C++ ABI

Мурашко И. В.

Введение

По роду своей деятельности мы часто сталкиваемся с библиотеками сторонних разработчиков которые предоставляются нам в бинарной форме, например Oracle OCCI. Вместе с тем в разработке мы используем разные версии компиляторов gcc, которые не всегда совместимы по умолчанию с используемыми бинарными библиотеками (Oracle OCCI). В данном докладе детально описываются соответствующие проблемы и способы их решения. Особое внимание уделяется изменениям привнесенным gcc ver. 5.1 в libstdc++.

1 API vs ABI

Когда кто-то собирается использовать внешнюю библиотеку, то первое что интересует - какой интерфейс она предоставляет. Этот интерфейс мы называем API (Application Program Interface). Кроме этого интерфейса значение имеет двоично представление данных библиотеки, которое называется ABI (Application Binary Interface).

API имеет важное значение в момент компиляции исходных кодов, т.е. ошибки использования API проявляются в момент компиляции.

ABI играет роль при сопряжении уже скомпилированных объектных файлов, то есть на момент компоновки (link). При этом стоит отметить, что диагностика ошибок на этапе компиляции более продвинутая чем на этапе компоновки, что позволяет решать проблемы использования API намного эффективнее, чем похожие проблемы использования ABI.

2 gcc5.1 ABI

Версия 4.8.1 компилятора gcc была одной из первых, которая полностью поддерживала новый стандарт c++11. Вместе с тем новые конструкции этого стандарта требовали изменений в двоичном представлении некоторых объектов STL, которые были сделаны в только в gcc 5.1.

Рассмотрим следующий фрагмент

```
#include <iostream>
#include <list>
#include <vector>
#include <string>
#include <queue>
#include <deque>
```

```
#include <set>
#include <stack>
#include <map>

int main() {
  std::vector<int> v;
  std::cout << "std::vector: " << sizeof(v) << std::endl;
  std::queue<int> q;
  std::cout << "std::queue: " << sizeof(q) << std::endl;
  std::priority_queue<int> pq;
  std::cout << "std::priority_queue: " << sizeof(pq) << std::endl;
  std::deque<int> dq;
  std::cout << "std::deque: " << sizeof(dq) << std::endl;
  std::stack<int> st;
  std::cout << "std::stack: " << sizeof(st) << std::endl;
  std::set<int> s;
  std::cout << "std::set: " << sizeof(s) << std::endl;
  std::multiset<int> ms;
  std::cout << "std::multiset: " << sizeof(ms) << std::endl;
  std::map<int,int> m;
  std::cout << "std::map: " << sizeof(m) << std::endl;
  std::multimap<int,int> mm;
  std::cout << "std::multimap: " << sizeof(mm) << std::endl;
  std::list<int> l;
  std::cout << "std::list: " << sizeof(l) << std::endl;
  std::string str;
  std::cout << "std::string: " << sizeof(str) << std::endl;
  return 0;
}
```

Запущенный под разными версиями компилятора gcc он дает следующие результаты

Контейнер STL	gcc 4.9.4	gcc 5.5.0
std::vector	24	24
std::queue	80	80
std::priority_queue	32	32
std::deque	80	80
std::stack	80	80
std::set	48	48
std::multiset	48	48
std::map	48	48
std::multimap	48	48
std::list	16	24
std::string	8	32

Как видно двоичное представление как минимум двух контейнеров (std::list и std::string) различается между двумя этими версиями компиляторов.

Связано это прежде всего с тем, что libstdc++ GCC 5.1 ввела [1] новую имплементацию данных контейнеров. Если посмотреть на размер std::list в

gcc 4 то видно что размер равен удвоенному размеру указателя (16 bytes), что соответствует представлению std::list как двусвязанного списка: каждый элемент содержит указатели на предыдущий и следующий элементы. Такой способ хранения не соответствует стандарту C++11, который явно требует чтобы вычисление размера контейнеров, и в частности списка имело бы сложность $O(1)$ [2], что требует хранения дополнительных данных вместе с указателями на предыдущий и последующий элементы.

2.1 std::string

За изменением размера std::string скрываются более глобальные изменения нежели те, что были сделаны в std::list. Рассмотрим следующий фрагмент

```
#include <iostream>
#include <string>
struct Data {
  std::string s;
};
void set(Data& data){
  std::string s = "abc";
  std::cout << (void*)(s.data()) << std::endl;
  data.s = s;
}
int main() {
  Data data;
  set(data);
  std::cout << (void*)(data.s.data()) << std::endl;
  return 0;
}
```

GCC 5 дает вполне предсказуемый вывод: мы получаем копию строки со своими внутренними данными

```
$ ./str5
0x7ffd1b434d90
0x7ffd1b434de0
$
```

Другая ситуация GCC 4:

```
$ ./str4
0x127fe88
0x127fe88
$
```

Как видно несмотря на то, что мы взяли копию объекта, реально данные скопированы не были. GCC 4 использует COW (copy-on-write) методологию для имплементации строк и в, частности счетчик (reference count) для отслеживания ссылок на используемые объекты. Таким образом, если объект копируется, то увеличивается счетчик ссылок. Реальное копирование объекта с выделением памяти происходит только при изменении данных. Такая реализация была лишь одной из возможных, и не смотря на

плюсы (скорость работы), имела и свои минусы связанные прежде всего с много-поточностью (необходимая синхронизация убивает все плюсы COW), в частности MS VS 2005 использовала другую методику, которая заключалась в реальном копировании данных.

Стандарт C++11 ввел move семантику, которая устранила основные минусы обычной реализации строк, с выделением памяти в момент создания. Таким образом в подавляющем большинстве случаев использование COW в данный момент неоправданно, что привело к прямому запрету использования COW для строк в стандарте C++11 [2].

3 Бинарные внешние библиотеки

В идеальном варианте внешняя библиотека поставляется в виде исходников по какой-то из свободных лицензий (GPL, MIT, etc.). В этом случае на целевой платформе может быть собрана бинарная версия библиотеки, которая будет совместима со всеми другими библиотеками собранными похожим образом.

В проблематичном случае бинарная библиотека собирается на некоторой платформе которая отличается от целевой, например Oracle OCCI (???) собирается с помощью компилятора gcc 4 и рассчитана на применение в системах RedHat Linux 7. В случае если целевая платформа подразумевает использование компилятора gcc 5.x вместе с другими библиотеками собранными этим компилятором (например boost) то возникнут проблемы на этапе компоновки, которые выглядят следующим образом

```
make tests
gcc -g -O0 -c main.cpp
docker run --rm -v "$PWD":/usr/src/myapp -w /usr/src/myapp gcc:4 \
gcc -shared -fPIC -o libtest4.so -c old.cpp
gcc -g -O0 main.o -o tests -lstdc++ -L. -ltest4
main.o: In function "main":
src/main.cpp:5: undefined reference to
"test::func(std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> >)"
collect2: error: ld returned 1 exit status
Makefile:31: recipe for target "tests" failed
make: *** [tests] Error 1
```

Для решения этих проблем gcc ABI предлагает реализацию контейнеров std::list и std::string удовлетворяющую требованиям стандарта c++11 в отдельном namespace, т.е. std::list<int> реально будет std::__cxx11::list<int>. Что позволяет использовать две версии ABI в одной программе. На уровне исходников выбор версии ABI выбирается с помощью следующего макроса

```
#define _GLIBCXX_USE_CXX11_ABI 0
#include <list>
...
// Old ABI be used
std::list<int> l;
...
```

По умолчанию используется (неявно) следующий вариант

```
#define _GLIBCXX_USE_CXX11_ABI 1
#include <list>
...
// New ABI be used
std::list<int> l;
...
```

Таким образом, если проекту требуется только одна внешняя библиотека с несовместимы интерфейсом которая недоступна в исходных кодах, то все остальные библиотеки, как и исходный код проекта должны быть собраны в режиме совместимости со старым ABI: следующий флаг должен быть добавлен в опции компилятора gcc:`-D_GLIBCXX_USE_CXX11_ABI=0`.

Сложнее ситуация когда имеется несколько несовместимых бинарных библиотек которые должны быть использованы в рамках одного проекта. В этом случае возможны варианты. Herb Sutter предложил [3] расширение языковых конструкций языка C++ в котором предполагается иметь две версии STL. Одна стандартная, которая доступна под стандартным namespace std, например std::string. В этой версии библиотеки хранятся самые последние версии реализации STL. При этом также существует стабильная версия, под namespace std::abi, например std::abi::string которая является переносимой с точки зрения бинарного интерфейса. Авторы внешних библиотек предлагают вариант своей библиотеки в режиме std::abi, что позволяет ее использовать с различными версиями компиляторов.

До тех пор пока предложение [3] еще не вступило в силу имеет смысл рассмотреть организацию взаимодействия с внешними библиотеками с помощью адаптеров, как это сделано в примерах ниже.

4 Примеры

Рассмотрим тестовую библиотеку, которая состоит из интерфейса (API): old.h

```
#include <string>

namespace test {
std::string func(std::string input);
}
```

и реализации

```
#include <string>

namespace test {

std::string func(std::string input) {
  std::string res(input.rbegin(), input.rend());
  return res;
}

} // namespace test
```

сборка осуществляется с помощью следующей команды

```
libtest4.so: old.cpp
        docker run --rm -v "$$PWD":/usr/src/myapp -w /usr/src/myapp gcc:4 \
        gcc -shared -fPIC -o libtest4.so -c old.cpp
```

Код который использует эту библиотеку выглядит следующим образом:
main.cpp

```
#include <iostream>
#include "old.h"

int main() {
  auto res = test::func("ABCD");
  std::cout << "Res:" << res << std::endl;
  res = test::func("ABCDEFG");
  std::cout << "Res:" << res << std::endl;
  res = test::func("ABCDEFGABCDEFGABCDEFGABCDEFGABCDEFGABCDEFGABCDEFGABCDEFGABCDEFGABCDEFG
  std::cout << "Res:" << res << std::endl;
  return 0;
}
```

Если использовать стандартные опции сборки

```
main.o: main.cpp
        gcc -g -O0 -c main.cpp

tests: main.o libtest4.so
        gcc -g -O0 main.o -o tests -lstdc++ -L. -ltest4
```

то получится следующий вывод об ошибке

```
make tests
gcc -g -O0 -c main.cpp
docker run --rm -v "$PWD":/usr/src/myapp -w /usr/src/myapp gcc:4 \
gcc -shared -fPIC -o libtest4.so -c old.cpp
gcc -g -O0 main.o -o tests -lstdc++ -L. -ltest4
main.o: In function "main":
src/main.cpp:5: undefined reference to
"test::func(std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> >)"
collect2: error: ld returned 1 exit status
Makefile:31: recipe for target "tests" failed
make: *** [tests] Error 1
```

Если перевести весь проект на старый ABI, т.е. использовать для сборки флаг `-D_GLIBCXX_USE_CXX11_ABI=0`, то Makefile будет выглядеть следующим образом

```
main.o: main.cpp
        gcc -D_GLIBCXX_USE_CXX11_ABI=0 -g -O0 -c main.cpp

tests: main.o libtest4.so
        gcc -g -O0 main.o -o tests -lstdc++ -L. -ltest4
```

и весь проект соберется успешно

```
make tests
gcc -D_GLIBCXX_USE_CXX11_ABI=0 -g -O0 -c main.cpp
docker run --rm -v "$PWD":/usr/src/myapp -w /usr/src/myapp gcc:4 \
gcc -shared -fPIC -o libtest4.so -c old.cpp
gcc -g -O0 main.o -o tests -lstdc++ -L. -ltest4
```

В качестве альтернативного варианта разработаем адаптер состоящий из следующих файлов - adapter.h:

```
#pragma once
namespace testadapter
{
const char* func( const char* input );
}
```

adapterold.cpp:

```
#define _GLIBCXX_USE_CXX11_ABI 0
#include "adapter.h"
#include "old.h"

namespace testadapter {
const char* func(const char* input){
  static thread_local std::string res;
  res = test::func(input);
  return res.c_str();
}
}
```

adapter.cpp:

```
#include <string>
#include "adapter.h"
namespace test {

std::string func(std::string input) {
  std::string res = testadapter::func(input.c_str());
  return res;
}

}
```

Сборка теперь осуществляется с помощью следующего Makefile:

```
adapter.o: adapter.cpp
        gcc -g -O0 -fPIC -o adapter.o -c adapter.cpp

adapterold.o: adapterold.cpp
        gcc -g -O0 -fPIC -o adapterold.o -c adapterold.cpp

libadapter.so: adapter.o adapterold.o
```

```
        gcc -shared -fPIC adapter.o adapterold.o -o libadapter.so

main.o: main.cpp
        gcc -g -O0 -c main.cpp

tests: main.o libtest4.so libadapter.so
        gcc -g -O0 main.o -o tests -lstdc++ -L. -ladapter -ltest4
```

Заключение

Несколько полезных советов по результатам этого исследования.

1. Пробуйте опции компилятора, такие как `-D_GLIBCXX_USE_CXX11_ABI=0`
2. Проектируйте использование библиотек через фасады, которые скрывают проблемные интерфейсы
3. Ждите реализации std::abi [3]

Ну, и на конец самый главный совет: избегайте использования библиотек, которые недоступны в исходных кодах.

Список литературы

[1] FSF. The gnu c++ library / FSF. — https://gcc.gnu.org/onlinedocs/libstdc++/manual/using_dual_abi.html.

[2] ISO. ISO/IEC 14882:2011 Information technology — Programming languages — C++ / ISO. — Geneva, Switzerland: International Organization for Standardization, 2012. — Feb. — P. 1338 (est.). — http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372.

[3] Sutter, H. Defining a portable c++ abi / Herb Sutter. — http://www.open-std.org/Jtc1/sc22/wg21/docs/papers/2014/n4028.pdf.