

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ РОССИЙСКОЙ ФЕДЕРАЦИИ
ГОУ ВПО САМАРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

От Си к Си++

*Учебное пособие по курсу «Языки программирования»
для студентов специальности «Компьютерная
безопасность»*

С.А.Бейлин

Самара
2009

УДК 917
ББК В 161.6
Б 88

Бейлин С.А.

Б 88 От Си к Си++. Учебное пособие для студентов :
– Самара : 2009. – 16с.

ISBN 978-5-467-00117-3

Данное учебное пособие ориентировано на студентов, имеющих навыки программирования на языке Си и начинающих изучать язык программирования Си++.

Учебное пособие предназначено для студентов специальности «Компьютерная безопасность» механико–математических факультетов классических университетов.

УДК 917
ББК В 161.6

ISBN 978-5-467-00117-3

©2008 С.А.Бейлин

Введение

Си++ — компилируемый, строго типизированный язык программирования общего назначения. Изначально возник как «Си с классами» [3], однако, кроме объектно-ориентированного программирования, поддерживает и другие парадигмы: унаследованную от Си императивную (процедурную), а также функциональную и обобщенную.

Создавая «Си с классами», а позднее Си++, Бьерн Страуструп старался сохранить универсальность, переносимость и быстродействие, присущие языку Си [3]. При этом практически все конструкции, общие для Си и Си++, имеют одинаковую, либо крайне близкую семантику. Большинство программ, написанных на Си, могут с минимальными изменениями быть откомпилированы Си++.

Си++ состоит из ядра языка и стандартной библиотеки, которые были стандартизированы совместным комитетом ANSI-ISO в 1998 г. (ISO/IEC 14882:1998 — Язык Программирования Си++), а исправленная версия стандарта Си++ выпущена в 2003 году.

Никто не обладает правами на язык Си++, он является свободным. При этом существует много коммерческих и свободных реализаций компиляторов Си++ для различных целевых платформ. Наиболее распространенные компиляторы Си++ — Intel® C++ Compiler; компилятор, входящий в состав Microsoft Visual C++; компилятор компании SUN Microsystems, входящий в состав Sun Studio; C++ Builder от Borland/CodeGear; и, разумеется, GNU C++ Compiler из состава GCC. Эти и другие компиляторы поддерживают различные версии стандартов Си++ и, кроме того, имеют свои расширения.

1. Си и Си++: такие похожие и такие разные

Рассмотрим простейшую программу, которая уже компилируется, но ничего полезного не делает:

```
1 main() {}
```

Данная программа скомпилируется и компилятором Си, и компилятором Си++, однако, с разными предупреждениями. Сначала скомпилируем программу компилятором Си:

```
$ gcc -Wall example_01.c
example_01.c:1: предупреждение: по умолчанию возвращаемый тип функции - 'int'
example_01.c: В функции 'main':
example_01.c:1: предупреждение: управление достигает конца не-void функции
```

а потом Си++:

```
$ g++ -Wall example_01.c
example_01.c:1: предупреждение: ISO C++ запрещает декларации 'main' без типа
```

Предупреждения, выданные в обоих случаях, достаточно подробны и говорят сами за себя.

Кстати, включение опции `-pedantic`¹ для компилятора Си++ приведет к тому, что программа скомпилирована не будет, а вышеуказанное предупреждение станет ошибкой:

```
g++ -Wall -pedantic example_01.c
example_01.c:1: ошибка: ISO C++ запрещает декларации 'main' без типа
```

Приведем теперь нашу программу к более приличному виду:

```
1 int main() {
2     return 0;
3 }
```

Теперь и компилятор Си, и Си++ компилируют нашу бесполезную программу без ошибок и предупреждений даже с опцией `-pedantic`.

Что-то подсказывает нам, что наша программа, скомпилированная тем и другим компилятором, должна работать одинаково, то есть одинаково почти ничего не делать: запускаться и тут же завершаться с кодом 0.

¹Режим более строгого соответствия стандартам. Подробнее см. `man 1 gcc`

Обратим, однако, внимание на одно маленькое отличие — размер двоичного файла. Для компилятора Си результирующий файл получился размером 10766 байт, тогда как для Си++ чуть-чуть больше, 10823 байта (разумеется, размеры зависят от аппаратной архитектуры, операционной системы, а также версии и параметров компилятора; полученные данные приведены для архитектуры x86_64, ОС Linux 2.6.27-9 SMP x86_64, и семейства компиляторов gcc 4.3.2 с параметрами, принятыми по умолчанию в Debian GNU/Linux).

Но заглянем еще несколько глубже, воспользовавшись утилитой `ldd`², которая выдаст нам список зависимостей от динамических библиотек. Вот вывод для компилятора Си:

```
$ ldd a.out
linux-vdso.so.1 => (0x00007fff2a5fe000)
libc.so.6 => /lib/libc.so.6 (0x00007f3d21ed5000)
/lib64/ld-linux-x86-64.so.2 (0x00007f3d22247000)
```

и вот для Си++:

```
$ ldd a.out
linux-vdso.so.1 => (0x00007fffad5ff000)
libstdc++.so.6 => /usr/lib/libstdc++.so.6 (0x00007f96a5030000)
libm.so.6 => /lib/libm.so.6 (0x00007f96a4dab000)
libgcc_s.so.1 => /lib/libgcc_s.so.1 (0x00007f96a4b93000)
libc.so.6 => /lib/libc.so.6 (0x00007f96a4821000)
/lib64/ld-linux-x86-64.so.2 (0x00007f96a533d000)
```

Мы видим, что (не считая `linux-vdso.so` и `ld-linux-x86-64.so`) к стандартной библиотеке Си `libc.so.6` компилятор Си++ добавил еще несколько, в том числе стандартную библиотеку Си++ `libstdc++.so.6`.

2. Си++ и стандартная библиотека Си

Итак, Си++, кроме стандартной библиотеки Си++, о которой речь впереди, содержит и стандартную библиотеку Си. Тому есть много объяснений, в том числе — обеспечение определенного уровня совместимости. Если в программе на Си++ возникает необходимость в использовании

²Подробнее смотрите `man 1 ldd`

функций из стандартной библиотеки Си, то вместо заголовочного файла `someheader.h` лучше использовать `csomeheader` — то есть без расширения и с буквой `c` в начале имени.

Например, наша любимая программа «Hello, World»

Listing 1: example_03.c

```
1 #include <stdio.h>
2
3 int main() {
4     printf("Hello, \uworld!\n");
5     return 0;
6 }
```

изменится так:

Listing 2: example_03.cpp

```
1 #include <cstdio>
2
3 int main() {
4     printf("Hello, \uworld!\n");
5     return 0;
6 }
```

Впрочем, и исходный вариант тоже будет работать³.

3. Ввод-вывод в Си++

Многие программисты на Си заслуженно критикуют семейство функций `printf` и `scanf`. В стандартной библиотеке Си++ появились более удобные механизмы для организации ввода-вывода. Вот как будет выглядеть наш «Hello, World»:

Listing 3: example_04.cpp

```
1 #include <iostream>
2
3 int main() {
4     std::cout << "Hello, \uworld!" << std::endl;
5     return 0;
6 }
```

³до поры до времени

```
6 }
```

Директива препроцессора `#include <iostream>` подключает заголовочный файл стандартной библиотеки Си++, в котором содержатся необходимые объявления. Строка `std::cout << "Hello, world!" << std::endl;` кажется понятной в общих чертах, но давайте обратим внимание на детали. Нетрудно догадаться, что `std::cout` — это стандартный поток вывода, то есть `stdout`. Оператор `<<`, знакомый нам как оператор побитового сдвига, в данном случае перегружен, и отправляет константную Си-строку `"Hello, world!"` в поток `std::cout`. Далее, в этот же поток добавляется `std::endl` — символ конца строки⁴.

Префикс `std::` означает, что идущий следом идентификатор находится в пространстве имен `std`. Впрочем, это пространство имен можно было бы включить «по умолчанию», и наша программа стала бы выглядеть вот так:

Listing 4: example_04_1.cpp

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main() {
6     cout << "Hello, \uworld!" << endl;
7     return 0;
8 }
```

В пространстве имен `std` находится много всего полезного из стандартной библиотеки Си++, поэтому обычно имеет смысл включать это пространство имен.

Вывод значений переменных при помощи потокового вывода в Си++ многим кажется более простым, чем использование форматной строки в функции `printf`. Вот простой пример:

Listing 5: example_05.cpp

```
1 #include <iostream>
2
3 using namespace std;
```

⁴В зависимости от платформы, это будет `\n`, `\r`, или `\r\n`

```
4
5 int main() {
6     int i = 7;
7     double d = 3.14;
8     cout << "i=" << i << ", d=" << d << endl;
9     return 0;
10 }
```

В результате выполнения данной программы будет выведена следующая строка:

```
|i=7, d=3.14
```

Аналогично можно и вводить данные со стандартного потока ввода, `stdin`, что иллюстрирует следующий пример:

Listing 6: example_06.cpp

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main() {
6     int i;
7     double d;
8     cin >> i >> d;
9     cout << "i=" << i << ", d=" << d << endl;
10    return 0;
11 }
```

Вот результат выполнения данной программы (числа в первой строке введены пользователем через пробел):

```
|4 6.28
|i=4, d=6.28
```

Но самое замечательное — впереди: ведь можно определить операторы `<<` и `>>` и для других типов, определенных программистом, и выводить их на экран таким простым и элегантным способом!

Задание 1

Скопируйте исходные коды лабораторных работ по Си в новые файлы с расширениями `.cpp`. Измените весь ввод-вывод, избавившись от функций `printf`, `scanf` и других, объявленных в `stdio.h`. Убедитесь, что все работает как и прежде. Используйте вызов компилятора `g++` с ключами `-Wall`, чтобы увидеть больше предупреждений.

4. Типы, переменные и управление памятью

Язык Си++ унаследовал от Си базовые типы, указатели, массивы, структуры, впрочем, внося несколько изменений и усовершенствований.

Впрочем, без нововведений не обошлось почти нигде. Вот как написал бы программист на Си лет 15 назад:

```
double dval;  
dval = 3.14;
```

— то есть, сначала переменная объявляется, а потом ей присваивается некоторое значение. Современная реализация Си позволяет объединить эти две различные операции — объявление переменной и присваивание ей начального значения — в одну:

```
double dval = 3.14;
```

С точки зрения Си++, запись может также выглядеть следующим образом:

```
double dval(3.14);
```

Это — *инициализация* переменной. Считается, что инициализация более предпочтительна, чем присваивание, даже для простейших, встроенных типов (а для многих типов, — единственно возможный вариант).

Структуры претерпели небольшие, но важные изменения. Если с точки зрения Си следующий фрагмент кода

```
struct {  
    double Re;  
    double Im;  
} x,y,z;
```

совершенно корректен и определяет три переменные `x`, `y`, `z` так называемой «анонимной структуры», то компилятор Си++ выведет как минимум предупреждение.

Далее, если у нас была определена «именованная» структура, например

```
struct complex {  
    double Re;  
    double Im;  
};
```

то для объявления переменной такого типа в Си необходимо было написать:

```
struct complex cvar;
```

то в Си++ ключевое слово `struct` не нужно, необходимо написать лишь

```
complex cvar;
```

— другими словами, структура `complex` с точки зрения Си++ — вполне самостоятельный тип.

А самое замечательное — теперь членами структуры могут быть функции⁵!

Выделение и высвобождение памяти. Что касается массивов и указателей (а и в Си++ это *почти* одно и то же), то изменения коснулись, в первую очередь, выделения и освобождения памяти.

⁵Разумеется, ведь с точки зрения Си++, структура — это частный случай класса, в котором все методы публичные!

Наконец-то в большинстве ситуаций можно забыть про функции `malloc` и `free` из стандартной библиотеки Си, и использовать вместо них операторы `new` и `delete`. Именно, вот такой код на Си

```
int* p = (int*)malloc(sizeof(int));  
...  
free(p);
```

станет в случае Си++ выглядеть так:

```
int* p = new int(5);    // заодно проинициализируем  
...  
delete p;
```

Аналогично в случае с динамическими массивами: код

```
int count = 10; // размер массива  
int p[] = (int*)malloc(sizeof(int) * count);  
...  
free(p);
```

станет в случае Си++ выглядеть так:

```
int count = 10; // размер массива  
int p[] = new int[count];  
...  
delete[] p;
```

— лишь пришлось при помощи квадратных скобок «напомнить» оператору `delete`, что придется высвободить память, выделенную массиву.

Впрочем, надо признать, что таким образом с массивами в Си++ работают достаточно редко.

Приведение типов. Язык Си++ поддерживает неявное и явное приведение типов. Типичный случай неявного приведения типа — использование целочисленных типов различной длины. Если в ходе такого преобразования может произойти потеря данных, компилятор выдает соответствующее предупреждение. Явное преобразование типов «в стиле Си», (новый_тип)значение, например `int* ip = (int*)malloc(1024)`, хотя и поддерживается компиляторами Си++ ради обратной совместимости с Си, но категорически не рекомендуется. В Си++ имеется четыре основных механизма приведения типов:

```
dynamic_cast <new_type> (expression)
reinterpret_cast <new_type> (expression)
static_cast <new_type> (expression)
const_cast <new_type> (expression)
```

Механизм `const_cast` предназначен для снятия ограничения, накладываемого спецификатором `const`; `static_cast` выполняет статическое приведение типов практически так же, как и стандартный механизм Си; `dynamic_cast` предназначен, в первую очередь, для приведения сложных типов в иерархии наследования; и наконец, `reinterpret_cast` — приводит что угодно к чему угодно, и нужны весьма весомые основания для применения этого механизма.

Приведенный чуть выше пример `int* ip = (int*)malloc(1024)` нам стоит переписать следующим образом:

```
int* ip = static_cast<int*>(malloc(1024))
```

(если, конечно, Вы все еще используете `malloc`).

NULL теперь на самом деле нуль. Уже привычный Вам «нулевой указатель» (“null pointer”), или просто `NULL`, на самом деле является макросом. Язык Си не регламентировал, как он должен быть определен, и это значение определялось по-разному в разных компиляторах. Например, компилятор `GNU C Compiler` определял его следующим образом:

```
#define NULL ((void *)0)
```

Однако, стандарт языка Си++ четко и однозначно определяет, что такое NULL:

```
#define NULL 0
```

Теперь это просто нуль, без приведения его к указателю `void*`. Кстати, Бьерн Страуструп рекомендует не использовать константу `NULL`, а использовать `0` [2]. Поступайте, как Вам больше нравится, но не забывайте — разыменовывание нулевого указателя является операцией с неопределенным поведением, и вызывает исключение, которое, если не обработано, приводит к подаче сигнала `SIGSEGV` («Segmentation fault»).

Задание 2

Доработайте старые лабораторные работы по Си: исправьте структуры таким образом, чтобы компилятор Си++ не выдавал предупреждений. Замените все выделение/высвобождение памяти при помощи функций `malloc` и `free` на операторы `new` и `delete`. Убедитесь, что все работает как и прежде. Используйте вызов компилятора `g++` с ключами `-Wall` и даже `-pedantic`, чтобы увидеть больше предупреждений.

5. Перегрузка операций

Одно из самых замечательных нововведений в Си++, причем относящееся не только к объектно-ориентированному подходу, — возможность перегрузки операций, а именно, функций и операторов. Суть перегрузки заключается в возможности одновременного существования в одной области видимости нескольких различных вариантов операции (оператора или функции), имеющих одно и то же имя, но различающихся типами параметров, к которым они применяются.

Вот характерный пример: предположим, мы хотим определить тип для работы с комплексными числами. Никаких проблем — мы объявим соответствующую структуру:

```
struct complex {  
    double Re;  
    double Im;  
};
```

Однако, мы хотели бы иметь возможность выполнять базовые операции с комплексными числами так, как мы привыкли это делать «на бумаге», при помощи операторов сложения, умножения, деления, присваивания, не помешала бы нам операция вычисления модуля и аргумента. Однако, вышеперечисленные операторы уже «заняты»; кстати, функцию вычисления модуля было бы неплохо назвать `abs`, также как функцию вычисления абсолютного значения для действительных чисел, объявленную в файле `cmath` как `double abs(double __x)`.

Cи++ дает нам возможность *перегрузить* функцию `abs`, чтобы она работала и с комплексными числами:

```
double abs(const complex& c) {  
    return sqrt(c.Re * c.Re + c.Im * c.Im);  
}
```

Какую из двух функций `abs` вызвать — решит компилятор, руководствуясь типом передаваемого функции аргумента.

Аналогичным образом можно перегрузить и стандартные операторы, например:

```
complex operator+(const complex& l, const complex& r) {  
    complex s;  
    s.Re = l.Re + r.Re;  
    s.Im = l.Im + r.Im;  
    return s;  
}
```

Теперь можно складывать комплексные числа привычным способом:

```
complex c1, c2, c3;  
.....  
c3 = c1 + c2;
```

Для удобства потокового вывода можно перегрузить оператор <<:

```
ostream& operator<<(ostream& os, const complex& c) {  
    os << .....;  
    return os;  
}
```

Задание 3

Реализуйте `operator<<` для описанной выше структуры `complex` (не забудьте подключить заголовочный файл `iostream` и задействовать пространство имен `std`). Реализуйте также операторы `+`, `-`, `*`, `/` для комплексных чисел. Подумайте, как сложить `complex` и `double`, и реализуйте соответствующие операторы. Если хватит терпения, вспомните и реализуйте какую-либо элементарную функцию комплексного переменного, например, показательную.

Список литературы

- [1] Стенли Б. Липпман, Жози Лажойе. «Язык программирования C++. Вводный курс»
- [2] Бьерн Страуструп. «Язык программирования C++»
- [3] Бьерн Страуструп. «Дизайн и эволюция языка C++»
- [4] Айра Пол. «Объектно-ориентированное программирование на C++»

Содержание

1. Си и C++: такие похожие и такие разные	4
2. Си++ и стандартная библиотека Си	5
3. Ввод-вывод в Си++	6
4. Типы, переменные и управление памятью	9
5. Перегрузка операций	13

Печатается в авторской редакции. Верстка в Л^AT_EX — С.А.Бейлин

Подписано в печать 13.02.2009

Гарнитура Computer Modern. Формат 60x84/16. Бумага офсетная. Печать оперативная.

Усл.-печ.л. 1. Уч.-изд.л. 0.75. Тираж 100 экз. Заказ №???

Издательство «Универс групп», 443011, Самара, ул.Академика Павлова, 1

Отпечатано ООО «Универс групп»