwe… a jak jest dla szablonów?

Zapewne nie jest niespodzianką to, iż również tutaj jest to dopuszczalne. Warto jednak uświadomić sobie, że **metody szablonów klas są szablonami metod**. Oznacza to ni mniej, ni więcej, ale to, iż powinniśmy je traktować podobnie, jak szablony funkcji. Wiąże się z tym głównie inna składnia.
Popatrz więc na przykład - oto szablonowa wersja konstruktora kopiującego:

```
template <typename TYP>    TArray<TYP>::TArray(const TArray& aTablica)
{
   // alokujemy pamięć
   m_uRozmiar = aTablica.m_uRozmiar;
   m_pTablica = new TYP [m_uRozmiar];

   // kopiujemy pamięć ze starej tablicy do nowej
   memcpy (m_pTablica, aTablica.m_pTablica, m_uRozmiar * sizeof(TYP));
}
```

I znowu możemy mieć *déja vu*: kod zaczynamy ponownie sekwencją `template <...>`. Łatwo to jednak uzasadnić: mamy tu bowiem do czynienia z szablonem, w którym używamy przecież jego parametru `TYP`. Koniecznie więc musimy użyc wspomnianej sekwencji po to, aby:

- kompilator wiedział, że ma do czynienia z szablonem, a nie zwykłym kodem
- możliwe było użycie nazw parametrów szablonu (tutaj mamy jeden - `TYP`) w jego wnętrzu

Każdy „kawałek szablonu" trzeba zatem zacząć od owego `template <...>`, aby te dwa warunki były spełnione. Jest to może i uciążliwe, lecz niestety konieczne.

Idźmy dalej - zostając jednak nadal w pierwszym wierszu kodu. Jest on nader interesujący z tego względu, że aż trzykrotnie występuje w nim nazwa naszego szablonu, `TArray` - na dodatek ma ona tutaj trzy różne znaczenia. Przenalizujmy je:

- w pierwszym przypadku jest to wyraz `TArray<TYP>`. Jak pamiętamy z szablonów funkcji, takie konstrukcje oznaczają zazwyczaj konkretne egzemplarze szablonu - specjalizacje. W tym jednak wypadku podajemy tu parametr `TYP`, a nie jakiś szczególny typ danych. W sumie cały ten zwrot pełni funkcję **nazwy typu klasy**; potraktuj to po prostu jako obowiązkową część nagłówka, występującą zawsze przed operatorem `::` w implementacji metod. Podobnie było np. z `CIntArray`, gdy chodziło o zwykłe metody zwykłych klas. Zapamiętaj zatem, że:

Sekwencja *nazwa_szablonu*<*typ*> pełni rolę nazwy typu klasy tam, gdzie jest to konieczne.

- drugi raz używamy `TArray` w charakterze nazwy metody - konstruktora. Może to nie być nieco mylące, bo przecież pisząc konstruktory normalnych klas po obu stronach operatora zasięgu podawaliśmy tę samą nazwę. Musisz więc zapamiętać, że:

Konstruktory i **destruktory** w **szablonach klas** mają nazwy odpowiadające **nazwom ich macierzystych szablonów i niczemu więcej**, tzn. **nie zawierają parametrów w nawiasach ostrych**.

- trzeci raz `TArray` jest użyta jako część typu parametru konstruktora kopiującego - `const TArray&`. Być może zabłyśniesz tu kompetencją i krzykniesz, że to niepoprawne i że jeśli chodzi nam o nazwę typu klasy szablonowej, to powinniśmy wstawić `TArray<TYP>`, bo samo `TArray` to tylko nazwa szablonu. Odpowiem jednak, że posunięcie to jest **równie poprawne**; mamy tu do czynienia z tak zwaną nazwą wtrąconą. Polega to na tym, iż:

Sama nazwa szablonu może być stosowana **wewnątrz niego** w tych miejscach, gdzie wymagany jest **typ klasy szablonowej**. Możemy więc posłużyć się nią do skrótowego deklarowania pól, zmiennych czy parametrów funkcji bez potrzeby pisania nawiasów ostrych i nazw parametrów szablonu.

Wobec nagłówka tak ciężkiego kalibru reszta tej funkcji nie przedstawia się chyba bardzo skomplikowanie? :) W rzeczywistości to niemal dokładna kopia treści oryginalnego konstruktora kopiującego - z tym, że typ `int` elementów `CIntArray` zastępuje tutaj nieznany z góry `TYP` - parametr szablonu.

W podobny sposób należałoby jeszcze zaimplementować operator przypisania. Sądzę, że nie sprawiłoby ci problemu samodzielne wykonanie tego zadania.

Korzystanie z tablicy

Gdy mamy już definiowany szablon klasy, chcielibyśmy zapewne skorzystać z niego. Spróbujmy więc stworzyć sobie obiekt tablicy; ponieważ przez cały zajmowaliśmy się tablicą `int`-ów, to teraz niech będzie to tablica napisów:

```
TArray<std::string> aNapisy(3);
```

Jak doskonale wiemy, to co widnieje po lewej stronie jest typem deklarowanej zmiennej. W tym przypadku jest to więc `TArray<std::string>` - specjalizowana wersja naszego szablonu klas. Używamy w niej składni, do której, jak sądzę, zaczynasz się już przyzwyczajać. Po nazwie szablonu (`TArray`) wpisujemy więc parę nawiasów ostrych, a w niej „wartość” parametru szablonu (typ `std::string`). U nas parametr ten określa jednocześnie **typ elementów tablicy** - powyższa linijka tworzy więc trójelementową tablicę łańcuchów znaków.
Całkiem podobnie wygląda tworzenie tablicy ze zmiennych innych typów, np.:

```
TArray<float> aLiczbyRzeczywiste(7);   // 7-el. tablica z liczbami float
TArray<bool> aFlagi(8)                 // zestaw ośmiu flag bool-owskich
TArray<CFoo*> aFoo;                    // tablica wskaźników na obiekty
```

Zwróćmy uwagę, że parametr(y) szablonu - tutaj: typ elementów tablicy - **musimy podać zawsze**. Nie ma możliwości wydedukowania go, bo i skąd? Nie jest to przecież funkcja, której przekazujemy parametry, lecz obiekt klasy, który tworzymy.

Postępowanie z taką tablicą nie różni się niczym od posługiwania się klasą `CIntArray`, a więc pośrednio - również zwykłymi tablicami w C++. W szablonach C++ obowiązują po prostu te same mechanizmy, co w zwykłych klasach: działają przeciążone operatory, niejawne konwersje i reszta tych nietuzinkowych możliwości OOPu. Korzystanie z szablonów klas jest więc nie tylko efektywne i elastycznie, ale i intuicyjne:

```
// wypełnienie tablicy
aNapisy[0] = "raz";
aNapisy[1] = "dwa";
aNapisy[2] = "trzy";

// pokazanie zawartości tablicy
for (unsigned i = 0; i < aNapisy.Rozmiar(); ++i)
   std::cout << aNapisy[i] << std::endl;
```

Przyznasz chyba teraz, że szablony klas przedstawiają się wyjątkowo zachęcająco?… Dowiedzmy się zatem wicej o tych konstrukcjach.

Dziedziczenie i szablony klas

Nowy wspaniały wynalazek języka C++ - szablony - może współpracować ze starym wspaniałym wynalazkiem języka C++ - dziedziczeniem. A tam, gdzie spotykają się dwa wspaniałe wynalazki, musi być doprawdy cudownie :) Zajmijmy się więc dziedziczeniem połączonym z szablonami klas.

Dziedziczenie klas szablonowych

Szablony klas (jak `TArray`) są podstawami do generowania specjalizowanych klas szablonowych (jak np. `TArray<int>`). Ten specjalizowane klasy zasadniczo niczym nie różnią się od innych uprzednio zdefiniowanych klas. Mogą więc na przykład być klasami bazowymi dla nowych typów.

Czas na ilustrację zagadnienia w postaci przykładowego kodu. Oto klasa wektora liczb:

```
class CVector : public TArray<double>
{
   public:
         // operator mnożenia skalarnego
         double operator*(const CVector&);
};
```

Dziedziczy ona z `TArray<double>`, czyli zwykłej tablicy liczb. Dodaje ona jednak dodatkową metodę - przeciążony operator mnożenia `*`, obliczający iloczyn skalarny:

```
double CVector::operator*(const CVector& aWektor)
{
   // jeżeli rozmiary wektorów nie są równe, rzucamy wyjątek
   if (Rozmiar() != aWektor.Rozmiar())
         throw CError(__FILE__, __LINE__, "Blad iloczynu skalarnego");

   // liczymy iloczyn
```

```
    double fWynik = 0.0;
    for (unsigned i = 0; i < Rozmiar(); ++i)
          fWynik += (*this)[i] * aWektor[i];

    // zwracamy wynik
    return fWynik;
}
```

W samym akcie dziedziczenia, jak i w implementacji klasy pochodnej, nie ma żadnych niespodzianek. Używamy po prostu `TArray<double>` tak, jak każdej innej nazwy klasy i możemy korzystać z jej publicznych i chronionych składników. Należy oczywiście pamiętać, że w tej klasie typ `double` występuje tam, gdzie w szablonie `TArray` pojawia się parametr szablonu - `TYP`. Dotyczy to chociażby rezultatu operatora `[]`, który jest właśnie liczbą typu `double`:

```
fWynik += (*this)[i] * aWektor[i];
```

Myślę aczkolwiek, że fakt ten jest intuicyjny i dziedziczenie specjalizowanych klas szablonowych nie będzie ci sprawiać kłopotu.

Dziedziczenie szablonów klas

Szablony i dziedziczenie umożliwiają również tworzenie nowych szablonów klas na podstawie już istniejących, innych szablonów. Na czym polega różnica?… Otóż na tym, że w ten sposób tworzymy **nowy szablon klas**, a nie pojedynczą, zwykłą klasę - jak to się działo poprzednio. Wtedy definiowaliśmy normalną klasę przy pomocy innej, niemalże normalnej klasy - różnica była tylko w tym, że tą klasą bazową była specjalizacja szablonu (`TArray<double>`). Teraz natomiast będziemy konstruowali s**zablon klas pochodnych przy użyciu szablonu klas bazowych**. Cały czas będziemy więc poruszać się w obrebie czysto szablonowego kodu z naszą ulubioną frazą `template <...>` ;)

Oto nasz nowy szablon - tablica, która potrafi dynamicznie zmieniać swój rozmiar w czasie swego istnienia:

```
template <typename TYP> class TDynamicArray : public TArray<TYP>
{
    public:
          // funkcja dokonująca ponownego wymiarowania tablicy
          bool ZmienRozmiar(unsigned);
};
```

Ponieważ jest to szablon, więc rozpoczynamy go od zwyczajowego początku i listy parametrów. Nadal będzie to jeden `TYP` elementów tablicy, ale nic nie stałoby na przeszkodzie, aby lista parametrów szablonu została w jakiś sposób zmodyfikowana. W dalszej kolejności widzimy znajomy początek definicji klasy. Jako klasę bazową wstawiamy tu `TArray<TYP>`. Przypomina to poprzedni punkt, ale pamiętajmy, że teraz korzystamy z parametru szablonu (`TYP`) zamiast z konkretnego typu (`double`). Nazwa klasy bazowej jest więc tak samo „zszablonowana” jak cała reszta definicji `TDynamicArray`.

Pozostaje jeszcze kwestia implementacji metody `ZmienRozmiar()`. Nie powinna być ona niespodzienka, bowiem wiesz już, jak kodować metody szablonów klas poza blokiem ich definicji. Treść funkcji jest natomiast niemal wierną kopią tej z rozdziału o wskaźnikach:

```
template <typename TYP>
    bool TDynamicArray<TYP>::ZmienRozmiar(unsigned uNowyRozmiar)
{
    // sprawdzamy, czy nowy rozmiar jest większy od starego
```

```
    if (!(uNowyRozmiar > m_uRozmiar)) return false;

    // alokujemy nową tablicę
    TYP* pNowaTablica = new TYP [uNowyRozmiar];

    // kopiujemy doń starą tablicę i zwalniamy ją
    memcpy (pnNowaTablica, m_pTablica, m_uRozmiar * sizeof(TYP));
    delete[] m_pTablica;

    // "podczepiamy" nową tablicę do klasy i zapamiętujemy jej rozmiar
    m_pTablica = pNowaTablica;
    m_uRozmiar = uNowyRozmiar;

    // zwracamy pozytywny rezultat
    return true;
}
```

Widzimy więc, że dziedziczenie szablonu klasy nie jest wcale trudne. W jego wyniku powstaje po prostu nowy szablon klas.

Deklaracje w szablonach klas

Pola i metody to najważniejsze składniki definicji klas - także tych szablonowych. Jeżeli jednak chodzi o szablony, to znacznie częściej możemy tam spotkać również inne deklaracje. Trzeba się im przyjrzeć, co teraz uczynimy.

Ten paragraf możesz pominąć przy pierwszym podejściu do lektury, jeśli wyda ci się zbyt trudny, i przejść dalej.

Aliasy `typedef`

Cechą wyróżniającą szablony jest to, iż operują one na typach danych w podobny sposób, jak inny kod na samych danych. Naturalnie, wszystkie te operacje są przeprowadzane w czasie kompilacji programu, a ich większą częścią jest konkretyzacja - tworzenie specjalizowanych wersji funkcji i klas na podstawie ich szablonów.

Proces ten sprawia jednocześnie, że niektóre przewidywalne i, zdawałoby się, znajome konstrukcje językowe nabierają nowych cech. Należy do nich choćby instrukcja `typedef`; w oryginale służy ona wyłącznie do tworzenia alternatywnych nazw dla typów np. tak:

```
typedef void* PTR;
```

Nie jest to żadna rewolucja w programowaniu, co zresztą podkreślałem, prezentując tę instrukcję. Ciekawie zaczyna się robić dopiero wtedy, jeśli uświadomimy sobie, że aliasowanym typem może być… parametr szablonu! Ale skąd on pochodzi?

Oczywiście - z szablonu klasy. Jeżeli bowiem umieścimy `typedef` wewnątrz definicji takiego szablonu, to możemy w niej wykorzystać parametryzowany typ. Oto najprostszy przykład:

```
template <typename TYP> class TArray
{
    public:
          // alias na parametr szablonu
          typedef TYP ELEMENT;

    // (reszta nieważna)
};
```

Instrukcja `typedef` pozwala nam wprowadzenie czegoś w rodzaju „składowej klasy reprezentującej typ". Naturalnie, jest to tylko składowa w sensie przenośnym, niemniej nazwa `ELEMENT` zachowuje się wewnątrz klasy i poza nią jako pełnoprawny typ danych - równoważny parametrowi szablonu, `TYP`.
Przydatność takiego aliasu może się aczkolwiek wydawać wątpliwa, bo przecież łatwiej i krócej jest pisać nazwę typu `float` niż `TArray<float>::ELEMENT`. `typedef` wewnątrz szablonu klasy (lub nawet ogólnie - w odniesieniu do szablonów) ma jednak znacznie sensowniejsze zastosowania, gdy współpracuje ze soba wiele takich szablonów. Koronnym przykładem jest Biblioteka Standardowa C++, gdzie w ten sposób całkiem można zyskać dostęp m.in. do tzw. iteratorów, wspomagającym pracę ze strukturami danych.

Deklaracje przyjaźni

Częściej spotykanym elementem w zwykłych klasach są deklaracje przyjaźni. Naturalnie, w szablonach klas nie moglo ich zabraknąć. Możemy tutaj również deklarować przyjaźnie z funkcjami i klasami.

Dodatkowo możliwe jest (obsługują to nowsze kompilatory) uczynienie deklaracji przyjaźni **szablonową**. Oto przykład:

```
template <typename T> class TBar       { /* ... */ };

template <typename T> class TFoo
{
   // deklaracja przyjaźni z szablonem klasy TBar
   template <typename U> friend class TBar<U>;
};
```

Taka deklaracja sprawia, że wszystkie specjalizacje szablonu `TBar` będą zaprzyjaźnione ze wszystkimi specjalizacjami szablonu `TFoo`. `TFoo<int>` będzie więc miała dostęp do niepublicznych składowych `TBar<double>`, `TBar<unsigned>`, `TBar<std::string>` i wszystkich innych specjalizacji szablonu `TBar`.
Zauważmy, że nie jest to równowaznaczne z zastosowaniem deklaracji:

```
friend class TBar<T>;
```

Ona spowoduje tylko, że zaprzyjaźnione zostaną te egzemplarze szablonów `TBar` i `TFoo`, które konkretyzowano z tym samym parametrem `T`. `TBar<float>` będzie więc zaprzyjaźniony z `TFoo<float>`, ale np. z `TFoo<short>` czy z jakąkolwiek inną specjalizacją `TFoo`.

Szablony funkcji składowych

Istnieje bardzo ciekawa możliwość[126]: metody klas mogą być szablonami. Naturalnie, możesz pomyśleć, że to żadna nowość, bo przecież w przypadku szablonów klas wszystkie ich metody są swego rodzaju szablonami funkcji. Chodzi jednak o coś innego, co najlepiej zobaczymy na przykładzie.

Nasz szablon `TArray` działa całkiem znośnie i umożliwia podstawową funkcjonalność w zakresie tablic. Ma jednak pewną wadę; spójrzmy na poniższy kod:

```
TArray<float> aFloaty1(10), aFloaty2;
TArray<int> aInty(7);
```

[126] Dostępna aczkolwiek tylko w niektórych kompilatorach (np. w Visual C++ .NET 2003), podobnie jak szablony deklaracji przyjaźni.

```
// ...

aFloaty1 = aFloaty2;        // OK, przypisujemy tablicę tego samego typu
aFloaty2 = aInty;           // BŁĄD! TArray<int> niezgodne z TArray<float>
```

Drugie przypisanie tablicy `int`-ów do tablicy `float`-ów nie jest dopuszczalne. To niedobrze, ponieważ, logicznie rzecz ujmując, powinno to być jak najbardziej możliwe. Kopiowanie mogłoby się przecież odbyć poprzez przepisanie poszczególnych liczb - elementów tablicy `aInty`. Konwersja z `int` do `float` jest bowiem jak całkowicie poprawna i nie powoduje żadnych szkodliwych efektów.
Kompilator jednak tego nie wie, gdyż w szablonie `TArray` zdefiniowaliśmy operator przypisania **wyłącznie dla tablic tego samego typu**. Musielibyśmy więc dodać kolejną jego wersję - tym razem uniwersalną, szablonową. Dzięki temu w razie potrzeby można by jej użyć w takich właśnie przypisaniach. Jak to zrobić? Spójrzmy:

```
template <typename T> class TArray
{
   public:
         // szablonowy operator przypisania
         template <typename U>
               TArray<T>& operator=(const TArray<U>&);

   // (reszta nieważna)
};
```

Mamy więc tutaj znowu zagnieżdżoną deklarację szablonu. Druga fraza `template <...>` jest nam potrzebna, aby uniezależnić od typu operator przypisania - uniezależnić nie tylko w sensie ogólnym (jak to ma miejsce w całym szablonie `TArray`), ale też w znaczeniu możliwej „inności” parametru tego szablonu (`U`) od parametru `T` macierzystego szablonu `TArray`. Zatem przykładowo: jeżeli zastosujemy przypisanie tablicy `TArray<int>` do `TArray<float>`, to `T` przyjmie „wartość” `float`, zaś `U` - `int`.

Wszystko jasne? To teraz czas na smakowity deser. Powyższy szablon metody trzeba jeszcze zaimplementować. No i jak to zrobić?… Cóż, nic prostszego. Napiszmy więc tę funkcję.
Zaczynamy oczywiście od `template <...>`:

```
template <typename T>
```

W ten sposób niejako otwieramy pierwszy z szablonów - czyli TArray. Ale to jeszcze nie wszystko: mamy przecież w nim kolejny szablon - operator przypisania. Co z tym począć?… Ależ tak, potrzebujemy **drugiej frazy `template <...>`**:

```
template <typename T>            // od szablonu klasy TArray
   template <typename U>         // od szablonu operatora przypisania
```

Ślicznie to wygląda, no ale to jeszcze nie wszystko. Dalej jednak jest już, jak sądzę, prosto. Piszemy bowiem zwykły nagłówek metody, posiłkując się prototypem z definicji funkcji. A zatem:

```
template <typename T>
   template <typename U>
         TArray<T>& TArray<T>::operator=(const TArray<U>& aTablica)
         {
               // ...
         }
```

Stosuję tu takie dziwne formatowanie kodu, aby unaocznić ci jego najważniejsze elementy. W normalnej praktyce możesz rzecz jasna skondensować go bardziej, pisząc np. obie klauzule `template <...>` w jednym wierszu i nie wcinając kodu metody.

Wreszcie, czas na ciało funkcji - to chyba najprostsza część. Robimy podobnie, jak w normalnym operatorze przypisania: najpierw niszczymy własną tablicę obiektu, tworzymy nową dla przypisywanej tablicy i kopiujemy jej treść:

```
template <typename T>
   template <typename U>
         TArray<T>& TArray<T>::operator=(const TArray<U>& aTablica)
         {
               // niszczymy własną tablicę
               delete[] m_pTablica;

               // tworzymy nową, o odpowiednim rozmiarze
               m_uRozmiar = aTablica.Rozmiar();
               m_pTablica = new T [m_uRozmiar];

               // przepisujemy zawartość tablicy przy pomocy pętli
               for (unsigned i = 0; i < m_uRozmiar; ++i)
                     m_pTablica = aTablica[i];

               // zwracamy referencję do własnego obiektu
               return *this;
         }
```

Niespodzianek raczej brak - może z wyjątkiem pętli użytej do kopiowania zawartości. Nie posługujemy się tutaj `memcpy()` z prostgo powodu: chcemy, aby przy przepisywaniu elementów zadziałały niejawne konwersje. Dokonują się one oczywiście w linijce:

```
m_pTablica = aTablica[i];
```

To właśnie ona sprawi, że w razie niedozwolonego przypisywania tablic (np. `TArray<std::string>` do `TArray<double>`) kompilacja nie powiedzie. Natomiast we wszystkich innych przypadkach, jeśli istnieją niejawne konwersje między elementami tablicy, wszystko będzie w porządku.

Do pełnego szczęścia należałoby jeszcze w podobny sposób zdefiniować konstruktor konwertujący (albo kopiujący - zależy jak na to patrzeć), będący również szablonem metody. To oczywiście zadanie dla ciebie :)

Korzystanie z klas szablonowych

Zdefiniowanie szablonu klasy to naturalnie dopiero połowa sukcesu. Pieczołowicie stworzony szablon chcemy przecież wykorzystać w praktyce. Porozmawiajmy więc, jak to zrobić.

Tworzenie obiektów

Najbardziej oczywistym sposobem korzystania z szablonu klasy jest tworzenie obiektów bazujących na specjalizacji tegoż szablonu.

Stwarzamy obiekt klasy szablonowej

W kreowaniu obiektów klas szablonowych nie ma niczego nadzwyczajnego; robiliśmy to już kilkakrotnie. Zobaczmy na najprostszy przykład - utworzenia tablicy elementów typu `long`:

```
TArray<long> aLongi;
```

`long` jest tu parametrem szablonu `TArray`. Jednocześnie cały wyraz `TArray<long>` jest typem zmiennej `aLongi`. Analogia ze zwykłych typów danych działa więc tak samo dla klas szablonowych.

Docierając do tego miejsca pewnie przypomniałeś już sobie o wskaźniku `std::auto_ptr` z poprzedniego rozdziału. Patrząc na instrukcję jego tworzenia nietrudno wyciągnąć wniosek: `auto_ptr` jest również szablonem klasy. Parametrem tego szablonu jest zaś typ, na który wskaźnik pokazuje.

Przy okazji tego banalnego punktu zwrócę jeszcze uwagę na pewien „fakt składniowy". Przypuśćmy więc, że zapragniemy stworzyć przy użyciu naszego szablonu tablicę dwuwymiarową. Pamiętając o tym, że w C++ tablice wielowymiarowe są obsługiwane jako tablice tablic, wyprodukujemy zapewne coś w tym rodzaju:

```
TArray<TArray<int>> aInty2D;     // no i co tu jest źle?...
```

Koncepcyjnie wszystko jest tutaj w porządku: `TArray<int>` jest po prostu parametrem szablonu, czyli określa tym elementów tablicy - mamy więc tablicę tablic elementów typu `int`. Nieoczekiwanie jednak kompilator wykazuje się tu kompletną ignoracją i zupełnym brakiem ogłady: problematyczne stają się bowiem dwa zamykające nawiasy ostre, umieszczone obok siebie. Są one interpretowane jako… uwaga… **operator przesunięcia bitowego w prawo**! Wiem, że to brzmi idiotycznie, bo przecież w tym kontekście operator ten jest zupełnie niemożliwy do zastosowania. Muszę więc przeprosić cię za większość nierozgarniętych kompilatorów, które w tym kontekście interpretują sekwencję `>>` jako operator bitowy[127].
No dobrze, ale co z tym fantem zrobić?… Otóż rozwiązanie jest nadzwyczaj proste: trzeba **oddzielić oba znaki**, aby nie mogło już dochodzić do nieporozumień na linii kompilator-programista:

```
TArray<TArray<int> > aInty2D;    // i teraz jest OK
```

Może wygląda to nieadnie, ale póki co należy tak właśnie pisać. Zapamiętaj więc, że:

W miejsach, gdy w **kodzie używającym szablonów** mają wystąpić **obok siebie dwa ostre nawiasy zamykające** (>>), należy wstawić między nimi **spację** (> >), by nie pozwolić na ich interpretację jako operatora przesunięcia.

O tym i o podobnych lapsusach językowych napomknę więcej w stosownym czasie.

Co się dzieje, gdy tworzymy obiekt szablonu klasy

Aby ten paragraf nie był jedynie prezentacją rzeczy oczywistych (tworzenie obiektów klas szablowych) i denerwujących (*vide* kwestia nawiasów ostrych), powiedzmy sobie jeszcze o jednej sprawie. Co w zasadzie dzieje się, gdy w kodzie napotka kompilator na instrukcję tworzącą obiekt klasy szablonowej?…
Oczywiście, ogólna odpowiedź brzmi „generuje odpowiedni kod maszynowy". Warto jednak zagłębić się nieco w szczegóły, bo dzięki temu spotka nas pewna miła niespodzianka…

A zatem - co się dzieje? Przede wszystkim musimy sobie uświadomić fakt, że takie „nazwy" jak `TArray<int>`, `TDynamicArray<double>` i inne nazwy szablonów klas z

[127] To właściwie problem nie tylko kompilator, ale samego Standardu C++, który pozwala im na takie beztroskie zachowanie. Pozostaje mieć nadzieję, że to się zmieni…

podanymi parametrami **nie reprezentują klas istniejących w kodzie programu**. Są one tylko instrukcjami dla kompilatora, mówiącymi mu, by wykonał dwie czynności:

- odnalazł wskazany szablon klas (`TArray`, `TDynamicArray` …) i sprawdził, czy podane mu parametry są poprawne
- wykonał jego konkretyzację, czyli wygenerował odpowiednie klasy szablonowe

Właściwe klasy są więc tworzone dopiero w czasie kompilacji - działa to na nieco podobnej zasadzie, jak rozwijanie makr preprocesora, choć jest oczywiście znacznie bardziej zaawansowane. Najważniejsze dla nas, programistów nie są jednak szczegóły tego procesu, lecz jedna cecha kompilatora - bardzo dla nas korzystna.
A chodzi o to, że kompilator jest… **leniwy** (ang. *lazy*)! Jego lenistwo polega na tym, że wykonuje on wyłącznie tyle pracy, ile jest konieczne do poprawnej kompilacji - i nic ponadto. W przypadku szablonów klas znaczy to po prostu tyle, że:

Konkretyzacji podlegają **tylko te składowe klasy**, które są faktycznie **używane**.

ten bardzo przyjemny dla nas fakt najlepiej zrozumieć, jeżeli przez chwilę wczujemy się w rolę leniwego kompilatora. Przypuśćmy, że widzi on taką deklarację:

```
TArray<CFoo> aFoos;
```

Naturalnie, odszukuje on szablon `TArray`; przypuśćmy, że stwierdza przy tym, iż dla typu `CFoo` nie był on jeszcze konkretyzowany. Innymi słowy, nie posiada definicji klasy szablonowej dla tablicy elementów typu `CFoo`. Musi więc ją stworzyć. Cóż więc robi? Otóż w pocie czoła generuje on dla siebie kod w mniej więcej takiej postaci[128]:

```
class TArray<CFoo>
{
	static const unsigned DOMYSLNY_ROZMIAR = 5;

	private:
		CFoo* m_pTablica;
		unsigned m_uRozmiar;

	public:
		explicit TArray(unsigned uRozmiar = DOMYSLNY_ROZMIAR)
				: m_uRozmiar(uRozmiar), m_pTablica(new CFoo [m_uRozmiar])	{ }

};
```

„Chwila! A gdzie są wszystkie pozostałe metody?!” Możesz się zaniepokoić, ale poczekaj chwilę… Powiedzmy, że oto dalej spotykamy instrukcję:

```
aFoos[0] = CFoo("Foooo!");
```

Co wtedy? Wracamy mianowicie do wygenerowanej przed chwilą definicji, a kompilator ją modyfikuje i teraz wygląda ona tak:

```
class TArray<CFoo>
{
	static const unsigned DOMYSLNY_ROZMIAR = 5;

	private:
		CFoo* m_pTablica;
		unsigned m_uRozmiar;
```

[128] Domniemane produkty pracy kompilatora zapisuję bez charakterystycznego formatowania.

```
        public:
                explicit TArray(unsigned uRozmiar = DOMYSLNY_ROZMIAR)
                                : m_uRozmiar(uRozmiar), m_pTablica(new CFoo [m_uRozmiar])    { }

                CFoo& operator[](unsigned uIndeks)      { return m_pTablica[uIndeks]; }
};
```

Wreszcie kompilator stwierdza, że wyszedł poza zasięg zmiennej `aFoos`. Co wtedy dzieje się z naszą klasą? Spójrzmy na nią:

```
class TArray<CFoo>
{
        static const unsigned DOMYSLNY_ROZMIAR = 5;

        private:
                CFoo* m_pTablica;
                unsigned m_uRozmiar;

        public:
                explicit TArray(unsigned uRozmiar = DOMYSLNY_ROZMIAR)
                                : m_uRozmiar(uRozmiar), m_pTablica(new CFoo [m_uRozmiar])    { }
                ~TArray()       { delete m_pTablica; }

                CFoo& operator[](unsigned uIndeks)      { return m_pTablica[uIndeks]; }
};
```

Czy już rozumiesz? Przypuszczam, że tak. Zaakcentujmy jednak to ważne stwierdzenie:

> Kompilator konkretyzuje **wyłącznie te metody klasy szablonowej**, które są **używane**.

Korzyść z tego faktu jest chyba oczywista: generowanie tylko potrzebnego kodu sprawia, że w ostatecznym rozrachunku jest go mniej. Programy są więc mniejsze, a przez to także szybciej działają. I to wszystko dzięki lenistwu kompilatora! Czy więc nadal można podzielać pogląd, że ta cecha charakteru jest tylko przywarą? :)

Funkcje operujące na obiektach klas szablonowych

Szablony funkcji są często przystosowane do manipulowanai obiektami klas szablonowych - w zbliżony sposób, w jaki czynią to zwykłe funkcje z normalnymi klasami. Popatrzmy na ten oto przykład funkcji `Szukaj()`:

```
template <typename TYP> int Szukaj(const TArray<TYP>& aTablica,
                                   TYP Szukany)
{
    // przelatujemy po tablicy i porównujemy elementy
    for (unsigned i = 0; i < aTablica.Rozmiar(); ++i)
          if (aTablica[i] == Szukany)
                return i;

    // jeśli nic nie znajdziemy, zwracamy -1
    return -1;
}
```

Sama jej treść do szczególnie odkrywczych nie należy, a przeznaczenie jest, zdaje się, oczywiste. Spójrzmy raczej na nagłówek, bo to on sprawia, że mówimy o tym szablonie w kategoriach współpracy z szablonem klas `TArray`. Oto bowiem parametr szablonu `TYP`

używany jest jako parametr od `TArray` (między innymi). Dzięki temu mamy więc ogólną funkcję do pracy z dowolnym rodzajem tablicy.

Taka współpraca pomiędzy szablonami klas i szablonami funkcji jest naturalna. Gdziekolwiek bowiem umieścimy frazę `template <...>`, powoduje ona uniezależnienie kodu od konkretnego typu danych. A jeśli chcemy tą niezależność zachować, to nieuknione jest tworzenie kolejnych szablonów. W ten sposób skonstruowanych jest mnóstwo bibliotek języka C++, z Biblioteką Standardową na czele.

Specjalizacje szablonów klas

Teraz porozmawiamy sobie o definiowaniu specjalnych wersji szablonów klas dla określonych parametrów (typów). Mechanizm ten działa dość podobnie jak w przypadku szablonów funkcji, więc nie powinno być z tym zbyt wielu problemów.

Specjalizowanie szablonu klasy

Specjalizacja szablonu klasy oznacza ni mniej więcej, jak tylko zdefiniowanie pewnej szczególnej wersji tegoż szablonu dla pewnego wyjątkowego typu (parametru szablonu). Dodatkowo, istnieje możliwość specjlizacji pojedynczej metody; zajmiemy się pokrótce oboma przypadkami.

Własna klasa specjalizowana

Jako przykład na własną, kompletną specjalizację szablonu klasy posłużymy się oczywiście naszym szablonem tablicy jednowymiarowej - `TArray`. Działa on całkiem dobrze w ogólnej wersji, lecz przecież chcemy zdefiniować jego specjalizację. W tym przypadku może to być sensowne w odniesieniu do typu `char`. Tablica elementów tego typu jest bowiem niczym innym, jak tylko łańcuchem znaków. Niestety, w obecnej formie klasa `TArray<char>` nie może być jako traktowana napis (obiekt `std::string`), bo dla kompilatora nie ma teraz żadnej praktycznej różnicy między wszystkimi typami tablic `TArray`.

Aby to zmienić, musimy rzecz jasna wprowadzić swoją własną specjalizację `TArray` dla parametru `char`. Klasa ta będzie różniła się od wersji ogólnej tym, iż wewnątrznym mechanizmem przechowywania tablicy będzie nie tablica dynamiczna typu `char*` (w ogólności: `TYP*`), lecz napis w postaci obiektu `std::string`. Pozwoli to na dodanie operatora konwersji, aczkolwiek zmieni nieco kilka innych metod klasy. Spójrzmy więc na tę specjalizację:

```
#include <string>

template<> class TArray<char>
{
   static const unsigned DOMYSLNY_ROZMIAR = 5;

   private:
         // rzeczona tablica w postaci napisu std::string
         std::string m_strTablica;

   public:
         // konstruktor
         explicit TArray(unsigned uRozmiar = DOMYSLNY_ROZMIAR)
                        : m_strTablica(uRozmiar, '\0') { }
         // (destruktor niepotrzebny)

         //-------------------------------------------------------------

         // (pomijam metody Pobierz() i Ustaw())
```

```
            unsigned Rozmiar() const
                  { return static_cast<unsigned>(m_strTablica.length()); }
            bool ZmienRozmiar(unsigned);

            //-------------------------------------------------------------

            // operator indeksowania
            char& operator[](unsigned uIndeks)  { return m_strTablica[i]; }

            // operator rzutowania na typ std::string
            operator std::string() const        { return m_strTablica; }
};
```

Cóż można o niej powiedzieć?… Naturalnie, rozpoczynamy ją, jak każdą specjalizację szablonu, od frazy `template<>`. Następnie musimy jawnie podać parametry szablonu (`char`), czyli nazwę klasy szablonowej (`TArray<char>`). Wymóg ten istnieje, bo definicja tej klasy może być **zupełnie różna** od definicji oryginalnego szablonu!

Popatrzmy choćby na naszą specjalizację. Nie używamy już w niej tablicy dynamicznej inicjowanej podczas wywołania konstruktora. Zamiast tego mamy obiekt klasy `std::string`, któremu w czasie tworzenia tablicy każemy przechowywać podaną liczbę znaków. Fakt, że sami nie alokujemy pamięci sprawia też, że i sami nie musimy jej zwalniać: napis `m_strTablica` usunie się sam - zatem destruktor jest już niepotrzebny. Poza tym nie ma raczej wielu niespodzianek. Do najciekawszych należy pewnie operator konwersji na typ `std::string` - dzięki niemu tablica `TArray<char>` może być używana tam, gdzie konieczny jest łańcuch znaków C++. Dodanie tej niejawnej konwersji było głównym powodem tworzenia własnej specjalizacji; jak widać, założony cel został osiągnięty łatwo i szybko.

Pozostaje jeszcze do zrobienia implementacja metody `ZmienRozmiar()`, którą umieścimy poza blokiem klasy. Kod wyglądać może tak:

```
bool TArray<char>::ZmienRozmiar(unsigned uNowyRozmiar)
{
   try
   {
         // metoda resize() klasy std::string zmienia długość napisu
         m_strTablica.resize (uNowyRozmiar, '\0');
   }
   catch (std::length_error&)
   {
         // w razie niepowodzenia zmiany rozmiaru zwracamy false
         return false;
   }

   // gdy wszystko się uda, zwracamy true
   return true;
}
```

Od razu zwróćmy uwagę na **brak klauzuli `template<>`**. Nie ma jej, bowiem tutaj nie mamy do czynienia ze specjalizacją szablonu `ZmienRozmiar()`. Metoda ta jest po prostu zwykłą funkcją klasy `TArray<char>` - podobnie było zresztą w oryginalnym szablonie `TArray`. Implementujemy ją więc jako normalną metodę. Nie ma tu zatem znaczenia fakt, że metoda ta jest częścią specjalizacji szablonu klasy. Najlepiej jest po prostu zapamiętać, że dany szablon **specjalizujemy raz** i to wystarczy; gdybyśmy także tutaj spróbowali dodać `template<>`, to przecież byłoby tak, jakbyśmy ponownie chcieli

sprecyzować fragment czegoś (metodę), co już zostało precyzyjnie określone jako całość (klasa).
Co do treści metody, to używamy tutaj funkcji `std::string::resize()` do zmiany rozmiaru napisyu. Funkcja ta może rzucić wyjątek w przypadku niepowodzenia. My ten wyjątek „przerabiamy" na rezultat funkcji: `false`, jeśli wystąpi, i `true`, gdy wszystko się uda.

Specjalizacja metody klasy

Przyglądając się uważnie specjalizacji `TArray` dla typu char można odnieść wrażenie, że przynajmniej częściowo został on stworzony poprzez skopiowanie składowych z definicji samego szablonu `TArray`. Przykładowo, funkcja dla operatora `[]` jest praktycznie identyczna z tą zamieszczoną w ogólnym szablonie (kwestia nazwy `m_strTablica` czy `m_pTablica` jest przecież czysto symboliczna).
To może nam się nieszczególnie podobać, ale jest do przyjęcia. Gorzej, jeśli w klasie specjalizowanej chcemy napisać nieco inną wersję **tylko jednej metody** z pierwotnego szablonu. Czy wówczas jesteśmy skazani na specjalizowanie całej klasy oraz niewygodne kopiowanie i bezsensowne zmiany prawie całego jej kodu?…

Odpowiedź brzmi na szczęście „Nie!" Niechęć do jednej metody szablonu klasy nie oznacza, że musimy obrażać się na ów szablon jako całość. Możliwe jest **specjalizowanie metod** dla szablonów klas; wyjaśnijmy to na przykładzie.

Przypuśćmy mianowicie, że zachciało nam się, aby tablica `TArray` zachowywała się w specjalny sposób w odniesieniu do elementów będacych wskaźnikami typu `int*`. Otóż pragniemy, aby przy niszczeniu tablicy zwalniana była także pamięć, do której odnoszą się te wskaźniki (elementy tablicy). Nie rozwodźmy się nad tym, na ile jest to dorzeczne programistycznie, lecz zastanówmy się raczej, jak to wykonać. Chwila zastanowienia i rozwiązanie staje się jasne: potrzebujemy trochę zmienionej wersji destruktora. Powinien on jeszcze przed usunięciem samej tablicy zadbać o zwolnienie pamięci przynależnej zawartym weń wskaźnikom. Zmiana mała, lecz ważna.
Musimy więc zdefiniować nową wersję destruktora dla klasy `TArray<int*>`. Nie jest to specjalnie trudne:

```
template<> TArray<int*>::~TArray()
{
   // przelatujemy po elementach tablicy (wskaźnikach) i każdemu
   // aplikujemy operator delete
   for (unsigned i = 0; i < Rozmiar(); ++i)
         delete m_pTablica[i];

   // potem jak zwykle usuwamy też samą tablicę
   delete[] m_pTablica;
}
```

Jak to zwykle w specjalizacjach, zaczynamy od `template<>`. Dalej widzimy natomiast normalną w zasadzie definicję destruktora. To, iż jest ona specjalizacją metody dla `TArray` z parametrem `int*` rozpoznajemy rzecz jasna po nagłówku - a dokładniej, po nazwie klasy: `TArray<int*>`.
Reszta nie jest chyba zaskoczeniem. W destruktorze `TArray<int*>` wpierw więc przechodzimy po całej tablicy, stosując operator `delete` dla każdego jej elementu (wskaźnika). W ten sposób zwalniamy bloki pamięci (zmienne dynamiczne), na które pokazują wskaźniki. Z kolei po skończonej robocie pozbywamy się także samej tablicy - dokładnie tak, jak to czyniliśmy w szablonie `TArray`.

Częściowa specjalizacja szablonu klasy

Pełna specjalizacja szablonu oznacza zdefiniowanie klasy dla konkretnego, precyzyjnie określonego zestawy argumentów. W naszym przypadku było to dosłowne podanie typ elementów tablict `TArray`, np. `char`.
Czasem jednak taka precyzja nie jest pożądana. Niekiedy zdarza się, że wygodniej byloby wprowadzić bardziej szczegółową wersję szablonu, która nie operowałaby przy tym konkretnymi typami. Wtedy właśnie wykorzystujemy **specjalizację częściową** (ang. *partial specialization*). Zobaczymy to tradycyjnie na odpowiednim przykładzie.

Problem natury tablicowej

A zatem… Nasz szablon tablicy `TArray` sprawdza się całkiem dobrze. Dotyczy to szczególnie prostych zastosowań, do których został pierwotnie pomyślany - jak na przykład jednowymiarowa tablica liczb czy lista napisów. Idąc dalej, łatwo można sobie jednak wyobrazić bardziej zaawansowane wykorzystanie tego szablonu - w tym także jako dwuwymiarowej tablicy tablic, np.:

```
TArray<TArray<int> > aInty2D;
```

Naturalnie chcielibyśmy, aby taka funkcjonalnośc była nam dana niejako „z urzędu”, z samej tylko definicji `TArray`. Zdawałoby się zresztą, że wszystko jest tutaj w porzadku i że faktycznie możemy się posługiwać zmienną `aInty2D` jak tablicą o dwóch wymiarach. Niestety, nie jest tak różowo; mamy tu przynajmniej dwa problemy.
Po pierwsze: w jaki sposób mielibyśmy ustalić rozmiar(y) takiej tablicy?… Typ zmiennej `aInty2D` jest tu wprawdzie „podwójny”, ale przy jej tworzeniu nadal używany jest normalny konstruktor `TArray`, który jest jednoparametrowy. Możemy więc podać wyłącznie jeden wymiar tablicy, zaś drugi zawsze musiałby być równy wartości domyślnej!
Oprócz tego oczywistego błędu (całkowicie wykluczającego użycie tablicy) mamy jeszcze jeden mankament. Mianowicie, zawartość tablicy nie jest rozmieszczona w pamięci w postaci jednego bloku, jak to czyni kompilator w wypadku statycznych tablic. Zamiast tego każdy jej wiersz (podtablica) jest umieszczony w innym miejscu, co przy większej ich liczbie, rozmiarach i częstym dostępie będzie ujemnie odbijało się na efektywności kodu.

Rozwiązanie: przypadek szczególniejszy, ale nie za bardzo

Co można na to poradzić? Rozwiązaniem jest specjalne potraktowanie klasy `TArray<TArray<`*`typ_elementu`*`> >` i zdefiniowanie jej odmiennej postaci - nieco innej niż wyjściowy szablon `TArray`. Przedtem jednak zwróćmy uwagę, iż nie możemy tutaj zastosować całkowitej specjalizacji tegoż szablonu, bowiem *`typ_elementu`* nadal jest tu parametrem o dowolnej „wartości” (typie).

Jak się pewnie domyślasz, trzeba tu zastosować specjalizację częściową. Będzie ona traktowała zagnieżdżone szablony `TArray` w specjalny sposób, zachowując jednak możliwość dowolnego ustalania typu elementów tablicy. Popatrzmy więc na definicję tej specjalizacji:

```
template <typename TYP>    class TArray<TArray<TYP> >
{
   static const unsigned DOMYSLNY_ROZMIAR = 5;

   private:
         // wskaźnik na tablicę
         TYP* m_pTablica;

         // wymiary tablicy
         unsigned m_uRozmiarX;
```

```
        unsigned m_uRozmiarY;

    public:
        // konstruktor i destruktor
        explicit TArray(unsigned uRozmiarX = DOMYSLNY_ROZMIAR,
                        unsigned uRozmiarY = DOMYSLNY_ROZMIAR)
            : m_uRozmiarX(uRozmiarX), m_uRozmiar(uRozmiarY),
              m_pTablica(new TYP [uRozmiarX * uRozmiarY])   { }
        ~TArray()   { delete[] m_pTablica; }

        //-------------------------------------------------------------

        // metody zwracające wymiary tablicy
        unsigned RozmiarX() const     { return m_uRozmiarX; }
        unsigned RozmiarY() const     { return m_uRozmiarY; }

        //-------------------------------------------------------------

        // operator () do wybierania elementów tablicy
        TYP& operator()(unsigned uX, unsigned uY)
            { return m_pTablica[uY * m_uRozmiarX + uX]; }

        // (pomijam konstruktor kopiujący i operator przypisania}
};
```

Tak naprawdę to w opisywanej sytuacji specjalizacja częściowa niekoniecznie może być uznawana za najlepsze rozwiązanie. Dość logiczne jest bowiem zdefiniowanie sobie zupełnie nowego szablonu, np. `TArray2D` i wykorzystywanie go zamiast misternej konstrukcji `TArray<TArray<...> >`. Ponieważ jednak masz tutaj przede wszystkim poznać zagadnienie specjalizacji częściowej, wyłącz na chwilę swój nazbyt czuły wykrywacz naciąganych rozwiązań i w spokoju kontynuuj lekturę :D

Rozpoczyna się ona od sekwencji `template <typename TYP>` (a nie `template<>`), co może budzić zaskoczenie. W rzeczywistości jest to logiczne i niezbędne: to prawda, że mamy do czynienia ze specjalizacją szablonu, jednak jest to specjalizacja częściowa, zatem nie określamy *explicité* wszystkich jego parametrów. Nadal więc posługujemy się faktycznym szablonem - choćby w tym sensie, że typ elementów tablicy pozostaje nieznany z góry i musi podlegać parametryzacji jako `TYP`. Klauzula `template <typename TYP>` jest zatem niezbędna - podobnie zresztą jak we wszystkich przypadkach, gdy tworzymy kod niezależny od konkretnego typu danych.

Tutaj klauzula ta wygląda tak samo, jak w oryginalnym szablnoie `TArray`. Warto jednak wiedzieć, że nie musi wcale tak być. Przykładowo, jeśli specjalizowalibyśmy szablon o dwóch parametrach, wówczas fraza `template <...>` mogłaby zawierać tylko jeden parametr. Drugi musiałby być wtedy narzucony odgórnie w specjalizacji.

Kompilator wie jednak, że nie jest to taki zwyczajny szablon podstawowy. Dalej bowiem określamy dokładnie, o jakie przypadki użycia `TArray` nam chodzi. Są to więc te sytuacje, gdy klasa parametryzowana nazwą `TYP` (`TArray<TYP>`) sama staje się parametrem szablonu `TArray`, tworząc swego rodzaju zagnieżdżenie (tablicę tablic) - `TArray<TArray<TYP> >`. O tym świadczy pierwsza linijka naszej definicji, czyli:

```
template <typename TYP>    class TArray<TArray<TYP> >
```

Sam blok klasy wynika bezpośrednio z tego, że programujemy tablicę dwuwymiarową zamiast jednowymiarowej. Mamy więc dwa pola określające jej rozmiar - liczbę wierszy i ilość kolumn. Wymiary te podajemy w nowym, dwuparametrowym konstruktorze:

```
explicit TArray(unsigned uRozmiarX = DOMYSLNY_ROZMIAR,
                unsigned uRozmiarY = DOMYSLNY_ROZMIAR)
   : m_uRozmiarX(uRozmiarX), m_uRozmiar(uRozmiarY),
     m_pTablica(new TYP [uRozmiarX * uRozmiarY])        { }
```

Ten zaś dokonuje alokacji pojedynczego bloku pamięci na całą tablicę - a o to nam przecież chodziło. Wielkość tego bloku jest rzecz jasna na tyle duża, aby pomieścić wszystkie elementy - równa się ona iloczynowi wymiarów tablicy (bo np. tablica 4×7 ma w sumie 28 elementów, itp.).
Niestety, fakt iż jest to tablica dwuwymiarowa, uniemożliwia przeciążenie w prosty sposób operatora `[]` celem uzyskania dostępu do poszczególnych elementów tablicy. Zamiast tego stosujemy więc inny rodzaj nawiasów - okrągłe. Te bowiem pozwalają na podanie dowolnej liczby argumentów (indeksów); my potrzebujemy naturalnie dwóch:

```
TYP& operator()(unsigned uX, unsigned uY)
   { return m_pTablica[uY * m_uRozmiarX + uX]; }
```

Używamy ich potem, aby zwrócić element o żądanych indeksach. Wewnętrzna `m_pTablica` jest aczkolwiek ciągła i jednowymiarowa (bo ma zajmować pojedynczy blok pamięci), dlatego konieczne jest przeliczenie indeksów. Zajmuje się tym formułka `uY * m_uRozmiar + uX`, sprawiając jednocześnie, że elementy tablicy są układane w pamięci wierszami. „Przypadkowo” zgadza się to ze sposobem, jaki stosuje kompilator języka C++.

Na koniec popatrzmy jeszcze na sposób użycia tej (częściowo) specjalizowanej wersji szablonu `TArray`. Oto przykład kodu, który z niej korzysta:

```
TArray<TArray<double> > aMacierz4x4(4, 4);

// dostęp do elementów tablicy
for (unsigned i = 0; i < aMacierz4x4.RozmiarX(); ++i)
   for (unsigned j = 0; j < aMacierz4x4.RozmiarY(); ++j)
         aMacierz4x4(i, j) = i + j;
```

Tak więc dzięki specjalizacji częściowej klasa `TArray<TArray<double> >` i inne tego rodzaju mogą działać poprawnie, co nie bylo możliwe, gdy obecna była jedynie podstawowa wersja szablonu `TArray`.

Domyślne parametry szablonu klasy

Szablon klasy ma swoją listę parametrów, z których każdy może mieć swoją „wartość” domyślną. Działa to w analogiczny sposób, jak argumenty domyślne wywołań funkcji. Popatrzmy więc na tę technikę.

Typowy typ

Zanim jednak popatrzymy na samą technikę, popatrzmy na taki oto szablon:

```
// para
template <typename TYP1, typename TYP2> struct TPair
{
   // elementy pary
   TYP1 Pierwszy;
   TYP2 Drugi;

   //-------------------------------------------------------------------

   // konstruktor
```

```
   TPair(const TYP1& e1, const TYP2& e2) : Pierwszy(e1), Drugi(e2) { }
};
```

Reprezentuje on parę wartości różnych typów. Taka struktura może się wydawać lekko dziwaczna, ale zapewniam, że znajduje ona swoje zastosowania w róznych nieprzewidzianych momentach :) Zresztą nie o zastosowania tutaj chodzi, lecz o parametey szablonu.
A mamy tutaj dwa takie parametry: typy obu obiektów. Użycie naszej klasy wyglądać więc może chociażby tak:

```
TPair<int, int>            Dzielnik(42, 84);
TPair<std::string, int>    Slownie("dwanaście", 12);
TPair<float, int>          Polowa(2.5f, 5);
```

Przypuśćmy teraz, że w naszym programie często zdarza się nam, iż jeden z obiektów w parze należy do jakiegoś znanego z góry typu. W kodzie powyżej na przykład każda z tych par ma jeden element typu `int`.
Chcąc zaoszczędzić sobie konieczności pisania tego podczas deklarowania zmiennych, możemy uczynić int argumentem domyślnym:

```
template <typename TYP1, typename TYP2 = int> struct TPair
{
   // ...
};
```

Pisząc w ten sposób sprawiamy, że w razie niepodania „wartości” dla drugiego parametru szablonu, ma on oznaczać typ `int`:

```
TPair<CFoo>    Wielkosc(CFoo(), sizeof(CFoo));     // TPair<CFoo, int>
TPair<double>  Pierwiastek(sqrt(2), 2);            // TPair<double, int>
TPair<int>     DwaRaz(12, 6);                      // TPair<int, int>
```

Określając parametr domyślny pamiętajmy jednak, że:

Parametr szablonu może mieć **wartość domyślną** tylko wtedy, gdy znajduje się na **końcu listy** lub gdy wszystkie **parametry za nim też** mają wartość domyślną.

Niepoprawny jest zatem szablon:

```
template <typename TYP1 = int, typename TYP2> struct TPair;    // ŹLE!
```

Nic aczkolwiek nie stoi na przeszkodzie, aby podać wartości domyślne dla wszystkich parametrów:

```
template <typename TYP1 = std::string, typename TYP2 = int>
   struct TPair;                                                // OK
```

Używając takiego szablonu, nie musimy już podawać żadnych typów, aczkolwiek należy zachować nawiasy kątowe:

```
TPair<>  Opcja("Ilość plików", 200);   // TPair<std::string, int>
```

Obecnie domyślne argumenty można podawać wyłącznie dla szablonów klas. Jest to jednak pozostałość po wczesnych wersjach C++, niemająca żadnego uzasadnienia, więc jest całkiem prawdopodobne, że ograniczenie to zostanie wkrótce usunięte ze Standardu. Co więcej, sporo kompilatorów już teraz pozwala na podawanie domyślnych argumentów szablonów funkcji.

Skorzystanie z poprzedniego parametru

Dobierając parametr domyślny szablonu, możemy też skorzystać z poprzedniego. Oto przykład dla naszej pary:

```
template <typename TYP1, typename TYP2 = TYP1> struct TPair;
```

Przy takim postawieniu sprawy i podaniu jednego parametru szablonu będziemy mieli pary identycznych obiektów:

```
TPair<int>           DwaDo(8, 256);
TPair<std::string>   Tlumaczenie("tablica", "array");
TPair<double>        DwieWazneStale(3.14, 2.71);
```

Można jeszcze zauważyć, że identyczny efekt osiągnęlibyśmy przy pomocy częściowej specjalizacji szablonu `TPair` dla tych samych argumentów:

```
template <typename TYP> struct TPair<TYP, TYP>
{
   // elementy pary
   TYP Pierwszy;
   TYP Drugi;

   //------------------------------------------------------------------------

   // konstruktor
   TPair(const TYP& e1, const TYP& e2) : Pierwszy(e1), Drugi(e2) { }
};
```

Domyślne argumenty mają jednak tę oczywistą zaletę, że nie zmuszają do praktycznego dublowania definicji klasy (tak jak powyżej). W tym konkretnym przypadku są one znacznie lepszym wyborem. Jeżeli jednak postać szablonu dla pewnej klasy parametrów ma się znacząco różnić, wówczas dosłownie napisana specjalizacja jest najczęściej konieczna.

***

Na tym kończymy prezentację szablonów funkcji oraz klas. To aczkolwiek nie jest jeszcze koniec naszych zmagań z szablonami w ogóle. Jest bowiem jeszcze kilka rzeczy ogólniejszych, o których należy koniecznie wspomnieć. Przejdźmy więc do kolejnego podrozdziału na temat szablonów.

Więcej informacji

Po zasadniczym wprowadzeniu w tematykę szablonów zajmiemy się nieco szczegółowiej kilkoma ich aspektami. Najpierw więc przestudiujemy parametry szablonów, potem zaś zwrócimy uwagę na pewne problemy, jakie moga wyniknąć podczas stosowania tego elementu języka. Najwięcej uwagi poświęcimy tutaj sprawie organizacji kodu szablonów w plikach nagłówkowych i modułach, gdyż jest to jedna z kluczowych kwestii.

Zatem poznajmy szablony trochę bliżej.

Parametry szablonów

Dowiedziałeś się na samym początku, że każdy szablon rozpoczyna się od obowiązkowej frazy w postaci:

```
[export] template <parametry>
```

O nieobowiązkowym słowie kluczowym `export` powiemy w następnej sekcji, w paragrafie omawiającym tzw. model separacji.

Nazywamy ją **klauzulą parametryzacji** (ang. *parametrization clause*). Pełni ona w kodzie dwojaką funkcję:

- informuje ona kompilator, że następujący dalej kod jest **szablonem**. Dzięki temu kompilator wie, że nie powinien dlań przeprowadzać normalnej kompilacji, lecz potraktować w sposób specjalny - czyli poddać konkretyzacji
- klauzula zawiera też deklaracje *parametrów* szablonu, które są w nim używane

Właśnie tymi deklaracjami oraz rodzajami i użyciem parametrów szablonu zajmiemy się obecnie. Na początek warto więc wiedzieć, że parametry szablonów dzielimy na trzy rodzaje:

- parametry będące typami
- parametry będące stałymi znanymi w czasie kompilacji (tzw. parametry pozatypowe)
- szablony parametrów

Dotychczas w naszych szablonach niepodzielnie królowały parametry będące typami. Nadal bowiem są to najczęściej wykorzystywane parametry szablonów; dotąd mówi się nawet, że szablony i kod niezależny od typu danych to jedno i to samo.
My jednak nie możemy pozwolić sobie na ignorację w zakresie ich parametrów. Dlatego też teraz omówimy dokładnie wszystkie rodzaje parametrów szablonów.

Typy

Parametry szablonów będące typami stanowią największą siłę szablonów, przyczynę ich powstania, niespotykanej popularności i przydatności. Nic więc dziwnego, że pierwsze poznane przez nas przykłady szablonów korzystały właśnie z parametryzowania typów. Nabrałeś więc całkiem sporej wprawy w ich stosowaniu, a teraz poznasz kryjącą się za tym fasadę teorii ;)

Przypominamy banalny przykład

W tym celu przywołajmy pierwszy przykład szablonu, z jakim mieliśmy do czynienia, czyli szablon funkcji `max()`:

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

Ma on jeden parametr, będący typem; parametr ten nosi nazwę `T`. Jest to zwyczajowa już nazwa dla takich parametrów szablonu, którą można spotkać niezwykle często. Zgodnie z tą konwencją, nazwę `T` nadaje się parametrowi będącemu typem, jeśli jest on jednocześnie jedynym parametrem szablonu i w związku z tym pełni jakąś szczególną rolę. Może to być np. typ elementów tablicy czy, tak jak tutaj, typ parametrów funkcji i zwracanej przez nią wartości.

Nazwa `T` jest tu więc symbolem zastępczym dla właściwego typu porównywanych wartości. Jeżeli pojęcie to sprawia ci trudność, wyobraź sobie, że działa ono podobnie jak alias `typedef`. Można więc przyjąć, że kompilator, stosując funkcję w konkretnym przypadku, definiuje `T` jako alias na właściwy typ. Przykładowo, specjalizację `max<int>` można traktować jako kod:

```
typedef int T;
T max (T a, T b)      { return (a > b ? a : b); }
```

Naturalnie, w rzeczywistości generowana jest po prostu funkcja:

```
int max<int>(int a, int b);
```

Niemniej powyższy sposób może ci z początku pomóc, jeśli dotąd nie rozumiałeś idei parametru szablonu będącego typem.

class zamiast typename

Parametr szablonu będący typem oznaczaliśmy dotąd za pomocą słowa kluczowego `typename`. Okazuje sie, że można to także robić poprzez słówko `class`:

```
template <class T> T max(T a, T b);
```

Nie oznacza to bynajmniej, że podany parametr szablonu może być wyłącznie klasą zdefiniowaną przez użytkownika[129]. Przeciwnie, otóż:

Slowa `class` i `typename` w są **synonimami** w deklaracjach **parametrów szablonu** będących **typami**.

Po co zatem istnieją dwa takie słowa?… Jest to spowodowane tym, iż pierwotnie jedynym sposobem na deklarowanie parametrów szablonu było `class`. `typename` wprowadzono do języka później, i to w całkiem innym przeznaczeniu (o którym też sobie powiemy). Przy okazji aczkolwiek pozwolono na użycie tego nowego slowa w deklaracjach parametrów szablonów, jako że znacznie lepiej pasuje tutaj niż `class`. Dlatego też mamy ostatecznie dwa sposoby na zrobienie tego samego.

Można z tego wyciągnąć pewną korzyść. Wprawdzie dla kompilatora nie ma znaczenia, czy do deklaracji parametrów używamy `class` czy `typename`, lecz nasz wybór może mieć przecież znaczenie dla nas. Logiczne jest mianowicie używanie `class` wyłącznie tam, gdzie faktycznie spodziewamy się, że przekazanym typem będzie klasa (bo np. wywołujemy jej metody). W pozostałych przypadkach, gdy typ może być absolutunie dowolny (jak choćby w uprzednich szablonach `max()` czy `TArray`), rozsądne jest stosowanie `typename`.
Naturalnie, to tylko sugestia, bo jak mówiłem już, kompilatorowi jest w tej kwestii wszystko jedno.

Stałe

Cenną właściwością szablonów jest możliwość użycia w nich innego rodzaju parametrów niż tylko typy. Są to tak zwane **parametry pozatypowe** (ang. *non-type parameters*), a dokładniej mówiąc: **stałe**.

Użycie parametrów pozatypowych

Ich wykorzystanie najlepiej będzie zobaczyć na paru rozsądnych przykładach.

[129] Czyli typem zdefiniowanym poprzez `struct`, `union` lub `class`.

Przykład szablonu klasy

W poprzednim paragrafie zdefiniowaliśmy sobie szablon klasy `TArray`. Służył on jako jednowymiarowa tablica dynamiczna, której rozmiar podawaliśmy przy tworzeniu i ewentualnie zmienialiśmy w trakcie korzystania z obiektu.
Można sobie jeszcze wyobrazić podobny szablon dla tablicy statycznej, której rozmiar jest znany podczas kompilacji. Oto propozycja szablonu `TStaticArray`:

```
template <typename T, unsigned N> class TStaticArray
{
   private:
         // tablica
         T m_aTablica[N];

   public:
         // rozmiar tablicy jako stała
         static const unsigned ROZMIAR = N;

         //-------------------------------------------------------------

         // operator indeksowania
         T& operator[](unsigned uIndeks)
               { return m_aTablica[uIndeks]; }

         // (itp.)
};
```

Jak słusznie zauważyłeś, szablon ten zawiera dwa parametry. Pierwszy z nich to typ elementów tablicy, deklarowany w znany sposób poprzez `typename`. Natomiast drugi parametr jest właśnie przedmiotem naszego zainteresowania. Stosujemy w nim typ `unsigned`, wobec czego będzie on stałą tego właśnie typu.
Popatrzmy najlepiej na sposób użycia tego szablonu:

```
TStaticArray<int, 10>      a10Intow;   // 10-elementowa tablica typu int
TStaticArray<float, 20>    a20Floatow; // 20 liczb typu float
TStaticArray<
   TStaticArray<double, 5>,
            8>             a8x5Double;  // tablica 8×5 liczb typu double
```

Podobnie jak w przypadku parametrów będących typami możesz sobie wyobrazić, że kompilator konkretyzuje szablon, definiując wartość `N` jako stałą. Klasa `TStaticArray<float, 10>` odpowiada więc mniej więcej zapisowi w takiej postaci:

```
typedef float T;
const unsigned N = 10;

class TStaticArray
{
   private:
         T m_aTablica[N];

   // ...
};
```

Wynika z niego przede wszystkim to, iż:

Parametry pozatypowe szablonów są traktowane wewnątrz nich jako **stałe**.

Oznacza to przede wszystkim, że muszą być one „wywoływane" z wartościami, które są **obliczalne podczas kompilacji**. Wszystkie pokazane powyżej konkretyzacje są więc poprawne, bo `10`, `20`, `5` i `8` są rzecz jasna stałymi dosłownymi, a więc znanymi w czasie kompilacji. Nie byłoby natomiast dozwolone użycie szablonu jako `TStaticArray<`*`typ`*`, `*`zmienna`*`>`, gdzie *`zmienna`* niezadeklarowana została z przydomkiem `const`.

Przykład szablonu funkcji

Gdy mamy już zdefiniowany nowy szablon tablicy, możemy spróbować stworzyć dla niego odpowiadającą wersję funkcji `Szukaj()`. Naturalnie, będzie to również szablon:

```
template <typename T, unsigned N>
   int Szukaj(const TStaticArray<T, N>& aTablica, T Szukany)
{
   // przegląd tablicy
   for (unsigned i = 0; i < N; ++i)
         if (aTablica[i] == Szukany)
               return i;

   // -1, gdy nie znaleziono
   return -1;
}
```

Widać tutaj, że parametr pozatypowy może być z równym powodzeniem użyty zarówno w nagłówku funkcji (typ `const TStaticArray<T, N>&`), jak i w jej wnętrzu (warunek zakończenia pętli `for`).

Dwie wartości, dwa różne typy

Wyobraźmy sobie, że mamy dwie tablice tego samego typu, ale o różnych rozmiarach:

```
TStaticArray<int, 20> a20Intow;
TStaticArray<int, 10> a10Intow;
```

Spróbujmy teraz przypisać tę mniejszą do większej, w ten oto sposób:

```
a20Intow = a10Intow;              // hmm...
```

Teoretycznie powinno być to jak najbardziej możliwe. Pierwszym 10 elementów tablicy `a20Intow` mogłoby być przecież zastąpione zawartością zmiennej `a10Intow`. Nie ma zatem przeciwwskazań.
Niestety, kompilator odrzuci taki kod, mówiąc, iż nie znalazł żadnego pasującego operatora przypisania ani niejawnej konwersji. I będzie to szczera prawda! Musimy bowiem pamiętać, że:

Szablony klas konkretyzowane **innym zestawem parametrów** są **zupełnie odmiennymi typami**.

Nic więc dziwnego, że `TStaticArray<int, 10>` i `TStaticArray<int, 20>` są traktowane jako odrębne klasy, niezwiązane ze sobą (obie te nazwy, wraz z zawartością nawiasów kątowych, są bowiem nazwami typów, o czym przypominam po raz któryś). W takim wypadku domyślnie generowany operator przypisania zawodzi. Warto więc pamiętać o powyższej zasadzie.

No ale skoro mamy już taki problem, to przydałoby się go rozwiązać. Odpowiednim wyjściem jest własny operator przypisania zdefiniowany jako **szablon składowej**:

```
template <typename T, unsigned N> class TStaticArray
{
```

```
    // ...

    public:
          // operator przypisania jednej tablicy do drugiej
          template <typename T2, unsigned N2>
                TStaticArray&
                operator=(const TStaticArray<T2, N2>& aTablica)
                {
                      // kontrola przypisania zwrotnego
                      if (&aTablica != this)
                      {
                            // sprawdzenie rozmiarów
                            if (N2 > N)
                                  throw "Za duza tablica";

                            // przepisanie tablicy
                            for (unsigned i = 0; i < N2; ++i)
                                  (*this)[i] = aTablica[i];
                      }

                      return *this;
                }
};
```

Może i wygląda on nieco makabrycznie, ale w gruncie rzeczy działa na identycznej zasadzie jak każdy rozsądny operator przypisania. Zauważmy, że parametryzacji podlega w nim nie tylko rozmiar źródłowej tablicy (`N2`), ale też typ jej elementów (`T2`). To, czy przypisanie faktycznie jest możliwe, zależy od tego, czy powiedzie się kompilacja instrukcji:

```
(*this)[i] = aTablica[i];
```

A tak będzie oczywiście tylko wtedy, gdy istnieje niejawna konwersja z typu `T2` do `T`.

Ograniczenia dla parametrów pozatypowych

Pozatypowe parametry szablonów w przeciwieństwie do parametrów funkcji nie mogą być „wywoływane” z dowolnymi wartościami. Typami tychże parametrów mogą być bowiem tylko:

- typy liczbowe, czyli `int` i jego pochodne (`signed` lub `unsigned`)
- typy wyliczeniowe (definiowane poprzed `enum`)
- wskaźniki do obiektów i funkcji globalnych
- wskaźniki do składowych klas

Lista ta jest dość krótka i może się wydawać nazbyt restrykcyjna. Tak jednak nie jest. Głównie ze względu na sposób działania szablonów ich parametry pozatypowe są ograniczone tylko do takich rodzajów.

Przyjrzyjmy się jeszcze kilku szczególnym przypadkom tych ograniczeń.

Wskaźniki jako parametry szablonu

Nie ma żadnych przeciwskazań, aby deklarować szablony z parametrami będącymi typami wskaźnikowymi. Wygląda to na przykład tak:

```
template <int* P> class TClass
{
    // ...
};
```

Gorzej wygląda sprawa z użyciem takiego szablonu. Otóż nie możemy przekazać mu wskaźnika ani na obiekt chwilowy, ani na obiekt lokalny, ani nawet na obiekt o zasięgu modułowym. Nie jest więc poprawny np. taki kod:

```
int nZmienna;
TClass<&nZmienna> Obiekt;        // ŹLE! Wskaźnik na obiekt lokalny
```

Wyjaśnienie jest tu proste. Wszystkie takie obiekty mają po prostu zbyt mały zakres, który nie pokrywa się z widocznością konkretyzacji szablonu. Aby tak było, obiekt, na który wskaźnik podajemy, musiałby być **globalny** (łączony zewnętrznie):

```
extern int g_nZmienna = 42;
// ...
TClass<&g_Zmienna> Cos;          // OK
```

Z identycznych powodów nie można do szablonów przekazywać łańcuchów znaków:

```
template <const char[] S> class TStringer    { /* ... */ };
TStringer<"Hmm..."> Napisowiec;              // NIE!
```

Łańcuch `"Hmm..."` jest tu bowiem obiektem chwilowym, zatem szybko przestałby istnieć. Typ `TStringer<"Hmm...">` musiałby natomiast egzystować i być potencjalnie dostępnym w całym programie. To oczywiście wzajemnie się wyklucza.

Inne restrykcje

Oprócz powyższych obostrzeń są jeszcze dwa inne.

Po pierwsze, w charakterze parametrów szablonu **nie można używać obiektów własnych klas**. Poniższe szablony są więc niepoprawne:

```
template <CFoo F> class TMetaFoo                  { /* ... */ };
template <std::string S> class TStringTemplate    { /* ... */ };
```

Poza tym, w charakterze parametrów pozatypowych teoretrycznie niedozwolone są wartości zmiennoprzecinkowe:

```
template <float F> class TCalc   { /* ... */ };
```

Mówię ‘teoretycznie’, gdyż wiele kompilatorów pozwala na ich użycie. Nie ma bowiem ku temu żadnych technicznych przeciwwskazań (w odróżnieniu od pozostałych ograniczeń parametrów pozatypowych). Niemniej, w Standardzie C++ nadal zakorzenione jest to przestarzałe ustalenie. Zapewne jednak tylko kwestią czasu jest jego usunięcie.

Szablony parametrów

Ostatnim rodzajem parametrów są tzw. **szablony parametrów szablonów** (ang. *template templates' parameters*). Pod tą dziwnie brzmiącą nazwą kryje się możliwość przekazania jako parametru nie konkretnego typu, ale uprzednio zdefiniowanego szablonu. Ponieważ zapewnie nie brzmi to zbyt jasno, najrozsądniej będzie dojść do sedna sprawy przy pomocy odpowiedniego przykładu.

Idąc za potrzebą

A więc… Swego czasu stworzyliśmy sobie szablon ogólnej klasy `TArray`. Okazuje się jednak, że niekiedy może być on niewystarczający. Chociaż dobrze nadaje się do samej czynności przechowywania wartości, nie pomyśleliśmy o żadnych mechanizmach operowania na tychże wartościach.

Z drugiej strony, nie ma sensu zmiany dobrze działającego kodu w coś, co nie zawsze będzie nam przyadtne. Takie czynności jak dodawnie, odejmowanie czy mnożenie tablic mają bowiem sens tylko w przypadku wektorów liczb. Lepiej więc zdefiniować sobie nowy szablon do takich celów:

```
template <typename T> class TNumericArray
{
   private:
         // wewnętrzna tablica
         TArray<T> m_aTablica;

   public:
         // ...

         // jakieś operatory...
         // (np. indeksowania)

         TNumericArray operator+(const TNumericArray& aTablica)
         {
               TNumericArray Wynik(*this);

               for (unsigned i = 0; i < Wynik.Rozmiar(); ++i)
                     Wynik[i] += aTablica[i];

               return Wynik;
         }

         // (itp.)
};
```

W sumie nic specjalnego nie możemy powiedzieć o tym szablonie klasy `TNumericArray`. Jak się pewnie domyślasz, to się za chwilę zmieni :)

Dodatkowy parametr: typ wewnętrznej tablicy

Może się okazać, że w naszym programie zmuszeni jesteśmy do operowania zarówno wielkimi tablicami, jak i mniejszymi. W wypadku tych drugim wewnętrzny szablon `TArray` służący do ich przechowywania pewnie zda egzamin, ale gdy liczba elementów rośnie, mogą być konieczne bardziej wyrafinowane techniki zarządzania pamięcią.

Aby sprostać temu wymaganiu, rozsądnie byłoby umożliwić wybór typu wewnętrznej tablicy dla szablonu `TNumericArray`:

```
template <typename T, typename TAB = TArray<T> >
   class TNumericArray
   {
         private:
               TAB m_aTablica;

         // ...
   };
```

Domyślnie byłby to nadal szablon `TArray`, niemniej przy takim szablonie `TNumericArray` możnaby w miarę łatwo deklarować zarówno duże, jak i małe tablice:

```
TNumericArray<int>                                  aMalaTablica(50);
TNumericArray<float, TOptimizedArray<float> >       aDuzaTablica(1000);
TNumericArray<double, TSuperFastArray<double> >     aGigaTablica(250000);
```

W tym przykładzie zakładamy oczywiście, że `TOptimizedArray` i `TSuperFastArray` są jakimiś uprzednio zdefiniowanymi szablonami tablic efektywniejszych od `TArray`. W uzasadnionych przypadkach dużej liczby elementów ich użycie jest więc pewnie pożądane, co też czynimy.

Drobna niedogodność

Powyższe rozwiązanie ma jednak pewien drobny mankament składniowy. Nietrudno mianowicie zauważyć, że dwa razy piszemy w nim typ elementów tablic - `float` i `double`. Pierwszy raz jest on podawany szablonowi `TNumericArray`, a drugi raz - szablonowi wewnętrznej tablicy.

W sumie powoduje to zbytnią rozwlekłość nazwy całego typu `TNumericArray<...>`, a na dodatek ujawnia osławiony problem nawiasów ostrych. Wydaje się przy tym, że informację o typie podajemy o jeden raz za dużo; w końcu zamiast deklaracji:

```
TNumericArray<float, TOptimizedArray<float> >      aDuzaTablica(1000);
TNumericArray<double, TSuperFastArray<double> >    aGigaTablica(250000);
```

równie dobrze mogłoby się sprawdzać coś w tym rodzaju:

```
TNumericArray<float, TOptimizedArray>        aDuzaTablica(1000);
TNumericArray<double, TSuperFastArray>       aGigaTablica(250000);
```

Problem jednak w tym, że parametry szablonu `TNumericArray` - `TOptimizedArray` i `TSuperFastArray` nie są zwykłymi typami danych (klasami), więc nie pasują do deklaracji `typename TAB`. One same są szablonami klas, zdefiniowanymi zapewne kodem podobnym do tego:

```
template <typename T> class TOptimizedArray        { /* ... */ };
template <typename T> class TSuperFastArray        { /* ... */ };
```

Można więc powiedzieć, że występuje to swoista „niezgodność typów” między pojęciami ‘typ’ i ‘szablon klasy’. Czy zatem nasz pomysł skrócenia sobie zapisu trzeba odrzucić?…

Deklarowanie szablonowych parametrów szablonów

Bynajmniej. Między innymi na takie okazje całkiem niedawno język C++ wyposażono w możliwość deklarowania szczególnego rodzaju parametrów szablonu. Te specjalne parametry charakteryzują się tym, że są **nazwami zastępczymi dla szablonów klas**. Jako takie wymagają więc podania nie konkretnego typu danych, lecz jego uogólnienia - szablonu klasy.

> Dobrą, nieszablonową analogią dla tej niecodziennej konstrukcji jest sytuacja, gdy funkcja przyjmuje jako parametr inną funkcję poprzez wskaźnik. W mniej więcej zbliżony koncepcyjnie sposób działają szablonowe parametry szablonów.

Oto jak deklaruje się i używa tych specjalnych parametrów:

```
template <typename T, template <typename> class TAB = TArray>
   class TNumericArray
   {
         private:
               TAB<T> m_aTablica;

         // ...
   };
```

Posługujemy się tu dwa razy słowem kluczowym `template`. Pierwsze użycie jest już powszechnie znane; drugie występuje w liście parametrów szablonu `TNumericArray` i o nie nam teraz chodzi. Przy jego pomocy deklarujemy bowiem szablon parametru. Składnia:

```
template <typename> class TAB
```

oznacza tutaj, że do parametru `TAB` pasują wszystkie szablony klas (`template <...> class`), które mają dokładnie jeden parametr będący typem (`typename`[130]). W przypadku niepodania żadnego szablonu, zostanie wykorzystany domyślny - `TArray`.

Teraz, gdy nazwa `TAB` jest już nie klasą, lecz jej szablonem, używamy jej tak jak szablonu. Deklaracja pola wewnętrznej tablicy wygląda więc następująco:

```
TAB<T> m_aTablica;
```

Jako parametr dla `TAB` podajemy `T`, czyli pierwszy parametr naszego szablonu `TNumericArray`. W sumie jednak możnaby użyć dowolnego typu (także podanego dosłownie, np. `int`), bo `TAB` zachowuje się tutaj tak samo, jak pełnoprawny szablon klasy.

Naturalnie, teraz poprawne stają się propozycje deklaracji zmiennych z poprzedniego akapitu:

```
TNumericArray<float, TOptimizedArray>       aDuzaTablica(1000);
TNumericArray<double, TSuperFastArray>      aGigaTablica(250000);
```

Na ile przydatne są szablony parametrów szablonów (zwane też czasem **metaszablonami** - ang. *metatemplates*) musisz się właściwie przekonać sam. Jest to jedna z tych cech języka, dla których trudno od razu znaleźć jakieś oszałamiające zastosowanie, ale jednocześnie może okazać się przydatna w pewnych szczególnych sytuacjach.

Problemy z szablonami

Szablony są rzeczywiście jednym z największych osiągnięć języka C++. Jednak, jak to jest z większością zaawansowanych technik, ich stosowanie może za soba pociągać pewne problemy. Nie, nie chodzi mi tu wcale o to, że szablony są trudne do nauczenia, choć pewnie masz takie nieodparte wrażenie ;) Chciałbym raczej porozmawiać o kilku pułapkach czyhających na programistę (szczególnie początkującego), który zechce używać szablonów. Dzięki temu być może łatwiej unikniesz mniej lub bardziej poważnych problemów z tymi konstrukcjami językowymi.

Zobaczmy więc, co może stanąć nam na drodze…

Ułatwienia dla kompilatora

Śledząc opis czynności, jakie wykonuje kompilator w związku z szablonami, można zauważyć, że zmuszony jest do iście ekwilibrystycznych wygibasów. To zrozumiałe, jeśli przypomnimy sobie, że kontrola typów jest w C++ jednym z filarów programowania, zaś szablony częściowo służą właśnie do jej obejścia.

---

[130] Nie podajemy nazwy parametru szablonu `TAB`, bo nie ma takiej potrzeby. Nazwa ta nie jest nam po prostu do niczego potrzebna.

Na kompilatorze spoczywa mnóstwo trudnych zdań, jeśli chodzi o kod wykorzystujący szablony. Dlatego też niekiedy potrzebuje on wsparcia ze strony programisty, które ułatwiłoby mu intepretację kodu źródłowego. O takich właśnie „ułatwieniach dla kompilatora" traktuje niniejszy paragraf.

Nawiasy ostre

Niejednego nowicjusza w używaniu szablonów zjadł smok o nazwie „problem nawiasów ostrych". Nietrudno przecież wyprodukować taki kod, wierząc w jego poprawność:

```
typedef TArray<TArray<double>> MATRIX;       // ojć!
```

Ta wiara zostaje jednak dość szybko podkopana. Coraz częściej wprawdzie zdarza się, że kompilator poprawnie rozpoznaje znaki `>>` jako zamykające nawiasy ostre. Niemniej, nadal może to jeszcze powodować błąd lub co najmniej ostrzeżenie.

Poprawna wersja kodu, działająca w każdej sytuacji, to oczywiście:

```
typedef TArray<TArray<double> > MATRIX;            // OK
```

Dodatkowa spacja wygląda tu rzecz jasna bardzo nieładnie, ale póki co jest konieczna. Wcale niewykluczone jednak, że za jakiś czas także pierwsza wersja instrukcji `typedef` będzie musiała być uznana za poprawną.

Nieoczywisty przykład

Można słusznie sądzić, że w powyższym przykładzie rozpoznanie sekwencji `>>` jako pary nawiasów zamykających (a nie operatora przesunięcia w prawo) nie jest zadaniem ponad siły kompilatora. Pamiętajmy aczkolwiek, że nie zawsze jest to takie oczywiste.

Spójrzmy choćby na taką deklarację:

```
TStaticArray<int, 16>>2>   aInty;      // chyba prosimy się o kłopoty...
```

Dla czytającego (i piszącego) kod człowieka jest całkiem wyraźne widoczne, że drugim parametrem szablonu `TStaticArray` jest tu `16>>2` (czyli `64`). Kompilator uczulony na problem nawiasów ostrych zinterpretuje aczkolwiek poniższą linijkę jako:

```
TStaticArray<int, 16> >2>  aInty;      // ojej!
```

W sumie więc nie bardzo wiadomo, co jest lepsze. Właściwie jednak wyrażenia podobne do powyższego są raczej rzadkie i prawdę mówiąc nie powinny być w ogóle stosowane. Gdyby zachodziła taka konieczność, najlepiej posłużyć się pomocniczymi nawiasami okrągłymi:

```
TStaticArray<int, (16>>2)> aInty;      // OK
```

Wniosek z tego jest jeden: kiedy chodzi o nawiasy ostre i szablony, lepiej być wyrozumiałym dla kompilatora i w odpowiednich miejscach pomóc mu w zrozumieniu, o co nam tak naprawdę chodzi.

Ciekawostka: dlaczego tak się dzieje

Być może zastanawiasz się, dlaczego kompilator ma w ogóle problemy z poprawnym rozpoznawianiem użycia nawiasów ostrych. Przecież nic podobnego nie dotyczy ani nawiasów okrągłych (wyrażenia, wywołania funkcji, itd.), ani nawiasów kwadratowych (indeksowanie tablicy), ani nawet nawiasów klamrowych (bloki kodu). Skąd więc problemy wynikają problemy objawiające się w szablonach?…

Przyczyną jest po części sposób, w jaki kompilatory C++ dokonują analizy kodu. Dokładne omówienie tego procesu jest skomplikowane i niepotrzebne, więc je sobie darujemy. Interesującą nas czynnością jest aczkolwiek jeden z pierwszych etapów przetwarzania - tak zwana **tokenizacja** (ang. *tokenization*).
Tokenizacja polega na tym, iż kompilator, analizując kod znak po znaku, wyróżnia w nim elementy leksykalne języka - **tokeny**. Do tokenów należą głównie identyfikatory (nazwy zmiennych, funkcji, typów, itp.) oraz operatory. Kompilator wpierw dokonuje ich analizy (parsowania) i tworzy listę takich tokenów.

Sęk polega na tym, że C++ jest **językiem kontekstowym** (ang. *context-sensitive language*). Oznacza to, że identyczne sekwencje znaków mogą w nim znaczyć zupełnie co innego w zależności od kontekstu. Przykładowo, fraza `a*b` może być zarówno mnożeniem zmiennej `a` przez zmienną `b`, jak też deklaracją wskaźnika na typ `a` o nazwie `b`. Wszystko zależy od znaczenia nazw `a` i `b`.
W przypadku operatorów mamy natomiast jeszcze jedną zasadę, zwaną **zasadą maksymalnego dopasowania** (ang. *maximum match rule*). Mówi ona, że należy zawsze próbować ująć jak najwięcej znaków w jeden token.

Te dwie cechy C++ (kontekstowość i maksymalne dopasowanie) dają w efekcie zaprezentowane wcześniej problemy z nawiasami ostrymi. Problem jest bowiem w tym, iż zależnie od kontekstu i sąsiedztwa znaki `<` i `>` mogą być interpretowane jako:

- operatory większości i mniejszości
- operatory przesunięcia bitowego
- nawiasy ostre

Nie ma to większego znaczenia, jeśli nie występują one w bliskim sąsiedztwie. W przeciwnym razie zaczynają się poważne kłopoty - jak choćby tutaj:

```
TSomething<32>>4 > FOO>    CosTam;     // no i?...
```

Najlogiczniej więc byłoby unikać takich ryzykownych konstrukcji lub opatrywać je dodatkowymi znakami (spacjami, nawiasami okrągłymi), które umożliwią kompilatorowi jednoznaczną interpretację.

Nazwy zależne

Problem nawiasów ostrych jest w zasadzie kwestią wyłącznie składniową, spowodowaną faktem wyboru takiego a nie innego rodzaju nawiasów do współpracy z szablonami. Jednak jeśli nawet sprawy te zostałyby kiedyś rozwiązane (co jest mało prawdopodobne, zważywszy, że piątego rodzaju nawiasów jeszcze nie wymyślono :D), to i tak kod szablonów w pewnych sytuacjach będzie kłopotliwy dla kompilatora.

O co dokładnie chodzi?… Otóż trzeba wiedzieć, że szablony są tak naprawdę kompilowane dwukrotnie (albo raczej w dwóch etapach):

- najpierw są one analizowane pod kątem ewentualnych błędów składniowych i językowych w swej „czystej” (nieskonkretyzowanej) postaci. Na tym etapie kompilator nie ma informacji np. o typach danych, do których odnoszą symboliczne oznaczenia parametrów szablonów (`T`, `TYP`, itd.)
- później produkty konkretyzacji są sprawdzane pod kątem swej poprawności w całkiem normalny już sposób, zbliżony do analizy zwykłego kodu C++

Nie byłoby w tym nic niepokojącego gdyby nie fakt, że w pewnych sytuacjach kompilator może nie być wystarczająco kompetentny, by wykonać fazę pierwszą. Może się bowiem okazać, że do jej przeprowadzania wymagane są informacje, które można uzyskać dopiero **po konketyzacji**, czyli w fazie drugiej.

Pewnie w tej chwili nie bardzo możesz sobie wyobrazić, o jakie informacje może tutaj chodzić. Powiem więc, że sprawa dotyczy głównie właściwej interpretacji tzw. **nazw zależnych**.

Nazwa zależna (ang. *dependent name*) to każda **nazwa użyta wewnątrz szablonu**, **powiązana** w jakiś sposób **z jego parametrami**.

Fakt, że nazwy takie są powiązane z parametrami szablonu, sprawia, że ich znaczenie może być różne w zależności od parametrów tego szablonu. Te wszystkie engimatyczne stwierdzenia staną się bardziej jasne, gdy przyjrzymy się konkretnym przykładom problemów i sposobom na ich rozwiązanie.

Słowo kluczowe `typename`

Czas więc na kawałek szablonu :) Popatrzmy na taką problematyczną funkcję, która ma za zadanie wybrać największy spośród elementów tablicy:

```
template <class TAB> TAB::ELEMENT Najwiekszy(const TAB& aTablica)
{
   // zmienna na przechowanie wyniku
   TAB::ELEMENT Wynik = aTablica[0];

   // pętla szukająca
   for (unsigned i = 1; i < aTablica.Rozmiar(); ++i)
         if (aTablica[i] > Wynik)
               Wynik = aTablica[i];

   // zwrócenie wyniku
   return Wynik;
}
```

Można się zdziwić, czemu parametrem szablonu jest tu typ tablicy (czyli np. `TArray<int>`), a nie typ jej elementów (`int`). Dzięki temu funkcja jest jednak bardziej uniwersalna i niekoniecznie musiy współpracować wyłącznie z tablicami `TArray`. Przeciwnie, może działać dla każdej klasy tablic (a więc np. dla `TOptimizedArray` i `TSuperFastArray` z paragrafiu o metaszablonach), która ma:

- operator indeksowania
- metodę `Rozmiar()`
- alias `ELEMENT` na typ elementów tablicy

Niestety, ten ostatni punkt jest właśnie problemem. Ściślej mówiąc, to fraza `TAB::ELEMENT` stanowi kłopot - `ELEMENT` jest tu bowiem nazwą zależną. My jesteśmy tu święcie przekonani, że reprezentuje ona typ (`int` dla `TArray<int>` itd.), jednak kompilator nie może brać takich informacji znikąd. On faktycznie musi to **wiedzieć**, aby mógł uznać m.in. deklarację:

```
TAB::ELEMENT Wynik;
```

za poprawną. A skąd ma się tego dowiedzieć?… Nie ma ku temu żadnej możliwości na etapie analizy samego szablonu. Dopiero konkretyzacja, gdy `TAB` jest zastępowane prawdziwym typem danych, daje mu taką możliwość. Tyle że aby w ogóle mogło dojść do konkretyzacji, szablon musi najpierw przejść test poprawności. Mówiąc wprost: aby skontrolować bezbłędność szablonu kompilator musi najpierw… skontrolować bezbłędność szablonu :D Dochodzimy zatem do błędnego koła.

A wyjście z niego jest jedno. Musimy w jakiś sposób dać do zrozumienia kompilatorowi, że `TAB::ELEMENT` jest typem, a nie statycznym polem - bo taka jest właśnie druga

możliwa interpretacja tej konstrukcji. Czynimy to poprzedzając problematyczną frazę słówkiem `typename`:

```
typename TAB::ELEMENT Wynik;
```

Deklaracja nieco nam się rozwlekła, ale w przy korzystaniu z szablonów jest to już chyba regułą :) W każdym razie teraz nie będzie już problemów ze zmienną `Wynik`; do pełnej satysfakcji należy jeszcze podobny zabieg zastosować wobec typu zwracanego przez funkcję:

```
template <class TAB>
   typename TAB::ELEMENT Najwiekszy(const TAB& aTablica)
```

Podobnie należy postąpić z każdym wystąpieniem `TAB::ELEMENT` w tym szablonie. Powiem nawet więcej, formułując ogólną zasadę:

> Należy **poprzedzać** słowem `typename` każdą **nazwę zależną**, która ma być **interpretowana jako typ danych**.

Stosując się do niej, nie będziemy wprawiać w kompilatora w zakłopotanie i oszczędzimy sobie dziwnie wyglądających komunikatów o błędach.

Ciekawostka: konstrukcje `::template`, `.template` i `->template`

Podobny, choć znacznie raczej ujawniający się problem dotyczy szablonów zagnieżdżonych. Oto nieszczególnie sensowny przykład takiej sytuacji:

```
template <typename T> class TFoo
{
   public:
         // zagnieżdżony szablon klasy
         template <typename U> class TBar
         {
               public:
                     // zagnieżdżony szablon statycznej metody
                     template <typename V> static void Baz();
         }
};
```

Pytanie brzmi: jak wywołać metodę `Baz()`? No cóż, wyglądać to może tak:

```
template <typename T> void Funkcja()
{
   // wywołanie jako statycznej metody bez obiektu
   TFoo<T>::template TBar<T>::Baz();

   // utworzenie lokalnego obiektu i wywołanie metody
   typename TFoo<T>::template TBar<T> Bar;
   Bar.template Baz<T>();

   // utworzenie dynamicznego obiektu i wywołanie metody
   typename TFoo<T>::template TBar<T>* pBar;
   pBar = new typename TFoo<T>::template TBar<T>;
   pBar->template Baz<T>();
   delete pBar;
}
```

Wiem, że wygląda to jak skryżowanie trolla z goblinem, ale mówimy teraz o naprawdę specyficznym szczególiku, którego użycie jest bardzo rzadkie. Powyższy kod wyglałby pewnie przejrzyściej, gdyby usunąć z niego wyrazy `typename` i `template`:

```
// UWAGA: ten kod NIE JEST poprawny!

// wywołanie jako statycznej metody bez obiektu
TFoo<T>::TBar<T>::Baz();

// utworzenie lokalnego obiektu i wywołanie metody
TFoo<T>::TBar<T> Bar;
Bar.Baz<T>();

// utworzenie dynamicznego obiektu i wywołanie metody
TFoo<T>::TBar<T>* pBar;
pBar = new TFoo<T>::TBar<T>;
pBar->Baz<T>();
delete pBar;
```

Tym samym jednak pozbawiamy kompilator informacji potrzebnych do skompilowania szablonu. Rolę `typename` znamy, więc zajmijmy się dodatkowymi użyciami `template`.

Otóż tutaj `template` (a właściwie `::template`, `.template` i `->template`) służy do poinformowania, że następująca dalej **nazwa zależna jest szablonem**. Patrząc na definicję `TFoo` wiemy to oczywiście, jednak kompilator nie dowie się tego aż do chwili konkretyzacji. Dla niego nazwy `TBar` i `Baz` mogą być równie dobrze składowymi statycznymi, zaś następujące dalej znaki `<` i `>` - operatorami relacji. Musimy więc wyprowadzić go błędu.

Stosuj kontrukcje `::template`, `.template` i `->template` **zamiast samych operatorów** `::`, `.` i `->` w tych miejscach, gdzie **podana dalej nazwa zależna jest szablonem**.

Stosowalność tych konstrukcji jest więc ograniczona i zawęża się do przypadków zagnieżdżonych szablonów. W codziennej i nawet trochę bardziej niecodziennej praktyce programistycznej można się bez nich obejść, aczkolwiek warto o nich wiedzieć, by móc je zastosować w tych nielicznych sytuacjach ujawniającej się niewiedzy kompilatora.

Organizacja kodu szablonów

Wykorzystanie szablonów może nastręczać problemów natury logistycznej. Nie chodzi o samą czynność ich implementacji czy późniejszego wykorzystania, ale o, zdawałoby się: prozaiczną, sprawę następującą: jak rozmieścić kod szablonów w plikach z kodem programu?…

Sprawa nie jest wcale taka prosta, bo kod korzystający z szablonów różni się znacznie pod tym względem od zwykłego, „nieszablonowego” kodu. W sumie można powiedzieć, że szablony są czymś między normalnymi instrukcjami języka, a dyrektywami preprocesora.
Ten fakt wpływa istotnie na sposób ich organizacji w programie. Obecnie znanych jest kilka możliwych dróg prowadzących do celu; nazywamy je **modelami**. W tym paragrafie popatrzymy sobie zatem na wszystkie trzy modele porządkowania kodu szablonów.

Model włączania

Najwcześniejszym i do dziś najpopularniejszym sposobem zarządzania szablonami jest **model włączania** (ang. *inclusion model*). Jest on jednocześnie całkiem prosty w stosowaniu i często wystarczający. Przyjrzyjmy mu się.

Zwykły kod

Wpierw jednak przypomnimy sobie, jak należy radzić sobie z kodem C++ bez szablonów. Otóż, jak wiemy, wyróżniamy w nim **pliki nagłówkowe** oraz **moduły kodu**. I tak:

- pliki nagłówkowe są opatrzone rozszerzeniami *.h*, *.hh*, *.hpp*, lub *.hxx* i zawierają deklaracje współużytkowanych części kodu. Należą do nich:
  - ✓ prototypy funkcji
  - ✓ deklaracje zapowiadające zmiennych globalnych (opatrzone słowem `extern`)
  - ✓ definicje własnych typów danych i aliasów, wprowadzane słowami `typedef`, `enum`, `struct`, `union` i `class`
  - ✓ implementacje funkcji *inline*
- moduły kodu są z kolei wyróżniane rozszerzeniami *.c*, *.cc*, *.cpp* lub *.cxx* i przechowują definicje (tudzież implementacje) zadeklarowanych w nagłówkach elementów programu. Są to więc:
  - ✓ instrukcje funkcji globalnych oraz metod klas
  - ✓ deklaracje zmiennych globalnych (bez `extern`) i statycznych pól klas

Ten system, spięty dyrektywami `#include`, działa wyśmienicie, oddzielając to, co jest ważne do stosowania kodu od technicznych szczegółów jego implementacji. Co się jednak dzieje, gdy na scenę wkraczają szablony?…

Próbujemy zastosować szablony

Spróbujmy więc podobną metodę zastosować wobec szablonu funkcji max(). Najpierw umieśćmy jej prototyp (deklarację) w pliku nagłówkowym:

```
// max.hpp

// prototyp szablonu max()
template <typename T> T max(T, T);
```

Następnie treść funkcji podamy w module kodu *max.cpp*:

```
// max.cpp

#include "max.hpp"

// implementacja szablonu max()
template <typename T> T max(T a, T b)
{
   return (a > b ? a : b);
}
```

Wreszcie, wykorzystamy naszą funkcję w programie, wypisując na przykład na ekranie większą z podanych liczb:

```
// TemplatesTryout - próba zastosowania szablonu funkcji

// main.cpp

#include <iostream>
#include <conio.h>
#include "max.hpp"

int main()
{
   std::cout << "Podaj dwie liczby:" << std::endl;

   double fLiczba1, fLiczba2;
```

```
    std::cin >> fLiczba1;
    std::cin >> fLiczba2;

    std::cout << "Wieksza jest liczba " << max(fLiczba1, fLiczba2);
    getch();
}
```

Pieczołowicie wykonując te proste w gruncie rzeczy czynności, mamy prawo czuć się zaskoczeni efektami. Próba wygenerowania gotowego programu skończy się bowiem komunikatem linkera zbliżonym do poniższego:

```
error LNK2019: unresolved external symbol "double __cdecl max(double,double)" (?max@@YANNN@Z)
referenced in function _main
```

Wynika z niego klarownie, że funkcja `max()` w wersji skonkretyzowanej dla `double`… nie istnieje! Jak to wyjaśnić?

Wytłumaczenie jest w miarę proste. Zwróć uwagę, że dołączenie pliku *max.hpp* włącza do *main.cpp* jedynie **deklarację szablonu**, a nie jego definicję. Nie mając definicji kompilator nie może natomiast skonkretyzować szablonu dla parametru `double`. Wobec tego czyni on założenie, że funkcja `max<double>()` została wygenerowana gdzie indziej. Nie ma w tym nic zdrożnego - ten sam mechanizm działa przecież dla zwykłych funkcji, które są deklarowane (prototypowane) w pliku nagłówkowym, a implementowane w innym module. Niestety, w tym przypadku jest to założenie błędne: konkretyzacja nie zostanie bowiem przeprowadzona z powodu wspomnianego braku informacji (definicji szablonu).
Ostatecznie więc powstaje zewnętrzne dowiązanie do specjalizacji szablonu `max()` dla parametru `double` - specjalizacji, która **nie istnieje**! Ten fakt nie umknie już uwadze linkera, czego skutkiem jest zaprezentowany wyżej błąd i porażka konsolidacji.

Rozwiązanie - istota modelu włączania

Sytuacja patowa? Bynajmniej. Istnieje oczywiście rozwiązanie tego problemu: trzeba po prostu zapewnić widoczność definicji szablonu `max()` (czyli zawartości *max.cpp*) w miejscu jego użycia (czyli *main.cpp*). Można to uczynić poprzez:

- dołączenie zawartości *max.cpp* do *max.hpp* (dodanie `#include "max.cpp"` na końcu *max.hpp*)
- dołączenie *max.cpp* w module *main.cpp* zamiast dołączania *max.hpp*
- przeniesienie zawartości modułu *max.cpp* (czyli definicję szablonu `max()`) do pliku nagłówkowego *max.hpp*

Wszystkie te sposoby są wariantami **modelu włączania**, o którym mówimy w tym paragrafie. Zastosowanie któregokolwiek spowoduje pożądany efekt, czyli poprawną kompilację kodu. W praktyce jednak najczęściej stosuje się sposób trzeci, czyli umieszczanie **całego kodu szablonów w pliku nagłówkowym**.

Mimo takiego postępowania funkcje szablonowe **nie będą rozwijane** w miejscu wywołania. Aby szablon funkcji był funkcją *inline*, należy jawnie poprzedzić ją przydomkiem `inline` po klauzuli `template <...>`.

Model włączania działa całkiem dobrze zarówno dla małych, jak i nieco większych średnich projektów. Jest z nim jednak związany pewien mankament: w oczywisty sposób powoduje on rozrost plików nagłówkowych. Sprawia to, że koszt ich dołączania staje się coraz większy, co w konsekwencji wydłuża czas kompilacji projektów. Staje się to aczkolwiek zauważalne i znaczące dopiero w naprawdę dużych programach (rzędu kilkunastu-kilkudziesięciu tysięcy linii).

W sumie można więc powiedzieć, że model włączania jest zadowolającym sposobem zarządzania kodem szablonów. Nie jest to jednak wystarczający argument za tym, aby nie przyjrzeć się także innym modelom :)

Konkretyzacja jawna

W błędnym przykładzie programu z szablonem `max()` problem polegał na tym, że kompilator nie miał okazji do właściwego skonkretyzowania szablonu. Model włączania umożliwiał mu to w sposób automatyczny.
Istnieje aczkolwiek inna metoda na rozwiązanie tego problemu. Możemy mianowicie zastosować model **konkretyzacji jawnej** (ang. *explicit instantiation*) i przejąć kontrolę nad procesem rozwijania szablonów. Zobaczmy zatem, jak można to zrobić.

Instrukcje jawnej konkretyzacji

Wyjaśniłem, że powodem komunikatu linkera i nieudanej konsolidacji przykładu z poprzedniego akapitu jest nieobecność funkcji `max<double>()` w żadnym ze skompilowanych modułów. Możemy to zmienić, sami wprowadzając rzeczoną funkcję - czyli **jawnie ją skonkretyzować**. Czynimy w następujący sposób:

```
// max_inst.cpp

#include "max.cpp"

// jawna konkretyzacjia szablonu max() dla parametru double
template double max<double>(double, double);
```

Mamy tutaj **dyrektywę konkretyzacji jawnej** (ang. *explicit instantiation directive*). Jak widać, składa się ona z samego słowa `template` (bez nawiasów ostrych) oraz pełnej deklaracji specjalizacji szablonu (czyli `max<double>()`). Tutaj akurat mamy funkcję, ale podobnie konkretyzacja jawna wygląda klas. W każdym przypadku **konieczna jest definicja konkretyzowanego szablonu** - stąd dołączenie do naszego nowego modułu pliku *max.cpp*.

Nalezy zwrócić uwagę, aby każda specjalizacja szablonu była wprowadzana jawnie **tylko jeden raz**. W przeciwnym razie zwróci na to uwagę linker.

Wady i zalety konkretyzacji jawnej

Zastosowanie takiego wybiegu spowoduje teraz poprawną kompilację i linkowanie programu. Możemy się więc przekonać, że konkretyzacja jawna faktycznie działa.

Nie ma jednak róży bez kolców. Ten sposób zarządzania specjalizacjami szablonu ma oczywistą wadę - jedną, ale za to bardzo dotkliwą. Wymaga on od programisty śledzenia kodu, który wykorzystuje szablony, celem rozpoznawania wymaganych specjalizacji oraz ich jawnego deklarowania. Zwykle robi się to w osobnym module (u nas *max_inst.cpp*), aby nie zaśmiecać właściwego kodu programu.
Nie da się ukryć, że niweluje to jedną z bezdyskusyjnych zalet szablonów, czyli możliwość zrzucenia na barki kompilatora kwestii wygenerowania właściwego ich kodu. Jest to szczególnie niezadowalające w przypadku szablonów funkcji, gdzie przy każdym ich wywołaniu musimy zastanowić się, jaka wersja szablonu zostanie w tym konkretnym wypadku użyta. Faktycznie więc trudno nawet czerpać korzyści z automatycznej dedukcji parametrów szablonu na podstawie parametrów funkcji - a to jest przecież jedno z głównych dobrodziejstw szablonów.

Konkretyzacja jawna ma aczkolwiek także kilka zalet, do których należą:

- możliwość sprawowania kontroli nad procesem rozwijania szablonów
- zapobieganie nadmiernego rozdęciu plików nagłówkowych, a więc potencjalne skrócenie czasu kompilacji

- umożliwie dokładnego określenia miejsca (modułu kodu), w którym egzemplarz szablonu (specjalizacja) zostanie utworzony

W większości przypadków te argumenty nie są jednak wystarczające, aby mogły przeważyć na rzecz wykorzystania modelu konkretyzacji jawnej w praktyce. Podobnie bowiem jak w przypadku modelu włączania, rozrost programu powoduje także wydłużenie czasu przeznaczonego na konkretyzację. Różnica tkwi jednakże w tym, że w tym pierwszym modelu całą pracą zajmuje się kompilator, który i tak nie ma nic ciekawszego do roboty, natomiast konkretyzacja jawna zrzuca ten obowiązek na barki wiecznie zapracowanego programisty.
W sumie więc ten model organizacji szablonów trudno uznać za praktyczny i wygodny. Być może sprawdziłby się nieźle w małych programach, ale tam można sobie przecież tym bardziej pozwolić na znacznie wygodniejszy model włączania.

Model separacji

Lekarstwem na bolączki modelu włączania ma być mechanizm **eksportowania szablonów**. Technika ta, nazywana również **modelem separacji**, jest częścią samego języka C++ i teoretycznie jest to właśnie ten sposób zarządzania kodem szablonów, który ma być preferowany. Przynajmniej tako rzecze Standard C++.

Tym niemniej już od razu powiadomię, że w miarę poprawna obsługa tego modelu jest dostępna dopiero w Visual Studio .NET 2003.

Wypadałoby zatem poznać bliżej to natywne rozwiązanie samego języka.

Szablony eksportowane

Idea tego modelu jest generalnie bardzo prosta:

- zachowany zostaje naturalny porządek oddzielania deklaracji/definicji od implementacji. W pliku nagłówkowym umieszczamy więc wyłącznie deklaracje (prototypy) szablonów funkcji oraz definicje szablonów klas. Postepujemy zatem tak, jak próbowaliśmy czynić na samym początku - dopóki linker nie sprowadził nas na ziemię
- zmiana polega jedynie na tym, że **deklarację szablonu w pliku nagłówkowym** opatrujemy słowem kluczowym `export`

Stosując te dwie wskazówki do naszego błędnego przykładu `TemplatesTryout`, należałoby jedynie zmodyfikować plik *max.hpp*. Zmiana ta jest zresztą niemal kosmetyczna:

```
// max.hpp

// prototyp szablonu max() jako szablon eksportowany
export template <typename T> T max(T, T);
```

Jak się wydaje, dodanie słowa `export` przed deklarację szablonu załatwia sprawę.

W rzeczywistości słowo to powinno się znaleźć **przed każdym** użyciem klauzuli `template <...>`. `export` ma jednak tę przyjemną właściwość, że po jednokrotnym jego zastosowaniu w obrębie danego pliku z kodem **wszystkie dalsze szablony** otrzymują ten przydomek niejawnie. A dzięki temu, że w pliku *max.cpp* znajduje się odpowiednia dyrektywa `#include`:

```
// max.cpp

#include "max.hpp"
```

```
// (dalej implementacja szablonu max())
```

również kod szablonu funkcji `max()` dostaje modyfikator `export` w prezencie od pliku nagłówkowego *max.hpp*. Jeśli więc zdecydujemy się pisać kod szablonów w identyczny sposób, jak zwykły kod C++, to nasza troska o właściwą kompilację szablonów powinna ograniczać się do dodawania słowa kluczowego `export` przed deklaracjami `template <...>` w plikach nagłówkowych.

Przynajmniej teoretycznie tak właśnie powinno być…

Nie ma róży bez kolców

Model separacji może ci się teraz wydawać rodzajem białej magii, likwidującej wszystkie mankamenty organizacji kodu szablonów. Trzeba sobie jednak zdawać sprawę, że nie jest on pozbawiony wad. Czas więc zdjąć z twarzy ten szczęśliwy uśmieszek i przyjrzeć się rzeczywistości.

A rzeczywistość skrzeczy. Przede wszystkim należy wiedzieć, że mimo kilkuletniej już obecności w Standardzie C++ i w świadomości sporej części programistów (przynajmniej tych co bardziej zainteresowanych rozwojem języka), szablony eksportowane są w pełni obsługiwane przez nieliczne kompilatory. Dopiero ich najnowsze wersje (jak na przykład Visual Studio .NET 2003) radzą sobie ze słowem kluczowym `export`.
Ze względu na tak nikłe doświadczenia praktyczne trudno też przewidzieć potencjalne problemy, jakie mogą (choć oczywiście nie muszą) przydarzyć się podczas korzystania z modelu separacji. Te rzadkie kompilatory radzące sobie z tym modelem mogą bowiem działać świetnie przy małych czy nawet średnich projektach, ale nie jest wcale powiedziane, czy przy większych programach nie ujawnią się w nich jakieś kłopoty. Wiadomo wszakże, że najlepszym probierzem jakości wszelkich produktów - także możliwości kompilatorów - jest ich intensywne wykorzystywanie przez rzesze użytkowników. W tym zaś przypadku nie jest to jeszcze powszechną praktyką (przynajmniej nie tak bardzo, jak inne elementy C++), choć należy rzecz jasna oczekiwać, że sytuacji będzie się z czasem poprawiać.

Druga sprawa związana jest z samym działaniem słowa kluczowego `export`. W przybliżeniu można je scharakteryzować jako ukrycie funkcjonalności nieeleganckiego modelu włączenia - oczywiście wraz z pewnymi usprawnieniami. Oznacza to więc, że nie dokonują się tu żadne cuda: pozorne zerwanie związku między definicją a konkretyzacją szablonu musi i tak być odtworzone przez kompilator. To sprawia, że jakoby niezależne od siebie moduły kodu stają się związane właśnie ze względu na obecność w nich implementacji szablonów. W ostateczności koszt czasowy kompilacji programu wcale nie musi być wiele mniejszy od tego, jaki jest doświadczany w modelu włączania.

Wszystko to nie znaczy jednak, że nie należy spoglądać na model separacji przychylnym okiem. Czas działa bowiem na jego korzyść. Gdy obsługa szablonów eksportowanych stanie się powszechna, postępować będzie także jej usprawnienie pod względem niezawodności i efektywności. Wcale niewykluczone, że na tym polu zostawi za jakiś czas daleko w tyle model włączania.
A już teraz model separacji oferuje nam zaletę niespotykaną w innych rozwiązaniach problemu szablonów: elegancję, podobną do tej znanej ze zwykłego, nieszablonowego kodu. Dalej będzie zapewne już tylko lepiej.

Współpraca modelu włączania i separacji

Ucieszyć może także fakt, że stosunkowo łatwo zorganizować kod szablonów w taki sposób, aby „przełączanie” między modelem włączania i separacji nie zajmowało więcej niż kilka sekund (nie licząc rekompilacji). Dosyć dobrze do tego celu nadają się dyrektywy preprocesora.

Pomysł jest prosty. Należy tak zmodyfikować plik nagłówkowy z deklaracją szablonu (u nas *max.hpp*), by w razie potrzeby „zawierał" on również jego definicję - czyli włączał ją z modułu kodu (*max.cpp*). Oto propozycja takiej modyfikacji:

```
// max.hpp

// zabezpieczenie przed wielokrotnym dołączaniem - ważne!
#pragma once

// w zależności od tego, czy zdefiniowano makro USE_EXPORT,
// wprowadzamy do programu słowo kluczowe export
#ifdef USE_EXPORT
   #define EXPORT export
#else
   #define EXPORT
#endif

// deklaracja szablonu
EXPORT template <typename T> T max(T, T);

// jeżeli nie używamy modelu separacji, to potrzebujemy także
// definicji szablonu. Włączamy ją więc
#ifndef USE_EXPORT
   #include "max.cpp"
#endif
```

Decyzja co do używanego modelu ograniczać się tu będzie do zdefiniowania lub niezdefiniowania makra `USE_EXPORT` przed dołączeniem pliku *max.hpp*:

```
// używanie modelu separacji; bez #define będzie to model włączania
#define USE_EXPORT
#include "max.hpp"
```

Trzeba jeszcze pamiętać, aby w tym pliku nagłówkowym przynajmniej pierwszą deklarację szablonu (a najlepiej wszystkie) opatrzyć nazwą makra `EXPORT`. W zależności od wybranego modelu będzie ono bowiem rozwinięte do słowa `export` lub do pustego ciągu, co w wyniku da nam zastosowanie wybranego modelu.

Opisana „sztuczka" opiera się, w przypadku użycia modelu włączania, o sprzężenie zwrotne dyrektyw `#include`: *max.hpp* dołącza bowiem *max.cpp*, zaś *max.cpp* próbuje dołączyć *max.hpp*. Trzeba rzecz jasna zadbać o to, by ta próba nie zakończyła się powodzeniem, stosując jedno z zabezpieczeń przeciw wielokrotnemu dołączaniu. Tutaj użyłem `#pragma once`, choć metoda z unikalnym makrem oraz `#ifndef`/`#endif` również zdałaby egzamin.

***

I tak oto zakończyliśmy drugi podrozdział poświęcony opisowi szablonów w C++. W zasadzie możesz uznać ten moment za koniec teorii tego skomplikowanego zagadnienia. Chociaż więc zajmowaliśmy się już sprawami bardziej praktycznymi (jak choćby modelem organizacji kodu), to dopiero w następnym podrozdziale poznasz prawdziwe zastosowania szablonów. Zacznie się więc robić bardzo ciekawie, jako że dopiero w konkretnych metodach na wykorzystanie szablonów widać prawdziwą potęgę tego składnika C++. Pora zatem ją ujarzmić!

Zastosowania szablonów

Jeszcze w początkach tego rozdziału powiedziałem, do czego służą szablony w języku C++. Przypominam: stosujemy je głównie tam, gdzie chcemy uniezależnić kod programu od konkretnego typu danych.

To ogólnikowe stwierdzenie jest z pewnością pomocne, ale mało konkretne. Na pewno będziesz bardziej zadowolony, jeżeli ujrzysz jakieś precyzyjniej określone zastosowania dla szablonów. I to jest właśnie treścią tego podrozdziału. Pomówimy sobie więc o niektórych sytuacjach, gdy skorzystanie z szablonów ułatwia lub wręcz umożliwia wykonanie ważnych programistycznych zadań.

Zastąpienie makrodefinicji

Gdyby to była bajka, to zaczęłoby się tak: dawno, dawno temu w królestwie Elastycznych Programów niepodzielnie rządziła okrutna kasta Makrodefinicji. Dość często utrudniała ona życie mieszkańcom, powodując większe lub mniejsze życiowe uciążliwości. Na szczęście pewnego dnia na pomoc przybyli dzielni rycerze Szablonów, którzy obalili tyranów i zapewnili królestwu szczęśliwe życie pod rządami nowych, łaskawych władców. I wszyscy żyli długo i szczęśliwie.

To tyle, jeśli chodzi o otoczkę baśniową, bo teraz należałoby wrócić do rzeczywistego zagadnienia. Jakiś czas temu mieliśmy okazję poznać dyrektywy preprocesora, zwracając przy tym szczególną uwagę na **makra**. Makra imitujące funkcje były kiedyś jedynym sposobem na tworzenie „kodu" niezwiązanego z żadnym typem danych. Teraz zaś mamy już szablony. Czy są one lepsze?…

Szablon funkcji i makro

Aby się o tym przekonać, porównajmy funkcję `max()` - napisaną raz w postaci szablonu i drugi raz w postaci makra:

```
// szablon funkcji max()
template <typename T> T max(T a, T b)  { return (a > b ? a : b); }

// makro MAX()
#define MAX(a,b)      ((a) > (b) ? (a) : (b))
```

Widać parę podobieństw, ale i mnóstwo różnic. Przede wszystkim interesuje nas to, w jaki sposób makra i szablony osiągają niezależność od typu danych - parametrów. W sumie wiemy to dobrze:

- w szablonach występują parametry będące typami (jak u nas `T`), nieodpowiadające jednak żadnemu konkretnemu typowi danych. Poprzez konkretyzację tworzone są potem specjalizowane egzemplarze funkcji, działające dla ściśle określonych już rodzajów zmiennych
- makra w ogóle nie posługują się pojęciem 'typ danych'. Ich istota polega na zwykłej zamianie jednego tekstu („wywołania" makra) w inny tekst (rozwinięcie makra). Dopiero to rozwinięcie jest przedmiotem zainteresowania kompilatora, który wedle swoich reguł - jak choćby poprawnego użycia operatorów - uzna je za poprawne bądź nie

Mamy więc dwa różne podejścia i zapewne już wiesz lub domyślasz się, że **nie są** one równoważne ani nawet równie dobre. Należy więc odpowiedzieć na proste pytanie - co jest lepsze?

Pojedynek na szczycie

W tym celu spróbujmy użyć obu zaprezentowanych wyżej konstrukcji, poddając je swoistym próbom:

```
// będziemy potrzebowali kilku zmiennych
int nA = 42;   float fB = 12.0f;

// i startujemy...
std::cout << max(34, 56)    << " | " <<  MAX(34, 56) << std::endl;   // 1
std::cout << max(nA, fB)    << " | " <<  MAX(nA, fB) << std::endl;   // 2
std::cout << max(nA++, fB)  << " | " <<  MAX(nA++, fB) << std::endl; // 3
```

Czy obie konstrukcje przejdą je z powodzeniem?… Cóż, odpowiedź jest niestety przecząca. Tylko pierwsza linijka nie wymaga żadnych uwag ani analizy. W tym przypadku nie ma po prostu żadnych wątpliwości: obie wartości do porównania są jednoznacznymi stałymi tych samych typów. Wszystko więc pójdzie gładko.
Jednak dalej zaczynają się już kłopoty…

Starcie drugie: problem dopasowania tudzież wydajności

Popatrzmy więc, co się właściwie stanie w tym kodzie. Pomyślmy mianowicie, w jaki sposób poradzi sobie z zadaniem szablon funkcji, a w jaki - makrodefinicja.

Jak zadziała szablon

Funkcjonowanie szablonów było przedmiotem sporej części aktualnego rozdziału, zatem odpowiedź na pytanie powyżej nie powinna ci nastręczać trudności. Szablon `max()` zadziała tak, jak się spodziewamy: jego użycie spowoduje konkretyzację dla właściwego parametru, co w wyniku da normalną funkcję, wykorzystywaną przez program.

Wpierw jednak musi być znany parametr `T` szablonu - zostanie on oczywiście wydedukowany z wywołania funkcji `max()`. Mamy w nim argumenty będące zmiennymi: pierwsza jest typu `int`, zaś druga typu `float`. Parametr szablonu `T` jest natomiast tylko jeden - cóż więc?… Naturalnie, kompilator wybierze tak, aby nie skrzywdzić żadnego z argumentów funkcji, decydując się na typ `float`. Pomieści on bowiem zarówno liczbę całkowitą, jak i rzeczywistą. Szablon `max()` zostanie więc skonkretyzowany do postaci:

```
float max<float>(float a, float b)        { return (a > b ? a : b); }
```

I wszystko byłoby w porządku, gdyby nie jeden drobny niuans, w zasadzie niedostrzegalny na pierwszy rzut oka. Jak to zwykle bywa w niejasnych sytuacjach, chodzi o wydajność. Zwróćmy uwagę, że parametry funkcji `max()` są tu przekazywane **poprzez wartość**. Potencjalnie więc może to prowadzić do dwóch niepotrzebnych kopiowań, wykonywanych podczas wywoływania funkcji w skompilowanym programie. Oczywiście, ma to znaczenie tylko dla dużych obiektów, lecz kto powiedział, że nie moglibyśmy chcieć użyć tej funkcji na przykład do 1000-elementowej tablicy?…

Powiesz pewnie, że jest to na to rada. Wystarczy skorzystać z wynalazku C++ znanego pod nazwą referencji. Przypomnijmy, że referencje, czyli „ukryte wskaźniki”, nie powodują przekazania do funkcji samego obiektu, lecz tylko jego adresu. Ich zaletą jest zaś to, że nie zmuszają do korzystania z kłopotliwej w gruncie rzeczy składni wskaźników.
Pamiętając o tym, ochoczo przerabiamy nasz szablon na wersję korzystającą z referencji:

```
template <typename T> T max(const T& a, const T& b)
{
   return (a > b ? a : b);
}
```

W ten sposób niechcący pozbawiliśmy kompilator ważnej możliwości: używania niejawnych konwersji. W momencie, gdy chcemy przekazać do funkcji nie obiekt, a referencję do niego, kompilator staje się po prostu ślepy na ten mechanizm języka. Łatwo to zresztą wyjaśnić: istotą referencji jest odwoływanie się do istniejącego obiektu bez kopiowania, zaś istniejący obiekt ma swój typ, którego zmienić nie można.

Więc co zrobić? Najlepiej po prostu… pogodzić się z tym „strasznym marnotrawstem", które i tak nie jest szczególnie wielkie, a przez dobry kompilator może być nawet z niezłym skutkiem minimalizowane.
Naturalnie, można próbowac kombinować dalej - chociażby dodać drugi parametr szablonu. Tyle że wtedy pozostanie nierozstrzygalny wybór, który z nich uczynić typem wartości zwracanej. Naturalnie, można ten typ dodać jako kolejny, trzeci już parametr szablonu i kazać go podawać wywołującemu. Wreszcie, można nawet użyć jednego z kilku dość pokrętnych (koncepcyjnie i składniowo) sposobów na obejście problemu - ale chyba nie zmartwisz się tym, że ci ich tutaj oszczędzę. Nadmierna komplikacja jest tu bowiem wysoce niewskazana; zaangażowane środki będą zwyczajnie niewspółmierne do zysków.

Jak zadziała makro

Przekonajmy się więc, co ma do powiedzenia makrodefinicja. Tutaj cała sprawa jest rzecz jasna znacznie łatwiejsza: preprocesor rozwinie nam po prostu kod `MAX(nA, fB)` do postaci następującego wyrażenia:

```
((nA) ? (nB) ? (nA) : (nB))
```

Nie ma tutaj absolutnie rzadnej różnicy z sytuacją, w której to wyrażenie zostałoby wpisane bezpośrednio do kodu. Żadna funkcja nie jest generowana, żadne konwersje argumentów nie są wykonywane, po prostu nie ma żadnego przeskoku z miejsca „wywołania" makra w inne miejsce programu. Kompilator jest wręcz utrzymywany w błogiej nieświadomości, gdyż dostaje wyklarowany już kod bez makr. Wszystkim zajmuje się preprocesor i to on sprawia, że makro działa.

Wynik

Ostatecznie możemy uznać remis obu rozwiązań, aczkolwiek z lekkim wskazaniem na makrodefinicje. Z wyjątkiem fanatyków wydajności nie ma jednak bodaj nikogo, kto uważałby „nieefektywne" działanie szablonów za wielki błąd. A tym, którzy rzeczywiście tak uważają, pozostaje chyba tylko przerzucenie się na język asemblera :)

Starcie trzecie: problem rozwinięcia albo poprawności

Próba trzecia jest w takim razie decydująca. Ponownie rozłożymy na czynniki pierwsze sposób działania szablonu i makra.

Jak zadziała szablon

Działanie szablonu będzie tu łudząco podobne do poprzedniej próby. Znowu bowiem argumenty funkcji `max()` musza być dopasowane do typu ogólniejszego - czyli do `float`. Powstanie więc specjalizacja `max<double>()`.

Funkcja ta będzie potem wywoływana z argumentami `nA++` i `fB`. Wobec tego zwróci ona większą spośród liczb: `nA+1` i `fB`. Właściwie więc nie ma nad czym dłużej deliberować; nasz szablon zachowa się zupełnie poprawnie, prawie jak zwyczajna funkcja. Naturalnie, stosują się tutaj wszystkie uwagi z poprzedniego akapitu - nie ma sensu ponownie ich przytaczać.
Ogółem test uważamy za zaliczony.

Jak zadziała makro

A teraz czas na analizę makrodefinicji i jej użycia w formie `MAX(nA++, fB)`. Pamiętając, jak działa preprocesor, słusznie można wywnioskować, że zamieni on „wywołanie" makra na takie oto wyrażenie:

```
((nA++) > (fB) ? (nA++) : (fB))
```

Wszystko jest zatem w porządku?… Nie całkiem. Wręcz przeciwnie. Mamy problem. Poważny problem. A jego przyczyną jest obecność instrukcji `nA++` **dwukrotnie**.
Fakt ten sprawi mianowicie, że zmienna `nA` zostanie **dwa razy** zwiększona o 1!
Ostatecznie warunek powyżej zwróci **błędny wynik** - różniący się od właściwego o ową problematyczną jedynkę.

Jeśli pamiętasz dokładnie rozdział o preprocesorze, takie zachowanie nie powinno być dla ciebie zaskoczeniem. Już wtedy zaprezentowałem przykład tego problemu i ostrzegłem przed stosowaniem makrodefinicji w charakterze funkcji.

Wynik

Cóż można więcej powiedzieć? Błędny rezultat użycia makra sprawia, że makrodefinicje nie tylko przegrywają, ale właściwie zostają zdyskwalifikowane jako narzedzia tworzenia kodu niezależnego od typu. Bezapelacyjnie wygrywają szablony!

Konkluzje

Wniosek jest właściwie jeden:

Należy używać **szablonów funkcji** zamiast **makr**, które mają **udawać funkcje**.

Makrodefinicje w rodzaju `MAX()`, `MIN()` czy innych tego rodzaju nie mają już więc właściwie racji bytu. Zastąpiły je całkowicie szablony funkcji, oferujące nie tylko te same rezultaty (przy zastosowaniu `inline` - również wydajnościowe), ale też jedną konieczną cechę, której makrom brak - **poprawność**.
Szablony są po prostu bardziej inteligentne, jako że odpowiada za nie przemyślnie skonstruowany kompilator, a nie jego ułomny pomocnik - preprocesor. Jak się też miałeś okazję przekonać w tym rozdziale, możliwości szablonów funkcji są nieporównywalnie większe od tych dawanych przez makrodefinicje.

Nie znaczy to oczywiście, że makra zostały całkowicie zastąpione przez szablony. Nadal bowiem znajdują one zastosowanie tam, gdzie chcemy dokonywać operacji na kodzie jak na zwykłym tekście - a więc na przykład do wstawiania kilku często wystepujących instrukcji, których nie możemy wyodrębnić w postaci funkcji. Niemniej należy podkreślać (co robię po raz n-ty), że makra **nie służą do imitacji funkcji**, gdyż same funkcje (lub ich szablony) doskonale radzą sobie ze wszystkimi zadaniami, jakie chcielibyśmy im powierzyć. Naocznie to zresztą zobaczyliśmy.

Struktury danych

Szablony funkcji mają więc swoje ważne zastosowanie. Właściwie jednak to szablony klas są użyteczne w znacznie większym stopniu. Wykorzystujemy je bowiem w celu implementacji w programach tzw. **struktur danych**.

Jak głosi stare programistyczne „równanie”, obok algorytmów to struktury danych są głównymi składnikami programów[131]. Jak wskazuje nazwa tego pojęcia, służą one do przemyślanej organizacji informacji przetwarzanych przez aplikację. Zazwyczaj też struktury danych ściśle współpracują z algorytmami programu.
Z najprostszymi strukturami danych zapoznałeś się już całkiem dawno temu. Typowym przykładem może być zwykła, jednowymiarowa tablica; inny to np. struktura języka C++ (definiowana poprzez `struct`), zwana czasem **rekordem**. To jednak tylko wierzchołek góry lodowej. Wśród wielu struktur danych większość jest o wiele bardziej wyspecjalizowana i funkcjonalna.

Cóż jednak ma to wspólnego z szablonami?… Otóż bardzo wiele. Dzięki mechanizmowi parametryzowanych typów (czyli szablonów klas) implementacja przeróżnych struktur danych w C++ jest prosta. Przynajmniej jest ona prosta w tym sensie, że nie nastręcza kłopotów związanych z nieokreślonymi typami danych. Szablony załatwiają za nas tę sprawę, dzięki czemu owe struktury mogą być uniwersalne.
Prawdopodobnie właśnie to zastosowanie było jednym z głównych powodów, dla którego w ogóle wprowadzono do języka C++ narzędzia szablonów. Nam pozostaje się tylko z tego cieszyć… no, możnaby jeszcze przyjrzeć się sprawie nieco bliżej :) Zróbmy więc to.

W tej sekcji porozmawiamy sobie zatem o tym, jak szablony pomogają w tworzeniu struktur danych w programach. Naturalnie, temat ten jest niezwykle szeroki i dlatego nie będziemy w niego wnikać dokładnie. Niemniej będzie to dobra rozgrzewka przez poznawaniem Biblioteki Standardowej, która szeroko używa szablonów do implementacji struktur danych.
Omówimy więc sobie dwie najprostsze kategorie takich struktur: krotki i kontenery (pojemniki).

Krotki

Krotką (ang. *tuple*, nie mylić ze stokrotką ;)) nazywamy połączenie **kilku wartości różnych typów** w **jedną całość**. C++, podobnie jak wiele innych języków programowania umożliwia na zrealizowanie takiej koncepcji przy użyciu struktury, zawierającej dwa, trzy, cztery lub większą liczbę pól dowolnych typów.

Tutaj jednak chcemy zobaczyć w akcji szablony, zatem stworzymy nieco bardziej elastyczne rozwiązanie.

Przykład pary

Najprotszą krotką jest oczywiście… pojedyncza wartość :) Ponieważ jednak w jej przypadku do szczęścia wystarcza normalna zmienna, zajmijmy się raczej zespołem dwóch wartości. Zwiemy go **parą** (ang. *pair*) lub **duetem** (ang. *duo*).

Definicja szablonu

Mając w pamięci fakt, iż chcemy otrzymać parę dwóch wartości dwóch różnych typów, wyprodukujemy zapewne szablon podobny do poniższego:

```
template <typename T1, typename T2> struct TPair
{
   T1 Pierwszy;      // wartość pierwszego pola
   T2 Drugi;         // wartość drugiego pola
};
```

[131] To równanie to *Algorytmy + struktury danych = programy*, bedące jednocześnie tytułem słynnej książki Niklausa Wirtha.

Zastosowań takiej prostej struktury jest całe mnóstwo. Przy jej użyciu możemy na przykład w łatwy sposób stosować technikę informowania o błędach przy pomocy rezultatu funkcji. Oto przykład:

```
TPair<bool, T> Wynik = Funkcja();      // funkcja zwraca parę wartości
if (Wynik.Pierwszy)
{
   // wykonanie funkcji powiodło się; jej właściwy rezultat to
   // Wynik.Drugi
}
```

Wynik jako zespół dwóch wartości pozwala na oddzielenie właściwego rezultatu od danych błędu. Jednocześnie nie zatracamy informacji o typie wartości zwracanej przez funkcję - tutaj ukrywa się on za `T` i jest widoczny w prototypie funkcji.

Pomocna funkcja

Do wygodnego używania pary przydałby się sposób na jej łatwie utworzenie. Na razie bowiem `Funkcja()` musiałaby wykonywać np. taki kod:

```
TPair<bool, int> Wynik;    // obiekt wyniku
Wynik.Pierwszy = true;     // informacja o ewentualnym błędzie
Wynik.Drugi = 42;          // zasadniczy rezultat
return Wynik;              // zwracamy to wszystko
```

Sytuację możemy poprawić, dodając konstruktor(y):

```
template <typename T1, typename T2> struct TPair
{
   T1 Pierwszy;      // wartość pierwszego pola
   T2 Drugi;         // wartość drugiego pola

   //---------------------------------------------------------------------

   // konstruktory
   TPair() : Pierwszy(), Drugi()                    { }
   TPair(const T1& Wartosc1, const T2& Wartosc2)
         : Pierwszy(Wartosc1), Drugi(Wartosc2)      { }
};
```

W zasadzie to są one niezbędne - inaczej nie możnaby tworzyć par z obiektów, których klasy nie mają domyślnych konstruktorów. Tak czy owak, skracamy już zapis do skromnego:

```
return TPair<bool, int>(true, 42);
```

Nadal jednak można trochę ponarzekać. Kompilator nie jest na przykład na tyle inteligentny, aby wydedukować parametry szablonu `TPair` z argumentów konstruktora. To jednak można łatwo uzyskać, jako że umiejętność takiej dedukcji jest nieodłączną cechą szablonów funkcji. Możemy zatem stworzyć sobie pomocną funkcję `Para()`, tworzącą duet:

```
template <typename T1, typename T2>
   inline TPair<T1, T2> Para(const T1& Wartosc1, const T2& Wartosc2)
   {
         return TPair<T1, T2>(Wartosc1, Wartosc2);
   }
```

To wreszcie pozwoli na stosowanie krótkiej i przemyślanej formy tworzenia pary:

```
return Para(true, 42);
```

Przydomek `inline` zabezpiecza natomiast przed „niewybaczalnym” uszczerbkiem na wydajności spowodowanym pośrednią drogą kreacji obiektu.

Dalsze usprawnienia

Możemy dalej usprawniać szablon `TPair` - tak, aby wygoda korzystania z niego nie ustępowała niczym przyjemności użytkowania typów wbudowanych. Dodamy mu więc:

- operator przypisania
- konstruktor kopiujący

„Ale po co?”, możesz spytać. „Przecież w tym przypadku wersje tworzone przez kompilator pasują jak ulał”. Owszem, masz rację. Można je jednak poprawić, definiując obie metody jako **szablony**:

```
template <typename T1, typename T2> struct TPair
{
   T1 Pierwszy;      // wartość pierwszego pola
   T2 Drugi;         // wartość drugiego pola

   //---------------------------------------------------------------------

   // konstruktory (zwykłe i kopiująco-konwertujący)
   TPair() : Pierwszy(), Drugi()                    { }
   TPair(const T1& Wartosc1, const T2& Wartosc2)
         : Pierwszy(Wartosc1), Drugi(Wartosc2)      { }
   template <typename U1, typename U2> TPair(const TPair<U1, U2>& Para)
         : Pierwszy(Para.Pierwszy), Drugi(Para.Drugi)    { }

   //---------------------------------------------------------------------

   // operator przypisania
   template <typename U1, typename U2>
         operator=(const TPair<U1, U2>& Para)
         {
               Pierwszy = Para.Pierwszy;
               Drugi = Para.Drugi;
               return *this;
         }
};
```

W ten sposób pieczemy dwa befsztyki na jednym ogniu. Nasze metody pełnią bowiem nie tylko „rolę kopiującą”, ale i „rolę konwertującą”. Pary stają się więc kompatybilne względem niejawnym konwersji swoich składników; zatem np. para `TPair<int, int>` będzie mogła być od teraz bez problemów przypisana do pary `TPair<float, double>`, itd. Konieczne konwersje będą dokonywane podczas inicjalizacji (konstruktor) lub przypisywania (operator `=`) pól.

Do pełni funkcjonalności brakuje jeszcze możliwości porównywania par. To zaś osiągamy, definiując operatory `==` i `!=`. Także tutaj może zajść konieczność konfrontowania duetów o różnych typach pól, zatem ponownie należy użyć szablonu:

```
// operator równości
template <typename T1, typename T2, typename U1, typename U2>
   inline bool operator==(const TPair<T1, T2>& Para1,
                          const TPair<U1, U2>& Para2)
   {
         return (Para1.Pierwszy == Para2.Pierwszy
                 && Para1.Drugi == Para2.Drugi);
```

```
    }

// operator nierówności
template <typename T1, typename T2, typename U1, typename U2>
    inline bool operator!=(const TPair<T1, T2>& Para1,
                           const TPair<U1, U2>& Para2)
    {
          return (Para1.Pierwszy != Para2.Pierwszy
                  || Para1.Drugi != Para2.Drugi);
    }
```

Trochę makabrycznie na pierwszy rzut oka może wyglądać szablon z czterema parametrami. Powód jego wystąpienia jest jednak banalny: potrzebujemy po prostu parametryzacji typów dla obu porównywanych par. W sumie więc mogą wystąpić cztery typy pól, co ładnie przedstawiają deklaracje parametrów funkcji.
O tym, czy typy te będą ze sobą współgrały, zdecydują już porównywania w ciele funkcji operatorowych. Naturalnie, w przypadku braku identyczności lub niejawnych konwersji, kompilacji problematycznego użycia operatora nie powiedzie się.

Stworzony szablon `TPair` wraz z „oprzyrządowaniem” w postaci pomocniczej funkcji i przeciążonych operatorów jest bardzo podobny do klasy `std::pair` z Biblioteki Standardowej.

Trójki i wyższe krotki

Przyglądając się uważniej szablonowi pary, nietrudno jest dostrzec miejsca, które należy zmodyfikować, by otrzymać krotki wyższego rzędu - trójki, czwórki, piątki, itd. Pewnym problemem jest stałe zwiększanie długości klauzul `template <...>` i nazw typów krotek, ale to już jest niestety nieuknione. W praktyce więc rzadko używa się wielkich krotek - powyżej trzech, czterech elementów - także z tego powodu, że nie ma dla nich zbyt wielu sensownych zastosowań.

Dlatego też tutaj popatrzymy sobie tylko na analogiczny do `TPair` szablon trójki - `TTriplet`:

```
template <typename T1, typename T2, typename T3> struct TTriplet
{
    T1 Pierwszy;      // wartość pierwszego pola
    T2 Drugi;         // wartość drugiego pola
    T3 Trzeci;        // wartość trzeciego pola

    //-------------------------------------------------------------------

    // konstruktory (zwykłe i kopiująco-konwertujący)
    TTriplet() : Pierwszy(), Drugi(), Trzeci()                        { }
    TTriplet(const T1& Wartosc1, const T2& Wartosc2, const T3& Wartosc3)
          : Pierwszy(Wartosc1), Drugi(Wartosc2), Trzeci(Wartosc3)     { }
    template <typename U1, typename U2, typename U3>
          TTriplet(const TTriplet<U1, U2, U3>& Trojka)
                : Pierwszy(Trojka.Pierwszy),
                  Drugi(Trojka.Drugi), Trzeci(Trojka.Trzeci)    { }

    //-------------------------------------------------------------------

    // operator przypisania
    template <typename U1, typename U2, typename U3>
          operator=(const TTriplet<U1, U2, U3>& Trojka)
          {
                Pierwszy = Trojka.Pierwszy;
```

```
                Drugi = Trojka.Drugi;
                Trzeci = Trojka.Trzeci;

                return *this;
            }
};

// operator równości
template <typename T1, typename T2, typename T3,
          typename U1, typename U2, typename U3>
   inline bool operator==(const TTriplet<T1, T2, T3>& Trojka1,
                          const TTriplet<U1, U2, U3>& Trojka2)
   {
         return (Trojka1.Pierwszy == Trojka2.Pierwszy
                 && Trojka1.Drugi == Trojka2.Drugi
                 && Trojka1.Trzeci == Trojka2.Trzeci);
   }

// operator nierówności
template <typename T1, typename T2, typename T3,
          typename U1, typename U2, typename U3>
   inline bool operator==(const TTriplet<T1, T2, T3>& Trojka1,
                          const TTriplet<U1, U2, U3>& Trojka2)
   {
         return (Trojka1.Pierwszy != Trojka2.Pierwszy
                 || Trojka1.Drugi != Trojka2.Drugi
                 || Trojka1.Trzeci != Trojka2.Trzeci);
   }

//---------------------------------------------------------------------------

// wygodna funkcja tworząca trojkę
template <typename T1, typename T2, typename T3>
   inline TTriplet<T1, T2, T3> Trojka(const T1& Wartosc1,
                                      const T2& Wartosc2,
                                      const T3& Wartosc3)
   {
         return TTriplet<T1, T2, T3>(Wartosc1, Wartosc2, Wartosc3);
   }
```

Wygląda on lekko strasznie, ale też pokazuje wyraźnie, że szablony w C++ to naprawdę potężne narzedzie. Pomyśl, czy w ogóle sensowne byłoby implementowanie krotek bez nich?

Wyższe krotki wygodnie jest programować w sposób rekurencyjny, wykorzystując jedynie szablon pary. Przy takim podejściu trójka np. typu `TTriplet<int, float, std::string>` jest przechowywana jako typ `TPair<int, TPair<float, std::string> >` - czyli parę, której elementem jest kolejna para. Analogicznie wygląda to dalej.
Takie podejście, w połączeniu z kilkoma innymi, maksymalnie wykręconymi technikami, daje możliwość tworzenia krotek dowolnego rzędu. Takie rozwiązanie jest częścią znanej biblioteki Boost.

Pojemniki

Nadeszła pora, by poznać główny powód wprowadzenia do C++ mechanizmu szablonów. Są nim mianowicie **klasy kontenerowe**.

Kontenery albo **pojemniki** (ang. *containers*) to specjalne struktury danych przeznaczone do zarządzania kolekcjami obiektów tego samego typu w określony sposób.

Ponieważ definicja ta jest bardzo ogólna, mamy mnóstwo rodzajów kontenerów. Spora ich część została zaimplementowana w Bibliotece Standardowej, a o wszystkich mówi dowolna książka o algorytmach i strukturach danych.

Nie będziemy tutaj omawiać każdego rodzaju pojemnika, lecz skoncentrujemy się jedynie na tym, w jaki sposób szablony pomagają im w prawidłowym funkcjonowaniu. Zobaczymy więc najdonioślejsze zastosowanie szablonów w programowaniu.

Przykład klasy kontenera - stos

Zgodnie ze zwyczajem, kontenery poznamy na przykładzie jednego z prostszych rodzajów. Będzie to stos.

Czym jest stos

Pojęcie stosu jest ci znane; podczas omawiania wskaźników na funkcje wyjaśniłem bowiem, że jest to pomocny obszar pamięci, poprzez który odbywa się transfer argumentów od wywołującego funkcję.

Stos (ang. *stack*) ma też inne znaczenie. Jest to rodzaj pojemnika przechowującego dowolne elementy, charakteryzujący się tym, iż:

- obiekty są na stos jedynie **odkładane** (ang. *push*) i **pobierane** (ang. *pop*)
- w danej chwili ma się dostęp jedynie do ostatnio położonego, szczytowego elementu
- obiekty są zdejmowane w odwrotnej kolejności niż były odkładane na stos

Widać więc analogię do stosu - obszaru pamięci. Tam obiektami odkładanymi były parametry funkcji. Położone w jednej kolejności, musiały być następnie podejmowane w porządku odwrotnym. Cały czas równie dobre jest porównanie do stosu książek: jeśli położymy na biurku słownik ortograficzny, na nim książkę kucharską, a na samej górze podręcznik fizyki, to aby poznać prawidłową pisownię słowa 'gżegżółka' będziemy musieli wpierw zdjąć dwie książki leżące na słowniku. Przy czym najpierw pozbędziemy się podręcznika, a potem książki z przepisami.

Definicja szablonu klasy

Na tej samej zasadzie działa stos - struktura danych. Jest to coś w rodzaju tablicy, przechowującej obiekty dowolnego typu, będące odłożonymi na stos elementami. Nie pozwala ona jednak na pobranie dowolnego elementu (o ustalonym indeksie), lecz wymaga zdejmowania obiektów w kolejności odwrotnej do porządku ich odkładania.

Najlepszym sposobem na wprowadzenie stosu do programowania w C++ jest zdefiniowanie odpowiedniego szablonu klasy. Dzięki temu wszystkie szczegóły implementacji zostaną ukryte (zaleta OOPu), a nasz stos będzie potrafił operować elementami dowolnych typów (zaleta szablonów).
Spójrzmy więc na propozycję takiego szablonu stosu:

```
template <typename T, unsigned N> class TStack
{
   private:
         // zawartość stosu
         T m_aStos[N];

         // aktualny rozmiar (liczba elementów) stosu
         unsigned m_uRozmiar;

   public:
```

```
        // konstruktor
        TStack()    : m_uRozmiar(0)   { }

        //-------------------------------------------------------------

        // odłożenie elementu na stos
        void Push(const T& Element)
        {
              if (m_uRozmiar == N)
                    throw "TStack::Push() - stos jest pełen";

              m_aStos[m_uRozmiar] = Element;      // dodanie elementu
              ++m_uRozmiar;                       // zwiększ. licznika
        }

        // pobranie elementu ze szczytu stosu
        T Pop()
        {
              if (m_uRozmiar == 0)
                    throw "TStack::Pop() - stos jest pusty";

              // zwrócenie elementu i zmniejszenie licznika
              return m_aStos[--m_uRozmiar];
        }
};
```

Jest to właściwie najprostsza możliwa wersja stosu. Dwa parametry szablonu określają w niej typ przechowywanych elementów oraz maksymalną ich liczbę. Drugi oczywiście nie jest konieczny - łatwo wyobrazić sobie (i napisać) stos, który używa dynamicznej tablicy i dostosowuje się do liczby odłożonych elementów.
Co do metod, to ich garnitur jest również skromny. Metoda `Push()` powoduje odłożenie na stos podanej wartoci, zaś `Pop()` - pobranie jej i zwrócenie w wyniku. To absolutne minimum; często dodaje się do tego jeszcze funkcję `Top()` ('szczyt'), która zwraca element leżący na górze bez zdejmowania go ze stosu.

Klasę można też usprawniać dalej: dodając szablonowy kostruktor kopiujący i operator przypisania, metody zwracające aktualny rozmiar stosu (liczbę odłożonych elementów) i inne dodatki. Możnaby nawet zmienić wewnętrzny mechanizm funkcjonowania klasy i zaprząc do pracy szablon `TArray` - dzięki temu maksymalny rozmiar stosu mógłby być ustalany dynamicznie.
Zawsze jednak istota działania pojemnika będzie taka sama.

Korzystanie z szablonu

Spożytkowanie tak napisanego stosu nie jest trudne. Oto najbanalniejszy z banalnych przykładów:

```
// deklaracja obiektu stosu, zawierającego maksymalnie 5 liczb typu int
TStack<int, 5> Stos;

// odłożenie paru liczb na stos
Stos.Push (12);
Stos.Push (23);
Stos.Push (34);

// podjęcie i wyświetlenie odłożonych liczb
for (unsigned i = 0; i < 3; ++i)
   std::cout << Stos.Pop() << std::endl;
```

W jego rezultacie zobaczylibyśmy wypisanie liczb:

```
34
23
12
```

Widać zatem wyraźnie, że metoda `Pop()` powoduje zwrócenie elementów stosu w kolejności przeciwnej do ich odkładania poprzez `Push()`. Na tym właśnie opiera się idea stosu.

Stos ma w programowaniu rozliczne zastosowania: począwszy od rekurencyjnego przeszukiwania hierarchicznych baz danych (jak chociażby katalogi na dysku twardym) po rysowanie trójwymiarowych modeli w grach komputerowych. Obok zwykłej tablicy, jest to chyba najczęściej wykorzystywany pojemnik.

Programowanie ogólne

Szablony, a szczególnie ich użycie do implementacji kontenerów, stały się podstawą idei tak zwanego **programowania ogólnego** (ang. *general programming*). Trudno precyzyjnie ją wyrazić i zdefiniować, ale można ją rozumieć jako poszukiwanie jak najbardziej abstrakcyjnych i ogólnych rozwiązań w postaci algorytmów i struktur danych. Rozwiązania powstałe w zgodzie z tą ideą są więc niesłychanie elastyczne.

Dobrym przykładem są właśnie kontenery. Istnieje wiele ich rodzajów, począwszy od prostych tablic jednowymiarowych po złożone struktury, jak np. drzewa. Dla każdego pojemnika logiczne jest jednak przeprowadzanie pewnych typowych operacji, jak na przykład wyszukiwanie określonego elementu. Operacje te nazywami **algorytmami**. Logiczne byłoby zaprogramowanie algorytmów jako metod klas kontenerowych. Rozwiązanie to ma jednak wadę: ponieważ każdy pojemnik jest zorganizowany inaczej, należałoby dla każdego z nich zapisać osobną wersję algorytmu. Problem ten rozwiązano poprzez dodanie abstrakcyjnego pojęcia **iteratora** - obiektu, który służy do przeglądania kontenera. Iterator ukrywa wszelkie szczegóły związane z konkretnym pojemnikiem, przez co algorytm oparty na wykorzystaniu iteratorów może być napisany raz i wykorzystywany wielokrotnie w odniesieniu do dowolnych kontenerów.

Ten zmyślny pomysł stał się podstawą stworzenia **Standardowej Biblioteki Szablonów** (ang. *Standard Template Library* - STL). Jest to główna część Biblioteki Standardowej języka C++ i zawiera wiele szablonów podstawowych struktur danych. Są one wsparte algorytmami, iteratorami i innymi pomocniczymi pojęciami, dzięki którym STL jest nie tylko bogata funkcjonalnie, ale i efektywna oraz elastyczna. To jedno z bardziej użytecznych narzędzi języka C++ i jednocześnie najważniejsze zastosowanie szablonów.

Podsumowanie

Ten rozdział kończy kurs języka C++. Na ostatku zapoznałeś się z jego najbardziej zaawansowanym mechanizmem - szablonami.

Wpierw więc zobaczyłeś sytuacje, w których ścisła kontrola typów w C++ jest powodem problemów. Chwilę później otrzymałeś też do ręki lekarstwo, czyli właśnie szablony. Przeszliśmy potem do dokładnego omówienia ich dwóch rodzajów: szablonów funkcji i szablonów klas.
W sposób ogólniejszy zajęliśmy się nimi w następnym podrozdziale. Poznałeś zatem trzy rodzaje parametrów szablonów, które dają im razem bardzo potężne właściwości. Zaraz jednak uświadomiłem ci także problemy związane z szablonami: począwszy od konieczności udzielania podpowiedzi dla kompilatora co do znaczenia niektórych nazw, a kończąc na kwestii organizacji kodu szablonów w plikach źródłowych.

W trzecim podrozdziale przyjrzeliśmy się natomiast najbardziej typowym zastosowaniom szablonów - czyli dowiedzieliśmy się, jak zdobyta wiedza może się przydać w praktyce.

Tak kończy się opis języka C++ z punktu widzenia składni i semantyki. Jego częścią jest jednak także Biblioteka Standardowa. Niejednokrotnie mieliśmy okazję korzystać z jej drobnych części, lecz dopiero w następnym rozdziale rozpoczniemy jej systematyczne omawianie.

Pytania i zadania

Teraz czeka cię jeszcze tylko odpowiedź na kilka sprawdzających wiedzę pytań i wykonanie zadań. Powodzenia!

Pytania

1. Co to znaczy, że C++ jest językiem o ścisłej kontroli typów?
2. W jaki sposób można stworzyć „ogólne funkcje", działające dla wielu typów danych?
3. Jakie są sposoby na implementację ogólnych klas pojemnikowych bez użycia szablonów?
4. Jak definiujemy szablon?
5. Jakie rodzaje szablonów są dostępne w C++?
6. Czym jest specjalizacja szablonu? Czym się różni specjalizacja częściowa od pełnej?
7. Skąd kompilator bierze „wartości" (nazwy typów) dla parametrów szablonów funkcji?
8. Które parametry szablonu funkcji mogą być wydedukowane z jej wywołania?
9. Co dzieje się, gdy używamy szablonu funkcji lub klasy? Jakie zadania spoczywają wówczas na kompilatorze?
10. Jakie trzy rodzaje parametrów może posiadać szablon klasy?
11. Jaka jest rola słowa kluczowego `typename`? Gdzie i dlaczego jest ono konieczne?
12. Na czym polega model włączania?
13. Który sposób organizacji kodu szablonów najbardziej przypomina tradycyjną metodę podziału kodu w C++?
14. Dlaczego nie należy używać makrodefinicji w celu imitowania szablonów funkcji?
15. Czym jest krotka?
16. Co rozumiemy pod pojęciem pojemnika lub kontenera?

Ćwiczenia

1. Napisz szablon funkcji `Suma()`, obliczający sumę wartości elementów podanej tablicy `TArray`.
2. (**Trudniejsze**) Zdefiniuj szablon klas tablicy wskaźników o nazwie `TPtrArray`, dziedziczący z `TArray`. Szablon ten powinien przyjmować jeden parametr, będący typem, na który pokazują elementy tablicy.
3. (**Bardzo trudne**) Dodaj do specjalizacji `TArray<TArray<TYP> >` przeciążony operator `[]`, który będzie działał w ten sam sposób, jak dla zwykłych wielowymiarowych tablic języka C++.
   Wskazówka: operator ten będzie wobec tablicy używany dwukrotnie. Pomyśl więc, jaką wartość (obiekt tymczasowy) powinno zwracać jego pierwsze użycie, aby drugie zwróciło w wyniku żądany element tablicy.
4. (**Trudniejsze**) Opracuj i zaimplementuj algorytm dokonujący przedstawiania liczby naturalnej w systemie rzymskim.
   Wskazówka: wykorzystaj tablicę przeglądową par: litera rzymska plus odpowiadająca jej liczba dziesiętna.

5. Napisz szablon `TQueue`, podobny do `TStack`, lecz implementujący pojemnik zwany kolejką. Kolejką działa w ten sposób, iż elementy są dodawane do jej pierwszego końca, natomiast pobierane są z drugiego - tak samo, jak obsługiwane są osoby stojące w kolejce w sklepie czy banku. Podobnie jak w przypadku stosu, możesz określić jej maksymalny rozmiar jako parametr szablonu.