Wydajność użycia funktorów z biblioteką STL języka C++

Irek Szcześniak[*], Maciek Sobczak[†]

1 października 2005 roku

Streszczenie

Artykuł dotyczy wydajności użycia funktorów z biblioteką STL (Standard Template Library) języka C++. Biblioteka STL oferuje struktury danych takie jak wektory, zbiory czy kolejki priorytetowe i wiele funkcji do operowania na strukturach, np. funkcję sortującą. Strukturom danych, takim jak `std::priority_queue`, oraz funkcjom, takim jak `std::sort` czy `std::transform`, oprócz danych można przekazać też funktor, który definiuje sposób operowania na danych. Dla trzech implementacji biblioteki STL dostarczanych z kompilatorami GNU, SGI i SUN przedstawiamy wyniki świadczące o tym, że pewne elementy biblioteki wykonują wiele kopii funktora (np. średnio 1330 kopii przy sortowaniu wektora o długości 1000 elementów). Co ważniejsze, pokażemy też na przykładzie, że źle zaprojektowany funktor w użyciu z biblioteką STL może bardzo spowolnić działanie programu. Na koniec artykułu pokażemy, jak wydajnie używać funktorów z biblioteką STL. Waga przedstawionego problemu wynika z faktu, że biblioteka STL jest częścią standardowej biblioteki języka C++ i sprzyja tworzeniu przenośnych programów – programiści korzystający z języka C++ mogą znaleźć w tym artykule wskazówki przydatne w ich własnej pracy.

1 Wprowadzenie

Język C++ wyrósł z języka C poprzez dodanie nowych możliwości, takich jak przeciążanie funkcji, programowanie obiektowe, czy szablony. Język C++ i jego biblioteka standardowa ciągle się rozwiją. Częścią biblioteki standardowej C++ jest Standard Template Library (STL) [1], której dotyczy ten artykuł.

Biblioteka STL jest przykładem programowania generycznego (ang. generic programming), które polega na implementacji rozwiązania postawionego problemu w taki sposób, że to rozwiązanie zależy jak najmniej od szczegółów technicznych postawionego problemu, takich jak zastosowanych algorytmów, oraz typów i struktury danych.

Biblioteka STL jest generyczna pod trzema względami. Po pierwsze, algorytmy są niezależne od struktur danych, na których mają operować, czyli temu samemu algorytmowi (funkcji biblioteki STL) możemy przekazać zarówno wektor (`std::vector`), jak i np. listę (`std::list`). Rozdzielenie algorytmów od struktur danych uzyskano dzięki zastosowaniu iteratorów, które zawsze stanowią element pośredni pomiędzy strukturą danych a algorytmem. Po drugie, struktury danych możemy budować z dowolnych typów danych, takich jak liczba całkowita albo zdefiniowana przez nas klasa, czyli możemy zadeklarować obiekt typu `std::vector<int>` albo typu `std::vector<moj_typ>`. Po trzecie, algorytmy i stuktury danych biblioteki STL możemy

[*]Instytut Informatyki Teoretycznej i Stosowanej Polskiej Akademii Nauk
[†]niezależny konsultant

rozszerzać przez dostarczenie własnych funktorów, które określają sposób operowania na danych. Przez funktor możemy np. określić sposób porównywania elementów podczas sortowania, porządek w jakim ma być utrzymana kolejka priorytetowa (`std::priority_queue`), działanie jakie ma być wykonane podczas transformacji sekwencji (`std::transform`), itp. Ważną zasadą obowiązującą w bibliotece STL jest to, że w dużej części jest ona zbiorem konwencji składniowych a nie interfejsów w tradycyjnym sensie klas bazowych, dziedziczenia i funkcji wirtualnych. Różne elementy STL współpracują ze sobą (oraz z elementami dostarczonymi przez programistę) dzięki przestrzeganiu tych konwencji, chociaż nie muszą być ze sobą formalnie powiązane.

Niniejszy artykuł skupia się na funktorach zastosowanych do rozszerzania biblioteki STL oraz na aspektach związanych z wydajnością funktorów.

2 Co to jest funktor?

Funktor jest to obiekt ze zdefiniowanym operatorem wywołania fukncji. Funktor nazywany jest także obiektem funkcyjnym. Na przykład, poniżej użyty obiekt `rd` jest funktorem, ponieważ możemy wykonać `rd(20)`.

```
#include <iostream>

struct razy_dwa
{
  int operator()(int i) { return 2 * i; }
};

int main()
{
  razy_dwa rd;
  std::cout << rd(102) << '\n';
}
```

Oczywiście ten sam efekt uzyskamy przez użycie zwykłej funkcji typu `int rd(int i)`. Jednak przewagą funktorów nad funkcją jest to, że każdy funktor może mieć "własne" dane, czyli stan, przechowywany pomiędzy wywołaniami, co jest ich zaletą w stosunku do funkcji korzystających ze zmiennych globalnych albo statycznych. W przykładzie niżej funktor `rn` zawiera w sobie informację (mnożnik), którą przechowuje i z której korzysta, gdy jest wywołany.

```
#include <iostream>

class razy_n
{
  int n;

  public:
  razy_n(int arg) : n(arg) {}
  int operator()(int i) { return n * i; }
};

int main()
{
  razy_n rn(3);
  std::cout << rn(102) << '\n';
}
```

Ale i tę cechę funktora możemy zrealizować przez wywoływanie dowolnej innej funkcji składowej (np. nazwanej `wynik`), niekoniecznie operatora wywołania funkcji, tak jak to jest przedstawione niżej.

```
#include <iostream>

class razy_dwa
{
  public:
  int wynik(int i) { return 2 * i; }
};

int main()
{
  razy_dwa rd;
  std::cout << rd.wynik(102) << '\n';
}
```

Prawdziwe zastosowanie funktora widać dopiero w połączeniu z szablonami, gdzie możemy zamiennie używać funktorów i funkcji, ponieważ użycie funktora jest składniowo zgodne z wywołaniem funkcji. Zdefiniujemy funkcję `suma`, która będzie zwrać sumę liczb zwracanych przez funktor. Funktorowi przekażemy wszystkie liczby całkowite od 1 do 99. Naszej funkcji `suma` możemy teraz przekazać nie tylko funktor, ale także zwykłą funkcję, jak pokazane jest niżej.

```
#include <iostream>

// zwykla funkcja
int funkcja(int i) { return 2 * i; }

// klasa funktora
class razy_n
{
  int n;

  public:
  razy_n(int arg) : n(arg) {}
  int operator()(int i) { return n * i; }
};

// szablon funkcji sumujacej
template<typename T>
int suma(T funk)
{
  int suma = 0;
  for(int i = 100; --i;)
    suma += funk(i);
  return suma;
}

int main()
{
  razy_n funktor(2);
  std::cout << suma(funktor) << '\n';
  std::cout << suma(funkcja) << '\n';
}
```

Poniższy przykład pokazuje, że funktory mogą również modyfikować swój stan w trakcie pracy, niezależnie od innych funktorów. Program ten wykorzystuje jedną definicję funktora do obliczenia średnich wartości dwóch różnych tablic. Średnia jest obliczana dla dwóch tablic równocześnie, ale każdy z funktorów operuje na swoich własnych danych, nie wpływając na siebie nawzajem.

```
#include <iostream>

```

```
class srednia
{
public:
     srednia() : suma(0), liczba(0) {}

     void operator()(int i)
     {
          suma += i;
          ++liczba;
     }

     double wynik() { return (double)suma / liczba; }

private:
     int suma;
     int liczba;
};

int main()
{
     int tab1[] = {1, 4, 2, 3, 6};
     int tab2[] = {7, 4, 6, 9, 3};

     srednia sr1;
     srednia sr2;

     for (int i = 0; i != 5; ++i)
     {
          sr1(tab1[i]);
          sr2(tab2[i]);
     }

     std::cout << "srednia␣z␣tab1:␣" << sr1.wynik() << '\n';
     std::cout << "srednia␣z␣tab2:␣" << sr2.wynik() << '\n';
}
```

2.1 Konwencja

Jak zostało już wspomniane na początku artykułu, wiele elementów biblioteki STL może ze sobą współpracować dzięki przestrzeganiu pewnych konwencji. Jedną z konwencji wprowadzonych przez standard C++ [2] jest zalecenie, aby funktor zawierał publiczne definicje typów argumentów i zwracanej wartości. Dla funktora unarnego (akceptującego jeden argument) tymi definicjami są:

```
typedef ... argument_type;
typedef ... result_type;
```

Natomiast dla funktora binarnego standard wymaga następujących definicji:

```
typedef ... first_argument_type;
typedef ... second_argument_type;
typedef ... result_type;
```

Dzięki przestrzeganiu tej konwencji, można implementować generyczne funktory i algorytmy, które składają ze sobą kilka funktorów, albo adaptują je do innych potrzeb. W dalszej części artykułu będziemy przestrzegać tej konwencji.

3 Przykład i pułapka

Naszym zadaniem jest odpowiednie uporządkowanie liczb całkowitych zapisanych w wektorze o nazwie `produkty` (typu `std::vector<int>`). Każda liczba tego wektora jest numerem oferowanego towaru. Do dyspozycji mamy tablicę asocjacyjną o nazwie `sprzedanych` (typu `std::map<int, int>`), w której kluczem jest numer towaru, a wartością jest liczba sprzedanych sztuk. Brak w tablicy jakiegoś numeru towaru oznacza, że nie sprzedano tego towaru.

Odpowiednie uporządkowanie wektora `produkty` polega na przemieszczeniu numerów częściej kupowanych towarów w kierunku końca wektora. Zatem jeżeli i, j, $i < j$ są indeksami uporządkowanego wektora, to:

$$\texttt{sprzedanych[produkty[i]]} \leqslant \texttt{sprzedanych[produkty[j]]}.$$

Do sortowania użyjemy funkcji `std::sort`, której przekażemy zdefiniowany przez nas funktor. Uporządkowany wektor zawiera liczby w tej kolejości: 10, 20, 5, 13, bo towaru numer 10 nie sprzedano wcale, a towaru numer 13 sprzedano największą liczbę sztuk.

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <map>
#include <vector>

// klasa funktora
class komparator
{
  std::map<int, int> sprzedanych;

public:

  typedef int  first_argument_type;
  typedef int  second_argument_type;
  typedef bool result_type;

  komparator(const std::map<int, int> &arg):
    sprzedanych(arg) {}

  bool operator()(const int a, const int b)
  {
    return sprzedanych[a] < sprzedanych[b];
  }
};

int main()
{
  std::vector<int> wektor;
  wektor.push_back(13);
  wektor.push_back(20);
  wektor.push_back(5);
  wektor.push_back(10);

  std::map<int, int> sprzedanych;
  sprzedanych[5] = 651;
  sprzedanych[20] = 99;
  sprzedanych[13] = 2148;

  komparator porzadek(sprzedanych);

  std::sort(wektor.begin(), wektor.end(), porzadek);
```

```
  std::copy(wektor.begin(), wektor.end(),
            std::ostream_iterator<int>(std::cout, "\n"));
}
```

Zakłada się, że funktory przeznaczone do użycia z biblioteką STL można kopiować (wywoływać konstruktor kopiujący) i przypisywać im nowe wartości (wywoływać operator przypisania). Co ważniejsze, zakłada się, że wykonanie tych operacji nie jest czasochłonne. Programista powinien pamiętać o tych założeniach.

Zadanie w przykładzie wyżej niestety nie zostało wykonane wydajnie, ponieważ funktor był wielokrotnie kopiowany a razem z nim jego tablica asocjacyjna. W tym przykładzie wydajność nie ma szczególnego znaczenia, bo porządkowany wektor jest bardzo mały. Jednak kiedy zadania są duże i często wykonywane, to trzeba zadbać o ich wydajność.

Standard języka C++ pozostawia dużą swobodę implementacjom biblioteki STL, w kontekście korzystania z funktorów – zarówno własnych, jak i tych dostarczonych przez użytkownika. Zakłada się jedynie, że funktory są przekazywane przez wartość. Oznacza to, że są kopiowane. To kopiowanie może odbywać się wielokrotnie, zwłaszcza w algorytmach rekurencyjnych. Im bardziej złożony jest funktor (zawiera dużo danych, ma skomplikowany konstruktor i destruktor), tym wolnieszy będzie nasz program. Jak bardzo ucierpi na tym wydajność programu zależy od tego, jak dużo kopii funktora wykona STL, oraz jaki jest koszt tego kopiowania w stosunku do pozostałych kosztów wykonywanego algorytmu. Następny podrozdział przedstawia konkretne testy wydajnościowe.

4 Wyniki

Przedstawimy wyniki dla trzech implementacji funkcji `std::sort` i struktury danych `std::priority_queue` dostarczanych z kompilatorami GNU GCC v. 3.3.2, SGI MIPSpro C++ v. 7.3.1.3m i SUN WorkShop C++ v. 5.0, do których będziemy się odnosić w dalszej części artykułu jako przypadki GNU, SGI i SUN. Prezentowane wyniki to liczby wykonanych kopii funktora (liczby wywołań konstruktora kopiującego i operatora przypisania).

Funkcję `std::sort` i strukturę `std::priority_queue` wybraliśmy jako przykłady tych elementów biblioteki STL, których implementacje są często rekurencyjne, co prowadzi do wielokrotnego kopiowania funktora, zależnie od ilości przetwarzanych danych. Istnieją również takie elementy biblioteki STL, gdzie funktory są kopiowane, ale liczba wykonanych kopii funktora nie zależy od ilości danych. Na przykład, funkcja `std::for_each` (przypadek GNU) kopiuje funktor dwa razy niezależnie od liczby elementów w strukturze, natomiast struktura danych `std::set` (przypadek GNU) kopiuje funktor tylko raz, niezależnie od liczby dodanych elementów do zbioru.

4.1 Wyniki dla funkcji `std::sort`

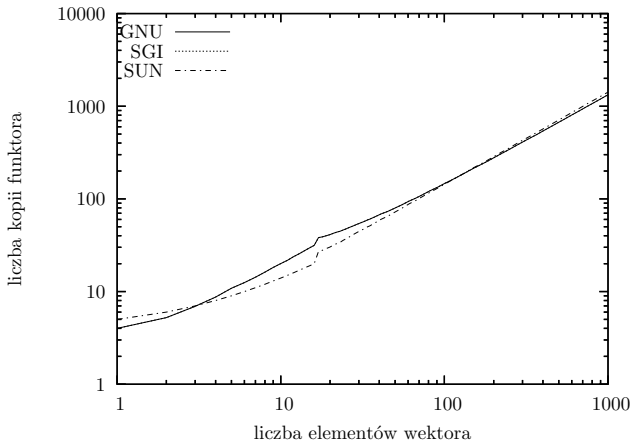
Przy użyciu funkcji `std::sort` sortujemy wektor liczb całkowitych (`std::vector<unsigned long>`) o długościach od jednego do tysiąca liczb. Liczby w wektorze są pseudolosowe, a do wygenerowania ich nie używamy generatora systemowego, ale naszego prostego generatora, aby mieć pewność, że do testów różnych implementacji użyte są te same liczby. Generujemy liczby trzydziesto dwu bitowe typu `unsigned long` według prostego wzoru ([3], strona 284):

```
idum = 1664525L * idum + 1013904223L.
```

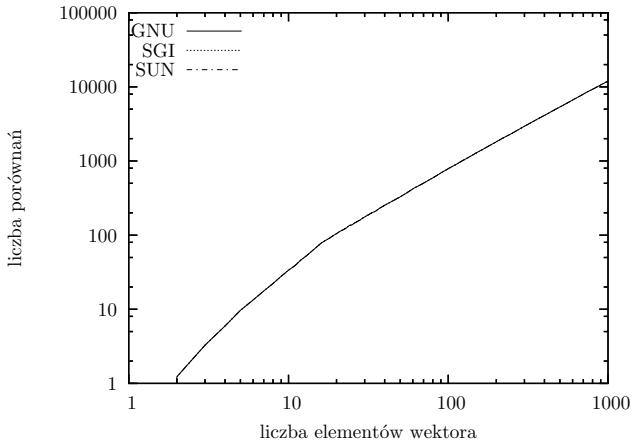
Przy każdej operacji sortowania biblioteka STL wykonuje pewną liczbę kopii naszego funktora, którą jesteśmy zainteresowani jako funkcją długości sortowanego wektora. Program testowy

pobiera z wejścia standardowego dwie liczby: długość wektora i wartość zarodka (pierwsza wartości zmiennej `idum`), a na standardowe wyjście wypisuje liczbę wykonanych kopii funktora i liczbę wywołań funktora. Dodatkowo, zliczamy również, ile razy funktor został faktycznie wykorzystany do porównywania danych. Program testowy został przedstawiony poniżej.

```
#include <algorithm>
#include <iostream>
#include <vector>

// klasa funktora
struct komparator
{
  static int liczba_kopii;
  static int liczba_porownan;

  typedef unsigned long first_argument_type;
  typedef unsigned long second_argument_type;
  typedef bool          result_type;

  komparator() {}
  komparator(const komparator &)
  {
    ++liczba_kopii;
  }

  komparator &operator=(const komparator &)
  {
    ++liczba_kopii;
    return *this;
  }

  bool operator()(const unsigned long a, const unsigned long b)
  {
    ++liczba_porownan;
    return a < b;
  }
};

int komparator::liczba_kopii = 0;
int komparator::liczba_porownan = 0;

int main()
{
  int dlugosc;
  unsigned long idum;

  std::cin >> dlugosc;
  std::cin >> idum;

  std::vector<unsigned long> wektor(dlugosc);

  while(dlugosc--)
    wektor[dlugosc] = idum = 1664525L * idum + 1013904223L;

  komparator porzadek;
  std::sort(wektor.begin(), wektor.end(), porzadek);

  std::cout << porzadek.liczba_kopii << '\n';
  std::cout << porzadek.liczba_porownan << '\n';
}
```



Rysunek 1: Liczba wykonanych kopii funktora przez funkcję `std::sort` w zależności od długości wektora.



Rysunek 2: Liczba wywołań funktora przez funkcję `std::sort` w zależności od długości wektora.

Liczba kopii zależy od wartości zarodka i dlatego sortowanie dla jednej długości wektora wykonujemy wielokrotnie dla wartości zarodka od 1 do 100, a następnie liczbę kopii uśredniamy. Wielokrotne wykonanie programu dla różnych długości wektora i wartości zarodka wykonuje już osobny skrypt, który nie jest tutaj przedstawiony.

Rysunek 1 jest dowodem na to, że źle zaprojektowany funktor spowolni sortowanie, bo będzie on wiele razy kopiowany. Rysunek przedstawia wyniki dla przypadków GNU, SGI i SUN. Jak widać dla wszystkich przypadków zależność między długością wektora i liczbą kopii jest liniowa. Rysunek 2 przedstawia liczbę wywołań funktora (liczbę porównań elementów) w funkcji długości wektora.

Z przedstawionych wykresów widać też, w jaki sposób różnice w implementacji danego algorytmu wpływają na złożoność obliczeń – nie tylko w sensie tradycyjnie rozpatrywanych porównań lub przemieszczeń danych w tablicy, ale również w sensie operacji kopiowania funktora.

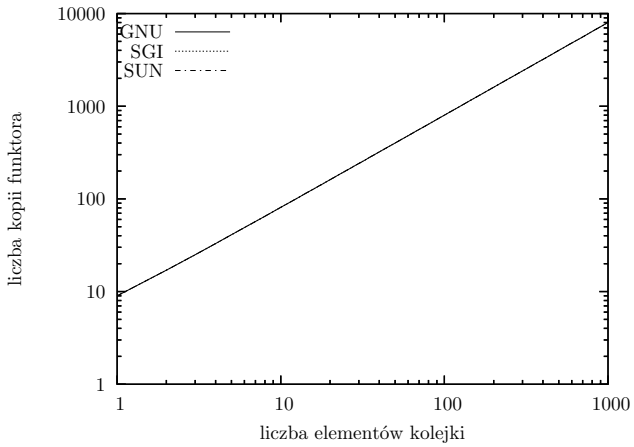
4.2 Wyniki dla struktury danych `std::priority_queue`

Niektóre struktury danych STL utrzymują pewien ustalony porządek, np. w celu uzyskania szybkiego dostępu do ich elementów. Do utrzymania porządku wykorzystywany jest funktor, który może być dostarczony przez programistę. Jedną z takich struktur jest `std::priority_queue`, którą teraz przetestujemy pod względem liczby kopii funktora.

Zadaniem programu testowego jest utworzenie kolejki priorytetowej z porządkiem zdefinio-

wanym przez nas, następnie umieszczeniu określonej liczby elementów do kolejki, a na sam koniec usunięciu wszystkich elementów. W całym procesie biblioteka STL wykona dużą liczbę kopii naszego funktora, a my jesteśmy zainteresowani ile tych kopii będzie. Ten program testowy zachowuje się tak, jak poprzedni program testowy (z tą różnicą, że wypisuje tylko liczbę kopii funktora) i jest wielokrotnie uruchamiany w ten sam sposób. Program testowy znajduje się niżej.

```
#include <iostream>
#include <queue>

// klasa funktora
struct komparator
{
  static int liczba_kopii;

  typedef unsigned long first_argument_type;
  typedef unsigned long second_argument_type;
  typedef bool          result_type;

  komparator() {}
  komparator(const komparator &)
  {
    liczba_kopii++;
  }

  komparator &operator=(const komparator &)
  {
    liczba_kopii++;
    return *this;
  }

  bool operator()(const unsigned long a, const unsigned long b)
  {
    return a < b;
  }
};

int komparator::liczba_kopii = 0;

int main()
{
  int dlugosc;
  unsigned long idum;

  std::cin >> dlugosc;
  std::cin >> idum;

  komparator porzadek;

  std::priority_queue<unsigned long,
    std::vector<unsigned long>, komparator> kolejka(porzadek);

  while(dlugosc--)
    kolejka.push(idum = 1664525L * idum + 1013904223L);

  while(!kolejka.empty())
    kolejka.pop();

  std::cout << porzadek.liczba_kopii << '\n';
}
```



Rysunek 3: Liczba kopii funktora wykonanych podczas operacji na kolejce `std::priority_queue` w zależności od maksymalnej długości kolejki.

Jak widać na rysunku 3 funktor jest kopiowany tysiące razy. Wyniki dla przypadków GNU, SGI i SUN są identyczne.

5 Rozwiązanie problemu

Niniejszy artykuł nie byłby kompletny bez wskazania na koszt kopiowania funktorów w kontekście całego kosztu wykonania danego algorytmu. Z wyników przedstawionych wcześniej widać, że liczba porównań elementów, wykonanych przy użyciu funktora, jest wielokrotnie większa (np. kilkadziesiąt razy), niż liczba wykonanych kopii funktora. Oznacza to, że pisząc wydajny program z użyciem STL należy pamiętać o kosztach związanych z kopiowaniem funktora, ale nie należy też liczyć na to, że zmniejszenie czasu trwania pojedynczego kopiowania funktora automatycznie spowoduje, że cały program wielokrotnie zyska na wydajności. Pewne problemy obliczeniowe wymagają dużej ilości pracy, przy której sam funktor nie zawsze jest elementem krytycznym. Oczywiście, źle napisany funktor może spowodować problemy wydajnościowe, zwłaszcza jeśli koszt jego obsługi jest nieproporcjonalnie wysoki. Jeżeli koszt kopiowania funktora zaczyna mieć wpływ na wydajność całego algorytmu, to należy podjąć specjalne kroki aby temu zapobiec. Na szczęście, w większości wypadków wystarczy trzymać się intuicyjnych reguł, stosowanych również w innych miejscach programu. Na przykład, zamiast przechowywać w obiekcie funkcyjnym duże ilości danych, wystarczy sam wskaźnik na te dane, które będą już wtedy istniały w jednym "egzemplarzu", niezależnie od liczby kopii funktora – rozmiar funktora zawierającego tylko wskaźnik jest bardzo mały i przy testowanych kompilatorach nie przekroczy rozmiaru samego wskaźnika. Przykład z podrozdziału 3 można zmienić w ten sposób, że funktor będzie zawierał tylko wskaźnik do tablicy asocjacyjnej, zamiast całą tablicę. W ten sposób kopiowanie funktora będzie polegać jedynie na kopiowaniu jednego wskaźnika.

Czasami zdarzają się też sytuacje, kiedy mamy do dyspozycji klasę funktora, ale z różnych powodów nie możemy jej zmodyfikować. Jeżeli zachodzi wtedy potrzeba zminimalizowania kosztów kopiowania tego funktora, można posłużyć się prostym adapterem, który przejmie na siebie obowiązki funktora, delegując każde wywołanie do właściwego, "ciężkiego" funktora. Sam adapter, przechowując jedynie wskaźnik do właściwego funktora, będzie bardzo tani w kopiowaniu. Przykład takiego adaptera oraz pomocniczej funkcji do jego tworzenia pokazano poniżej (zakłada się, że właściwy funktor przestrzega konwencji dotyczących publicznie definiowanych typów, o których była mowa wcześniej):

```
template <class Funktor>
```

```
struct lekki_adapter_t
{
   typedef typename Funktor::first_argument_type  first_argument_type;
   typedef typename Funktor::second_argument_type second_argument_type;
   typedef typename Funktor::result_type          result_type;

   lekki_adapter_t(Funktor &f) : pf(&f) {}

   result_type operator()(first_argument_type a, second_argument_type b)
   {
     return (*pf)(a, b);
   }

   Funktor *pf;
};

template <class Funktor>
lekki_adapter_t<Funktor> lekki_adapter(Funktor &f)
{
  return lekki_adapter_t<Funktor>(f);
}
```

Wykorzystanie tego adaptera pokazuje następujący przykład:

```
std::sort(wektor.begin(), wektor.end(),
    lekki_adapter(porzadek));
```

Należy jednak zauważyć, że taki adapter, chociaż bardzo tani w kopiowaniu, wprowadza dodatkowy koszt związany z wywołaniem pośrednim właściwego funktora przez wskaźnik. Prawie na pewno spowoduje to, że wywołania do oryginalnego funktora nie będą rozwijane inline, nawet jeśli były tak zdefiniowane. Z tego względu, trudno jest podać ogólne stwierdzenie, czy taki adapter będzie zawsze korzystny – to zależy od tego, jak kosztowne było kopiowanie pierwotnego funktora. Przed podjęciem decyzji o zastosowaniu powyższej techniki, zachęcamy programistów do samodzielnych testów wydajnościowych z różnymi wersjami – w tych miejscach programu, które są "podejrzane" o stwarzanie problemów wydajnościowych.

6 Podsumowanie

Przy projektowaniu funktorów, które mają być przekazywane algorytmom i strukturom biblioteki STL należy pamiętać o tym, że funktory mogą być często kopiowane, ponieważ projektanci biblioteki STL założyli, że kopiowanie funktora nie jest czasochłonne.

Pokazaliśmy, że STL może wykonać tysiące kopii funktora. W przypadku gdy kopiowanie funktora jest bardzo czasochłonne, wykonanie tak wielu kopii z pewnością bardzo spowolni wykonanie programu. Musimy zatem zaprojektować funktor tak, żeby jego kopiowanie było szybkie.

Uwaga

Wszystkie funktory pokazane w tym artykule były pisane bez użycia kwalifikatora `const` przy operatorze wywołania. Jeżeli dany funktor nie zawiera własnych danych lub operowanie na nim nie wiąże się z modyfikowaniem jego stanu, warto rozważyć dodanie kwalifikatora `const` przy operatorze wywołania.

Podziękowanie

Irek dziekuje wielu osobom za rady udzielone na łamach grup dyskusyjnych `comp.lang.c++.moderated` i `comp.lang.c++`, które pomogły mu przy pisaniu tego artykułu.

Literatura

[1] The Standard Template Library. http://www.sgi.com/tech/stl/.

[2] International Standard, Programming Languages – C++. ISO/IEC: 14882, 2003.

[3] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1992.