**С/С++ Лабораторная работа 6 - Динамические массивы**

**Введение**

**Динамическое распределение памяти**

Динамическое распределение памяти – способо выделение пяти для программы во время ее выполнения. В отличие, от статическое памяти, которая известная на этапе компиляции программы и выделяется на стеке, динамическая памяти выделяет во время выполнения программы вручную в коде, как и в случае C/C++, однако в других языка, к примеру Python, мы явно не в коде. прописываем выделение динамической памяти, но это сокрыто от программиста для удобства.

**Динамически распределяемая память (куча)**

Куча (heap) – область памяти, в которой хранится данные, которые выделяются динамически. В языке C данные выделяются через функцию *malloc* и ее производную *realloc*, *calloc,* освобождение происходит через функцию *free.* В C++ для выделения памяти есть операторы *new*, а для освобождения *delete, delete []*

В этом примере нельзя точно сказать потребуется ли нам массив 10 значений int32\_t. Это зависит от параметра *flag*. Однако, в этом примере, в случае, если мы выделяем память, то потратим всего 40 байт. Возможно, в данном примере было бы целесообразно использовать память на стеке, так как, когда мы просим операционную систему выделить программе память в куче, то это занимает слишком много(относительно) времени и ресурсов.

|  |
| --- |
| **void** **func**(**bool** flag) {  **if** (flag) {  **uint32\_t**\* ptr = **new** **uint32\_t**(**10**);  **delete** ptr;  }  } |

Стек (stack) – область памяти, которая хранит данные, которые известны на этапе компиляции. Стек выделяет и освобождает память автоматически.

В примере происходит выделение переменной x равно 4 байтов на стеке. При вызове функции этот участок в 4 байта ниже по стоке памяти всегда будет отвечать за переменную x, т.к это уже известно на этапе компиляции

|  |
| --- |
| **void** **func**() {  **uint32\_t** x = **10**;  } |

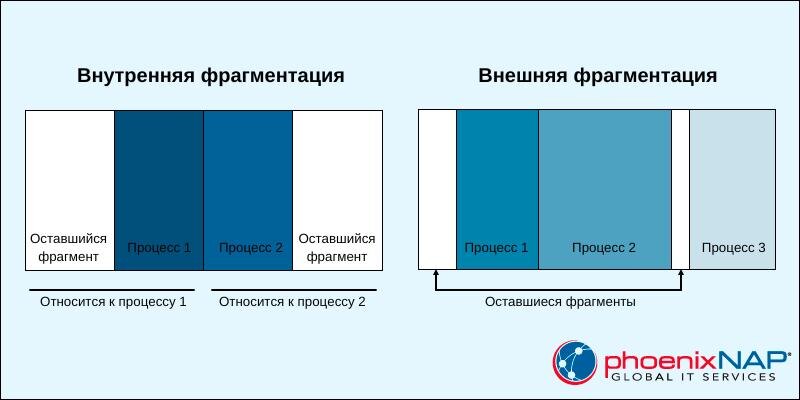
Данные на стеке или на куче могут быть любыми. Нам не важно, что это за типа, запрашивая динамическую память или используя стек, мы используем **выделенной пространство памяти(байтов).** Как им уже пользоваться решать нам.

Однако, плюсов у динамической памяти не много, а минусы существенные. Как и мы было сказано ранее, процессе запроса памяти у ОС является, относительно дорогим, но есть и следующие проблемы. На выделенную память у нас есть указатель – получение доступ через указатель также является дорогостоящей операцией, следовательно любое взаимодействия с данными (запись, чтение) будем дороже, чем на стеке. Также существует такое явление как фрагментация памяти.

Фрагментация памяти — это явление, возникающее при динамическом управлении памятью в операционных системах, когда свободная память распределяется неэффективно, что приводит к потере её части или затруднению выделения новых блоков. Фрагментация приводит к тому, что даже при наличии свободной памяти программа может не получить её в нужном объеме. Существует два основных типа фрагментации памяти:

Внешняя фрагментация возникает, когда в памяти имеется достаточно свободного места, но оно разбито на множество маленьких несмежных блоков, которые не могут быть использованы для удовлетворения запроса на выделение памяти большого размера.

Внутренняя фрагментация возникает, когда выделенный блок памяти больше, чем требуется для хранения данных, и часть этой памяти остаётся неиспользованной. Это происходит из-за того, что память выделяется фиксированными блоками, размер которых не всегда совпадает с размером запрашиваемых данных.



Оператор sizeof используется в C и C++ для определения количества байтов, занимаемых в памяти переменной, типа данных или объекта. Она является ключевой при работе с памятью, так как помогает точно выделять память для динамических структур, а также при создании массивов и структур.

Массив переменной длины — массив, размер которого может быть определен во время выполнения программы. В отличие от динамических массивов, массив переменной длины выделяется на стеке (а не в куче) и не требует явного освобождения памяти. Ниже представлен пример такого массива.

|  |
| --- |
| **void** **func**(**int** n) {  **int** arr[n];  **for** (**int** i = **0**; i < n; i++) {  arr[i] = i + **1**;  }  } |

Функция *calloc* используется для динамического выделения памяти под массив и автоматической инициализации всех элементов массива нулями. Это основное отличие от malloc, которая не инициализирует выделенную память.

|  |
| --- |
| **void**\* **calloc**(**size\_t** num, **size\_t** size); |

Функция *realloc* используется для изменения размера уже выделенного блока памяти. Она может расширить или уменьшить выделенный блок. При увеличении нового размера содержимое памяти сохраняется (если это возможно), а при уменьшении лишняя память освобождается.

|  |
| --- |
| **void**\* **realloc**(**void**\* ptr, **size\_t** new\_size); |

Одна из самых распространённых ошибок при использовании функции *realloc* — это потеря исходного указателя при неудачном выделении памяти.

Если *realloc* не сможет выделить новый блок памяти, она вернёт NULL, а исходный указатель на старый блок будет утерян. Это приведёт к утечке памяти, так как программа больше не сможет освободить ранее выделенный блок. Ниже в примере указан безопасный способ использования функции *realloc.*

|  |
| --- |
| **int**\* temp = realloc(arr, new\_size);  **if** (temp != NULL) {  arr = temp;  } **else** {  **return** E\_ALLOC\_MEMORY;  } |

**Передача аргументов**

**Передача динамического массива в функцию**

Передача динамических массивов в функции осуществляется с использованием указателей, так как массивы не могут быть переданы по значению.

|  |
| --- |
| **void** **printArray**(**int**\* arr, **int** size) {  **for** (**int** i = **0**; i < size; i++) {  printf("%d ", arr[i]);  }  printf("**\n**");  }  **int** **main**() {  **int** n = **5**;  **int**\* array;  array = (**int**\*)malloc(n \* **sizeof**(**int**));  **if** (array == NULL)  **return** E\_ALLOC\_MEMORY;  }  **for** (**int** i = **0**; i < n; i++) {  array[i] = i + **1**;  }  printArray(array, n);  free(array);  **return** **0**;  } |

**Передача содержимого массива по адресу**

Если необходимо передать содержимое динамического массива и иметь возможность изменять его в функции, это делается путем передачи указателя на массив.

|  |
| --- |
| **void** **modifyArray**(**int**\* arr, **int** size) {  **for** (**int** i = **0**; i < size; i++) {  arr[i] \*= **2**;  }  }  **int** **main**() {  **int** n = **5**;  **int**\* array;  array = (**int**\*)malloc(n \* **sizeof**(**int**));  **if** (array == NULL) {  **return** E\_ALLOC\_MEMORY;  }  **for** (**int** i = **0**; i < n; i++) {  array[i] = i + **1**;  }  modifyArray(array, n);  **for** (**int** i = **0**; i < n; i++) {  printf("%d ", array[i]);  }  printf("**\n**");  free(array);  **return** **0**;  } |

При передаче динамических массивов могут возникнуть ошибки, если:

Передача некорректного указателя. Если указатель на массив не инициализирован или указывает на освобожденную память, это приведет к неопределенному поведению программы.

Забыли передать размер массива. Поскольку массивы в C передаются как указатели, информация о размере массива теряется. Если функция не получает информацию о размере массива, это может привести к ошибкам при работе с массивом.

Изменение указателя на массив внутри функции. Если функция изменяет указатель, переданный в неё, это не повлияет на указатель в вызывающей функции (если указатель не передается по адресу).

**Передача указателя на массив по адресу**

Если нужно изменить указатель на массив внутри функции (например, при перераспределении памяти), необходимо передавать указатель по адресу, то есть использовать указатель на указатель.

|  |
| --- |
| **int** **resizeArray**(**int**\*\* arr, **int** new\_size) {  \*arr = (**int**\*)realloc(\*arr, new\_size \* **sizeof**(**int**));  **if** (\*arr == NULL) {  **return** E\_ALLOC\_MEMORY;  }  }  **int** **main**() {  **int** n = **5**;  **int**\* array = (**int**\*)malloc(n \* **sizeof**(**int**));  **if** (array == NULL) {  **return** E\_ALLOC\_MEMORY;  }  resizeArray(&array, **10**);  **for** (**int** i = **5**; i < **10**; i++) {  array[i] = i + **1**;  }  **for** (**int** i = **0**; i < **10**; i++) {  printf("%d ", array[i]);  }  printf("**\n**");  free(array);  **return** **0**;  } |

Задания

1. (C/C++) Реализация строки (динамический массив). Также дописать функции для чтения с потока ввода и запись в этот массив и вывод в поток вывода.

2. (С/С++) Реализация динамического массива структур. Напишите программу, которая создает массив динамических структур, где каждая структура содержит информацию о студенте: имя, возраст и средний балл. Размер массива структур должен быть задан через поток ввода.

3. (C/C++) Имитация динамического массива переменной длины. Напишите программу, которая имитирует динамический массив переменной длины с возможностью добавления элементов в конец массива.

4. (C/C++) Программа для работы с двухмерным динамическим массивом. Напишите программу, которая выделяет память для двумерного динамического массива с использованием указателей на указатели. Программа должна запрашивать у пользователя количество строк и столбцов, затем выделить память для массива и инициализировать его случайными числами

5. (C++) Перегрузить оператор *new* и *delete, delete []* для отладки размера передачи памяти для выделения и освобождения

6. (С++, gdb, ISO, GCC/Clang) Исследуете разницу операторов *delete* и *delete[].* Для более хорошего результат используйте отладчик gdb и покажите в чем разница хранения между динамической переменной и динамическим массивом. Результатом принимается дамп памяти в gdb, где показана разница. Отлично, если показано, где это описывается в стандарте и исходный код из любого компилятора.

Примечание:

(C/C++) – указано, что требуется сделать программу на C и на С++. Хоть и C++ образован от языка С, это **ДРУГОЙ** язык. Пишите программа, используя средства языка, на котором пишется программа. То есть, prinft в языке C++ **НЕЛЬЗЯ**, std::cout, в целом STL, в С **НЕЛЬЗЯ**.

(С++, gdb, ISO, GCC/Clang) – использовать такие инструменты C++, ISO C++. Возможно, стоит использовать gdb, GCC или Clang