УДК 004.4'422

НЕКОТОРЫЕ СИТУАЦИИ НЕОДНОЗНАЧНОГО ВЫПОЛНЕНИЯ СТРОК КОДА КОМПИЛЯТОРАМИ C++

Дмитриев В.Л.
Стерлитамакский филиал Башкирского государственного университета
Российская Федерация, г. Стерлитамак

В представленной работе рассматриваются некоторые неоднозначности выполнения кода программы, написанной на языке программирования C++, при использовании различных компиляторов языка. Рассматриваются ряд примеров, дающих отличающиеся результаты для таких компиляторов, как Borland C++ 5.02, wxDev C++7.4, Microsoft Visual C++ 2005-2013 Express.

В большинстве случаев выбор компилятора обусловлен соображениями наибольшей популярности или удобства использования (а в некоторых случаях, возможно, и силой привычки). Одними из самых распространенных сред разработки программ на языке C++ являются компиляторы Borland C++ 5.02, wxDev C++ 7.4, Microsoft Visual C++. В некоторых случаях в зависимости от используемого компилятора можно получить прирост производительности до 10% и более (разумеется, в рамках решаемой задачи). Но сразу ответить на вопрос о том, какой компилятор выбрать, чтобы он сгенерировал код максимально правильно и быстро, к сожалению, невозможно. Дело в том, что одни компиляторы лучше оптимизируют виртуальные вызовы, другие – лучше работают с памятью. Кроме того, в некоторых, даже самых простых ситуациях, могут возникать разного рода неоднозначности сбора проекта (программы) из программного кода при использовании разных компиляторов [1]. Ниже приводятся некоторые из таких типичных ситуаций.

Рассмотрим вначале пример простой программы, в которой осуществляется присваивание значений элементам одномерного массива, состоящего из 4 целых чисел. Ниже показаны результаты выполнения программы, построенной отмеченными выше компиляторами.

Первый вариант фрагмента кода:

```
int i=1, m[4];
int main()
{  m[i]=(++i)++;
  for (int j=0; j<4; cout<<m[j++]<<"\t"); getch(); return 0;}
```

Второй вариант фрагмента кода:

```
int i=1, m[4];
int main()
{  m[i]=++i;
  for (int j=0; j<4; cout<<m[j++]<<"\t"); getch(); return 0;}
```

Третий вариант фрагмента кода:

```
int m[4];
int main()
{ int i=1; m[i]=++i;
  for (int j=0; j<4; cout<<m[j++]<<"\t"); getch(); return 0;}
```

Четвертый вариант фрагмента кода:

```
int m[4];
int main()
{ int i=1; m[i]=(++i)++;
  for (int j=0; j<4; cout<<m[j++]<<"\t"); getch(); return 0;}
```

Значения элементов массива после выполнения данных вариантов кода были следующими. Для компилятора wxDev C++ 7.4:

{0,0,2,0}, {0,2,0,0}, {0,0,2,0} и {0,0,2,0}.

Для компилятора Borland C++ 5.02:

{0,0,0,2}, {0,0,2,0}, {0,0,2,0} и {0,0,0,2}

Для компилятора Microsoft Visual C++ 2005 Express:

{0,0,2,0}, {0,0,2,0}, {0,0,2,0} и {0,0,2,0}

Как видно, результаты вычислений зависят от используемого компилятора – один и тот же код дает различающиеся результаты для разных компиляторов. Стоит, однако, отметить компилятор Microsoft Visual C++ 2005 Express, который дал ожидаемый по написанному коду результат для всех рассмотренных случаев.

Рассмотрим теперь примеры, демонстрирующие использование функций с переменным количеством параметров. Здесь также могут возникать неоднозначности. Как известно, количество параметров можно задать двумя способами [2]: передать это количество вызываемой функции явным образом через обязательный аргумент; просто договориться о некотором специальном признаке конца списка аргументов.

В обоих случаях доступ к аргументам осуществляется с использованием указателей. Рассмотрим реализацию первого способа на примере задачи о нахождении суммы заранее неизвестного количества элементов.

```
int summa1(int n, ...)
{ int s=0; int *a=&n;
  while (n--) s+=*(++a); return s;}
int main()
{ cout<<"Сумма элементов 1, 5, 7, 3: "<<summa1(4, 1, 5, 7, 3)<<endl;
  cout<<"Сумма элементов 4, 11, 5, 3, 4: "<<summa1(5, 4, 11, 5, 3, 4);
  getch(); return 0;}
```

Здесь используется указатель *a* для перебора суммируемых аргументов; его значение инициализируется адресом обязательного первого аргумента *n* (адресом начала списка аргументов) и затем в цикле наращивается для получения доступа к следующему аргументу.

Рассмотрим теперь реализацию второго способа на примере той же задачи.

```
double summa2(double x, ...)
{ double s=0.0; double *a=&x;
  while (*a) s+=*(a++); return s;}
int main()
{ cout<<"Сумма элементов 4.1,1.0,-5.1,7.3: "<<summa2(4.1,1.0,-5.1,7.3,0.0)<<endl;
  cout<<"Сумма элементов 5.2,4.3,11.1,5.1,3.1: "<<summa2(5.2,4.3,11.1,5.1,3.1,0.0);
  getch(); return 0;}
```

В этом случае цикл суммирования перебирает аргументы до тех пор, пока не встретит нулевой элемент, который должен быть обязательно использован при вызове функции summa2. Здесь стоит отметить, что разные компиляторы также дают для приведенного здесь кода (для второго способа реализации) разные результаты в зависимости от префиксной или постфиксной формы записи. Так, для цикла вида

```
while (*a) s+=*(a++);
```

компиляторы Borland C++ 5.02 и Microsoft Visual C++ 2005 Express дадут результат 7.3 и 28.8, а компилятор wxDev-C++ 7.4 – результат 15.5 и 39.2. Для цикла

```
while (*a) s+=*(++a);
```

компиляторы Borland C++ 5.02 и Microsoft Visual C++ 2005 Express дадут результат 3.2 и 23.6, а компилятор wxDev-C++ 7.4 – результат 7.3 и 28.8.

Первый способ реализации для используемых здесь компиляторов при этом дает одинаковый результат.

Результат работы компилятора wxDev-C++ 7.4, на первый взгляд, не совсем ясен. Также отмечу, что если при описании приведенной здесь функции summa2 тип double везде поменять на тип float, то программа будет работать некорректно.

Функцию summa1, реализующую первый способ передачи количества параметров, можно несколько видоизменить, чтобы она позволяла работать с аргументами типа double – измененный текст программы приведен ниже.

```
double summa1(int n, double k, ...)
{ double s=0. ;
  double *a=&k;
  while (n--) s+=*(a++);
  return s;}
int main()
{ cout<<"Сумма эл-в 1.15, 10.5, 2.7, 5.3: "<<summa1(4, 1.15, 10.5, 2.7, 5.3)<<endl;
  cout<<"Сумма эл-в 4, 1.11, 5.1, 2.23, 4.7: "<<summa1(5, 4., 1.11, 5.1, 2.23, 4.7);
  getch(); return 0;}
```

Представленный здесь код при использовании компилятора wxDev-C++ 7.4 снова дает неправильный результат. Очевидно, это связано с преобразованием типов и моделью памяти, принимаемым в данном компиляторе. Вот код того же примера, который будет нормально работать с компилятором wxDev-C++ 7.4:

```
long double summa1(int n, long double k, ...)
{ long double s=0. ;
  long double *a=&k;
  while (n--) s+=*a++;
  return s;}
int main()
{ cout<<"Сумма эл-в 1.15,10.5,2.7,5.3: "<<summa1(4,1.15L,10.5L,2.7L,5.3L)<<endl;
  cout<<"Сумма эл-в 4,1.11,5.1,2.23, 4.7: "<<summa1(5,4.L,1.11L,5.1L,2.23L,4.7L);
  getch(); return 0;}
```

Этот код также будет нормально работать и с компилятором Microsoft Visual C++ 2005 Express, однако с компилятором Borland C++ 5.02 будет приводить к ошибке.

Перейдем теперь к неоднозначностям, имеющим место при работе с потоками ввода-вывода. При вводе данных операция >> читает всё, начиная с первого символа (не являющегося символом-разделителем) до следующего символа, который не соответствует типу объекта назначения. Поэтому наиболее распространенной ошибкой при использовании потоков *istream* является ошибка, связанная с вводом данных, не соответствующих указанному формату. В таких случаях нужно или проверять состояние потока ввода перед использованием считанных данных, или использовать механизм исключений [3, 4]. Рассмотрим, например, следующий фрагмент кода:

```
float a, b;  cin>>a>>b;  cout<<a<<" "<<b;
```

Если при вводе данных указать, например, 1.234w, то значение переменной a будет равно 1.234, но значение переменной b будет неопределенно, т.к. сразу после ввода 1.234w программа перейдет к строке с инструкцией *cout*. Дело в том, что как только в потоке встретится символ w, являющийся некорректным с точки зрения ожидаемого формата вводимых данных, последним принятым символом будет 4. Символ w останется во входном потоке, и следующая операция >> начнет чтение с этой точки, что и приводит к неопределенности. Поэтому следующий фрагмент кода

```
float a; char b; cin>>a>>b; cout<<a<<" "<<b;
```

будет работать верно при вводе строки 1.234w.

Может случиться так, что самое первое введенное значение будет ошибочным с точки зрения формата. В этом случае операция >> оставит значение переменной без изменений и вернет значение 0 (false), что позволит проверить, отвечал ли ввод требованиям установленного формата данных для переменной. Ниже представлен фрагмент программы, в котором вводятся числа и ведется подсчет их суммы до тех пор, пока вводятся корректные значения для переменной.

```
float a, s=0.;
while(cin>>a) s+=a;
cout<<"\nЗначение переменной a: "<<a;
cout<<"\nСумма чисел: "<<s<<endl;
```

Пусть в процессе ввода часть чисел вводятся через нажатие клавиши "Enter", причем это будут корректные данные; затем, в следующей строке, вводится последовательность чисел через символ-разделитель (пробел):

1.37
2.45
1.23 4 13.4 2.2b 3.4 5.6

Результатом работы фрагмента программы будет:

Значение переменной a: 2.2
Сумма чисел: 24.65

Ввод *cin* буферизуется, поэтому третья строка значений, введенных с клавиатуры, не будет пересылаться программе до тех пор, пока не будет нажата клавиша "Enter" для каждой из выше расположенных строк. Цикл прекратит работу, как только встретится символ "*b*" (третья строка), так как он не соответствует формату вещественного числа: несоответствие символа "*b*" ожидаемому формату приводит к тому, что выражение *cin>>a* будет оценено как *false*.

Компилятор Borland C++ 5.02 и Microsoft Visual C++ 2005 Express, как и ожидалось, дали в примере выше для переменной a один и тот же результат: 2.2. Однако компилятор wxDev-C++ 7.4 сбрасывает значение переменной *a* в ноль. Последнее противоречит принципу, принятому в C++: если попытаться прочитать в некоторую переменную и операция не выполняется, значение переменной должно остаться неизменным.

Приведенные примеры показывают, что стабильных правильных результатов среди рассмотренных компиляторов можно добиться только при использовании компилятора Microsoft Visual C++ 2005 Express. Результаты, которые получаются с использованием компиляторов Borland C++ 5.02 и wxDev C++7.4, сильно зависят от содержания исходного кода программы.

Список литературы

1. Дмитриев В.Л. Теория и практика программирования на C++. – Стерлитамак: РИО СФ БашГУ, 2013. – 308 с.
2. Дмитриев В.Л. Теория и практика решения задач по программированию. Учебное пособие. Часть 1. – Уфа: РИЦ БашГУ, 2007. – 264 с.
3. Страуструп Б. Язык программирования C++. Специальное издание. – М.: Бином, 2004. – 1054 с.
4. Stroustrup Bjarne. The C++ programming language / Bjarne Stroustrup. – Fourth edition. – Boston: Addison-Wesley, 2013. – 1368 p.