Глава 6. Указатели и динамическая память

Что такое указатель?

Указатель – это переменная, содержащая в программе в качестве значения некоторый адрес: поименованной величины или просто ячейки памяти.

В отличие от базовых типов языка Си, указатель — это так называемый производный тип: формируемый из базового при помощи некоторых символов языка, в данном случае применяется символ * (звёздочка). Вот как выглядят простые объявления указателей в программе:

```
int *iPtr; // указатель с именем iPtr на значение типа int
char *cPtr; // указатель с именем cPtr на значение типа char
```

Именами типов в этих примерах являются последовательности *int* * и *char* *, соответственно. Так как величины могут занимать несколько соседних байт в памяти, а память адресуется побайтно, помимо «звёздочки» в указании типа переменной-указателя присутствует и тип самой величины, на которую "указывает" адрес. Важно также понимать, что пользоваться указателем допустимо лишь после того, как он получит значение в результате инициализации (которая в этих примерах отсутствует!) или значение будет присвоено в процессе работы программы.

Примеры инициализации указателей:

```
int k = 5, m = 3;    // целые переменные, на которые будем создавать указатели
int *kPtr = &k;      // указатель с именем kPtr на значение типа int в переменной k

int **pPtr = &kPtr; // а это – "указатель на указатель", так как kPtr - самостоятельная
                    // переменная типа int*, то у неё есть свой адрес, поэтому можно
                    // создать и указатель, который будет на нее указывать

int *kPtr2 = &k, *mPtr = &m; // тут в одном выражении объявляются два указателя,
                             // обратите внимание, как ставятся "звездочки" при этом;
                             // также подчеркнем: на одну и ту же переменную k
                             // никто не запрещает создать несколько указателей!

void *vPtr = &k;     // создаем указатель "на неизвестный тип" (void), для того, чтобы
                     // указываемым значением воспользоваться, его обязательно нужно
                     // будет явно преобразовать к указателю на какой-то известный тип
```

Здесь мы видим целочисленную переменную *k*, уже имеющую конкретное значение (это важно для последующего), и указатель *kPtr*, инициализируемый адресом этой переменной (символ &, называемый амперсанд, обозначает оператор взятия адреса величины, указываемой после него). Стоит ещё раз отметить, что иметь указатель на неинициализированную величину бесполезно, поскольку используется указатель для доступа к тем величинам, на которые "указывает", позволяя "безымянный" доступ – без указания имени (в примере выше мы можем получить значение переменной k, не зная её имени, а зная лишь её адрес).

Напомним, что синтаксис языка Си позволяет добавлять пробельные символы (табуляции, перевода строки и сами пробелы) в программу, если это не нарушает суть программы. Поэтому формально записать объявление указателя можно по-разному: *int* * *kPtr*; или *int* **kPtr*; или даже так: *int***kPtr*; (хотя, конечно, последнее объявление смотрится странно).

Операции с указателями

Точно так же, как и другие типы величин в языке, указатели допускают осуществление над ними некоторых операций: разыменования, инкремента, декремента и, соответственно, прибавления или вычитания целочисленного значения, а также сравнения указателей. Рассмотрим каждую из них подробнее.

Разыменование

Это наиболее важная операция с указателями, поскольку они, как уже говорилось выше, содержат (при правильном использовании) адреса значений, а необходимо иметь доступ и к самим значениям. Операция разыменования, обозначаемая в языке Си тоже символом «звёздочка», позволяет по указателю (с адресом величины) получить значение самой величины, то есть, то значение, на которое он "указывает". Операция разыменования, в отличие от обозначаемой такой же звездочкой бинарной операции умножения, является унарной: принимающей только один операнд – указатель.

Для предшествующего примера с указателем *kPtr* значением выражения **kPtr* будет целочисленная величина типа *int*, находящаяся по адресу, содержащемуся в указателе. А так как инициализирован указатель адресом переменной k, то **kPtr* даёт текущее значение переменной k, т. е., число 5.

Сравнение объявления указателя и операции разыменования обнаруживает наличие некоторой

двойственности в приводимых выше примерах. С одной стороны, в объявлении указано, что переменная с именем *kPtr* имеет тип *int**, с другой стороны, это и напоминание: если мы будем разыменовывать содержащийся в *kPtr* указатель (то есть, получать величину **kPtr*), то мы получим величину типа int.

Инкремент/декремент

Эти операции изменяют адрес, хранимый указателем, в сторону увеличения или уменьшения адреса. А вот каким будет это изменение – зависит от типа указателя (*Type* *), точнее – от размера в памяти той величины, на которую он указывает (этот размер легко определить, он равен *sizeof*(*Type*)).

В любом случае адрес оказывается изменён так, чтобы указывать на следующую или предыдущую величину этого же типа (предполагается, что величины расположены в памяти "вплотную", без каких-либо "пустых" промежутков). Такое поведение указателя по отношению к этим операциям позволяет последовательно переходить от одной величины в памяти к другой в любом направлении (вперёд или назад), давая возможность их перебирать с какой-либо целью (для вывода, изменения, поиска и т.п.).

Прибавление/вычитание целого

Здесь изменение адреса тоже может быть сделано в сторону увеличения или уменьшения, правда, последняя операция (вместе с операцией декремента) должна использоваться с особой осторожностью - чтобыуказатель не получил недопустимое значение адреса (скажем, выходящее за пределы выделенной памяти).

Заметим, что целое значение, модифицирующее значение указателя, приводит к такому изменению адресав байтах, чтобы новое значение указателя можно было использовать для извлечения/изменения величины соответствующего типа в памяти, т.е., добавление целого значения *k* к указателю p типа *Type* * изменит значение адреса в нём на величину *k***sizeof*(*Type*).

С помощью указателей можно "перемещаться" (в цикле) по последовательности расположенных "вплотную" однотипных величин с помощью **p*++ (вперёд) или **p*-- (назад); значение указателя при этом изменяется (тут надо вспомнить таблицу приоритетов: инкремент более приоритетен, чем разыменование). Можно делать почти то же самое, изменяя смещение *k*при нём (сам указатель остаётся неизменным):

*(*p*+*k*) или *(*p*-*k*);

Массивы и указатели

Важно понимать, что любой массив определяется адресом его самого первого элемента. Поэтому имя массива в операциях с указателями приводится к адресу этого элемента:

```
int A[3] = { 1, 2, 3};
int *p = A; // создаем указатель на первый по порядку элемент массива: &A[0]
```

Операции с массивами можно выполнять как через индексацию с помощью квадратных скобок *x*[*i*], так и через разыменование соответствующего указателя *(*x*+*i*) Более того, компилятор не будет считать ошибкой обращение *i*[*x*], хотя оно и выглядит странновато. Это происходит именно потому, что имя массива компилятор заменяет на его адрес. Соответственно, все три варианта сводятся к обращению через указатель *(*x*+*i*).

Сравнение указателей

Как видно из предыдущего раздела, указатель – это тоже целочисленная величина (смещение от виртуального «начала оперативной памяти» компьютера). Следовательно, значения двух указателей можно сравнивать:

```
if ( kPtr == mPtr ) ...  // ложь, так как они в примере выше указывают на
                         // две разные переменные и у этих переменных точно разные
                         // адреса

int *p2 = &A[1];         // создаем указатель на второй по порядку элемент массива
if ( p2 < &A[2] ) ...    // сравнение имеет смысл, так как указатель и второй
                         // элемент операции сравнения указывают внутрь одного
                         // и того же массива, непрерывного в памяти

if ( p2 < kPtr ) ...     // а эти два сравнения – полная бессмыслица, так как
if ( mPtr >= kPtr ) ...  // p2 указывает на элемент массива, а kPtr – на переменную
                         // к, которая неизвестно, где будет расположена в памяти,
                         // так что результат такого сравнения – непредсказуем!
```

Опасности указателей

Не следует забывать и об "опасностях" работы с указателями. В принципе, указателю можно присвоить любое значение адреса, но вот можно ли обратиться к величине по этому адресу? Это большой вопрос... Поэтому программа с применением указателей весьма чувствительна к ошибкам в их значениях (адресах в памяти) и последствия здесь могут быть самые разные: от неправильного функционирования до прекращения работы программы вслучае её обращения в "запретные" места (об этом обычно "заботится" операционная система, запустившая программу). Так что важно всегда верифицировать значения, присваиваемые указателям и контролировать их возможные изменения, удерживая в допустимых пределах. Приведем пример указателя с некорректным адресом:

```
int *p3 = p2 + 8; // указатель p2 указывает на второй элемент небольшого массива,
                  // а указатель p3 мы проинициализировали адресом, который выходит
```

```
                    // за пределы этого массива, если его разыменовать, то программа,
                    // скорее всего, завершится с ошибкой, либо будет получен
                    // некорректный (к тому же случайный) результат ее работы!
```

Нежелательно создавать указатели, которые «никуда не указывают»:

```
int *p4; // указатель p4 не инициализирован, значение адреса в нем будет случайно,
         // а попытка его разыменовать приведет к плохим последствиям, как и в
         // предыдущем примере
```

Вместо этого указатель можно инициализировать специальным «нулевым» адресом:

```
int *p4 = NULL; // такой указатель при попытке разыменования гарантированно
                // приведет к аварийному завершению программы и ошибку найти
                // будет гораздо легче!
```

Функции работы с динамической памятью

Обычная память под переменные программы выделяется компилятором при создании исполняемого файла программы и эта память либо является частью "тела" самой программы (глобальная и статическая память); либо это память под локальные переменные функции, которые располагаются в так называемом стеке программы.

Помимо такой памяти для размещения переменных и массивов величин разного типа, можно также запрашивать память у операционной системы с помощью специальных библиотечных функций. Эта память называется динамической; программа заранее не знает, где будет располагаться эта память и сколько её понадобится, поэтому ничего кроме указателя на начало выделенного блока памяти она получить не может.

Рассмотрим прототипы основных функций для работы с динамической памятью:

```
void* malloc(size_t size);
```

Функция выделяет блок динамической памяти размером *size* (в байтах) и возвращает указатель на его начало (самый первый байт). Если память выделить не удалось (памяти не хватило), возвращается специальное значение указателя: *NULL* (так называемый "невозможный" указатель) и нужно обязательно проверять – что нам вернули?

```
void* realloc(void* ptr, size_t size);
```

Функция изменяет размер ранее выделенного блока динамической памяти с адресом начала *ptr* до размера *size* (в байтах) и возвращает указатель на его начало (самый первый байт). Если память выделить не удалось, возвращается *NULL*. Пользовательское содержимое блока в пределах размера *size* в любом случае остаётся неизменным. Если память выделена, возвращается указатель на её начало, возможно, с другим значением адреса (если выделение произведено в другом месте памяти).

```
void free(void* ptr);
```

Функция объявляет ранее выделенный блок динамической памяти с адресом начала *ptr* "ненужным"; чтобы ошибочно не воспользоваться далее этим адресом, указателю с таким значением разумно изменить значение на *NULL*.

Упомянутые функции выделения динамической памяти никак не инициализируют её; это означает, что в выделенном блоке байты могут иметь произвольное значения (оставшиеся после предыдущих выделений памяти) и не должны использоваться без размещения там нужных программе значений. Возвращаемые и принимаемые значения типа void* символизируют специальный тип "обобщённого" указателя: передать функции с подобным типом параметра можно указатель любого типа, а возвращаемый указатель необходимо всегда преобразовывать к нужному нам типу:

```
int* p = (int*)malloc(size);
```

Организация контейнеров данных

Безадресная память (как явствует из её названия) не предполагает при сохранении указания адреса, где будет находиться величина. Это простой и хороший вариант: не требуется указывать никаких параметров,а лишь само действие размещения где-то в памяти: *push*(), "поместить". Для извлечения величин тоже хотелось бы поступать аналогично, не указывая ничего дополнительно: *pop*(), "вернуть"; величина будет возвращена функцией. Но вот какая из величин в этой памяти должна быть возвращена? Здесь надо принять некоторое разумное соглашение. Конечно, если в такой безадресной памяти сначала не было ничего, то, делая после "заталкивания" величины операцию извлечения, мы ожидаем получить то, что там было. Тут всё понятно. А если что-то в этой безадресной памяти уже находилось?

Имеет смысл хранить все величины последовательно (в противном случае надо было бы ещё запоминать порядок их поступления: после извлечения одной надо понимать, какая будет следующей). Нетрудно сообразить, что в этой нашей последовательной памяти "выделенными" будут лишь две величины: первая и последняя (из остальных непонятно, что можно выбрать, не указывая больше ничего дополнительно). Таким образом, можно принять, например, такое соглашение: возвращается при извлечении последняя добавленная величина.

Стек

Нас можно поздравить: мы "придумали" безадресную память, называемую стек. Можно запоминать величины, можно их извлекать (ничего "лишнего" указывать не нужно), но извлекается всегда величина, поступившая последней (такова цена за отсутствие дополнительных указаний). Аналогией подобному взаимодействию памяти и значений может служить, например, детская пирамидка с кружками на штыре: мы добавляем туда кружки по одному, складывая их друг на друга, но извлечь оттуда один кружок можно, только если он будет верхним, т.е., последним добавленным. Ещё один пример – стопка книг в узкой коробке (чтобы нельзя было взять книгу из середины стопки, а только сверху).

Но нужна ли такая странная память? Оказывается, да. И весьма часто используется. Например, так осуществляется вызов функций практически в любом языке программирования: перед вызовом функции адрес следующей инструкции помещается в стек, поэтому тогда, когда исполнение функции будет завершено, остаётся только извлечь этот адрес из стека - и выполнение программы можно продолжить.

Именно так работает оператор возврата *return*: из функции как бы предлагается "возвратиться" в место исполнения программного кода, которое следует сразу после вызова. И это очень удобно, поскольку функция может быть вызвана из разных мест программы; возврат автоматически будет туда, откуда она была вызвана. Возможные многочисленные вложенные друг в друга вызовы функций здесь тоже предусмотрены, потому что другие адреса возврата будут далее размещаться в стеке, не нарушая ранее уже занесённые адреса.

Такой вариант взаимодействия с памятью традиционно называется LIFO, это сокращение от выражения Last In First Out, то есть, «последним вошёл, первым вышел»).

Дадим формальное определение стека:

Стек (по-английски «стопка») - это динамическая структура данных (с упорядоченным набором элементов), в которой добавление новых элементов и удаление имеющихся производится с одного конца (он называется вершиной стека).

Таким образом, у стека должны быть как минимум две операции: занесение, функция *push()*, и извлечение, функция *pop*(), но для реальной работы их может понадобиться и больше: удобно, например, знать, что из стека ничего извлечь нельзя (он пуст), иметь возможность проверить, что места для последующего занесения не осталось (стек заполнен); часто полезно проверять величину в вершине стека без её извлечения.

Организовать стек можно из любого упорядоченного хранилища однотипных элементов, например, из одномерного массива, снабдив его параметром, называемым указателем стека, в виде указателя или просто индекса. Указатель или индекс могут быть связаны как с последним размещённым в стеке элементом, так и, как вариант, с первым "свободным" местом после только что добавленного элемента.

Если места в хранилище для очередного размещения не будет хватать (всё свободное уже было заполнено), память можно перевыделить с помощью функции *realloc()*, чтобы её стало больше, причём с определённым запасом; в зависимости от выбираемого способа увеличения размера памяти различают стратегии удвоения (размер памяти увеличивается вдвое) или изменения на константу (размер памяти увеличивается на некоторую постоянную величину).

И еще нужно решить техническую проблему: если размер выделенной памяти будет изменен, то и адрес, который на эту память указывает, может измениться, его нужно будет сообщить «вызывающей стороне». Для этого удобно сделать типом данных, который будет обозначать текущий стек, тип «указатель на указатель»: мы передаем в функции адрес указателя на стек и, если он внутри функции будет изменен, то вызывающая сторона получит это изменение. Аналогичный прием можно использовать и для возврата значений предельной емкости стека и количества реальных элементов в этом стеке.

В результате программный интерфейс стека может выглядеть так:

- добавить элемент в стек:

```
void push(double **stack, size_t* size, size_t* capacity, double element);
```

- получить элемент из стека:

```
double top(double **stack, size_t size);
```

- удалить элемент из стека:

```
void pop(double **stack, size_t* size);
```

- очистка памяти:

```
void destroy(double **stack);
```

При помещении первого элемента в новый стек можно в качестве адреса стека передавать просто нулевой указатель:

```
double* pStack = NULL;
size_t size = 0, capacity = 0;
push(&pStack, &size, &capacity, 6.02E+23);
```

При этом функция *push()* самостоятельно должна выделить некоторое начальное количество динамической памяти под стек и заполнить все возвращаемые по указателям значения. Аналогично, разумным

будет, если функция *destroy()* будет обнулять указатель на уничтоженный стек. Отметим еще один нюанс предлагаемой реализации: функция удаления элемента из стека не очищает память, таким образом, выделенная динамическая память при любой работе со стеком – только растет, а освобождается она только один раз – при вызове функции *destroy()*.

Очередь

А что, если в упомянутую выше безадресную память заносить величины, а извлекать их потом не "с того же конца", а из начала? Тогда получится безадресная память с организацией FIFO (сокращение выражения First In First Out, «первым вошёл, первым вышел»): это очередь.

Подлежащие обслуживанию люди (события, предметы) формируют очередь (упорядоченную последовательность), добавляясь в её конец, а обслуживаются, начиная с начала очереди. Здесь аналогия с обычной людской очередью более чем очевидна, а потому дополнительного объяснения не требует.

Формальное определение очереди:

Очередь - это динамическая структура данных (с упорядоченным набором элементов), в которой добавление новых элементов производится с одного конца ("хвост"), а извлечение имеющихся - с другого ("голова").

Таким образом, для очереди должны быть (как минимум) определены операции: добавление элемента в конец и извлечение элемента из начала. Удобно иметь способ проверки очереди на "пустоту" (когда извлечь ничего нельзя); желательна также проверка на невозможность добавления: если очередь организуется в ограниченной области памяти (например, статической), это позволит "не переполнить" её, если используется динамическая память - вовремя увеличить её размер.

Реализовать очередь можно с помощью массива, в котором элементы извлекаются из начала массива, сдвигая оставшиеся элементы к началу и добавляются в конец (возможны и другие варианты исполнения).

Программный интерфейс очереди при этом может выглядеть в точности так же, как выглядел интерфейс стека:

- добавить элемент в очередь:

```
void push(double** queue, size_t* size, size_t* capacity, double element);
```

- получить элемент из очереди:

```
double top(double** queue, size_t size);
```

- удалить элемент из очереди:

```
void pop(double** queue, size_t* size);
```

- очистка памяти:

```
void destroy(double** queue);
```

Но результат работы с очередью будет другим, не таким, как у стека. Это будет определяться реализацией функций, приведенных выше. Работа при первом добавлении элемента в очередь, при удалении элемента и при полной очистке очереди аналогична тому, как это было описано в разделе про стек.

Дека

Дека - это обобщение очереди. Англоязычное название deque образовано как полу-аббревиатура от английского Double Ended Queue, очередь с двумя концами. В деке быстро добавлять и извлекать элементы можно с любого конца.

Дека - это динамическая структура данных (с упорядоченным набором элементов), в которой добавление новых и удаление имеющихся элементов может производиться с обоих её концов.

Таким образом, для деки должны быть определены операции: добавление элемента в начало, добавление элемента в конец, извлечение элемента из начала, извлечение элемента из конца. При реализации нужно не забыть сделать проверку деки на "пустоту" (когда извлечь ничего нельзя ни с одной стороны); желательна также проверка на невозможность добавления элементов с каждой из сторон: если дека организуется в ограниченной области памяти, это позволит "не переполнить" её, если же используется динамическая память, то можно вовремя увеличить размер памяти под деку.

Реализовать деку можно с помощью массива, в котором элементы начинают размещаться "в центре" и добавляются по обе стороны от этого центра (возможны и другие варианты исполнения).

- добавить элемент в начало деки:

```
void push_front(double** deque, size_t* begin, size_t* end,
                size_t* capacity, double element);
```

- добавить элемент в конец деки:

```
void push_back(double** deque, size_t* begin, size_t* end,
                size_t* capacity, double element);
```

- получить элемент из начала деки:

```
double get_front(double** deque, size_t begin);
```

- получить элемент с конца деки:

```
double get_back(double** deque, size_t end);
```

- удалить элемент из начала деки:

```
void pop_front(double** deque, size_t* begin, size_t end);
```

- удалить элемент с конца деки:

```
void pop_back(double** deque, size_t begin, size_t* end);
```

- очистка памяти:

```
void destroy(double** deque);
```

Обратите внимание: при удалении элементов с разных концов деки по-разному передаются параметры начала и конца деки: из них только один из параметров будет меняться, поэтому только один из них передается по указателю, а второй – по значению.

Работа при первом добавлении элемента в деку, при удалении элемента и при полной очистке деки аналогична тому, как это было описано в разделе про стек.

Обратите внимание, что из деки можно сделать и обычную очередь и стек: можно просто не использовать часть возможных для деки операций.

Вектор (динамический массив)

Вектор — это динамический массив, который сам следит за своим размером. Все детали реализации функций описаны выше, например, в разделе про стек, тут приведем только программный интерфейс вектора:

- вставка элемента в позицию **index**:

```
void insert(double** array, size_t* size, size_t* capacity,
            size_t index, double element);
```

- получение элемента на позиции **index**:

```
double at(double** array, size_t index);
```

- изменение размера вектора:

```
void resize(double** array, size_t* size, size_t* capacity, size_t new_size);
// подчеркнем, что тут можно и увеличивать размер вектора, инициализируя новые
// элементы, например, нулями, и уменьшать его: при этом обязательно должны
// сохраниться значения остающихся элементов!
```

- очистка памяти:

```
void destroy(double** array);
```

Двумерные динамические массивы

В программах на языке Си элементы статического двумерного массива (матрицы) могут храниться в памяти построчно (без промежутков), строка за строкой: сначала элементы первой строки, потом - элементы следующей строки и так далее. Это позволяет вычислить, где начинается любая строка: первая (с индексом 0) располагается в самом начале, вторая начинается через W элементов (где W - размер строки матрицы в элементах), третья - через 2*W (после двух строк) и т.д. Поэтому адрес начала строки матрицы с индексом i при таком её расположении даётся выражением p+i*W, где p – адрес начала матрицы, а адрес элемента с индексами i, j - выражением p+i*W+j. Сам элемент может быть получен после операции разыменования: *(p+i*W+j). Напомним, что это выражение полностью эквивалентно такому: p[i*W+j].

При вычислении выражения типа A[i][j] первая операция индексации должна давать указатель на начало *i*-й строки матрицы. Для этого достаточно выделить дополнительно вспомогательный (одномерный) массив указателей с числом элементов, равным количеству строк в матрице.

Где при этом будут располагаться сами строки? Возможны как минимум два варианта: либо каждая строка – это отдельно выделяемый фрагмент динамической памяти (каждый – со своим адресом начала строки), либо можно выделить один фрагмент на всё содержимое матрицы, а массив указателей заполнить адресами начала каждой строки (либо вообще не использовать массив указателей на начала строк матрицы, тогда придется сделать один метод «дай i,j элемент»).

Используем первый способ и выделим память под указатели на адреса начала каждой из строк матрицы:

```
double** a=(double**)malloc(n*sizeof(double*));
```

После этого нужно выделим память для каждой строки матрицы:

```
for(int i=0; i<n; i++)
      a[i]=(double*)malloc(m*sizeof(double));
```

При удалении такого массива придется сначала удалить строки:

```
for(int i=0; i<n; i++)
```

```
	free(a[i]);
```
А потом удалить указатели на них:
```
free(a);
```
Оформим эти действия в виде функций
```
double** create_matrix(int n, int m) // создание массива n*m
{
	double** a=(double**)malloc(n*sizeof(double*));
	for(int i=0; i<n; i++)
		a[i]=(double*)malloc(m*sizeof(double));
	return a; // возвращаем созданный массив
}
void delete_matrix(double** a, int n)// освобождение массива из n строк
{
	for(int i=0;i<n;i++)
		free(a[i]);
	free(a);
}
```
Добавим теперь функции ввода-вывода
```
void print_matrix(double** a, int n, int m)// вывод матрицы на экран
{
	for(int i=0;i<n;i++)
	{
		for(int j=0;j<m;j++)
		{
			printf("%lf\t",a[i][j]);
		}
		printf("\n");
	}
	printf("\n");
}
void input_matrix(double** a, int n, int m)// ввод матрицы с клавиатуры
{
	for(int i=0;i<n;i++)
	{
		for(int j=0;j<m;j++)
		{
			printf("[%d][%d]:",i,j);
			scanf("%lf",&a[i][j]);
		}
	}
	getchar(); // убрать оставшийся от ввода символ '\n'
}
```
При этом получение i,j-элемента матрицы можно реализовать обычным для языка Си образом:
```
double x = a[i][j];
```
Но можно и сделать отдельную функцию для этого:
```
double matrix_element(double** a, int i, int j) // вернуть i,j элемент матрицы
{
	return a[i][j];
}
```
Обработку данных тоже целесообразно оформлять в виде функций. Рассмотрим пример обработки суммирующий значения столбцов матрицы и заполняющий заранее выделенный под эти суммы массив:
```
// функция суммирующая столбцы матрицы a[n][m] в массив res[m]
void sum(double** a, int n, int m, double* res)
{
	for(int j=0;j<m;j++)
	{
		res[j]=0;
		for(int i=0;i<n;i++)
		{
			res[j]+=a[i][j];
		}
	}
}
```
Таким образом, для решения задач по обработке матриц получается полезный набор функций (его можно расширить), который целесообразно записать в отдельный файл и подключить его к проекту.

Для вызова этих функций необходимы объявления (прототипы) этих функций, которые можно поместить в заголовочный файл (например array.h)

```
double** create_matrix(int n, int m);              // создание матрицы n*m
double matrix_element(double** a, int i, int j); // вернуть i,j элемент матрицы
void delete_matrix(double** a, int n);             // освобождение матрицы из n строк
void print_matrix(double** a, int n, int m);       // вывод матрицы на экран
void input_matrix(double** a, int n, int m);       // ввод матрицы с клавиатуры
void sum(double** a, int n, int m, double* res); // суммирование столбцов матрицы
                                                   // в массив res[m]
```

После этого решение задания на матрицы практически сведётся к последовательному вызову функций:

```
#include <stdio.h>
#include <stdlib.h>
#include "array.h"

int main()
{
	int n=3, m=4; // задали размер массива
	double** a=create_matrix(n, m);// создали массив
	input_matrix(a, n, m); // ввели
	print_matrix(a, n, m); // вывели
	double* s=(double*)malloc(m*sizeof(double)); // создали массив результата
	sum(a,n,m,s); // обработали массив и получили результат
	for(int i=0; i<m; i++)
		printf("%lf\t", s[i]); // вывели результат
	delete_matrix(a, n); // освободили исходный массив
	free(s);   // освободили массив результата
	getchar(); // задержка закрытия окна
	return 0;
}
```

Варианты задач для решения

В типовой задаче необходимо написать реализацию требуемого контейнера и тестовую функцию *main*() для нее, которая будет выполнять описанные в задаче действия и тестировать все реализованные для контейнера функции. Если стратегия увеличения ёмкости контейнера не указана – реализовать стратегию удвоения памяти при её нехватке. Если не указан тип элемента контейнера – реализовать для типа *double*. Контейнер заполнять вводимыми с консоли элементами.

Реализацию функций контейнера нужно поместить в отдельный файл (например, *array.cpp*), функцию тестирования поместить в другой файл (например, *task6.cpp*). Объявления функций поместить в заголовочный файл (например, *array.h*).

1. Вариант Реализовать отсортированный вектор для элементов типа **double** со стратегией удвоения ёмкости, который при вставке новых элементов сохраняет свою сортировку.
2. Вариант Реализовать сортировку вектора вещественных чисел без его изменения: создается дополнительный динамический массив указателей на элементы исходного массива. Дальше производится сортировка этого массива указателей, без изменения исходного вектора чисел.
3. Вариант Есть два динамических массива: массив «ключей» и массив «значений». Также есть третий динамический массив, в котором попарно (один за другим) хранятся указатели на соответствующие элементы первых двух массивов. Необходимо реализовать функцию сортировки пар в третьем массиве по возрастанию ключей, а также функцию добавления новой пары ключ-значение ко всей коллекции (с изменением всех трех массивов).
4. Вариант Есть два динамических массива: массив «ключей» и массив «значений». Также есть третий массив, в котором попарно (один за другим) хранятся указатели на соответствующие элементы первых двух массивов. Необходимо реализовать функцию удаления пар ключ-значение с дубликатами ключей: для двух пар с одинаковыми ключами, оставляется та пара, у которой значение больше.

5. Вариант

Реализовать стек значений со стратегией удвоения ёмкости памяти при её нехватке.

6. Вариант

Реализовать стек значений со стратегией увеличения ёмкости памяти на константу при её нехватке. (константа задается через #define)

7. Вариант

Реализовать очередь значений, используя в качестве указателя начала очереди индекс первого элемента в очереди, а в качестве указателя конца очереди индекс первого свободного элемента. Использовать стратегию удвоения ёмкости памяти при её нехватке.

8. Вариант

Реализовать очередь значений, используя в качестве указателя начала очереди идекс первого элемента в очереди, а в качестве указателя конца очереди индекс последнего размещенного элемента. Использовать стратегию увеличения ёмкости памяти на константу при её нехватке. (константа задается через #define)

9. Вариант

Реализовать деку значений, используя в качестве указателя начала деки индекс первого элемента, а в качестве указателя конца деки индекс последнего размещенного элемента.

10. Вариант

Реализовать деку значений, используя в качестве указателя начала деки индекс первого элемента, а в качестве указателя конца деки индекс первого свободного элемента.

11. Вариант

Реализовать двумерную матрицу, для которой реализованы функции взятия элемента по двум индексам (номер строки и номер столбца) и возможностью вставить как новый столбец, так и новую строку. Матрицу реализовать как массив указателей на массивы, хранящие ее строки.

12. Вариант

Реализовать двумерную матрицу, для которой реализованы функции взятия элемента по двум индексам (номер строки и номер столбца) и возможностью вставить как новый столбец, так и новую строку. Матрицу реализовать посредством одного динамического массива.

13. Вариант

Реализовать двумерную матрицу, для которой реализованы функции взятия элемента по двум индексам (номер строки и номер столбца) и возможностью удалить как указанный столбец, так и указанную строку. Матрицу реализовать как массив указателей на массивы, хранящие ее строки.

14. Вариант

Реализовать двумерную матрицу, для которой реализованы функции взятия элемента по двум индексам (номер строки и номер столбца) и возможностью удалить как указанный столбец, так и указанную строку. Матрицу реализовать посредством одного динамического массива.

15. Вариант

Реализовать функцию, которая перемножает две матрицы и возвращает результат в создаваемой третьей матрице. Матрицы реализовать как массивы указателей на массивы, хранящие их строки.

16. Вариант

Реализовать четыре функции, представляющие собой четыре арифметических действия с другими неизвестными функциями, представленными указателями, а также функцию, реализующую суперпозицию двух неизвестных функций, представленных указателями. В качестве элементарных функций использовать: **sin()**, **cos()**, **log()**, **exp()**, **sqrt(), pow()**. По запросу пользователя – построить сложную функцию, реализующую запрошенное функциональное выражение и посчитать значение получившейся функции в заданной точке.

17. Вариант

Дан целочисленный вектор. Реализовать функцию, которая строит и возвращает в новом векторе гистограмму уникальных значений из него. Количество ячеек гистограммы задается пользователем. Распечатать получившуюся гистограмму в «графическом» виде построчно: в каждой строке терминала вывести необходимое для очередной ячейки гистограммы количество символов «*».

18. Вариант

Реализовать очередь на основе двух динамических массивов. Реализовать функции добавления в конец, добавления в начало и взятия элемента по индексу из этой пары массивов. Будем считать, что первый из двух массивов хранит начало очереди в обратном порядке – то есть, последний элемент массива является головным элементом очереди, предпоследний – вторым и т.д. Элементом очереди, следующим за нулевым элементом первого массива является нулевой элемент второго массива, следом – первый элемент второго массива и т.д. до последнего элемента, второго массива, который является последним элементом очереди. Каждый из массивов содержит заранее выделенный запас элементов, который позволит быстро добавлять и удалять элементы в начало и в конец очереди. Нужно учесть, что когда один из двух массивов опустеет – очередь нужно будет «сбалансировать»: переместить в него половину элементов из второго массива.

19. Вариант

Работа с разреженными матрицами. Дана двумерная вещественная матрица, многие элементы которой являются нулями. Написать функцию для преобразования такой матрицы в упакованный набор: массив значений, массив указателей, переводящий двумерные индексы к позиции в массиве значений, при этом элементы матрицы, по модулю меньшие **10-10**, не сохраняются в упакованный набор. Также написать функцию, которая возвращает **[i][j]**-элемент матрицы и функцию печати такой матрицы в обычном двумерном виде. При заполнении матрицы воспользоваться генератором случайных чисел – сделать так, чтобы примерно каждый третий элемент матрицы был равен нулю.

20. Вариант

Реализовать работу с квадратными верхними треугольными матрицами: создание, удаление, умножение, взятие **[i][j]**-элемента матрицы. Нулевые элементы из нижней треугольной области не хранить.

21. Вариант

Реализовать работу с ленточными квадратными матрицами: создание, удаление, умножение, взятие **[i][j]**-элемента матрицы. Нулевые элементы из верхней и нижней треугольных областей не хранить. Ширина «ленты» ненулевых значений около главной диагонали задается пользователем.

22. Вариант

Реализовать сортированную очередь из строк символов (каждая строка – это динамический массив с элементами типа char). Пользователь вводит текстовые строки, которые добавляются в массив указателей, отсортированный по алфавиту. Сначала пользователь добавляет пять строк подряд, затем после каждой добавляемой строки печатается и удаляется одна строка из головы очереди. Программа завершается после ввода строки «**end**».

23. Вариант

Сортировка слиянием. Дан массив, длина которого равна степени числа 2. Разделить массив пополам на два новых массива. Слить эти два массива в исходный массив, соблюдая порядок сортировки в парах «элемент из первого массива»-«элемент из второго массива». Получившийся массив опять разделить пополам и слить два массива в исходный, соблюдая порядок сортировки для четверок элементов. Далее повторить операции для восьмерок элементов и т.д., пока весь массив не окажется отсортированным. В конце – удалить служебные массивы.

24. Вариант

Реализовать двумерную матрицу произвольной размерности, хранящую отдельные биты (так называемый bitmap) в виде битового вектора. Биты хранить упакованными по восемь бит – в байтах. Реализовать функцию, которая возвращает **[i][j]**-бит из матрицы в виде булевского значения. Реализовать функции вставки новых строки и колонки бит.

25. Вариант

Найти максимальный элемент каждой строки матрицы и записать их в стек. Затем извлечь все значения из стека и распечатать их.

26. Вариант

Написать программу, которая удаляет все близкие к указанному пользователем значению элементы вещественного динамического массива. Допустимая степень близости значений задается при помощи #define.

27. Вариант

Построить вектор из порядковых номеров максимальных по модулю элементов в столбцах матрицы.

28. Вариант Для данной матрицы построить два динамических массива: со средним значением каждого столбца и со стандартным уклонением величин в нём.
29. Вариант Построить вектор, в котором попарно (один за другим) будут сохранены индексы i и j элементов матрицы, максимальных по модулю в каждом её столбце.
30. Вариант Создать матрицу из копий полных строк целочисленной исходной матрицы, которые содержат либо минимальный, либо максимальный элемент во всей исходной матрицы. Учтите, что и минимальный и максимальный элементы могут повторяться в разных строках. Обе матрицы реализуйте в форме единых динамических массивов их элементов.