Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА
Институт Информационных Технологий
Кафедра Промышленной Информатики

ПРОЦЕДУРНОЕ ПРОГРАММИРОВАНИЕ

Тема лекции «Перечисления. Указатели»

Лектор Каширская Елизавета Натановна (к.т.н., доцент, ФГБОУ ВО "МИРЭА - Российский технологический университет") e-mail: liza.kashirskaya@gmail.com

Лекция № 4

ПЕРЕЧИСЛЕНИЯ

Средство C++ enurn представляет собой альтернативный по отношению к const способ создания символических констант. Он также позволяет определять новые типы, но в очень ограниченной манере. Например, рассмотрим следующий оператор:

enumspectrum {red, orange, yellow, green, blue, violet, indigo, ultraviolet};

Этот оператор делает две вещи.

• Объявляет имя нового типа — spectrum; при этом spectrum называется перечислением.

• Устанавливает red, orange, yellow и т.д. в качестве символических констант для целочисленных значений 0…7. Эти константы называются перечислителями.

По умолчанию перечислителям присваиваются целочисленные значения, начиная с 0 для первого из них, 1 — для второго и т.д. Это правило по умолчанию можно переопределить, явно присваивая целочисленные значения. Чуть позже вы увидите, как это делается.

Имя перечисления можно использовать для объявления переменной с этим типом перечисления:

spectrum band; // band — переменная типа spectrum

Переменные типа перечислений имеют ряд специальных свойств, которые мы сейчас рассмотрим.

Единственными допустимыми значениями, которые можно присвоить переменной типа перечисления без необходимости приведения типов, являются значения, указанные в определении этого перечисления. Рассмотрим пример:

band = blue; // правильно, blue - перечислитель

Таким образом, переменная spectrum ограничена только восемью допустимыми значениями. Некоторые компиляторы выдают ошибку, если вы пытаетесь присвоить некорректное значение, в то время как другие выдают только предупреждения.

Для максимальной переносимости вы должны трактовать присваивание переменным типа enum значений, не входящих в определение enum, как ошибку.

Для перечислений определена только операция присваивания. В частности, арифметические операции не предусмотрены:

band = orange; // правильно

++band; // неправильно (операция ++ уже обсуждалась в нашем курсе)

band = orange + red; // неправильно, но довольно хитро

Однако некоторые реализации не накладывают таких ограничений. Это позволяет нарушить ограничения типа. Например, если band равно ultraviolet, или 7, а затем выполняется ++band, и если такое разрешено компилятором, то band получит значение, недопустимое для типа spectrum. Опять-таки, для достижения максимальной переносимости вы должны придерживаться ограничений.

Перечисления — целочисленные типы, и они могут быть представлены в виде int, однако тип int не преобразуется автоматически в тип перечисления:

```
int color = blue; // правильно, тип spectrum приводится к int
band =3; // неправильно, int не преобразуется в spectrum
color = 3 + red; // правильно, red преобразуется в int
```

Обратите внимание, что хотя значение 3 в этом примере соответствует перечислителю green, все же присваивание 3 переменной band вызывает ошибку несоответствия типа. Но присваивание green переменной band законно, потому что оба они имеют тип spectrum. И снова, некоторые реализации не накладывают такого ограничения. В выражении 3 + red сложение не определено для перечислений. Однако red преобразуется в тип int, в результате чего получается значение типа int. Благодаря преобразованию перечисления в int в данной ситуации, вы можете использовать перечислители в арифметических выражениях, комбинируя их с обычными целыми, даже несмотря на то, что такая арифметика не определена для самих перечислителей.

Предыдущий пример

band = orange + red; // неправильно, но довольно хитро

не работает по другой причине. Да, действительно, операция + не определена для перечислителей. Но также верно и то, что перечислители преобразуются в целые числа, когда применяются в арифметических выражениях, поэтому выражениеorange + red превращается в 1 + 0, что вполне корректно. Но это выражение имеет тип int, поэтому оно не может быть присвоено переменной band типа spectrum.

Вы можете присвоить значение int переменной enum, если полученное значение допустимо и применяется явное приведение типа:

band = spectrum(3); // приведение 3 к типу spectrum

Но что будет, если вы попытаетесь выполнить приведение типа для недопустимого значения? Результат не определен, в том смысле, что попытка не будет воспринята как ошибочная, но вы не можете полагаться на полученное в результате значение:

band = spectrum(40003); // результат не определен

Обсуждение того, какие значения являются приемлемыми, а какие неприемлемыми, мы сегодня же разберем.

Как видите, правила, которым подчиняются перечисления, достаточно строги. На практике перечисления чаще используются как способ определения взаимосвязанных символических констант, нежели как средство определения новых типов. Например, вы можете применять перечисления для определения символических констант для операторов switch. Если вы собираетесь только использовать константы и не создавать переменные перечислимого типа, то в этом случае можете опустить имя перечислимого типа, как показано ниже:

```
enum {red, orange, yellow, green, blue, violet, indigo, ultraviolet};
```

Присваиваемые значения должны быть целочисленными. Можно также явно устанавливать только некоторые из перечислителей:

```
enum bigstep{first, second = 100, third};
```

В этом случае first получает значение 0 по умолчанию. Каждый последующий неинициализированный перечислитель увеличивается на единицу по сравнению с предыдущим. Поэтому third имеет значение 101.

И, наконец, допускается указывать одно й тоже значение для нескольких перечислителей:

enum {zero, null = 0, one, numero_uno = 1};

Здесь zero и null имеют значение 0, aone и numero_uno — значение 1. В ранних версиях C++ элементам перечислений можно было присваивать только значения типа int (или неявно преобразуемые к int), но теперь это ограничение снято, и также можно использовать значения типа long или даже longlong.

Изначально правильными значениями перечислений являются лишь те, что названы в объявлении. Однако C++ расширяет список допустимых значений, которые могут быть присвоены перечислимым переменным, за счет использования приведения типа. Каждое перечисление имеет диапазону и с помощью приведения к типу переменной перечисления можно присвоить любое целочисленное значение в пределах этого диапазона, даже если данное значение не равно ни одному из перечислителей.

Например, предположим, что bits и my flag определены следующим образом:

```
enum bits {one = 1, two = 2, four = 4, eight = 8};
bits myflag;
```

В таком случае показанный ниже оператор является допустимым:

```
myflag = bits (6); // правильно, потому что 6 находится в пределах диапазона
```

Здесь 6 не является значением ни одного из перечислителей, однако находится в диапазоне этого определенного перечисления.

Диапазон определяется следующим образом.

Для нахождения верхнего предела выбирается перечислитель с максимальным значением. Затем ищется наименьшее число, являющееся степенью двойки, которое больше этого максимального значения, и из него вычитается единица. (Например, максимальное значение bigstep, как определено выше, равно 101. Минимальное число, представляющее степень двойки, которое больше 101, равно 128, поэтому верхним пределом диапазона будет 127.)

Минимальное число, представляющее степень двойки, которое больше 101, равно 128, поэтому верхним пределом диапазона будет 127.) Для нахождения минимального предела выбирается минимальное значение перечислителя. Если оно равно 0 или больше, то нижним пределом диапазона будет 0. Если же минимальное значение перечислителя отрицательное, используется такой же подход, как при вычислении верхнего предела, но со знаком минус. (Например, если минимальный перечислитель равен -6, то следующей степенью двойки будет -8, и нижний предел получается равным -7.)

Идея состоит в том, чтобы компилятор мог выяснить, сколько места необходимо для хранения перечисления. Он может использовать от 1 байт или менее для перечислений с небольшим диапазоном, и до 4 байт — для перечислений со значениями типа long.

УКАЗАТЕЛИ

При выполнении любой программы все необходимые для ее работы данные должны быть загружены в оперативную память компьютера. Для обращения к переменным, находящимся в памяти, используются специальные адреса, которые записываются в шестнадцатеричном виде.

Если переменных в памяти потребуется слишком большое количество, которое не сможет вместить в себя сама аппаратная часть, произойдет перегрузка системы или её зависание.

Если мы объявляем переменные статично, так как мы делали в предыдущих уроках, они остаются в памяти до того момента, как программа завершит свою работу, после чего уничтожаются.

Такой подход может быть приемлем в простых примерах и несложных программах, которые не требуют большого количества ресурсов. Если же наш проект является огромным программным комплексом с высоким функционалом, объявлять таким образом переменные, естественно, было бы довольно глупо.

Можете себе представить, если бы в компьютерных играх использовался такой метод работы с данными, геймерам пришлось бы перезагружать свои высоконагруженные системы после нескольких секунд игры.

Дело в том, что, играя в стратегию, геймер в каждый новый момент времени видит различные объекты на экране монитора, например, сейчас он стреляет во врага, а через долю секунды он уже падает убитым, создавая вокруг себя множество спецэффектов, таких как пыль, тени, и т.п.

Вы знаете, как на экране возникает иллюзия движения? Рисуется новое положение объекта и при этом старое стирается.

Естественно, все это занимает какое-то место в оперативной памяти компьютера. Если не стирать из памяти неиспользуемые объекты, очень скоро они заполнят весь объем ресурсов ПК.

На FullHD-экране (1920*1080 пикселей) и 24 бит на цвет каждого и с обновлениями 60 раз в секунду потребуется 356 Мб памяти каждую секунду. Поиграть можно будет примерно 10 секунд!

По этим причинам в большинстве языков, в том числе и C/C++, имеется понятие указателя. Указатель — это переменная, хранящая в себе адрес ячейки оперативной памяти.

Мы можем обращаться, например, к массиву данных через указатель, который будет содержать адрес начала диапазона ячеек памяти, хранящих этот массив.

После того, как этот массив станет не нужен для выполнения остальной части программы, мы просто освободим память по адресу этого указателя, и она вновь станет доступна для других переменных.

УКАЗАТЕЛИ

Существует три основные свойства, которые должна отслеживать компьютерная программа, когда она сохраняет данные.

- где хранится информация;
- какое значение сохранено;
- разновидность сохраненной информации.

Пока что вы использовали только одну стратегию: объявление простых переменных. В операторе объявления предоставляется тип и символическое имя значения. Он также заставляет программу выделить память для этого значения и внутренне отслеживать ее местоположение.

Теперь мы рассматриваем другую стратегию, важность которой проявляется при разработке классов в ООП C++. Эта стратегия основана на указателях, которые представляют собой переменные, хранящие адреса значений вместо самих значений. Но прежде чем обратиться к указателям, давайте поговорим о том, как явно получить адрес обычной переменной. Для этого применяется операция взятия адреса, обозначаемая символом &, к переменной, адрес которой интересует. Например, если home — переменная, то &home — ее адрес. В листинге демонстрируется использование этой операции.

```
#include <iostream>
int main()
{
using namespace std;
int donuts = 6;
double cups = 4.5;
cout << "donuts value = " << donuts;
cout << " and donuts address = " << donuts << endl;
// ПРИМЕЧАНИЕ: может понадобиться использовать
// unsigned (Sdonuts) и unsigned (&cups)
cout << "cups value = " << cups;
cout << " and cups address = " << &cups << endl;
return 0;
}
```

Ниже показан вывод программы из листинга:

```
donuts value = 6 and donuts address = 6
cups value = 4.5 and cups address = 0x7ffd6d7c6bb8
```

В показанной здесь конкретной реализации cout используется шестнадцатеричная нотация при отображении значений адресов, т.к. это обычная нотация, применяемая для указания адресов памяти. (Некоторые реализации применяют десятичную нотацию.) Наша реализация сохраняет donuts в памяти с меньшими адресами, чем cups. Разница между этими двумя адресами составляет 0x0065fd44-0x0065fd40, или 4 байта. Конечно, в разных системах вы получите разные значения этих адресов. К тому же некоторые системы могут сохранять cups перед donuts, и разница между адресами составит 8, потому что cups имеет тип double. Другие системы могут даже разместить эти переменные в памяти далеко друг от друга, а не рядом.

Таким образом, использование обычных переменных трактует значение как именованную величину, а ее местоположение — как производную величину. Теперь рассмотрим стратегию указателей, которая представляет важнейшую часть философии программирования C++ в части управления памятью.

Приведем сравнительный пример обращения к переменным через указатель и напрямую.

Пример использования статических переменных

```
#include <iostream>
using namespace std;

int main()
{
int a; // Объявление статической переменной
int b = 5; // Инициализация статической переменной b

   a = 10;
   b = a + b;
   cout << "b is " << b << endl;
   return 0;
}
```

Пример использования динамических переменных

```
#include <iostream>
using namespace std;

int main()
{
   int *a = new int; // Объявление указателя для переменной типа int
   int *b = new int(5); // Инициализация указателя

   *a = 10;
   *b = *a + *b;

   cout << "b is " << *b << endl;

   delete b;
   delete a;

   return 0;
}
```

Синтаксис первого примера вам уже быть знаком. Мы объявляем/инициализируем статичные переменные **a** и **b**, после чего выполняем различные операции напрямую с ними.

Во втором примере мы оперируем динамическими переменными посредством указателей.

Новая стратегия хранения данных изменяет трактовку местоположения как именованной величины, а значения — как производной величины. Для этого как раз и предусмотрен этот специальный тип переменной — указатель, который может хранить адрес значения. Таким образом, имя указателя представляет местоположение. Применяя операцию *, называемую косвенным значением или операцией **разыменования**, можно получить значение, хранящееся в указанном месте. (Да, это тот же символ *, который применяется для обозначения арифметической операции умножения; C++ использует контекст для определения того, что подразумевается в каждом конкретном случае — умножение или разыменование.) Предположим, например, что manly — это указатель. В таком случае manly представляет адрес, а *manly — значение, находящееся по этому адресу. Комбинация *manly становится эквивалентом простой переменной типа int. Эти идеи демонстрируются в листинге на следующем слайде. Также там показано, как объявляется указатель.

```
#include<iostream>
int main()
{
using namespace std;
int updates =6; // объявление переменной
int * p_updates; // объявление указателя на int
p_updates = &updates; // присвоить адрес int указателю
// Выразить значения двумя способами
cout << "Values: updates = " << updates;
cout << ", *p_updates = " << *p_updates << endl;
// Выразить адреса двумя способами
cout << "Addresses: Supdates = " << &updates;
cout << ", p_updates = " << p_updates << endl;
// Изменить значение через указатель
*p_updates = *p_updates + 1;
cout << "Now updates = " << updates << endl;
return 0;
}
```

Ниже показан пример выполнения программы из листинга

```
Values: updates = 6, *p_updates = 6
Addresses: Supdates = 0x7fff3eb28104, p_updates = 0x7fff3eb28104
Now updates = 7
```

Как видите, переменная updates типа int и переменная-указатель p_updates — это две стороны одной монеты. Переменная update,s в первую очередь, представляет значение, а для получения его адреса используется операция &, в то время как p_updates представляет адрес, а для получения значения применяется операция * . Поскольку p_updates указывает на updates, конструкции *p_updates и updates полностью эквивалентны. Вы можете использовать *p_updates точно так же, как используете переменную типа int. Как показано в примере из листинга, можно даже присваивать значения *p_updates. Это изменяет значение указываемой переменной — updates.

Объявление и инициализация указателей

Давайте рассмотрим процесс объявления указателей. Компьютеру нужно отслеживать тип значения, на которое ссылается указатель. Например, адрес char обычно выглядит точно так же, как и адрес double, но char и double использует разное количество байт и разный внутренний формат представления значений. Поэтому объявление указателя должно задавать тип данных указываемого значения.

Например, предыдущий пример содержит следующее объявление:

```
int * p_updates;
```

Этот оператор устанавливает, что комбинация *p_updates имеет тип int.

Поскольку вы используете операцию *, применяя ее к указателю, сама переменная p_updates должна быть указателем. Мы говорим, что p_update указывает на тип int. Мы также говорим, что тип p_updates — это указатель на int, или точнее, int *. Итак, повторим еще раз: p_updates — это указатель (адрес), а *p_updates — это int, а не указатель (рис. 11.1). К слову, пробелы вокруг операции * не обязательны. Традиционно программисты на C используют следующую форму:

```
int *ptr;
```

Это подчеркивает идею, что комбинация *ptr является значением типа int.

Объявление и инициализация указателей

С другой стороны, многие программисты на C++ отдают предпочтение такой форме:

int* ptr;

Это подчеркивает идею о том, что int* — это тип "указатель на int". Для компилятора не важно, с какой стороны вы поместите пробел. Можно даже записать так:

int*ptr;

Однако учтите, что следующее объявление создает один указатель (pi) и одну обычную переменную типа int (p2):

int* pi, p2;

Знак * должен быть помещен возле каждой переменной типа указателя.

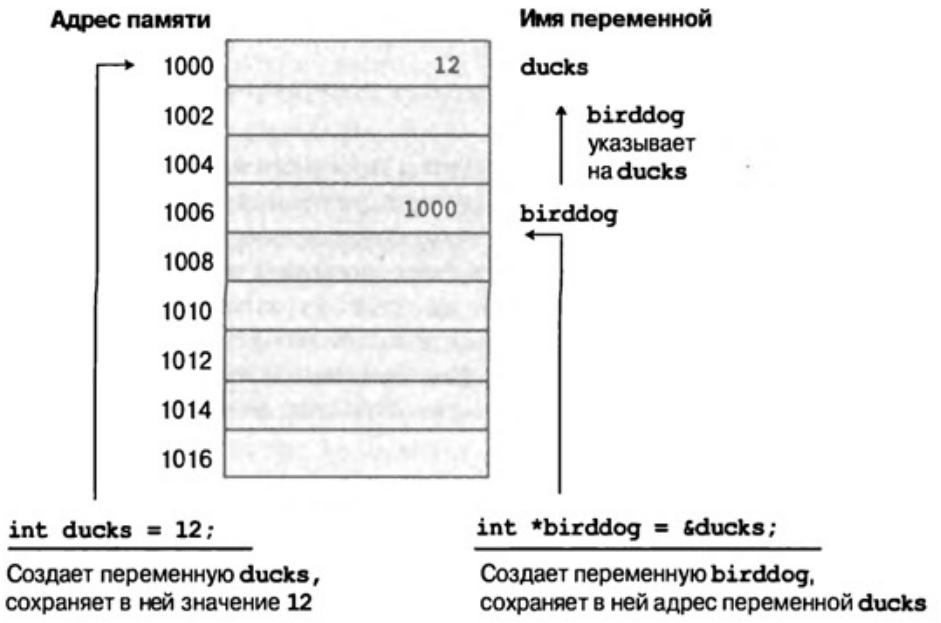

Рисунок 11.1. Указатели хранят указатели

Опасность подстерегает тех, кто использует указатели неосмотрительно. Очень важно понять, что при создании указателя в коде C++ компьютер выделяет память для хранения адреса, но не выделяет памяти для хранения данных, на которые указывает этот адрес. Выделение места для данных требует отдельного шага. Если пропустить этот шаг, как в следующем фрагменте, то это обеспечит прямой путь к проблемам:

```
long * fellow; // создать указатель на long
*fellow = 223323; // поместить значение в неизвестное место
```

Конечно, fellow — это указатель. Но на что он указывает? Никакого адреса переменной fellow в коде не присвоено. Так куда будет помещено значение 223323? Ответить на это невозможно. Поскольку переменная fellow не была инициализирована, она может иметь какое угодно значение. Что бы в ней ни содержалось, программа будет интерпретировать это как адрес, куда и поместит 223323. Если так случится, что fellow будет иметь значение 1200, компьютер попытается поместить данные по адресу 12 00, даже если этот адрес окажется в середине вашего программного кода. На что бы ни указывал fellow, скорее всего, это будет не то место, куда вы хотели бы поместить число 223323. Ошибки подобного рода порождают самое непредсказуемое поведение программы и такие ошибки очень трудно отследить.

Указатели числа

Указатели — это не целочисленные типы, даже несмотря на то, что компьютеры обычно выражают адреса целыми числами. Концептуально указатели представляют собой типы, отличные от целочисленных. Целочисленные значения можно суммировать, вычитать, умножать, делить и т.д. Но указатели описывают местоположение, и не имеет смысла, например, перемножать между собой два местоположения. В терминах допустимых над ними операций указатели и целочисленные типы отличаются друг от друга. Следовательно, нельзя просто присвоить целочисленное значение указателю:

```
int * pt;
pt = 0xB8000000; // несоответствие типов
```

Здесь в левой части находится указатель на int, поэтому ему можно присваивать адрес, но в правой части задано просто целое число. Вы можете точно сказать, что 0xB8000000 — комбинация сегмент-смещение адреса видеопамяти в устаревшей системе, но этот оператор ничего не говорит программе о том, что данное число является адресом. В языке C до появления C99 подобного рода присваивания были разрешены. Однако в C++ применяются более строгие соглашения о типах, и компилятор выдаст сообщение об ошибке, говорящее о несоответствии типов.

Указатели числа

Если вы хотите использовать числовое значение в качестве адреса, то должны выполнить приведение типа, чтобы преобразовать числовое значение к соответствующему типу адреса:

```
int * pt;
pt = (int *) 0xB8000000; // теперь типы соответствуют
```

Теперь обе стороны оператора присваивания представляют адреса, поэтому такое присваивание будет допустимым. Обратите внимание, что если есть значение адреса типа int, это не значит, что сам pt имеет тип int. Например, может существовать платформа, в которой тип int является двухбайтовым значением, то время как адрес — четырехбайтовым значением.

Указатели обладают и рядом других интересных свойств, которые мы обсудим, когда доберемся до соответствующей темы. А пока давайте посмотрим, как указатели могут использоваться для управления выделением памяти во время выполнения.

Рассмотрим общий синтаксис указателей в C++.

Выделение памяти осуществляется с помощью оператора **new** и имеет вид: **тип_данных *имя_указателя = new тип_данных;** *например*, **int *a = new int;**.

После удачного выполнения такой операции в оперативной памяти компьютера происходит выделение диапазона ячеек, необходимого для хранения переменной типа **int**.

Для разных типов данных выделяется разное количество памяти. Следует быть особенно осторожным при работе с памятью, потому что именно ошибки программы, вызванные утечкой памяти, являются одними из самых трудно находимых. На отладку программы в поисках одной ничтожной ошибки может уйти час, день, неделя, в зависимости от упорности разработчика и объема кода.

Инициализация значения, находящегося по адресу указателя, выполняется схожим образом, только в конце ставятся круглые скобки с нужным значением:

тип данных *имя_указателя = new тип_данных(значение).

В нашем примере это **int *b = new int(5)**.

Получение адреса

Для того, чтобы получить **адрес** в памяти, на который ссылается указатель, используется имя переменной-указателя с префиксом **&** перед ним (*не путать со знаком ссылки в C++*).

Например, чтобы вывести на экран адрес ячейки памяти, на который ссылается указатель **b** во втором примере, мы пишем cout << "Address of b is " << &b << endl;. В моей системе, я получила значение **0x1aba030**. У вас оно может быть другим, потому что адреса в оперативной памяти распределяются таким образом, чтобы максимально уменьшить фрагментацию. Поскольку в любой системе список запущенных процессов, а также объем и разрядность памяти могут отличаться, операционная система сама распределяет данные для обеспечения минимальной фрагментации.

Пример получения адреса

```
cout << a << "!\n";
cout << &a << "!\n";
cout << a << "!\n";
cout << *a << "!\n";
```

У меня выводится так:
0x7f6c38d989c8!
0x3d88a00!
0x7f6c38d989c8!
0x3d88a00!

Адрес указателя не меняется.
&a - это адрес, хранимый в указателе.

А если при выводе написать просто b, он выведет адрес памяти?
Да. Указательная переменная хранит адрес смещения от начала.
При ***b** выведет значение, при **b** выведет адрес в HEX.

Для того, чтобы получить **значение**, которое находится **по адресу**, на который ссылается указатель, **используется префикс *.** Данная операция называется **разыменованием указателя**.

Во втором примере мы выводим на экран значение, *которое находится в ячейке памяти* (у меня это **0x1aba030**): cout << "b is " << *b << endl; . В этом случае необходимо использовать знак ***.

Изменение значения

Чтобы изменить значение, находящееся по адресу, на который ссылается указатель, нужно также использовать звездочку, например, как во втором примере:

```
*b = *a + *b;
```

Когда мы оперируем **данными**, то используем знак ***

Когда мы оперируем **адресами**, то используем знак **&**

Для того, чтобы освободить память, выделенную оператором **new**, используют оператор delete.

Пример освобождения памяти

```
#include <iostream>
using namespace std;

int main()
{
   // Выделение памяти
   int *a = new int;
   int *b = new int;
   float *c = new float;

   // ... Любые действия программы

   // Освобождение выделенной памяти
   delete c;
   delete b;
   delete a;

   return 0;
}
```

Выделение памяти с помощью операции new

До сих пор мы инициализировали указатели адресами переменных; переменные — это именованная память, выделенная во время компиляции, и каждый указатель, до сих пор использованный в примерах, просто представлял собой псевдоним для памяти, доступ к которой и так был возможен по именам переменных. Реальная ценность указателей проявляется тогда, когда во время выполнения выделяются неименованные области памяти для хранения значений. В этом случае указатели становятся единственным способом доступа к такой памяти.

Выделение памяти с помощью операции new

В языке С память можно выделять с помощью библиотечной функции **malloc** (). Ее можно применять и в С++, но язык С++ также предлагает лучший способ — **операцию new**.

Создадим неименованное хранилище во время выполнения программы для значения типа int и обеспечим к нему доступ через указатель. Ключом ко всему является операция **new**. Вы сообщаете операции **new**, для какого типа данных запрашивается память; операционная система находит блок памяти нужного размера и возвращает операции **new** его адрес. Вы присваиваете этот адрес указателю, и на этом все. Ниже показан пример:

```
int *pn = new int;
```

Выделение памяти с помощью операции new

Правая часть **new int** сообщает программе, что требуется некоторое новое хранилище, подходящее для хранения int. Операция **new** использует тип для того, чтобы определить, сколько байт необходимо выделить. Затем этот размер сообщается операционной системе, и она находит память и возвращает адрес. Далее вы присваиваете адрес переменной pn, которая объявлена как указатель на int. Теперь **pn — адрес,** а ***pn — значение, хранящееся по этому адресу**. Сравните это с присваиванием адреса переменной указателю:

```
int higgens;
int *pt = &higgens;
```

В обоих случаях (pn и pt) **вы присваиваете адрес** значения int **указателю**. Во втором случае вы также можете обратиться к int по имени **higgens**. В первом случае доступ возможен только через указатель. Возникает вопрос: поскольку память, на которую указывает pn, не имеет имени, как обращаться к ней? Мы говорим, что **pn** указывает на объект данных. Это не "объект" в терминологии объектно-ориентированного программирования. Это просто объект, в смысле "вещь". Термин "объект данных" является более общим, чем "переменная", потому что он означает любой блок памяти, выделенный для элемента данных. Таким образом, переменная — это тоже объект, но память, на которую указывает pn, не является переменной. Метод обращения к объектам данных через указатель может показаться поначалу несколько запутанным, однако он обеспечивает программе высокую степень управления памятью.

Выделение памяти с помощью операции new

Общая форма получения и назначения памяти отдельному объекту данных, который может быть как структурой, так и фундаментальным типом, выглядит следующим образом:

имяТипа *имя_указателя = new имяТипа;

Тип данных используется дважды: один раз для указания разновидности запрашиваемой памяти, а второй — для объявления подходящего указателя.

Разумеется, если вы уже ранее объявили указатель на конкретный тип, то можете его применить вместо объявления еще одного. В листинге демонстрируется применение **new** для двух разных типов. Естественно, точные значения адресов памяти будут варьироваться от системы к системе.

Выделение памяти с помощью операции new

```
#include<iostream>
int main()
{
using namespace std;
int nights = 1001;
int * pt = new int; // выделение пространства для int
*pt = 1001; // сохранение в нем значения
cout << "nights value = "; // значение nights
cout << nights << " : location " << &nights << endl; // расположение nights
cout << "int "; // значение и расположение int
cout << "value = " << *pt << " : location = " << pt << endl;
double * pd = new double; // выделение пространства для double
*pd = 10000001.0; // сохранение в нем значения double
cout << "double ";
cout << "value = " << *pd << ": location = " << pd << endl;
// значениерасположение double
cout << "location of pointer pd: " <<&pd << endl;
// расположениеуказателя pd
cout << "size of pt = " << sizeof(pt);
cout << " : size of *pt = " << sizeof(*pt) << endl;
cout << "size of pd = " << sizeof pd;
cout << ": size of *pd = " << sizeof (*pd) << endl;
return 0;
}
```

Ниже показан вывод программы из листинга:

```
nights value = 1001 : location 0x7ffe7bdf0d1c
int value = 1001 : location = 0x2387c20
double value = 1e+07: location = 0x2387c40
location of pointer pd: 0x7ffe7bdf0d20
size of pt = 8 : size of *pt = 4
size of pd = 8: size of *pd = 8
```

Использование операции new для создания динамических массивов

Если все, что нужно программе — это единственное значение, вы можете объявить обычную переменную, поскольку это намного проще (хотя и не так впечатляет), чем применение **new** для управления единственным небольшим объектом данных. Использование операции **new** более типично с крупными фрагментами данных, такими как массивы, строки и структуры. Именно в таких случаях операция **new** является полезной.

Зачем вообще нужны new и delite?

Операции new и delete в C++ нужны для создания и удаления динамических объектов.

Основная особенность динамических объектов в том, что временем их жизни нужно управлять вручную. Противоположность им с этой точки зрения составляют автоматические объекты, временем жизни которых управляет компилятор (статические объекты).
Автоматические объекты удаляются неявно в соответствии с чёткими правилами, которые реализованы в компиляторе. Локальные переменные функции удаляются, когда поток управления покидает область видимости, в которой они объявлены.
А вот для динамических объектов таких правил нет. Их нужно всегда удалять явно.

Динамические объекты в C++ создаются с помощью new, а удаляются с помощью delete. Вот отсюда и все проблемы: никто не говорил, что эти конструкции следует использовать напрямую! Это низкоуровневые вызовы, они как бы под капотом. И не нужно лезть под капот без необходимости.

В чем проблема с new и delite?

С самого момента своего изобретения операторы new и delete используются неоправданно часто. Самые большие проблемы относятся к оператору delete:
Можно вообще забыть вызвать delete (утечка памяти, memory leak).
Можно забыть вызвать delete в случае исключения или досрочного возврата из функции (тоже утечка памяти).
Можно вызвать delete дважды (двойное удаление, double delete).
Можно вызвать не ту форму оператора: delete вместо delete[] или наоборот (неопределённое поведение, undefined behavior).
Все эти ситуации приводят в лучшем случае к падениям программы, а в худшем к утечкам памяти.

Использование операции new для создания динамических массивов

Если все, что нужно программе — это единственное значение, вы можете объявить обычную переменную, поскольку это намного проще (хотя и не так впечатляет), чем применение **new** для управления единственным небольшим объектом данных. Использование операции **new** более типично с крупными фрагментами данных, такими как массивы, строки и структуры. Именно в таких случаях операция **new** является полезной.

Литература

Материалы из открытого университета INTUIT.RU

Алгоритмизация. Введение в язык программирования C++ [Электронный ресурс]. Режим доступа: https://www.intuit.ru/studies/courses/16740/1301/info