

Łączenie języków Perl i C/C++

Klejenie na szybko

Aplikacjami, których nie brakuje dla Linuksa, są bez wątpienia języki programowania. I bardzo dobrze – mamy przynajmniej z czego wybierać. Jednym z powodów tak wielkiej różnorodności jest to, że obecnie nie istnieje jeden, idealny język programowania.

ARMIJN HEMEL

Języki (także języki ogólnego stosowania) opracowano z myślą o określonych celach – twórcy języków programowania często mają zaś różne cele, a więc ich języki będą różniły się od pozostałych. W wyniku tego często zdarza się, że pewne zadania możemy wykonać szybciej używając określonego języka.

Niektóre osoby uparcie twierdzą, że poważne programy powinny być pisane w języku C, a program napisany w języku skryptowym nigdy nie będzie wystarczająco wydajny. Rzeczywistość jest jednak odmienna. Wiele języków skryptowych zostało wdrożonych w bardzo wydajny sposób i już przy niewielkim wysiłku możemy napisać dobry, odpowiedniej wydajności kod programu. Znane powiedzenie – „właściwe narzędzie do właściwego zadania” szczególnie sprawdza się, gdy trzeba wybrać odpowiedni język programowania.

Ulubionym językiem programowania autora tego artykułu (w chwili obecnej) jest Python. W języku tym można dużo łatwiej i szybciej niż w C stworzyć kod programu. Należy jednak pamiętać, że nie wszystko można zrobić przy pomocy języka Python. Istnieją całe stopy kodu C lub C++, dla których nie ma odpowiednika w modułach Pythona.

Jednym z przykładów (zajmiemy się nim w dalszej części artykułu) jest kod umożli-



wiający dostęp do szyny i urządzeń USB – libusb (biblioteka ta jest wykorzystywana przez takie programy jak SANE czy gPhoto). Dzięki libusb możemy pisać aplikację działającą z poziomu użytkownika, a zatem bez obawy, że jednym, źle przygotowanym sterownikiem spowodujemy niestabilną pracę systemu.

Chcielibyśmy oczywiście napisać program w języku Python, ale dla tego języka nie istnieje biblioteka libusb. Będziemy zatem musieli użyć języka C. Co zrobić w takim przypadku?

Rozwiązaniem tego konkretnego problemu mogłaby być ponowna implementacja funkcjonalności libusb w Pythonie. Jest to niestety zadanie podatne na błędy (możemy nieświadomie wprowadzić do biblioteki błędy, których nie było w oryginalnej implementacji) i bardzo czasochłonne (kodowanie i testowanie całej biblioteki wymaga czasu). Z punktu widzenia jakości oprogramowania, ponowna implementacja działającego kodu w innym języku programowa-

nia jest zdecydowanie nieprawidłowym działaniem.

Lepiej byłoby skorzystać z istniejącego kodu C w obrębie języka skryptowego i wykorzystać pracę wykonaną już przez innych. To właśnie umożliwia SWIG (Simplified Wrapper and Interface Generator) – generator interfejsów i kodu osłonowego. Jest to narzędzie do tworzenia kodu łączącego istniejący kod C/C++ z całą gamą języków skryptowych np. Perl i Python oraz języków typu Java czy C#.

Wiele interpreterów posiada wbudowane mechanizmy, które wywołują kod C/C++ znajdujący się poza interpreterem. Perl posiada mechanizm XS, a język Java korzysta z usług JNI (ang. Java Native Interface). Zaletą SWIG jest automatyzacja (możliwa do pewnego stopnia oczywiście), dzięki czemu łączenie kodu jest dużo prostsze niż przy zastosowaniu metod tradycyjnych.

Zasada działania SWIG

Cały proces łączenia kodu C/C++ z językiem skryptowym można podzielić na cztery etapy:

- zapis pliku interfejsu określającego, które funkcje należy wyeksportować,
- stworzenie przez SWIG kodu osłaniającego

go dla języka przeznaczenia przy pomocy pliku interfejsu,

- kompilacja kodu osłaniającego dla obiektu,
- powiązanie plików obiektu i bibliotek w bibliotece wspólnej.

A więc, jak to działa? Pamiętajmy, że wiele interpreterów (jeżeli nie wszystkie), dla których SWIG generuje kod, zostało napisanych w języku C lub C++. Wywołanie pewnych funkcji w tym samym kodzie co interpreter, to żadna magia.

Plik interfejsu opisuje funkcje w kodzie C/C++, które powinny być widoczne dla języka skryptowego. W prostym przypadku plik ten będzie składać się tylko z kilku linii. Ale już w bardziej złożonych sytuacjach może on osiągnąć spore rozmiary.

Kod C, który generuje SWIG, bardzo trudno jest analizować, gdyż służy on w zasadzie tylko do umieszczenia właściwych plików nagłówkowych dla wybranego interpretera. O kodzie osłaniającym najlepiej myśleć jak o czarnej skrzynce i nie wnikać w sposób jego działania.

Prosty przykład

Wyobraźmy sobie, że mamy następujący program (*hello.c*):

```
#include <stdio.h>
#include "hello.h"

int printhelloworld() {
    printf("hello world\n");
    return 0;
}

int main(int argc, char* argv) {
    printhelloworld();
    return 0;
}
```

używającym pliku nagłówkowego *hello.h*:

```
int printhelloworld();
```

Program wyświetla na ekranie słowa *hello world* i kończy działanie. Załóżmy, że chcemy użyć funkcji *printhelloworld* z poziomu języka Python.

Jak to opisaliśmy wcześniej, mamy plik interfejsu:

```
%module printhelloworld
%{
#include "hello.h"
%}
int printhelloworld();
```

W tym przypadku interesuje nas wyłącznie funkcja *printhelloworld()*. Ponieważ w naszym przykładzie są to w zasadzie wszystkie funkcje programu (dla uproszczenia pomijamy *main()*), możemy zatem napisać:

```
%module printhelloworld
%{
#include "hello.h"
%}
#include "hello.h"
```

Zostanie tutaj przetworzony nagłówek pliku, a następnie generowany jest kod dla wszystkich funkcji, do których chcemy mieć dostęp z poziomu Pythona, tak jak to określiliśmy w pliku nagłówka.

Moduł Pythona, który będziemy generować, będzie nosił nazwę *printhelloworld* (zdefiniowany słowem kluczowym *module*). Treść z nawiasów klamrowych (linie 2 i 4) zostanie umieszczona bezpośrednio w kodzie osłaniającym, generowanym przy pomocy polecenia:

```
$ swig -i python hello.i
```

W jego wyniku otrzymujemy plik o nazwie *hello_wrap.c* zawierający gotowy kod. Dla innych języków skryptowych należy użyć innych parametrów (np. dla języka Perl -i *perl*).

```
$ gcc -c hello_wrap.c -I /usr/include/python2.2/
```

W tym przypadku musimy podać ścieżkę dostępu do pliku nagłówka *python.h*, ponieważ plik nie znajduje się w domyślnej ścieżce wyszukiwania. Tak jest np. w dystrybucjach opartych o Red Hat (np. Fedora). Plik nagłówkowy może też znajdować się w różnych miejscach, np. dla Fedora Core 2 musimy użyć */usr/include/python2.3/*.

Aby powiązać rzeczywisty kod programu z pliku *hello.c*, musimy najpierw skompilować plik obiektowy: źródłowy do postaci pliku obiektowego:

```
$ gcc -c hello.c
```

Ostatnim krokiem będzie połączenie wszystkich plików obiektowych:

```
$ gcc -shared hello_wrap.o hello.o -o _printhelloworld.so
```

Wskazówka: Jeżeli pracujemy regularnie z programem SWIG, ciągle wpisywanie tych

samych poleceń może być nudne i denerwujące, dlatego zalecamy utworzenie prostego, uniwersalnego pliku *Makefile*.

To już wszystko! Aby sprawdzić poprawność działania, możemy ręcznie uruchomić interpreter Pythona lub napisać mały program, który zrobi to za nas. Interpreter języka Python rozpocznie przeszukiwanie ścieżki dostępu do bibliotek Pythona pod kątem *_printhelloworld.so* i jeżeli próba wyszukania nie powiedzie się, musimy sprawdzić, czy podaliśmy właściwy katalog roboczy w katalogu głównym (np. */usr/lib/python2.2/*) lub też, czy uruchomiliśmy skrypt z tego samego katalogu, w którym znajduje się biblioteka.

```
import printhelloworld

printhelloworld.printhello
world()
```

Uruchomienie programu spowoduje wyświetlenie na ekranie komunikatu następującej treści:

```
[armijn@swig]$ python test.py
hello world
```

Wymiana danych między językami Python i C

W rzeczywistości nie wszystko jest jednak takie proste jak w naszym przykładzie. Często zdarza się np. wymiana danych pomiędzy kodem wywołującym (skrypcem) a biblioteką – co wtedy zrobić?

SWIG posiada własną obsługę konwersji typów podstawowych C (*int*, *short*, *long*, *char*, *bool*) z i na języki skryptowe. Wszystko inne jest traktowane przez SWIG jako wskaźnik. Ma to pewne znaczenie przy przekazywaniu zaawansowanych struktur danych.

Listing 1: Zamiana tablicy

```
%typemap(in) (char *bytes, int
size) {
    if (!PyString_Check($input)) {
        PyErr_SetString(PyExc_ValueError,
            'Expecting a string');
        return NULL;
    }
    $1 = (void *) PyString_AsString($input);
    $2 = PyString_Size($input);
}
```

Przykładowo dla użytkownika łańcuch znaków w Pythonie i łańcuch znaków w C/C++ nie różnią się zbytnio od siebie, ale jeśli spojrzymy na ich realizację, zauważymy z pewnością duże różnice. Łańcuch znaków Pythona nie jest typu `char*`, lecz `PyString` (inny rodzaj danych). Nie możemy zatem podać w języku C funkcji oczekującej na wartość `char*` wartości `PyString`.

Inny przykład: w języku Python zakres liczb całkowitych jest większy niż w przypadku języka C. Jeżeli dokonamy teraz wymiany danych innych niż typów prostych, będziemy musieli dokonać konwersji. W przypadku SWIG nie jest to zbyt trudne, ale ma swoją cenę: konwersja zależy od języka, więc tracimy możliwość używania jednego pliku interfejsu do tworzenia kodu dla różnych języków programowania.

Aby dostosować dane, SWIG korzysta z tzw. map typów. Dzięki nim możemy ponownie zdefiniować zachowanie programu SWIG przy tworzeniu kodu osłaniającego. Mapy typów znajdują zastosowanie w kontroli typów argumentów, obsłudze wyjątków (w C++) oraz przy konwersji argumentów. My zajmiemy się tylko konwersją argumentów (ze względu na bibliotekę `libusb`).

Typowym sposobem wymiany danych w języku C jest przekazywanie wskaźnika do części pamięci, gdzie może on przechowywać dane, które `libusb` wykorzystuje do przeniesienia danych do urządzenia USB:

```
int usb_bulk_write(usb_dev_handle *dev, int ep, char *bytes, int size, int timeout);
```

Funkcja zapisuje tablicę znaków do punktu końcowego na urządzeniu USB. Jest to możliwe dzięki wskaźnikowi `char`.

W języku Python chcemy po prostu przekazać tablicę znaków (będącą tym samym co łańcuch znaków w Pythonie). Ponadto nie można statycznie zmieniać wielkości tablic w tym języku – możliwa jest zmiana dynamiczna, więc podawanie argumentu dotyczącego wielkości nie ma tutaj sensu.

Z podaną mapą typów (patrz Listing 1) możemy zamienić tablicę z Pythona na `char*` i `int` języka C.

W Pythonie wpisujemy teraz:

```
libusb.usb_bulk_write(device, endpoint, bytearray, 2048)
```

gdzie `bytestring` to łańcuch znaków, `device` to obiektowa reprezentacja urządzenia USB, a `endpoint` to liczba punktów końcowych tego urządzenia.

Taka mapa typów zamienia dane przepływające z języka skryptowego na język C. Mapy typów działają schematycznie, dopasowując argumenty. Działa to dla każdej definicji (`char*`, `int`), która zostanie odnaleziona w kodzie programu (przy zachowaniu kolejności).

Powinniśmy cały czas pamiętać o tym, że dopasowywanie w mapie typów opisuje ciąg parametrów po stronie języka C/C++, a nie po stronie języka skryptowego.

Jak widać na przykładzie pokazanej mapy typów, wykonywana jest kontrola, czy przekazane dane są rzeczywiście typu `PyString` (`$input` to specjalny obiekt utrzymujący zamienianą wartość) i jeżeli tak jest – tworzone są dwa argumenty dla funkcji w języku C: konwersja oryginalnego `PyString` do `char*` i obliczenie parametru `int`, z uwzględnieniem rozmiaru `PyString`, który został przekazany. Dalej następuje wywołanie funkcji `usb_bulk_write` ze wszystkimi oryginalnymi

parametrami oraz dwoma nowymi parametrami utworzonymi w miejsce `PyString`.

W podobny sposób uzyskuje się dane z języka C. Funkcja języka C `usb_bulk_read` zdefiniowana została na Listingu 2.

Mapa typu `argout` działa w drugą stronę – zamienia parametry wychodzące na język Python. Poza pierwszym bitem (który sprawdza, czy istnieje związek ze zmienną w Pythonie) kod jest bardzo prosty – `char*` jest rzutowany na łańcuch znaków Pythona i zwracany przy pomocy specjalnej zmiennej `$result`.

W języku Python wystarczy wpisać:

```
read_result = libusb.usb_bulk_read(device, ep, 16, 1024)
```

Poza mapą typu `argout` istnieje także mapa typów `out`, która zamienia wynik funkcji języka C. W naszym przypadku użycie mapy typów `out` nie jest zbyt przydatne, ponieważ interesujące nas dane zostały już przekazane przez `char*`. Wynikiem działania funkcji `usb_bulk_read` jest liczba całkowita wskazująca, czy odczyt danych zakończył się sukcesem, czy też nie – nie dotyczy to samych danych.

Po zdefiniowaniu mapy typów każda deklaracja funkcji, która pojawi się w pliku interfejsu SWIG, zostanie „przykryta” mapą typów. Można je zastąpić nowymi mapami typów lub ponownie zdefiniować wersje poprzednie.

Królicza nora

Tworzenie bezpośredniego mapowania z języka C na język skryptowy nie jest trudne; przygotowanie podstawowego mapowania Pythona dla `libusb` zajęło nam około jednego dnia, w zasadzie bez żadnej dodatkowej wiedzy, poza znajomością SWIG (nie wykraczającą zresztą poza przykłady typu *hello world*).

Prawdziwym wyzwaniem przy doklepaniu języka skryptowego do kodu C/C++ jest natomiast przygotowanie funkcji w taki sposób, który jest naturalny dla programistów określonego języka skryptowego, np. w postaci modułów dla języka Python. Jest to chyba najtrudniejsze zadanie do wykonania przy pomocy SWIG, a prawdopodobnie jedyną, właściwą drogą do osiągnięcia tego celu jest praktyka. ■

Listing 2: usb_bulk_read

```
int usb_bulk_read(usb_dev_handle *dev, int ep, char *bytes, int size, int timeout);

%typemap(argout) (char *bytes, int size) {
    Py_XDECREF($result);
    if (result < 0) {
        free($1);
        PyErr_SetFromErrno(PyExc_IOError);
        return NULL;
    }
    $result = PyString_FromStringAndSize($1, result);
    free($1);
}
```

INFO

Strona domowa SWIG: <http://www.swig.org>