Тема 2.1 Динамические структуры данных и их организация с помощью указателей. Динамические массивы

Чтобы лучше понять указатели, рассмотрим устройство оперативной памяти компьютера. Оперативная память разделена на последовательно пронумерованные ячейки. Каждая переменная размещается в одной или нескольких последовательно расположенных отдельных ячейках памяти; адрес первой из них называется адресом переменной. На рис. 8.1 схематически представлено размещение переменной theAge типа unsigned long int в оперативной памяти компьютера.

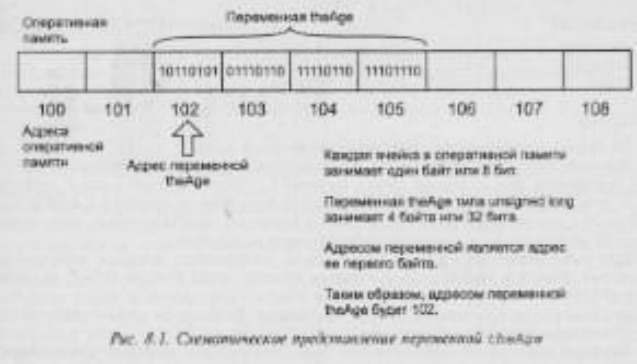
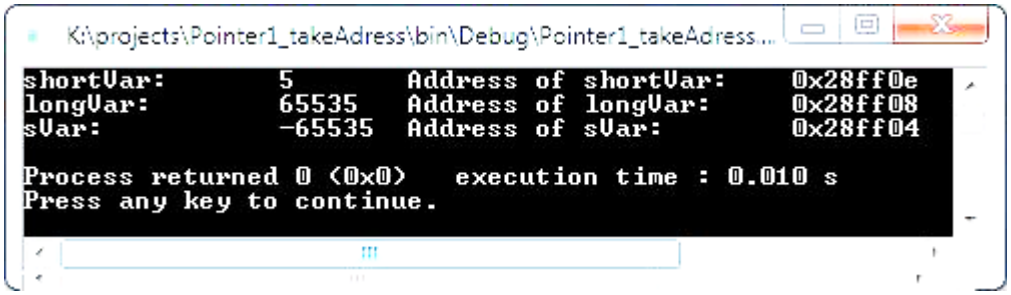


Рис. 8.1. Схематическое представление переменной theAge

Различные виды компьютеров используют различные схемы памяти. Обычно программисту не нужно знать реальный адрес каждой переменной, поскольку компилятор способен сам позаботиться о таких подробностях. Но если необходимость в этой информации все же возникнет, то получить ее можно с помощью оператора обращения к адресу (&), который возвращает адрес расположенного в памяти объекта, что и проиллюстрировано в листинге 8.1.

```
// Листинг 8.1. Применение оператора обращения к
// адресу и адреса локальной переменной

#include <iostream>
using namespace std;

int main()
{
    unsigned short shortVar = 5;
    unsigned long longVar = 65535;
    long sVar = -65535;

    cout << "shortVar:\t" << shortVar;
    cout << "\tAddress of shortVar:\t";
    cout << &shortVar << endl;

    cout << "longVar:\t" << longVar;
    cout << "\tAddress of longVar:\t";
    cout << &longVar << endl;

    cout << "sVar:\t\t" << sVar;
    cout << "\tAddress of sVar:\t";
    cout << &sVar << endl;

    return 0;
}
```



Каждая переменная имеет адрес. Даже не зная сам адрес (номер ячейки), значение его можно сохранить в другой переменной. Такая переменная, хранящая адрес другого объекта, называется **указателем** (pointer).

Указатель (pointer) — это переменная, содержащая адрес области в памяти компьютера.

Предположим, существует целочисленная переменная howOld. Чтобы объявить указатель на нее по имени pAge, применяется следующая форма записи: int * pAge = 0;

Этой строкой переменная pAge объявляется указателем на тип int. В результате будет получена переменная pAge, предназначенная для хранения адреса значения типа int.

Следует заметить, что pAge — обычная переменная. При объявлении переменной целочисленного типа (например, int) указывается, что в ней будет храниться целое число, и компилятор выделит соответствующее количество памяти. Когда переменная объявляется указателем на какой-либо тип, это значит, что она будет хранить адрес переменной определенного типа (обычно размер указателя равен четырем байтам). Таким образом, указатели являются просто отдельным типом переменных.

Имена указателей

Поскольку указатели — это не более, чем обычные переменные, им можно присваивать любые имена, допустимые для других переменных, но большинство программистов, следуя соглашению об именовании, пишут имена всех указателей с маленькой буквы p (от pointer — указатель), например, pAge и pNumber или PtrAge и PtrNumber.

Например: int *pAge = 0;

В этом примере переменная pAge инициализируется нулевым значением. Указатели, значения которых равны 0, называют пустыми (null pointer).

После объявления указателю обязательно нужно присвоить какое-либо значение. Если предназначенный для хранения в указателе адрес заранее не известен, ему следует присвоить значение 0 или NULL (где NULL – это символическая константа, опеределенная в заголовочном файле). Неинициализированные указатели называются дикими (wild pointer). Они очень опасны.

Присвоить адрес переменной howOld указателю pAge можно следующим образом:

```
unsigned short int howOld = 50;    // объявить переменную
unsigned short int * pAge = 0;     // объявить указатель
pAge = &howOld; // присвоить адрес howOld указателю pAge
```

В первой строке объявлена переменная howOld типа unsigned short int, которая сразу инициализируется значением 50. Во второй строке объявлен указатель pAge на переменную типа unsigned short int, который инициализируется значением 0. Символ "звездочка" (*), расположенный между объявляемым типом и именем переменной, свидетельствует о том, что это указатель.

В последней строке указателю pAge присваивается адрес переменной howOld. На это указывает оператор взятия адреса (&) перед именем переменной howOld. Если бы этого оператора не было, присваивался бы не адрес, а значение переменной, которое также может быть корректным адресом.

В данном случае значением указателя pAge будет адрес переменной howOld, значение которой равно 50. Две последние строки можно объединить в одну:

```
unsigned short int howOld = 50;    // объявить переменную
unsigned short int * pAge = &how01d; // объявить указатель на нее
```

Теперь указатель pAge содержит адрес переменной howOld.

Доступ к значению переменной

С помощью указателя можно получить и значение переменной, на которую он указывает (в случае с указателем pAde это значение 50). Доступ к значению переменной по указателю на нее называется **косвенным**, поскольку осуществляется не совсем по правилам.

Косвенное обращение подразумевает получение доступа к значению переменной по адресу, содержащемуся в указателе.

Оператор косвенного доступа (*) называется также **оператором взятия значения**, **разыменования**, или **ссылкой (dereference).** *При извлечении значения из указателя будет возвращено то значение, которое хранится по адресу, содержащемуся в указателе.*

Нормальные переменные обладают прямым доступом к собственным значениям. Чтобы создать новую переменную yourAge типа unsigned short int и присвоить ей значение другой переменной, howOld, можно применить следующий код:

```
unsigned short int yourAge;
yourAge = howOld;
```

Указатель обеспечивает лишь косвенный доступ к значению переменной, адрес которой он, хранит. Чтобы переменной `yourAge` присвоить значение переменной `howOld` с помощью указателя pAge, применяется следующая запись:

```
unsigned short int yourAge;
yourAge = * pAge;
```

Оператор косвенного доступа (*) перед переменной pAge означает, что "само значение находится по адресу...". Иными словами, "возьмите значение, хранимое по адресу, находящемуся в переменной pAge, и присвойте его переменной yourAge".

Исключив оператор косвенного доступа, получим следующую строку:

yourAge = pAge; // ошибка!!

Если попытаться присвоить значение указателя pAge (т.е. адреса в памяти) целочисленной переменной YourAge, то компилятор, вероятно, выдаст предупреждение об ошибке.

Способы применения звездочки

В указателях звездочка (*) используется двумя различными способами: *при объявлении указателя* и в *операторе косвенного доступа*.

При объявлении указателя звездочка (*) является частью синтаксиса и располагается после указания типа объекта. Например:

```
// объявить указатель на тип unsigned short
unsigned short *pAge = 0;
```

При косвенном доступе звездочка означает, что речь идет о значении в памяти, находящемся по адресу в данной переменной, а не о самом адресе.

// разместить значение 5 по адресу, находящемуся в pAge

*pAde = 5;

Заметьте, что тот же символ (*) используется как оператор умножения. Что имени: , имел в виду программист, поставив звездочку, компилятор определяет, исходя и: ! контекста.

Очень важно понимать разницу между указателем, адресом, хранимым в указателе, и значением, расположенным по адресу, содержащемуся в указателе. Обычно именно это и составляет основную сложность при изучении указателей.

Рассмотрим фрагмент кода:

int theVariable = 5;

int * pPointer = &theVariable;

В первой строке объявляется целочисленная переменная theVariable, которой присваивается значение 5. В следующей строке объявляется указатель pPointer на тип int, которому присваивается адрес переменной theVariable. Переменная pPointer является указателем и содержит адрес

переменной theVariable. Значение, хранящееся по адресу, записанному в указателе pPointer, равно 5. На рис. 8.2 схематически показана переменная theVariable и указатель pPointer, содержащий ее адрес.

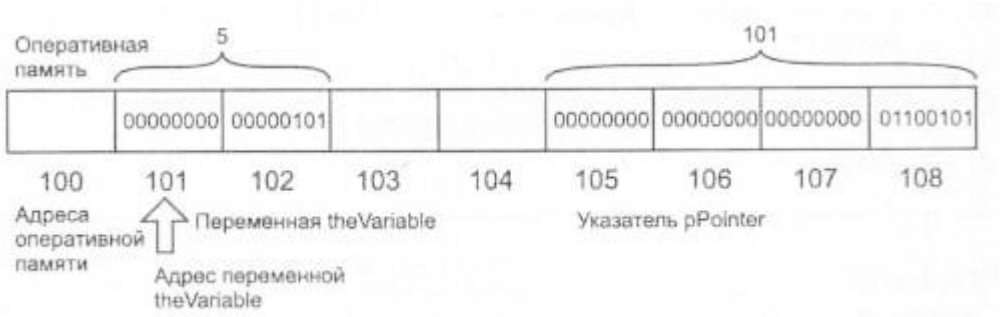


Рис. 8.2. Схематическое представление размещения в памяти переменной и указателя

На этом рисунке значение 5 (двоичное представление 0000 0000 0000 0101) хранится в ячейке по адресу 101 (двоичное представление 0000 0000 0000 0000 0000 0000 0110 0101). В двоичном представлении десятичному числу 5 типа int соответствуют два байта, или 16 битов (по восемь битов на байт). Далее следует двоичное представление числа 101, которое является адресом переменной theVariable, содержащей значение 5.

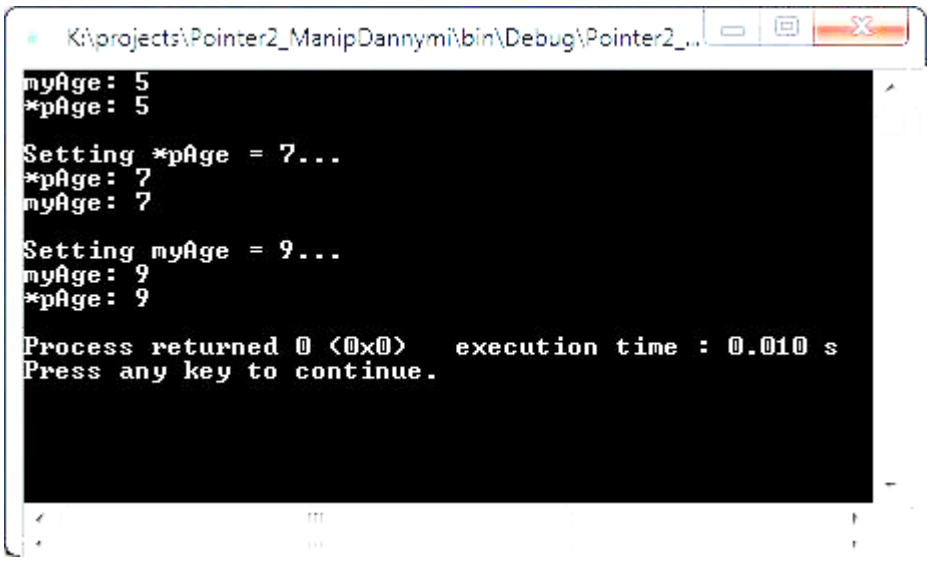
Распределение памяти представлено здесь, конечно, схематически, но оно иллюстрирует концепцию хранения адреса переменной в указателе.

Манипулирование данными с помощью указателей

После того как указателю присвоен адрес какой-либо переменной, его можно использовать для работы со значением этой переменной. В листинге 8.2 показан пример обращения к значению локальной переменной через указатель на нее.

```
// Листинг 8.2. Использование указателей
#include <iostream>
using namespace std;

typedef unsigned short int USHORT;
using namespace     std;

int main()
{
    USHORT     myAge; // переменная
    USHORT     * pAge = 0; // указатель

    myAge =    5;

    cout <<    "myAge: " << myAge << endl;
    pAge =     &myAge;
    cout <<    "*pAge: " << *pAge << endl << endl;

    cout <<    "Setting *pAge = 7..." << endl;
    *pAge =    7;

    cout <<    "*pAge: " << *pAge << endl;
    cout <<    "myAge: " << myAge << endl << endl;

    cout <<    "Setting myAge = 9..." << endl;
    myAge =    9;
```

```
    cout <<    "myAge: " << myAge << endl;
    cout <<    "*pAge: " << *pAge << endl;

    return     0;
}
```



Анализ

В программе объявлены две переменные: myAge типа unsigned short и pAge, являющаяся указателем на этот тип. В строке 14 переменной pAge присваивается значение 5, а в строке 16 это значение выводится на экран.

Затем в строке 17 указателю pAge присваивается адрес переменной myAge. В строке 18 при помощи оператора * извлекается значение по ссылке на указатель pAge и выводится на экран, демонстрируя, что по этому адресу можно извлечь значение переменной myAge, которое равно 5.

В строке 21 по адресу в указателе pAge размещается число 7. Впоследствии оказывается, что значение переменной myAge волшебным образом изменилось на 7, хотя непосредственно к этой переменной никто не обращался. Убедиться в этом можно после вывода значений на экран в строках 23 и 24. Здесь при обращении к значению также используется символ *, но на сей раз — для косвенного доступа.

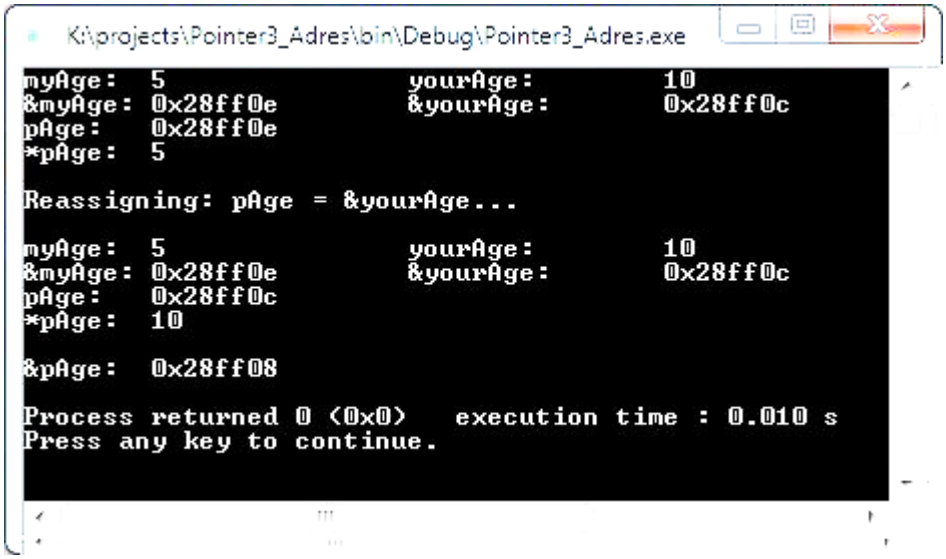
В строке 27 переменной myAge присваивается значение 9. Затем происходит обращение к ее значению непосредственно через переменную (в строке 29) и косвенно, через указатель на нее (в строке 30).

Адреса

Указатели позволяют манипулировать адресами, не обращая внимания на их реальные значения. Ранее уже говорилось, что когда адрес переменной присваивается указателю, то на самом деле он содержит реальный адрес именно этой переменной. Теперь пришло время проверить это на практике. Листинг 8.3 предоставляет такую возможность.

```
// Листинг 8.3. Что хранится в указателях?
#include <iostream>
using namespace std;

int main()
{
    unsigned short int myAge = 5, yourAge = 10;
```

```
    // указатель
    unsigned short int * pAge = &myAge;

    cout << "myAge:\t" << myAge
         << "\t\tyourAge:\t" << yourAge << endl;

    cout << "&myAge:\t" << &myAge
        << "\t&yourAge:\t" << &yourAge << endl;

    cout << "pAge:\t" << pAge << endl;
    cout << "*pAge:\t" << *pAge << endl;

    cout << "\nReassigning: pAge = &yourAge..." << endl << endl;
    pAge = &yourAge;     // переприсвоить указатель

    cout << "myAge:\t" << myAge
         << "\t\tyourAge:\t" << yourAge << endl;

    cout << "&myAge:\t" << &myAge
         << "\t&yourAge:\t" << &yourAge << endl;

    cout << "pAge:\t" << pAge << endl;
    cout << "*pAge:\t" << *pAge << endl;

    cout << "\n&pAge:\t" << &pAge << endl;

    return 0;
}
```



Анализ

На другом компьютере результат может получиться иным.

В строке 9 объявлены две переменные типа unsigned short — myAge и yourAge. Далее в строке 12 объявлен указатель на этот тип (pAge), который инициализируется адресом переменной myAge.

В строках с 14-ой по 18-ую значения и адреса переменных pAge и ту Аде выводятся на экран. В строке 20 на экран выводится содержимое переменной pAge, которое является истинным адресом переменной myAge. В строке 21 на экран выводится результат ссылки (взятия значения) на адрес, содержащийся в указателе pAge, который представляет собой значение переменной myAge, а именно — число 5.

В этом и состоит сущность указателей. Строка 20 демонстрирует, что pAде хранит адрес переменной myAge, а в строке 21 показано, как получить значение, хранимое в переменной myAge, сославшись на указатель, содержащий ее адрес. Прежде чем продолжить изучение книги, удостоверьтесь, что четко понимаете сущность и назначение указателя.

В строке 25 указателю pAде присваивается адрес переменной yourAge, а затем их значения и адреса выводятся на экран. Проанализировав результат выполнения программы, можно убедиться, что теперь указатель pAде содержит адрес переменной yourAge, и ссылка на него возвращает, соответственно, значение переменной yourAge.

Строка 36 выводит на экран адрес самого указателя pAде. Как и любая другая переменная, указатель имеет адрес, значение которого может храниться в другом указателе. О хранении в указателе адреса другого указателя речь пойдет несколько позже.

Для чего нужны указатели

В предыдущих разделах процедура присвоения указателю адреса другой переменной была рассмотрена подробно. Но на практике такое использование указателей встречается достаточно редко. К тому же зачем задействовать еще и указатель, если значение уже хранится в переменной? Рассмотренные выше примеры приведены только для демонстрации механизма работы указателей. Теперь, разобравшись в синтаксисе применения указателей, можно приступить к их реальному применению. Наиболее часто указатели применяются в следующих случаях:

- для манипулирования данными в динамически распределяемой памяти;
- для доступа к переменным-членам и функциям класса;
- для передачи данных между функциями по ссылке.

Стек и динамически распределяемая память(Heap)

Локальные переменные и параметры функций размещаются в стеке. Объектный код программ размещается в сегментах, а глобальные переменные — в области глобальных переменных. Регистры используются для внутренних целей процессора, например, для контроля вершины стека и указателя команд. Остальная часть памяти составляет так называемую динамически распределяемую память, или адресуемую память, или **heap** (кучу).

Особенностью локальных переменных является то, что после выхода из функции, в которой они были объявлены, память, выделенная для их хранения, освобождается, а значения переменных уничтожаются. Глобальные переменные позволяют частично решить эту проблему ценой неограниченного доступа к ним из любой точки программы, что значительно усложняет восприятие текста программы. Использование динамической памяти полностью решает обе проблемы.

Динамически распределяемую память можно представить как огромный массив последовательно пронумерованных ячеек, предназначенных для хранения данных. В отличие от стека, ячейкам свободной памяти нельзя присвоить имя, но можно, зарезервировав определенное количество ячеек, запомнить адрес первой из них в указателе.

Чтобы лучше понять изложенное выше, рассмотрим такой пример. Допустим, существует номер телефона службы заказов товаров по почте. Этот номер можно поместить в память телефона, а листок бумаги, на котором он был записан, выбросить. Нажимая на соответствующую кнопку телефона, можно соединяться со службой заказов, не имея понятия ни о номере, ни об адресе этой службы, поскольку для доступа достаточно помнить, какую именно кнопку надо нажать. Служба заказов в этом случае представляет собой данные в динамической памяти. Необязательно знать, где именно находятся данные, нужно знать, как их найти. Для обращения к данным используется их адрес, роль которого играет номер телефона службы доставки. Причем помнить адрес (или номер) необязательно — достаточно лишь записать его значение в указатель (или телефон). Указатель позволяет обращаться к данным, забыв о подробностях.

Когда функция возвращает значение, стек очищается автоматически. Когда область видимости локальных переменных заканчивается, они также удаляются из стека. Но динамическая память не очищается до завершения самой программы. Поэтому ответственность за освобождение всей памяти, зарезервированной для всех использованных данных, ложится на программиста.

Важным преимуществом динамической памяти является то, что выделенная в ней область не может использоваться в других целях до тех пор, пока не будет освобождена явно. Поэтому, если во время работы функции в динамической памяти выделяется область, ее можно использовать даже по завершении работы функции.

Недостатком динамической памяти является то, что выделенные в ней участки остаются зарезервированными до тех пор, пока они не будут освобождены явно. Если этого не сделать, то области памяти так и останутся занятыми, что через какое-то время может привести к исчерпанию ресурсов системы.

Еще одним преимуществом динамического выделения памяти по сравнению с глобальными переменными является то, что доступ к данным можно получить только из тех функций, которые обладают доступом к указателю, хранящему нужный адрес. Это "позволяет жестко контролировать характер манипулирования данными, а также избегать нежелательного или случайного их изменения. Для этого необходимо создать указатель на распределяемую область динамической памяти и передать его соответствующей функции.

Оператор new

Для выделения участка в динамически распределяемой области памяти используется ключевое слово **new**. После ключевого слова new следует указать тип объекта, который будет размещен в памяти. Это необходимо для того, чтобы компилятор мог определить размер области памяти, необходимой для размещения объекта. Следовательно, выражение new unsigned short int выделит два байта динамической памяти, а выражение new long — четыре.

В качестве результата оператор new возвращает адрес выделенного фрагмента памяти, который должен быть присвоен указателю. Например, чтобы создать в динамически распределяемой памяти переменную типа unsigned short, необходимо написать:

```
unsigned short int * pPointer;
pPointer = new unsigned short int;
```

Безусловно, инициализировать указатель можно сразу в момент его создания:

```
unsigned short int * pPointer = new unsigned short int;
```

В любом случае pPointer указывает теперь на переменную типа unsigned short int, размещенную в динамически распределяемой памяти. Его можно использовать как любой другой указатель на переменную и передавать с его помощью значения в эту область памяти. Например:

```
*pPointer = 72;
```

Эту строку можно прочитать так: "Присвоить число 72 значению указателя pPointer" или "Разместить число 72 в той области динамически распределяемой памяти, на которую указывает pPointer".

Оператор delete

По завершении работы с выделенной областью памяти ее нужно освободить. Для этого применяется оператор **delete**, после которого следует имя указателя. Оператор delete освобождает область памяти, на которую указывает указатель.

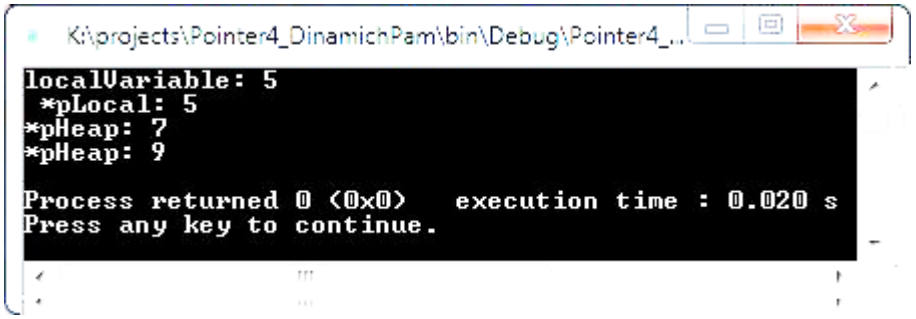
Помните, что сам указатель, в отличие от области памяти, на которую он указывает, является локальной переменной. Поэтому, когда объявившая указатель функция завершает работу, указатель выходит из области видимости, а содержащийся в нем адрес теряется. Поскольку распределенная с помощью оператора new память автоматически не освобождается, в случае потери адреса ее не удастся ни удалить, ни использовать. Такой участок памяти становится абсолютно недоступным, а подобная ситуация называется утечкой памяти.

Это название очень точно отражает сложившуюся ситуацию, поскольку блокированные участки памяти не могут быть восстановлены до завершения программы, и если такое случится в каком-либо цикле, то свободная память компьютера утечет, как вода в дыру (обычно до конца).

Для освобождения выделенной памяти используется ключевое слово delete, например: delete pPointer;

При удалении указателя с помощью оператора delete происходит реальное освобождение участка памяти, адрес которого находится в указателе. Иными словами, отдается команда: "Вернуть в динамически распределяемую память участок, на который указывает этот указатель". Но сам указатель остается (ведь это обычная переменная), и ему может быть передан на хранение другой адрес. Листинг 8.4 демонстрирует размещение переменной в динамической памяти, ее использование и удаление.

```
// Создание, использование и удаление указателей Листинг 8.4.
#include <iostream>
using namespace std;
int main()
{
    int localVariable = 5;
    int * pLocal= &localVariable;
    int * pHeap = new int;
    *pHeap = 7;

    cout << "localVariable: " << localVariable << endl;
    cout << " *pLocal: " << *pLocal << endl;
    cout << "*pHeap: " << *pHeap << endl;
    delete pHeap;

    pHeap = new int;
    *pHeap = 9;

    cout << "*pHeap: " << *pHeap << endl;
    delete pHeap;

    return 0;
}
```



Анализ

В строке 7 объявляется и инициализируется локальная переменна^ localVariable. Затем объявляется указатель, которому присваивается адрес этой переменной (строка 8). В строке 9 объявляется другой указатель (pHeap), который инициализируется результатом операции new int. Таким образом, в динамически распределяемой памяти выделяется пространство для переменной типа int. В строке . этому участку памяти присваивается значение 7.

Строка 11 выводит на экран значение локальной переменной, а строка 12 — значение, на которое указывает pLocal. Как и ожидалось, они одинаковы. Строка 13 выводит на экран значение, на которое указывает pHeap. Это доказывает, что значение присвоенное в строке 10, вполне доступно.

В строке 14 оператор delete освобождает участок динамически распределяемо памяти, занятый в строке 9. Это не только освобождает память, но и ликвидируй связь указателя с этим участком. Теперь указатель pHeap пуст и пригоден для запис адреса другого участка памяти. Это осуществляется в строках 15 и 16, а строка 17 демонстрирует результат. Строка 18 вновь освобождает этот участок памяти.

Несмотря на то что строка 18 является избыточной (завершение программы сам: по себе освободило бы эту область памяти), правила хорошего тона требуют освобождать все распределенные участки памяти явно. Если в программу будут внесены изменения или она будет расширена, либо этот участок кода будет использован в друто'-месте, то забытые указатели станут причиной серьезных проблем.

Арифметические действия с указателями

С указателями может выполняться ограниченное количество арифметических операций. Указатель можно увеличивать (++), уменьшать (--), складывать с указателем целые числа (+ или +=), вычитать из него целые числа (- или -=) или вычитать один указатель из другого.

Допустим, что объявлен массив `int v[10]` и его первый элемент находится в памяти в ячейке 3000. Допустим, что указателю `vPtr` было присвоено начальное значение путем указания на `v[0]`, т.е. значение `vPtr` равно 3000.

```
vPtr = v;
vPtr = &v[0];
```

Когда целое складывается или вычитается из указателя, указатель не просто увеличивается или уменьшается на это целое, но это целое предварительно умножается на размер объекта, на который ссылается указатель. Количество байтов зависит от типа данных. Например, оператор

```
vPtr += 2;
```

выработал бы значение 3008 (3000 + 2 * 4) в предположении, что целое хранится в 4 байтах памяти. В массиве v указатель `vPtr` теперь указал бы на `v[2]`.

Если указатель увеличивается или уменьшается на 1, можно использовать операции инкремента или декремента. Каждый из операторов

```
++ vPtr;
vPtr++;
```

увеличивает указатель так, что он указывает на следующий элемент массива.

Каждый из операторов

```
-- vPtr;
vPtr--;
```

уменьшает указатель так, что он указывает на предыдущий элемент массива.

Переменные указатели модно вычитать один из другого. Например, если `vPtr` содержит ячейку 3000, а `v2Ptr` содержит адрес 3008, оператор

```
x = v2Ptr – vPtr;
```

присвоит x значение разности номеров элементов массива, на которые указывают `vPtr` и `v2Ptr`, в данном случае 2.

Типичная ошибка: вычитание или сравнение двух указателей, которые не ссылаются на элементы одного и того же массива.

Указатель можно присваивать другому указателю, если оба указателя имеют одинаковый тип. В противном случае нужно использовать операцию приведения типа, чтобы преобразовать значение указателя в правой части присваивания к типу указателя в левой части присваивания. Исключением из этого правила является указатель на void (т.е. `void*`), который является общим указателем, способным представлять указатели любого типа. Указателю на void можно присваивать все типы указателей без приведения типа. Однако указатель на void не может быть присвоен непосредственно указателю другого типа – указатель на void сначала должен быть переведен к типу соответствующего указателя.

Указатель `void*` не может быть разыменован. Например, компилятор знает, что указатель на `int` ссылается на 4 байта памяти на машине с 4-байтовыми целыми, но указатель на void просто содержит ячейку памяти для неизвестного типа данных – точное количество байтов, на которое ссылается указатель, неизвестно компилятору. Компилятор должен знать тип данных, чтобы определить количество байтов, которое должно быть разыменовано для определенного указателя. В случае указателя на void это количество байтов не может быть определено из типа.

Указатели можно сравнивать, используя операции проверки равенства и отношения, но такое сравнение бессмысленно, если указатели не указывают на элементы одного и того же массива. При сравнении указателей сравниваются адреса, хранимые в указателях. Сравнение двух указателей, указывающих на один и тот же массив, может показать, например, что один указатель указывает на элемент массива с более высоким номером, чем другой указатель. Типичным использованием сравнения указателя является определение, равен ли указатель 0.

Способы обращения к функциям: вызов по значению и вызов по ссылке.

Параметры, передаваемые функции, помешаются в стек. Если функции передается значение как ссылка (с помощью либо указателей, либо ссылок), то в стек помещается не сам объект, а его адрес.

В действительности в некоторых компьютерах адрес хранится в специальном регистре, а в стек ничего не помещается. В любом случае компилятору известно, как добраться до исходного объекта, и при необходимости все изменения производятся прямо над объектом, а не над его временной копией.

При передаче объекта как ссылки функция может изменять объект, просто ссылаясь на него.

В листинге 5.5 демонстрируется, что после обращения к функции swap() значения в вызывающей функции не изменяются.

Листинг 9.5. Демонстрация передачи по значению

```
1:    // Листинг 9.5. Передача параметров как значений
2:
3:    #include <iostream.h>
4:
5:    void swap(int x, int y);
6:
7:    int main()
8:    {
9:      int x = 5, y = 10;
10:
11:       cout << "Main. Before swap, x: " << x << " y: " << y << "\n";
12:       swap(x,y);
13:       cout << "Main. After swap, x: " << x << " y: " << y << "\n";
14:          return 0;
15:    }
16:
17:    void swap (int x, int y)
18:    {
19:      int temp;
20:
21:      cout << "Swap. Before swap, x: " << x << " y: " << y << "\n";
22:
23:      temp = x;
24:      x = y;
25:      y = temp;
26:
27:      cout << "Swap. After swap, x: " << x << " y: " << y << "\n";
28:
29:    }
```

Передавая указатель, мы передаем адрес объекта, а следовательно, функция может манипулировать значением, находящимся по этому переданному адресу. Чтобы заставить функцию swap() изменить реальные значения с помощью указателей, ее нужно объявить так, чтобы она принимала два указателя на целые значения. Затем путем разыменования указателей значения переменных x и y будут на самом деле меняться местами. Эта идея демонстрируется в листинге 9.6.

Листинг 9.6. Передача аргументов как ссылок с помощью указателей

```
1:      // Листинг 9.6. Пример передачи аргументов как ссылок
2:
3:      #include <iostream.h>
4:
5:      void swap(int *x, int *y);
6:
7:      int main()
8:      {
9:        int x = 5, y = 10;
10:
11:        cout << "Main. Before swap, x: " << x << " y: " << y << "\n";
12:        swap(&x,&y);
13:        cout << "Main. After swap, x: " << x << " y: " << y << "\n";
14:     return 0;
15:     }
16:
17:     void swap (int *px, int *py)
18:     {
19:       int temp;
20:
21:       cout << "Swap. Before swap, *px: " << *px << " *py: " << *py << "\n";
22:
23:       temp = *px;
24:       *px = *py;
25:       *py = temp;
26:
27:       cout << "Swap. After swap, *px: " << *px << " *py: " << *py << "\n";
28:
29:     }
```

Приведенная выше программа, конечно же, работает, но синтаксис функции swap() несколько громоздок. Во-первых, необходимость неоднократно разыменовывать указатели внутри функции swap() создает благоприятную почву для возникновения ошибок, кроме того, операции разыменовывания трудно читаются. Во-вторых, необходимость передавать адреса переменных из вызывающей функции нарушает принцип инкапсуляции выполнения функции swap().

Суть программирования на языке C++ состоит в сокрытии от пользователей функции деталей ее выполнения. Передача параметров с помощью указателей перекладывает ответственность за получение адресов переменных на вызывающую функцию, вместо того чтобы сделать это в теле вызываемой функции. Другое решение той же задачи предлагается в листинге 9.7, в котором показана работа функции swap() с использованием ссылок.

Листинг 9.7. Та же функция swap(), но с использованием ссылок

```
1:      // Листинг 9.7. Пример передачи аргументов
2:      // ссылок с помощью ссылок!
3:
4:      #include <iostream.h>
5:
6:      void swap(int &x, int &y);
7:
8:      int main()
9:      {
10:          int x = 5, y = 10;
11:
12:          cout << "Main. Before swap, x: " << x << " y: " << y << "\n";
13:          swap(x,y);
14:          cout << "Main. After swap, x: " << x << " y: " << y << "\n";
15:          return 0;
16:     }
17:
18:         void swap (int &rx, int &ry)
19:         {
20:           int temp;
21:
22:           cout << "Swap. Before swap, rx: " << rx << " ry: " << ry << "\n";
23:
24:           temp = rx;
25:           rx = ry;
26:           ry = temp;
27:
28:           cout << "Swap. After swap, rx: " << rx << " ry: " << ry << "\n";
29:
30:         }
```

Взаимосвязи между указателями и массивами

Предположим, что объявлены массив целых чисел b[5] и целая переменная указатель bPtr. Поскольку имя массива (без индекса) является указателем на первый элемент массива, мы можем задать указателю bPtr адрес первого элемента массива b с помощью оператора

bPtr = b;

Это эквивалентно присваиванию адреса первого элемента массива следующим образом

bPtr = &b[0];

Сослаться на элемент массива b[3] можно с помощью выражения указателя

**(bPtr + 3)*

В приведенном выражении 3 является **смещением** указателя. Когда указатель указывает на начало массива, смещение показывает, на какой элемент массива должна быть ссылка, так что значение смещения эквивалентно индексу массива. Предыдущую запись называют **записью указатель-смещение**. Скобки необходимы, потому что приоритет * выше, чем приоритет +. Т.к. элемент массива может быть указан указателем выражением, адрес

&b[3]

может быть записан также выражением указателем

bPtr + 3

Сам массив можно рассматривать как указатель и использовать в арифметике указателей. Например, выражение

**(b + 3)*

тоже ссылается на элемент массива b[3].

Указатели можно индексировать точно так же, как и массивы. Например, выражение

bPtr[1]

ссылается на элемент массива b[1]; это выражение рассматривается как **запись указатель-индекс**.

Т.к. что имя массива, по существу, является постоянным указателем; оно всегда указывает на начало массива, выражение

b += 3

не разрешено, потому что оно пытается модифицировать значение имени массива с помощью арифметической операции над указателем.

Использование оператора new для создания динамических массивов

Предположим, например, что создается программа, для которой массив может либо потребоваться, либо нет, в зависимости от информации, получаемой в ходе выполнения программы. Если создается массив путем объявления, часть памяти будет отведена под него уже при компиляции программы. Независимо от того, использует ли программа созданный массив или нет, память он занимать будет в любом случае. Распределение массива в процессе компиляции называется статическим связыванием, что означает, что массив встроен в программу во время компиляции. При необходимости можно создавать массив с помощью оператора new во время выполнения программы, а если нет такой необходимости, то создавать его не нужно. Можно также выбрать размер массива в процессе выполнения программы. Это называется динамическим связыванием, что означает, что массив будет создан в ходе выполнения программы. Такой массив называется динамическим. При статическом связывании необходимо определить размер массива во время создания программы. При динамическом связывании программа принимает решение о размере создаваемого массива в ходе своего выполнения.

Создать динамический массив в C++ просто. Необходимо указать оператору new тип элементов массива и их желаемое количество. Синтаксис требует, чтобы было указано имя типа, а за ним – количество элементов в квадратных скобках. Например, если нужен массив из десяти переменных типа int, выполните следующее:

*int * psome = new int [10]; // получение блока из 10 значений типа int*

Оператор new возвращает адрес первого элемента выделенного блока. В приведенном примере это значение присваивается указателю psome. Необходимо согласовать количество вызовов оператора new с количеством вызовов оператора delete, чтобы тогда, когда программа закончит использование этого блока памяти, она его освободила.

Когда для создания массива используется оператор new, следует употребить другую форму оператора delete, которая указывает, что освобождается именно массив:

delete [] psome; // освобождается динамический массив

Квадратные скобки свидетельствуют о том, что необходимо освободить целый массив, а не только элемент, на который указывает указатель.

Например,

```
int * pi = new int;
short * ps = new short [500];
delete [] pi; // результат не определен, не делайте так
delete ps; // // результат не определен, не делайте так
```

Правила, которые необходимо соблюдать при использовании операторов new и delete:

1. Не используйте оператор delete для освобождения памяти, которую не распределяли с помощью оператора new.
2. Не используйте оператор delete для освобождения одного и того же блока памяти дважды подряд.

3. Используйте оператор delete [], если был применен оператор new [] для распределения массива.

4. Используйте оператор delete (без квадратных скобок), если был применен оператор new для распределения одного элемента.

5. Безопасно применять оператор delete к нулевому указателю (ничего не происходит).