УКАЗАТЕЛИ (продолжение)

Что такое указатель

Указатель (pointer) — это переменная, в которой хранится адрес другого объекта (как правило, другой переменной). Например, если одна переменная содержит адрес другой переменной, говорят, что первая переменная ссылается (point) на вторую.

В С++ указатели представляют собой составной тип данных.

Объявление указателей

Переменная, хранящая адрес ячейки памяти, должна быть объявлена как указатель. Объявление указателя состоит из имени базового типа, символа * и имени переменной. Общая форма этого объявления такова.

*тип_указателя * имя_указателя;*

Здесь *тип_указателя* означает базовый тип указателя. Им может быть любой допустимый тип. Базовый тип указателя определяется типом переменной, на которую он может ссылаться. С формальной точки зрения указатель любого типа может ссылаться на любое место в памяти. Однако операции адресной арифметики тесно связаны с базовым типом указателей, поэтому очень важно правильно их объявить.

Пробелы вокруг операции * не обязательны. Верными являются следующие строки:

```
int   *ptr;  // в стиле С
int*   ptr;  // в стиле С++
int*ptr;     // для компилятора пробелы не важны
```

Однако учтите, что следующее объявление создает один указатель (p1) и одну обычную неременную типа int (p2):

```
int* p1, p2;
```

Знак * должен быть помещен возле каждой переменной типа указателя.

Инициализировать указатель можно в операторе объявления. В этом случае инициализируется указатель, а не значение, на которое он указывает.

Пример.

```
int a =6;    // объявление переменной
int * p_a;   // объявление указателя на int
p_a = &a;    // присвоить адрес int указателю
// Выразить значения двумя способами
cout << a << *p_a;    // 6  6
// Выразить адреса двумя способами
cout << &a << p_a;    // 0x23ef20 0x23ef20 - формат вывода может быть другим
// Изменить значение через указатель
*p_a = *p_a + 1;
cout << a;   // 7
```

Как видите, переменная `a` типа int и переменная-указатель `p_a` — это две стороны одной монеты. Переменная `a` в первую очередь представляет значение, а для получения его адреса используется операция `&`, в то время как `p_a` представляет адрес, а для получения значения применяется операция `*`.

Поскольку `p_a` указывает на `a`, конструкции `*p_a` и `a` полностью эквивалентны. Вы можете использовать `*p_a` точно так же, как используете переменную типа `int`.

Однако в С++ после связывания указателя с определённой областью памяти нельзя изменить тип указателя, не используя явное приведение типов:

```
int *p;
double x=4;
p=&x;         // Ошибка компиляции:
              // error: cannot convert 'double*' to 'int*' in assignment
p=(int*)&x; // OK - в явном виде указали, что будем использовать
              // именованную область памяти как память, хранящую
              // целочисленное значение
```

Опасность, связанная с указателями

Важно!!! 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.

Ошибки подобного рода порождают самое непредсказуемое поведение программы и такие ошибки очень трудно отследить.

Всегда инициализируйте указатель, чтобы определить точный и правильный адрес, прежде чем применять к нему операцию разыменования (*).

Указатели и числа

Указатели — это не целочисленные типы, даже, несмотря на то, что компьютеры обычно выражают адреса целыми числами.

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. 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.

Выделение памяти с помощью операции new

До сих пор мы инициализировали указатели адресами переменных, которые являются *именованной памятью*, выделенной 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. В языке С память можно выделять с помощью библиотечной функции malloc(). Ее можно применять и в C++, но язык C++ также предлагает лучший способ — операцию ***new***:

```
имяТипа * имя_указателя = new имяТипа;
```

Пример.

```
int * pt = new int;           // выделение пространства для double
*pt = 1001;                   // сохранение в нем значения
cout << sizeof(pt) << sizeof(*pt);  // 8  4
```

Динамические значения (значения доступные через указатели) размещаются не в стеке и не в сегменте данных. Память, выделяемая операцией `new`, находится в области, называемой *кучей* или *свободным хранилищем*.

Может случиться так, что у компьютера не окажется достаточно доступной памяти, чтобы удовлетворить запрос new. Когда такое происходит, операция new обычно реагирует генерацией исключения. В более старых реализациях new возвращает значение 0.

В С++ указатель со значением 0 называется *null-указателем (нулевым указателем)* – ***nullptr***. С++ гарантирует, что нулевой указатель никогда не указывает на допустимые данные, поэтому он часто используется в качестве признака неудачного завершения операций или функций, которые в противном случае должны возвращать корректные указатели.

Освобождение памяти с помощью операции delete

Использование операции new для запрашивания памяти, когда она нужна — одна из сторон пакета управления памятью С++. Второй стороной является операция delete, которая позволяет вернуть память в пул свободной памяти, когда работа с ней завершена. Это — важный шаг к максимально эффективному использованию памяти. Память, которую вы возвращаете, или освобождаете, затем может быть повторно использована другими частями программы. Операция delete применяется с указателем на блок памяти, который был выделен операцией new:

```
int * ps = new int; // выделить память с помощью операции new
...         // работа с выделенной памятью
delete ps; // по завершении освободить память
           // с помощью операции delete
```

Это освобождает память, на которую указывает ps, но не удаляет сам указатель ps. Вы можете повторно использовать ps — например, чтобы указать на другой выделенный new блок памяти. Вы всегда должны обеспечивать сбалансированное применение new и delete; в противном случае вы рискуете столкнуться с таким явлением, как *утечка памяти*, т.е. ситуацией, когда память выделена, но более не может быть использована. Если утечки памяти слишком велики, то попытка программы выделить очередной блок может привести к ее аварийному завершению.

Вы не должны пытаться освобождать блок памяти, который уже был однажды освобожден. Стандарт С++ гласит, что результат таких попыток не определен, а это значит, что последствия могут оказаться любыми. Кроме того, вы не можете с помощью операции delete освобождать память, которая была выделена посредством объявления обычных переменных:

```
int * ps = new int; // нормально
delete ps;          // нормально
delete ps;          // теперь - не нормально!
                    // Вызовет ошибку времени выполнения программы
int jugs = 5;       // нормально
int * pi = &jugs;   // нормально
delete pi;          // не допускается, память не была выделена new
                    // Вызовет ошибку времени выполнения программы
```

Обратите внимание, что обязательным условием применения операции delete является использование ее с памятью, выделенной операцией new. Но это не значит, что вы обязаны применять тот же указатель, который был использован с new – просто нужно задать тот же адрес:

```
int * ps = new int; // выделение памяти
int * pq = ps;      // установка второго указателя на тот же блок
delete pq;          // вызов delete для второго указателя
```

Обычно не стоит создавать два указателя на один и тот же блок памяти, т.к. это может привести к ошибочной попытке освобождения одного и того же блока дважды.

Использование операции new для создания динамических массивов

Использование операции new более типично с крупными фрагментами данных, такими как массивы, строки и структуры. Именно в таких случаях операция new является полезной.

Если вы создаете, например, массив простым объявлением, пространство для него распределяется раз и навсегда — во время компиляции. Будет ли востребованным массив в программе или нет — он все равно существует и занимает место в памяти. Распределение массива во время компиляции называется *статическим связыванием* и означает, что массив встраивается в программу во время компиляции. Но с помощью new вы можете создать массив, когда это необходимо, во время выполнения программы, либо не создавать его, если потребность в нем отсутствует. Или же вы можете выбрать размер массива уже после того, как программа запущена.

Это называется *динамическим связыванием* и означает, что массив будет создан во время выполнения программы. Такой массив называется *динамическим массивом*. При статическом связывании вы должны жестко закодировать размер массива во время написания программы. При динамическом связывании программа может принять решение о размере массива во время своей работы.

Для создания динамического массив на C++ необходимо сообщить операции `new` тип элементов массива и требуемое количество элементов:

```
int * psome = new int [10]; // получение блока памяти из 10 элементов типа int
```

Общая форма выделения и назначения памяти для массива выглядит следующим образом:

```
имя_типа * имя_указателя = new имя_типа [количество_элементов];
```

Операция `new` возвращает адрес первого элемента в блоке.

Однако использовать операцию new при работе со статическим массивом считается ошибкой, т.к. они не размещаются в динамической области памяти:

```
int m[]=new int [3]; // Error: array must be initialized
                     // with a brace-enclosed initializer
```

Как всегда, вы должны сбалансировать каждый вызов `new` соответствующим вызовом `delete`, когда программа завершает работу с этим блоком памяти. Однако использование `new` с квадратными скобками для создания массива требует применения альтернативной формы `delete` при освобождении массива:

```
delete [] psome; // освобождение динамического массива
```

Присутствие квадратных скобок сообщает программе, что она должна освободить весь массив, а не только один элемент, на который указывает указатель. Обратите внимание, что скобки расположены между `delete` и указателем. При этом размер массива указывать не нужно.

Итак: при использовании `new` и `delete` необходимо придерживаться перечисленных ниже правил:

• Не использовать `delete` для освобождения той памяти, которая не была выделена `new`.

• Не использовать `delete` для освобождения одного и того же блока памяти дважды.

• Использовать `delete []`, если применялась операция `new[]` для размещения массива.

• Использовать `delete` без скобок, если применялась операция `new` для размещения отдельного элемента.

• Помнить о том, что применение `delete` к нулевому указателю является безопасным (при этом ничего не происходит).

```
int * pt = new int; // pt - указатель на динамическую переменную
short * ps = new short [500]; // ps - указатель на динамический массив
delete [] pt; // эффект не определен, не делайте так
delete ps;    // эффект не определен, не делайте так
```

Обратите внимание, что `psome` — это указатель на отдельное значение `int`, являющееся первым элементом блока. Отслеживать количество элементов в блоке возлагается на вас как разработчика. То есть, поскольку компилятор не знает о том, что `psome` указывает на первое из 10 целочисленных значений, вы должны писать свою программу так, чтобы она самостоятельно отслеживала количество элементов в процессе работы с этим масиивом.

На самом деле программе, конечно же, известен объем выделенной памяти, так что она может корректно освободить ее позднее, когда вы воспользуетесь операцией `delete[]`. Однако эта информация не является открытой; вы, например, не можете применить операцию `sizeof`, чтобы узнать количество байт в выделенном блоке:

```
cout << sizeof ps;  // 8 - размер адреса, но не блока памяти,
                    // на который он ссылается
```

Использование динамического массива

Размер и местоположение элементов динамического массива в памяти не известны в момент компиляции. Создаётся такой массив в процессе выполнения программы:

```
double * p3 = new double [3]; // пространство для 3х значений double
p3[0] = 0.2; // трактовать p3 как имя массива
p3[1] = 0.5
p3[2] = 0.8
cout << p3[1]; // вывод p3[1]
```

```
p3 = p3 + 1;    // увеличение указателя
cout << p3[0] << p3[1];
p3 = p3 - 1;    // возврат указателя в начало
delete [] p3;   // освобождение памяти
return 0;
```

Замечания к программе:

①Доступ к элементам p возможен с использованием нотации статических массивов посредством квадратных скобок. При этом:

```
p3[0] == *p  // т.к. p3 указывает на первый элемент массива,
             // то *p3 и есть значение первого элемента;
p3[1] == 0.5 // второй элемент массива, и т.д.
```

!!! С и С++ внутренне работают с массивами через указатели. Подобная эквивалентность указателей и массивов — одно из замечательных свойств С и С++. Указатель p3 используется, как если бы он был именем массива: p3[0] для первого элемента и т.д.

②Фундаментальное отличие между именем массива и указателем проявляется в следующей строке:

```
p3 = p3 + 1; // допускается для указателей, но не для имен массивов
```

Вы не можете изменить значение для имени массива. Но указатель — переменная, а потому ее значение можно изменить.

③Обратите внимание на эффект от добавления 1 к p3. Теперь выражение p3[0] ссылается на бывший второй элемент массива. То есть добавление 1 к p3 заставляет p3 указывать на второй элемент вместо первого. Вычитание 1 из значения указателя возвращает его назад, в исходное значение, поэтому программа может применить delete[] с корректным адресом.

Действительные адреса соседних элементов int отличаются на 2 или 4 байта (в зависимости от платформы), поэтому тот факт, что добавление 1 к p3 дает адрес следующего элемента, говорит о том, что арифметика указателей устроена специальным образом.

АРИФМЕТИКА УКАЗАТЕЛЕЙ

С формальной точки зрения указатель любого типа может ссылаться на любое место в памяти. Однако операции адресной арифметики тесно связаны с базовым типом указателей.

Операторы для работы с указателями

Как уже известно, существуют два специальных оператора для работы с указателями:

- *получения адреса* **&** - возвращает адрес своего операнда;
- *разыменования указателя* ***** - является антиподом оператора & и возвращает значение, хранящееся по указанному адресу.

Оба оператора являются унарными. Приоритет операторов & и * выше, чем приоритет арифметических операторов, за исключением унарного минуса (информация по приоритетам всех оператор С++ см. на https://ru.cppreference.com/w/cpp/language/operator_precedence).

Выражения с указателями

①Присваивание указателей

Указатель можно присваивать другому указателю.

```
int x;   int *p1, *p2;
p1 = &x;
p2 = pi; // Теперь на переменную x ссылаются оба указателя p1 и p2.
```

②Сложение и вычитание указателей

К указателям можно применять только две арифметические операции: сложение и вычитание.

Правила сложения и вычитания: при увеличении указатель ссылается на ячейку, в которой хранится следующий элемент базового типа, а при уменьшении он ссылается на предыдущий элемент. Т.е. указатели увеличиваются или уменьшаются на длину соответствующих переменных, на которые они ссылаются.

Исключение составляет указатель на символ (значение типа char), для которого сохраняются правила “обычной” арифметики, поскольку размер символов равен 1 байт.

Рассмотрим *особенности* приведённых правил на примере:

```
1   int a[2];        // статический массив
2   char * pa;
3   pa = (char*)a;  // т.к. a == (int * ) требуется приведение типов
4   a[0]=3;  a[1]=5;
5   cout << int(*pa) << " " << int(*(pa+1)) << " "
         << int(*(pa+2)) << " " << int(*(pa+3)) << " ";
6   pa = pa+4;
7   cout << int(*pa) << " " << int(*(pa+1)) << " "
         << int(*(pa+2)) << " " << int(*(pa+3)) << " ";
8   cout << int(pa[0]);
```

Предположим, что переменные располагаются в памяти, начиная с адреса 1000. Тогда после выполнения операторов 1-3 получим:

Адрес	*Содержимое памяти*	*Обращение через указатели*	*Обращение через нотацию массивов*
1000	?	← pa	a[0]
1001	?		
1002	?		
1003	?		
1004	?		a[1]
1005	?		
1006	?		
1007	?		

Далее (операторы 4 и 5) происходит заполнение памяти значениями и при помощи указателя pa оператор cout побайтно выводит содержимое памяти: 3 0 0 0 .

Адрес	*Содержимое памяти*	*Обращение через указатели*	*Обращение через нотацию массивов*
1000	3	← pa	a[0]
1001	0	pa+1	
1002	0	pa+2	
1003	0	pa+3	
1004	?		a[1]
1005	?		
1006	?		
1007	?		

Оператор 6 изменяет значения адреса, хранящееся в переменной pa на 4*1байт = 4 байта. В результате pa ссылается на второй элемент массива a, значение которого и выводит оператор 7:

5 0 0 0 .

Адрес	*Содержимое памяти*	*Обращение через указатели*	*Обращение через нотацию массивов*
1000	3		a[0]
1001	0		
1002	0		
1003	0		
1004	5	← pa=pa+4	a[1]
1005	0	pa+1	
1006	0	pa+2	
1007	0	pa+3	

Оператор 8 показывает использование указателя в качестве имени массива для доступа к его элементу:

```
pa[0];  // 5
```

Вывод из примера:

– Добавление к указателю единицы означает добавление *одной единицы хранения* типа указателя (увеличение адреса на `sizeof ТипУказателя`);

– При использовании указателя вместо имени массива, С++ осуществляет преобразование:

`имя_указателя[i]    в    *(имя_указателя + i)`

что позволяет взаимозаменять имена указателей и имена массивов.

Вопросы для самостоятельного изучения:

1. Что будет выведено на экран без приведения типа (см. код выше)?
```
pa = (char*)a;
cout << *(pa+4); // ???
```
2. А если изменить значение, на которое ссылается `pa`?
```
*pa = ' ';
cout << a[0];  // ???
```

Рассмотрим особенности правил сложения/вычитания указателей применительно к динамическими и статическими *массивам*:

```
int * b = new int [2]; // динамический массив
b[0]=4; b[1]=6;        // инициализация значениями
b = b+1;              // смещаемся на 4 байта к следующему элементу массива
cout << b[0] << b[-1]; // на экране  6 4
b--;                   // вернуть b исходное значение
delete [] b;           // для верного освобождения памяти

int a[2]={3,5};  // статический массив
// поскольку a – указатель, можно попробовать его увеличить на 1
// по аналогии с арифметикой указателей
a = a+1;
// Error: incompatible types in assignment of 'int*' to 'int [2]'
// Однако:
cout << *a << *(a+1); // на экране  3 5 - т.к. имя массива - указатель
// НО!!!
cout << *a+1; // на экране  4 – т.к. приоритет * выше, чем у +
              // сначала вычисляется *a==3, к которому потом добавляется 1
```

Выводы из примера:

– Подтверждение того, что система обозначения массивов – это замаскированное использование указателей.

– Адресом объекта большого размера (например, int – 4 байта) является адрес его первого байта.

– Добавление к имени динамического массива единицы означает увеличение адреса до адреса следующего элемента, а не просто до следующего байта. В общем случае, всякий раз, когда вы используете нотацию массивов, С++ выполняет следующее преобразование:

`имя_массива[i]    в    *(имя_массива + i)`

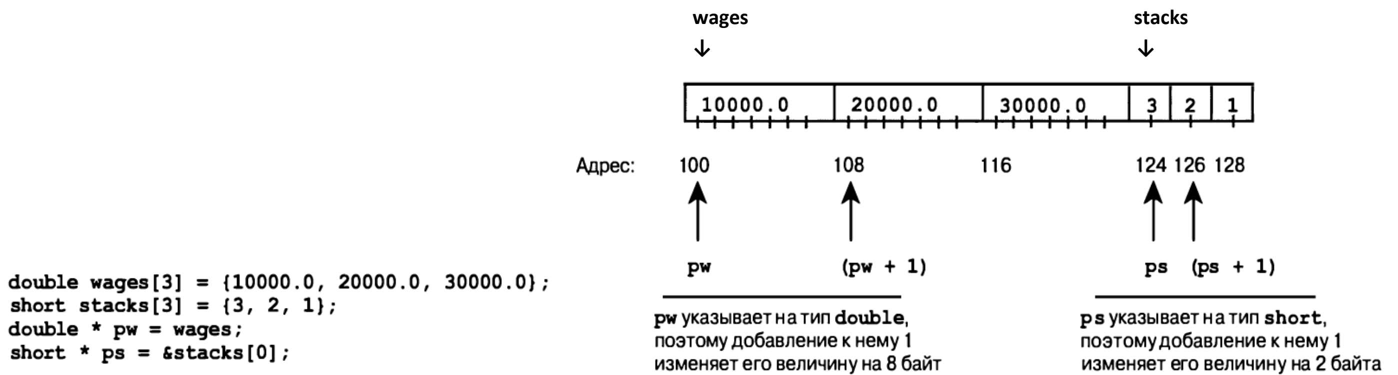
И, как уже было показано ранее, если вы используете указатель вместо имени массива, С++ осуществляет то же самое преобразование:

`имя_указателя[i]    в    *(имя_указателя + i)`

Таким образом, во многих отношениях имена указателей и имена массивов можно использовать одинаковым образом. Нотация квадратных скобок применима и там, и там. К обоим можно применять операцию разыменования (*). В большинстве выражений каждое имя представляет адрес.

– Добавление к имени статического массива единицы приводит к ошибке несоответствия типов, т.к. имя статического массива – неизменная величина

Ещё один графический пример на сложение указателей (**для самостоятельного изучения**).



③Сравнение указателей

Указатели можно сравнивать между собой. Например, следующий оператор, в котором сравниваются указатели `p` и `q`, является совершенно правильным.

```
if (p<q)
   cout << "Указатель p содержит меньший адрес, чем указатель q\n";
```

Однако базовые типы указателей при этом должны быть одинаковыми, иначе требуется явное приведение типов в выражении:

```
double *pd = new double;
char * pc  = new char;
cout << (pd<(double*)pc) << ((char*)pd<pc);
```

Как правило, указатели сравниваются между собой, когда они ссылаются на один и тот же объект, например массив. Рассмотрим в качестве примера организацию и работу со структурой «стек».

Стек — это список, в котором доступ к элементам осуществляется по принципу “первым вошел, последним вышел” (LIFO). Его часто сравнивают со стопкой тарелок на столе — нижняя тарелка будет снята последней. Стеки используются в компиляторах, интерпретаторах, программах обработки электронных таблиц и других системных утилитах.

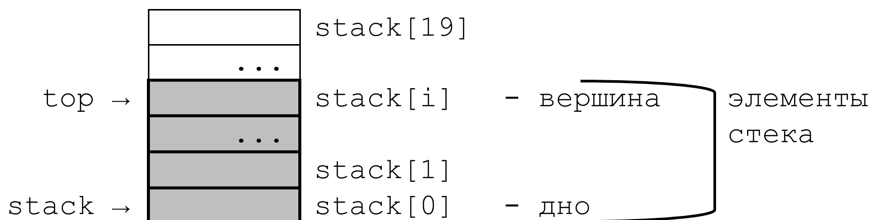
Для работы стека необходимы две функции: push() и pop(). Функция push() заносит элементы в вершину стек, а функция pop() – извлекает их оттуда.

1. Описание необходимых <u>структур данных</u>:
   - Для моделирования стека возьмём статический массив целочисленных значений ограниченной вместимости:

```
const int size=20;
int stack[size];
```

   - Необходим указатель на вершину стека:

```
int * top;
```



2. <u>Общение с пользователем</u> реализуем в интерактивном режиме в виде диалога с ситемой:

```
int ch;     // выбор пользователя
do {        // Перечисляем доступные действия:
 cout << "\nВыберите действие: \n"
      << "1 - Вывести содержимое стека\n"
      << "2 - Добавить элемент в стек\n"
      << "3 - Извлечь элемент из вершины стека\n"
      << "4 - Выход\n";
   // ожидаем выбор пользователя
```

```
    cin >> ch;
    switch(ch) {
        case 1: // выводим содержимое стека
                ...
                break;
        case 2:
                // добавляем элемент в вершину стека
                ...
                break;
        case 3:
                // извлекаем элемент из стека и выводим его значение на TV
                ...
                break;
    };
} while (ch<4);
```

3. Реализуем выбранные действия:

Для вывода содержимого стека необходимо просмотреть все его элементы последовательно, начиная с вершины. Процесс завершается, когда указатель на текущий элемент стека будет совпадать с его «дном» (последним элементом).

```
int * pt = top; // указатель на текущий элемент ссылается на вершину стека
cout << "\nСодержимое стека: вершина - ";
while (pt != stack) { //
   cout << *pt << ", "; // выводим текущий элемент стека
                        // как значение, на которое ссылается указатель pt
   pt = pt-1;           // перемещаемся к следующему элементу стека
   }
cout << "\b\b - дно \n";
```

Здесь имя массива `stack` рассматривается как указатель на его первый элемент и одновременно как указатель на дно стека. Поэтому признаком достижения дна стека является совпадение указателя на текущий элемент `pt` и указателя на дно стека `stack`, что проверяется сравнением:

```
pt != stack
```

В случае, если стек пуст, достаточно вывести соответствующее предупреждение пользователю:

```
  if (top != stack) {
    //  выводим содержимое стека
  } else
     cout << "\nСтек пуст!\n";
```

Признаком пустого стека является совпадение адресов, находящихся в `top` и `stack`, что проверяется сравнением указателей:

```
top != stack
```

4. Функция `Push()` помещает новый элемент `NewElem` в вершину стека. Это возможно только в том случае, если стек ещё не заполнен полностью. Признаком отсутствия свободного места в стеке является ссылка указателя вершины `top` на последний элемент массива `stack`, который расположен по адресу `stack+size`. Выражение `stack+size` даёт адрес последнего элемента массива `stack`, т.к. сложение указателя с числом приводит к его увеличению на `(sizeof int)*size` байт.

```
void Push(int NewElem){
  if (top < stack+size) { // проверяем есть ли место в стеке
    top = top+1;    // смещаем указатель стека к следующему пустому элементу
    *top = NewElem;// запоминаем значение нового элемента
  cout << "\nДобавили один элемент.\n";
  }
  else // в стеке нет места
     cout << "\nСтек полон!\n";
};
```

Вопросы для самостоятельного изучения:

а). А как ещё можно проверить, что указатель вершины `top` ссылается на последний элемент массива `stack`?

б). Какой смысл следующих выражений `&stack[size]`, `&(stack[size])`, `(&stack)[size]`, `&(*stack)` ?

5. Функция `Pop()` <u>извлекает элемент</u>, находящийся в вершине стека, и возвращает в вызывающую подпрограмму его значение. Это действие возможно только в том случае, если стек не пуст. Признаком пустоты стека является ссылка указателя его вершины на дно стека: `top==stack` .

```
int Pop(){
  if (top==stack)
     cout << "\nСтек пуст!\n";
  else {              // если в стеке есть элементы
     top=top-1;       // смещаем указатель вершины к предыдущему элементу
     return *(top+1); // и возвращаем значение «потерянного» элемента
  }
```

Вопрос для самостоятельного изучения:

а). Какой смысл имеет выражение `*top+1`?

6. «Собираем» все части программы вместе:

```
#include <iostream>
using namespace std;
const int size=20;  // ограничение на максимальное кол-во элементов в стеке

int stack[size] {}; // место хранения элементов стека,
                    // stack указывает на "дно" стека
int * top;          // указатель на вершину стека

void Push(int);
int Pop();
void OutStack();

int main()
{
int ch;     // выбор пользователя
top=stack; // сначала стек пуст
           // делать new int для top не надо,
           // т.к. моделируем стек статическим массивом
do {    // Перечисляем доступные действия:
 cout << "\nВыберите действие: \n"
      << "1 - Вывести содержимое стека\n"
      << "2 - Добавить элемент в стек\n"
      << "3 - Извлечь элемент из вершины стека\n"
      << "4 - Выход\n";
   cin >> ch;  // запоминаем выбор пользователя
   switch(ch) {
       case 1: // выводим содержимое стека
               OutStack(); break;
       case 2:
           // добавляем элемент в вершину стека по возможности
           cout << "Введите новый элемент: ";
           int Elem; cin >> Elem;
           Push(Elem);
           break;
       case 3:
           // извлекаем элемент из стека и выводим его значение на TV
           cout << "Извлекли элемент: " << Pop() << endl;
           break;
   };
} while (ch<4);
return 0;
}
```

```
void OutStack(){
  if (top != stack) {
     int * pt = top;
     cout << "\nСодержимое стека: вершина - ";
     while (pt != stack) {
              cout << *pt << ", ";
                 pt = pt-1;
     }
     cout << "\b\b - дно \n";
  } else
     cout << "\nСтек пуст!\n";
};

void Push(int NewElem){
  if (top < stack+size) {
     top = top+1;
     *top = NewElem;
     cout << "\nДобавили один элемент.\n";
  }
  else
     cout << "\nСтек полон!\n";
};

int Pop(){
  if (top==stack)
     cout << "\nСтек пуст!\n";
  else {
     top--;
     return *(top+1);
  }
  return 0;
};
```

Упражнение по программированию.

Измените приведённую программу, добавив пятое действие по подсчёту текущего количества элементов в стеке (выделены серым цветом в п. «Описание необходимых структур данных»).