Michał Andrzejewski
Aleksy Barcz

31.05.2010
Warszawa

StlContainers

Dokumentacja końcowa

Wydział Elektroniki i Technik Informacyjnych - Politechnika Warszawska
Projekt z przedmiotu (Średnio)zaawansowane Programowanie w C++

1. Skład zespołu

— Michał Andrzejewski - kontrola jakości, dokumentacja, programista, tester
— Aleksy Barcz - właściciel projektu, koordynator, programista, tester

2. Cel projektu

Biblioteka StlContainers ma na celu:

- umożliwienie programiście wykorzystania kontenerów biblioteki standardowej C++:
  - std::vector
  - std::map
  - std::set

  w kodzie napisanym w Pythonie stosując typowe dla Pythona metody, takie jak *append(), len* itp.

- dostarczenie narzędzi do łatwej konwersji pomiędzy kontenerami Pythona a kontenerami STL:
  - std::vector <-> Python list
  - std::map <-> Python dict
  - std::set <-> Python list

Cele te realizowane są przy wykorzystaniu istniejących mechanizmów biblioteki boost::python i rozwiązań własnych.

3. Udostępnienie kontenerów STL w Pythonie - zestaw wrapperów

Do udostępnienia pod Pythonem kontenerów STL wykorzystany został mechanizm *boost::python::class_*, pozwalający opakowywać klasy C++ oraz stworzony zestaw szablonów:

- VectorWrapper
- MapWrapper
- SetWrapper

Każdy z tych szablonów pozwala w mocno zautomatyzowany sposób opakować pożądaną konkretyzację kontenera dla danego typu/zestawu typów.
Konkretyzacje dla kilku typów wbudowanych C++ oraz dla std::string zawiera biblioteka StlContainers, co pozwala użytkownikowi korzystać z nich bez znajomości mechanizmów opakowywania poprzez zwykły import. Do konkretyzacji zostały wybrane następujące kontenery i typy:

- *vector<int>, vector<double>* oraz *vector<std::string>*
- *set<int>, set<double>* oraz *set<std::string>*
- *map<int, std::string>* oraz *map<std::string, double>*

Dla ww. kontenerów zostały napisane testy prezentujące ich działanie. Nic nie stoi jednak na przeszkodzie aby do *StlContainers.cpp* dodać inne konkretyzacje typów w celu ułatwienia korzystania z nich w Pythonie. Dzięki zastosowaniu opisanego wyżej mechanizmu możliwe jest napisanie poniższego kodu:

```
import StlContainers
vector = StlContainers.vector_int()
vector.append(43)
print vector.__len__()          #len == 1
vector.print_()                 #wypisuje zawartość wektora na ekran
                                #w konwencji Python'owego list
```

Dla ilustracji sposobu udostępniania kontenera elementów własnego typu (bez konkretyzacji w bibliotece) posłużymy się przykładem:

Mając klasę *Foo*, której kolekcję, np. *std::vector<Foo>* chcemy udostępnić pod Pythonem, musimy:

1. W zależności od tego, czy klasa Foo() oferuje operatory porównania i/lub może być wypisywana na ekran poprzez *std::cout << foo_instance;* należy ustawić właściwe parametry opakowywania, przed załączeniem plików nagłówkowych dostarczanych przez bibliotekę.
   Domyślny zestaw parametrów to: brak obsługi operatora porównania oraz brak obsługi wypisywania na ekran.
   Jeśli jednak kolekcja elementów typu Foo() ma udostępniać pod Pythonem zestaw operacji zależnych od ww. parametrów, należy odpowiednio zdefiniować następujące stałe:

   ```
   #define STL_TO_PYTHON_CONTAINERS_COMPARABLE
   #define STL_TO_PYTHON_CONTAINERS_PRINTABLE
   ```

2. Załączyć plik nagłówkowy odpowiedniego wrappera (VectorWrapper, SetWrapper lub MapWrapper) oraz (w obrębie modułu *boost::python)* dokonać opakowania, podając nazwę dla widocznego pod Pythonem typu kolekcji.

   Poniższy przykładowy kod pokazuje jak opakować *std::vector<Foo>* jako *vector_foo*, zawarty w nowym w module Foo:

   ```
   #define STL_TO_PYTHON_CONTAINERS_COMPARABLE     //jeśli Foo porównywalna
   #define STL_TO_PYTHON_CONTAINERS_PRINTABLE      //jeśli Foo wypisywalna
   #include <boost/python/module.hpp>
   #include "VectorWrapper_py.hpp"                 //wrapper dla std::vector
   #include <Foo.hpp>                              //zewnętrzna definicja klasy Foo
   BOOST_PYTHON_MODULE(Foo) {
       /* tu opakowanie klasy Foo, aby można było nią manipulować pod Pythonem */

       StlContainersWrappers::VectorWrapper<Foo>::wrap("vector_foo");
       //opakowanie wektora Foo
       }
   ```

   Ograniczeniem wynikającym z wykorzystania kompilacji warunkowej dla rozróżnienia kolekcji elementów o różnych właściwościach jest fakt, że w obrębie jednego modułu mogą się znaleźć wyłącznie kolekcje elementów klas o tym samym zestawie właściwości (porównywalna, wypisywalna).

3. Koniec. Kolekcja elementów naszej klasy Foo będzie widoczna po Pythonem po zaimportowaniu do niego modułu Foo, udostępniając interfejs analogiczny do opowiadającego typu kolekcji Pythona.

Mechanizm ten, oprócz wykorzystania szybkości działania kontenerów STL pod Pythonem umożliwia również przekazywanie wypełnionych kolekcji pomiędzy Pythonem a C++.
Jeśli pomimo wygodnego interfejsu wrappera, użytkownik będzie potrzebował przenieść uzyskane w ten sposób dane do kolekcji Pythona, może wykorzystać wygodne metody:

- VectorWrapper::get_list()
- MapWrapper::get_dict()
- SetWrapper::get_list()

z których dwie pierwsze zwracają gotowe odpowiednie kolekcje pythonowe, natomiast ostatnia zwraca listę, którą można jednym poleceniem przerobić na pythonowy set.

4. Konwersje pomiędzy Pythonem a STL – konwertery

W celu konwersji kolekcji pythonowych na kolekcje STL, udostępnione zostały jawnie wywoływane konwertery:

- list -> std::vector : PyListToVector
- list -> std::set : PyListToSet
- dict -> std::map : PyDictToMap.

Wykorzystują one interfejsy kolekcji: *boost::python::list* oraz *boost::python::dict* do łatwego przeniesienia całości kolekcji do kolekcji standardowej. Wykorzystany mechanizm boost::python::extract umożliwił dość dokładną kontrolę typów przekazywanych elementów. Po określeniu, na kolekcję jakiego typu elementów ma być dokonana konwersja, użytkownik dostaje kolekcję zawierającą wszystkie elementy (pary elementów) podanego typu (typów) zawarte w kolekcji pythonowej. Mechanizm konwerterów umożliwia jawne informowanie użytkownika o odfiltrowanych elementach niewłaściwego typu poprzez ustawienie flagi *converterWarningsEnable* w pliku Flags.hpp.

Znane błędy:
Próba konwersji typu *long* na typ *int* kończy się rzuceniem wyjątku *Overflow Error*. Wynika to z niewłaściwego działania mechanizmu *extract* i nie może być skutecznie przechwycone po stronie biblioteki StlContainers.

5. Testy

Użyte testy wykonywane są za pomocą wbudowanego interpretera Pythona, co umożliwia zarówno przeprowadzenie testów jednostkowych konkretnych wrapperów / mechanizmów konwersji, jak i proste testy przekazywania kolekcji pomiędzy C++ a Pythonem i vice versa. Testy jednostkowe, sprawdzające działanie wszystkich metod wrapperów i konwerterów umieszczone są w osobnym katalogu *tests/*. Za ich wykonanie odpowiada plik *External.cpp* (po kompilacji: *External_tests).* W pliku *Embedded.cpp* znajduje się dodatkowo przedstawienie łatwego sposobu przekazywania kontenerów wraz z ich zawartością pomiędzy C++ i Pythonem i odwrotnie.

Pliki *Python_test_maps.py, Python_test_sets.py* oraz *Python_test_vectors.py* zawierają grupy testów sprawdzających działanie odpowiednich kontenerów biblioteki STL w Pythonie. Kontenery przetestowano dla typów *int, double* oraz *std::string.* Testy zawierają funkcje, sprawdzające działanie wszystkich metod kontenerów tj. tworzenie kontenera, dodawanie elementów, usuwanie elementów itp.

Pliki *Python_test_vector_foobar.py* oraz *Python_test_vector_foo.py* zawierają podobne testy dla przykładowych klas użytkownika, przy czym klasa *Foobar* dostarcza operatora porównania, natomiast klasa *Foo* nie.

6. Kompilacja i uruchomienie

Kompilacja bibliotek i programów testujących odbywa sie poprzez mechanizm Scons, przez co cały proces jest zautomatyzowany i ogranicza się do jednego polecenia – *scons.* W efekcie kompilacji powstają 3 biblioteki współdzielone – *libStlContainers.so*, *libFoo.so* oraz *libFoobar.so.* Pierwsza zawiera konkretyzacje szablonów typów wbudowanych oraz std::string, dla których powstały testy. Dwie ostatnie zawierają konkretyzacje szablonów dwóch różnych typów użytkownika, z których jeden zawiera operator porównania a drugi nie. Po kompilacji powstają również pliki *External_tests* i *Embedded_tests* umożliwiające uruchomienie testów o których mowa w pkt. 5.

7. Dokumentacja

Kod źródłowy udokumentowany jest zgodnie ze standardem Doxygen, umożliwiając łatwe wygenerowanie dokumentacji HTML lub LaTeX.

8. Napotkane problemy, niezrealizowane założenia

Głównym problemem w trakcie tworzenia projektu była niepełna i nieco chaotycznie napisana dokumentacja biblioteki *Boost.Python*. Wiele mechanizmów było opisanych w sposób szczątkowy, a potrzebne informacje często były znajdowane w innych miejscach, niż możnaby się tego spodziewać. W dodatku wiele elementów tej biblioteki jest w fazie ciągłego rozwoju, za czym nie nadąża lekko przestarzała dokumentacja.

Udało się z powodzeniem zrealizować założenia koncepcji i dokumentacji wstępnej projektu z zastrzeżeniem, że zamiast kontenera *std::list* został opakowany i udostępniony kontener *std::map* stanowiący większe wyzwanie implementacyjne.