Базовые понятия

Шокуров Антон В.
shokurov.anton.v@yandex.ru
http://машинноезрение.рф

10 сентября 2017 г.

Версия: 0.10

Аннотация

В данной заметке будут введены базовые понятия языка C++: полиморфизм, ссылка, константные выражения. На них основан язык. Без них он не был бы столь эффективно выглядящем и популярным.

Цель. Изучить базовые понятия языка C++: полиморфизм, ссылка, константные выражения.

Предварительный вариант!

1 Базовые понятия

В данной заметки будут введены базовые понятия языка C++. В первом подразделе введено понятие полиморфизма, что позволяет называть общим именем одинаковые действия вне зависимости от типа данных. Во втором разделе вводится понятие ссылки, что позволяет упростить код. В третьем разделе вводится понятие константного выражения, которое позволяет устранить часть противоречий вводимое ссылкой.

1.1 Полиморфизм

В языке C вызовы функции определяется её названием и только. Если нам требуется функция по смысле схожая с существующей, и объявить функцию с тем же названием, но для другого типа данных, то это будет нарушением языка Си и компилятор будет ругаться. Поэтому в языке Си для решения такой ситуации приходится немного видоизменять само название функции. Например, в языке Си существует разные варианты для вычисления абсолютной величины:

```
1  //из библиотеки stdlib.h
2  int a = -5;// Обявили переменную a типа int
3  a = abs(a);// Вычислили абсолютную велечину с abs.
4  long int b = -50;//Тип для целых больший чем int.
5  b = labs(b);// Вычислили абсолютную велечину с labs.
6  //из библиотеки math.h
7  float q = -1.5;
8  q = fabsf( q ); // Вычислили абсолютную велечину с fabsf.
9  double k = -1.5;
10 k = fabs( k ); // Вычислили абсолютную велечину с fabs.
```

Как мы видим, в зависимости от типа данных использовалась своя функция вычисления абсолютной величины. Так для обычных целых чисел (int) используется функция abs, а для целых чисел большей разрядности используется функция labs. Для чисел с плавающей точкой используется функция fabsf, а для чисел с плавающей точкой большей разрядности используется функция fabs. Для полноты приведем таблицу для вычисления абсолютной величины в зависимости от типа дынных:

Тип	функция	тип	функция
int	abs	float	fabsf
long int	labs	double	fabs
long long int	llabs	long double	fabsl

Как мы видим легко запутаться. Более того, если тип данных где-то в коде поменялся, то придется вручную в коде поменять все вызовы. Следует отметить, что аналогичные функции существуют (только для плавающей точки), например, для вычисления sin и тому подобное.

Для того чтобы с этим побороться в языке C++ функции отделяются друг от друга не только названием, но и типом аргументов. Поэтому можно объявить функции с одним и тем же именем, но с разным типом у аргументов. Тогда вызываемая функция определяется не только именем функции, но и типом аргументов. Например, я языке C++ можно сделать так:

```
1  double aa(int k, in c)
2  {
3    return k + c;
4  }
5
6  double aa(double k, double c)
7  {
```

```
8     return k * c;
9  }
```

Повторю, в языке Си такой код вызовет ошибки компилятора. Ему не понравится, что функция с одним и тем же именем имеет разный тип аргументов. Для C++ же это корректный код, а именно – эти функции отличаются типом аргументов. У одной они оба целые (int), а у другой оба вещественные (double).

Объявив две функции указанные ранее их можно вызвать следующим образом:

```
1  #include <stdio.h>
2
3  // Далее идут те две функции
4  ...
5  //
6
7  int main()
8  {
9     printf("a ~ %f\n", aa(1,1) );
10    return 0;
11 }
```

Упражнение. Поиграйся с кодом, а именно – поизменяй аргументы при вызове функции aa в строчке 9. Замени сначала оба тина на вещественный (например, вместо 1 используй 1.). Убедись, что вызывается именно та функция, которая соответствует типу аргументов. Для большей убедительности вставить печать внутрь функций.

Упражнение. Далее сделай так, чтобы только один из типов был целым, а другой вещественный. Какую ошибку выдает компилятор. Запомни ошибку, она может пригодится в будущем.

Упражнение. Объявите свое название для вычисления абсолютной величины и обеспечьте вызов правильной функции.

Упражнение. Напиши две функции для вычисления факториала. Одна для целых чисел, а другая для вещественных (Гамма-функция, например, через приближение – Формула Стирлинга).

1.2 Ссылка

В языке Си есть понятие указателя. При его применении может возникнуть много неудобств. В языке C++ вводится понятие ссылки, которое многое упрощает.

Указатели Указатели в языке Си могут использоваться по разным причинам. Одна из них заключается в том, чтобы уменьшить объем передаваемых данных при вызове функции. Так пусть у нас есть структура матрицы:

```
1 typedef
2 struct
3 {
4   int data[100][100];
5 }Matrix;
```

Пусть нам требуется что-то он неё вычислить, например, определитель. Тогда мы могли бы написать код:

```
1 double det(Matrix a)
2 {
3   ...
4 }
```

Такой код был бы неэффективен ввиду того, что аргумент является переменной типа матрица. Напомню, что я языке Си функции вызываются по значению. Последнее означает, что сначала вычисляется значения всех аргументов, а потом они копируются в переменные при функции. Значит, при вызове такой функции во-первых, будет определена (выделена память) данная переменная, а во-вторых, будет выполнено копирование переменной. Например, рассмотрим такой код вызова:

```
1 Matrix d;
2 double v = det( d );
```

При вызове функции det переменная d будет скопирована в аргумент функции det, т.е. в новую (локальную) переменную функции det. Данная операция приведет не только к увеличению объема потребляемой памяти, но и вычислительные ресурсы требуемые для копирования.

Ввиду последнего в таких случаях гораздо более рационально использовать указатели. Они позволяют дать доступ к переменной без копирования. Так, предыдущую функцию можно переписать так:

```
1 double det(Matrix *a)
2 {
3   ...
4 }
```

Напомню, что указатель на объект образуется путем присоединения звездочки (*) справа от типа данных. В данном случае:

```
1 Matrix *
```

является указателем на тип данных Matrix.

Тогда при вызове нужно будет передать лишь указатель (он занимает мало места) на объект. Вызываться функция будет так:

```
1 Matrix d;
2 double v = det( &d );
```

Напомню, что оператор амперсанд (&) вычисляет указатель на объект.

С точки зрения идеологии вызова по значению в данном случае значением будет указатель. Он очень мало место занимает. Поэтому позволяет повысить эффективность вызова.

И именно указатель передается функции, а не сам объект.

Некрасиво использовать Понятие ссылки является синтаксическим сахаром. По сути оно позволяет избавится от неудобств связанных с указателем, но по сути является тем же самым.

Так, при написании тела кода функции, например, стр 1 необходимо будет выполнять разыменование указателя. В случае структур это обычно подразумевает использование оператора стрелочка (3), но можно и звездочку и точкой(4):

```
1 double det(Matrix *a)
2 {
3   double v = p->data[0][0];
4   double v = (*p).data[0][0];
5   ...
6 }
```

Звездочка используется для разыменования в общем случае. Это крайне не удобно и ухудшает читабельность кода.

Более того, каждый раз при вызове функции нужно вычислять указатель, т.е. писать аперсанд. Представим себе, что нам каким то образом удалось арифметические знаки переопределять (благодаря полиморфизму). Тогда для знака плюс («+») можно написать свою типа функцию для сложения двух матриц.

```
1 double operator+(Matrix *a, Matrix *b)
2 {
3   ...
4 }
```

Тогда при вызове код будет выглядеть так:

```
1  &d + &q
```

Ясно что такой код крайне не красив. Для устранения таких эффектов и появились ссылки.

Ссылка Ссылка является синтаксическим сахаром позволяющим избавиться от явного разыменования (символа звездочки) в теле самой функции (3) и знака аперсанда при вызове функции (8). Функцию вычисляющий определитель можно переписать так:

```
1  double det(Matrix &a)
2  {
3    double v = p.data[0][0];
4       ...
5  }
6  ...
7  Matrix d;
8  double v = det( d );// Нет амперсанда!
```

Для этого в аргументе функции используется знак амперсанда вместо звездочки. Фактически это полностью эквивалентно прошлому варианту. Например, если взглянуть на строчку 8, но может создастся впечатление, что будет передана сама матрица функции, т.е. большой объем данных. Но на сама деле ввиду того, что тип аргумента является ссылкой (см. стр. 1), будет передан только указатель на матрицу (а не сама матрица).

Дабы объяснение был более полным приведем пример, но вместо матрицы возьмем всем известный тип, тип int.

```
1  void myabs(int *a)
2  {
3    if( (*a) < 0 )
4      *a = -*a);
5
6  }
7
8  void myabs2(int &a)
9  {
10   if( a < 0 )
11     a = -a;
12 }
```

Видно что вторая функция выглядит проще. Но по сути (на машинном коде) это одно и тоже.

Для вызова этих функций соответственно используется следующий код.

```
1 int k = -5;
2 myabs( &a );
3 printf("abs ~ %d\n", a);
4
5 int k2= -10;
6 myabs2( b );
7 printf("abs ~ %d\n", b);
```

В чем может быть проблема в прошлом коде? Что будет если сделать так:

```
1 myabs( &-5 );
2 myabs2( -10 );
```

Ясно что нельзя вычислить указатель от числа (см. 1). Указатель можно вычислить только от объекта находящегося постоянно в памяти. Ввиду того, что ссылка это просто другая запись той же сути, запись в строчке 2 также вызовет ошибку компилятора.

Упражнение. Проведите данный эксперимент на компьютере. Какую ошибку выдаст компилятор? В чем её суть?

1.3 Константные выражения

С точки зрения логики проблема в прошлом примере заключалась в том, что в функции изменялось выражение и было не ясно куда записать ответ. Конкретно было не понятно куда записать абсолютную величину от входного числа. Фактически мы пытались записать в число -5 число 5. Что является некорректным. Фактически, последнее и означает, что передаваемый указатель указывает на константное (постоянное) выражение.

С функцией, которая изменяет входной аргумент бессмысленно бороться. Так как это противоречит самой сути. Поэтому для начала видоизменим функцию так, чтобы она не меняла входного значения (точнее значения на которое указывает аргумент).

```
1 int myabs3(int *a)
2 {
3   if( (*a) < 0 )
4     *a = -*a);
5
```

```
6   return *a;
7 }
8
9 int myabs4(int &a)
10 {
11   if( a < 0 )
12     return -a;
13   return a;
14 }
```

Данные функции не изменяют аргумент. Они возвращают результат. В данном случае действительно хотелось бы иметь возможность вызывать функцию myabs4 с числом в качестве аргумента.

Ещё раз о том, что происходит при вызове функции. Допустим у нас такой код:

```
1 int a, b;
2 ...
3 printf("%d\n", a + b);
```

Как неоднократно говорилось, функции вызываются по значению, что означает, что сначала вычисляются значения всех аргументов, а потом вызывается функция. Тогда по сути предыдущий код эквивалентен:

```
1 int a, b;
2 ...
3 int tmp = a + b;//На самом деле тип будет чуть другой.
4 printf("%d\n", tmp);
```

Причем переменная tmp создана компилятором. Стандартом налагается ряд требований на эту переменную. В частности, что она не будет изменена (т.е. что она является константной).

Сами функции мы переписали, чтобы они не меняли значения соответствующие входному аргументу. Как об этом сказать компилятору? Это делается просто. Для этого существуют ключевое слово const. Последнею функцию (такое возможно только Си++):

```
1 int myabs4(const int &a)
2 {
3   if( a < 0 )
4     return -a;
5   return a;
6 }
```

Упражнение. Убедись в том, что данную функцию можно вызвать и передав число.

Что же происходит? Что-то на подобии этого:

```
1 const int tmp = -10;//Добавлено компилятором.
2 printf("%d\n", myabs4( tmp ) );
```

Последняя функция не вызовет ошибок компилятора.

В более сложном варианте:

```
1 int a, b;
2 ...
3 printf("%d\n", myabs4( a + b ) );
4 //Эквивалентно
5 const int tmp = a + b;
6 printf("%d\n", myabs4( tmp ) );
```

Сишные же варианты функций не сработают. Хоть и можно поставить const при указателе, при вызове возникнут проблемы:

```
1 myabs3( &(-5) );//Ошибка компилятора. Указатель не сущ.
2 myabs3( &(a + b) );//Указатель не сущ. от суммы.
```