ИМПОРТ C++ КОДА В PYTHON ПРИ ПОМОЩИ ПАКЕТА SWIG

Иванов А.В., ©31 августа 2010 г.[1]
aivanov(злая собака)keldysh(жирная точка)ru
Институт прикладной математики им. М.В. Келдыша РАН

В одну телегу впрячь неможно
Коня и трепетную лань.
Забылся я неосторожно:
Теперь плачу безумства дань...

А.С. Пушкин «Полтава»

Содержание

1 Введение

Для импорта C++ кода в Python может быть много причин. В общем случае, скорость разработки небольших приложений на Python на порядок выше чем на C++, а их производительность на один–три порядка ниже (в зависимости от специфики задачи). Поэтому возникает вполне разумное желание написать на Python прототип,

[1] Последняя правка 02 Фев 2011 22:47

провести т.н. профилирование (замер производительности), определить критические по производительности участки кода (обычно они весьма невелики), переписать эти участки на C++ и встроить их в прототип. В итоге, производительность полученного приложения будет на проценты ниже, чем если бы оно было целиком написано на C++, а скорость разработки окажется в разы выше.

В ряде случаев можно заранее сказать, какие участки кода должны быть написаны на C++, а какие на Python. Например, в численном моделировании, т.н. вычислительное ядро пишется на C++ (поскольку производительность ядра крайне важна), а интерфейс пишется на Python (поскольку для интерфейса важны скорость разработки, гибкость и возможность быстрой модификации кода). Для задач численного моделирования такую архитектуру можно считать идеальной.

Впервые об импорте C++ кода в Python я прочитал у Марка Лутца в [2]. Там же было описано использование пакета SWIG (Simplified Wrapper and Interface Generator — упрощённый генератор оболочек и интерфейсов, см. http://swig.org) — инструмента, на мой взгляд великолепно подходящего для импорта всего во вся. Существенную помощь, как ни странно, оказала документация SWIG.

Хотя по данному вопросу есть много различных источников, но тема нетривиальная. За последние годы накопился некоторый опыт решения сугубо прагматических задач, которым и хочется поделиться.

2 Первые шаги

Типичная схема включения пользовательского C++ кода в Python при помощи пакета SWIG представлена на рис. 1. Про утилиту make лучше всего прочитать у тех, кто ее придумал [3], а без нее будет тяжко — команд для сборки несколько, их последовательность имеет значение, а компиляция ..._wrap.cxx файла может занимать десятки секунд — он обычно здоровый[2].

Файл mymodule.i с инструкциями для SWIG в минимальной комплектации имеет следующий вид:

```
File "mymodule.i":
-------------------------------------------
%module mymodule
%{
#include "mymodule.hpp"
%}
%include "mymodule.hpp"
-------------------------------------------
```

Здесь просто задается имя модуля, и указываются заголовочные файлы, из которых SWIG должен извлекать классы и функции для импорта в Python. Первое упоминание заголовочного файла будет включено во _wrap.cxx без изменений, как и весь код из .i файла, находящийся между %{ и %}. А вот второе упоминание заголовочного файла говорит о том, что этот файл должен быть проанализирован SWIG и все находящиеся в нем классы и функции должны быть проимпортированы. Не следует начинать имя класса со знака подчеркивания, это может привести к непредсказуемым последствиям при импорте. Если класс или функция не должны импортироваться, используйте директиву %ignore.

Файл для утилиты make (обычно он называется Makefile) выглядит чуть сложнее:

```
File "Makefile":
-------------------------------------------
PY=/usr/include/python

mymodule : mymodule.py _mymodule.so;

mymodule.py mymodule_wrap.cxx : mymodule.i mymodule.hpp
     swig -c++ -python mymodule.i

mymodule_wrap.o : mymodule_wrap.cxx mymodule.hpp
     g++ -I$(PY) -c mymodule_wrap.cxx
```

[2] К вопросу о том, зачем нужен SWIG — загляните на досуге в ..._wrap.cxx и представьте, что все **это** придётся писать руками.

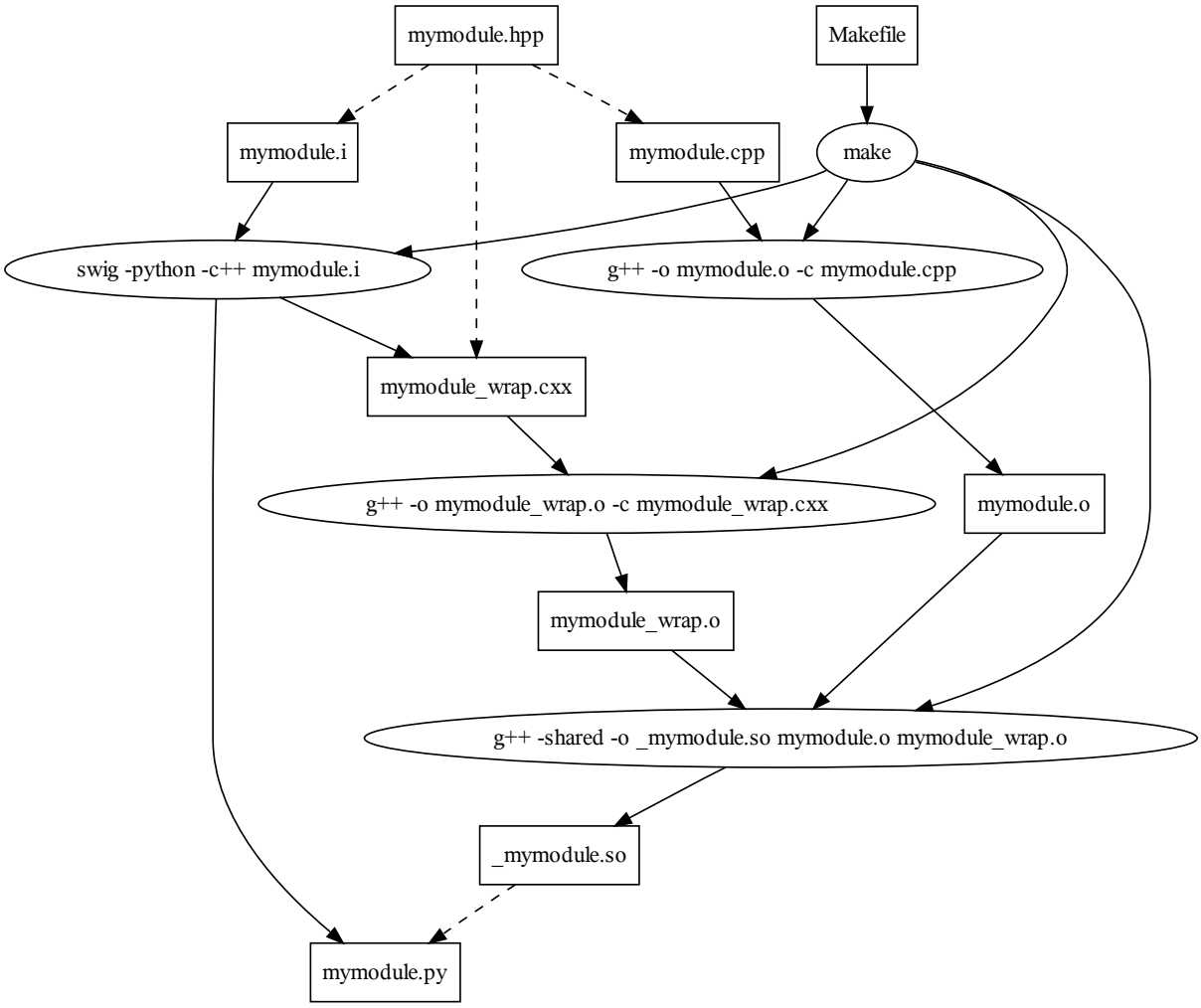

Рис. 1: Типичная схема включения пользовательского `C++` кода в `Python`. Код находится в файлах `mymodule.hpp` и `mymodule.cpp`. Пользователь дополнительно создает файл `mymodule.i` для работы пакета `SWIG` и `Makefile` [3] для управления сборкой (специфичные для системы опции компилятора и линкера не указанны). Пакет `SWIG` на основе `mymodule.i` и указанного в нем заголовочного файла `mymodule.hpp` создает файл `mymodule_wrap.cxx` с `C++` кодом оболочек для функций и методов классов из `mymodule.hpp` и внешнюю оболочку `mymodule.py`, в которую импортируется полученный после сборки модуль `_mymodule.so` (`.so` это аналог `.dll` в `Windows`). На первый взгляд выглядит жутковато? Тогда подождите, пока у Вас не появится **несколько** модулей!

```
mymodule.o : mymodule.hpp mymodule.cpp
     g++ -I$(PY) -c mymodule.cpp

_mymodule.so : mymodule_wrap.o mymodule.o
     g++ -shared -o _mymodule.so mymodule_wrap.o mymodule.o
-----------------------------------------
```

Здесь указано какая цель (файл) от каких целей (файлов) зависит и как именно цели нужно порождать. Для сборки достаточно дать команду

```
$ make
```

Собираться будут лишь те цели, которые нуждаются в сборке (т.е. те цели, которые по времени создания старше чем их зависимости). Например, если после сборки внести изменения только в `mymodule.cpp` и запустить `make`, то будут пересобраны лишь `mymodule.o` и за ним `_mymodule.so`, остальные файлы останутся без изменений. Если изменить `mymodule.hpp`, будет пересобрано все. Это удобно — выполняется лишь необходимый минимум действий.

Здесь существенно, что имя `.so` файла именно `_mymodule.so`. Руководствуясь именем модуля из `mymodule.i`, утилита `swig` создаст именно файл `mymodule.py`, в который будет импортироваться именно файл `_mymodule.so`.

При компиляции необходимо указывать путь к файлу `Python.h` (переменная `PY` в `Makefile`). Для поиска можно воспользоваться утилитой `locate`

```
$ locate Python.h
```

Если у Вас `Python.h` отсутствует, необходимо установить пакет `python-devel`.

3 Способы «ручного управления» импортом кода

Вопрос в том, что делать, когда `SWIG` по умолчанию не делает всего, чего от него хотелось бы. Условно можно выделить пять уровней, на которых можно влиять на импорт кода. Ряд проблем, возникающих при импорте кода, приходиться решать согласованно сразу на нескольких уровнях.

3.1 Обычный `C++` код

Просто пишем функции и классы на `C++`, а `SWIG` их по умолчанию импортирует. На самом деле, `SWIG` прекрасно импортирует по умолчанию большую часть всего существенного из `C++` кода, и приведённой в предыдущем разделе информации достаточно в 99.9% случаев. В частности, `SWIG` корректно преобразует базовые типы к объектам `Python` и обратно, импортирует классы (обеспечивая доступ только к `public` атрибутам), автоматически перегружает функции и операторы.

Все, что написано дальше, относится к некоторыми изыскам.

3.2 `C++` код со специальными именами методов классов

Это обычный `C++` код, который `SWIG` импортирует по умолчанию. Например так реализуются операторы индексации массива `[]` — `SWIG` сам с этим не справляется, поскольку в `Python` методы чтения и записи элемента массива это разные методы. Если в классе определён оператор

```
T& operator[] ( const I& i );
```

где `T` и `I` соответствующие типы, то для его импорта в `Python` придётся добавить два метода

```
void __setitem__( const I& i, const T& v ){ (*this)[i] = v; }
T    __getitem__( const I& i ){ return (*this)[i]; }
```

Подробнее названия и назначения таких методов можно посмотреть в [1], раздел 11.6.3 (специальные методы), стр. 153.

3.3 Создание/разбор объектов Python в пользовательском C++ коде

Речь идёт о пользовательском C++ коде, где функции и методы классов принимают/возвращают объекты Python т. е. указатели PyObject*, а SWIG их по умолчанию импортирует. В заголовочном файле Python.h (точнее во включаемых в него файлах) объявлены функции, образующие Python API. Их имена достаточно описательны, можно разобраться без справки, при помощи этих функций можно собирать или разбирать объекты Python непосредственно в C++ коде.

Например, специальный метод __str__() возвращает объект в виде строки для печати. Естественно можно в классе определить одноимённый метод, возвращающий const char*, но возникает проблема со сборкой мусора (где собственно лежит возвращаемая строка, доживет ли она до использования, и будет ли удалена после того, как надобность в ней отпадет). Есть много решений, но одно из самых на мой взгляд изящных состоит в возврате строки Python — тогда сборку мусора проведет сам Python.

```
PyObject* __str__() const {
    char buf[1024]; snprintf( buf, 1024, /*выводим объект*/ );
    return PyString_FromString(buf);
}
```

Или, допустим для инициализации размеров массива удобно передавать в метод init кортеж с целыми числами. Тогда

```
void init( PyObject* t ){
     int len = PyTuple_Size( t );
     array_size = new int[ len ];
     for( int i=0; i<len; i++ ){
         array_size[i] = PyInt_AsLong( PyTuple_GetItem(t,i) );
     }
     ........
}
```

Следует помнить, что в Python отсутствует проверка типов аргументов методов, поэтому если вместо кортежа передать скажем строку, возникнет ошибка.

Такой подход следует использовать с большой осторожностью, поскольку сборка мусора Python, основанная на подсчёте ссылок, на уровне Python API имеет весьма невнятный вид. Да, для инкрементации/декрементации счётчика ссылок объекта предоставляются макросы Py_INCREF/Py_DECREF, но внутреннюю логику работы системы мне понять так и не удалось. Объект создается со счётчиком ссылок равным единице (что уже странно), а дальше, при добавлении объекта в другие объекты счётчик ссылок обычно не увеличивается (что совсем странно), но в некоторых случаях увеличивается (что просто непонятно). Может поэтому утилита valgrind (проверяющая корректность работы с памятью) выдает такие результаты:

```
$ valgrind python -c 'print "Hello, world!"'
    ....
  ERROR SUMMARY: 527 errors from 33 contexts
    ....
```

3.4 Инструкции для SWIG, передаваемые в .i файле

Если SWIG что то не может сделать по умолчанию, ему вполне можно объяснить, что от него требуется. Именно так инстанцируются шаблоны (параметризованные классы), обрабатываются указатели, исключения и т.д. В SWIG входит большое количество различных библиотек, существенно расширяющих его функциональность — надо только заглянуть в документацию. Некоторые библиотеки будут рассмотрены ниже.

SWIG может создавать дополнительные объекты и функции для преобразования указателей, создания C массивов и других библиотечных объектов, и вносить пользовательский код в генерируемые оболочки обрабатываемого модуля во ..._wrap.cxx файле.

Использование гибко настраиваемых директив %typemap дает возможность пользователю при помощи функций Python API изменять заданные по умолчанию преобразования аргументов и результатов выполнения

функций от типов C++ в объекты Python и обратно. Изменения вносятся на уровне оболочек в ..._wrap.cxx файле.

Директивы %typemap занимают в работе SWIG ключевое место. Следует отметить, что пользовательские директивы %typemap должны задаваться в .i файле после задания имени модуля, но до включения обрабатываемых заголовочных файлов. Несколько примеров будет рассмотрено ниже, за подробностями лучше обратиться к руководству http://www.swig.org/Doc1.3/Typemaps.html.

Грамотное использование SWIG позволяет полностью избежать применения функций Python API в пользовательском коде, описанного в предыдущем разделе — весь такой код может быть перенесён в .i файл, причём в существенно более лаконичной форме.

3.5 Python код

Речь идёт о коде Python, встраиваемом утилитой swig в создаваемый .py файл при помощи команды в .i файле вида

```
%pythoncode %{ ... %}
```

Естественно, все что делается на этом уровне может быть сделано и после загрузки модуля в Python, но с точки зрения дизайна приложения лучше, чтобы вся функциональность модуля была в этом модуле и локализована. Кроме того, некоторые вещи на Python удается сделать гораздо изящнее чем на C++.

4 Всякая всячина

4.1 Где мои глобальные переменные?!

Один из первых вопросов человека, пересевшего с OS Windows под OS Linux, — «где мои диски?!» Если Вы объявили глобальные не-константные переменные, то загрузив свой модуль в Python Вы тут же издадите аналогичный вопль — «Где мои глобальные переменные?!» Причём глобальные переменные, объявленные как константы, никуда не денутся. Ответ сродни ответу на вопрос про диски (в /mnt/, RTFM!) — все Ваши не-константные глобальные переменные являются атрибутами объекта с именем cvar. Имя объекта может быть изменено при вызове утилиты swig через опцию -global.

Дело в том, что Python, хоть местами и похож на C/C++, идеологически отличается очень сильно. В частности, все объекты в Python передаются по ссылке. И, что ещё хуже, все объекты делятся на изменяемые и неизменяемые. В частности, в коде Python

```
a = 1; b = a
```

индентфикаторы a и b ссылаются на один и тот же объект типа int. Но после

```
a += 1
```

индентфикаторы a и b будут ссылаются на **разные** объекты.

Поэтому константные глобальные переменные содержатся в импортируемом модуле непосредственно — если в исходном C++ коде есть глобальная переменная a

```
const int a = 1;
```

то в импортируемом модуле в Python появится переменная a, равная единице. Но при попытке ее изменить, изменится переменная в модуле mymodule.py, а переменная в скомпилированном пользовательском модуле _mymodule.so об этом никогда не узнает, и это естественное для Python поведение.

Допустим переменная a объявлена не как константа в файле mymodule.hpp

```
extern int a;
```

При этом в файле mymodule.cpp конечно должно быть

```
int a = 1;
```

иначе линкер Вас не поймёт.[3] Если при импорте переменная a окажется непосредственно в mymodule.py, то возможности изменить ее значение не будет. Поэтому все глобальные переменные, которые не являются константами, помещаются в виде атрибутов в специальный объект cvar модуля mymodule.py. Если нужно изменить значение переменной a, достаточно указать

```
cvar.a = 2
```

Если глобальная переменная принадлежит к базовому неизменяемому типу, то тут ничего не поделаешь — объект cvar (или его аналоги), это единственный способ хранения таких переменных, допускающий их изменение. Но, если глобальная переменная принадлежит к изменяемому типу, и пользователю необходимы лишь ее поля и методы, то такую переменную можно перенести непосредственно в модуль mymodule.py. Делать это правда придётся в Python (пятый уровень предыдущего раздела), добавив в mymodule.i строчку

```
%pythoncode %{my_global_var = cvar.my_global_var %}
```

К сожалению, SWIG (у меня SWIG Version 1.3.35) корректно импортирует глобальные переменные лишь из первого пользовательского модуля, который включен в .i-файл директивой include, возможно в следующих версиях это будет исправлено.

Конечно, использование глобальных переменных это плохой стиль программирования. Но иногда просто нет другого выхода, например если речь идёт о стандартных потоках ввода/вывода.

4.2 Указатели и ссылки

В отличии от C++, в Python нет различия между указателями и ссылками — все объекты всегда передаются по ссылкам (указателям). В исходном C коде Python везде стоят указатели PyObject*, но в самом Python работа с объектами соответствует работе со ссылками на объекты в C++.

SWIG тоже не различает указатели и ссылки, то есть если Ваша функция возвращает MyObject* или MyObject&, после импорта в Python результат будет один и тот же.

Если тип MyObject был определён в заголовочном файле, обработанным SWIG, то никаких проблем не будет — все поля и методы объекта, возвращённого по указателю, будут доступны. Проблемы неожиданно начнутся, если по указателю или ссылке возвращается экземпляр базового типа double, int и т.д.

Допустим, необходимо импортировать пользовательский класс, реализующий какой то хитро устроенный массив, и в этом классе есть оператор доступа к члену

```
T& operator[] ( const I& i );
```

где T и I соответствующие типы. Как уже отмечалось выше, для импорта этого оператора в Python классу придётся добавить два метода

```
void __setitem__( const I& i, const T& v ){ (*this)[i] = v; }
T    __getitem__( const I& i ){ return (*this)[i]; }
```

С методом __setitem__ все однозначно, но вот __getitem__ может возвращать как T так и T&.

В первом случае, если T относится к базовому типу, все работает. Напротив, если T относиться к некоторому пользовательскому классу, то возвращённое значение можно будет только читать — все изменения, вносимые в возвращённое значение на элементе массива естественно никак не отобразятся, поскольку возвращается копия объекта.

Во втором случае (при возврате T&), все работает с пользовательскими типами, поскольку объект возвращается по ссылке и доступны поля и методы именно того элемента, к которому производился доступ. Но с базовыми типами в Python ничего сделать не удается — SWIG заворачивает объект как нечто специфическое «свиговое», которое нельзя ни складывать ни перемножать, хотя можно передавать как аргументы в функции, принимающие соответствующие указатели.

Конечно, можно правильно выбирать возвращаемый тип (отличие в одном символе '&' в исходном C++ коде), и все будет прекрасно работать. Проблемы начинаются, когда класс параметризован по типу — благодаря различию в один символ придётся создавать два варианта шаблона, что уже весьма накладно.

Конечно, SWIG умеет разадресовывать «свиговые» ссылки и указатели, и об этом даже написано в документации. Во первых, в .i файле нужно создать класс соответствующего указателя (пример для типа int):

[3] Если переменная объявлена в mymodule.hpp без extern и этот файл включается куда то кроме mymodule_wrap.cxx, то линкер Вас не поймёт тем более.

```
%include "cpointer.i"
%pointer_class( int, int_ptr )
```

Команда `%poniter_class( type, name )` создает класс с именем `name` являющийся «свиговой» обёрткой для `type*` (взято непосредственно из документации по `SWIG` версии 1.3, раздел 8.2.1, `http://www.swig.org/Doc1.3/Library`

```
struct name {
   name();                           // Create pointer object
  ~name();                           // Delete pointer object
   void assign(type value);          // Assign value
   type value();                     // Get value
   type *cast();                     // Cast the pointer to original type
   static name *frompointer(type *);  // Create class wrapper from existing
};
```

Альтернативным вариантом является

```
%include "cpointer.i"
%pointer_functions( int, int_ptr )
```

Команда `%poniter_funcs( type, name )` создает пять функций

```
type *new_name();                      // создает новый объект в куче
type *copy_name(type x);               // создает новый объект в куче и
                                       // копирует в него значение x
void  delete_name(type *x);            // освобождает память по указателю
void  name_assign(type *x, int value); // копирует *x = value
type  name_value(type *x);             // разадресует указатель
```

Тогда, доступ к элементу `i` экземпляра массива с именем `arr` в `Python` будет иметь вид

```
int_ptr_frompointer( arr[i] ).value()
```

или

```
int_ptr.frompointer( arr[i] ).value()
```

или

```
int_ptr_value( arr[i] )
```

Но каждый раз писать в `Python` такие вещи несколько громоздко, поэтому мы можем подправить метод `__getitem__` в `.py` файле, естественно через `.i` файл (пусть тип массива `myarray`):

```
%pythoncode %{
_myarray_old_getitem = myarray.__getitem__
myarray.__getitem__ = lambda self, i : int_ptr_value( _myarray_old_getitem( self, i ) )
%}
```

В итоге, доступ в `Python` будет иметь вполне традиционный вид, но для этого пришлось вносить согласованные изменения сразу на трёх уровнях (создавать метод `__getitem__` в `C++`, создавать `SWIG` оболочку для `int*` в `.i` файле и подправлять метод `__getitem__` через `.i` файл в Питоне).

Тот же результат можно получить при помощи директивы `%typemap`. Если в `.i` файле указать

```
%typemap(out) int& %{
   $result = PyInt_FromLong( *$1 );
%}
```

то все методы, возвращавшие int&, после импорта будут возвращать объект Python типа int. Здесь $result — то, что будет возвращено методом в Python, $1 — результат выполнения пользовательского метода.

Такой вариант выглядит компактнее, чем создание объекта int_ptr и изменение через Python метода myarray.__getitem__, но это может быть не совсем то, чего Вы хотите — возможно некоторые методы все же должны возвращать int& (например для дальнейшего использования адреса результата). Директива %typemap позволяет настроить преобразование типа только для метода myarray.__getitem__

```
%typemap(out) int& myarray::__getitem__ %{
   $result = PyInt_FromLong( *$1 );
%}
```

4.3 Массивы

В C++ иногда удобно принимать/возвращать массив (то есть указатель на некоторый участок памяти, в котором друг за другом лежит известное количество элементов заданного типа). Поскольку SWIG указатели заворачивает во что то своё, то передать из Python такой массив в функцию ещё можно, но не более того. Для нормальной работы с массивами типа type в .i файле необходимо указать

```
%include "carrays.i"
%array_class( type, name )
```

что приведет к созданию класса с именем name являющийся «свиговой» обёрткой для массива из элементов типа type (взято непосредственно из документации по SWIG версии 1.3, раздел 8.2.2, http://www.swig.org/Doc1.3/Library.h

```
struct name {
   name(int nelements);                  // Create an array
  ~name();                               // Delete array
   type getitem(int index);              // Return item
   void setitem(int index, type value);  // Set item
   type *cast();                         // Cast to original type
   static name *frompointer(type *);     // Create class wrapper from
                                         // existing pointer
};
```

Аналогично, команда

```
%array_functions( type, name )
```

приведет к созданию функций

```
type *new_name(int n);                       // создает новый массив длиной n
type *delete_name(type *arr);                // удаляет массив
type name_getitem(type *arr, int i);         // возвращает arr[i]
void name_setitem(type *arr, int i, type v); // arr[i] = v
```

Пример использования — пусть есть C++ функция, выводящая на печать содержимое массива типа double

```
void double_arr_out( double* p, int N ){
   for( int i=0; i<N; i++ ) printf( "%f\n", p[i] );
}
```

Объявляем в .i файле массив

```
%array_class( double, double_arr )
```

И после импорта в Python можем

```
>>> from mymodule import *
>>> arr = double_arr(10)
>>> for i in range(10) : arr.setitem( i, .5**i )
>>> double_arr_out( p, 10 )
1.000000
0.500000
0.250000
0.125000
0.062500
0.031250
0.015625
0.007812
0.003906
0.001953
```

Директива %typemap позволяет создать более изящный вариант в стиле Python (это в C необходимо передавать размер массива отдельным аргументом, а в Python в качестве массивов используется объекты высокого уровня, которые «знают» размер). Будем в качестве массива передавать из Python в функцию double_arr_out список чисел с плавающей точкой:

```
%typemap(in) (double *p, int N) {
   if (PyList_Check($input)) {
      $2 = PyList_Size($input);
      $1 = new double[$2];
      for( int i = 0; i < $2; i++){
         PyObject *o = PyList_GetItem($input,i);
         if( PyFloat_Check(o) ) $1[i] = PyFloat_AsDouble(o);
         else{
            PyErr_SetString(PyExc_TypeError,"list must contain doubles");
            delete [] $1;
            return NULL;
         }
      }
   }else{
      PyErr_SetString(PyExc_TypeError,"not a list");
      return NULL;
   }
}

%typemap(freearg) (double *p, int N) {
   delete [] $1;
}
```

Тогда в Python

```
>>> from module import *
>>> double_arr_out( [ .5**i for i in range(10) ] )
1.000000
0.500000
0.250000
0.125000
0.062500
0.031250
0.015625
0.007812
0.003906
```

```
0.001953
```

В Python работа становиться несколько более комфортной, но за это приходиться платить написанием довольно развесистого кода в .i файле.

4.4 Генерация исключений для Python

Исключения в Python используются повсеместно. Например, преобразование произвольной последовательности L (объекта, имеющего специальный метод __getitem__) к кортежу при помощи функции tuple(L) производиться извлечением элементов последовательности одного за другим вплоть до генерации исключения IndexError, что воспринимается как конец последовательности.

Таким образом, создание методов C++ классов со специальными именами Python, описанное в разделе 3.2, должно в некоторых случаях сопровождаться возбуждением исключений в Python.

Для этого, во первых, требуется установить пользовательский перехват исключений в ..._wrap.cxx файле, для чего необходимо в .i файле указать

```
%exception { try{ $action }catch(...){ return NULL; } }
```

При генерации ..._wrap.cxx файла SWIG на место $action подставит вызов пользовательской функции. Такой код при возбуждении любого исключения приведет к возврату NULL вместо объекта Python, созданного на основе результатов вызова пользовательской функции, что будет воспринято интерпретатором Python как возбуждение исключительной ситуации. Директива %exception должна указываться в .i файле после задания имени модуля, но до включения обрабатываемых заголовочных файлов.

Подробности (тип исключения и его содержание) должны быть установлены в C++ коде до возбуждения исключительной ситуации при помощи функции

```
PyErr_SetString( PyObject* ErrClass, const char* ErrMessage )
```

Например

```
double myarray::__getitem__( int i ){
   if( i<0 || i>=N ){ // если индекс меньше нуля или больше размера массива
      PyErr_SetString( PyExc_IndexError, "index out of range" );
      throw "";
   }
   return p[i];
}
```

что позволит в Python приводить экземпляры класса myarray к кортежу при помощи стандартного конструктора кортежа tuple().

5 Сериализация объектов

5.1 Общие замечания

Под сериализацией понимается взаимно однозначное преобразование объекта в строку. В Python для сериализации используется модуль pickle (или его аналог cPickle написанный на C). Сериализация является очень мощным инструментом, поскольку избавляет от необходимости придумывать специфические форматы для хранения или передачи данных по сети — произвольные данные обрабатываются единообразно.

Попытка сериализации по умолчанию пользовательского класса, импортированного при помощи SWIG, не даст нужного результата — поля класса, созданного в .py файле, находятся не в словаре класса __dict__, а эмулируются при помощи специальных методов __getattr__/ __setattr__ (см. [1]). Сами поля находятся в скомпилированном C++ модуле, а доступ к ним осуществляется через таблицы класса __swig_setmethods__/ __swig_getmethods__, содержащие имена полей и функции доступа.

Для сериализации можно определить методы

```
__getstate__( self )
__setstate__( self, state )
```

Метод __getstate__ должен возвращать некоторый объект, который сериализуется. Метод __setstate__ принимает сериализованный ранее объект (возвращённый методом __getstate__) и должен настраивать self соответствующим образом.

Обратите внимание — метод __setstate__ не вызывает конструктор класса. Для обычного класса Python это как правило не важно (если состояние класса полностью определяется его полями в словаре класса __dict__), но для C++ класса необходимо произвести некоторые действия по инициализации.

При восстановлении класса, при необходимости автоматически подгружается модуль, в котором класс был определён, что может приводить к выполнению каких то побочных действий, поэтому следует разносить описание классов/функций и их использование по разным модулям. Модуль определяется по имени, которое сохраняется при сериализации, — если модуль был изменён без сохранения обратной совместимости последствия будут непредсказуемыми.

Описанные выше объекты SWIG (указатели и массивы) не сериализуются.

5.2 Сериализация произвольного не содержащего указателей C++ класса с открытыми полями и конструктором без аргументов

Для классов, имеющих только открытые (объявленные как public) поля, конструктор по умолчанию и не содержащих указателей, которые должны быть проинициализированы актуальными адресами, методы __setstate__/__gets
могут быть заданны единообразно. Можно определить эти методы в отдельном модуле и подключать их в .py файле, можно определить их непосредственно в .py файле. В простейшем случае это выглядит следующим образом:

```
def _sp_getstate( self ):
    state = {}
    for k in self.__swig_getmethods__.keys() : state[k] = getattr( self, k )
    return state

def _sp_setstate( self, state ):
    if not hasattr( self, 'this' ) : self.__init__()
    for k, v in state.items() :
        if self.__swig_setmethods__.has_key( k ) : setattr( self, k, v )

...

myclass.__setstate__, myclass.__getstate__ = _sp_setstate, _sp_getstate
```

Но при таком подходе должен существовать конструктор по умолчанию (без аргументов). Конечно, функции _sp_getstate/ _sp_setstate можно усложнить и вызывать конструктор с ранее сохранёнными аргументами, но в общем случае это выглядит слишком громоздко и необходимо учитывать специфику задачи. Кроме вызова конструктора при сериализации можно обеспечить вызов различных методов класса, выделение памяти, чтение/запись файлов и т.д.

Если поля класса myclass являются пользовательскими классами которые сериализуются и имеют конструкторы копирования, то класс myclass так же сериализуется.

5.3 Сериализация на уровне определения методов __getstate__/__setstate__ в C++ коде

Принципиально другим подходом является определение методов __getstate__/__setstate__ в C++ коде. При этом метод __getstate__ должен возвращать сериализуемое значение, которое должно корректно обрабатываться методом __getstate__. Можно возвращать значения базовых типов, либо объекты Python (строки, кортежи, списки, словари и т.д.). Строки Python не обязательно должны содержать только текстовые данные, можно создать строку на основе массива явно указав его длину в байтах — тогда нулевые байты будут включены в строку.

Например, сериализацию массива можно организовать следующим образом

```
class my_double_array{
   double* p; int N;    // содержимое массива и его размер
public:
      ...
   void init( int AN ); // инициализация, устанавливает N=AN и выделяет память
      ...
   PyObject* __getstate__(){
      return PyString_FromStringAndSize( (const char*)p, sizeof(double)*N );
   }
   void _C_setstate( PyObject* state ){
      init( PyString_Size( state ) );
      double* Ap = (double*)PyString_AsString( state );
      for( int i=0; i<N; i++ ) p[i] = Ap[i];
   }
      ...
};
```

Тем не менее, необходимо обеспечить вызов конструктора — именно поэтому в C++ коде создан метод `_C_setstate` а не `__setstate__`. В `.py` файле необходимо указать

```
def _sp_C_setstate( self, state ):
    if not hasattr( self, 'this' ) : self.__init__()
    self._C_setstate( state )
   ...
my_double_array.__setstate__ = _sp_C_setstate
```

6 Шаблоны классов и функций

Про шаблоны в C++ написано довольно много. Здесь мы рассмотрим, как импортировать в Python параметризованные классы и функции.

Поскольку C++ код должен быть скомпилирован перед импортом в Python, все необходимые реализации шаблонов должны быть инстанцированы. Как для классов, так и для функций, инстанцирование можно осуществить в `.i` файле при помощи команды SWIG

```
   %template( name ) type;
```

где `name` — имя инстанцированного шаблона, под которым он будет виден в Python, `type` — инстацируемый тип, где указанны все необходимые параметры шаблона (если значения заданных по умолчанию параметров шаблона актуальны, их можно не указывать).

Например, пусть у нас есть C++ код

```
template <class T, int D=0> class myclass{
   ...
};
template <class T> void myfunc( T x ){
   ...
}
```

После указания в `.i` файле

```
   %template( myclass_double1 ) myclass<double,1>;
   %template( myclass_int  ) myclass<int>;
   %template( myfunc ) myfunc<double>;
   %template( myfunc ) myfunc<int>;
```

в Python появятся классы `myclass_double1` соответствующий `myclass<double,1>`, `myclass_int` соответствующий `myclass<int,0>` и две перегруженные функции `myfunc` соответствующие

```
void myfunc<double> ( double x );
void myfunc<int> ( int x );
```

Шаблоны позволяют создавать семейства классов и функций, но следует соблюдать некоторую осторожность — импорт многочисленных реализаций в Python приводит к раздуванию файла ..._wrap.cxx и может существенно увеличивать время компиляции.

7 Универсальный Makefile для импорта C++ кода в Python

Конечно, SWIG существенно упрощает импорт C++ кода в Python, но все же необходимо вручную писать Makefile и .i файл, что неизбежно приводит к ошибкам. Между тем, возможно создать Makefile достаточно общего вида, автоматически генерирующий .i файлы и обеспечивающий сборку разнообразных приложений. Для начала напишем библиотеку, которая при включении в пользовательский Makefile будет создавать необходимые правила на основе заданных ранее пользователем переменных make:

```
file /usr/include/cpp2py.mk:
#  make library for import C++ code to Python. (c) A.V. Ivanov, Aug 2010, Moscow
#-------------------------------------------------------------------------------
CXX=g++ -fPIC          # compiler
LD=g++ -shared         # linker
PY=/usr/include/python # path to Python.h
catchall=yes           # catch all C++ exception, [yes/no]
unref=bool char short int long float double # convert 'type& --> type' for all return values
#-------------------------------------------------------------------------------
# name     --- main target and module name
# headers  --- user's headers for SWIG (with extentions)
# modules  --- user's modules for compile and link (with extentions)
# objects  --- user's object files for linking (need make by users self)
# istart   --- user's .i-file to include at begin SWIG file "$(name).i"
# ifinish  --- user's .i-file to append to end SWIG file "$(name).i"
# pickling --- user's objects for pickling
# setstate --- user's objects with __C_setstate__ methods
# CXXOPT   --- compile options
# LINKOPT  --- linker options
# SWIGOPT  --- SWIG options
#-------------------------------------------------------------------------------
#   main target
#-------------------------------------------------------------------------------
$(name) : _$(name).so $(name).py;
#-------------------------------------------------------------------------------
#   start swig
#-------------------------------------------------------------------------------
$(name).py $(name)_wrap.cxx : $(name).i $(headers)
        swig $(SWIGOPT) -python -c++ $(name).i
#-------------------------------------------------------------------------------
#   link shared library
#-------------------------------------------------------------------------------
_$(name).so : $(name)_wrap.o $(addsuffix .o,$(basename $(modules))) $(objects)
        $(LD) $(LINKOPT) -o $@ $(name)_wrap.o $(addsuffix .o,$(basename $(modules))) $(objects)
#-------------------------------------------------------------------------------
#   compile object files
#-------------------------------------------------------------------------------
define compile
$(CXX) $(CXXOPT) -I$(PY) -c $m
```

```
endef
$(foreach m,$(modules),$(subst \,,$(shell g++ -DPYTHON -I$(PY) $(CXXOPT) -M $m)); $(compile))
#--------------------------------------------------------------------------------
$(subst \,,$(shell if [ -f $(name)_wrap.cxx ]; then g++ -M -I$(PY) $(CXXOPT) $(name)_wrap.cxx;
else echo $(name)_wrap.o : $(name)_wrap.cxx $(headers); fi))
        $(CXX) -I$(PY) -c $(name)_wrap.cxx
#--------------------------------------------------------------------------------
#   make .i file
#--------------------------------------------------------------------------------
$(name).i : $(lastword $(MAKEFILE_LIST)) $(istart) $(ifinish)
        echo "%module $(name)" > $@
        if [ -f "$(istart)" ] ; then cat $(istart) >> $@ ; fi
ifeq ($(catchall),yes)
        echo "%exception { try{ \$$action }catch( const char *e ){ PyErr_SetString(
PyExc_RuntimeError, e ); return NULL; }catch(...){ return NULL; } }" >> $@
endif
ifneq ($(setstate),)
        echo "%pythoncode %{def _sp_C_setstate( self, state ):" >> $@
        echo "        if not hasattr( self, 'this' ) : self.__init__()" >> $@
        echo "        self.__C_setstate__(state)" >> $@
        echo "%}" >> $@
endif
ifneq ($(pickling),)
        echo "%pythoncode %{def _sp_setstate( self, state ):" >> $@
        echo "        if not hasattr( self, 'this' ) : self.__init__()" >> $@
        echo "        for k, v in state.items() :" >> $@
        echo "           if self.__swig_setmethods__.has_key( k ) : setattr( self, k, v )" >> $@
        echo "%}" >> $@
endif
        if [ "$(findstring bool,$(unref))"   ]; then echo "%typemap(out) bool&   %{ \$$result=
PyBool_FromLong    ( *\$$1 ); %}" >> $@; fi
        if [ "$(findstring char,$(unref))"   ]; then echo "%typemap(out) char&   %{ \$$result=
PyInt_FromLong     ( *\$$1 ); %}" >> $@; fi
        if [ "$(findstring short,$(unref))"  ]; then echo "%typemap(out) short&  %{ \$$result=
PyInt_FromLong     ( *\$$1 ); %}" >> $@; fi
        if [ "$(findstring int,$(unref))"    ]; then echo "%typemap(out) int&    %{ \$$result=
PyInt_FromLong     ( *\$$1 ); %}" >> $@; fi
        if [ "$(findstring long,$(unref))"   ]; then echo "%typemap(out) long&   %{ \$$result=
PyInt_FromLong     ( *\$$1 ); %}" >> $@; fi
        if [ "$(findstring float,$(unref))"  ]; then echo "%typemap(out) float&  %{ \$$result=
PyFloat_FromDouble ( *\$$1 ); %}" >> $@; fi
        if [ "$(findstring double,$(unref))" ]; then echo "%typemap(out) double& %{ \$$result=
PyFloat_FromDouble ( *\$$1 ); %}" >> $@; fi
        echo "%{" >> $@
        for i in $(headers); do echo "#include \"$$i\"" >> $@ ; done
        echo "%}" >> $@
        for i in $(headers); do echo  "%include \"$$i\"" >> $@ ; done
        for i in $(setstate); do echo "%pythoncode %{$$i.__setstate__ = _sp_C_setstate%}" >> $@;
done
        for i in $(pickling); do echo "%pythoncode %{$$i.__setstate__, $$i.__getstate__ =
_sp_setstate, lambda self : dict([ (k,getattr( self, k )) for k in
self.__swig_getmethods__.keys() ])%}" >> $@; done
        if [ -f "$(ifinish)" ] ; then cat $(ifinish) >> $@ ; fi
#--------------------------------------------------------------------------------
```

```
clean:;  -rm $(name)_wrap.cxx $(addsuffix .o,$(basename $(modules))) $(objects) _$(name).so
$(name).py
cleani: clean; -rm $(name).i
#-----------------------------------------------------------------------------
```

Здесь используются следующие переменные:

- `name` — имя создаваемого модуля, утилита `make` при запуске создаст файл `$(name).i`, из которого `SWIG` создаст файлы `$(name).py` и `$(name)_wrap.cxx`, а после сборки будет создан файл `_$(name).so`.
- `headers` — список пользовательских заголовочных файлов, которые должны быть обработаны утилитой `swig`, вводятся через пробел, с расширениями.
- `modules` — список пользовательских модулей (файлов с кодом `C++`), которые должны быть скомпилированы и слинкованы в `_$(name).so`, вводятся через пробел, с расширениями, списки зависимостей для каждого модуля генерируются автоматически при помощи команды `g++ -M ...`
- `objects` — список пользовательских объектным файлов, которые должны быть слинкованы в `_$(name).so`, вводятся через пробел, с расширениями, правила для сборки каждого из этих файлов пользователь задает самостоятельно.
- `istart`, `ifinish` — пользовательские файлы с дополнительными директивами `SWIG`, содержимое которых будет включено сразу после задания имени модуля и добавлено в конец файла `$(name).i`.
- `catchall` — по умолчанию установлена как `yes`, добавляет в `$(name).i` директиву

  ```
  %exception { try{ \$$action }catch(...){ return NULL; } }
  ```

  см. раздел 4.4.
- `pickling` — список пользовательских классов, для которых должны быть установлены методы `__setstate__` и `__getstate__` с вызовом конструктора, работающие на основе сериализации полей класса, см. раздел 5.2.
- `setstate` — список пользовательских классов с методом `__C_setstate__(state)`, для которых должен быть установлен метод `__setstate__` с вызовом конструктора, см. раздел 5.3.
- `CXXOPT`, `LINKOPT` и `SWIGOPT` — дополнительные опции компилятора, линкера и утилиты `swig`.
- `PY` — путь к файлу `Python.h`, по умолчанию задала как `/usr/include/python`.

Допустим у нас есть три заголовочных файла `a1.hpp`, `a2.hpp`, b.h и два модуля `a.cpp`, `b.c`. Тогда для импорта классов и функций из `a1.hpp` и `b.h` в `Python` в виде модуля `ab` пользовательский `Makefile` будет иметь вид

```
name = ab
headers = a1.hpp b.h
modules = a1.cpp b.c
include cpp2py.mk
```

Модуль может быть собран командой

```
$ make ab
```

Файл `cpp2py.mk` должен лежать в доступном для включения утилитой `make` месте, в простейшем случае это текущая папка, или например директория `/usr/include`.

Пользовательский `Makefile` может содержать правила для сборки нескольких модулей для импорта в `Python` — для каждого модуля придется создать свой `make` файл (например `module1.mk`, `module2.mk` и т.д.), а общий `Makefile` будет иметь вид

```
module1 : ; $(MAKE) -f module1.mk
module2 : ; $(MAKE) -f module2.mk
...
```

Предложенный вариант библиотеки `cpp2py.mk` следует рассматривать как некоторый прототип, который можно (и нужно) расширять согласно специфике решаемых задач.

8 Взаимодействие нескольких модулей

Иногда создаваемое приложение содержит несколько модулей, которые импортируются и сообираются поотдельности и после сборки взаимодействуют между собой. В принципе это может быть небезопасно — необходимо заботиться о том, что бы разделяемые участки кода в точности совпадали и собирались с одними и теми же опциями компилятора, иначе последствия могут быть непредсказуемыми.

Тем не менее, такие ситуации встречаются. При этом иногда возникают странные на первый взгляд ошибки — при попытке запуска функции из одного модуля, с объектами созданными в другом модуле, `SWIG` сообщает о том, что не может найти подходящую функцию (если функция была перегружена), или что аргумент функции имеет неверный тип, хотя типы всех аргументов с Вашей точки зрения правильные. Такие ошибки как правило возникают, когда аргументы функции, созданные во втором модуле, относятся к параметризованному классу, заголовочный файл которого не был проимпортирован в первый модуль.

Дело в том, что при анализе заголовочных файлов `SWIG` создает во `_wrap.cxx` файле т.н. `TYPES TABLE` (таблицу используемых типов) на основе которой при вызове функции происходит проверка типов аргументов. Если заголовочный файл не был проанализирован, упоминания о содержащихся в нем классах тем не менее могут попасть в таблицу типов, если они встречаются в аргументах функций того заголовочного файла который был проанализирован (был упомянут в `.i` файле при помощи директивы `%include`). При этом обычные, непараметризованные классы, попадают в таблицу типов и в дальнейшем обрабатываются без проблем.

Параметризованные классы `SWIG` обрабатывает специфическим образом, и в подобной ситуации параметризованные классы не попадают в таблицу типов со всеми вытекающими последствиями. Для решения этой проблемы необходимо включать заголовочные файлы с параметризованными классами в `.i` файл при помощи директивы `%include`, даже если в данном модуле классы не будут инстанцироваться. При этом параметризованные классы попадают в таблицу типов и экземпляры таких классов, созданные в других модулях, могут использоваться как аргументы функции данного модуля. Если инстанцирование параметризованных классов не производилось, то параметризованные классы отмечаются во `_wrap.cxx` файле лишь в таблице типов.

Список литературы

[1] Г. Россум, Ф.Л.Дж. Дрейк, Д.С. Откидач. «Язык программирования `Python`». 2001 — 454С.

[2] Марк Лутц. Программирование на `Python`. С-Пб.: «Символ». 2002 — 1135С.

[3] Ричард Столлман, Роланд МакГрат. «`GNU Make`. Программа управления компиляцией» 1995.