0.1 Wydajność

Szablony grają kluczową rolę w programowaniu w *C++* dla wydajnych aplikacji. Ta wydajność ma trzy źródła:

- eliminacja wywołań funkcji na korzyść *inliningu*
- łączenie informacji z różnych kontekstów w celu lepszej optymalizacji
- unikanie generowania kodu dla niewykorzystanych funkcji

Pierwszy punkt nie odnosi się tylko do szablonów ale ogólnie do cech funkcji *inline* w *C++*. Jakkolwiek, *inlining* jest istotny dla drobno-granularnej parametryzacji, którą powszechnie stosuje się w bibliotece *STL* i innych bibliotekach bazujących na generycznych technikach programowania. Wydajność ta przekłada się zarówno na czas wykonania jak i pamięć. Szablony mogą równocześnie zmniejszyć obie wydajności. Zmniejszenie rozmiaru kodu jest szczególnie ważne, ponieważ w przypadku nowoczesnych procesorów zmniejszenie rozmiaru kodu pociąga za sobą zmniejszenie ruchu w pamięci i poprawienie wydajności pamięci podręcznej.

```
template<typename FwdIter, typename T>
T suma(FwdIter first, FwdIter last, T init){
   for(FwdIter cur = first, cur != last, T init)
      init = init + *cur;
   return init;
}
```

Funkcja `suma` zwraca sumę elementów jej sekwencji wejściowej używając trzeciego argumentu ("akumulatora") jako wartości początkowej

```
vector<zespolona<double>> v;
zespolona<double> z = 0;
z = suma(v.begin(), v.end(), z);
```

By wykonać swoją pracę, `suma` użyje operatorów dodawania i przypisania na elementach typu `zespolona<double>` i dereferencji iteratorów `vector<zespolona<double>>`. Dodanie wartości typu `zespolona<double>` pociąga za sobą dodanie wartości typu `double`. By zrobić to wydajnie wszystkie te operacje muszą być *inline*. Zarówno `vector` jak i `zespolona` są typami zdefiniowanymi przez użytkownika. Oznacza to, że typy te jak i ich operacje są zdefiniowane gdzie indziej w kodzie źródłowym *C++*. Obecne kompilatory *C++* radzą sobie z tym przykładem, dzięki czemu jedyne wygenerowane wywołanie to wywołanie funkcji `suma`. Dostęp do pól zmiennej `vector` staje się prostą operacją maszyny ładującej, dodawanie wartości typu `zespolona` staje się dwiema instrukcjami maszyny dodającej dwa elementy zmiennoprzecinkowe. Aby to osiągnąć, kompilator potrzebuje dostępu do pełnej definicji

`vector` i `zespolona`. Jednak wynik jest ogromną poprawą (prawdopodobnie optymalną) w stosunku do naiwnego podejścia generowania wywołania funkcji dla każdego użycia operacji na parametrze szablonu. Oczywiście instrukcja dodawania wykonuje się znacznie szybciej niż wywołanie funkcji zawierającej dodawanie. Poza tym, nie ma żadnego wstępu wywołania funkcji, przekazywanie argumentów itd., więc kod wynikowy jest również wiele mniejszy. Dalsze zmniejszanie rozmiaru generowanego kodu uzyskuje się nie wysyłając kodu niewykorzystywanych funkcji. Klasa szablonu `vector` ma wiele funkcji, które nie są wykorzystywane w tym przykładzie. Podobnie szablon klasy `zespolona` ma wiele funkcji i funkcji nieskładowych (nienależących do funkcji klasy). Standard *C ++* gwarantuje, że nie jest emitowany żaden kod dla tych niewykorzystanych funkcji.

Aby kontrastować, rozważ bardziej konwencjonalny przypadek, w którym argumenty są dostępne za pośrednictwem interfejsów zdefiniowanych jako wywołania funkcji pośrednich. Każda operacja staje się wtedy wywołaniem funkcji w pliku wykonywalnym generowanym dla kodu użytkownika, takiego jak suma. Co więcej, byłoby wyraźnie nietypowe unikać odkładania kodu nieużywanych (wirtualnych) funkcji składowych. Jest to poza zdolnością obecnych kompilatorów *C++* i prawdopodobnie pozostanie takie dla głównych programów *C++*, gdzie oddzielna kompilacja i łączenie dynamiczne jest normą. Ten problem nie jest wyjątkowy dla *C++*. Opiera się on na podstawowej trudności w ocenieniu, która część kodu źródłowego jest używana, a która nie, gdy jakakolwiek forma wysyłki czasu wykonania ma miejsce. Szablony nie cierpią na ten problem bo ich specjalizacje są rozwiązywane w czasie kompilacji.

Przykład funkcji `suma` nie jest idealny do zilustrowania subtelności generowania kodu obiektu z kodu źródłowego znalezionego w różnych częściach programu. Nie polega na niejawnych konwersjach lub nietypowych przeciążaniach. Jednak, rozważ wariant gdzie wartości int są sumowane w obiekcie `zespolona<double>`:

```
vector<int> v;
zespolona<double> s = 0;
s = suma(v.begin(), v.end(), s);
```

Tu dodawanie jest wykonane przez konwertowanie wartości `int` do wartości `double` i potem dodawanie tego do akumulatora `s`, używając operatora + typu `zespolona<double>` i `double`. To podstawowe dodawanie zmiennoprzecinkowe. Kwestia jest taka, że operator + w funkcji `suma` zależy od dwóch parametrów szablonu i leży to w kwestii kompilatora by wybrać bardziej odpowiedni operator + bazując na informacji o tych dwóch argumentach. Byłoby możliwe utrzymanie lepszego rozdzielenia między różnymi kontekstami przez zawsze przekształcanie typu elementu w typ akumulatora. W takim przypadku spowodowałoby to powstanie dodatkowego

`zespolona<double>` dla każdego elementu i dodania dwóch wartości typu `zespolona`. Rozmiar kodu i czas wykonywania byłyby większe niż dwukrotnie.

Nie spodziewalibyśmy się zobaczyć tego ostatniego przykładu bezpośrednio w kodzie źródłowym. Gdybyśmy go zobaczyli, uznaliśmy, że jest on źle napisany. Jednakże, równoważny kod jest powszechny w wyniku zagnieżdżonych abstrakcji. Jest to szczególnie ważne by generować dobry kod w takich przypadkach ponieważ nie robienie tego byłoby zniechęcające dla abstrakcji.

Warto zauważyć, że te optymalizacje są wspólnym miejscem. Duże ilości prawdziwego oprogramowania zależą od nich. W konsekwencji udoskonalone sprawdzanie typu, co zostało obiecane przy użyciu konceptów, nie może kosztować tych optymalizacji.