1 Пространства имён

Нередка такая ситуация, когда программа использует одновременно несколько библиотек. Каждая из библиотек вводит свою систему понятий и основанную на ней иерархию классов. Названия сущностей библиотеки выбираются таким образом, чтобы соответствовать назначению сущностей. Поскольку библиотеки, в общем случае, разрабатываются независимо друг от друга, нет никаких гарантий того, что названия сущностей разных библиотек не вступят в конфликт друг с другом.

Например, библиотека классов трёхмерного геометрического моделирования может предоставлять класс `Point` для представления точки в трёхмерном аффинном пространстве. В то же самое время графическая библиотека интерфейса пользователя может предоставлять класс `Point` для отображения точки в двухмерном окне на экране. В программе, которая одновременно использует эти две библиотеки, почти наверняка возникнут трудности с использованием класса `Point`.

Чтобы избежать таких проблем в библиотеках на **Си** часто используется механизм «префиксов». Например, точка в библиотеке трёхмерного моделирования может называться `L3D_Point`, а в библиотеке интерфейса пользователя — `GUI_Point`. При этом каждая из библиотек согласовано использует один и тот же префикс при именовании всех своих сущностей.

Язык **Си++** предоставляет более развитый механизм для разрешения подобного рода конфликтов — механизм пространства имён (namespaces). С использованием пространства имён гипотетический файл определения библиотеки трёхмерного моделирования `l3d.h` может принять следующий вид.

```
namespace L3D
{
  class Point { ... };
}
```

С другой стороны, заголовочный файл библиотеки интерфейса пользователя `gui.h` примет вид.

```
namespace GUI
{
  class Point { ... };
}
```

Теперь для доступа к классу `Point` геометрического моделирования мы можем писать `L3D::Point`, а для доступа к другому классу `Point` — `GUI::Point`.

Даже если программа целиком использует обе библиотеки одновременно, в программе наверняка существуют достаточно большие фрагменты (функции или даже целые классы), внутри которых используется класс `Point` только из одной библиотеки. Например, гипотетическая функция `make_projection` может использовать только класс `L3D::Point`. Очевидно, что каждый раз повторять внутри функции полное квалифицированное имя класса бессмысленно. Чтобы избежать этого в начале тела функции можно поместить директиву использования имени из пространства имён следующим образом:

```
void make_projection()
{
  using L3D::Point;
  //...
  Point z;
  L3D::Point y;
```

```
  //...
}
```

После директивы **using** соответствующий идентификатор может использоваться без идентификатора пространства имён. Таким образом, определения переменных z и y в примере выше эквиваленты. Директива **using** может использоваться на любом уровне вложенности: в блоке, в функции, в классе, в пространстве имён и глобально.

Если программа использует только одну библиотеку, то поимённое перечисление всех используемых имён из пространства имён становится бессмысленным, так как конфликтов с другими библиотеками всё равно не возникнет. В этом случае проще всего использовать всё пространство имён целиком с помощью директивы **using namespace**.

```
using namespace L3D;
```

Любой идентификатор из пространства имён L3D теперь может использоваться без квалификатора L3D.

Если используются несколько директив **using** могут возникать конфликты из-за неоднозначности привязки имени. Например, в случае, если программа содержит фрагмент

```
using namespace L3D;
using namespace GUI;
```

возникает неоднозначность в определении принадлежности идентификатора Point. Тогда использование идентификатора Point приведёт к возникновению ошибки компиляции. Необходимо записывать имя класса полностью: L3D::Point или GUI::Point. Если некоторый идентификатор присутствует только в одном пространстве имён, он может использоваться без явного указания пространства имён.

Таким образом, директива **using** не вводит новых имён сущностей, а позволяет сокращать имя за счёт опускания общего префикса. Директива **using** введена исключительно для удобства программиста.

Теперь посмотрим на пространства имён с точки зрения разработчика библиотеки. Как создавать пространства имён в своих программах? Пространства имён отличаются от классов следующими свойствами:

- Пространства имён — это абстракция исключительно периода компиляции программы. В работающей программе они не существуют, и им не соответствуют никакие объекты. Пространство имён можно понимать как удобный способ префиксирования всех сущностей.

- Пространство имён не может иметь сущностей, недоступных извне его (то есть отсутствуют аналоги квалификаторов доступа **private**, **protected**). Любой идентификатор, объявленный в пространстве имён, может использоваться вне его (естественно, если это допускается, например, правами доступа вложенного класса).

- Пространство имён *открыто*, то есть в любом месте программы можно добавить новое имя в уже существующее пространство имён. Следующий фрагмент программы является допустимым:

```
namespace A
{
  void func(int);
}
//...
namespace A
```

```
{
  void func(double);
}
```

В итоге пространство имён A содержит две функции func: одну с целым, другую с вещественным параметром.

При определении пространства имён у программиста есть альтернатива: размещать тела определяемых сущностей внутри пространства имён, или оставлять только заголовки, а тела выносить вовне. Например:

```
namespace N1
{
  class C1
  {
    int v;
  public:
    C1(int _v = 0) { v = _v; }
  };
}
```

или

```
namespace N2
{
  class C2;
}
class N2::C2
{
  int v;
public:
  C2(int _v = 0);
};
N2::C2::C2(int _v)
{
  v = _v;
}
```

Обратите внимание на полное имя конструктора класса: N2::C2::C2. Оно необходимо, так как класс C2 находится в пространстве имён N2, а метод C2 (конструктор) находится в классе C2.

2 Интерфейс с языком Си

Язык Си++ долгое время пытался обеспечивать совместимость с языком Си на уровне исходного текста. Но аспект совместимости на уровне исходного текста в настоящее время является несущественным. Гораздо важнее обеспечить совместимость языков на уровне компоновки программы. Использование объектных модулей и библиотек, написанных на Си++, из программы на языке Си проблематично, так как в Си отсутствуют многие понятия, присутствующие в языке Си++ (например, как использовать класс из программы на чистом Си?). Обратная задача — обеспечить в программе на Си++ использование библиотек, написанных на чистом Си, крайне важна, так как в настоящее время большинство системных библиотек всех операционных систем написано на чистом Си.

Проблема возникает на этапе компоновки программы. Компилятор языка **Си++**, как правило, использует стандартный компоновщик операционной системы, тот же компоновщик используется и компилятором **Си**. Для того, чтобы поддерживать пространства имён, классы, перегрузку имён функций компилятор **Си++** выполняет «кодирование» (mangling) имён сущностей таким образом, что в имени сущности в объектном файле оказывается закодированной вся необходимая информация. Например, конструктор класса `C2` из примера, представленного выше, в объектном файле будет иметь имя `_ZN2N22C2C2Ei`.

Тогда «наивная» попытка вызова, например, системного вызова `time` стандартной библиотеки языка **Си** из программы на **Си++** не увенчается успехом:

```
long time(long *t);
int main(void)
{
  long x = time(0);
  return 0;
}
```

Будет получена примерно следующая диагностика:

```
/tmp/ccw1h17q.o(.text+0x18): In function 'main':
/home/cher/4sem/cm.cpp:4: undefined reference to 'time(long*)'
collect2: ld returned 1 exit status
```

Если посмотреть в объектный файл, окажется, что компилятор закодировал имя `time` и параметры функции в идентификатор `_Z4timePl`. Поскольку компилятор **Си** не выполняет кодирования имён, и функция `time` в объектном файле сохранит свой идентификатор, возникает ошибка компоновки.

Чтобы избежать таких эффектов, функции, скомпилированные компилятором языка **Си** нужно явно помечать, например, следующим образом:

```
extern "C" long time(long *t);
int main(void)
{
  long x = time(0);
  return 0;
}
```

Класс памяти `extern "C"` может использоваться для одного объявления (как на примере выше), так и для нескольких объявлений и даже для заголовочных файлов.

```
extern "C"
{
  void f1(void);
  void f2(void);
}

extern "C"
{
#include "myheader.h"
}
```

Все заголовочные файлы стандартной библиотеки языка **Си** в настоящее время совместимы с компиляторами **Си++**. В них находится директива условной компиляции, которая в случае компиляции компилятором **Си++** включает директиву `extern "C"`. Ниже приведён пример файла `myheader.h`.

```
// защита от повторного включения
#ifndef __MYHEADER_H__
#define __MYHEADER_H__

// в случае Си++ включить режим Си
#ifdef __cplusplus
extern "C" {
#endif

void foo(void);
//...

// закрыть extern "C"
#ifdef __cplusplus
}
#endif

// закрыть защиту от повторного включения
#endif
```

3 Символьные строки

Перейдём к рассмотрению стандартной библиотеки языка **Си++**. Как правило, все классы и функции стандартной библиотеки определены в пространстве имён `std`. Поэтому чтобы использовать стандартную библиотеку нужно либо указывать префикс `std::` идентификаторов, либо использовать директиву **`using`**.

Рассмотрение начнём с класса, который обеспечивает работу с символьными строками. Язык **Си** предоставляет большое количество функций для работы со строками, но они все низкоуровневые, и программист должен сам заботится о выделении и освобождении необходимой памяти под строки, копировании строк и т. д., что приводит к большому количеству ошибок. Стандартная библиотека языка **Си++** включает класс `string`, который берёт на себя все низкоуровневые операции с памятью для строк. Чтобы использовать класс `string` в программе необходимо подключить заголовочный файл `<string>`. Например, это делает следующий фрагмент программы:

```
#include <string>
using namespace std;
```

Обратите внимание, что в имени заголовочного файла отсутствует суффикс `.h`. Все заголовочные файлы стандартной библиотеки **Си++** имеют имена без суффикса `.h`, а имена с этим суффиксом используются для обеспечения совместимости со старыми реализациями языка **Си++** и с языком **Си**.

Далее приводятся упрощённые определения конструкторов и членов класса `string`.

Класс `string` имеет следующие конструкторы:

```
// по умолчанию - пустая строка
string();
// литеральная строка или Си-строка
string(const char *);
// строка длины n из символа c
string(size_t n, char c);
```

```
// конструктор копирования
string(const string &);
// подстрока строки s от символа с индексом f, l символов
string(const string &s, size_t f, size_t l);
// n символов, начиная с p
string(const char *p, size_t n);
```

Символы в строке нумеруются, начиная с 0. Реализация класса `string` не использует специальный символ-терминатор `'\0'`, в классе явно хранится длина строки. Если индекс `f` является недопустимым, конструктор выбрасывает исключение. Если же длина выделяемой подстроки `l` недопустима, предполагается, что выделяются все символы до конца строки.

Методы доступа к элементам строки:

```
// оба метода возвращают длину строки
int length() const;
int size() const;

// неконтролируемый доступ к символам
char & operator [] (size_t);
// контролируемый доступ к символам
char & at(size_t);

// получить Си-строку
const char *c_str() const;
```

Метод `[]` предоставляет доступ к символам, составляющим строку. Метод возвращает ссылку на символ, а не значение, что позволяет использовать его как в правой, так и в левой части оператора присваивания. Например.

```
string s = "abcd";

s[2] = 'C'; // теперь s == "abCd"
s.at(3) = 'D'; // теперь s == "abCD"
```

Метод `[]` не проверяет индекс на допустимость. Запись или чтение за пределами памяти, выделенной под строку, приведёт к фатальным последствиям. Напротив, метод `at` выполняет такие проверки и в случае ошибки выбрасывает исключение.

Метод `c_str` возвращает указатель на начало строки, которая заканчивается терминатором `'\0'`, то есть преобразует строку в представление, используемое в языке **Си**. Обратите внимание, что класс `string` может содержать символы `'\0'` в середине строки. Такие строки не могут быть корректно обработаны большинством функций работы со строками языка **Си**.

Класс `string` содержит следующие методы для присваивания нового значения строке.

```
// операция присваивания с разными правыми частями
string &operator =(const string &);
string &operator =(const char *);
string &operator =(char);

// добавление к строке
string &operator += (const string &);
string &operator += (const char *);
string &operator += (char);
```

```
// другая форма операции добавления к строке
// (см. соответствующие конструкторы)
string &append(const string &);
string &append(const string &s, size_t f, size_t l);
string &append(const char *p);
string &append(const char *p, size_t n);
string &append(size_t n, char c);

// вставить в строку перед символом с индексом pos
string &insert(size_t pos, const string &);
string &insert(size_t pos, const string &s, size_t f, size_t l);
string &insert(size_t pos, const char *p);
string &insert(size_t pos, const char *p, size_t n);
string &insert(size_t pos, size_t n, char c);
```

Для класса `string` поддерживаются операции конкатенации + и операции сравнения строк ==, !=, <=, <, >=, >. Например, операции конкатенации имеют следующие прототипы:

```
friend string operator +(const string &s1, const string &s2);
friend string operator +(const string &s1, const char *p2);
friend string operator +(const string &s1, char c2);
friend string operator +(const char *p1, const string &s2);
friend string operator +(char c1, const string &s2);
```

Набор методов замены `replace` позволяет заменить символы, начиная с символа с индексом `pos` в количестве `z` символов (то есть символы с индексами `pos`, `pos + 1` ... `pos + z - 1`), на другую строку.

```
string &replace(size_t pos, size_t z, const string &);
string &replace(size_t pos, size_t z, const string &s,
                                      size_t f, size_t l);
string &replace(size_t pos, size_t z, const char *p);
string &replace(size_t pos, size_t z, const char *p, size_t n);
string &replace(size_t pos, size_t z, size_t n, char c);
```

Метод `substr` позволяет извлекать подстроку из строки.

```
string substr(size_t pos = 0, size_t n = (size_t) -1);
```

Кроме этого существует большое количество методов для поиска подстроки, выделения памяти и т. д.

4 Ввод/вывод в языке Си++

Стандартная библиотека языка предлагает ряд классов для обеспечения ввода/вывода. Программе в начале её работы уже доступны три стандартных потока: `cin` — стандартный ввод, `cout` — стандартный вывод, `cerr` — стандартный поток ошибок. Эти переменные становятся доступными при подключении заголовочного файла <iostream>. Они описаны в пространстве имён `std`.

Для вывода в поток используется перегруженный оператор <<, а для ввода из потока — перегруженный оператор >>. В стандартной библиотеке определены операторы вывода и ввода для всех стандартных типов, включая `char*` и `string`. Например,

```
#include <iostream>
using namespace std;

int main(void)
{
  string s;

  cout << "Enter string:";
  cin >> s;
  cout << "You typed: " << s << "!\n";
}
```

Для потоков ввода определены операции преобразования в тип `void*` и логического отрицания.

```
operator void *() const;
bool operator !() const;
```

Их можно использовать для проверки состояния потока.

```
char c;

while (cin >> c) cout << c;
```

В данном случае операция `cin >> c` вырабатывает результат типа `istream &`, который затем преобразовывается в `void*` с помощью оператора приведения типа. Оператор преобразования в `void*` возвращает нулевой указатель, если при чтении очередного символа произошла ошибка или был достигнут конец файла.

Для работы с файлами используются классы `fstream`, `ifstream` и `ofstream`, определённые в заголовочном файле `<fstream>`. Конструкторы этих классов позволяют задавать имя файла для открытия, а также режимы открытия. Кроме этого определён метод `open`. Для явного закрытия потока может использоваться метод `close`.

Пользователь может добавлять методы ввода из потока и вывода в поток для своих классов. Например, как можно определить метод для вывода в поток комплексных чисел (класс Complex).

```
ostream &operator <<(ostream &s, const Complex &z)
{
  s << '(' << z.re << ',' << z.im << ')';
  return s;
}
```