*Первому курсу ИиТП посвящается...*

Указатель – страшное, но не менее великое слово. Как бы ни хотелось без него обойтись, сделать это не получится.

Чтобы понять указатель, нужно понять, на что он указывает. А указывает он на память, поэтому пройдёмся по устройству памяти в программе.

Стек и куча

! Память в программе делится на две части: *стек (stack)* и *кучу (heap).*

* Стек – это особая область в оперативной памяти, в которую данные заносятся по умолчанию. Стеком мы пользуемся повсеместно. Любое обычное объявление переменной заносит её в стек. После выхода из программы стек автоматически очищается. Стек довольно быстрый и удобный из-за автоматического управления памятью, но у него есть один существенный недостаток – размер. Размер стека очень мал (в ОС Windows - 1 МБ), что не позволяет загружать в него большие объемы данных. Тут на помощь приходит его товарищ – куча.
* Куча – ещё одна область оперативной памяти. Но есть несколько нюансов.

1. Программа по окончании не очищает кучу. Вы должны прописать очищение ресурсов самостоятельно. Ваш с любовью выделенный массив из 1000 элементов программа с не меньшей любовью оставит в нетронутом виде в качестве занятого куска памяти, что называется *утечкой памяти (memory leak),* и ни один процесс не сможет посягнуть на занятое массивом место. Решается эта проблема перезапуском компьютера или при помощи специальных утилит.

2. Память в куче выделяется и очищается при помощи специальных функций самим программистом.

3. Непосредственно взаимодействовать с памятью, находящейся в куче, мы не можем. Куча гораздо больше стека, и её размеры ограничены только размерами оперативной памяти. Мы заинтересованы в её использовании, но как же нам получить к ней доступ?

4. Непосредственно взаимодействовать с переменными, находящимися в куче мы не можем. Для этого существует специальный тип данных – *указатели (pointers)*.

Вот и они самые. Перейдем же непосредственно к ним.

Указатели

! [type]\* [name] – объявление указателя на переменную типа type.

Положение звездочки значение не имеет. Главное, чтобы она находилась между именем указателя и типом, на который мы указываем. Сами указатели на любой тип данных являются типами данных. Следовательно мы можем сделать указатель на указатель.

[type]\*\* [name]

Чем является указатель? По факту это переменная нашего любимого типа int, которая хранит число, которая является адресом в оперативной памяти. Адреса в оперативной памяти представляют собой числа, обычно записываемые в шестнадцатеричной системе счисления. Следовательно, занося в указатель адрес памяти, мы передаем туда число. Таким образом мы можем преобразовывать указатель к целому числу и наоборот. Инициализировать указатель каким-либо значением не обязательно.

У нас есть выделенная память, есть её адрес. Однако нам не хватает возможности к ней обратиться. А делается это через оператор звездочки \*

\*[name]

Используя оператор \*, мы можем работать с выделенной памятью как с обычной переменной, что будет продемонстрировано на примерах далее. Следует помнить, что действия с указателем не влияют на переменную, расположенную по указываемому адресу.

Для упрощения понимания можно представить кучу как поле, в котором находятся некоторые объекты. Выделяя память, мы помещаем объекты в поле. А указателем будет являться всегда стоящий перед нами столб, который указывает, где в этом поле найти тот или иной объект. И при помощи \* мы приходим к нужному нам объекту и получаем возможность непосредственно с ним работать.

Выделение и очистка памяти в с++

В С++ динамическое выделение памяти довольно просто:

new type

Поздравляю, вы выделили свою первую память!!! Но есть нюанс. Память вы выделили, но запомнить, где она находится, предназначен указатель, т.к. оператор new возвращает адрес выделенной памяти.

[type]\* [name] = new [type];

Выделили память – прекрасно, но её нужно самостоятельно очистить, чтобы избежать утечки.

! Освобождение (очистка) выделенной памяти в С++ выглядит так:

delete name; //где name – имя указателя

После освобождения обращение к этому адресу через оператор \* вызовет ошибку и выбросит соответствующее *исключение (exception).*

Жизненный цикл динамически выделенной переменной:

int\* ptr = new int; // динамическое выделение переменной типа int

\*ptr = 4; // присваивание переменной, на которую указывает указатель, значения 4

int b = \*ptr; //присваивание переменной b значения переменной по адресу ptr

int c = (int)ptr; //запись в переменную с значения адреса, на который указывает указатель ptr

some\_func(c, \*ptr, b); //передача переменной по адресу ptr в аргументы функции

delete ptr; //очистка памяти по адресу ptr

*Примечание.* Дважды очищать динамически выделенную память нельзя, иначе будет выброшено исключение. Так же хорошим тоном считается зануление указателя. Для всего этого существует специальное значение: nullptr, которое обозначает, что указатель в данный момент ни на что не указывает. Вызов оператора delete на указатель со значением nullptr не вызовет исключения. Таким образом к вышеописанному примеру стоит добавить:

ptr = nullptr;

! Динамическое выделение памяти позволяет использовать *динамические массивы (dynamic arrays)* – массивы, чей размер не известен на момент компиляции кода (в отличие от статических *(static),* находящихся в стеке).

Выделение памяти под динамический массив имеет следующий вид:

new {type}[{размер массива}]; // размер массива может быть любой числовой переменной

В данном случае оператор new так же вернет указатель на переменную типа type, однако про глубокие отношения массивов и указателей будет рассказано чуть ниже. Обращение к элементам динамически выделенного массива ничем не отличается от обращения в статическом массиве. Очистка динамически выделенного массива:

delete[] ptr; //ptr – указатель на массив

*Пример:*

unsigned int size;

cin >> size; //ввод числа элементов массива

int\* ptr = new int[size]; //выделение памяти под массив

delete[] ptr; //освобождение памяти, выделенной под массив

Массивы и указатели. Арифметика указателей

Пора раскрыть страшную правду. Переменные массивов на самом деле являются указателями. В записи

int arr[5];

Переменная arr является указателем на переменную типа int. На что же она указывает? На первый элемент массива. НО. Но. Но ведь все элементы статического массива находятся в стеке.

Верно, стек такая же область оперативной памяти, что и куча, следовательно, каждая статическая переменная имеет свой адрес, который можно получить и положить в указатель, и указатель будет корректно работать с памятью, находящейся в стеке. Получить этот адрес можно через оператор &

{type} {name};

{type} \*{name1} = &{name}; //получение адреса только что объявленной переменной

! Взаимодействие с указателем не меняется в зависимости от расположения переменной. Однако указатель, указывающий на переменную из стека, НЕЛЬЗЯ ОЧИЩАТЬ.

В названии раздела написано словосочетание "арифметика указателей". Что же оно означает? А то, что с указателями можно проводить арифметические операции сложения и вычитания целого числа. Однако, несмотря на то что по факту указатель является целым числом, значение, которое мы получим после такой операции, не будет арифметически корректным. А все дело в том, что операция

ptr2 = ptr1 + 1;//ptr1, ptr2 – указатели на один тип данных type

запишет в указатель ptr2 адрес следующего в памяти объекта типа type. Таким образом, обращение к элементу массива по индексу равносильно обращению к указателю на начало массива, увеличенного на индекс элемента:

int arr[] = {1,2,3};

int a = \*(arr+0); //== \*arr == arr[0]

int b = \*(arr+1); //==arr[1]

int \*ptr = new int[3];

int c = \*(ptr+2); //==ptr[2]

! Стоит отметить важное отличие от переменной статического массива и указателя: значение самого этого указателя на массив изменить нельзя. Запись arr = /\*smth\*/; не позволит скомпилировать программу.

Указатели на void

Переменные типа void мы создавать не можем, тогда откуда взяться типу void\*? А все дело в том, что такие указатели позволяют указывать на переменные любого типа.

void\* voidptr;

char t = '0';

voidptr = &t;

int a = 9;

voidptr = &a; //приведения указателей к указателю на void не не нужно

int\* ptr = (int\*) voidptr; //из указателя на void приводить необходимо

Выделение памяти в С

Такой перерыв между разборами выделений памяти сделан не случайно, ведь выделение в памяти в С использует все вышеописанные особенности указателей.

! Для выделения памяти в языке С используются 2 функции:

void\* malloc(unsigned int size) и void\* calloc(unsigned int num, unsigned int size).

malloc выделяет в куче size байтов памяти и возвращает указатель на эту память, calloc же выделяет память под num объектов размером size байтов и заполняет эти объекты нулями. Стоит отметить, что функции возвращают указатель на void, так что каждый раз придется делать приведение указателя к нужному типу.

! Освобождение памяти: free(void\* ptr);

Эта функция просто освобождает память расположенную по адресу ptr.

*Пример*

int\* ptr;

ptr = (int\*) malloc(sizeof(int));

\*ptr = 4;

free(ptr);

ptr = (int\*) calloc(1, sizeof(int));

free(ptr);

Также стоит упомянуть функцию языка C:

void\* realloc(void\* ptr, unsigned int size);

Эта функция позволяет изменить размер участка памяти, на который указывает указатель ptr. Новый размер участка станет равен size, а новый адрес будет возвращен в виде void\*.

*Пример*

int\* arr = (int\*)malloc(sizeof(int)\*2);

arr[0] = 0;

arr[1] = 1;

arr = (int\*) realloc(arr, sizeof(int)\*3); //массив расширяется и значения первых двух его элементов сохраняются

arr[3] = 3; //arr == {1,2,3};

arr = (int\*) realloc(arr, sizeof(int)\*2); //массив уменьшается и значения после первых двух элементов теряются

// arr == {1,2};

! ВАЖНО: память, выделенная оператором new, должна быть освобождена оператором delete, а память, выделенная функциями malloc, calloc, realloc, должна быть освобождена функцией free.

Двумерные массивы

Как же организовать такую полезную вещь, как двумерный массив *(two-dimensional array)?* Для этого используется механизм указателей на указатели:

int\*\* arr = new int\*[{size1}];

for (int i = 0; i < size1; ++i) arr[i] = new int[{size2}];

// пример обращения к элементу – arr[0][1]

Что здесь происходит?

int\*\* arr = new int\*[{size1}];

Динамически создается массив указателей. Чтобы указывать на массив данных, используем указатель на тип этих данных. А типом данных, на который мы указываем, является указатель на int. Следовательно нам нужен указатель на указатель на int.

for (int i = 0; i < size1; ++i) arr[i] = new int[{size2}];

В цикле проходимся по всем выделенным элементам. Массив arr хранит указатели. В каждый элемент этого массива записывается адрес памяти нового массива. Таким образом мы создали массив указателей на массивы, что эквивалентно массиву массивов.

Расшифровка записи arr[i][k]:

arr[i][k] == \*(arr[i]+k) == \*(\*(arr + i) + k).

Удаление двумерного массива происходит в обратном порядке: сначала удаляются массивы, которые находятся за указателями в основном массиве, а потом очищается основной массив.

*Пример*

int\*\* arr = new int\*[5];

for (int i = 0; i < 5; ++i) arr[i] = new int[5];

\*(arr[1] + 2) = 5; // arr[1][2] = 5

\*\*arr = 0; // \*\*arr == \*(\*arr) == \*(\*(arr+0)+0) == arr[0][0]

\*(arr+3)[4] = 7;// arr[3][4] = 7

for(int i = 0; i < 5; ++i) delete[] arr[i];

delete[] arr;

arr = nullptr;

! Трехмерные массивы (*three-dimensional arrays*) выделяются аналогично с поправкой на количество измерений:

int\*\*\* arr = new int\*\*[size1];

for(int i = 0; i < size1; ++i) arr[i] = new int\*[size2];

for(int i = 0; i < size1; ++i)

for(int k = 0; k < size2; ++k) arr[i][k] = new int[size3];

Обращение к элементу: arr[i][j][k] == \*(\*(\*(arr+i)+j)+k)

Все те же возможности представляет и язык C с учетом замены на соответствующие функции.

int\*\* arr = (int\*\*)malloc(sizeof(int\*) \* size1);

for(int i = 0; i < size1; ++i) arr[i] = (int\*) malloc(sizeof(int) \* size2);

/\*Некоторый код\*/

for(int i = 0; i < size1; ++i) free(arr[i]);

free(arr);

Указатели на функцию

! Любая функция в языках имеет свой адрес, следовательно, существуют и указатели на функцию.

{type /\*тип, который функция возвращает\*/} (\*{name})(/\*типы параметров, которые принимает функция\*/);

Указатели на функцию позволяют вызывать указываемую функцию как любую другую: через скобочки.

{name}(/\*Передача аргументов\*/);

А получить адрес функции можно просто написав ее имя.

*Пример*

int sum(int a, int b) { return a+b;}

int mult(int a, int b) {return a\*b;}

void meow() {cout << "meow";}

/\*Некоторый код\*/

int (\*fun)(int,int) = sum;

fun(2,3); //5

fun = mult;

fun(2,3); //6

void (\*v)() = meow;

meow();

! Указатели на функции с различными типами возвращаемых и принимаемых значений являются такими же типами данных, так что из них можно делать массивы, можно на них указывать, а можно передавать в аргументы функции.

float sum(float a, float b) {return a+b;}

float div(float a, float b) {return a/b;}

void action\_with\_3f(float (\*f)(float, float), float a) { cout << f(a,3);}

/\*код\*/

typedef void (\*)(float (\*)(float, float), float) func; // теперь вместо void (\*)(float (\*)(float, float), float) можно писать func (при таких заменах имя писать не надо

func f = action\_with\_3f;

f(sum, 4); //7

f(div, 6); //2

Указатели и const

Указатели позволяют указывать на константы. И сами указатели могут быть константыми. Разница в расположении ключевого слова const. Если const стоит перед звездочкой, то оно относится к типу и указатель указывает на константное значение. Сам указатель можно изменить (поменять адрес, на который указывает указатель), а вот значение за ним изменить нельзя. Если const стоит после звездочки, то свойства константы касаются самого указателя. Указываемый объект может быть изменен, а вот сам указатель не изменишь. И это можно комбинировать

*Пример*

const double \*ptr = new doouble(14); // == double const\* ptr

\*ptr = 9; //ошибка, константу нельзя изменить

ptr = new double(69); //все ок

\*ptr = 9; //ошибка, константу нельзя изменить

double \* const ptr1 = new double(6);

\*ptr1 = 7; // ok

ptr1 = new double(8); // не ok

double const \* const f = new double(14);

double \*ptr2 = f;// нельзя, т.к. указываем на константу

Еще можно указывать константность для указателей высоких порядков на каждый порядок, но показывать я это не буду.