1. Структуры: установка структурного шаблона, определение структурных переменных, инициализация структуры. Доступ к элементам структуры.

Рассмотрим новый тип данных - структуру. Он не только гибок для представления разнообразных данных, но и позволяет создавать новые (пользовательские) типы данных. Пример использования - создание каталога книг. Каждая книга имеет следующие атрибуты: шифр, название, автора, издательство, год издания, число страниц, тираж, цену. Это несколько массивов. Очень сложно организовать одновременную работу с каталогом, если нужно их упорядочить по названиям, авторам, цене и так далее. Лучше иметь один массив, в котором каждый элемент содержит всю информацию о книге.

**Структура** – это объект, состоящий из последовательностей поименованных элементов. Каждый элемент имеет свой тип. Для определения нового типа данных нужно его описать (задать шаблон структуры):

*struct book {*

*char title [81];*

*char author[30];*

*float value;*

*};*

*book* – это имя нового типа данных.

**Структурный шаблон** является основной схемой, описывающей, как образуется новый тип. *struct* - ключевое слово, имя типа структуры *book* - необязателен, если сразу определить имя переменной, то его можно не вводить.

*struct {*

*char title [81];*

*char author[30];*

*float value;*

*}libry;*

Каждый элемент структуры определяется своим собственным описанием. Это

переменные и массивы стандартных типов данных. Шаблон является схемой без содержания. Он сообщает компилятору, как сделать что-то, но ничего не делает в программе, а вот создание структурной переменной, это и есть смысл слова «структура». Согласно шаблону под эту переменную выделяется память, равная сумме всех элементов (81).

Поля структуры располагаются в памяти в том порядке, в котором они объявлены.

*struct book* играет ту же роль, что и int, float перед именем переменной.

(struct book doyle, panshin;)

Для доступа к элементам структурной переменной используется операция точка. Имя переменной, точка, имя элемента структуры.

*void main(void) {*

*struct book libry; //описание перем-й типа book*

*puts("Введите название книги");*

*gets(libry.title);*

*puts("Введите фамилию автора"); gets(libry.author);*

*puts("Введите цену книги");*

*scanf("%f",&libry.value);*

*printf("%s, %s, %p.2f",libry.title,libry.author,libry.value);*

*}*

Структурную переменную можно инициализировать:

*struct book libry* ={"Руслан и Людмила", "А.С.Пушкин", 1.50};

1. Массивы структур: описание, определение элементов массива структур.

Если переменных типа структура много, то определяется массив структур.

*void main(void){*

*struct book libry[100];*

*int i;*

*for(i=0; i<100; i++){*

*puts("Введите название книги");*

*gets(libry[i].title);*

*puts("Введите автора книги");*

*gets(libry[i], author);*

*puts("Введите цену книги");*

*scanf("%f",&libry[i].value);*

*}*

*}*

Индекс применяется к имени массива структур libry[i].

Если libry[2].title[3] – это 4-й элемент в title в 3-й структуре типа book.

1. Вложенные структуры.

Если одна структура содержится или "вложена" в другую, то говорят, что это вложенные структуры.

*struct names{ char name[20];*

*char fio[20];};*

*struct worker{ struct names people;*

*char job[20];*

*float money;};*

*void main(void){*

*struct worker driver = {{"Иван", "Иванов"},*

*"водитель", 1234.1};*

Для обращения к элементу вложенной структуры применяется две операции

«точка».

*puts(driver .people.name);*

1. Битовые структуры.

В некоторых задачах для экономии памяти необходимо упаковывать несколько объектов в одно машинное слово. В Си для этого определяются поля и доступ к ним. **Поле** – это последовательность битов внутри одного целого значения.

*struct { unsigned a:8;*

*unsigned b:6;*

*unsigned c:2;}d;*

Определяем структуру *d*, содержащую поле *а* – 8 битов, поле *b* – 6 битов, *с* – 2 бита.

Поля описываются как unsigned, чтобы подчеркнуть, что это величины без знака. Отдельные поля теперь обозначаются как d.a, d.b, d.c. С полями можно выполнять различные операции.

*d.a= d.b=( d.c<<2)+6;*

Поля не могут переходить за границу слова в ЭВМ. Если же очередное поле не помещается в частично заполненное слово, то под него выделяется новое слово. Поля могут быть безымянными. Используются как заполнители. Для принудительного перехода на новое слово используется специальный размер 0.

*struct {*

*unsigned a:8;*

*:2;*

*unsigned b:6;*

*:0;*

*unsigned c:12;} d;*

Битовые поля и объединения можно применять для неявного преобразования типов.

Пример 1.

*struct DOS\_DATE { unsigned int day:5;*

*unsigned int month:4;*

*unsigned int year:7;};*

*union DATE\_CONV { unsigned int packed\_date;*

*struct DOS\_DATE unpacked\_date;};*

*typedef union DATE\_CONV DATE*

*void main(void) {*

*struct ffblk ff;* //структура в которую читается информация о

//файле из каталога, описана в <dir.h>

*int done=findfirst(“\*.\*”, &ff,0);* //ищет первый файл в каталоге

*if(!done) {*

*DATE d;*

*d.packed\_date=ff.ff\_date;*

*printf(%2d/%2d/%4d”, d.unpacked\_date.day, d.unpacked\_date.month, d.unpacked\_ date.year+1980);*

*}*

*}*

1. Адресация в языке программирования. Указатели. Операции с указателями.

Для работы с указателями в Си++ определены две операции:

**операция \* (звездочка**) — позволяет получить значение объекта по его адресу — определяет значение переменной, которое содержится по адресу, содержащемуся в указателе;

**операция & (амперсанд)** — позволяет определить адрес переменной.

• **указатель** – это переменная, в которой записан адрес другой переменной или участка памяти;

• при объявлении указателя надо указать тип переменных, на которых он будет указывать, а перед именем поставить знак \*;

• знак & перед именем переменной обозначает ее адрес;

• знак \* перед указателем в рабочей части программы (не в объявлении) обозначает значение ячейки, на которую указывает указатель;

• нельзя записывать по указателю, который указывает непонятно куда – это вызывает сбой программы, поскольку что-то стирается в памяти;

• для обозначения недействительного указателя используется константа NULL;

• при изменении значения указателя на n он в самом деле сдвигается к n-ому следующему числу данного типа, то есть для указателей на целые числа на n\*sizeof(int) байт (запись pI+=4 или pI=pI+4 сдвигает указатель на 4 целых числа дальше, то есть на 16 байт);

• указатель печатаются по формату %p.

1. Массивы и указатели.

Динамическими называются массивы, размер которых неизвестен на этапе написания программы. Прием, о котором будем говорить, относится уже не к стандартному языку Си, а к его расширению Си ++. Существуют и стандартные способы выделения памяти в языке Си (с помощью функций malloc и calloc), но они не очень удобны.

Следующая простейшая программа, которая использует динамический массив, вводит с клавиатуры размер массива, все его элементы, а затем сортирует их и выводит на экран.

*#include <stdio.h> main()*

*{*

*int N;* // размер массива (заранее неизвестен)

*int \*A;* // указатель для выделения памяти

*printf ("Размер массива > "); //* ввод размера массива

*scanf ("%d", &N);*

*A = new int [N];* // выделение памяти

*if ( A == NULL ) {* // если не удалось выделить

*printf("Не удалось выделить память");*

*return 1;* // выход по ошибке, код ошибки 1

*}*

*for (i = 0; i < N; i ++ ) {* // дальше так же, как для массива

*printf ("\nA[%d] > ", i+1);*

*scanf ("%d", &A[i]);*

*}*

// здесь сортировка и вывод на экран

*delete A;* // освободить память

*}*

Итак, мы хотим выделить в памяти место под новый массив целых чисел во время работы программы.

Мы уже знаем его размер, пусть он хранится в переменной N (это число обязательно должно быть больше нуля). Оператор выделения памяти new вернет нам адрес нового выделенного блока, и для работы с массивом нам надо где-то его запомнить. Вы уже знаете, что есть специальный класс переменных для записи в них адресов памяти – указатели.

• динамические массивы используются тогда, когда на момент написания программы размер массива неизвестен

• для того, чтобы работать с динамическим массивом, надо объявить указатель соответствующего типа (в нем будет храниться адрес первого элемента массива);

• int \*A;

• когда требуемый размер массива стал известен, надо использовать оператор new языка Си++, указав в скобках размер массива (в программе для краткости нет проверки на положительность N);

*A = new int[N];*

• нельзя использовать оператор new при отрицательном или нулевом N;

• после выделения памяти надо проверить, успешно ли оно прошло; если память выделить не удалось, то значение указателя будет равно NULL, использование такого массива приведет к сбою программы;

• работа с динамическим массивом, на который указывает указатель А, идет также, как и с обычным массивом размера N (помните, что язык Си не следит за выходом за границы массива);

• после использования массива надо освободить выделенную память, вызвав оператор

*delete A*;

• после освобождения памяти значение указателя не изменяется, но использовать его уже нельзя, потому что память освобождена;

• учитывая, что при добавлении числа к указателю он сдвигается на заданное число ячеек ЗАДАННОГО ТИПА, то следующие записи равносильны и вычисляют адрес i-ого элемента массива:

*&A[i] и A+i*

Отсюда следует, что A совпадает с &A[0], и поэтому имя массива можно использовать как адрес его начального элемента;

Ошибки, связанные с выделением памяти

Самые тяжелые и трудно вылавливаемые ошибки в программах на языке Си связаны именно с неверным использованием динамических массивов. В таблице перечислены наиболее тяжелые случаи и способы борьбы с ними.

|  |  |
| --- | --- |
| **Ошибка** | **Причина и способ лечения** |
| Запись в чужую область памяти | Память была выделена неудачно, а массив используется. Вывод: надо всегда делать проверку указателя на NULL. |
| Повторное удаление  указателя | Массив уже удален и теперь удаляется снова. Вывод: если массив удален из памяти, обнулите указатель – ошибка быстрее выявится. |
| Выход за границы массива | Запись в массив в элемент с отрицательным индексом или индексом, выходящим за границу массива |
| Утечка памяти | Неиспользуемая память не освобождается. Если это происходит в функции, которая вызывается много раз, то ресурсы памяти скоро будут израсходованы.  Вывод: убирайте за собой «мусор» (освобождайте память). |

new позволяет выделять только одномерные массивы, поэтому для работы с многомерными массивами необходимо воспринимать их как массив указателей на другие массивы.

Для примера рассмотрим задачу выделения динамической памяти под массив чисел размера n на m.

**1ый способ**

На первом шаге выделяется указатель на массив указателей, а на втором шаге, в цикле каждому указателю из массива выделяется массив чисел в памяти:

*int \*\* a = new int\*[n];*

*for (int i = 0; i != n; ++i)*

*a[i] = new int[m];*

Однако, этот способ плох тем, что в нём требуется n+1 выделение памяти, а это достаточно дорогая по времени операция.

**2ой способ**

На первом шаге выделение массива указателей и массива чисел размером n на m. На втором шаге каждому указателю из массива ставится в соответствие строка в массиве чисел.

*int \*\* a = new int\*[n];*

*a[0] = new int[n\*m];*

*for (int i = 1; i != n; ++i)*

*a[i] = a[0] + i\*m;*

В данном случае требуется всего 2 выделения памяти. Для освобождения памяти требуется выполнить:

**1ый способ:**

*for (int i = 0; i != n; ++i)*

*delete [] a[i];*

*delete [] a;*

**2ой способ:**

*delete [] a[0];*

*delete [] a;*

Таким образом, второй способ опять же требует гораздо меньше вызовов функции delete [], чем первый.

1. Структуры и указатели.

Указателями на структуры легче пользоваться, чем самими структурами. Структура не может передаваться в качестве аргумента функции, а указатель на структуру может.

*struct worker \*pdrv;*

*pdrv = &driver;*

*struct worker driver[2];* //массив структур

*а) pdrv = driver; // pdrv <=> &driver[0];*

*pdrv+1 <=> &driver[1].*

Доступ к элементу структуры осуществляется через операцию ->.

*pdrv->job -> driver[0].job ->(\*prdv).job*

*б) pdrv->people.name*

Про память

Память, которую использует программа делится на три вида:

1. Статическая память (static memory)
   1. хранит глобальные переменные и константы;
   2. размер определяется при компиляции.
2. Стек (stack)
   1. хранит локальные переменные, аргументы функций и промежуточные значения
   2. вычислений;
   3. b. размер определяется при запуске программы (обычно выделяется 4 Мб).
3. Куча (heap)
   1. динамически распределяемая память;
   2. ОС выделяет память по частям (по мере необходимости)
4. Динамическая память (функции и операции).

**Функции выделения и освобождения памяти.**

1. Функция **void\* malloc(размер)** - выделяет в «куче» n байтов и возвращает адрес на 1-й байт, иначе возвращает 0(NULL), в реальности происходит обращение к операционной системе с просьбой выделить участок памяти необходимого размера и операционная система через менеджер памяти пытается данную просьбу удовлетворить и результатом работы данной функции будет либо адрес начала выделенной памяти, либо 0, что означает отсутствие выделения памяти. Необходимо применять преобразование типов.

*void main(void){*

*char \*original=”Исходная строка”;*

*char \*copy;*

*copy=(char\*)malloc(strlen(original)+1);*

*if(copy==NULL) {*

*puts(“Ошибка выделения памяти”);*

*exit(1);*

*}*

*strcpy(copy,original);*

*cout<<copy<<endl;*

*cout<<original<<endl;*

*free(copy);*

*}*

При выделении памяти она не очищается. Размер указывается в байтах.

1. Функция **void\* calloc(n,size type);**

long\* newmas = (long\*)calloc(100,sizeof(long));

первый параметр - количество требуемых ячеек памяти;

второй параметр - размеp каждой ячейки.

Функция calloc() обнуляет выделенную память.

*#define SIZE 128*

*void main(void) {*

*char \*str=(char \*)calloc(1,SIZE);*

*if(str==NULL) {*

*puts(“Ошибка выделения памяти”);*

*exit(1);*

*}*

*else {*

*cout<<”Введите строку”;*

*gets(str);*

*cout<<str;*

*free(s);*

*}*

*}*

3. **void\* realloc(void \*, n)** - изменяет размер ранее выделенного блока памяти. Если дополнительная память выделяется в новом месте оперативной памяти, то уже введенная информация

переписывается в новое место.

ptr = realloc(nptr, size);

где nptr - указатель на ранее выделенный блок; size размер выделяемой памяти.

*void main(void) {*

*char \*p1, \*p2;*

*puts(«Выделяем 128 байт»);*

*p1=malloc(128);*

*if(p1) {*

*puts(“Изменяем размер блока на 256 байтов”);*

*p2=realloc(p1,256);* //блок теперь равен 256 байтам

*}*

*if(p2)*

*free(p2);*

*else*

*free(p1);*

*}*

4. **void free(void \*ptr)** - освобождение блока памяти, адресуемого указателем ptr. Замечание: необходимо отметить, что при выделении памяти функцией malloc(), освободить можно только весь ранее запрошенный участок памяти, а при выделении памяти функцией calloc(), возможно освобождение памяти начиная с адреса большего чем начальный адрес памяти и до конца

выделенного пространства.

**Операции выделения и освобождения памяти.**

В С++ есть свой механизм выделения и освобождения памяти — это функции new и delete.

Пример использования new:

*int \* p = new int[1000000];* // выделение памяти под 1000000 int`ов

Т.е. при использовании функции new не нужно приводить указатель и не нужно использовать sizeof().

Освобождение выделенной при помощи new памяти осуществляется посредством следующего

вызова:

*delete [] p;*

Если требуется выделить память под один элемент, то можно использовать

*int \* q = new int;*

*или*

*int \* q = new int(10);* // выделенный int проинциализируется значением 10

в этом случае удаление будет выглядеть следующим образом:

*delete q;*

Замечание:

Выделять динамически небольшие кусочки памяти (например, под один элемент простого типа данных) не целесообразно по двум причинам:

1. При динамическом выделении памяти в ней помимо значения указанного типа будет храниться служебная информация ОС и С/С++. Таким образом потребуется гораздо больше памяти, чем при хранении необходимых данных на стеке.

2. Если в памяти хранить большое количество маленьких кусочков, то она будет сильно

фрагментирована и большой массив данных может не поместиться.

1. Списки.

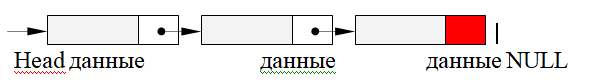
Часто в серьезных программах надо использовать данные, размер и структура которых должны меняться в процессе работы. Динамические массивы здесь не выручают, поскольку заранее нельзя сказать, сколько памяти надо выделить – это выясняется только в процессе работы. Например, надо проанализировать текст и определить, какие слова и в каком количество в нем встречаются, причем эти слова нужно расставить по алфавиту.

В таких случаях применяют данные особой структуры, которые представляют собой отдельные элементы, связанные с помощью ссылок.

Каждый элемент (узел) состоит из двух областей памяти: поля данных и ссылок. Ссылки – это адреса других узлов этого же типа, с которыми данный элемент логически связан. В языке Си для организации ссылок используются переменные указатели. При добавлении нового узла в такую структуру выделяется новый блок памяти и (с помощью ссылок) устанавливаются связи этого элемента с уже существующими. Для обозначения конечного элемента в цепи используются нулевые ссылки (NULL).

Линейный список

В простейшем случае каждый узел содержит всего одну ссылку. Для определенности будем считать, что решается задача частотного анализа текста – определения всех слов, встречающихся в тексте и их количества. В этом случае область данных элемента включает строку (длиной не более 40 символов) и целое число.



Каждый элемент содержит также ссылку на следующий за ним элемент. У последнего в списке элемента поле ссылки содержит NULL. Чтобы не потерять список, мы должны где- то (в переменной) хранить адрес его первого узла – он называется «головой» списка. В программе надо объявить два новых типа данных – узел списка Node и указатель на него PNode. Узел представляет собой структуру, которая содержит три поля - строку, целое число и указатель на такой же узел. Правилами языка Си допускается объявление

*struct Node {*

*char word[40];* // область данных

*int count;*

*Node \*next;* // ссылка на следующий узел

*}; typedef Node \*PNode;* // тип данных: указатель на узел

В дальнейшем мы будем считать, что указатель Head указывает на начало списка, то есть, объявлен в виде

*PNode Head = NULL;*

Первая буква «P» в названии типа PNode происходит от слова pointer – указатель (англ.) В начале работы в списке нет ни одного элемента, поэтому в указатель Head записывается нулевой адрес NULL.

**Создание элемента списка**

Для того, чтобы добавить узел к списку, необходимо создать его, то есть выделить память под узел и запомнить адрес выделенного блока. Будем считать, что надо добавить к списку узел, соответствующий новому слову, которое записано в переменной NewWord.

Составим функцию, которая создает новый узел в памяти и возвращает его адрес. Обратите внимание, что при записи данных в узел используется обращение к полям структуры через указатель.

*PNode CreateNode ( char NewWord[] )*

*{*

*PNode NewNode = new Node;* // указатель на новый узел

*strcpy(NewNode->word, NewWord);* // записать слово

*NewNode->count = 1;* // счетчик слов = 1

*NewNode->next = NULL;* // следующего узла нет

*return NewNode;* // результат функции – адрес узла

*}*

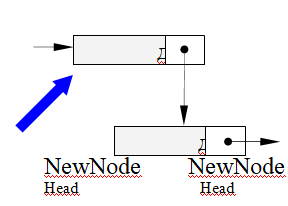
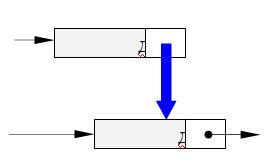
После этого узел надо добавить к списку (в начало, в конец или в середину).

Добавление узла Добавление узла в начало списка

При добавлении нового узла NewNode в начало списка надо

1) установить ссылку узла NewNode на голову существующего списка и

2) установить голову списка на новый узел.

1.  2) 

По такой схеме работает процедура AddFirst. Предполагается, что адрес начала списка хранится в Head. Важно, что здесь и далее адрес начала списка передается по ссылке, так как при добавлении нового узла он изменяется внутри процедуры.

*void AddFirst (PNode &Head, PNode NewNode)*

*{*

*NewNode->next = Head; Head = NewNode;*

*}*

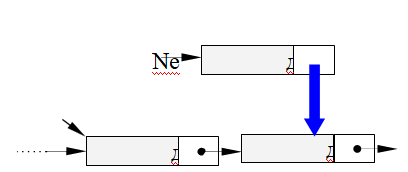
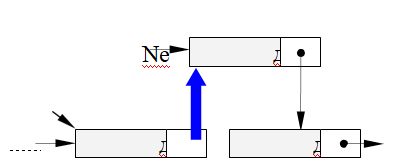
**Добавление узла после заданного**

Дан адрес NewNode нового узла и адрес p одного из существующих узлов в списке.

Требуется вставить в список новый узел после узла с адресом p. Эта операция выполняется в два этапа:

1) установить ссылку нового узла на узел, следующий за данным;

2) установить ссылку данного узла p на NewNode.

1.  2) 

Последовательность операций менять нельзя, потому что если сначала поменять

ссылку у узла p, будет потерян адрес следующего узла.

*void AddAfter (PNode p, PNode NewNode)*

*{*

*NewNode->next = p->next;*

*p->next = NewNode;*

*}*

**Добавление узла перед заданным**

Эта схема добавления самая сложная. Проблема заключается в том, что в простейшем линейном списке (он называется односвязным, потому что связи направлены только в одну сторону) для того, чтобы получить адрес предыдущего узла, нужно пройти весь список сначала. Задача сведется либо к вставке узла в начало списка (если заданный узел – первый), либо к вставке после заданного узла.

*void AddBefore(PNode &Head, PNode p, PNode NewNode)*

*{*

*PNode q = Head; if (Head == p) {*

*AddFirst(Head, NewNode);* // вставка перед первым узлом

*return;*

*}*

*while (q && q->next!=p)* // ищем узел, за которым следует p

*q = q->next; if ( q )* // если нашли такой узел,

*AddAfter(q, NewNode);* // добавить новый после него

*}*

Такая процедура обеспечивает «защиту от дурака»: если задан узел, не присутствующий в списке, то в конце цикла указатель q равен NULL и ничего не происходит.

Существует еще один интересный прием: если надо вставить новый узел NewNode до заданного узла p, вставляют узел после этого узла, а потом выполняется обмен данными между узлами NewNode и p. Таким образом, по адресу p в самом деле будет расположен узел с новыми данными, а по адресу NewNode – с теми данными, которые были в узле p, то есть мы решили задачу. Этот прием не сработает, если адрес нового узла NewNode запоминается где-то в программе и потом используется, поскольку по этому адресу будут находиться другие данные.

**Добавление узла в конец списка**

Для решения задачи надо сначала найти последний узел, у которого ссылка равна NULL, а затем воспользоваться процедурой вставки после заданного узла. Отдельно надо обработать случай, когда список пуст.

*void AddLast(PNode &Head, PNode NewNode)*

*{*

*PNode q = Head;*

*if (Head == NULL) {* // если список пуст,

*AddFirst(Head, NewNode);* // вставляем первый элемент

*return;*

*}*

*while (q->next) q = q->next;* // ищем последний элемент

*AddAfter(q, NewNode);*

*}*

**Проход по списку**

Для того, чтобы пройти весь список и сделать что-либо с каждым его элементом, надо начать с головы и, используя указатель next, продвигаться к следующему узлу.

*PNode p = Head;* // начали с головы списка

*while ( p != NULL ) {* // пока не дошли до конца

// делаем что-нибудь с узлом p

*p = p->next;* // переходим к следующему узлу

*}*

**Поиск узла в списке**

Часто требуется найти в списке нужный элемент (его адрес или данные). Надо учесть, что требуемого элемента может и не быть, тогда просмотр заканчивается при достижении конца списка. Такой подход приводит к следующему алгоритму:

начать с головы списка;

пока текущий элемент существует (указатель – не NULL), проверить нужное условие и перейти к следующему элементу;

закончить, когда найден требуемый элемент или все элементы списка просмотрены.

Например, следующая функция ищет в списке элемент, соответствующий заданному слову (для которого поле word совпадает с заданной строкой NewWord), и возвращает его адрес или NULL, если такого узла нет.

*PNode Find (PNode Head, char NewWord[])*

*{*

*PNode q = Head;*

*while (q && strcmp(q->word, NewWord))*

*q = q->next;*

*return q;*

*}*

Вернемся к задаче построения алфавитно-частотного словаря. Для того, чтобы добавить новое слово в нужное место (в алфавитном порядке), требуется найти адрес узла, перед которым надо вставить новое слово. Это будет первый от начала списка узел, для которого «его» слово окажется «больше», чем новое слово. Поэтому достаточно просто изменить условие в цикле while в функции Find., учитывая, что функция strcmp возвращает

«разность» первого и второго слова.

*PNode FindPlace (PNode Head, char NewWord[])*

*{*

*PNode q = Head;*

*while (q && (strcmp(q->word, NewWord) > 0))*

*q = q->next;*

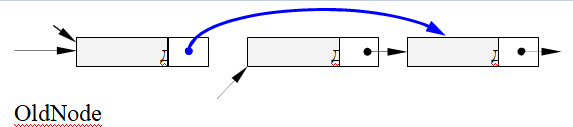
*return q;*

*}*

Эта функция вернет адрес узла, перед которым надо вставить новое слово (когда функция strcmp вернет положительное значение), или NULL, если слово надо добавить в конец списка.

**Удаление узла**

Эта процедура также связана с поиском заданного узла по всему списку, так как нам надо поменять ссылку у предыдущего узла, а перейти к нему непосредственно невозможно. Если мы нашли узел, за которым идет удаляемый узел, надо просто переставить ссылку.



Отдельно обрабатывается случай, когда удаляется первый элемент списка. При удалении узла освобождается память, которую он занимал.

Отдельно рассматриваем случай, когда удаляется первый элемент списка. В этом случае адрес удаляемого узла совпадает с адресом головы списка Head и надо просто записать в Head адрес следующего элемента.

*void DeleteNode(PNode &Head, PNode OldNode)*

*{*

*PNode q = Head;*

*if (Head == OldNode)*

*Head = OldNode->next;* // удаляем первый элемент

*else*

*{*

*while (q && q->next != OldNode)* // ищем элемент

*q = q->next;*

*if ( q == NULL ) return;* // если не нашли, выход

*q->next = OldNode->next;*

*}*

*delete OldNode;* // освобождаем память

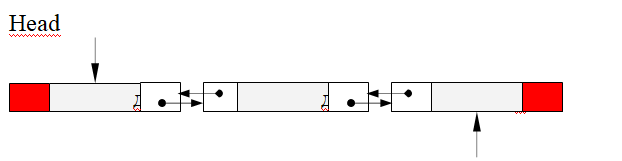
*}*

**Барьеры**

Для рассмотренного варианта списка требуется отдельно обрабатывать граничные случаи: добавление в начало, добавление в конец, удаление одного из крайних элементов. Можно значительно упростить приведенные выше процедуры, если установить два барьера – фиктивные первый и последний элементы. Таким образом, в списке всегда есть хотя бы два элемента-барьера, а все рабочие узлы находятся между ними.

**Двусвязный список**

Многие проблемы при работе с односвязным списком вызваны тем, что в них невозможно перейти к предыдущему элементу. Возникает естественная идея – хранить в памяти ссылку не только на следующий, но и на предыдущий элемент списка. Для доступа к списку используется не одна переменная-указатель, а две – ссылка на «голову» списка (Head) и на «хвост» - последний элемент (Tail).



Каждый узел содержит (кроме полезных данных) также ссылку на следующий за ним узел (поле next) и предыдущий (поле prev). Поле next у последнего элемента и поле prev у первого содержат NULL. Узел объявляется так:

*struct Node {*

*char word[40];* // область данных

*int count;*

*Node \*next, \*prev;* // ссылки на соседние узлы

*};*

*typedef Node \*PNode;* // тип данных «указатель на узел»

В дальнейшем мы будем считать, что указатель Head указывает на начало списка, а указатель Tail – на конец списка:

PNode Head = NULL, Tail = NULL;

Для пустого списка оба указателя равны NULL.

**Операции с двусвязным списком**

**Добавление узла в начало списка**

При добавлении нового узла NewNode в начало списка надо

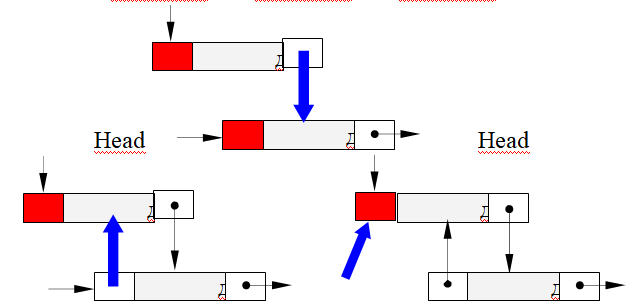
установить ссылку next узла NewNode на голову существующего списка и его ссылку prev в NULL;

установить ссылку prev бывшего первого узла (если он существовал) на NewNode;

3) установить голову списка на новый узел;

4) если в списке не было ни одного элемента, хвост списка также устанавливается на новый узел.

1) NewNode 2) NewNode 3) NewNode



По такой схеме работает следующая процедура:

*void AddFirst(PNode &Head, PNode &Tail, PNode NewNode)*

*{*

*NewNode->next = Head;*

*NewNode->prev = NULL;*

*if ( Head )*

*Head->prev = NewNode;*

*Head = NewNode;*

*if ( ! Tail )*

*Tail = Head;* // этот элемент – первый

*}*

**Добавление узла в конец списка**

Благодаря симметрии добавление нового узла NewNode в конец списка проходит совершенно аналогично, в процедуре надо везде заменить Head на Tail и наоборот, а также поменять prev и next.

**Добавление узла после заданного**

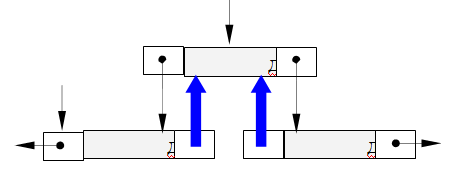
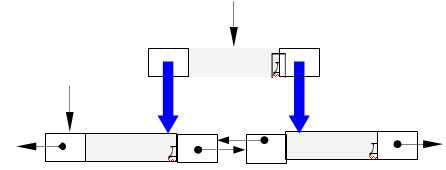
Дан адрес NewNode нового узла и адрес p одного из существующих узлов в списке.

Требуется вставить в список новый узел после p. Если узел p является последним, то операция сводится к добавлению в конец списка (см. выше). Если узел p – не последний, то операция вставки выполняется в два этапа:

установить ссылки нового узла на следующий за данным (next) и предшествующий ему (prev);

установить ссылки соседних узлов так, чтобы включить NewNode в список.

1. NewNode 2) NewNode



Такой метод реализует приведенная ниже процедура (она учитывает также возможность вставки элемента в конец списка, именно для этого в параметрах передаются ссылки на голову и хвост списка):

*void AddAfter (PNode &Head, PNode &Tail, PNode p, PNode NewNode)*

*{*

*if ( ! p->next )*

*AddLast (Head, Tail, NewNode);* // вставка в конец списка

*else {*

*NewNode->next = p->next;* // меняем ссылки нового узла

*NewNode->prev = p;*

*p->next->prev = NewNode;* // меняем ссылки соседних узлов

*p->next = NewNode;*

*}*

*}*

Добавление узла перед заданным выполняется аналогично.

**Поиск узла в списке**

Проход по двусвязному списку может выполняться в двух направлениях – от головы к хвосту (как для односвязного) или от хвоста к голове.

**Удаление узла**

Эта процедура также требует ссылки на голову и хвост списка, поскольку они могут измениться при удалении крайнего элемента списка. На первом этапе устанавливаются ссылки соседних узлов (если они есть) так, как если бы удаляемого узла не было бы. Затем узел удаляется и память, которую он занимает, освобождается. Эти этапы показаны на рисунке внизу. Отдельно проверяется, не является ли удаляемый узел первым или последним узлом списка.

OldNode



*void Delete(PNode &Head, PNode &Tail, PNode OldNode)*

*{*

*if (Head == OldNode) {*

*Head = OldNode->next;* // удаляем первый элемент

*if ( Head )*

*Head->prev = NULL;*

*else*

*Tail = NULL;* // удалили единственный элемент

*}*

*else {*

*OldNode->prev->next = OldNode->next;*

*if ( OldNode->next )*

*OldNode->next->prev = OldNode->prev;*

*else Tail = NULL;* // удалили последний элемент

*}*

*delete OldNode;*

*}*

**Циклические списки**

Иногда список (односвязный или двусвязный) замыкают в кольцо, то есть указатель next последнего элемента указывает на первый элемент, и (для двусвязных списков) указатель prev первого элемента указывает на последний. В таких списках понятие «хвоста» списка не имеет смысла, для работы с ним надо использовать указатель на «голову», причем «головой» можно считать любой элемент.

1. Объединения. Преобразование массивов.

Объединение – это группирование переменных, которые разделяют одну и ту же область памяти. В зависимости от интерпретации осуществляется обращение к той или другой переменной объединения. Все переменные, что включены в объединение начинаются с одной границы. Объединение позволяет представить в компактном виде данные, которые могут изменяться. Одни и те же данные могут быть представлены разными способами с помощью объединений. Точно также как и структуры, объединения требуют объявления типа (шаблона) и объявления переменной этого типа.

**Форма объявления типа (шаблона) объединения?**

**Ключевое слово union**

Объявление объединения (типа объединения или шаблона объединения) начинается с ключевого слова union.

*union имя\_типа\_объединения*

*{*

*тип переменная1;*

*тип переменная2;*

*...*

*тип переменнаяN;*

*};*

где

* имя\_типа\_объединения – непосредственно имя новосозданного типа;
* переменная1, переменная2, переменнаяN – переменные, которые есть полями объединения. Эти переменные могут быть разных типов;
* тип – тип переменной, который есть полем объединения.

Тип переменной может быть:

* базовым типом, принятым в языке C++/CLI;
* тип структура;
* тип объединения;
* тип класс.

**Длина объединения.**

**Длина объединения** – это размер памяти в байтах, которая выделяется для одной переменной этого типа объединения.

Длина объединения вычисляется как максимум из всех длин (размеров в байтах) отдельных полей шаблона. Следует напомнить, что одно поле – это объявление одной

переменной в объединении.

**Пример:**

Пусть задан тип объединения, которое содержит переменные типов с плавающей точкой

// объявление типа "объединение Floats"

*union Floats*

*{*

*float f;* // рассматривается 4 байта

*double d;* // рассматривается 8 байт

*};*

Тип объединения Floats содержит 2 переменные с именами f и d. Переменная f есть типа float, переменная d есть типа double. Для переменной f типа float рассматривается (принимается во внимание) 4 байта. Для переменной d типа double принимается во

внимание 8 байт, так как компилятор выделяет для этого типа именно 8 байт.

Чтобы использовать объединение Floats в другом программном коде (методе, обработчике события и т.п.) нужно объявить переменную типа Floats как показано ниже

*Floats Fl;*

*int d;*

*Fl.f = 20.5;* // Fl.d не определено

*Fl.d = -100.35;* // теперь Fl.f не определено

*d = sizeof(Fl);* // d = 8

Поскольку размещение переменных в памяти условно начинается с одного адреса, то для переменной Fl типа Floats выделяется 8 байт памяти. Это связано с тем, что переменная типа double требует больше памяти для своего представления чем переменная типа float.

На рисунке отображено размещение (интерпретация) переменных f, d из объединения Floats.



5. Как осуществляется доступ к полям объединения?

Доступ к полям объединения осуществляется так же, как и для структуры:

* с помощью символа ‘ . ‘;
* с помощью последовательности символов ‘->’ в случае, когда объявлена переменнаяуказатель на объединение.

**Пример объявления и использования указателя (\*) на объединение**

Работа объединений с неуправляемыми (\*) указателями точно такая же, как и работа

структур с неуправляемыми указателями.

В нижеследующем примере объявляется неуправляемый указатель на объединение типа

Ints// указатель на объединение

Ints \*p; // неуправляемый указатель

// выделить память для объединения

p = new Ints;

// доступ к полям с помощью указателя

pI->a = 200;

pI->b = 3400;

⇑

**Как объявить вложенные объединения (структуры, классы) в шаблоне объединения?**

Шаблон объединения может включать поля, что есть структурами, объединениями и классами.

В примере ниже объявляется шаблон объединения с именем Types, содержащий два вложенных объединения Floats и Ints, структуру ArrayOfChars и класс MyPoint. Объявление структур и объединений имеет следующий вид

// объединение целочисленных типов

*union Ints*

*{*

*unsigned short int a;*

*unsigned int b;*

*unsigned long int c;*

*};*

// структура, содержащая 2 строки

*struct ArrayOfChars*

*{*

*char A[10];*

*char B[8];*

*};*

// объявление типа "объединение Floats"

*union Floats*

*{*

*float f;* // рассматривается 4 байта

*double d;* // рассматривается 8 байт

*};*

Объявление шаблона класса имеет следующий вид:

// класс объявляется в отдельном модуле, например MyPoint.h

*#pragma once*

*class MyPoint*

*{*

*public:*

*int x; int y;*

// методы класса

*int GetX(void) { return x; }*

*int GetY(void) { return y; }*

*void SetXY(int nx, int ny) { x = nx; y = ny; }*

*};*

Объявление типа объединение Types с вложенными сложными типами Ints, Floats,

*ArrayOfChars, MyPoint.*

// подключить модуль с объявленным классом MyPoint

*#include "MyPoint.h"*

*...*

// объявление типа "объединение Types"

*union Types*

*{*

*Floats Fl;* // объединение

*Ints I;* // объединение

*ArrayOfChars A;* // структура

*MyPoint MP;* // класс

*};*

Использование объединения Types в некотором программном коде:

// объявить переменную типа "объединение Types"

*Types T;*

// изменить значения полей переменной T

*T.Fl.f = (float)20.35;* // объединение Floats

*T.I.b = 230;* // объединение Ints

*T.A.A[2] = 'A';* // структура ArrayOfChars

*T.MP.SetXY(3,8);* // класс MyPoint

*int d;*

*d = T.MP.GetX();* // d = 3

**Массив объединений.**

// Пример объявления и использования массива объединений

*Floats F[5];* // объявляется массив из 5 объединений типа Floats

// заполнение значений полей

*for (int i=0; i<5; i++)*

*{*

*F[i].d = i\*0.2 + i\*i;*

*}*

1. Типы функций и их определение. Описание параметров. Аргументы функции: формальные и фактические. Наличие нескольких аргументов. Вызов функции. Обращение к функции до ее определения. Возвращение значения функции.

**Функция** — это самостоятельная единица программы, которая спроектирована для реализации конкретной подзадачи.

Функция является подпрограммой, которая может содержаться в основной программе, а может быть создана отдельно (в библиотеке). Каждая функция выполняет в программе определенные действия.

Сигнатура функции определяет правила использования функции. Обычно сигнатура представляет собой описание функции, включающее имя функции, перечень формальных параметров с их типами и тип возвращаемого значения.

Семантика функции определяет способ реализации функции. Обычно представляет собой тело функции.

**Определение функции**

Каждая функция в языке Си должна быть определена, то есть должны быть указаны:

*тип возвращаемого значения;*

*имя функции;*

*информация о формальных аргументах;*

*тело функции.*

Определение функции имеет следующий синтаксис:

*ТипВозвращаемогоЗначения ИмяФункции(СписокФормальныхАргументов)*

*{*

*ТелоФункции;*

*...*

*return(ВозвращаемоеЗначение);*

*}*

Пример: Функция сложения двух вещественных чисел

*float function(float x, float z)*

*{*

*float y;*

*y=x+z;*

*return(y);*

*}*

В указанном примере возвращаемое значение имеет тип float. В качестве возвращаемого значения в вызывающую функцию передается значение переменной y. Формальными аргументами являются значения переменных x и z.

Если функция не возвращает значения, то тип возвращаемого значения для нее указывается как void. При этом операция return может быть опущена. Если функция не принимает аргументов, в круглых скобках также указывается void.

**Различают системные (в составе систем программирования) и собственные функции.**

**Системные функции** хранятся в стандартных библиотеках, и пользователю не нужно вдаваться в подробности их реализации. Достаточно знать лишь их сигнатуру. Примером системных функций, используемых ранее, являются функции printf() и scanf().

**Собственные функции** — это функции, написанные пользователем для решения конкретной подзадачи.

Разбиение программ на функции дает следующие преимущества:

Функцию можно вызвать из различных мест программы, что позволяет избежать повторения программного кода.

Одну и ту же функцию можно использовать в разных программах.

Функции повышают уровень модульности программы и облегчают ее проектирование.

Использование функций облегчает чтение и понимание программы и ускоряет поиск и исправление ошибок.

С точки зрения вызывающей программы функцию можно представить как некий «черный ящик», у которого есть несколько входов и один выход. С точки зрения вызывающей программы неважно, каким образом производится обработка информации внутри функции. Для корректного использования функции достаточно знать лишь ее сигнатуру.

**Вызов функции**

Общий вид вызова функции

*Переменная = ИмяФункции(СписокФактическихАргументов);*

**Фактический аргумент** — это величина, которая присваивается формальному аргументу при вызове функции. Таким образом, **формальный аргумент** — это переменная в вызываемой функции, а **фактический аргумент** — это конкретное значение, присвоенное этой переменной вызывающей функцией. Фактический аргумент может быть константой, переменной или выражением. Если фактический аргумент представлен в виде выражения, то его значение сначала вычисляется, а затем передается в вызываемую функцию. Если в функцию требуется передать несколько значений, то они записываются через запятую. При этом формальные параметры заменяются значениями фактических параметров в порядке их следования в сигнатуре функции.

**Возврат в вызывающую функцию**

По окончании выполнения вызываемой функции осуществляется возврат значения в точку ее вызова. Это значение присваивается переменной, тип которой должен соответствовать типу возвращаемого значения функции. Функция может передать в вызывающую программу только одно значение. Для передачи возвращаемого значения в вызывающую функцию используется оператор return в одной из форм:

*return(ВозвращаемоеЗначение);*

*return ВозвращаемоеЗначение;*

Действие оператора следующее: значение выражения, заключенного в скобки, вычисляется и передается в вызывающую функцию. Возвращаемое значение может использоваться в вызывающей программе как часть некоторого выражения.

Оператор return также завершает выполнение функции и передает управление следующему оператору в вызывающей функции. Оператор return не обязательно должен находиться в конце тела функции.

Функции могут и не возвращать значения, а просто выполнять некоторые вычисления. В этом случае указывается пустой тип возвращаемого значения void, а оператор return может либо отсутствовать, либо не возвращать никакого значения:

*return;*

Пример: Посчитать сумму двух чисел

*#define \_CRT\_SECURE\_NO\_WARNINGS* // для возможностииспользования scanf

*#include <stdio.h>*

// Функция вычисления суммы двух чисел

*int sum(int x, int y)* // в функцию передаются два целых числа

*{*

*int k = x + y;* // вычисляем сумму чисел и сохраняем в k

*return k;* // возвращаем значение k

*}*

*int main()*

*{*

*int a, r;* // описание двух целых переменных

*printf("a= ");*

*scanf("%d", &a);* // вводим a

*r = sum(a, 5); /*/ вызов функции: x=a, y=5

*printf("%d + 5 = %d", a, r);* // вывод: a + 5 = r

*getchar(); getchar();* // мы использовали scanf(),

*return 0;* // поэтому getchar() вызываем дважды

*}*

**Результат выполнения**

Сумма двух чисел

В языке Си нельзя определять одну функцию внутри другой.

В языке Си нет требования, чтобы семантика функции обязательно предшествовало её вызову. Функции могут определяться как до вызывающей функции, так и после нее. Однако если семантика вызываемой функции описывается ниже ее вызова, необходимо до вызова функции определить прототип этой функции, содержащий:

*тип возвращаемого значения;*

*имя функции;*

*типы формальных аргументов в порядке их следования.*

Прототип необходим для того, чтобы компилятор мог осуществить проверку соответствия типов передаваемых фактических аргументов типам формальных аргументов. Имена формальных аргументов в прототипе функции могут отсутствовать.

Если в примере выше тело функции сложения чисел разместить после тела функции main, то код будет выглядеть следующим образом:

*#define \_CRT\_SECURE\_NO\_WARNINGS* // для возможностииспользования scanf

*#include <stdio.h>*

*int sum(int, int);* // сигнатура

*int main()*

*{*

*int a, r;*

*printf("a= ");*

*scanf("%d", &a);*

*r = sum(a, 5);*  // вызов функции: x=a, y=5

*printf("%d + 5 = %d", a, r);*

*getchar(); getchar();*

*return 0;*

*}*

*int sum(int x, int y)* // семантика

*{*

*int k;*

*k = x + y;*

*return(k);*

*}*

1. Функции. Способы передачи информации в функции.

В языке С++ данные в подпрограмму можно передавать тремя способами: по значению, по адресу и по ссылке.

**Передача данных по значению.**

Этот способ передачи данных в подпрограмму является основным и действует по-умолчанию. Фактический параметр вычисляется в вызывающей функции и его значение передаётся на место формального параметра в вызываемой функции. На этом связь между фактическим и формальным параметрами прекращается.

В качестве фактического параметра можно использовать константу, переменную или более сложное выражение. Передача данных по значению пригодна только для простых данных, которые являются входными параметрами. Если параметр является выходным данным или массивом, то передача его в функцию по значению не приемлема.

Но что делать, когда выходных параметров два или более? Через имя функции можно вернуть только один объект, остальные придётся возвращать через список. Позволит ли этот способ (передача по значению) вернуть через список параметров изменённые значения? Нет, не позволит. Давайте проверим это на несложном примере. Этот пример наглядно показывает, что через параметры, передаваемые по значению, нельзя вернуть результаты работы функции.

**Передача данных по адресу**

По адресу в функцию всегда передаются массивы (рассмотрим это в следующих темах). Для массива это вообще единственный способ передачи данных в языках С/С++. Так же по адресу можно передать те простые объекты, которые являются выходными данными (или входными и выходными одновременно).

В случае передачи данных по адресу фактический параметр может быть только переменной (константа или выражение не имеют адреса!). Печать на экране монитора наглядно показывает, что обмен произведён, и исходные переменные теперь имеют новые значения, т.е. передача данных по адресу действительно позволяет вернуть в вызывающую функцию результат работы вызываемой подпрограммы.

В вызывающей функции (в нашем случае — в main()) вычисляются адреса объектов, передаваемых по адресу ( у нас — адреса переменных a и b. Пусть это будут числа 1000 и 1008), и затем эти адреса копируются в ячейки памяти — указатели, память под которые выделена в функции Obmen() (это x и y). Зная адрес переменной, например, адрес переменной a, который теперь хранится в указателе x, можно, пользуясь операцией разыменование, не только прочитать, но и изменить значение исходной переменной.

Ни какой реальной передачи данных (в смысле копирования) из подпрограммы Obmen() назад в main() не делается. Мы на самом деле через указатели работаем с исходными объектами! Поэтому после выхода из функции Obmen() имеем изменённые переменные a и b (если быть точнее, переменные изменятся ещё до выхода из функции,то есть в момент перестановки в самой функции Obmen()).

**Передача данных по ссылке**

Это ещё один из способов вернуть результат работы функции через список параметров. Напомним, что применяется только для С++. В языке С такого варианта нет.

При передаче данных по ссылке в функцию, куда передаются данные, создаются синонимы исходных объектов. Поэтому работа в подпрограмме ведётся именно с исходными объектами. Если в подпрограмме ссылочная переменная изменит значение, то это сразу отразится на исходной переменной.

В вызывающей функции параметр, передаваемый по ссылке, может быть только простой переменной любого известного типа.

Вернёмся снова к примеру обмена, только данные передадим по ссылке. Обратите внимание на то, что для объекта, передаваемого по ссылке, в списке формальных параметров указывается значок & перед именем переменной (работаем с ссылкой на объект), а далее в теле подпрограммы используется просто имя этой переменной. При вызове в списке фактических параметров задаётся имя нужной переменной.

Для простых объектов передача данных по ссылке предпочтительнее, чем передача по адресу, так как в этом случае текст функции проще, легче читается, не требуется выполнять операции разыменования.

К недостатку способа (передача данных по ссылке) можно отнести то, что по вызову функции нельзя понять, что параметр передаётся именно по ссылке, а не по значению, и как следствие, этот параметр скорее всего изменяется функцией. Чтобы не было сомнений, всегда смотрите на запись прототипов функций, использованных в программе. Прототипы всегда доступны, даже если сами функции имеются только в виде объектных модулей.

1. Функции: передача аргументов (числовых, символьных, строк символов) в функции

Ответ логически из 11 и 12.

1. Файлы. Установка файловой переменной. Функции, работающие с файлами: FOPEN(), FCLOSE(), FREAD().

**Файл** – это именованная область ячеек памяти, в которой хранятся данные одного

типа. Файл имеет следующие характерные особенности:

* уникальное имя;
* однотипность данных;
* произвольная длина, которая ограничивается только емкостью диска.51

Файлы бывают текстовыми и двоичными.

**Текстовый файл** – файл, в котором каждый символ из используемого набора хранится в виде одного байта (код, соответствующий символу). Текстовые файлы разбиваются на несколько строк с помощью специального символа "конец строки". Текстовый файл

заканчивается специальным символом "конец файла".

**Двоичный файл** – файл, данные которого представлены в бинарном виде. При записи в двоичный файл символы и числа записываются в виде последовательности байт (в своем внутреннем двоичном представлении в памяти компьютера).

Все операции ввода-вывода реализуются с помощью функций, которые находятся в библиотеке С++. Библиотека С++ поддерживает три уровня ввода-вывода:

* потоковый ввод-вывод;
* ввод-вывод нижнего уровня;
* ввод-вывод для консоли и портов (зависит от ОС).

**Поток** – это абстрактное понятие, относящееся к любому переносу данных от источника к приемнику.

Функции библиотеки ввода-вывода языка С++, поддерживающие обмен данными с файлами на уровне потока, позволяют обрабатывать данные различных размеров и форматов, обеспечивая при этом буферизованный ввод и вывод. Таким образом, поток представляет собой этот файл вместе с предоставленными средствами буферизации.

Чтение данных из потока называется извлечением, вывод в поток – помещением (включением). Для работы с файлом в языке C++ необходима ссылка на файл. Для определения такой ссылки существует структура FILE, описанная в заголовочном файле stdio.h. Данная структура содержит все необходимые поля для управления файлами, например: текущий указатель буфера, текущий счетчик байтов, базовый адрес буфера ввода-вывода, номер файла.

**Открытие файла fopen()**

Функцией управляют три параметра.

*FILE \*in;* //указатель потока

Для связывания указателя с файлом служит функция открытия файла fopen(), которая объявлена в заголовочном файле <stdio.h>.

*in = fopen("test", "r");*

1 параметр - имя открываемого файла

2 параметр -

"r"-по чтению "r+"-чтение и запись

"w"-по записи "w+"-запись и чтение, если файл уже был, он перезаписываетя

"a"-дозапись "a+"-чтение и дозапись, если файла еще не было, он создается

"b"-двоичный файл

"t"-текстовый файл

in является указателем на файл "test". Теперь будем обращаться к файлу через этот указатель.

Если файл не был открыт (его нет, нет места на диске), то возвращается в указатель 0.

*if((in=fopen("test", "r"))==0)*

*puts("Ошибка открытия файла");*

Можно по-другому

*in=fopen("test", "r");*

*if (!in)*

*puts("Ошибка открытия файла");*

**Закрытие файла fclose()**

fclose(FILE \*stream); //Если файл закрыт успешно, то возвращается 0 иначе -1.

**Функции ввода/вывода одного символа fgetc(), fputc()**

1. Чтение из файла посимвольно

*int fgetc(FILE \*stream);*

1. Запись в файл посимвольно

*int fputc(int c, FILE \*stream);*

Код этого же символа и возвращается.

*ch=fgetc(in);*

*fputc(ch,out);*

*# include <stadio.h>*

*void main(void){*

*FILE \*in,\*out;*

*char ch;*

*if((in=fopen("prog1", "r"))==0)*

*fputs("Ошибка открытия prog1");*

*if((out=fopen("prog2", "w"))==0)*

*fputs("Ошибка открытия prog2);*

*while((ch=getc(in))!=EOF)* //”End Оf File” константа определенная в

dos.h

*fputc(ch, out);*

*fclose(in);*

*fclose(out);*

*}*

**Функции форматированного ввода/вывода в файл**

1. Форматированный вывод в текстовый файл

*int fprintf(FILE \*stream,”управл.cтрока”,arg1,…)*

Возвращает количество записанных байтов.

1. Форматированный ввод из текстового файла

*int fscanf(FILE \*stream,”управл.cтрока”,&arg1,…)*

Возвращает количество прочитанных байтов.

*# include<stadio.h>*

*void main(void){*

*FILE\*in;*

*int age;*

*in=fopen("prog1", "r");*

*fscanf(in,"%d",&age);*

*fclose(in);*

*in=fopen("prog2", "w");*

*fprintf(in,"число %d\n",age);*

*fclose(in);*

*}*

Модификаторы и спецификаторы, те же, что и в printf().

**Функции ввода/вывода строки символов в файл**

1. Чтение текстовой строки из файла.

*char\* fgets(char \*str, int n, FILE \*stream);*

Читает до перевода строки или n-1 байт и к концу строки присоединяет 0 байт, если прочитан \n.

*void main(void){*

*FILE\*in;*

*char string[80];*

*in=fopen("story", "r");*

*while(fgets(string,80, in)!=0)*

*puts(string);*

*}*

Считывается до конца строки '\n' или 80-1 байт. При встрече EOF возвращает 0.

1. Запись текстовой строки в файл

*int fputs(char \*str,FILE \*stream);*

*y=fputs(“Это строка”,in);*

y-целое число, которое устанавливается в EOF, если fputs() встре-чает EOF или ошибку. fputs не добавляет '\n' в конeц строки.

Все эти функции работают с текстовыми файлами и называются функциями последовательного доступа к файлу. Указатель внутри файла перемещается автоматически при чтении или записи.

Существуют функции прямого доступа к файлу.

**Функции управления указателем в файле**

Функция позволяет работать с файлом как с массивом. Достигать любого байта.

*int fseek(FILE \*stream, смещение, start)*

Возвращает число типа int:

0 - если все хорошо;

-1 - ошибка.

**Смещение** – это количество байт на которое нужно сместить указатель по файлу с +(вперед), -(назад);

start - код начальной точки:

SEEK\_SET или 0 – от начала файла;

SEEK\_END или 2 – от конца файла;

SEEK\_CUR или 1 – от текущего положения курсора.

fseek(in,0,0) - установить курсор на начало файла.

long int ftell(FILE \*stream)- возвращает текущее положение курсора в файле.

**Ввод/вывод записей фиксированной длины**

Под записью фиксированной длины можно понимать размер элемента массива или структуры.

1. Чтение данных из двоичного файла.

*int fread(void \*ptr, size type, size n, FILE \*stream)*

void \*ptr – адрес массива, куда записываются данные;

size type – размер типа в байтах;

size n – количество данных;

FILE \*stream – указатель на файл.

void main(void){

struct STOK record;

FILE \*in;

in=fopen("data", "r");

int n=fread(&record, sizeof(record), 1, in);

}

Возвращает число считанных записей или EOF.

*void main(void){*

*float mas[100];*

*FILE \*in;*

*In=fopen("data", "rb");*

*fread(mas, sizeof(mas), 1, in);*

*}* //можно так - fread(mas, sizeof(float), 100, in);

1. запись данных в двоичный файл.

*int fwrite(void \*ptr, size type, size n, FILE \*stream)*

Возвращает число записанных байт.

void \*ptr – адрес массива, куда записываются данные;

size type – размер типа в байтах;

size n – количество данных;

FILE \*stream – указатель на файл.

*fwrite(mas, sizeof(mas), 1, in);*

Пример 1. Запись во временный файл и чтение из него в массив.

*#include <stdio.h>*

*#include <stdlib.h>*

*void main(void) {*

*int array[100];*

*//создать временный файл*

*FILE \*tempf=tmpfile();*

*if(!tempf) {puts(“нельзя открыть временный файл”);*

*exit(1);*

*}*

*for(int index=0; index<100; index++)//пишем в файл*

*fwrite(array,sizeof(int),1,tempf);*

*rewind(tempf); //указатель вернуть на начало*

*fread(array,sizeof(int),100,tempf);*

*rmtmp(); //закрыть и уничтожить временный файл*

*}*

Пример 2. Проверить конец файлового потока

*void main(void) {*

*int buff[100];*

*FILE \*fp;*

*fp=fopen(“prog.txt”,”r”);*

*if(!fp) {*

*puts(“нельзя открыть файл”);*

*}*

*else {*

*while(!feof(fp))*

*if(fgets(buff,100,fp)!=NULL)*

*fputs(buff,stdout);*

*fclose(fp);*

*}*

*}*

Пример 3. Создание копии файла

*#include <stdio.h>*

*#include <stdlib.h>*

*void main(void)*

*{*

*FILE \*in,\*out;*

*char c;*

*fopen(“a.ddd”,”r”);*

*fopen(“b.ddd”,”r”);*

*fread(&c,1,1,in)*

*while(!feof(in))*

*{*

*fwrite(&c,1,1,out);*

*fread(&c,1,1,in)*

*}*

*fcloseall();*

*}*

1. Понятие класса и его свойства: абстрагирование, абстракция, ограничение доступа, иерархия, типизация, параллелизм, устойчивость.

**Класс (class)** - это тип, определяемый пользователем, включающий в себя данные и функции, называемые методами или функциями-членами класса.

**Данные класса** - это то, что класс знает.

**Функции-члены (методы) класса** - это то, что класс делает.

Объектно-ориентированная технология основывается на так называемой объектной модели. Основными ее принципами являются: абстрагирование, инкапсуляция, модульность, иерархичность, типизация, параллелизм и сохраняемость. Каждый из этих

принципов сам по себе не нов, но в объектной модели они впервые применены в совокупности.

Объектно-ориентированное программирование - это методология

программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

В данном определении можно выделить три части:

1) OOП использует в качестве базовых элементов объекты, а не алгоритмы (иерархия быть частью);

2) каждый объект является экземпляром какого-либо определенного класса;

3) классы организованы иерархически.

Программа будет объектно-ориентированной только при соблюдении всех трех указанных требований. В частности, программирование, не основанное на иерархических отношениях, не относится к OOП, а

называется программированием на основе абстрактных типов данных.

Для объектно-ориентированного стиля концептуальная база – это объектная модель. Она имеет четыре главных элемента:

* абстрагирование;
* инкапсуляция;
* модульность;
* иерархия.

Эти элементы являются главными в том смысле, что без любого из них модель не будет объектно-ориентированной. Кроме главных, имеются еще три дополнительных элемента:

* типизация;
* параллелизм;
* сохраняемость.

Называя их дополнительными, имеем в виду, что они полезны в объектной модели, но не обязательны.

Абстрагирование является одним из основных методов, используемых для решения сложных задач. Абстракция выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя.

Вот эти абстракции, начиная от наиболее полезных к наименее полезным:

|  |  |
| --- | --- |
| Абстракция сущности | Объект представляет собой полезную модель некой сущности в предметной области |
| Абстракция поведения | Объект состоит из обобщенного множества операций |
| Абстракция виртуальной машины | Объект группирует операции, которые либо вместе используются более высоким уровнем управления, либо сами используют некоторый набор операций более низкого уровня |
| Произвольная абстракция | Объект включает в себя набор операций, не имеющих друг с другом ничего общего |

**Инкапсуляция**

Абстракция и инкапсуляция дополняют друг друга: абстрагирование направлено на наблюдаемое поведение объекта, а инкапсуляция занимается внутренним устройством.

Чаще всего инкапсуляция выполняется посредством скрытия информации, то есть маскировкой всех внутренних деталей, не влияющих на внешнее поведение. Обычно скрываются и внутренняя структура объекта, и реализация его методов.

Инкапсуляция, таким образом, определяет четкие границы между различными абстракциями.

**Понятие модульности.**

Модульность - это свойство системы, которая была разложена на внутренне связные, но слабо связанные между собой модули.

Таким образом, принципы абстрагирования, инкапсуляции и модульности являются взаимодополняющими. Объект логически определяет границы определенной абстракции, а инкапсуляция и модульность делают их физически незыблемыми.

**Иерархия**

Абстракция - вещь полезная, но всегда, кроме самых простых ситуаций, число абстракций в системе намного превышает наши умственные возможности. Инкапсуляция позволяет в какой-то степени устранить это препятствие, убрав из поля зрения внутреннее содержание абстракций. Модульность также упрощает задачу, объединяя логически связанные абстракции в группы. Но этого оказывается недостаточно. Значительное упрощение в понимании сложных задач достигается за счет образования из абстракций иерархической структуры. Определим иерархию следующим

образом:

**Иерархия** - это упорядочение абстракций, расположение их по уровням.

Основными видами иерархических структур применительно к сложным системам являются структура классов (иерархия "is-a") и структура объектов (иерархия "part of").

**Типизация**

**Типизация** - это способ защититься от использования объектов одного класса вместо другого, или по крайней мере управлять таким использованием. Типизация заставляет нас выражать наши абстракции так, чтобы язык программирования, используемый в реализации, поддерживал соблюдение принятых проектных решений.

**Параллелизм**

Есть задачи, в которых автоматические системы должны обрабатывать много событий одновременно. В других случаях потребность в вычислительной мощности превышает ресурсы одного процессора. В каждой из таких ситуаций естественно использовать несколько компьютеров для решения задачи или задействовать многозадачность на многопроцессорном компьютере. Процесс (поток управления) - это

фундаментальная единица действия в системе. Каждая программа имеет по крайней мере один поток управления, параллельная система имеет много таких потоков: век одних недолог, а другие живут в течении всего сеанса работы системы. Реальная параллельность

достигается только на многопроцессорных системах, а системы с одним процессором имитируют параллельность за счет алгоритмов разделения времени. "Тяжелые" процессы управляются операционной системой независимо от других, и под них выделяется отдельное защищенное адресное пространство. "Легкие" сосуществуют в одном адресном пространстве. "Тяжелые" процессы общаются друг с другом через операционную систему, что обычно медленно и накладно. Связь "легких" процессов осуществляется гораздо проще, часто они используют одни и те же данные.

**Параллелизм** - это свойство, отличающее активные объекты от пассивных.

|  |
| --- |
| 1. Типы отношений между классами.   Большинство объектно-ориентированных языков непосредственно поддерживают разные комбинации следующих видов отношений в классах:   * **Ассоциация**   Ассоциации обеспечивают взаимодействия объектов, принадлежащих разным классам. Они являются клеем, соединяющим воедино все элементы программной системы. Благодаря ассоциациям мы получаем работающую систему. Без ассоциаций система превращается в набор изолированных классов-одиночек.   * **Наследование**   Наследование — это отношение, при котором один класс разделяет структуру и поведение, определенные в одном другом (простое наследование) или во многих других (множественное наследование) классах.   * **Агрегация**   Агрегация обеспечивает отношения целое-часть, объявляемые для экземпляров классов.   * **Использование**   Использование — это отношение, которое показывает, что изменение в одном классе (независимом) может влиять на другой класс (зависимый), который использует его.  Фиксирует отношение между клиентом, запрашивающим услугу, и сервером, предоставляющим эту услугу.   * **Инстанцирование** * **Метакласс**   Метакласс — это класс классов, понятие, позволяющее обращаться с классами как с объектами. |
| 1. Понятие объекта и его элементы: состояние, поведение, индивидуальность.   Объект — это абстракция предметной области, имеющая интерфейс в виде именованных операций и собственные данные, с ограничением доступа к ним.  Объект – это совокупность свойств и методов, а также событий, на которые он может реагировать. Понятия класса и объекта настолько тесно связаны, что невозможно говорить об объекте безотносительно к его классу. Однако существует важное различие этих двух понятий. В то время как объект обозначает конкретную сущность, определенную во времени и в пространстве, класс определяет лишь абстракцию существенного в объекте.  **Состояние**  Состояние объекта характеризуется перечнем всех  свойств данного объекта и текущими значениями  каждого из этих свойств  Свойство — это характеристики, черты, качества или способности, делающие данный объект самим собой  **Поведение**  Состояние объекта может измениться только в результате вызова методов. Объекты не существуют изолированно, а подвергаются воздействию или сами воздействуют на другие объекты. Поведение – это то, как объект действует или реагирует Поведение – это набор операций (методов) объекта Операция – это действие, которую объект может предоставить своим клиентам (другим объектам) Операцией называется определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию.  **Идентичность**  **Идентичность** - это такое свойство объекта, которое отличает его от всех других объектов.  Большинство баз данных различают постоянные объекты по ключевому атрибуту, тем самым смешивая идентичность и значение данных. |
| 1. Отношения между объектами.   Объекты взаимодействуют друг с другом, посылая и получая сообщения.  **Сообщение** – это запрос на выполнение действия, дополненный набором аргументов, которые могут понадобиться при выполнении действия.  В качестве реакции на сообщение получатель запустит некоторый метод, чтобы удовлетворить заданный запрос.  У сообщения имеется конкретный получатель. Интерпретация одного и того же сообщения зависит от получателя и является различной для  различных получателей. Получатель может по разному интерпретировать (вызывать различные методы) одно и то же сообщение в зависимости от передаваемых аргументов. |
| 1. Взаимоотношения между классами и объектами. |

Классы и объекты - это отдельные, но тесно связанные понятия. В частности, каждый объект является экземпляром какого-либо класса; класс может порождать любое число объектов. В большинстве практических случаев классы статичны, то есть все их особенности и содержание определены в процессе компиляции программы. Из этого

следует, что любой созданный объект относится к строго фиксированному классу. Сами объекты, напротив, в процессе выполнения программы создаются и уничтожаются. На этапе анализа и ранних стадиях проектирования решаются две основные задачи:

Выявление классов и объектов, составляющих словарь предметной области. Построение структур, обеспечивающих взаимодействие объектов, при котором выполняются требования задачи.

В первом случае говорят о ключевых абстракциях задачи (совокупность классов объектов), во втором - о механизмах реализации (совокупность структур). На ранних стадиях внимание проектировщика сосредоточивается на внешних проявлениях ключевых абстракций и механизмов. Такой подход создает логический

каркас системы: структуры классов и объектов. На последующих фазах проекта, включая реализацию, внимание переключается на внутреннее поведение ключевых абстракций и механизмов, а также их физическое представление. Принимаемые в процессе проектирования решения задают архитектуру системы: и архитектуру процессов, и

архитектуру модулей.

1. Приватные, общедоступные и защищенные члены класса.

Для каждого объекта класса устанавливается область видимости либо явно – указанием уровня доступа одним из ключевых слов public, private, protected с двоеточием, либо неявно – по умолчанию. Указание области видимости относится ко всем последующим объектам класса, пока не встретится указание другой области видимости. Область видимости public разрешает доступ к объектам класса из любой части программы, в которой известен этот объект (общедоступный). Область видимости private разрешает доступ к объектам

класса только из методов этого класса. Объекты с такой областью видимости называют частными. Область видимости protected определяется для защищенных объектов, она имеет смысл только в иерархической системе классов и разрешает доступ к объектам этой области из методов производных классов. В теле класса ключевое слово области видимости может использоваться неоднократно. Область видимости для объектов типа «класс» по умолчанию

private.

1. Конструктор и деструктор класса: объявление, описание и использование.

**Конструктор** – это метод класса, имя которого совпадает с именем класса. Конструктор вызывается автоматически после выделения памяти для переменной и обеспечивает инициализацию компонент – данных. Конструктор не имеет никакого типа (даже типа void) и не возвращает никакого значения в результате своей работы. Конструктор

нельзя вызывать как обычную компонентную функцию в программе. Для класса может быть объявлено несколько конструкторов, различающихся числом и типами параметров. При этом

даже если для объектного типа не определено ни одного конструктора, компилятор создает для него конструктор по умолчанию, не использующий параметров, а также конструктор копирования, необходимый в том случае, если переменная объектного типа передается в конструктор как аргумент. В этом случае создаваемый объект будет точной копией аргумента конструктора.

Еще одним специальным методом класса является деструктор. **Деструктор** вызывается перед освобождением памяти, занимаемой объектной переменной, и предназначен для выполнения дополнительных действий, связанных с уничтожением объектной переменной, например, для освобождения динамической памяти, закрытия, уничтожения файлов и т.п. Деструктор всегда имеет то же имя, что и имя класса, но перед именем записывается знак ~ (тильда). Деструктор не имеет параметров и подобно конструктору не возвращает никакого значения. Таким образом, деструктор не может быть перегружен и должен существовать в классе в единственном экземпляре. Деструктор вызывается автоматически при уничтожении объекта. Таким образом, для статически определенных объектов деструктор вызывается, когда заканчивается блок программы, в

котором определен объект (блок в данном случае – составной оператор или тело функции). Для объектов, память для которых выделена динамически, деструктор вызывается при уничтожении объекта операцией delete.

1. Конструктор с аргументами.

Конструктору можно передать параметры. Для этого нужно добавить необходимые параметры в объявление и определение конструктора. Затем при объявлении объекта параметры задаются в качестве аргумента.

Пример:

*#include <iostream.h>*

*class OurClass {*

*int a;*

*public:*

*OurClass(int x);* // конструктор

*void show();*

*};*

*OurClass::OurClass(int x)*

*{*

*cout << "В конструкторе\n";*

*a = x;*

*}*

*void OurClass::show(){*

*cout << a << "\n";*

*}*

*main()*

*{*

*OurClass ob(4);*

*ob.show();*

*return 0;*

*}*

Конструктор OurClass имеет один параметр. Значение, передаваемое в OurClass() используется для инициализации а. Аргумент 4 передается в ob(4) в качестве аргумента. Деструктор в отличие от конструктора параметров не имеет. В данном примере конструктору передавали константы, но так же можно передавать переменные:

Пример:

*#include <iostream.h>*

*class OurClass {*

*int i, j;*

*public:*

*OurClass(int a, int b);*

*void show();*

*};*

*OurClass::OurClass(int a, int b)*

*{*

*i = a;*

*j = b;*

*}*

*void OurClass::show()*

*{*

*cout << i << ' ' << j << "\n";*

*}*

*main()*

*{*

*int x, y;*

*cout << "Введите два целых: ";*

*cin >> x >> y;*

*OurClass ob(x, y);*// использование переменных для создания ob

*ob.show();*

*return 0;*

*}*

1. Объявление, описание, и использование функций-членов классов.

Функции, объявленные внутри описания класса называются функциями-членами (member functions) или методами.

Для определения методов используется форма:

*тип имя класса:: имя метода (параметры)*

*{*

*тело функции*

*}*

Два двоеточия после имени класса называются операцией расширения области видимости (scope resolution operator). Определение класса только определяет тип объектов, а сами объекты не задает (т.е.

память для них не выделяется). Для создания объектов имя класса используется как спецификатор типа данных.

После создания объекта к открытым членам класса можно обращаться, используя операцию точка.

Пример.

*#include <iostream.h>*

*class class1 { //*объвлен класс class1

*int a;* //доступна для методов class1

*public:*

*int kwadrat(int b);*//метод класса class1

*};*

*int class1::kwadrat(int b)* //определение метода kwadrat()

*{*

*a=b\*b;*

*return a;*

*}*

*main()*

*{*

*class1 c;* //создается объект с типа class1

*cout<<"\n"<<c.kwadrat(3)<<"\n";*//вычисление и вывод квадрата трех

*return 0;*

*}*

*class имя класса*

*{*

*закрытые функции–члены(методы) и переменные*

*public:*

*открытые функции, функции-члены (методы) и переменные*

*}*

*список объектов; //не является обязательным*

1. Статические члены-данные. Приватные статические члены-данные.

Статический член данных в классе объявляется с помощью ключевого слова static. Поскольку, это есть статический член данных, то нужно его определить (описать переменную) за пределами класса. Доступ к этому статическому члену данных имеют все экземпляры (объекты) класса.

Чтобы использовать статический член данных в классе,

необходимо:

• объявить статический член данных в теле класса (в любом разделе класса);

• определить статический член данных за пределами класса.

Если статический член данных объявлен в разделе public, то доступ к

нему можно получить одним из трех способов:

• с помощью символа ‘ . ‘ (точка), если объявлен объект (экземпляр)

класса;

• с помощью последовательности символов ‘->’, если объявлен указатель

на объект класса;

• с помощью имени класса и символов ‘::’ (расширение области

видимости).

1. Приватный конструктор и деструктор.

При объявлении обычного объекта класса, конструкторы, которые

размещены в разделе private (приватные конструкторы), есть недоступными.

Чтобы использовать приватные конструкторы, нужно выполнение

одного из условий:

• конструктор вызывается статическим членом класса;

• конструктор вызывается дружественным классом;

Использование приватных деструкторов целесообразно в тех случаях, когда обычным пользователям запрещается освобождать память для прежде созданных объектов (уничтожать раньше созданные объекты).

Если в классе объявлен приватный деструктор, то возникают следующие ограничения:

• невозможно создать автоматический объект класса (объект класса,

который объявляется в некотором методе);

• невозможно создать статический объект;

• невозможно создать глобальный объект класса.

Это связано с тем, что такие объекты в дальнейшем невозможно будет

уничтожить.

1. Объекты класса как члены-данные.

В классах могут объявляться члены данных класса, которые являются

объектами других классов.

*class A {*

*int x1;*

*public:*

*A() { cout << "\nconstructor A"; }*

*~A(){ cout << "\ndestructor A"; }*

*void f1(){ cout << "\n f1() out of A"; }*

*};*

*class B {*

*int x1;*

*public:*

*B() { cout << "\nconstructor B"; }*

*~B() { cout << "\ndestructor B"; }*

*void f1() { cout << "\n f1() out of B"; }*

*A W;*

*};*

*int main()*

*{*

*B Q;*

*}*

В классе B объявляется объект класса A, то при объявлении объекта

класса B конструкторы обоих классов вызываются в следующей

последовательности:

•конструктор класса A;

•конструктор класса B.

Такую последовательность действий компилятор формирует автоматически.

Память для членов-данных класса, которые есть объектами класса,

выделяется при выделении памяти для объекта класса, в котором эти

объекты реализованы. То есть, если описать объект класса B, в котором есть объект класса A, то память для объекта класса B выделяется с учетом (включает в себя) памяти объекта класса A.

1. Статические функции-члены: объявление, описание и использование.
2. Указатели на члены-данные класса. Указатели на члены-данные объекта.