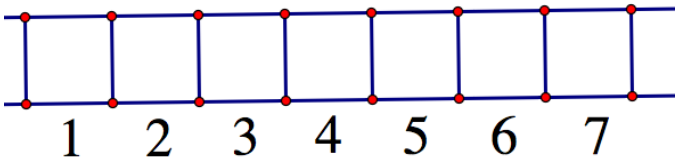
Курс с++

Царьков Олег

Описание тикета OTSN-4

Указатели (pointer) и ссылки(reference / ref). Память, в которой программа размещает используемые ей переменные можно представлять себе как ленту из подряд идущих ячеек



Указатель — переменная, содержащая в себе номер ячейки.

С указателями можно производить следующие операции:

• разыменовывание(pointing) — доступ к содержимому, записанному в ячейке, номер которой записан в указателе.

• сдвиг на k ячеек вправо

• сдвиг на k ячеек влево

• разность двух указателей – количество ячеек между ними.

Любая создаваемая в ходе программы переменная, безусловно, размещена в некотором месте в памяти.

Ссылка — адрес памяти, по которому лежит переменная.

С ссылками можно проводить следующие операции:

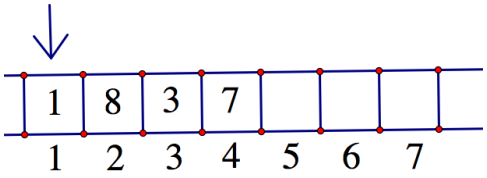
• получение ссылки на объект (referring)

• все те операции, которые производятся с объектом, размещенным по данному адресу, можно производить и со ссылкой. При этом на самом деле операции производятся именно над размещенным в памяти объектом, а ссылка играет символическую роль "второго названия" переменной, лежащей в данной области памяти.

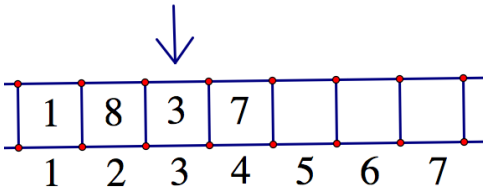
Подробнее. Возникает следующая проблема: ячейки памяти все имеют одинаковый размер, но объекты могут иметь разный размер.

Например, целое число может занимать всего лишь две ячейки, а действительное - 8 ячеек.

Пусть, например, в памяти записано два числа подряд – 18 и 37, и указатель указывает на место, где написано 18.



Как понять, что он указывает на число 18, а не на число 1837?



И как понять, что при сдвиге на 1 вправо сдвиг на самом деле должен быть на две ячейки(так как если мы сдвинемся на одну ячейку и начнем читать середину числа 18, то получим совсем не то, что ожидали).

Чтобы избежать таких проблем, при сдвиге указателя на k, происходит сдвиг не на k ячеек, а на $k*sizeof(var)$ ячеек, где $sizeof(var)$ — количество ячеек, требуемое для размещения переменной var(в примере выше $sizeof(var) = 2$)

Для того, чтобы это было возможным, указатель должен не просто представлять из себя номер ячейки памяти, а должен обладать информацией о том, на переменную какого типа он указывает (чтобы определять размер памяти, занимаемой переменной такого типа).

Создание указателя:

$$typename * \ pointername;$$

где $typename$ — имя типа переменной, на которую будет указывать указатель(фактически ему нужен только размер переменной такого типа), $*$ — специальный символ, и $pointname$ – имя указателя, которое можно выбрать каким угодно, как и имя любой переменной в программе.

Указатель — это тоже переменная, значение которой — номер ячейки, и он тоже сам содержится в какой то ячейке.

Поэтому можно определить указатель, указывающий на указатель

$$typename * * \ pointername;$$

но, в обычных ситуациях, этого не нужно делать.

Сочетания определения указателя со словом *const* могут быть следующими:

• const typename * pointername; указатель на переменную типа *const typename*

• typename const * pointername; – то же самое (вообще говоря, слово const следует размещать после имени типа, но когда имя типа является первым словом в строке, то имеется возможность помещать const перед ним, как в пункте выше, и почему-то принято делать именно так, как в предыдущем пункте, так что этот вариант использовать не рекомендуется).

• typename * const pointername; – указатель на переменную типа *typename*, который сам не может изменяться.

• const typename * const pointername; — указатель на тип *const typename*, который не изменяется.

Указатели, которые не могут изменяться, — довольно бессмысленная вещь.

Для этих целей просто применяется ссылка(или псевдоним).

Если имеется переменная var, то взятие от нее ссылки делается следующим образом:

$$\&var$$

Сылка — это тоже как бы указатель на место в памяти, где лежит переменная, с тем лишь исключением, что он четко привязан именно к этой переменной, с его помощью нельзя управлять участком памяти, где она лежит(как будто мы работаем с константным указателем).

Опишем как делаются остальные процедуры:

• сдвиг указателя на целое число k (на k * sizeof(typename) ячеек)

$$pointername = pointername + k;$$

или

$$pointername+ = k;$$

• расстояние между указателями

$$k = first\_pointer - second\_pointer;$$

• разыменование указателя

$$*pointername;$$

Стоит понимать, что символ '*' в определении указателя и в разыменовывании указателя — это вообще разные никак не связанные друг с другом вещи. Например, рассмотрим код

$int * a;$ // определение указателя
$* a = 5;$ // получаем доступ к содержимому с помощью
//разыменования и присваиваем содержимому число 5

Тут все хорошо, но предположим, что мы обделены интеллектом, и написали

$int * a = 5;$

Тут звездочка относится к имени типа: переменная a имеет тип $int*$, и мы пытаемся сразу же присвоить ей значение 5. То есть значение 5 присваивается не тому, что лежит по адресу a, а самой переменной a, будто мы заставляем указывать на пятую ячейку памяти. Естественно, пятая ячейка памяти нашей программе вряд ли дана в использование, поэтому копьютер взорвется при первой же попытке запустить такую программу:)

Вот как можно присвоить значение указателю сразу:

$int\ b;$
$int * a = \&b;$

Тут мы применили операцию взятия ссылки на b. Ссылка на b указывает на b (КЭП) и мы заставили указатель a тоже указывать на b.

Зачем все это нужно? Это все появилось как удобное оружие для работы с вызовом функций. Рассмотрим пример программы:

```
const int plus_one(int a){
a = a + 1;
return a;
}

int main(){
int a = 4;
std :: cout << plus_one(a);
std :: cout << a;
}
```

Первый раз выведется 5, второй раз – 4. Все дело в том, что при передаче параметра a в функцию plus_one произошло его копирование, поэтому сама функция прибавляет 1 уже не к исходному

параметру a, а к его копии, в результате чего то a, которое определено в функции $main$, так и остается равное 4.

Иногда это, может быть, и полезно. А что если я действительно хочу изменить значение параметра, который передаю?

Возникает и более простая проблема: а что если параметр, который я передаю, весит 2 гигабайта? Я должен буду ждать 3 часа пока он скопируется, но этого мне совсем не хочется делать.

Для решения первой проблемы стоит написать программу так:

```
void plus_one(int * pointer){
* pointer = *pointer + 1;
}

int main(){
int a = 4;
plus_one(&a);
std :: cout << a;
}
```

Выведется 5.

Заметим, что мне даже не пришлось ничего возвращать из функции (тип void означает пустоту, если функция объявлена как void, то она может ничего не возвращать, но тогда ее значение нельзя использовать в выражении).

Копирование при передаче параметра произошло и в этом примере тоже, только на этот раз копировалась не переменная a, а ее адрес — то есть просто номер ячейки памяти, где она лежит. В результате я обращаюсь к объекту, который лежит в ячейке со скопированным номером, то есть к тому же самому объекту(я могу переписать адрес человека на другой листочек, но все равно по этому адресу будет жить тот же самый человек). Поэтому на этот раз я изменю именно переменную a, не смотря на то, что она определена только в $main$ и функция $plus\_one$ вообще не знает, что такое a.

Для решения второй проблемы стоит написать программу так:

```
const int plus_one(const int &ref){
return ref + 1;
}

int main(){
int a = 4;
```

$std :: cout << plus\_one(a);$
$std :: cout << a;$
}

Вначале выведется 5, а потом 4. Ничего не изменилось по сравнению с первым вариантом, кроме того, что теперь копируется не переменная a, а ссылка на нее.

Я мог и изменить ref внутри функции $plus\_one$ и эти изменения отразились бы на переменной a функции $main$, но так делать не принято. Люди договорились использовать указатели, когда внутри функции переменная изменяется, и использовать ссылки когда не нужно менять параметров, но просто нужно передать ссылку на параметр, чтобы лишний раз его не копировать(переписать адрес человека на листочек легче, чем переселить его в другую квартиру).

Третий пример слегка сложен для понимания по двум причинам:

1) функция вроде бы принимает ссылку, но однако вызываем мы ее так: plus_one(a), а не так: plus_one(&a).

Да, это странно.

2) вроде бы внутри функции $plus\_one$ ref – это ссылка, однако мы пишем $ref + 1$ а не $*ref + 1$.

Да, это странно.

Выводы. Два последних примера нужно запомнить и просто всегда так писать, не задавая лишних вопросов. Препоследний пример – пример того, когда мы хотим изменить переменную, передаваемую в функцию. Последний пример — когда мы не хотим ничего изменять, а просто хотим избежать копирования параметра(а мы всегда хотим избежать копирования параметра!!)

Имеются исключения из этих двух правил: если нам нужно передать в функцию переменную типа int или double то можно сделать это просто так

$void\ smth(int\ a, double\ b)$

Пусть переменные копируются, ведь размеры переменных int и double не сильно превышают размер переменной, хранящей указатель(я не хочу переписывать адрес квартиры, в которой лежит бумажка, на которой написан размер моих ботинок, я лучше перепишу себе размер моих ботинок – это число, которое короче, чем адрес моей квартиры – мне вообще в данном случае незачем пользоваться адресами).

Еще немного. Кроме всего описанного выше, указатели являются хорошим оружием для управления с массивами (наборами) чисел.

Массив чисел — это большой кусок памяти, в котором подряд записаны числа некоторого типа. Имеется указатель на начало этого списка

$int * a$;

Я могу обращаться к любому элементу массива, сдвинув указатель и разименовав его

$*(a + k)$

для обращения к $(k + 1)$ -ому элементу.

В языке предусмотрена более короткая запись для этого

$a[k]$

Итак, написать $a[k]$ это все равно, что написать $*(a + k)$.

Когда мы объявляем указатель вот так

$int * a$;

то выделяется лишь только одна ячейка памяти произвольным образом, и ее номер присваивается a.

Но что если мы хотим выделить много ячеек под массив?

$a = static\_cast < int* > (malloc(10 * sizeof(int)))$;

Выделит нам 10 ячеек. При этом надо в конце программы написать $free(a)$, чтобы память освободилась.

Что это за непонятная строчка?

$malloc$ – это функция выделения памяти. Нам нужно было 10 целых чисел, и мы передали в эту функцию размер памяти, который нам нужно выделить под 10 целых чисел — 10*sizeof(int). $static\_cast < typename > (...)$ — преобразование того, что стоит в скобках, к типу typename. Чтобы осознать, зачем это нужно, вернемся к примеру, описанному ранее, где подряд написаны два числа – 18 и 37. Указатель должен быть устроен так, чтобы он знал, что число занимает именно две ячейки. Функция $malloc$ в языке только одна, она не знает, какой именно указатель нам нужен, и она возвращает указатель типа $void*$, который думает, что переменная, на которую он указывает, помещается целиком в одну ячейку. Нам такой указатель не нужен, поэтому приходится насильно преобразовывать к типу $int*$.

Вот эти две строчки

$int * a$; $a = static\_cast < int* > (malloc(10 * sizeof(int)))$;

эквивалентны одной

$int\ a[10]$;

"Фуф, теперь можно все забыть и не париться с этой адской длинной строчкой". – А нет! Нельзя!

Нельзя написать

```
int k;
std :: cin >> k;
int a[k];
```

Потому что в такой записи $int\ a[...]$; можно писать только константное заранее известное число, так как компилятор пытается сразу это прочесть и сразу еще до выполнения программы выделить память.

А вот написать

```
int k;
std :: cin >> k;
int * a;
a = static_cast < int > (malloc(sizeof(int) * k));
....
free(a);
```

всегда можно, потому что *malloc* выполняется во время работы программы.

Задания.

1) Написать программу, которая получает размер массива при вводе с клавиатуры (cin), потом создает массив этого размера, заполняя его числами от 1 до size, и затем выводит его элементы на экран через запятую.

Вывод на экран и создание массива должны быть вынесены в отдельные функции

int * create_array(const int size)

void output_array(int * array, const int size)

2) Написать функцию, переворачивающую массив

void reverse(int * begin, int * end)

где begin – указатель на начало массива, end – указатель на ячейку, находящуюся сразу за последним элементом массива.

Пример использования

```
int a[3];
a[0] = 1;
a[1] = 2;
a[2] = 3;
reverse(a, a + 3);
```

a + 2 указывает на последний элемент, a + 3 указывает на ячейку памяти за последним элементом.

3) Написать функцию, которая делает циклический сдвиг на k вправо.

void shift(int * begin, int * end)

Например, если дан массив

1 2 3 4 5

то циклический сдвиг на 2 это

3 4 5 1 2