1 Szablony - definicja, zastosowania

Szablony są jedną z głównych cech C++. Dzięki nim możemy dostarczać generyczne typy i funkcje, bez kosztów czasu wykonania. Skupiają się na pisaniu kodu w sposób niezależny od konkretnego typu, dzięki czemu wspierają programowanie generyczne. Grają kluczową rolę w projektowaniu obecnych, znanych i popularnych bibliotek i systemów. Stanowią podstawę technik programowania w różnych dziedzinach, począwszy od konwencjonalnego programowania ogólnego przeznaczenia do oprogramowywania wbudowanych systemów bezpieczeństwa.

Szablon to coś w rodzaju przepisu, z którego translator C++ generuje deklaracje.

```
template<typename T>
T kwadrat (T x) {
        return x * x;
}
```

Kod ten deklaruje rodzinę funkcji indeksowanych po parametrze typu. Można odnieść się do konkretnego członka tej rodziny przez zastosowanie konstrukcji `kwadrat<int>`. Mówimy wtedy, że żądana jest specjalizacja szablonu dla funkcji `kwadrat` z listą argumentów szablonu `<int>`. Proces tworzenia specjalizacji nosi nazwę inicjalizacji szablonu, potocznie zwany inicjalizacją. Kompilator C++ stworzy odpowiedni odpowiednik definicji funkcji:

```
int kwadrat(int x) {
        return x * x;
}
```

Argument typu int jest podstawiony za parametr typu T. Kod wynikowy jest sprawdzany pod względem typu, by zapewnić brak błędów wynikających z podmiany. Inicjalizacja szablonu jest wykonywana tylko raz dla danej specyfikacji nawet jeśli program zawiera jej wielokrotne żądania.

W przeciwieństwie do języków takich jak Ada czy System F, lista argumentów szablonu może być pominięta z żądania inicjalizacji szablonu funkcji. Zazwyczaj, wartości parametrów szablonu są dedukowane.

```
double d = kwadrat(2.0);
```

Argument typu jest dedukowany na double. Warto zauważyć, że odmiennie niż w językach takich jak Haskell czy System F, parametry szablonu w C++ nie są ograniczone względem typów.

Szablonów używa się do zmniejszania kar abstrakcji i zjawiska *code bloat* w systemach wbudowanych w stopniu, który jest niepraktyczny w standardowych systemach obiektowych. Robi się to z dwóch powodów:

- Po pierwsze, inicjalizacja szablonu łączy informacje zarówno z definicji, jak i z kontekstu użycia. To oznacza, że pełna informacja zarówno z definicji jak i z wywołanych kontekstów (włączając w to informacje o typach) jest udostępniana generatorowi kodu. Dzisiejsze generatory kodu dobrze sobie radzą z używaniem tych informacji w celu zminimalizowania czasu wykonania i przestrzeni kodu. Różni się to od zwykłego przypadku w języku obiektowym, gdzie wywołujący i wywoływany są kompletnie oddzieleni przez interfejs, który zakłada pośrednie wywołania funkcji.

- Po drugie, szablon w C++ jest zazwyczaj domyślnie tworzony tylko jeśli jest używany w sposób niezbędny dla semantyki programu, automatycznie minimalizując miejsce w pamięci, które wykorzystuje aplikacja. W przeciwieństwie do języka Ada czy System F, gdzie programista musi wyraźnie zarządzać inicjalizacjami.

1.1 Parametryzacja szablonów

Parametry szablonu są specyfikowane na dwa sposoby:

1. *parametry szablonu* – wyraźnie wspomniane jako parametry w deklaracji szablonu

2. *nazwy zależne* - wywnioskowane z użycia parametrów w definicji szablonu

W C++ nazwa nie może być użyta bez wcześniejszej deklaracji. To wymaga od użytkownika ostrożnego traktowania definicji szablonów. Np. w definicji funkcji `kwadrat` nie ma widocznej deklaracji symbolu *. Jednak, podczas inicjalizacji szablonu `kwadrat<int>` kompilator może sprowadzić symbol * do (wbudowanego) operatora mnożenia dla wartości `int`. Dla wywołania `kwadrat(zespolona(2.0))`, operator * zostałby rozwiązany do (zdefiniowanego przez użytkownika) operatora mnożenia dla wartości `zespolona`. Symbol * jest więc *nazwą zależną* w definicji funkcji `kwadrat`. Oznacza to, że jest to ukryty parametr definicji szablonu. Możemy uczynić z operacji mnożenia formalny parametr:

```
template<typename Mul, typename T>
T square(T x) {
        return Mul() (x,x);
}
```

Pod-wyrażenie `Mul()` tworzy obiekt funkcji, który wprowadza operacje mnożenia wartości typu `T`. Pojęcie *nazw zależnych* pomaga utrzymać liczbę jawnych argumentów.

1.2 Inicjalizacje i sprawdzanie

Minimalne przetwarzanie semantyczne odbywa się, gdy po raz pierwszy widzi definicję szablonu lub jego użycie. Pełne przetwarzanie semantyczne jest przesuwane na czas inicjalizacji (tuż przed czasem linkowania), na podstawie każdej instancji. Oznacza to, że założenia dotyczące argumentów szablonu nie są sprawdzane przed czasem inicjalizacji. Np.

```
string x = "testowy tekst";
kwadrat(x);
```

Bezsensowne użycie string jako argumentu funkcji kwadrat nie jest wyłapane w momencie użycia. Dopiero w czasie inicjalizacji kompilator odkryje, że nie ma odpowiedniej deklaracji dla operatora *. To ogromny praktyczny błąd bo inicjalizacja może być przeprowadzona przez kod napisany przez użytkownika, który nie napisał definicji funkcji kwadrat ani definicji string. Programista, który nie znał definicji funkcji kwadrat ani string miałby ogromne trudności w zrozumieniu komunikatów błędów związanych z ich interakcją (np. "illegal operand for *").

Istnienie symbolu operatora * nie jest wystarczające by zapewnić pomyślną kompilację funkcji `kwadrat`. Musi istnieć operator *, który przyjmuje argumenty odpowiednich typów i ten operator * musi być bezkonkurencyjnym dopasowaniem według zasad przeciążania C++. Dodatkowo funkcja `kwadrat` przyjmuje argumenty przez wartość i zwraca swój wynik przez wartość. Z tego wynika, że musi być możliwe skopiowanie obiektów dedukowanego typu. Potrzebny jest rygorystyczny framework do opisywania wymagań definicji szablonów na ich argumentach.

Doświadczenie podpowiada nam, że pomyślna kompilacja i linkowanie może nie gwarantować końca problemów. Udana budowa pokazuje tylko, że inicjalizacje szablonów były poprawne pod względem typów, dostając argumenty które przekazaliśmy. Co z typami argumentów szablonu i wartościami, z którymi nie próbowaliśmy użyć naszych szablonów? Definicja szablonu może zawierać przypuszczenia na temat argumentów, które przekazaliśmy ale nie zadziała dla innych, prawdopodobnie rozsądnych argumentów. Uproszczona wersja klasycznego przykładu:

```
template<typename FwdIter>
bool czyJestPalindromem(FwdIter first, FwdIter last){
        if(last <= first) return true;
        if(*first != *last) return false;
        return czyJestPalindromem(++first, --last);
}
```

Testujemy czy sekwencja wyznaczona przez parę iteratorów do jego pierwszego i ostatniego elementu, jest palindromem. Przyjmuje się, że te iteratory są z kategorii *forward iterator*. To znaczy, że zakładają wspieranie co najmniej operacje: *, != i ++. Definicja funkcji `czyJestPalindromem` bada czy

elementy sekwencji zmierzają z początku i końca do środka. Możemy przetestować tę funkcję używając `vector`, tablicą stylu `C` i `string`. W każdym przypadku nasz szablon funkcji zainicjalizuje się i wykona się poprawnie. Niestety, umieszczenie tej funkcji w bibliotece byłoby dużym błędem. Nie wszystkie sekwencje wspierają – i ¡=. Np. listy pojedyncze nie wspierają. Eksperci używają wyszukanych, regularnych technik by uniknąć takich problemów. Jednakże, fundamentalny problem jest taki że definicja szablonu nie jest (według siebie) dobrą specyfikacją jego wymagań na jego parametry.