Лекция 6. Указатели. Использование свободной памяти

Что такое переменная? Переменная – это определенное место в памяти, имеющее имя, по которому мы его используем. Переменная имеет значение (текущее значение), грубо говоря, это то, что записано сейчас по определенному месту в памяти. Переменная также имеет тип, описывающий, какого рода значения в ней хранятся.

Переменные устроены таким образом, что мы можем работать с ними по их имени, не задумываясь, где на самом деле в памяти и как они хранятся – это их главная идея.

Прежде чем использовать переменную, ее надо объявить и определить ее тип. Одновременно с этим можно задать начальное значение, а можно установить значение переменной после, в любой момент.

float f; // объявляем перем. f типа float

char s = ‘w’; // объяв.перем. s типа char и присв. ей знач. ‘w’ f = -3.2; //перем. f присв. значение-3.2

int a = 5; // объявляем перем. a типа int и присв. ей значение 5 int b = 6; // объявляем перем. b типа int и присв. ей значение 6 int t; // объявляем перем. t типа int

t = a; // переменной t присваиваем значение переменной a a = b; // переменной a присваиваем значение переменной b b = t; // переменной b присваиваем значение переменной t

Во многих языках это было бы практически все, что можно делать с переменными, как с ними работать. Однако в C++, как в языке низкого уровня, с переменными можно работать и по-другому,а именно – косвенно.

Для того, чтобы в этом разобраться, нужно для начала разобраться в том, что представляет из себя память, в которой переменные хранятся. Ведь переменные имеют различные типы, а память одна – как в одной и той же памяти хранятся и целые числа, и дробные числа, и символы, и так далее.

Память

Память – это нумерованная последовательность числовых двоичных байтов. Именно нумерованная (то есть каждый байт имеет номер), именно последовательность (то есть каждый следующий байт имеет номер, отличающийся на единицу), именно двоичных (то есть байт содержит 8 двоичных битов, содержащих, в свою очередь, 0 или 1), именно байтов (то есть можно обратиться только[1] к байту целиком, но не к отдельному биту), и именно числовых (то есть каждый байт представляет из себя число).

Каждый байт этой последовательности имеет свой номер и текущее значение, которое, как несложно догадаться, в двоичном виде представляет из себя число от 00000000 до 11111111, в шестнадцатеричном виде – число от 00 до FF, и в десятичном виде – число от 0 до 255:

Как уже говорилось выше, каждая переменная занимает некоторое место в памяти. Разные типы требуют различного количества байтов. Еще раз: переменная – это определенное место в памяти, имеющее имя, по которому мы его используем. Теперь мы можем сказать более точно. Определенное место – это некоторая определенная непрерывная последовательность байтов в памяти:

Что мы здесь видим? Во-первых,все переменные занимают разное количество (подряд идущих!) байтов.Во-вторых,некоторые байты могут быть не заняты, но, смотря только на значения этих байтов, не скажешь, заняты они или нет, и значениями каких типов заняты. Незанятые байты могут иметь произвольные значения, оставшиеся откаких-топеременных, которые там раньше размещались, но теперь не размещаются. Иногда мы такие байты будем помечать символами ‘?’, но надо обязательно держать в уме, что там не вопросы находятся, акакие-тореальные значения.

И, в-третьих,и в самых главных, все переменные хранятся в той же самой памяти, которая состоит из числовых байтов. Внимание, вопрос. Как различные типы данных могут храниться в

одной и той же памяти? Как в числовой памяти можно хранить символ ‘w’? Как в двух ячейках, каждая из которых может содержать число от 0 до 255, можно хранить число 258?

Для каждого типа данных известно, сколько байтов в памяти занимает переменная его типа (для char – один байт, для int – два байта, и так далее), и, самое главное, – как любое значение этого типа представить в виде последовательности числовых байтов. Для каждого значения char, например, выделяется один байт, и в этом числовом байте хранится номер этого символа в специальной таблице ascii.

Адрес переменной

Адрес переменной – это, грубо говоря, номер первого байта, занимаемого этой переменной в памяти. В вышеприведенном примере, адрес переменной s есть 1, адрес переменной a есть 2, адрес переменной b есть 4. Зная адрес некоторой переменной, и зная ее тип, можно точно узнать, с какого по какой байты в памяти эта переменная занимает. А именно, байты с номерами с адрес(переменной) по адрес(переменной) + размер(переменной) - 1.

Для каждой переменной можно узнать ее адрес, с помощью операции &. Адрес – это, грубо говоря, номер, то есть число. Но не все операции над адресом допустимы и, что важнее, осмысленны.

Приведу такую аналогию. Адрес – это, например, номер почтового ящика. Если рассмотреть ящик номер 5, то его адрес – это, соответственно 5. К этому адресу я могу прибавить какое-точисло, например, 2, и получить новый адрес, а именно 7. Если рассмотреть ящик номер 5 и ящик номер 8, то я могу из адреса второго вычесть адрес первого и получить уже не адрес, но число, равное количеству ящиков между ящиком номер 5 и ящиком номер 8. Но складывать два адреса нельзя, это бессмысленная операция.

Итак, над адресами переменных возможны следующие операции: адрес2 – адрес1 => число1

–количество байтов между байтами по адресам адрес1 и адрес2; адрес1 + число1 => адрес2 –адрес байта, отстоящего от байта по адресу адрес1 на число1 байтов

```
int a = 5;
int b = 6;
if (&b - &a > 10) /* переменные далеко друг от друга */
if (&a + sizeof(a) == &b) /* перем. b идет в памяти сразу за a */
```

Важный вопрос – а если мы захотим получить адрес некоторой переменной и сохранить его временно где-тов памяти, какого размера область памяти нам надо выделить? Или подобный вопрос: в переменную какого типа можно записать адрес? Если адрес – это

значение	?	5	0	?				?
номер	0	1	2	3	4	5	6	7
переменная	не занято	a		ptr				не занято
тип		int		указатель на int				
значение		5		не определено				

значение	?	5	0	номер 1				?
номер	0	1	2	3	4	5	6	7
переменная	не занято	a		ptr				не занято
тип		int		указатель на int				
значение		5		адрес перем. a				

номер, то ответ зависит от того, как много может быть таких адресов? В общем-то,адресов может быть столько, сколько ячеек в памяти, а это может быть и 10^9, и даже больше.

Переменные-указатели

Что такое указатель? Указатель – это «всего-лишь»переменная, значением которой является адрес. То есть это именно такая переменная, в которой можно сохранить адрескакой-тодругой переменной.

Представьте себе переменную как почтовый ящик с бумажкой внутри, на которой что-тонаписано. Адрес этой переменной – это номер почтового ящика. Так вот, указатель – это почтовый ящик с бумажкой внутри, на которой написан номеркакого-тодругого почтового ящика.

Зачем это в принципе может быть нужно? Если продолжать аналогию с почтовыми ящиками, то можно привести такой пример. Пусть у меня есть два почтовых ящика, ящик1 и ящик2. Я хочу, чтобы мои письма доставлялись в какие-тодругие почтовые ящики.

Первый вариант этого добиться – это поставить перед почтальоном задачу вида: «Доставь письмо из ящика1 в ящик5». Завтра мне понадобится доставить письмо в ящик6, и мне придется менять задание почтальону, и в конце концов он все перепутает (грубо говоря – нехорошо каждый раз менять задание, это требует дополнительной информации почтальону).

Другой вариант этого добиться – это поставить перед почтальоном постоянную задачу вида: «Доставь письмо из ящика1 в ящик, номер которого находится в ящике2». Да, это чуть более сложная задача для почтальона, но она постоянная, она не меняется, и он всегда может действовать по одному и тому же шаблону.

Второй вариант называется косвенной адресацией, и ящик2, как вы уже догадались – это указатель, то есть такая переменная, которая содержит адрес другой переменной.

Так же как переменные различаются типами, указатели тоже бывают разными. Так при объявлении указателя обязательно задается, на переменную какого типа это указатель. То есть, по сути – адрес переменной какого типа будет содержаться в этом указателе. Это может показаться лишним, потому что, в любом случае, указатель – это адрес первого байта в памяти, которую занимает переменная (вне зависимости от того, это переменная какого типа). Но так или иначе, этот тип нужно указывать.

int a = 5; // объявляем перем. a типа int и присв. ей значение 5 int *ptr; // объявляем ptr как перем.-

указательна перем. типа int

В данном случае мы объявляем новую переменную ptr, в которой будет храниться адрес какой-топеременной типа int. Другими словами, в переменной ptr будет храниться адрес первого байта области в памяти, в которой хранится переменная типа int.

Пока мы только объявили эту переменную, и она пока не содержит никакого осмысленного значения (не содержит осмысленного адреса). Занесем теперь в ptr адрес, например, переменной a.

ptr = &a;

Так же, как с самой переменной a, мы могли значение указателя (значением указателя является адрес!) установить сразу при объявлении указателя:

значение	?	5	0	**номер 1**				**номер 1**				**номер 1**			
номер	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
переменная	не занято	a		ptr				ptr2				ptr3			
тип		int		указатель на int				указатель на int				указатель на int			
значение		5		адрес перем. a				адрес перем. a				адрес перем. a			

значение	?	5	0	номер 1				5	0
номер	0	1	2	3	4	5	6	7	8
переменная	не занято	a		ptr				b	
тип		int		указатель на int				int	
значение		5		адрес перем. a				5	

значение	?	6	0	номер 1				5	0
номер	0	1	2	3	4	5	6	7	8
переменная	не занято	a		ptr				b	
тип		int		указатель на int				int	
значение		6		адрес перем. a				5	

int *ptr = &a; int *ptr2 = &a; int *ptr3 = ptr;

В приведенном примере создается три указателя на переменную типа int. В переменные-указателиptr и ptr2 сразу заносится адрес переменной a, а в переменную ptr3 заносится значение указателя ptr, а раз значением указателя является адрес, и в ptr к тому моменту находится адрес переменной a – в переменной ptr3 тоже окажется адрес переменной a.

Итак, пусть есть имеется переменная ptr, которая является указателем на переменную типа int. Как работают и для чего используют переменные-указатели?

```
int a = 5;
int *ptr = &a;
//int *ptr; - объявление ptr как новой переменной-указателя
//ptr – переменная-указатель,ее значением является адрес
//*ptr – значение по адресу, хранимому в переменной ptr
```

То есть самой главной операцией у указателей является *. При объявлении указателя, символ * показывает, что это именно не переменная типа int, а указатель на переменную типа int. А при использовании указателя, символ * перед именем указателя позволяет получить не сам адрес, а значение по этому адресу.

Рассмотрим такой пример:

```
int a = 5;
int *ptr = &a; int b = a;
//a => 5
//ptr =>адрес a
//b => 5
//*ptr => 5
```

Что теперь произойдет, если мы в переменную a запишем 6?

```
a = 6;
```

Как видим, значение ptr и значение b не изменились, но:

значение	?	104	101	108	108	111	0	номер 1				?			
номер	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
переменная	не занято	c1	c2	c3	c4	c5	c6	ptr				nptr			
тип		char	char	char	char	char	char	указатель на char				указатель на char			
значение		'h'	'e'	'l'	'l'	'o'	0	адрес перем. c1				не определено			

```
//a => 6
//ptr =>адрес a
//b => 5
//*ptr => 6
```

(*ptr) раньше было равно 5, а теперь стало равно 6.

Операции с указателями

Как уже говорилось выше, указатели допускают несколько видов операций над ними. В частности разность двух указателей есть число, а указатель плюс число есть опять указатель.

Разберемся поподробнее. char c1 = ‘h’;

char c2 = ‘e’; char c3 = ‘l’; char c4 = ‘l’; char c5 = ‘o’; char c6 = 0;
// C++ позволяет вот так символ рассматривать как число от

//0 до 255 и оперировать с ним, как с числом char *ptr = &c1;
char *nptr;
//*ptr => ‘h’
Врезультате, в памяти будет следующая последовательность байтов:

Теперь прибавим к первому указателю 4 и запишем результат во второй указатель: nptr = ptr+4;
// *nptr => ‘o’

То есть второй указатель теперь указывает на последний из символов. Также справедливо следующее:
//nptr-ptr=> 4;
//*(ptr+1) => ‘e’
//*(ptr+2) => ‘l’
//*(ptr+3) => ‘l’
//*(ptr+4) => ‘o’
//*(ptr+5) => 0

То есть указатель плюс число есть новый указатель, у которого потом с помощью символа * можно получить значение переменной по новому адресу. Тонкий момент, который здесь нужно учесть заключается в следующем.

значение	?	104	101	108	108	111	0	номер 1			
номер	0	1	2	3	4	5	6	7	8	9	10
переменная	не занято	s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	ptr			
тип		char	char	char	char	char	char	указатель на char			
значение		'h'	'e'	'l'	'l'	'o'	0	адрес перем. s[0]			

Если к указателю ptr прибавить число i, то в результате получится адрес, смещенный не на i байтов, а на i*sizeof(type) байтов, где type – это тип переменной, на которую указывает указатель. То есть отсчет идет не в байтах, а как бы в единицах того типа, на который указывает указатель. Если ptr есть указатель на char, то так как char занимает в памяти один байт, ptr+1 указывает на область памяти, отстоящую от ptr действительно на 1 байт. А если ptr есть указатель на int, то ptr+1 указывает на область памяти, отстоящую от ptr на два байта, потому что переменная типа int занимает в памяти 2 байта.

Аналогично, разность ptr2-ptr1даст не количество байтов между областями памяти, на которые указывают ptr1 и ptr2, а количество переменных, которые могут «уместиться» в памяти между ptr1 и ptr2. Причем переменных именно того типа, указателями на который ptr1 и ptr2 являются.

Массивы

Волей разработчиков языка, массивы тесно связаны с указателями. Объявим, например, массив символов и запишем в него строчку (которая, как вы помните, обязательно неявно заканчивается символом с кодом 0):
char s[6] = ”hello”;
//s[0] => ‘h’
//s[1] => ‘e’
//s[2] => ‘l’
//s[3] => ‘l’
//s[4] => ‘o’

//s[5] => 0 char *ptr; ptr = s;

Поэтому к элементам массива можно обращаться как с помощью обычного синтаксиса имя[индекс], так и используя возможности указателей:

```
s[0] => ‘h’
//*s => ‘h’

//*ptr =>‘h’ s[1] => ‘e’
//*(s+1) => ‘e’
//*(ptr+1) => ‘e’
```

Прибавив к любому из указателей, например, 1, мы заставим этот указатель указывать на следующий символ массива:

```
//*ptr => ‘h’ ptr++;
//*ptr => ‘e’
```

Обратите внимание, что *(s+1) и (*s)+1 – это совсем разные вещи. Почему так?

В первом случае сначала к указателю s будет прибавлена единица, так что s+1 – это указатель на следующий символ за тем, на который указывает s. Следовательно *(s+1) – это

сам следующий символ, то есть ‘e’.

Во втором случае, сначала выполнится операция *, а стало быть, *s – это символ ‘h’. А после выполнения операции *, к результату ее будет прибавлена единица. Следовательно символ ‘h’ будет расценен как число 104, и результатом (*s)+1 будет число 105, или символ ‘i':

```
char c;
c = *(s+1)

//c => ‘e’ c = (*s)+1
//c => ‘i’
```

Динамические массивы

Динамическим называется массив, размер которого может меняться во время исполнения программы. Для изменения размера динамического массива язык программирования, поддерживающий такие массивы, должен предоставлять встроенную функцию или оператор. Динамические массивы дают возможность более гибкой работы с данными, так как позволяют не прогнозировать хранимые объёмы данных, а регулировать размер массива в соответствии с реально необходимыми объёмами.

При объявлении массива необходимо задать точное количество элементов. На основе этой информации при запуске программы автоматически выделяется необходимый объем памяти. Иными словами, размер массива необходимо знать до выполнения программы. Во время выполнения программы увеличить размер существующего массива нельзя.

Чтобы произвести увеличение массива во время выполнения программы необходимо выделить достаточный объем памяти с помощью оператора new, перенести существующие элементы, а лишь затем добавить новые элементы. Управление динамической памятью полностью лежит на плечах программиста, поэтому после завершения работы с памятью необходимо самим возвратить память операционной системе с помощью оператора delete. Если память не возвратить операционной системе, то участок памяти станет недоступным для дальнейшего использования. Подобные ситуации приводят к утечке памяти. Сделать эти участи опять доступными можно только после перезагрузки компьютера.

Выделение памяти под массив производится следующим образом: <Указатель> = new <Тип данных>[<Количество элементов>]; Освободить выделенную память можно так:

```
delete [] <Указатель>;
```

Пример объявления динамического массива на языках C/C++ Одномерный динамический массив: Создаем массив с 10-юэлементами типа int: int *mas = new int[10];

Получить доступ к значению каждого элемента можно по индексу (порядковый номер):

```
mas[0] = 2; // присвоили значение 2 нулевому элементу массива mas mas[1] = 7; // присвоили значение 7 первому элементу массива mas //... и т.д.
```

Следовательно, если брать такой подход, то вам понадобится около десяти строк кода, чтобы проинициализировать весь массив. Для того, чтобы этого избежать напишем тоже самое в цикле:

```
for(int i = 0; i < 10; i++){
cin>>mas[i]; // пользователь вводит значение каждого i-тогоэлемента
массива
}
```

После чего работаем с массивом. Также его можно вывести на экран:

```
for(int i = 0; i < 10; i++){ cout << mas[i] << endl;
}
```

Для освобождения из памяти одномерного динамического массива используем оператор delete:

```
delete []mas;
```

В случае, если массиву память будет выделена при помощи malloc, то для освобождения такого массива из памяти следует использовать функцию

```
free(mas);
```

Дружественные (friend) функции

Мы уже познакомились с основным правилом ООП - данные(внутренние переменные) объекта защищены от воздействий из вне и доступ к ним можно получить только с помощью методов(функций) объекта. Но бывают такие случаи, когда нам необходимо получить доступ к данным объекта не используя его интерфейс. Зачем это может понадобиться ? Как я уже как-тоупоминал, при доступе к внутренним переменным объекта через его методы уменьшается эффективность работы за счет затрат на вызов метода. В большинстве случаев нам это не критично, но не всегда. В некоторых случаях это может играть существенную роль. Конечно, можно добавить новый метод к классу для получения прямого доступа к внутренним переменным. Однако, в большинстве случаев, интерфейс объекта (методы) спланирован для выполнения определенного круга операций, и наша функция может оказаться как бы ни к месту. А если хуже того, нам необходимо получить прямой доступ к внутренним данным двух разных объектов ? Возникает проблема. Именно для решения подобных задач и существует возможность описание функции, метода другого класса или даже класса как дружественного(friend).

Итак, как же работает этот механизм.

Для описания дружественной тому или иному классу функции(метода или класса) необходимо в описании этого класса объявить (описать) дружественную функцию с указанием ключевого слова friend. Если функция дружественна нескольким классам, то надо добавить это описание во все классы, к внутренним данным которых производим обращение. В большинстве языков ООП не имеет различия в какой раздел описания класса(publuc, protected или private) вставлено описание дружественной функции.

```
// Описание класса A
class A
{
//...
void z(); // Описание функции z класса A };

    // Описание класса B
    class B
    {
    //...
    friend void A::z(); // Описание функции                z   класса A как дружественной
    // классу B, т.е. из функции z класса                  A   можно
//получить доступ к внутренним переменным класса B
};
//Описание класса C
class C
{
```

```
  //...
friend class A; // Описание класса A как дружественного классу C, // все функции класса A будут
  дружественны классу C и
  //из любой функции класса A можно получить доступ к
  //внутренним переменным класса C
```