Robert Nowak

Wydajne i elastyczne programy

Łączenie C++ i Pythona przy pomocy boost_python

Aplikacje wydajne najwygodniej tworzy się w językach kompilowanych do kodu maszynowego, możemy wtedy wykorzystać wszystkie możliwości, które daje sprzęt. Rozwiązania elastyczne tworzymy, wykorzystując interpreter – nie ma potrzeby translacji do kodu maszynowego. W artykule omówiono przykład stosowania obu podejść jednocześnie dla języków C++ i Python. Komunikację pomiędzy modułami tej samej aplikacji, utworzonymi w różnych tych językach, upraszcza biblioteka **boost_python**.

Języki kompilowane translują kod źródłowy programu komputerowego do kodu binarnego lub kodu pośredniego i zapisują wynik translacji. Jeżeli wynikiem kompilacji jest kod binarny, algorytmy mogą być zaimplementowane wydajnie – procesor nie robi niczego innego, tylko wykonuje nasz kod. Podejście takie ma pewną wadę: kod binarny jest nieprzenośny, nie można go wykonać na procesorze z inną architekturą niż procesor, dla którego wykonano kompilację. Ze względu na wykorzystywanie różnych udogodnień systemu operacyjnego, nasz program w wersji binarnej może nie działać poprawnie na komputerze z innym (niż przeznaczono) systemem operacyjnym. Jeżeli chcemy zapewnić przenośność takiego programu, to tylko na poziomie kodu źródłowego, dla każdej platformy potrzebna będzie niezależna kompilacja.

Języki interpretowane wykonują kod źródłowy, translacja odbywa się ,,na bieżąco''. Nie ma podziału na kod źródłowy i kod wynikowy (np. binarny), procesor wykonując nasz kod, jednocześnie go transluje. Programy nie mogą więc być tak wydajne, jak te poprzednie. Ponieważ mamy do czynienia tylko z kodem źródłowym, programy są przenośne, działają podobnie, niezależnie od systemu operacyjnego, architektury komputera czy architektury procesora.

Kod binarny jest bardzo trudny do czytania przez człowieka, głównie z powodu swojej objętości (miliony instrukcji …), dlatego dostarczając program w wersji binarnej zakładamy, że nasze rozwiązania są ukryte przed użytkownikiem. Nie są więc elastyczne, użytkownik nie może ich modyfikować. Kod źródłowy natomiast jest, a przynajmniej powinien być, czytelny, dostęp do niego pozwala poznać i zrozumieć strukturę programu i użyte algorytmy, tym samym je modyfikować i dostosowywać do własnych potrzeb. Programy pisane w językach interpretowanych są elastyczne, ponieważ użytkownik ma zagwarantowany dostęp do kodu źródłowego.

W naszych aplikacjach istnieją fragmenty, które powinny być wydajne, więc należy je tworzyć w językach kompilowanych, oraz fragmenty, które wygodniej dostarczać jako fragmenty kodu interpretowanego, aby umożliwić użytkownikowi dostosowywanie aplikacji do własnych potrzeb. Pewne moduły chcemy chronić, ukryć rozwiązania przed użytkownikiem, więc dostarczamy je w formie binarnej. Języki interpretowane wybieramy także ze względu na to, że implementacja trwa krócej: nie trzeba kompilować i mamy bogatszy zbiór udogodnień, np. możliwość tworzenia nowych klas w czasie działania programu.

Jeżeli chcemy, aby nasz produkt miał kilka wymienionych cech (wydajny, elastyczny, szybka implementacja, ukryte fragmenty zawierające istotne algorytmy), to warto rozważyć wykorzystanie kilku różnych języków programowania. Narzędzia (translatory) pośrednie, które mają cechy kompilatorów i interpreterów (np. kompilacja do kodu pośredniego, kompilacja przy pierwszym uruchomieniu), nie pozwalają osiągnąć takiej wydajności, jak kompilatory do kodu binarnego i takiej elastyczności jak interpretery. Stosowanie wielu języków jednocześnie wymaga większych umiejętności niż używanie tylko jednego, trzeba je znać i zapewnić komunikację pomiędzy modułami utworzonymi w różnych językach. Dodatkowo trzeba starannie opracować interfejsy modułów, uwzględniając ograniczenia języków, które będą z nich korzystać. W wielu wypadkach proponowane podejście jest uzasadnione. Warto stosować specjalistyczne narzędzia (a takim jest język programowania) w zależności od potrzeb, przy okazji zmniejszając nakład pracy (np. rezygnując z tworzenia własnego interpretera do plików konfiguracyjnych, który będzie działał w aplikacji dostarczanej binarnie).

Artykuł przedstawia sposoby łączenia fragmentów kodu napisanych w C++ (język kompilowany do kodu binarnego) i w Pythonie (język interpretowany), przy użyciu biblioteki **boost_python**. Biblioteka ta pozwala tworzyć moduły binarne, które mogą być używane w Pythonie jak pakiety. Dostarcza także obiektowy interfejs do interpretera Pythona, upraszczając jego używanie w aplikacjach tworzonych w C++.

Szybki start

Aby uruchomić przedstawione przykłady, należy mieć dostęp do interpretera języka Python (http://python.org), kompilatora C++ (http://gcc.gnu.org, http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-cpp-express lub http://mingw.org) oraz edytora tekstu. Dodatkowo należy zainstalować zbiór bibliotek boost (http://www.boost.org). Aby poprawnie zbudować przedstawione przykłady, należy dołączyć bibliotekę **boost_python**. Dla konsolidatora **g++** należy dodać opcję **-lboost_python**, dla konsolidatora Visual Studio (program **link**) dodatkowe opcje nie są potrzebne, biblioteka **boost_python** jest dodawana automatycznie. Na wydrukach pominięto dołączanie odpowiednich nagłówków oraz udostępnianie przestrzeni nazw, pełne źródła umieszczono jako materiały pomocnicze. Materiały te wykorzystują narzędzie do budowania aplikacji Scons (http://scons.org). Przykłady testowano na platformie Ubuntu 11.10 64 bit i Windows 7 32 bit, wykorzystując Python2.7, g++ 4.6, Visual Studio 2010 Express, boost 1.49.

PODSTAWOWE INFORMACJE O BOOST_PYTHON

Moduły utworzone w C++ woła się z języka Python, gdy wykorzystywane algorytmy powinny wykonywać się wydajnie. Taki proces nazywa się rozszerzaniem Pythona w C++. Innym powodem rozszerzania jest potrzeba ukrycia kodu przed użytkownikiem.

Biblioteka **boost_python** dostarcza zbioru szablonów, które pozwalają utworzyć interfejs do funkcji i klas C++. Tak obudowane klasy i funkcje będą dostępne dla programów tworzonych w języku Python. Kod utworzony w C++ musi być dostarczany jako biblioteka dzielona, biblioteka **boost_python** dodaje odpowiedni interfejs. Przykład pokazano na Listingu 1. Dostarczamy tam moduł zawierający funkcję **f** oraz klasę **Foo**.

Listing 1. Przykład modułu C++, który można wołać z Pythona

```
int f() { return 43; }//przykładowa funkcja
class Foo { //przykładowa klasa
public:
   Foo() : val_(0) {}
   int get() const { return val_; }
   void set(int v) { val_ = v; }
private:
   int val_;
};
BOOST_PYTHON_MODULE( cppmodule ) //nazwa pakietu
{
   boost::python::def( "f", f );//eksportuje funkcję
   boost::python::class_<Foo>("Foo",
boost::python::init<>())//eksportuje klasę
      .def( "get", &Foo::get )
      .def( "set", &Foo::set )
   ;
} //koniec definicji pakietu
```

Definicję interfejsu rozpoczynamy, podając nazwę pakietu Pythona (argument makrodefinicji **BOOST_PYTHON_MODULE**). Nazwa ta musi być taka sama jak nazwa biblioteki dzielonej, która zawiera udostępniane byty. Wewnątrz bloku umieszczonego za makrodefinicją umieszczamy adaptery eksportujące funkcje i klasy. Funkcję eksportujemy za pomocą szablonu **def**, podając nazwę, która będzie używana w Pythonie oraz wskaźnik na funkcję w C++. Klasę eksportujemy za pomocą szablonu **class_**, parametrem tego szablonu jest typ w C++, natomiast argumentami: nazwa klasy, która będzie używana w Pythonie, oraz argumenty dla metody **__init__** (wykorzystywany jest szablon pomocniczy **boost::python::init<>**). Możemy dodać (wykorzystując literał **.def**) metody, dostarczając napis i wskaźnik do kodu.

Po poprawnej kompilacji i konsolidacji przedstawionego kodu tworzona jest biblioteka dynamiczna o nazwie **cppmodule.so** dla Linux, **cppmodule.pyd** dla Windows. Jeżeli jest ona dostępna dla interpretera Pythona (znajduje się na ścieżce, np. jest w katalogu, z którego interpreter został uruchomiony), możemy wykonać następujące instrukcje:

```
>>> import cppmodule
>>> cppmodule.f()
43
>>> foo = cppmodule.Foo()
>>> foo.get()
0
>>> foo.set(7)
>>> foo.get()
7
```

Jeżeli chcemy, aby pewne akcje uruchamiać przy imporcie pakietu (inicjować zmienne, startować wątki itp.), to możemy dodać kod wewnątrz bloku definiowanego przez makrodefinicję **BOOST_PYTHON_MODULE**. Na Listingu 2 pokazano przykład, podczas importu pakietu będzie uruchamiana funkcja **initModule()**.

Listing 2. Akcje uruchamiane podczas importu pakietu

```
BOOST_PYTHON_MODULE( cppmodule )
{
   initModule(); //f. wołana podczas importu pakietu
   //...
}
```

KONWERSJE OBIEKTÓW

Biblioteka **boost_python** pozwala przekazać do C++ argumenty, które są typami wbudowanymi (**int, double, bool** itp.) oraz napisami (**std::string**). Odpowiednie konwersje są dodawane przez tę bibliotekę i są wołane automatycznie (wykorzystywaliśmy to na Listingu 1, w metodach klasy **Foo**). Ponadto, możemy przekazywać obiekty klas, które zostały wyeksportowane, więc przykładowo możemy pobierać lub zwracać wartości typu **Foo** (z Listingu 2).

Jeżeli chcemy przekazywać jako argument lub zwracać obiekty innych typów, biblioteka dostarcza klasę **boost::python::object**. Reprezentuje ona obiekt zarządzany przez interpreter Pythona i dostarcza m.in. metodę **attr** umożliwiającą odczyt składowych. Możemy więc funkcje i metody przekazywane do Pythona tworzyć, używając tych udogodnień. Dostarczane są także klasy pochodne po **object**, m.in. **dict** – reprezentuje słownik z Pythona, **list** – reprezentuje listę. Dla wspomnianych kontenerów istnieje przeciążony operator indeksowania oraz metoda **len** (zwraca ilość elementów). Przykład użycia pokazano na Listingu 3, gdzie funkcja **getDate** konwertuje zmienną Pythona typu **datetime.date** na obiekt typu **boost::gregorian::date**. Funkcja ta odczytuje poszczególne składowe dostarczonego obiektu (**year**, **month** itd.) i wykorzystuje konwersje dla typów wbudowanych, wołając szablon **extract**.

Aby użytkownik nie musiał jawnie wołać funkcji konwertującej (takiej jak **getDate** z Listingu 3), możemy je dodać (zarejestrować) w strukturach biblioteki **boost_python**. Wtedy funkcje te będą wołane automatycznie przy pobieraniu i przekazywaniu argumentów odpowiedniego typu, oraz przy wołaniu szablonu **extract<nasz_typ>**, podobnie jak dla typów wbudowanych. Własne konwersje przy tworzeniu obiektu klasy na podstawie obiektu Pythona są rejestrowane przez metodę obiektu **boost::python::converter::registry**, zaś przy tworzeniu obiektu Pythona na podstawie obiektu C++ wykorzystujemy szablon **boost::python::to_python_converter**.

Na Listingu 4 pokazano przykład użycia wspomnianego mechanizmu do konwersji pomiędzy **boost::posix_time::ptime** a **datetime**. Funkcje konwertujące są w tym przykładzie metodami (statycznymi) klasy **BoostPosixPTimeConverter**.

Konwersja z obiektu Pythona na obiekt C++ jest realizowana przez metodę **construct**. Obiekt źródłowy jest pierwszym argumentem (argument **object**), zaś bufor, w którym ma być utworzony obiekt docelowy (obiekt C++), jest drugim argumentem. Konwersja odbywa się przy użyciu funkcji **extract**, która woła wcześniej zarejestrowane konwertery (np. dla typów wbudowanych) lub przy pomocy innych funkcji – w przykładzie są to funkcje Python C API do konwersji daty i czasu. Na koniec inicjujemy wskazywany bufor, umieszczając w nim obiekt C++. W przykładzie do tego celu jest wykorzystywany operator **new** dla wskazywanego bufora (ang. *placement new*). Przy rejestracji przedstawionego konwertera dostarczana jest także funkcja, która bada, czy konwersja jest możliwa, tutaj rolę tę pełni metoda **convertible**.

Listing 4. Klasa zawierająca funkcje konwertujące obiekty typu `boost::posix_time::ptime` na Python `datetime` i odwrotnie

```
struct BoostPosixPTimeConverter {
   static void registerConverter() { //rejestruje konwertery
      boost::python::converter::registry::push_back(  &convertible, &construct,
      boost::python::type_id<boost::posix_time::ptime>() );
      boost::python::to_python_converter<boost::posix_time::ptime, BoostPosixPTimeConverter>();
   }
   //konwersja z boost::posix_time::ptime na obiekt Pythona
   static PyObject* convert(const boost::posix_time::ptime& time) {
      PyDateTime_IMPORT;
      int year = time.date().year();
      int month = time.date().month();
      int day = time.date().day();
      int hours = time.time_of_day().hours();
      int minutes = time.time_of_day().minutes();
      int seconds = time.time_of_day().seconds();
      int mi_s = time.time_of_day().total_microseconds() % 1000000;
      return PyDateTime_FromDateAndTime(year, month, day, hours, minutes, seconds, mi_s);
   }
   //bada, czy jest możliwa konwersja z obiektu Pythona na wskazany typ C++
   static void* convertible(PyObject *object) {
      PyDateTime_IMPORT;
      if(!PyDateTime_Check(object))
         return 0L;
      return object;
   }
   //konwersja z obiektu Pythona na obiekt boost::posix_time::ptime
   static void construct(  PyObject *object,
      boost::python::converter::rvalue_from_python_stage1_data *data) {
      PyDateTime_IMPORT;
      PyDateTime_DateTime const* py_dt = reinterpret_cast<PyDateTime_DateTime *>(object);
      boost::gregorian::date d( PyDateTime_GET_YEAR(py_dt),
      PyDateTime_GET_MONTH(py_dt),
      PyDateTime_GET_DAY(py_dt) );
      boost::posix_time::time_duration t =
      boost::posix_time::hours(PyDateTime_DATE_GET_HOUR(py_dt)) +
      boost::posix_time::minutes(PyDateTime_DATE_GET_MINUTE(py_dt)) +
      boost::posix_time::seconds(PyDateTime_DATE_GET_SECOND(py_dt)) +
      boost::posix_time::microseconds(PyDateTime_DATE_GET_MICROSECOND(py_dt));
      void *storage =
      ((boost::python::converter::rvalue_from_python_storage<boost::posix_time::ptime> *)data)->storage.bytes;
      new (storage) boost::posix_time::ptime(d,t);
      data->convertible = storage;
   }
};
```

Listing 5. Funkcja konwertująca obiekty klasy Foo na słownik Pythona (`dict`). Założono, że klasa Foo dostarcza metodę `get` dla typu, który już posiada konwersję (np. jest to typ wbudowany)

```
PyObject* convert(Foo& foo) { //nazwa tej funkcji jest dowolna
   boost::python::dict result; //zwracamy słownik Pythona
   result["value"] = foo.get(); //element słownika, dla klucza "value"
   return boost::python::incref(result.ptr()); //trzeba zwiększyć licznik odniesień do obiektu, bo będzie on zarządzany
przez interpreter Pythona
}Listing 6. Udostępnianie wyliczeń zdefiniowanych w C++
class Connection { //klasa definiuje wyliczenie
   enum ConnectionState { UNKNOWN, NOT_CONNECTED, CONNECTED };
   //inne elementy interfejsu klasy
};
BOOST_PYTHON_MODULE(nazwa)
{
   boost::python::enum_<Connection::ConnectionState>("ConnectionState")
      .value("UNKNOWN", Connection::UNKNOWN)
      .value("NOT_CONNECTED", Connection::NOT_CONNECTED)
      .value("CONNECTED", Connection::CONNECTED)
      .export_values()
      ;
}
```

Listing 7. Wykorzystanie konwerterów dla `boost::tuple` dostarczonych wraz z `boost_python`.

```
boost::tuple<int, int> getMinMax(int a, int b) {
   return boost::minmax(a, b); //funkcja z biblioteki boost algorithm
}
```

Listing 8. Udostępnianie kontenera `std::vector` wykorzystując szablony zdefiniowane w `boost/python/suite/indexing`. Przedstawiona definicja powinna się znaleźć wewnątrz bloku `BOOST_PYTHON_MODULE`

```
class_< std::vector<Foo> >("VectorFoo", no_init ) //zakładamy dostępność konwersji dla Foo
   .def( vector_indexing_suite< std::vector<Foo> > () )
   ;
```

Utworzenie obiektu Pythona z obiektu C++ jest, w przedstawionym przykładzie, realizowane przez metodę **convert**. Na podstawie argumentu, który jest obiektem C++, metoda ta tworzy obiekt typu **PyObject**. Przy implementacji konwerterów musimy zadbać o odpowiednią inicjację licznika odniesień do tworzonych obiektów Pythona, ponieważ będzie on zarządzany przez środowisko interpretera. Pokazano to na Listingu 5, gdzie konwertujemy obiekt klasy **Foo**, w ostatniej linii zwiększamy licznik odniesień do obiektu. Na Listingu 4 tego nie widać, do konwersji wykorzystujemy funkcje Python C API, które same wykonują przedstawione zabiegi. Obiekty tworzone w C++ i zwracane do Pythona przez wartość są zarządzane przez Pythona i prawidłowo usuwane, więc w prostych przypadkach nie musimy się martwić o czas życia obiektów.

Listing 3. Funkcja w C++, która odczytuje składowe obiektu utworzonego w Pythonie

```
//funkcja demonstrująca odczyt składowych (attr) i
//konwersje (extract)
//Do konwersji wygodniej używać rozwiązań pokazanych na
//Listingu 4.
boost::gregorian::date(const object& d) {
   return boost::gregorian::date(extract<int>(d.
attr("year") ),
   extract<int>(d.attr("month") ),
   extract<int>(d.attr("day") ) ) );
}
```

Do tworzenia modułów, które będą wołane w Pythonie, zazwyczaj wystarczą nam omówione poprzednio udogodnienia: rejestracja funkcji, klas oraz własnych konwersji. Dodatkowo biblioteka pozwala uprościć rejestrację typów wyliczeniowych (patrz Listing 6) oraz wspiera automatyczne konwersje dla obiektów typu **tuple** (**boost::tuple**), **vector** i innych. Upraszcza to przekazywanie najczęściej wykorzystywanych struktur danych.

Listing 6. Udostępnianie wyliczeń zdefiniowanych w C++

```
class Connection { //klasa definiuje wyliczenie
   enum ConnectionState { UNKNOWN, NOT_CONNECTED, CONNECTED };
   //inne elementy interfejsu klasy
};
BOOST_PYTHON_MODULE(nazwa)
{

boost::python::enum_<Connection::ConnectionState>("ConnectionState")
       .value("UNKNOWN", Connection::UNKNOWN)
       .value("NOT_CONNECTED", Connection::NOT_CONNECTED)
       .value("CONNECTED", Connection::CONNECTED)
       .export_values()
       ;
}
```

Obiekty **boost::tuple** są automatycznie konwertowane na krotki Pythona (obiekty typu **tuple**), tak jak pokazano na Listingu 7. Warto wykorzystać tę cechę biblioteki **boost_python**, ponieważ pozwala ona przekazywać proste obiekty z kilkoma składowymi (dla większości kompilatorów C++ z obiektami do 10 składowych) bez konieczności tworzenia konwerterów.

Zbiór szablonów **suite/indexing**, który jest częścią biblioteki, pozwala dostarczać jednowymiarowe kontenery biblioteki standardowej (**vector**, **list**, **set** itp.). Przykład (Listing 8) pokazuje definicje dla typów generowanych z szablonu **std::vector**.

OSADZANIE PYTHONA W C++

Kod interpretera jest elastyczny, można go zmieniać i uruchamiać bez konieczności kompilacji, więc zmiany tego kodu są proste i dostępne dla użytkownika oprogramowania. Jeżeli pliki konfiguracyjne aplikacji są skryptami interpretera (np. Pythona), to użytkownik może je zmieniać, a ponadto może umieszczać własne rozszerzenia, np. obliczać wartości parametrów konfiguracyjnych, co często jest wygodne. Używanie interpretera, dostarczonego jako biblioteka, w module napisanym w C++ nazywamy osadzaniem Pythona w C++. Aby poprawnie zbudować taką aplikację, musimy konsolidować (linkować) interpreter dostarczany w formie biblioteki dzielonej, np. **libpython27.so** lub **libpython27.dll**. Przykład wykorzystujący **boost_python** pokazano na Listingu 9, gdzie interpreter otrzymuje napis **res = 2*3*4**, który wykonuje, traktując go jak program. Następnie (w C++) badamy zawartość zmiennych globalnych interpretera, które odczytujemy za pomocą szablonu **extract**.

PODSUMOWANIE

Przedstawiony tekst ma zachęcić czytelnika do tworzenia aplikacji, wykorzystując jednocześnie różne języki programowania. Aplikacje takie mogą posiadać pożądane cechy (np. wydajność i elastyczność) stosunkowo niewielkim kosztem – trzeba opanować sposób komunikacji pomiędzy modułami pisanymi w różnych językach. Istnieje wiele narzędzi upraszczających to zadanie, w artykule pokazano proste przykłady łączenia C++ i Pythona, wykorzystując bibliotekę **boost_python**.

W Sieci

- http://hepunx.rl.ac.uk/BFROOT/dist/releases/24.2.1h/boost/libs/python/doc/PyConDC_2003/bpl.pdf – artykuł wprowadzający do boost_python (Building Hybrid Systems with Boost. Python);
- http://steve.vinoski.net/pdf/IEEE-Multilingual_Programming.pdf – artykuł o korzyściach z programowania w wielu językach.

Więcej w książce

Omówienie współcześnie stosowanych technik, wzorce projektowe, programowanie generyczne, prawidłowe zarządzanie zasobami przy stosowaniu wyjątków, programowanie wielowątkowe, łączenie C++ i Pythona, ilustrowane przykładami stosowanymi w bibliotece standardowej i bibliotekach boost, opisano w książce Robert Nowak, Andrzej Pająk ,,Język C++: mechanizmy, wzorce, biblioteki", BTC 2010.

Robert Nowak rno@o2.pl

Adiunkt w Zakładzie Sztucznej Inteligencji Instytutu Systemów Elektronicznych Politechniki Warszawskiej, zainteresowany tworzeniem aplikacji wykorzystujących algorytmy sztucznej inteligencji i fuzji danych. Autor biblioteki faif.sourceforge.net. Programuje w C++ od ponad 15 lat.