Курс с++

Царьков Олег

Описание тикета OTSN-5

Еще раз о ссылках(reference). Есть указатели, хранящие в себе адрес переменной, с помощью которых можно получить доступ операцией разыменовывания (звездочку впереди дописать).

Есть переменные, хранящие в себе свое значение, с их помощью можно получить номер ячейки памяти, в которой они лежат операцией взятия ссылки. Что такое ссылка? Есть страница в интернете, у нее есть адрес. Вот этот адрес и есть ссылка на страницу. У страницы можно взять ссылку. Вот и у переменной в языке программирования можно взять ссылку, что обозначает узнать номер ячейки памяти, где она находится.

Операция взятия ссылки обозначается диезом &. Операции разыменовывания и взятия ссылки обратны друг другу.

Пример:

$int\ a;$
$int * pointer = \&a;$
Теперь указатель $pointer$ хранит в себе номер ячейки, в которой лежит a.

Для того, чтобы получить доступ к a, можно написать $*pointer$.

$int * pointer;$
$int\ a = *pointer;$

Ссылки очень нужны. Если хочется, чтобы параметр не копировался при вызове функции, нужно писать

$void\ some\_func(const\ int\&\ a)\{$
...
}

Потом во время вызова
$some\_func(a);$

Тогда во время вызова $some\_func$ копируется не a, а только ссыка, то есть адрес переменной a. Конечно, очень странно, что при вызове мы пишем $some\_func(a)$, а не $some\_func(\&a)$, но ничего, привыкнуть можно.

Конечно, для типов int и $double$ бесполезно использовать такую технику, потому что они сами по размеру не сильно превышают

размер указателя, и не зачем заменять их копирование на копирование указателя — от этого никакого выигрыша не будет. Но вот если у нас есть какой-то новый, другой тип, который занимает очень много памяти, то вместо него надо передавать ссылку на него.

Если не просто хочется избежать копирования большого объекта, но еще и хочется изменять его внутри функции, то лучше использовать указатели(можно тоже использовать ссылки, но тогда читателю программы было бы трудно отличить функцию, которая что-то меняет, от функции, которая не меняет ничего).

Пример:

$void\ add\_one(int * number)\{$
$* number+ = 1;$
$\}$

а при вызове

$int\ a = 1;$
$add\_one(\&a);$

Здесь уже при вызове мы передаем $\&a$, так как нужно передать адрес памяти.

Шаблонные функции. Предположим, нам хочется написать функцию, считающую сумму двух переменных, но мы не знаем заранее, какого они типа.

Хочется, чтобы функция работала для параметров любого типа — int, $size\_t$, $double$, ... Придется писать несколько функций:

$int\ sum(const\ int\ a, const\ int\ b)\{$
$return\ a + b;$
$\}$
$double\ sum(const\ double\ a, const\ double\ b)\{$
$return\ a + b;$
$\}$
$size\_t\ sum(const\ size\_t\ a, const\ size\_t\ b)\{$
$return\ a + b;$
$\}$

Оказывается, не придется, потому что в языке есть шаблонизаторы. Шаблонизатор — это такая как бы "переменная", хранящая в себе тип. Для ее объявления используется слово $typename$. Для

объявления функции, которая ее использует, используется слово *template*. Выглядит это все так:

```
template
< typename T >
T sum(const T& a, const T& b){
return a + b;
}
```

при вызове

```
int a, b;
sum < int > (a, b);
```

При вызове вначале подставляется *int* вместо T(то, что в треугольных скобочках подставляется вместо того, что в треугольных скобочках).

Можно вызвать эту функцию и так:

$sum(a, b)$;

потому что, если тип a и b *int*, то он может сам угадать, что вместо T нужно *int* подставить.

Стоит заметить, что мы получим ошибку, если напишем

```
int a;
double b;
sum(a, b);
```

Потому что, что бы вместо T не подставить, походящей функция все равно не станет.

Поправить это можно так

```
template
< typename T1, typename T2 >
T1 sum(const T1& a, const T2& b){
return a + b;
}
```

Но ошибок все равно не избежать, например, если типы $T1$ и $T2$ нельзя вообще складывать друг с другом, или если можно, но результат сложения не является типа $T1$.

Задание.

1) Переписать все функции из предыдущего задания(OTSN-4) так, чтобы они работали не только с массивом целых чисел, но и с массивом любых чисел. Протестировать для массивов типа *int* и *double*.

2) Написать функцию, принимающую коэффициенты многочлена, его степень, и значение аргумента, и вычисляющую значение многочлена в данной точке. Функция должна работать с коэффициентами любого типа.

3) Написать функцию, удваивающую число любого типа. Написать другую функцию, которая просто печатает на экране вдвое большее число, но само число, подаваемое ей, не изменяет.