1-20стр Доп – 15 раздел

21+стр основные лекции

Дополнительная тема для дополнительных баллов

**15. Классы памяти**

Классы памяти определяют область видимости и время жизни переменных и/или функций в программе, написанной на языке С. Они предшествуют типу, который модифицируют. В языке С различают четыре класса памяти:

auto (автоматический)

register (регистровый)

static (статический)

extern (внешний).

**15.1 ,,Зоопарк,, классов памяти**

(тут ничего, ибо это наименование всех следующих тем и классов памяти)

**15.1.1 Автоматические переменные**

Стандарт C++ определяет исходное и измененное значение для этого ключевого слова. До Visual Studio 2010 auto ключевое слово объявляет переменную в классе автоматического хранения, то есть переменную с локальным временем существования. Начиная с Visual Studio 2010, ключевое auto слово объявляет переменную, тип которой выводится из выражения инициализации в объявлении. Параметр /Zc:auto[-] компилятора управляет значением ключевого auto слова. Ключевое auto слово направляет компилятору использовать выражение инициализации объявленной переменной или лямбда-параметра выражения, чтобы вывести его тип.

Мы рекомендуем использовать ключевое auto слово для большинства ситуаций , если вам действительно не требуется преобразование, так как оно обеспечивает следующие преимущества:

Надежность: Если тип выражения изменяется ( включает в себя изменение типа возвращаемого значения функции), он просто работает.

Производительности: Вы гарантируете, что преобразование не будет выполнено.

Удобство использования: Вам не нужно беспокоиться о орфографических трудностях и опечаток имени типа.

Эффективность: Код может быть более эффективным.

**15.1.2 Регистровые переменные**

Регистровые переменные фактически являются также автоматическими переменные, но при их определении используется ключевое слово register. Это слово сообщает компилятору, что данная переменная будет интенсивно использоваться в приложении, поэтому ее желательно поместить в регистр процессора, что увеличит быстродействие.

Ключевое слово register применяется к переменным:

1

register int x = 8;

И к параметрам функции:

1

2

3

4

void display(register int a)

{

printf("a=%d \n", a);

}

Однако использование слова register еще не является гарантией, что переменная действительно будет помещена в регистр, так как много зависит от аппаратной части компьютера и установленных для нее ограничений. В то же время если аппаратные возможности не поддерживают регистровых переменных, то слово register просто будет игнорироваться, и его использование не вызовет ошибку.

**15.1.3 Статические переменные**

Кроме глобальных и автоматических есть особый тип переменных - статические переменные. Они определяются на уровне функций с помощью ключевого слово **static**. Если автоматические переменные определяются и инициализируются при каждом входе в функцию, то статические переменные инициализируются только один раз, а при последующих вызовах функции используется старое значение статической переменной.

Например, пусть у нас будет функция со стандартной автоматической переменной:

#include <stdio.h>

void display()

{

int i = 0;

i++;

printf("i=%d \n", i);

}

int main(void)

{

display();

display();

display();

return 0;

}

Функция display вызывается три раза, и при каждом вызове программа повторно будет выделять память для переменной i, которая определена в функции. А после завершения работы display, память для переменной i будет освобождаться. Соответственно ее значение при каждом вызове будет неизменно:

i=1

i=1

i=1

Теперь сделаем переменную i статической:

#include <stdio.h>

void display()

{

static int i = 0;

i++;

printf("i=%d \n", i);

}

int main(void)

{

display();

display();

display();

return 0;

}

К переменной был добавлено ключевое слово static, поэтому при завершении работы функции display переменная не уничтожается, ее память не очищается, наоборот, она сохраняется в памяти. И соответственно результат работы программы будет иным:

i=1

i=2

i=3

**15.1.4 Внешние переменные**

Переменная, имеющая внешние связи, называется внешней переменной. Она может использоваться как в файле, в котором определена, так и в других файлах. Глобальные переменные — это еще одно название внешних переменных. Такие переменные указываются даже вне метода. Во всей реализации метода такие переменные остаются доступными глобально. Методы могут изменять значение универсальных переменных. Внешние переменные объявляются и указываются с использованием «внешней» терминологии. Они не ограничены каким-либо методом. Они присутствуют в коде, т.е. поскольку такие переменные являются глобальными. Глобальные переменные имеют стандартное нулевое значение инициализации. Срок их жизни ограничен продолжительностью реализации программы. Такие внешние переменные должны быть указаны и описаны в другом месте кода, обычно в отдельном исходном файле. Каждую глобальную переменную можно было объявить в документе только один раз. Описание внешней переменной будет в примере ниже.

int hocus;

main()

{

extern int hocus; /\* hocus описана внешней \*/

...

}

magic()

{

extern int hocus;

...

}

В следующем примере будет внешняя переменная hocus, которая будет известна всем функциям main() и magic()

int hocus;

main()

{

extern int hocus; /\* hocus описана внешней \*/

…

}

magic()

{

/\* hocus не описана совсем \*/

...

}

Эти примеры иллюстрируют область действия внешних переменных. Они существуют, пока работает программа, и так как эти переменные доступны любой функции, они не исчезнут, если какая-нибудь одна функция закончит свою работу.

**15.1.5 Внешние статические переменные**

Можно так же описать любую статическую переменную вне любой функции. Разница в том, что у них разная область действия. Обычная внешняя переменная может пользоваться в любом файле, а статическая только внутри функции.

static randx = 1;

rand()

{

…

…

}

**15.2 Объявление переменных на внутреннем уровне**

Все 4 спецификатора класса памяти можно использовать для объявления переменных на внутреннем уровне. По умолчанию используется auto.

Переменная, объявленная на внутреннем уровне со спецификатором класса памяти static, имеет глобальное время жизни, но видима только в том блоке, в котором она объявлена. В отличие от переменных auto, переменные static сохраняют свои значения при выходе из блока. Можно инициализировать переменную static при помощи констант. По умолчанию начальное значение — ноль.

Переменная, объявленная со спецификатором класса памяти extern, является ссылкой на переменную с таким же именем, описанную на внешнем уровне в любом исходном файле программы. Внутреннее объявление extern используется для того, чтобы сделать видимым внутри блока описание переменной, сделанное на внешнем уровне. Это демонстрируется в следующей программе:

#include<stdio.h>

#include<iostream.h>

int ivalue1=1;

void function\_a(void)

{

// запоминается адрес глобальной переменной ivalue1;

static int \*pivalue1= &ivalue1;

// создается новая локальная переменная ivalue1; при этом глобальная

// переменная ivalue1 становится недоступной

int ivalue1 = 32;

// новая локальная переменная ivalue2

// видима только внутри функции function\_a

static int ivalue2 = 2;

ivalue2 += 2;

// печатаются числа 32, 4 и 1:

cout << " "<<ivalue1<<" " << ivalue2<< " "<< \*pivalue1;

}

main ()

{

// обращается к переменной ivalue1, описанной выше

extern int ivalue1;

//по умолчанию — начальное значение = 0, переменная ivalue2

// видима только в main()

static int ivalue2;

// запоминается в регистре (по возможности),

// начальное значение = 0

register int rvalue = 0;

//по умолчанию - класс памяти auto, переменная int\_value3

//получает начальное значение 0

int int\_value3 = 0;

// печатаются числа 1, 0, 0, 0:

cout << ivalue1 <<" "<< rvalue <<" "<<ivalue2 <<" "<< int\_value3;

function\_a() ;}

**15.3 Объявление переменных на внешнем уровне**

**При объявлении** переменных на внешнем уровне можно использовать только static и exern. Описание обьявления переменных внешних:

static int ivalue1; // по умолчанию подразумевается 0

static int ivalue1 = 10 // явная инициализация

int ivalue2 =20; // явная инициализация

Эта переменнная будет видна во всем файле, но не будет видна до ее описания. Но можно описать только один раз такую переменную.

Так как каждое статическое описание видимо только в соответствующем файле, то конфликтов не будет. Описание extern будет ниже:

//Исходный файл А

#include <iostream> //cout

#include <conio.h> //\_getch();

using namespace std;

void function\_a(void);

void function\_b(void);

int ivalue = 10; // действительное описание переменной

// ivalue

int main()

{

ivalue++; // используется объявление extern, приведенное выше

cout << ivalue << "\n"; // печатается 11

function\_a();

\_getch();

return(0);

}

extern int ivalue; // переменная ivalue определяется видимой

void function\_a(void)

{

ivalue++; // обращение к ivalue

cout << ivalue << "\n"; // печатается 12

function\_b();

}

//

//Исходный файл В

//

#include <iostream> //cout

using namespace std;

extern int ivalue; // ссылка на переменную ivalue,

// объявленную в Исходном файле А

void function\_b(void)

{

ivalue++;

cout << ivalue; // печатается 13

}

**15.4 Переменные класса volatile**

Бывают такие случаи, когда возможность модификации нужно посмотреть на другой программе или устройстве, а для этого как раз нужнен модификатор volatile. Описание таких переменных будет ниже:

volatile short sTest;

volatile const int vciTest;

Вообще volatile это квалификатор переменной, говорящий компилятору, что значение переменной может быть изменено в любой момент и что часть кода, которая производит над этой переменной какие-то действия (чтение или запись), не должна быть оптимизирована. Выделяют 3 ошибки при использовании класса:

1. Неиспользование там, где нужно
2. Использование там где нужно, но не так, как нужно
3. Использование там, где не нужно

Пример использования ниже:

void func(void) {

static volatile int \*\*ipp;

static volatile int \*ip;

static volatile int i = 0;

printf("i = %d.\n", i);

ipp = &ip;

\*ipp = &i;

if (\*ip != 0) {

/\* ... \*/

}

}

int main () {

func();

system("PAUSE");

return 0;

}

**15.5 Ключевое слово mutable**

Это слово используется тогда, когда есть необходимость изменить объект внутри класса, гарантируя неприкосновенность остальных элементов. В отличии от const он не все запрещает, а лишь нужное.

**class Exm**

**{**

**int a;**

**int b;**

**public:**

**int getA() const**

**{**

**return a; // все правильно**

**}**

**int setA(int i) const**

**{**

**a = i;// ошибка доступа**

**}**

**}**

Тут мешает const, поэтому попробуем использовать mutable:

**class Exm**

**{**

**mutable int a; // добавили в объявление ключевое слово mutable**

**// позволяющие игнорировать модификатор const**

**// по отношению к данной переменной**

**int b;**

**public:**

**int getA() const //**

**{**

**return a; // все правильно**

**}**

**int setA(int i) const**

**{**

**a = i;// теперь всё правильно. Мы можем изменять переменную а**

**b = i; // Ошибка! Переменная b по прежнему не доступна для изменения.**

**}**

**}**

Mutable используется, чтобы разрешить const переменной быть измененной. Он особенно полезен, если большое количество переменных должны быть константными, но некоторые из них с возможностью изменения.

**15.6 Классы памяти и область действия**

В основном, локальные переменные известны только функциям. В с++ есть возможность сделать переменные известными всем функциям.

Это можно сделать с помощью слова extern. Вообще есть 4 класса памяти, описывающих положение переменных в области: extern, auto, register, static. Определение класса памяти переменной зависит от того, где переменная описана и какое ключевое слово используется.

Класс памяти позволяет:

1. Определить, какие функции имеют доступ к переменной, а какие нет
2. Определить время нахождения переменной в памяти.

Пример работы ниже:

#include<stdio.h>

#include<iostream.h>

int units; /\* внешняя переменная \*/

critic()

{

extern int units;

printf (" You haven't good luck. Try again\n");

//Тебе не везет, попробуй еще

scanf (" %d", &units);

}

main()

{

extern int units;

printf (" How much pounds of butter is on a barrel?\n");

//Сколько фунтов масла в бочонке?

scanf (" %d", &units);

while (units != 56)

critic();

printf(" You must looking through the directory \n");

//Вы должны посмотреть в справочнике

}

Кстати, как раз значение units было прочитано critic(), хотя main() тоже узнала это новое значение, когда оно вышло из цикла while

**15.6.1 Правила области действия переменной**

Основные правила области действия переменной:

1. Блок
2. Функция
3. Файл
4. Программа

Переменная внутри блока и функции известна только в блоке и функции. Переменная в файле известна всем функциям. Аналогично и с программой – известна во всех функциях.

**15.6.2 Операция уточнения области действия в С++**

Обычно переменные в функциях и блоках имеют область только тут же. Если же вы хотите использовать переменную во всех функциях, то описывать ее нужно вне функции, в самом файле либо программе.

Есть такой вариант: задать переменную с областью действия внутри файла, а затем включить ее в описание функции, сделав область ее действия локальной. В C++ имеется новое программное средство, называемое операцией уточнения области действия (::). Когда используется эта операция, выполняется обращение не к локальной переменной, а к переменной с областью действия внутри всего файла. Тогда переменная становится глобальной. Пример его использования ниже:

#include <iostream> //cout

#include <conio.h> //\_getch();

using namespace std;

int iproduct(int iw,int ix);

int in=10;

int main()

{

int il=3;

int im=7;

int io;

io=iproduct(il,im);

cout << "The product of " << il <<" \* " << im

<< " \* " << in << " is: " << io;

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

int iproduct(int iw,int ix)

{

int iy;

int in=2;

iy=iw\*ix\*(::in); //iy=iw\*ix\*(in);

return(iy);}

Данная операция дает большие возможности в С++. Правила области действия для переменных обуславливают появление своеобразных программных ошибок. Дополнительные примеры применения этой операции приводятся в последующем изложении материала.

**15.6.3 Выбор класса памяти**

По умолчанию всегда выбран класс auto. Сделано это из-за того, что в большинстве своем внешняя переменная в одном классе изменяется и изменяется она в другом классе, поэтому по умолчанию стоит класс auto. Опыт, накопленный в использовании языков, свидетельствует, что такая проблема будет важнее, чем соблазн использования внешних переменных, которые звучат очень удобными в использовании. Наилучшим способом будет автоматизация каждой функции отдельно, насколько это возможно.

Класс памяти определяет область действия переменной и продолжительность ее существования в памяти. Класс памяти устанавливается при описании переменной с соответствующим ключевым словом. Переменные, определенные вне функции, являются внешними и имеют глобальную область действия. Переменные, определенные внутри функции, являются автоматическими и локальными, если только не используются другие ключевые слова. Внешние переменные, определенные раньше функции, доступны ей, даже если не описаны внутри ее.

Классы памяти определяют область видимости и время жизни переменных и/или функций в программе, написанной на языке С. Они предшествуют типу, который модифицируют. В языке С различают четыре класса памяти:

auto (автоматический)

register (регистровый)

static (статический)

extern (внешний).

**15.7 Пространства имен**

На данный момент определения функций и переменных в заголовочных файлах связаны с понятием пространства имен. Раньше все объявления переменных помещались в один заголовочный файл, но возникала проблема: постоянные конфликты имен. Так же часто возникали проблемы, когда в одной программе использовались библиотеки, разработанные различными производителями.

Введение понятия пространства имен позволило значительно снизить количество подобных конфликтов имен. Когда в программу включается заголовок нового стиля, его содержимое помещается не в глобальное пространство имен, а в пространство имен std. Если в программе требуется определить некоторые идентификаторы, которые, как Вы подозреваете, могут переопределить уже имеющиеся, просто заведите свое собственное, новое пространство имен. Это достигается путем использования ключевого слова namespace:

namespace \_имя\_пространства\_

{

// объявления

…

}

Таким образом, объявления внутри нового пространства имен будут находится только внутри видимости определенного имени \_пространства\_имён, предотвращая тем самым возникновение конфликтов.

Если в программе обращения к собственному пространству имен производятся довольно часто, такой синтаксис вызывает определенные неудобства. В качестве альтернативы можно воспользоваться инструкцией using, синтаксис которой имеет две формы:

using namespace имя\_пространства имен;

или

using имя\_пространства\_имен::идентификатор;

При использовании первой формы компилятору сообщается, что в дальнейшем необходимо использовать идентификаторы из указанного именного пространства вплоть до того момента, пока не встретится следующая инструкция using.

Может допускаться такое, что вы хотите объявить несколько именованных пространств с одним и тем же именем, что позволяет разделить его на несколько файлов. Несмотря на это, содержимое всех частей будет храниться в одном и том же пространстве имен. Пример ниже:

namespace Nspace

{

char с;

int i;

void Fund (char Flag);

}

void Nspace::Fund(char Flag)

{

// тело функции

…

}

**15.8 Функции и классы памяти**

Как бы удивительно не звучало, но функции тоже могут иметь класс extern и static. Такое может быть, если функция описана в другом файле, и она будет внешняя по отношению к нашей программе с помощью extern. Так же с помощью static можно сделать функцию такой, чтобы к ней невозможно было обращаться извне.

Переменные в функции являются ее внутренними переменными и неизвестны снаружи. Для этого мы используем return, чтобы мы могли вывести итог работы функции в другую.

**15.8.1 Объявления функций на внешнем уровне**

И снова возвращаемся к static и extern. Да-да, их тоже можно использовать.

Функции, объявленные как static, видимы только в том исходном файле, в котором они описаны. Функции, находящиеся в том же файле, могут вызывать статическую функцию, а функции из других файлов — не могут. Кроме того, в другом исходном файле можно объявить другую статическую функцию с таким же именем, и не будет конфликта.

Функции, объявленные как extern, видимы во всех исходных файлах, образующих программу (если вы где-то не переопределите функцию как static). Внешнюю функцию могут вызывать все функции.

Но главное запомнить! По умолчанию стоит external.

**15.9 Функция получения случайных чисел**

При создании массивов, при создании других структур данных, где нужны случайные числа, вы можете воспользоваться к функциям получения рандомных чисел. Первый ,,генератор,, чисел - randx. Но она неудобна тем, что у нее должно быть число, на которое она будет опираться, и взамен которого будет ставиться рандомное число. Описание на примере ниже:

rand()

{

static int randx = 1;

randx = (randx \* 25173 + 13849) % 65536; /\* магическая формула \*/

return( randx);

}

Переменная randx начинает со значения 1 и изменяется на рандомное число с помощью формулы. Теперь вам выведется число в диапазоне от -32768 до 32767. Но этот диапазон можно изменить типом переменной int. Но проблема в том, что при повторном запуске программы у нас выведутся те же самые числа. Как это исправить? Есть замечательная функция srand(), которая возвращает рандомное число к единице (изначальному значению) и заново рандомайзит его. Пример ниже:

/\* файл для rand() и srand() \*/

static int randx =1;

rand()

{

randx = (randx \* 25173 + 13849) % 65536;

return(randx);

}

srand (unsigned x)

{

randx = x;

}

Используем другой драйвер:

/\* драйвер2 функции rand() \*/

main()

{

int count;

int seed;

printf(" Please, enter your value of a corn\n");

//Пожалуйста, введите свое значение зерна

scanf(" %d" , &seed);

srand(seed);/\* установите зерно в начальное значение \*/

for (count = 1; count <= 5; count++ )

printf(" %d\n" , rand());

}

}

Теперь у нас каждый раз будут выводиться разные числа!

**15.10 Игра в кости**

Давайте начнем с 6 сторон, затем сделаем обобщение. Нам нужны случайные числа от 1 до 6, а функция rand() создает числа в диапазоне от —32768 до 32767, поэтому мы должны внести некоторые поправки. Вот один подход

Разделим случайное число на 32768. В результате получим число х в диапазоне -1 < = х < 1. (Мы должны превратить его в тип float, чтобы иметь десятичные дроби.)

Добавим 1. Наше новое число удовлетворяет отношению 0 <= х < 2.

Разделим на 2. Теперь имеем 0 < = х < 1.

Умножим на 6. Имеем 0 < = х < 6. (Близко к тому, что нужно, но 0 не является возможным значением.)

Прибавим 1: 1 < = х < 7. (Заметим, что эти числа все еще являются десятичными дробями.)

Преобразуем в целые числа. Теперь мы имеем целые в диапазоне от 1 до 6.

Для обобщения достаточно заменить значение 6 в п. 4 на число сторон.

Вот функция, которая выполняет эти действия:

/\* электронное бросание костей \*/

#define SCALE 32768.0

rollem (sides);

float sides;

{

float roll;

roll = ( (float) rand()/SCALE + 1.0) \* sides/2.0 + 1.0;

return ( (int) roll);

}

Мы включили в программу два явных описания типа, чтобы показать, где выполняются преобразования типов.

Обратимся к программе, которая использует эти средства:

/\* многократное бросание кости \*/

main()

{

int dice, count, roll, seed; float sides;

printf(" Введите, пожалуйста, значение зерна. \n");

scanf(" %d", &seed);

srand (seed);

printf(" Введите число сторон кости, 0 для завершения\n");

scanf(" %d" , &sides);

while (sides > 0)

{

printf(" Сколько костей? \n" );

scanf(" %d", dice);

for ( roll = 0, count = 1; count <= dice; count++ )

roll += rollem(sides); /\* бросание всех костей набора \*/

printf("У вас выпало %d, для %d %. 0f-сторонних костей.\n" , roll, dice, sides);

printf(" Сколько сторон? Введите 0 для завершения.\n");

scanf("%f", &sides);

}

printf(" Удачи вам!\n");

}

Введите значение зерна.

10

Введите количество ходов; введите 0 для завершения.

18

Сколько сторон и сколько костей?

6 3

Здесь 18 ходов для 3 6-сторонних костей.

7 5 9 7 12 10 7 12 10 14

9 8 13 9 10 7 16 10

Сколько ходов? Введите 0 для завершения.

0

Использование функции rand() [но не rollem()] изменило бы вашу программу угадывания чисел: компьютер стал бы выбирать, а вы угадывать, вместо того чтобы сделать наоборот.

**15.11 Функция получения целых чисел: getint()**

Существует функция scanf() с форматом %d,которая берет целое число и изменяет на него. Но есть небольшой недостаток: если мы нажмем на букву случайную, а не на цифру, то не выведется нужное число, а выведется другое. Нам же нужно создать функцию, которая будет предупреждать, если мы ввели нецелое число. Функция getint будет принимать значение и проверять его:

getint(ptint)

int \*ptint; /\* указатель на целое число \*/

{

int status;

...

return(status);

}

Теперь мы должны сделать следующее:

1. Принимаем символ
2. Читаем полученную строку до конца
3. Проверяем строку и при возможности преобразуем символы в целое число

Теперь же построим программу, которая будет все это реализовывать:

#include <stdio.h>

#define LEN 81 /\* максимальная длина строки \*/

#define STOP -1 /\* коды состояний \*/

#define NONUM 1

#define YESNUM 0

getint(ptint)

int \*ptint; /\* указатель на вывод целого числа \*/

{

char intarr[LEN]; /\* запоминание вводимой строки \*/

int ch;

int ind = 0; /\* индекс массива \*/

/\* обход начальных символов «новая строка», пробелов и табуляций \*/

while ((ch = getchar()) == ' \n' || ch == " || ch == ' \t');

while (ch != EOF && ch != '\n' && ch != ' ' && ind < LEM)

{

intarr[ind++] = ch; /\* запись символа в массив \*/

ch = getchar(); /\* получение очередного символа \*/

}

intarr[ind] = ' \0'; /\* конец массива по нуль-символу \*/

if (ch == EOF)

return(STOP);

else

return ( stoi(intarr, ptint) ); /\* выполнение преобразования \*/

}

И подключим ее к main():

#define STOP -1

#define NONUM 1

#define YESNUM 0

main()

{

int num, status;

printf(" Программа прекращает считывание чисел, если

встречает EOF.\n" );

while ( (status = getint(&num)) != STOP)

if (status == YESNUM)

printf(" Число %d принято \n", num);

else

printf(" Это не целое число! Попытайтесь снова\n");

printf("Это оно.\n");

}

**15.12 Сортировка**

Сортировка используется в большинстве программах, ведь это очень востребованная функция. Нам нужно создать функцию сортировки целых чисел. Сначала же в самой программе сортировки нам нужно определить, что она будет делать:

Считывание чисел.

Сортировка чисел.

Печать отсортированных чисел.

Теперь мы имеем три подфункции, которая принимает и выводит все что нам нужно.

Теперь создадим каждую функцию по отдельности

**Считывание чисел.**

#define STOP - 1 /\* признак EOF \*/

#define NONUM 1 /\* признак нецифровой строки \*/

#define YESNUM 0 /\* признак строки цифр \*/

getarray(array, limit);

int array [], limit;

{

int num, status;

int index = 0; /\* индекс массива \*/

printf (" Эта программа прекращает считывание чисел после %d значений\n" , limit);

printf (" или если введен символ EOF.\n" );

while(index < limit && (status = getint(&num)) != STOP)

{/\* прекращает считывание после достижения limit или EOF \*/

if (status == YESNUM)

{

array [index++ ] = num;

printf(" число %d принято. \n" , num);

}

else

if (status == NONUM)

printf (" Это было не целое число! Попытайтесь снова\n" );

else

printf (" Этого не может быть! Что-то неправильно. \n" );

if (index == limit) /\* сообщить, если массив заполнен \*/

printf (" Все %d элементов массива заполнены. \n" , limit);

return (index);

}

}

**Сортировка чисел**

sortarray(array, limit)

int array[], limit;

int top, search;

{

for (top = 0; top < limit - 1; top++ )

for (search = top + 1; search < limit; search\*+ )

if (array[search] > array[top])

interchange( &array[search], &array[top] );

}

**Печать чисел**

print(array, limit)

int array[], limit;

{

int index;

for (index = 0; index <= limit; index++ )

printf(" %d\n", array[index]);}

**Результаты**

Давайте скомпилируем и протестируем нашу программу сортировки. Чтобы упростить проверку граничных условий, временно изменим MAXSIZE на 5.

В нашем первом тесте будем снабжать программу числами до тех пор, пока она не откажется их принимать.

Эта программа прекращает считывание чисел после 5 значений,

или если встретился символ EOF.

Программа считала 5 чисел и отсортировала их. Теперь посмотрим, как будет выглядеть результат, если она остановится, встретив символ EOF.

Эта программа прекращает считывание чисел после 5 значений, или если встретился символ EOF.

456 928

-23 +16

Клавиша [control z] (передает EOF в нашу систему)

928

456

16

-23

Быстрее чем вы сможете сказать «экология это наука о домашнем хозяйстве», целый огромный массив отсортирован.

Успех! Это было не просто, но не невозможно. Разделив задачу на небольшие части и продумав, какой информацией должна обмениваться каждая из них, мы свели задачу к частям, поддающимся управлению. Кроме того, отдельные модули, которые мы создали, можно использовать как части подобных программ.

Основной конспект лекций ОАП

РАЗДЕЛ 18 – СТРУКТУРЫ И ДРУГИЕ ТИПЫ ДАННЫХ.

**18. Структуры и другие типы данных**

Структуры данных очень полезны во многих сферах жизни. Основными структурами являются структуры и объединения. Основное внимание здесь уделено двум важным средствам, общим для С и C++: структурам и объединениям. Структура С или C++ аналогична массиву или вектору тесно связанных атрибутов. Однако, в отличие от массива или вектора структура позволяет иметь смешанные атрибуты различных типов данных. Это усовершенствованные дети классов. Отличие объединения от структуры в том, что объединение хранит в себе данные различных типов в одном промежутке памяти. В этом разделе будет подробно описано про каждый из этих типов.

**18.1 Структуры в С и С++**

Структурные данные нужны практически во всех сферах жизни. Начиная от расписания поездов, где нужны дата приезда, место и кол-во мест, заканчивая списком сотрудников с их данными рождения, место проживания, любимая еда, кол-во рабочих часов и т.д. Но нужно же как-то все эти данные объединить в один объект. Для этого и были придуманы структуры данных в С и С++.

**18.1.1. Структуры в С и С++: синтаксис и правила.**

Объявить структуру в С++ нужно словом struct. За ним следуют поле тега и список элементов структуры. Синтаксис структуры представлен на примере:

struct поле\_тега {

тип\_элемента элемент 1;

тип\_элемента элемент 2;

тип\_элемента элемент З;

...

...

...

тип\_элемента\_элемент N;

};

Структура всегда должна закрываться ;, т.к. оно является оператором в С++.

Полный пример с описанием элементов на примере:

struct stboat { // катер

char sztype [iSTRING15 + iNULL\_CHAR]; //тип

char szmodel[iSTRING15 + iNULL\_CHAR]; //модель

char sztitle[iSTRING20 + iNULL\_CHAR]; //бортовой номер

int iyear; // год выпуска

long int lmotor\_hours; // ресурс двигателя

float fsaleprice; // продажная стоимость

};

Теперь у нас есть переменная stboat, которая содержит в себе переменные. Но как к ней обратиться? С помощью его имени, синтаксис на примере ниже:

struct book libry;

struct stboat stused\_boat;

ЗАМЕЧАНИЕ: если структура объявлена внутри функции, то вне функции к ней обратиться уже нельзя!

**18.1.2. Создание простой структуры. Типовая задача 2 – инвентаризация книг.**

Задача: Гвен Гленн хочет напечатать опись своих книг. Она хотела бы занести в нее различную информацию о каждой книге: ее название, фамилию автора, издательство, год издания, число страниц, тираж и цену.

Построенная структура в примере ниже:

/\* инвентаризация одной книги \*/

#include <stdio.h>

#define MAXTIT 41 /\* максимальная длина названия +1 \*/

#define MAXAUT 31 /\* максимальная длина фамилии автора +1 \*/

/\* шаблон первой структуры: book является именем типа структуры \*/

struct book {

char title [MAXTIT]; /\* символьный массив для названия \*/

char author [MAXAUT]; /\* символьный массив для фамилии автора \*/

float value; /\* переменная для хранения цены книги \*/

}; /\* конец шаблона структуры \*/

void main()

{

struct book libry; /\* описание переменной типа book \*/

printf(" Enter the name of a book, please.\n");

gets(libry.title); /\* доступ к элементу title \*/

printf(" Now enter surname of the author.\n");

gets(libry.author);

printf(" And now, enter the price\n");

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

printf(" %s, %s: %.2f rub.\n", libry.title, libry.author, libry.value);

printf("%s: \" %s \" \(%.2f rub.\) \n", libry.author, libry.title, libry.value);

}

Созданная нами структура состоит из названия, фамилии автора и цены.

**18.1.3. Создание простой структуры. Типовая задача 2 – учет катеров.**

Основной код представлен ниже:

/\*Программа на С, иллюстрирующая создание структуры.

Программа запоминает информацию о вашем катере в некоторой структуре С \*/

#include <stdio.h>

#define iSTRING15 15

#define iSTRING20 20

#define iNULL\_CHAR 1

struct stboat {

char sztype [iSTRING15 + iNULL\_CHAR];

char szmodel[iSTRING15 + iNULL\_CHAR];

char sztitle[iSTRING20 + iNULL\_CHAR];

int iyear;

long int Imotor\_hours;

float fsaleprice;

} stused\_boat;

int main(void)

{

printf ("\nPlease enter the type of the boat: "); /\* Введите, пожалуйста, тип катера: \*/

gets (stused\_boat.sztype);

printf ("\nPlease enter the model of the boat: "); /\* Введите, пожалуйста, модель катера: \*/

gets(stused\_boat.szmodel);

printf("\nPlease enter the title number for the boat: "); /\* Введите, пожалуйста, бортовой номер катера: \*/

gets(stused\_boat.sztitle);

printf("\nPlease enter the model year for the boat: "); /\* Введите, пожалуйста, год выпуска модели катера: \*/

scanf("%d",&stused\_boat.iyear);

printf("\nPlease enter the current hours on "); /\* Введите, пожалуйста, количество часов, отработанных двигателем катера: \*/

printf("the motor for the boat: ");

scanf("%d",&stused\_boat.Imotor\_hours);

printf("\nPlease enter the purchase price of the boat: "); /\* Введите, покупную стоимость катера \*/

scanf("%f",&stused\_boat.fsaleprice);

printf("\n\n\n");

printf("A %d %s %s with title number #%s\n", stused\_boat.iyear, stused\_boat.sztype,

stused\_boat.szmodel,stused\_boat.sztitle); /\* Катер ... с бортовьм номером .\*/

printf("currently has %d motor hours", stused\_boat.Imotor\_hours); /\* в настоящий момент отработал ... часов \*/

printf(" and was purchased for $%8.2f\n",stused\_boat.fsaleprice); /\* и был куплен за ...\*/

return (0);

}

Результат работы программы показывает, как информация обрабатывается в структуре:

А 1952 Chris Craft with title number #CC1011771018C

currently has 34187 motor hours and was purchased for $68132.98

**18.1.4. Структуры C++: расширения синтаксиса и правил.**

С – дочка С++. Поэтому все, что работает на С будет работать на С++

Но в то же время некоторое, что будет на С++, не будет работать на С.

Но С++ имеет много усовершенствований языка С, о которых будет ниже:

/\* Объявление структуры, допустимое в С и C++ \*/

struct stboat stused\_boat;

// Объявление структуры, возможное только в C++

stboat stused\_boat;

**18.1.5. Доступ к элементам структуры**

Для обращения к структуре используют оператор точку (.). Синтаксис операции ниже:

Strict.name.

В этой записи stname является переменной, связанной со структурным типом, а mname — это имя любой переменной-члена структуры.

В С можно записать в переменную данные с помощью оператора:

gets(stused\_boat.szmodel);

В этом операторе stused\_boat — имя структуры, a szmodel — переменная-член структуры.

Доступ к элементам структуры в C++ по сути такой же:

cin >> stused\_boat.sztype;

Следующий оператор выводит на экран продажную стоимость stused\_boat

cout << stused\_boat.fsaleprice;

**18.1.6. Инициализация структуры.**

Инициализация структуры на примере ниже:

static struct book libry; //создание структурной переменной

static struct book libry = {

"Пират и девица",

"Рене Вивот",

1р.95 // Работает студенческая соображалка?

}; //инициализация

**18.1.7. Передача структур в функции.**

Структуру можно передать в функцию при помощи следующей конструкции:

fname(stvariable);

Ну а само использование полностью на примере ниже:

/\* Программа на С, показывающая передачу структуры в функцию \*/

#include <stdio.h>

#define iSTRING15 15

#define iSTRING20 20

#define iNULL\_CHAR 1

struct stboat {

char sztype [iSTRING15 + iNULL\_CHAR];

char szmodel[iSTRING15 + iNULL\_CHAR];

char sztitle[iSTRING20 + iNULL\_CHAR];

int iyear;

long int lmotor\_hours;

float fsaleprice;

};

void vprint\_data(struct stboat stany\_boat);

int main(void)

{

struct stboat stused\_boat;

printf("\nPlease enter the type of the boat: ");

gets(stused\_boat.sztype);

printf("\nPlease enter the model of the boat: ");

gets(stused\_boat.szmodel);

printf("\nPlease enter the title number for the boat: ");

gets(stused\_boat.sztitle);

printf("\nPlease enter the model year for the boat: ");

scanf("%d",&stused\_boat.iyear);

printf("\nPlease enter the current hours on ");

printf("the motor for the boat: ");

scanf("%ld",&stused\_boat.lmotor\_hours);

printf("\nPlease enter the purchase price of the boat: ");

scanf("%f",&stused\_boat.fsaleprice);

vprint\_data(stused\_boat);

return (0);

}

void vprint\_data(struct stboat stany\_boat)

{

printf("\n\n");

printf("A %d %s %s with title number #%s\n",stany\_boat.iyear,

stany\_boat.sztype,stany\_boat.szmodel,stany\_boat.sztitle);

printf("currently has %ld motor hours", stany\_boat.lmotor\_hours);

printf(" and was purchased for $%8.2f\n", stany\_boat.fsaleprice);

}

**18.2. Массив структур.**

Массив структур используется, если нам нужно создать отдельные структуры для определенных, допустим, людей. Если у нас 20 человек, то нужно создать 20 структур. Тут нам и поможет массив структур. Инициализация массива структур на примере ниже:

HOUSE mDistr[30];

for(int i=0; i<30; i++)

itoa(i+l,mDistr[i], HouseNum,10);

for(int i=0; i<30; i++)

cout << mDistr[i].HouseNum << ' \n ';

Доступ к элементам совершается по индексу и имени элемента. В нашем случае mDistr[i].HouseNum – mDistr – имя структуры, i –номер структуры, HouseNum - имя элемента структуры

**18.2.1. Описание массива структур.**

Описание структуры ниже:

struct book libry [MAXBKS];

Этот оператор объявляет libry массивом, состоящим из MAXBKS-элементов. Каждый элемент массива представляет собой структуру типа book. Таким образом, libry[0] является book-структурой, libry[1] — второй book-структурой и т. д. Имя libry само по себе не является именем структуры; это имя массива, содержащего структуры.

**18.2.2. Определение элементов массива структур.**

Определение работает так: мы указываем имя структуры в массиве, потом индекс структуры, после чего указываем ииям элемента структуры. Пример ниже:

libry [0].value value — первый элемент массива

libry [4].title title — пятый элемент массива

ЗАМЕЧАНИЕ! НЕ ОШИБАЙТЕСЬ:

libry.value[2] /\* неправильно \*/

libry[2].value /\* правильно \*/

Так же можно создавать массивы внутри структур. Пример ниже:

libry[2].title[4]

Это был бы пятый элемент элемента title (т. е. title [4]) структуры типа book, описанный третьей структурой (т. е. libry[2]). Это означает, что индексы, находящиеся справа от операции ".", относятся к отдельным элементам, в то время как индексы, расположенные слева от операции, относятся к массивам структур.

**18.2.3. Масив структур. Типовая задача 2.**

**Задача: структура, содержащая** катера не дороже 45000 долларов, и все катера, имеющие одну каюту,

Рабочая структура на примере ниже:

/\* Программа на С, использующая массив структур.

В этом примере создается "реестр подержанных катеров" для фирмы

Nineveh Boat Sales. \*/

#include <stdio.h>

#define iSTRING15 15

#define iSTRING20 20

#define iNULL\_CHAR 1

#define iMAX\_BOATS 50

struct stboat {

char sztype [iSTRING15 + iNULL\_CHAR];

char szmodel[iSTRING15 + iNULL\_CHAR];

char sztitle[iSTRING20 + iNULL\_CHAR];

char szcomment [80];

int iyear;

long int lmotor\_hours;

float fretail;

float fwholesale;

};

int main(void)

{

int i, iinstock;

struct stboat astNineveh[iMAX\_BOATS];

printf("How many boats in inventory? "); /\* Сколько катеров в реестре \*/

scanf("%d",&iinstock);

for (i=0; i<iinstock; i++) {

flushall(); /\* очистить буфер клавиатуры \*/

printf("\nPlease enter the type of the boat: ");

gets(astNineveh[i].sztype);

printf("\nPlease enter the model of the boat: ");

gets(astNineveh[i].szmodel);

printf("\nPlease enter the title number for the boat: ");

gets(astNineveh[i].sztitle);

printf("\nPlease enter a one line comment about the boat: "); /\* Введите, пожалуйста, строку комментария: \*/

gets(astNineveh[i].szcomment);

printf("\nPlease enter the model year for the boat: ");

scanf("%d",&astNineveh[i].iyear);

printf("\nPlease enter the current hours on ");

printf("the motor for the boat: ");

scanf("%ld",&astNineveh[i].lmotor\_hours);

printf("\nPlease enter the retail price of the boat :"); /\* Введите, пожалуйста, розничную цену катера: \*/

scanf("%f",&astNineveh[i].fretail);

printf("\nPlease enter the wholesale price of the boat :"); /\* Введите, пожалуйста, оптовую цену катера: \*/

scanf("%f",&astNineveh[i].fwholesale);

}

printf("\n\n\n");

for (i=0; i<iinstock; i++) {

printf("A %d %s %s beauty with %1d low hours.\n",

astNineveh[i].iyear,astNineveh[i].sztype, astNineveh[i].szmodel,

astNineveh[i].lmotor\_hours);

printf("%s\n", astNineveh[i].szcomment);

printf ("Grab the deal by asking your Nineveh salesperson for\n");

/\* Поторопитесь связаться с дилером фирмы Nineveh насчет \*/

printf (" #%s ONLY! $%8.2f.\n",astNineveh[i] .sztitle, astNineveh[i].fretail);

printf("\n\n");

}

return (0);

}

Первый элемент массива имеет нулевой индекс. Следовательно, к информации о первом катере, хранящейся в массиве структур, можно обратиться при помощи такого оператора:

gets(astNineveh[0].sztitle);

Изучая программу, обратите внимание на то, что доступ к элементам массива осуществляется в цикле. В этом случае для доступа к элементам нужно написать следующее:

gets(astNineveh[i].sztitle);

**18.3. Вложенные структуры.**

Вложенные структуры – структуры, которые содержат в себе другие структуры. Это очень удобно, когда нам нужно создать структуру, каждая из которой будет содержать свои структуры и можно будет обращаться к элементам, обращаясь к разным структурам и не путаться в именах.

Пример вложенной структуры:

#include <stdio.h>

#define LEN 20

#define M1 " Thank you for a beautiful evening,"

#define M2 " You, certainly, right, that"

#define M3 "-original guy. We should get together"

#define M4 " to taste very good"

#define M5 " and enjoy a little bit."

struct names { /\*первый структурный шаблон \*/

char first[LEN];

char last[LEN];

};

struct guy

{ /\* второй шаблон \*/

struct names handle; /\* вложенная структура \*/

char favfood[LEN];

char job[LEN];

float income;

};

void main()

{

static struct guy fellow = { /\* инициализация переменной \*/

{"Franco", "Otel"},

" eggplant",

" knitter of doormat",

15435.00

};

printf (" Dear %s, \n \n", fellow.handle.first);

printf (" %s %s.\n", M1, fellow.handle.first);

printf (" %s %s\n", M2, fellow.job);

printf (" %s\n", M3);

printf (" %s%s%s\n\n", M4, fellow.favfood, M5);

printf (" %40s%s\n" , " " , " See you later," );

printf (" %40s%s\n", " " , "Shalala");

}

struct names handle;

Это означает, что handle является переменной типа struct names. Конечно, файл должен также содержать шаблон для структуры типа names.

**18.3.1. Структуры внутри структур. Типовая задача 2.**

Задача: создать структуру, содержащую переменные структурного типа.

Решение в коде ниже:

struct strepair { // ремонт

int ioilchange; // замененное масло

int iplugs; // втулки

int lairfilter; // воздушный фильтр

int ibarnacle\_cleaning; // ветошь для чистки

};

Структуру strepair можно включить в главную структуру следующим образом:

struct stboat {

char sztype [iSTRING15 + iNULL\_CHAR];

char szmodel[iSTRING15 + iNULL\_CHAR];

char sztitle[iSTRING20 + iNULL\_CHAR];

char szcomment[80];

struct strepair strepair\_record;

int iyear;

long int lmotor\_hours;

float fretail;

float fwholesale;

} astNineveh[iMAX\_BOATS] ;

Для доступа к некоторому элементу структуры strepair\_record можно использовать такой оператор:

printf("%d\n",astNineveh[0].strepair\_record.ibarnacle\_cleaning);

**18.4. Использование указателей на структуры.**

Да-да , указатели и тут) Тут есть бонусы при использовании их:

1. точно так же как указатели на массивы, они легче в использовании

(скажем, в задаче сортировки), чем сами массивы, а указателями на структуры легче пользоваться, чем самими структурами.

2) структура не может использоваться в качестве аргумента функции, а указатель на структуру может.

3) многие удобные представления данных являются структурами, содержащими указатели к другим структурам

**18.4.1. Описание и инициализация указателя на структуру.**

Описание указателя на структуру на прмиере ниже:

struct guy \*him;

Первым стоит ключевое слово struct, затем слово guy, являющееся именем структурного шаблона, далее \* и за нею имя указателя.

Мы можем создать указатель him для ссылок на любые структуры типа guy. Мы инициализируем him, который будет ссылаться на fellow[0]; тут же мы используем операцию получения адреса:

him = &fellow[0];

ЗАМЕЧАНИЕ! him ссылается на fellow [0], a him + 1 — на fellow[1]. Заметим, что добавление 1 к him прибавляет 84 к адресу. Это происходит потому, что каждая структура занимает 84 байта памяти

**18.4.2. Доступ к элементу структуры при помощи указателя.**

Мы можем использовать новую операцию - >. Она заключается в вводе дефиса (-) и следующего за ним символа (>). Пример ниже:

him - > income — это fellow[0].income, если him = &fellow[0]

Структурный указатель, за которым следует операция - >, работает так же, как имя структуры с последующей операцией ".". (Мы не можем сказать him.income, потому что him не является именем структуры.)

ЗАПОМНИ - him — указатель, а him - > income — элемент структуры, на которую делается ссылка. Таким образом, в этом случае him - > income является переменной типа float.

Полный пример ниже:

/\* указатель на структуру \*/

#include<stdio.h>

#define LEN 20

struct names {

char first [LEN];

char last [LEN];

};

struct guy {

struct names handle;

char favfood [LEN];

char job [LEN];

float income;

};

void main()

{

static struct guy fellow [2] = {

{

{"Franco", "Otel"},

"eggplant",

"knitter of doormat",

15435.00

},

{

{"Rodney", "Svilbely"},

"salmon mousse",

"interior decorator",

35000.00

}

};

struct guy \*him; /\* ЭТО------указатель на структуру \*/

printf ("address #1: %u #2: %u\n", &fellow[0], &fellow[1]);

him = &fellow [0]; /\* сообщает указателю, на что ссылаться \*/

printf ("pointer #1: %u #2: %u\n", him, him+1);

printf ("him->income $%.2f: (\*him).income $%.2f\n", him->income, (\*him).income);

him++; /\* указывает на следующую структуру \*/

printf ("him->favfood is %s: him->names.last is %s\n",him->favfood, him->handle.last);

}

**18.4.3. Использование указателя. Типовая задача 2:**

Выполненный код в примере ниже:

/\* Программа на С, использующая указатель на массив структур.

Опять взят пример реестр катеров фирмы Nineveh Boat Sales \*/

#include <stdio.h>

#define iSTRING15 15

#define iSTRING20 20

#define iNULL\_CHAR 1

#define iMAX\_BOATS 50

struct stboat {

char sztype [iSTRING15 + iNULL\_CHAR];

char szmodel[iSTRING15 + iNULL\_CHAR];

char sztitle[iSTRING20 + iNULL\_CHAR];

char szcomment[80];

int iyear;

long int lmotor\_hours ;

float fretail;

float fwholesale;

};

int main(void)

{

int i, iinstock;

struct stboat astNineveh[iMAX\_BOATS], \*pastNineveh;

pastNineveh=&astNineveh[0];

printf("How many boats in inventory? ");

scanf("%d",&iinstock) ;

for (i=0; i<iinstock; i++) {

flushall(); /\* очистить буфер клавиатуры \*/

printf("\nPlease enter the make of the boat: ");

gets(pastNineveh->sztype);

printf("\nPlease enter the model of the boat: ");

gets(pastNineveh->szmodel);

printf("\nPlease enter the title number for the boat: ");

gets(pastNineveh->sztitle);

printf("\nPlease enter a one line comment about the boat: ");

gets(pastNineveh->szcomment);

printf("\nPlease enter the model year for the boat: ");

scanf("%d",&pastNineveh->iyear);

printf("\nPlease enter the current hours on ");

printf("the motor for the boat: ");

scanf("%d",&pastNineveh->lmotor\_hours);

printf("\nPlease enter the retail price of the boat: ");

scanf("%f",&pastNineveh->fretail);

printf("\nPlease enter the wholesale price of the boat: ");

scanf("%f",&pastNineveh->fwholesale);

pastNineveh++;

}

pastNineveh =&astNineveh[0];

printf("\n\n\n") ;

for (i=0; i < iinstock; i++) {

printf("A %d %s %s beauty with %d low hours.\n",pastNineveh->iyear,

pastNineveh->sztype,pastNineveh->szmodel,pastNineveh->lmotor\_hours);

printf("%s\n",pastNineveh->szcomment);

printf("Grab the deal by asking your Nineveh salesperson for:");

printf("\n#%s ONLY! $ %.2f \n",pastNineveh->sztitle, pastNineveh->fretail);

printf("\n\n");

pastNineveh++;

}

return (0);

}

**18.5. Структуры и функции.**

**18.5.1. Передача структур по значению.**

Если нам нужно передать структуру в тело функции, том мы можем их передавать в качестве аргумента в функцию:

struct ALLNUMB

{

int nVar;

long lVar;

short shVar;

unsigned int uiVar;

}

void Punc (ALLNUMB);

Тут происходит передача в тело функции параметра типа структуры по значению.

ЗАМЕЧАНИЕ: оригинал структурированного объекта модификации не подлежит.

Полное использование на примере ниже:

#include <iostream>

using namespace std;

struct HOUSE

{

unsigned short RegNum;

char Street[51]; // с учетом '\0'

char HouseNum[6];

unsigned short MaxFloorNum;

unsigned short MaxFlatNum;

bool Parking;

};

void OutAddress(HOUSE);

int main ()

{

setlocale(LC\_ALL, "Rus");

HOUSE MyHouse;

MyHouse.RegNum = 524;

strcpy(MyHouse.Street, "ул. Гоголя");

strcpy(MyHouse.HouseNum, "2-a");

MyHouse.MaxFloorNum = 7;

MyHouse.MaxFlatNum = 84;

MyHouse.Parking = true;

OutAddress(MyHouse);

getchar(); getchar();

return 0;

}

void OutAddress(HOUSE house)

{

cout << house.Street << ",";

cout << house.HouseNum << '\n';

}

**18.5.2. Передача посредством указателей.**

Структуры тоже можно передавать с помощью указателей. Для этого должна быть объявлена переменная типа указателя на структуру, синтаксис которой может быть представлен в виде:

тип структуры\* идентификатор\_указателя;

Доступ к элементам структуры через указатель осуществляется с использованием символа стрелки (->) Пример ниже:

HOUSE \*pHouse;

pHouse = &MyHouse;

pHouse->Parking = true;

cout << MyHouse.Parking;

Мы должны применять операцию & для получения адреса структуры. В отличие от имени массива имя структуры само по себе не является синонимом своего адреса.

**18.5.3. Передача структур посредством ссылок.**

Так же можно вместо указателей использовать ссылки на структуры. Описание такой функции ниже:

тип структуры &имя\_ссылки = имя\_переменной;

Ссылки и указатели на структуры данных могут быть переданы в качестве аргументов в тело функции.

СОВЕТ БЫВАЛОГО:

Используйте ссылки, ибо значительно снижается время, за которое данный параметр передается в функцию, а также экономится стековая память.

Полный синтаксис:

// Пример передачи указателя и ссылки на

// целочисленную переменную:

bool Func(int\* ptr, int& ref);

// Передача указателя и ссылки на

// структуру типа HOUSE:

char Func2 (HOUSE\* pMh, HOUSES rMh);

**18.5.4. Передача элементов структуры.**

Элемент структуры является переменной с единственным значением, значит он может быть передан как аргумент функции. Простая программа финансовых расчетов, которая прибавляет взнос клиента к его счету на примере ниже:

/\* передача элементов структуры как аргументов функции \*/

#include<stdio.h>

struct funds {

char \*bank;

float bankfund;

char \*save;

float savefund; } stan = {

"Senior tomatoes Bank",

1023.43,

"Savings and lendings of Snoopy",

4239.87

};

float sum(float x, float y);

void main()

{

float total;

extern struct funds stan; /\* необязательное описание \*/

total=sum(stan.bankfund, stan.savefund);

printf ("Stan has only %.2f dollars.\n", total);

}

/\* складывает два числа типа float \*/

float sum(float x, float y)

{

return x+y;

}

**18.5.5. Передача массива структур.**

Имя массива является синонимом его адреса, поэтому его можно передать функции. Но функции будет необходим доступ к структурному шаблону. Чтобы показать, как такая программа работает, мы расширим нашу программу таким образом, чтобы она обслуживала двух человек, а мы, следовательно, имели бы массив двух структур funds.

/\* передача массива структур в функцию \*/

#include<stdio.h>

struct funds {

char \*bank;

float bankfund;

char \*save;

float savefund; } jones[2] ={

{

"Senior tomatoes Bank",

1023.43,

"Savings and lendings of Snoopy",

4239.87

},

{

"Honest Jack's Bank",

976.57,

"Accumulation by prior plan",

1760.13

}

};

float sum(struct funds \*);

void main()

{

float total;

total=sum(jones);

printf ("Jonsons have only %.2f dollars.\n" , total);

}

float sum (struct funds \*money)

{

float total;

int i;

for( i = 0, total = 0; i < 2; i++ , money++)

total+= money->bankfund + money->savefund;

return(total);

}

ЗАМЕЧАНИЕ:

1. Имя массива можно использовать для передачи в функцию указателя на первую структуру в массиве.

2. Затем можно использовать арифметическую операцию над указателем, чтобы передвигать его на последующие структуры в массиве. Заметим, что вызов функции

**18.5.5.1. Передача информации о структурах функциям. Использование массива. Типовая задача 2 на С.**

Главное преимущество от использования указателей при передаче массива структур в функцию - массив вызывается по имени или по ссылке. Функция обращается к исходному массиву, а не к его копии.

Выполненная задача ниже:

/\* Программа на С, иллюстрирующая обращение функции к массиву

структур с использованием указателя

Опять используется реестр катеров фирмы Nineveh Boat Sales! \*/

#include <stdio.h>

#define iSTRING15 15

#define iSTRING20 20

#define iNULL\_CHAR 1

#define iMAX\_BOATS 50

int iinstock;

struct stboat {

char sztype [iSTRING15 + iNULL\_CHAR];

char szmodel[iSTRING15 + iNULL\_CHAR];

char sztitle[iSTRING20 + iNULL\_CHAR];

char szcomment[80];

int iyear;

long int lmotor\_hours;

float fretail;

float fwholesale;

};

void vprint\_data(struct stboat \*stany\_boatptr);

int main(void)

{

int i;

struct stboat astNineveh[iMAX\_BOATS], \*pastNineveh;

pastNineveh=&astNineveh[0];

printf("How many boats in inventory?\n");

scanf("%d",&iinstock);

for (i=0; i < iinstock; i++) {

flushall(); /\* очистить буфер клавиатуры \*/

printf("\nPlease enter the make of the boat: ");

gets(pastNineveh->sztype);

printf("\nPlease enter the model of the boat: ");

gets(pastNineveh->szmodel);

printf("\nPlease enter the title number for the boat: ");

gets(pastNineveh->sztitle);

printf("\nPlease enter a one line comment about the boat: ");

gets(pastNineveh->szcomment);

printf("\nPlease enter the model year for the boat: ");

scanf("%d",&pastNineveh->iyear);

printf("\nPlease enter the current hours on ");

printf("the motor for the boat: ");

scanf("%d",&pastNineveh->lmotor\_hours);

printf("\nPlease enter the retail price of the boat: ");

scanf("%f",&pastNineveh->fretail);

printf("\nPlease enter the wholesale price of the boat: ");

scanf("%f",&pastNineveh->fwholesale);

pastNineveh++;

}

pastNineveh = &astNineveh[0];

vprint\_data(pastNineveh);

return (0);

}

void vprint\_data(struct stboat \*stany\_boatptr)

{

int i;

printf("\n\n\n");

for (i=0; i < iinstock; i++) {

printf("A %d %s %s beauty with %ld low hours.\n", stany\_boatptr->iyear,

stany\_boatptr->sztype, stany\_boatptr->szmodel, stany\_boatptr->lmotor\_hours);

printf ("%s\n",stany\_boatptr->szcomment);

printf ("Grab the deal by asking your Nineveh salesperson for\n");

printf ("#%s ONLY! $%8.2f.\n",stany\_boatptr->sztitle, stany\_boatptr->fretail);

printf("\n\n");

stany\_boatptr++;

}

}

**18.5.5.2. Передача информации о структурах функциям. Использование массива. Типовая задача 2 на C++.**

Пример ниже:

// Программа на C++, иллюстрирующая использование указателей при

// обращении к данным в структуре из функции.

// Примечание: Строка комментария заканчивается точкой (.)

#include <iostream.h>

#define iSTRING15 15

#define iSTRING20 20

#define iNULL\_CHAR 1

#define iMAX\_BOATS 50

int iinstock;

struct stboat {

char sztype [iSTRING15 + iNULL\_CHAR];

char szmodel[iSTRING15 + iNULL\_CHAR];

char sztitle[iSTRING20 + iNULL\_CHAR];

char szcomment[80];

int iyear;

long int lmotor\_hours;

float fretail;

float fwholesale;

};

void vprint\_data(stboat \*stany\_boatptr);

int main(void)

{

int i;

char newline;

stboat astNineveh[iMAX\_BOATS],\*pastNineveh;

pastNineveh=&astNineveh[0];

cout << "How many boats in inventory? ";

cin >> iinstock;

for (i=0; i<iinstock; i++) {

cout << "\nPlease enter the make of the boat: ";

cin >> pastNineveh->sztype;

cout << "\nPlease enter the model of the boat: ";

cin >> pastNineveh->szmodel;

cout << "\n Please enter the title number for the boat: ";

cin >> pastNineveh->sztitle;

cout << "\nPlease enter the model year for the boat: ";

cin >> pastNineveh->iyear;

cout << "\nPlease enter the current hours on "<< "the motor for the boat: ";

cin >> pastNineveh->lmotor\_hours;

cout << "\nPlease enter the retail price of the boat: ";

cin >> pastNineveh->fretail;

cout << "\nPlease enter the wholesale price of the boat: ";

cin >> pastNineveh->fwholesale;

cout << "\nPlease enter a one line comment about the boat: ";

cin.get(newline); // обрабатывается возврат каретки

cin.get(pastNineveh->szcomment,80,'.');

cin.get(newline); // обрабатывается возврат каретки pastNineveh++;

}

pastNineveh=&astNineveh[0];

vprint\_data(pastNineveh);

return (0);

}

void vprint\_data(stboat \*stany\_boatptr)

{

int i;

cout << "\n\n\n";

for (i=0; i<iinstock; i++) {

cout << "A[{|"<< stany\_boatptr->iyear <<"|}] [{|"<< stany\_boatptr->sztype<<"|}] [{I"

<< stany\_boatptr->szmodel <<"I}] beauty with [{|"<< stany\_boatptr->lmotor\_hours <<"|}] low hours.\n";

cout << stany\_boatptr->szcomment << endl;

cout << "Grab the deal by asking your Nineveh "<< "salesperson for #";

cout << stany\_boatptr->sztitle << "ONLY! $"<< stany\_boatptr->fretail << "\n\n";

stany\_boatptr++;

}

}

**18.5.6. Функция – член структуры**

Для того чтобы функция стала членом структуры, достаточно поместить ее описание внутрь описания структуры. Пример ниже:

struct \_3d {

double x, y, z;

double mod ();

} vect;

Здесь функция mod () выступает как член структуры.

Для описания функции надо после типа возвращаемого значения указать имя шаблона структуры и имя функции, отделив их знаком :: . Пример.

double \_3d::mod()

{

return sqrt (x \* x + y \* y + z \* z);

}

Преимущество такой структуры - функция обращается к своим переменным по имени. Если функция не использует посторонние данные для достижения результата и не возвращает результат через список параметров, то список параметров может быть пустым.

**18.5.6.1. Особенность возврата const.**

Важно помнить!

int\* - pointer на int

int const \* - pointer на const int

int \* const - const pointer на int

int const \* const - const pointer на const int

Значения const:

1. Постоянная ссылка

int var0 = 0;

const int &ptr1 = var0;

ptr1 = 8; // Error

var0 = 6; // OK

1. Константные указатели

int var1 = 1;

int var2 = 0;

int \*const ptr2 = &var1;

ptr2 = &var2; // Error

1. Указатель на константу

int const \* ptr3 = &var2;

\*ptr3 = 4; // Error

1. Постоянный указатель на константу

int var3 = 0;

int var4 = 0;

const int \* const ptr4 = &var3;

\*ptr4 = 1; // Error

ptr4 = &var4; // Error

**18.5.7. Возврат структуры из функции**

Когда мы передаем структуру в функцию, она копируется, так же как и при возвращении структуры. Поэтому любые изменения сделанные внутри функции будут потеряны, мы возвращаем структуру после изменения.

Основной пример ниже:

#include <stdio.h>

#include <iostream>

struct EnemySpaceShip {

int x\_coordinate;

int y\_coordinate;

int weapon\_power;

};

EnemySpaceShip getNewEnemy() {

EnemySpaceShip ship;

ship.x\_coordinate = 0;

ship.y\_coordinate = 0;

ship.weapon\_power = 20;

return ship;

}

EnemySpaceShip upgradeWeapons(EnemySpaceShip ship) {

ship.weapon\_power += 10;

return ship;

}

int main() {

EnemySpaceShip enemy = getNewEnemy();

printf("%d\n", enemy.x\_coordinate);

printf("%d\n", enemy.y\_coordinate);

printf("%d\n\n", enemy.weapon\_power);

enemy = upgradeWeapons(enemy);

printf("%d\n", enemy.x\_coordinate);

printf("%d\n", enemy.y\_coordinate);

printf("%d\n", enemy.weapon\_power);

system("PAUSE");

return 0;

}

**18.6. Структуры и битовые поля.**

Битовые поля в структурах обычно используются в низкоуровневом программировании, когда работа идет со значениями, способными занимать не байты, а отдельные биты

Во время создания полей структуры можно указывать размер полей. Минимальный размер поля структуры – один бит. Общий синтаксис:

struct имя{

тип\_поля1 имя\_поля1: размер1;

тип\_поля2 имя\_поля2: размер2;

...

тип\_поляN имя\_поляN: размерN;

};

ЗАМЕЧАНИЕ: для части полей можно указывать размер, а для других – нет. Если поле имеет размер в один бит, в качестве его типа указывается unsigned.

**18.7. Структуры. Их дальнейшее использование.**

Разработчики придумывают усовершенствования структур. К ним относятся очереди, двоичные деревья, неупорядоченные массивы, рандомизированные таблицы и графы. Все из них будут в будущих разделах.

**18.8. Объединения.**

Объединение — структура данных в программировании, члены которой расположены по одному и тому же адресу. В отдельный момент времени объединение может хранить информацию только об одном типе данных.

**18.8.1. Объединения: синтаксис и правила.**

Объединение создается при помощи ключевого слова union и с использованием следующего синтаксиса:

union поле\_тега {

тип поле1;

тип поле2;

тип поле3;

...

…

…

тип полеN;

};

ЗАМЕЧАНИЕ: в отличие от структур, переменная типа объединения может быть проинициализирована только значением первого объявленного члена:

union MyData

{

int iVar1;

unsigned long ulFreq;

char Symb[10];

} myX = {25};

**18.8.2. Создание простого объединения.**

ЗАМЕЧАНИЕ: объединения занимают память, необходимую для самого большого типа данных, хранящегося в них. Все другие типы данных в объединении используют совместно часть этой памяти или всю эту память.

Создание простого объединения в примере ниже:

// Программа на C++, иллюстрирующая использование объединения.

// Создается объединение, содержащее несколько типов данных

#include <iostream.h>

union unmany\_types {

char c;

int ivalue;

float fvalue;

double dvalue, j;

} unmy\_union;

int main(void)

{

// valid I/O

unmy\_union.c='b';

cout << unmy\_union.c << "\n";

unmy\_union.ivalue=1990;

cout << unmy\_union.ivalue << "\n";

unmy\_union.fvalue=19.90;

cout << unmy\_union.fvalue << "\n";

unmy\_union.dvalue=987654.32E+13;

cout << unmy\_union.dvalue << "\n";

// неправильный ввод/вывод

cout << unmy\_union.c << "\n";

cout << unmy\_union. ivalue << "\n";

cout << unmy\_union.fvalue << "\n";

cout << unmy\_union.dvalue << "\n";

// размер объединения

cout << "The size of this union is: "<< sizeof(unmany\_types) <<" bytes." << "\n";

return (0);

}

**18.9. Вспомогательные средства.**

При правильном использовании оба средства, typedef и enum, упрощают текст программы.

**18.9.1. Использование typedef.**

Формирование пользовательских типов осуществляется с использованием ключевого слова typedef, за которым указывается какой-либо из имеющихся типов данных и следующий за ним идентификатор. Пример ниже:

typedef unsigned char byte;

Основной смысл в том, что typedef создает новый тип данных и называет его определенным именем. Пример полной программы:

/\* Программа на С, иллюстрирующая использование typedef.

Создаются два новых типа, "whole" и "real",

которые можно использовать вместо "int" и "float" \*/

#include <stdio.h>

typedef int whole;

typedef float real;

int main(void)

{

whole wvalue=123;

real rvalue=5.6789;

/\* Целое число \*/

printf("The whole number is %d.\n",wvalue);

/\* Вещественное число \*/

printf("The real number is %.4f.\n",rvalue);

return (0);

}

ЗАМЕЧАНИЕ! не используйте слишком много новых типов; это может вызвать обратный эффект и ухудшить читаемость программы и усложнить ее

**18.9.2. Использование enum.**

Переменная типа с плавающей точкой – это переменная, которая может содержать действительное число, десятичная точка может «плавать»; то есть она может поддерживать переменное количество цифр до и после себя.

Enum(enumeration) – перечисляемый тип данных.

enum weddingAnn {chintz = 1, paper, leather, linen, wooden} year;

chintz = 1 тк перечисление начинается с 0, а нам нужно с 1

year является типом переменных {chintz = 1, paper, leather, linen, wooden}

эти переменные будут являться константами

**18.9.2.1. Перечисляемые типы**

Переменная типа с плавающей точкой – это переменная, которая может содержать действительное число, десятичная точка может «плавать»; то есть она может поддерживать переменное количество цифр до и после себя.

Enum(enumeration) – перечисляемый тип данных.

enum weddingAnn {chintz = 1, paper, leather, linen, wooden} year;

chintz = 1 тк перечисление начинается с 0, а нам нужно с 1

year является типом переменных {chintz = 1, paper, leather, linen, wooden}

эти переменные будут являться константами

**18.9.2.2. Имена перечислений**

Идентификаторы перечислений начинаются с заглавной буквы, а имена перечислителей вообще состоят только из заглавных букв. Перечислители находятся в одном и том же пространстве имён, что и само перечисление, а значит одно и то же имя одного перечислителя не может быть использовано в нескольких перечислениях одного и того же пространства имён.

**18.9.2.3. Значения перечислителей**

ЗАМЕЧАНИЕ! Не присваивайте свои значения перечислителям.

Правило: Не присваивайте одинаковые значения двум перечислителям в одном перечислении, если на это нет веской причины.

Пример ниже:

// Определяем новый перечисляемый тип Animals

enum Animals

{

ANIMAL\_PIG = -4,

ANIMAL\_LION, // присваивается -3

ANIMAL\_CAT, // присваивается -2

ANIMAL\_HORSE = 6,

ANIMAL\_ZEBRA = 6, // имеет то же значение, что и ANIMAL\_HORSE

ANIMAL\_COW // присваивается 7

};

**18.9.2.4. Обработка перечислений**

Пример обработки ниже:

#include <iostream>

using namespace std;

enum Colors

{

COLOR\_PURPLE, // присваивается 0

COLOR\_GRAY, // присваивается 1

COLOR\_BLUE, // присваивается 2

COLOR\_GREEN, // присваивается 3

COLOR\_BROWN, // присваивается 4

COLOR\_PINK, // присваивается 5

COLOR\_YELLOW, // присваивается 6

COLOR\_MAGENTA // присваивается 7

};

void printColor(Colors color);

int main()

{

setlocale(LC\_ALL, "Rus");

//Colors color;

//std::cin >> color; // приведёт к ошибке компиляции

int inputColor;

std::cin >> inputColor;

Colors color = static\_cast<Colors>(inputColor);

std::cout << color; // выводится на экран inputColor

std::cin >> inputColor;

system("pause>>void");

return 0;

}

**18.9.2.5. Вывод перечислителей**

Пример кода ниже:

#include <iostream>

using namespace std;

enum Colors

{

COLOR\_PURPLE, // присваивается 0

COLOR\_GRAY, // присваивается 1

COLOR\_BLUE, // присваивается 2

COLOR\_GREEN, // присваивается 3

COLOR\_BROWN, // присваивается 4

COLOR\_PINK, // присваивается 5

COLOR\_YELLOW, // присваивается 6

COLOR\_MAGENTA // присваивается 7

};

void printColor(Colors color);

int main()

{

setlocale(LC\_ALL, "Rus");

//Colors color;

//std::cin >> color; // приведёт к ошибке компиляции

int inputColor;

std::cin >> inputColor;

Colors color = static\_cast<Colors>(inputColor);

printColor(color);

system("pause>>void");

return 0;

}

void printColor(Colors color)

{

if (color == COLOR\_PURPLE)

std::cout << "Purple";

else if (color == COLOR\_GRAY)

std::cout << "Gray";

else if (color == COLOR\_BLUE)

std::cout << "Blue";

else if (color == COLOR\_GREEN)

std::cout << "Green";

else if (color == COLOR\_BROWN)

std::cout << "Brown";

else if (color == COLOR\_PINK)

std::cout << "Pink";

else if (color == COLOR\_YELLOW)

std::cout << "Yellow";

else if (color == COLOR\_MAGENTA)

std::cout << "Magenta";

else

std::cout << "Who knows!";

}

**18.9.2.6. Выделение памяти для перечислений**

Поскольку перечисляемые типы воспринимаются как целочисленные, то компилятор сам выбирает кол-во памяти для выделения.

**18.9.2.7. Польза от перечислений**

Перечисляемые типы лучше всего использовать при определении набора связанных идентификаторов.

enum имеет единственное назначение — сделать текст программы более читаемым. В других компьютерных языках этот тип называется пользовательским, есть определяемым пользователем. Объявление перечисляемого типа выглядит так:

enum поле\_тега { значение1,. . .значениеN } переменная;

Полный код реализации ниже:

/\* Программа на С, иллюстрирующая использование перечисляемых типов.

Программа определяет количество прошедших и остающихся месяцев

в году при помощи типа enum \*/

#include <stdio.h>

enum emonths {

January=1,

February,

March,

April,

May,

June,

July,

August,

September,

October,

November,

December

} emcompleted;

int main(void)

{

int ipresent\_month;

int isum,idiff;

/\* Введите, пожалуйста, текущий месяц (от 1 до 12): \*/

printf("\nPlease enter the present month (1 to 12): ");

scanf("%d",&ipresent\_month);

emcompleted = December;

isum = ipresent\_month;

idiff = (int)emcompleted - ipresent\_month;

/\* ... месяцев прошло, ... месяцев остается. \*/

printf("\n%d month(s) past, %d months to go.\n",isum,idiff);

return (0);

}

**18.10. Сложные формы данных.**

Язык Си позволяет использовать одновременно более одного модификатора, что дает возможность создавать множество типов:

int board[8] [8]; /\* массив массивов типа int \*/

int \*\*ptr; /\* указатель на указатель на тип int \*/

int \*risks[10]; /\* 10-элементный массив указателей на тип int \*/

int (\*wisks) [10]; /\* указатель на 10-элементный массив типа int \*/

int \*oof[3] [4]; /\* 3-элементный массив указателей на 4-элементный массив типа int \*/

int (\*uuf) [3] [4]; /\* указатель на массив 3x4 типа int \*/

ЗАПОМНИТЕ!

1. Чем ближе модификатор стоит к идентификатору, тем выше его приоритет

2. Модификаторы [] и () имеют приоритет выше, чем \*.

3. Круглые скобки используются для объединения частей выражения, имеющих самый высокий приоритет.

**18.11. Отличие stract от class**

Отличий class от struct всего два

1

Члены класса, определенного с помощью ключевого слова class, по умолчанию являются private. Члены класса, определенного с помощью ключевого слова struct или union, по умолчанию являются public.

2

При отсутствии спецификатора доступа (т.е. private/protected/public) у базового класса, базовый класс будет public если класс определен с помощью struct и private если класс определен с помощью class.

**18.12. Функции работы с датой и временем.**

Основная функция ниже:

#include <time.h>

#include <iostream>

using namespace std;

int main()

{

setlocale(LC\_ALL, "Rus");

time\_t tt;

tm \*pMyTime;

tt = time(NULL);

pMyTime = localtime(&tt);

cout << "Текущее время: ";

cout << asctime(pMyTime);

getchar(); getchar();

return 0;

}

В рассматриваемом примере используется функция time (), имеющая прототип

time\_t time(time\_t \*timer);

Возвращающая число секунд, прошедших с 00:00:00 1 января 1970 г. Параметр timer также принимает возвращаемое функцией значение.

РАЗДЕЛ 19 – ЛИНЙЕНЫЙ СПИСОК.

1. **Динамические структуры данных**

**1.1. Динамические структуры данных**

Это данные, которые меняют свои данные, размер и саму структуру во время работы. В них входят линейный список, двусвязный список, барьер, циклические списки, стек, очередь, дек, деревья, графы. О них мы будем говорить весь этот раздел.

**1.2. Линейный список**

Это список, где 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.

Пример ниже:

struct Node{

char word[40]; //область данных

int count;

Node \*next; //ссылка на следующий узел

};

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

**1.2.1. Создание элемента списка (CreateNode)**

Создание будет в примере ниже:

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

}

**1.2.2. Добавление узла**

**1.2.2.1. Добавление узла в начало списка (AddFirst)**

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.

Пример реализации ниже:

void AddFirst(PNode &Head, PNode NewNode)

{

// установить ссылку узла NewNode на голову существующего списка

NewNode->next = Head;

//установить голову списка на новый узел

Head = NewNode;

}

**1.2.2.2. Добавление узла после заданного (AddAfter)**

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.

Пример реализации ниже:

void AddAfter(PNode p, PNode NewNode)

{

//установить ссылку нового узла на узел, следующий за данным

NewNode->next = p->next;

//установить ссылку данного узла p на NewNode

p->next = NewNode;

}

**1.2.2.3. Добавление узла перед заданным (AddBefore)**

Замечание

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.

Пример реализации ниже:

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

}

**1.2.2.4. Добавление узла в конец списка (AddLast)**

Пример реализации ниже:

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);

}

**1.2.3. Проход по списку**

**Пример реализации ниже:**

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

}

**1.2.4. Поиск узла в списке (Find, FindPlace)**

Такой подход приводит к следующему алгоритму:

1. начать с головы списка;

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

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

Пример реализации ниже:

PNode Find(PNode Head, char NewWord[])

{

// начать с головы списка

PNode q = Head;

// пока текущий элемент существует (указатель – не NULL),

//проверить нужное условие и перейти к следующему элементу

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

q = q->next;

// закончить, когда найден требуемый элемент или

//все элементы списка просмотрены

return q;

}

**1.2.5. Алфавитно-частотный словарь**

Вся сборка программы представлена ниже:

#include <iostream>

#include <string.h>

using namespace std;

struct Node{

char word[40]; //область данных

int count;

Node \*next; //ссылка на следующий узел

};

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

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

}

void AddFirst(PNode &Head, PNode NewNode)

{

NewNode->next = Head;

Head = NewNode;

}

void AddAfter(PNode p, PNode NewNode)

{

NewNode->next = p->next;

p->next = NewNode;

}

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

}

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);

}

PNode Find(PNode Head, char NewWord[])

{

PNode q = Head;

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

q = q->next;

return q;

}

PNode FindPlace (PNode Head, char NewWord[])

{

PNode q = Head;

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

q = q->next;

return q;

}

void main()

{

setlocale(LC\_ALL, "Rus");

PNode Head = NULL, p, where;

FILE \*in, \*out;

char word[80];

int n;

in = fopen("input.dat", "r");

while (1){

n = fscanf(in, "%s", word); //читаем слово из файла

if(n <= 0) break;

p = Find(Head, word); //ищем слово в списке

if(p != NULL) // если нашли слово,

p->count ++; // увеличить счетчик

else{

p = CreateNode(word); // создаем новый узел

where = FindPlace(Head, word); // ищем место

if(!where)

AddLast(Head, p);

else

AddBefore(Head, where, p);

}

}

fclose(in);

out = fopen("output.dat", "w");

p = Head;

while(p) { // проход по списку и вывод результатов

fprintf(out, "%-20s\t%d\n", p->word, p->count);

p = p->next;

}

fclose(out);

system("PAUSE");

}

**1.2.6. Удаление узла (DeleteNode)**

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

Реализация ниже:

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

}

**1.3. Барьеры**

Барьер – это механизм синхронизации, который позволяет приостановить выполнение потоков у некоторой точки программы до тех пор, пока все потоки не дойдут до этого места, и только затем продолжить выполнение дальше.

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.

**1.4. Двусвязный список**

Это такой список, где ля доступа к списку используется не одна переменная-указатель, а две – ссылка на «голову» списка и на «хвост» - последний элемент.

Пример реализации ниже:

struct Node

{

char word[40]; //область данных

int count;

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

};

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

**1.4.1. Операции с двусвязным списком**

**1.4.1.1. Добавление узла в начало списка**

**ЗАПОМНИ**

При добавлении нового узла NewNode в начало списка надо

1. 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;

2. установить голову списка на новый узел;

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

Пример кода ниже:

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

{

// установить ссылку next узла NewNode на голову существующего списка

NewNode->next = Head;

// установить ссылку prev в NULL

NewNode->prev = NULL;

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

if(Head)

Head->prev = NewNode;

// установить голову списка на новый узел

Head = NewNode;

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

if(! Tail)

Tail = Head; //этот элемент – первый

}

**1.4.1.2. Добавление узла в конец списка**

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

**1.4.1.3. Добавление узла после заданного**

Два этапа:

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

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

Пример кода ниже:

void AddAfter (PNode &Head, PNode &Tail,

PNode p, PNode NewNode)

{

if(! p->next)

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

else {

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

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

NewNode->prev = p;

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

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

p->next = NewNode;

}

}

**1.4.1.4. Поиск узла в списке**

Проход по двусвязному списку может выполняться в двух направлениях – от головы к хвосту или от хвоста к голове.

**1.4.1.5. Удаление узла**

Код представлен ниже:

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;

}

**1.5. Циклические списки**

Циклический (кольцевой) список – это структура данных, представляющая собой последовательность элементов, последний элемент которой содержит указатель на первый элемент списка, а первый (в случае двунаправленного списка) – на последний.

**1.6. Стек**

Стек – это упорядоченный набор элементов, в котором добавление новых и удаление существующих элементов допустимо только с одного конца, который называется вершиной стека. Он работает методом LIFO.

В современных компьютерах стек используется для

• размещения локальных переменных;

• размещения параметров процедуры или функции;

• сохранения адреса возврата (по какому адресу надо вернуться из процедуры);

• временного хранения данных, особенно при программировании на Ассемблере.

**1.6.1. Реализация стека с помощью массива**

**Реализация представлена ниже:**

#include <iostream>

#include <string.h>

using namespace std;

const int MAXSIZE = 100;

struct Stack

{

char data[MAXSIZE];

int size;

};

void Push(Stack &S, char x)

{

if(S.size == MAXSIZE){

printf("Стек переполнен");

return;

}

S.data[S.size] = x;

S.size ++;

}

char Pop(Stack &S)

{

if(S.size == 0){

printf("Стек пуст");

return char(255);

}

S.size --;

return S.data[S.size];

}

void main()

{

setlocale(LC\_ALL, "Rus");

char br1[3] = { '(', '[', '{' }; // открывающие скобки

char br2[3] = { ')', ']', '}' }; // закрывающие скобки

char s[80], upper;

int i, k, OK;

Stack S; //стек символов

printf("Введите выражение со скобками> ");

gets(s);

S.size = 0; //сначала стек пуст

OK = 1;

for(i = 0; OK && (s[i] != '\0'); i ++)

for(k = 0; k < 3; k ++ ){ //проверить 3 вида скобок

if(s[i] == br1[k]){ // открывающая скобка

Push(S, s[i]);

break;

}

if(s[i] == br2[k]){ // закрывающая скобка

upper = Pop(S);

if(upper != br1[k])

OK = 0;

break;

}

}

if(OK && (S.size == 0))

printf("\nВыpажение пpавильное\n");

else

printf("\nВыpажение непpавильное \n");

system("PAUSE");

}

**1.6.2. Реализация стека с помощью списка**

**Пример ниже:**

struct Node

{

char data;

Node \*next, \*prev;

};

typedef Node \*PNode;

struct Stack

{

PNode Head, Tail;

};

**1.6.2.1. Добавление элемента на вершину стека**

В параметрах процедуры указывается не новый узел, а только данные для этого узла, то есть целое число. Память под новый узел выделяется в процедуре, то есть, скрыта от нас и снижает вероятность ошибки.

Добавление элемента на вершину:

void Push(Stack &S, char x)

{

PNode NewNode;

NewNode = new Node; //создать новый узел...

NewNode->data = x; //и заполнить его данными

NewNode->next = S.Head;

NewNode->prev = NULL;

if (S.Head) //добавить в начало списка

S.Head->prev = NewNode;

S.Head = NewNode;

if(! S.Tail)

S.Tail = S.Head;

}

**1.6.2.2. Получение верхнего элемента с вершины стека**

Это добавление нового элемента в начало двусвязного списка.

**Пример получения внрхнего элемента ниже:**

char Pop(Stack &S)

{

PNode TopNode = S.Head;

char x;

if(! TopNode) //если стек пуст, то

return char(255); //вернуть символ с кодом 255

x = TopNode->data;

S.Head = TopNode->next;

if(S.Head)

S.Head->prev = NULL; //переставить ссылки

else

S.Tail = NULL;

delete TopNode; //освободить память

return x;

}

void main()

{

setlocale(LC\_ALL, "Rus");

char br1[3] = { '(', '[', '{' }; // открывающие скобки

char br2[3] = { ')', ']', '}' }; // закрывающие скобки

char s[80], upper;

int i, k, OK;

Stack S; //стек символов

S.Head=NULL;

S.Tail=NULL;

printf("Введите выражение со скобками> ");

gets(s);

//S.size = 0; //сначала стек пуст

OK = 1;

for(i = 0; OK && (s[i] != '\0'); i ++)

for(k = 0; k < 3; k ++ ){ //проверить 3 вида скобок

if(s[i] == br1[k]){ // открывающая скобка

Push(S, s[i]);

break;

}

if(s[i] == br2[k]){ // закрывающая скобка

upper = Pop(S);

if(upper != br1[k])

OK = 0;

break;

}

}

//if(OK && (S.size == 0))

if(OK && (S.Head == NULL) && (S.Tail == NULL))

printf("\nВыpажение пpавильное\n");

else

printf("\nВыpажение непpавильное \n");

system("PAUSE");

}

**1.6.3. Системный стек в программах**

Программа использует стек для хранения

• адресов возврата из процедур и функций (это адреса, на которые переходит программа после выполнения процедуры или функции);

• параметров, передаваемых в процедуры и функции;

• локальных переменных в процедурах и функциях;

• временных данных (в основном в программах на ассемблере).

**1.7. Очередь**

Очередь – это упорядоченный набор элементов, который реализовывается методом FIFO (Очередь называют структурой типа FIFO (First In – First Out) – первым пришел, первым ушел.), в котором добавление новых элементов допустимо с одного конца (он называется концом очереди), а удаление существующих элементов – только с другого конца, который называется началом очереди.

**1.7.1. Реализация очереди с помощью массива**

**Реализация очереди ниже:**

int Pop(Queue &Q)

{

int temp;

if(Q.size == 0){

printf("Очередь пуста\n");

return 32767; //сигнал об ошибке

}

temp = Q.data[Q.head];

Q.head ++;

if(Q.head >= MAXSIZE)

Q.head = 0;

Q.size --;

return temp;

}

**1.7.2. Реализация очереди с помощью списка**

**Реализация ниже:**

struct Node{

int data;

Node \*next, \*prev;

};

typedef Node \*PNode;

Чтобы не работать с отдельными указателями на хвост и голову списка, объявим структуру, в которой будет храниться вся информация об очереди:

struct Queue

{

PNode head, tail;

};

**1.7.2.1. Добавление элемента в конец очереди**

**Реализация добавления элемента в конце очереди ниже:**

void PushTail(Queue &Q, int x)

{

PNode NewNode;

NewNode = new Node; //создать новый узел

NewNode->data = x; //заполнить узел данными

NewNode->prev = Q.Tail;

NewNode->next = NULL;

if (Q.tail) //добавить узел в конец списка

Q.tail->next = NewNode;

Q.tail = NewNode;

if (! Q.head)

Q.head = Q.tail;

}

**1.8. Дек**

Дек— структура данных, представляющая из себя список элементов, в которой добавление новых элементов и удаление существующих производится с обоих концов.

Дек (от англ. deque — double ended queue) — структура данных, представляющая из себя список элементов, в которой добавление новых элементов и удаление существующих производится с Для дека разрешены четыре операции:

• добавление элемента в начало;

• добавление элемента в конец;

• удаление элемента с начала;

• удаление элемента с конца.

обоих концов.

**1.9. Деревья**

**1.9.1. Основные понятия**

**1.9.1.1. Термины и определения**

Дерево – это узлы и соединяющие их направленных ребер, причем в каждый узел ведет ровно одна дуга.

Корень – это начальный узел дерева, в который не ведет ни одной дуги.

Примером может служить генеалогическое дерево - в корне дерева находитесь вы сами, от вас идет две дуги к родителям, от каждого из родителей - две дуги к их родителям и т.д.

**1.9.1.2. Рекурсивное определение**

Дерево представляет собой типичную рекурсивную структуру. Определение дерева состоит из двух частей – первая определяет условие окончания рекурсии, а второе – механизм ее использования

**1.9.1.3. Двоичные деревья**

Двоичным деревом называется дерево, каждый узел которого имеет не более двух сыновей.

Можно определить двоичное дерево и рекурсивно:

• пустая структура является двоичным деревом;

• дерево – это корень и два связанных с ним двоичных дерева, которые называют левым и правым поддеревом.

Строго двоичным деревом называется дерево, у которого каждая внутренняя вершина имеет непустые левое и правое поддеревья.

Полным двоичным деревом называется дерево, у которого все листья находятся на одном уровне и каждая внутренняя вершина имеет непустые левое и правое поддеревья.

**1.10. Графы**

**1.10.1. Основные понятия**

**1.10.1.1. Определения**

Граф - это совокупность узлов (вершин) и соединяющих их ребер (дуг).

Цепью называется последовательность ребер, соединяющих две (возможно не соседние) вершины u и v. В направленном графе такая последовательность ребер называется «путь».

Граф называется связным, если существует цепь между любой парой вершин. Если граф не связный, то его можно разбить на k связных компонент – он называется k-связным.

В практических задачах часто рассматриваются взвешенные графы, в которых каждому ребру приписывается вес (или длина). Такой граф называют сетью.

Циклом называется цепь из какой-нибудь вершины v в нее саму.

Деревом называется граф без циклов.

Полным называется граф, в котором проведены все возможные ребра (для графа, имеющего n вершин таких ребер будет n(n-1)/2.

**1.10.2.2. Жадные алгоритмы**

Представим себе зимовщика, которому предоставили некоторый запас продуктов на всю зиму. Конечно, он может сначала съесть все самое вкусное – шоколад, мясо и т.п., но за такой подход придется жестоко расплачиваться в конце зимовки, когда останется только соль и маргарин.

Подобным образом, если оптимальное решение строится по шагам, обычно нельзя выбирать на каждом этапе «самое вкусное» – за это придется расплачиваться на последних шагах. Такие алгоритмы называют жадными.

**1.10.3.2. Алгоритм Дейкстры**

• Массив {ai}, в котором ai =0, если вершина i еще не рассмотрена, и ai =1, если вершина i уже рассмотрена.

• Массив {bi}, в котором bi – текущее кратчайшее расстояние от выбранной стартовой вершины x до вершины i.

• Массив {ci}, в котором ci – номер предпоследней вершины в текущем кратчайшем пути из выбранной стартовой вершины x до вершины i.

Сам алгоритм состоит из трех этапов: инициализации, основного цикла и вывода результата.

**1.10.3.5. Алгоритм Флойда-Уоршелла**

Сначала в матрице {dij} записаны расстояние между вершинами напрямую. Затем рассмотрим все пути, которые проходят через первую вершину. Если путь через нее короче, чем напрямую, то заменяем значение в матрице на новый кратчайший путь. В конце элемент матрицы {dij} содержит длину кратчайшего пути из i в j. Каждая строка матрицы {pij} сначала заполняется так, как массив c в алгоритме Дейкстры, а в конце элемент {pij} равен номеру предпоследней вершины для кратчайшего пути из вершины i в вершину j.

**1.10.6. Задача о паросочетаниях**

**1.10.6.1. Формулировка задачи**

Задача. Есть m мужчин и n женщин. Каждый мужчина указывает несколько (от 0 до n) женщин, на которых он согласен жениться. Каждая женщина указывает несколько мужчин (от 0 до m), за которых она согласна выйти замуж. Требуется заключить наибольшее количество моногамных браков.

Для формулировки задачи в терминах теории графов надо ввести несколько определений.

Двудольным называется граф, все вершины которого разбиты на две доли (части), а ребра проходят только между вершинами разных частей.

Паросочетанием называется множество ребер, не имеющих общих вершин.

Таким образом, эта задача равносильна следующей:

Задача о наибольшем паросочетании. В заданном двудольном графе найти наибольшее паросочетание.

Для описания графа введем матрицу возможных браков A={aij}, где aij =1, если возможен брак между мужчиной i (i=1..m) и женщиной j (j=1..n) (оба согласны) и aij =0, если брак невозможен.

**1.11. Примеры динамических структур данных**

**1.11.1. Односвязный список**

**1.11.1.1. Пример #01. Односвязный список**

Пример ниже:

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <locale.h>

//using namespace std;

#define MAX 30 /\* максимальная длина фамилии \*/

struct student {

unsigned char f[MAX];

int ekz[4];

student \*next;

};

int show (student \*head) {

int count=0;

while (1) {

//вывести данные очередного узла

printf ("\n%s", head->f);

for (int i=0; i<4; i++) printf (" %d",head->ekz[i]);

count++;

//проверка на конец списка

if (head->next == NULL) break;

//переход к следующему узлу

head = head->next;

}

printf ("\nAll=%d",count);

return count;

}

student \*search (student \*head, char \*st) {

if (head==NULL) return NULL; //список пуст

student \*next = head;

st=\_strlwr((char \*)st); //чтобы поиск не зависел от регистра символов

do {

char \*find = strstr (strlwr((char \*)next->f), st);

if (find!=NULL) return next;

if (next->next==NULL) break;

next = next->next;

} while (1);

return NULL; //не найдено

}

student \*exclude (student \*head, char \*f) {

if (head==NULL) return NULL;

student \*current = head, //текущая запись

\*start = head, //начало списка

\*prev = NULL; //предыдущая запись

while (current) {

if (stricmp((char \*)current->f,f)==0) {

if (prev) prev->next = current->next;

else start=current->next;

free(current); //1

break; //2

}

prev = current;

current = current->next;

}

return start;

}

void copy0(student \*to, student \*from) {

strcpy ((char \*)to->f,(char \*)from->f);

for (int i=0; i<4; i++) to->ekz[i]=from->ekz[i];

}

student \*add1 (student \*head,student \*st) {

student \*current = (student \*) malloc (sizeof(student));

copy0 (current,st);

if (head==NULL) current->next=NULL;

else {

current->next = head;

head = current;

}

return current;

}

student \*add2 (student \*head,student \*st) {

student \*last=NULL;

if (head) {

last=head;

while (last->next) last=last->next;

}

student \*current = (student \*) malloc (sizeof(student));

copy0 (current,st);

current->next=NULL;

if (last) last->next = current;

return current;

}

int check (student \*head, char \*s) {

while (head) {

if (stricmp((char \*)head->f,s)==0) return 1;

head = head->next;

}

return 0;

}

student \*sort (student \*ph) {

student \*q,\*out=NULL,\*p,\*pr; //out - выход - сначала пуст

while (ph !=NULL) { //пока не конец входного списка

q = ph; ph = ph->next; //исключить очередной элемент

for (p=out,pr=NULL;

p!=NULL && stricmp((char \*)q->f,(char \*)p->f)>0;

pr=p,p=p->next) ;

//ищем, куда включить очередной элемент - тут stricmp

//задает критерий сравнения элементов, в вашей задаче м.б. другой

if (pr==NULL) { q->next=out; out=q; } //включение в начало

else { q->next=p; pr->next=q; } //или после предыдущего

}

return out;

}

int main (void) {

setlocale(LC\_ALL, "Rus");

FILE \*f=fopen("data.txt","rt");

if (f==NULL) {

printf ("\nFile data.txt not found"); exit(1);

}

char buf[40]; //буфер

student \*current = NULL, //текущий элемент

\*prev = NULL, //есть ли предыдущий?

\*head; //начало списка

do {

fgets (buf, 40, f);

if (strlen(buf)<10) continue; //Убираем слишком короткие строки

if (current!=NULL) prev=current; //если был тек.эл-т, запомнили его

current = (student \*) malloc(sizeof(student));

if (current==NULL) {

printf ("\nNo memory"); exit(2);

}

current->next = NULL; //добавляемый эл-т пометили как последний

if (prev!=NULL) prev->next = current;

//Если добавляемый эл-т current не первый,

//предыдущий эл-т заставили показывать на него

else head = current;

//Иначе запомнили, что он – начало списка

sscanf (buf,"%s %d %d %d %d",current->f,

&current->ekz[0], &current->ekz[1],

&current->ekz[2], &current->ekz[3]);

} while (!feof(f));

printf ("\nЧтение записей из файла и внесение их в список"); //

show (head);

printf ("\n\nДобавление записи вначало и конец списка"); //

student test1 = {"Popova",4,4,4,4};

head=add1 (head,&test1);

student test2 = {"Vasilenko",5,5,5,4};

add2 (head,&test2);

show (head);

printf ("\n\nПроверка записи в списке"); //

printf ("\n%d- запись \"%s\" есть ",check(head,(char \*)test1.f), test1.f); //1

printf ("\n%d - записи \"NeponyatnoKto\" нет",check(head,"NeponyatnoKto")); //0

printf ("\n\nСортированый список"); //

head = sort (head);

show (head);

fclose (f);

fflush (stdin);

getchar ();

return 0;

}

**1.11.2. Двухсвязный список**

**1.11.2.1. Пример #01. Двухсвязный список**

**Реализация двусвязного списка ниже:**

/\* Процедурно-ориентированная реализация двусвязных списков

Элемент списка - динамическая строковая переменная

\*/

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <locale.h>

#include <windows.h>

#include <stdio.h>

#include <conio.h>

#include <string.h>

#include <malloc.h>

typedef struct list {

list \*next, \*prev;

char \*info;

};

char buffer[80];

char \*input\_info(list \*ptr) {

fflush(stdin);

scanf("%s", &buffer[0]);

if (buffer[0]=='0') return NULL;

ptr->info=(char \*)malloc((strlen(buffer)+1)\*sizeof(char));

if (ptr->info == NULL) {

puts("Прием информации завершен: нет памяти");

return NULL;

}

strcpy (ptr->info,buffer);

return ptr->info;

}

list \* printOn(list \* ptr) {

printf("\n%s", ptr -> info);

return (ptr -> next);

}

list \* display(list \* head) {

list \* ret;

printf ("\n");

if (head == NULL) { /\* если список пуст \*/

puts("Список пуст");

return NULL;

}

do {

printOn(head);

ret = head;

head = head -> next;

} while(head != NULL);

printf ("\n");

return (ret);

}

void destroy (list \*element) {

free (element->info);

free (element);

}

int hasMember (list \* head, list \* work) {

while(head != NULL) {

if (!strcmpi (head->info,work->info)) return 1;

head = head -> next;

}

return 0;

}

int add(list \*\* head, list \* new\_ptr) {

list \* first, \* second;

if ((\*head) == NULL) { /\* если список пуст \*/

(\* head) = new\_ptr;

new\_ptr->next = NULL;

new\_ptr->prev = NULL;

return 0;

}

if ((\*head) -> next == NULL) { /\* если в нем всего один элемент \*/

if (strcmpi((\*head)->info,new\_ptr->info)>0) {

/\* новый элемент становится первым в списке \*/

second = (\*head); /\* сохраним указатель на второй \*/

(\*head) = new\_ptr;

new\_ptr -> next = second;

new\_ptr -> prev = NULL;

second -> next = NULL;

second -> prev = new\_ptr;

}

else { /\* новый элемент становится в конец списка \*/

(\*head) -> next = new\_ptr;

(\*head) -> prev = NULL;

new\_ptr -> next = NULL;

new\_ptr -> prev = new\_ptr;

}

return 1;

}

else {

if ( strcmpi((\*head)->info,new\_ptr->info)>0) { /\* новый эл-т ставится первым в списке \*/

second = (\*head);

(\*head) = new\_ptr;

new\_ptr -> next = second;

new\_ptr -> prev = NULL;

second -> prev = new\_ptr;

return 4;

}

first = (\* head);

second = first -> next;

while (first -> next != NULL) { /\* цикл поиска места в списке \*/

if ( (strcmpi(first->info,new\_ptr->info)<0) &&

(strcmpi(second->info,new\_ptr->info)>0) ) {

/\* вставляем элемент между first и second \*/

first->next = new\_ptr;

second->prev= new\_ptr;

new\_ptr->next = second;

new\_ptr->prev = first;

return 2;

}

first = second;

second = first->next;

}

/\* если добрались сюда, элемент ставим в конец списка \*/

first->next=new\_ptr;

first->prev=second;

new\_ptr->next = NULL;

new\_ptr->prev=first;

return 3;

}

}

/\* DETACH() - Удаляет элемент element из списка head \*/

void detach(list \*\* head, list \* element) {

list \* prev;

if(\* head == NULL) return; /\* если список пуст \*/

if((\*head) == element) { /\* если удаляемый эл-т - первый \*/

(\* head) = element -> next;

destroy(element);

return;

}

/\* Цикл поиска предыдущего для element элемента. \*/

prev = (\* head);

while((prev -> next) != element)

prev = prev -> next; /\* на выходе из цикла - адрес предыдущ\*/

prev -> next = element -> next;

destroy(element);

return;

}

int main(void) {

char ch;

list \* head; /\* указатель на начало списка \*/

list \* new\_ptr;/\* указатель на новый элемент списка \*/

list \* cur; /\* указатель на текущий элемент списка \*/

list work;

head = NULL; /\* вначале список пуст \*/

setlocale(LC\_ALL,"Rus");// SetConsoleCP(1251); SetConsoleOutputCP(1251);

puts ("Вводите строки для списка, 0 - завершение ввода");

while(input\_info(&work) != NULL) { /\* цикл ввода описаний \*/

if (hasMember(head, &work) != 1) {

new\_ptr = (list \*)malloc(sizeof(list));

if (new\_ptr == NULL) {

puts("Прием информации завершен: нет памяти");

break;

}

new\_ptr->info = work.info;/\* копируем введен. информацию \*/

add(&head, new\_ptr); /\* добавляем элемент в список \*/

//display (head);

}

else {

puts("Элемент уже есть в списке");

}

}

if (head == NULL) /\* список остался пустым ? \*/

return 1; /\* да, завершение программы \*/

//puts("\*\*\* СВЯЗАННЫЙ СПИСОК ПОСЛЕ ЗАВЕРШЕНИЯ ВВОДА \*\*\*");

printf("\n\*\*\* СВЯЗАННЫЙ СПИСОК ПОСЛЕ ЗАВЕРШЕНИЯ ВВОДА \*\*\*");

display(head); /\* вывод всех элем-тов списка \*/

/\* Цикл выборочного удаления элементов, начиная с первого. \*/

cur = head;

//puts("\n\*\*\* ВЫБОРОЧНОЕ УДАЛЕНИЕ ЭЛЕМЕНТОВ СПИСКА \*\*\*");

printf("\n\*\*\* ВЫБОРОЧНОЕ УДАЛЕНИЕ ЭЛЕМЕНТОВ СПИСКА \*\*\*");

do {

new\_ptr = printOn(cur); /\* вывод элемента списка \*/

printf(" Удаляете элемент ? (Y/N) ");

if ((ch = getch()) == 'Y' || ch == 'y') {

//puts("Yes");

printf("Yes");

detach(&head, cur); /\* удаление элемента из спсика \*/

}

else

printf("No");

//puts("No");

cur = new\_ptr;

}

while(new\_ptr != NULL);

//puts("\*\*\* СВЯЗАННЫЙ СПИСОК ПОСЛЕ ВЫБОРОЧНОГО УДАЛЕНИЯ \*\*\*");

printf("\n\*\*\* СВЯЗАННЫЙ СПИСОК ПОСЛЕ ВЫБОРОЧНОГО УДАЛЕНИЯ \*\*\*");

display(head); /\* вывод всех элем-тов списка \*/

fflush (stdin); getchar();

return 0;

}

**1.11.3. Стек**

**1.11.3.1. Cтек #01**

**Реализация стека ниже:**

#include <iostream>

using namespace std;

/\*НАША СТРУКТУРА\*/

struct List

{

int x; //информационный элемент

List \*Next,\*Head; //Голова стека и указатель на следующий элемент

};

/\*ФУНКЦИЯ ДОБАВЛЕНИЯ ЭЛЕМЕНТА В СТЕК (в список LIFO)\*/

void Add(int x, List \*&MyList) //Принимаем элемент стека и указатель на стек,

//при этом говорим, что принимаемый указатель будет сам по себе указателем

{

List \*temp=new List; //Выделяем память для нового элемента

temp->x=x; //Записываем в поле x принимаемый в функцию элемент x

temp->Next=MyList->Head; //Указываем, что следующий элемент это предыдущий

MyList->Head=temp; //Сдвигаем голову на позицию вперед

}

/\*ФУНКЦИЯ ОТОБРАЖЕНИЯ СТЕКА\*/

void Show(List \*MyList) //Нужен только сам стек

{

List \*temp=MyList->Head; //Объявляем указатель и Указываем ему, что его позиция в голове стека

//с помощью цикла проходим по всему стеку

while (temp!=NULL){ //выходим при встрече с пустым полем

cout<<temp->x<<" "; //Выводим на экран элемент стека

temp=temp->Next; //Переходим к следующему элементу

}

}

/\*ФУНКЦИЯ УДАЛЕНИЯ СТЕКА ИЗ ПАМЯТИ\*/

void ClearList(List \*MyList)

{

while (MyList->Head!=NULL){ //Пока по адресу не пусто

List \*temp=MyList->Head->Next; //Временная переменная для хранения адреса следующего элемента

delete MyList->Head; //Освобождаем адрес обозначающий начало

MyList->Head=temp; //Меняем адрес на следующий

}

}

int main()

{

setlocale(LC\_ALL, "Rus");

List \*MyList=new List; //Выделяем память для стека

MyList->Head=NULL; //Во избежание ошибок инициализируем первый элемент

for (int i=0;i<10;i++)

Add(i,MyList); //Заносим данные в стек

cout<<"В стеке записано:\n"; //Выводим на экран элемент стека

Show(MyList); //Выводим стек на экран

ClearList(MyList); //Очищаем память.

cout<<"\n";

system("PAUSE");

delete MyList->Head;

delete MyList;

}

**1.11.4. Очередь**

**1.11.4.1. Очередь #01**

**Реализация очереди ниже:**

/\*

Процедурно-ориентированная реализация очереди

\*/

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <locale.h>

#include <windows.h>

#include <stdio.h>

#include <string.h>

#include <malloc.h>

typedef struct QUEUE {

QUEUE \* next;

int info;

};

int input\_info(QUEUE \* ptr) {

fflush(stdin);

scanf("%d", &(ptr -> info));

return ptr->info;

}

QUEUE \* printOn(QUEUE \* ptr) {

printf("%d ", ptr -> info);

return (ptr -> next);

}

QUEUE \* display (QUEUE \* head) {

QUEUE \* ret;

if (head == NULL) { /\* если список пуст \*/

puts("Список пуст");

return NULL;

}

do {

printOn(head);

ret = head;

head = head -> next;

} while(head != NULL);

return (ret);

}

/\* получить элемент из очереди \*/

QUEUE \* get(QUEUE \*\* head) {

QUEUE \* current, \* previous;

if((\*head) == NULL) /\* если список пуст \*/

return NULL;

if ((\* head) -> next == NULL) { /\* если всего один элемент \*/

current = (\* head);

(\* head) = NULL;

return current;

}

previous = (\* head);

current = previous -> next;

while (current -> next != NULL) { /\* цикл поиска конца \*/

previous = current;

current = previous -> next;

}

previous -> next = NULL;

return current;

}

/\* поместить элемент в очередь \*/

void put(QUEUE \*\* head, QUEUE \* newitem) {

QUEUE \* old\_head;

if ((\* head) == NULL) { /\* список пуст \*/

(\* head) = newitem;

newitem -> next = NULL;

}

else {

old\_head = (\* head);

(\* head) = newitem;

newitem -> next = old\_head;

}

}

int main(void) {

QUEUE \* head; /\* указатель на начало списка \*/

QUEUE \* newitem; /\* элемент очереди \*/

QUEUE work;

head = NULL; /\* вначале список пуст \*/

setlocale(LC\_ALL,"Rus"); //SetConsoleCP(1251); SetConsoleOutputCP(1251);

puts ("Вводите целые числа в очередь, 0 - завершение ввода");

while(input\_info(&work) != 0) { /\* цикл ввода описаний \*/

if ((newitem = (struct QUEUE \*)malloc(sizeof(struct QUEUE))) == NULL) {

puts("Прием информации завершен: нет памяти");

break;

}

newitem -> info = work.info; /\* копируем введен. информацию \*/

put(&head, newitem); /\* добавляем элемент в очередь \*/

}

if(head == NULL) /\* список остался пустым ? \*/

return 1; /\* да, завершение программы \*/

puts("\n\*\*\* ОЧЕРЕДЬ ПОСЛЕ ЗАВЕРШЕНИЯ ВВОДА \*\*\*");

display(head); /\* вывод всех элементов списка \*/

/\* Цикл чтения/удаления элементов очереди, начиная с первого

пришедшего в очередь.

\*/

puts("\n\*\*\* ПОЭЛЕМЕНТНОЕ ЧТЕНИЕ ЭЛЕМЕНТОВ ОЧЕРЕДИ \*\*\*");

while((newitem = get(&head)) != NULL) {

printOn(newitem); /\* вывод элемента очереди \*/

free(newitem);

}

puts("\n\*\*\* ОЧЕРЕДЬ ПОСЛЕ ЧТЕНИЯ ЭЛЕМЕНТОВ \*\*\*");

display(head); /\* вывод всех элем-тов списка \*/

QUEUE n,n2;

n.info=111; n2.info=222; put(&head, &n); put(&head, &n2);

puts("\n\*\*\* ОЧЕРЕДЬ ПОСЛЕ НОВОГО ДОБАВЛЕНИЯ 2 ЭЛЕМЕНТОВ ПРОГРАММНО \*\*\*");

display(head);

fflush (stdin); getchar();

return 0;

}

**1.11.4.2. Очередь #02 с использованием шаблона**

Методы очереди

Метод — это та же самая функция, но она работает только с контейнерами STL. Например, очередь и стек.

Для работы с очередью вам понадобится знать функции: push(), pop(), front(), back(), empty(). Кстати, если хотите узнать, как в C++ работают функции и как их правильно использовать в проекте, то можете узнать все это здесь.

• Для добавления в очередь нового элемента нужно воспользоваться функцией — push(). В круглых скобках должно находится значение, которое мы хотим добавить.

• Если нам понадобилось удалить первый элемент нужно оперировать функцией pop(). В круглых скобках уже не чего не нужно указывать, но по правилам они в обязательном порядке должны присутствовать! Эти функции тоже не нуждаются в указании аргумента: empty(), back() и front().

• Если вам понадобилось обратиться к первому элементу очереди, то вам понадобится функция front().

• Чтобы обратиться к последнему элементу в очереди вам поможет функция back().

• Чтобы узнать пуста ли очередь нужно воспользоваться функцией empty().

Если ваша очередь пуста — возвратит true.

Если же в ней что-то есть — возвратит false

**1.11.5. Дек**

**1.11.5.1. Дек #01 простой пример**

**Пример:**

include <iostream>

const int N\_global=10; //Количество элементов статического одномерного массива на котором строится дек

void showarr(int \*Arr,const int N) //Функция показывает одномерный массив на экране

{

for (int i=0;i<N;i++) std::cout<<Arr[i]<<"\t";

std::cout<<"\n";

}

void addright(int Arr[N\_global],int &N, const int el) //Функция добавляет элемент в конец очереди

{

if (N==N\_global) //Если размер очереди стал равен размеру массива

{

std::cout<<"Дек полон\n"; //Сообщаем, что дек полон

std::cout<<el<<" не попал в дек\n";

return; // и сразу выходим из функции

}

//Если размер очереди меньше размера массива

Arr[N]=el; //Записываем элемент в конец массива

N++; //Увеличиваем счетчик размерности очереди

}

void addleft(int Arr[N\_global],int &N,const int el) //Функция добавляет элемент в начало очереди

{

if (N==N\_global) //Если размер дека равен размеру массива

{

std::cout<<"Дек полон\n"; //Выводим сообщение, что дек полон

std::cout<<el<<" не попал в дек\n";

return; //и сразу выходим из функции

}

int temp=Arr[0]; //Запоминаем первый элемент во временную переменную

Arr[0]=el; //Записываем в первый элемент значение

N++; //Увеличиваем счетчик размерности очереди

for (int i=1;i<N;i++) std::swap(Arr[i],temp); //как бы отталкиваем весь массив вправо, не трогаем самый левый элемент

}

int main()

{

int mas[N\_global];

int len=0; //Сначала дек пустой, размер = 0

//Добавление элементов в начало очереди

addleft(mas,len,1);

addleft(mas,len,7);

addleft(mas,len,4);

addleft(mas,len,14);

addleft(mas,len,54);

addleft(mas,len,24);

addright(mas,len,1000); //В конец 1000

addleft(mas,len,2000); //В начало 2000

showarr(mas,len); //Показываем созданную очередь

len=0;//Создали иллюзию обнуления очереди.

addleft(mas,len,777); //Создаем другую очередь

addleft(mas,len,1);

showarr(mas,len); //Показываем вторую очередь

std::cin.get();

}

**1.11.6. Дерево**

**1.11.6.1. Дерево #01 первое знакомство**

#include <iostream>

#include <conio.h>

using namespace std;

//Наша структура

struct node

{

int info; //Информационное поле

node \*l, \*r;//Левая и Правая часть дерева

};

node \* tree=NULL; //Объявляем переменную, тип которой структура Дерево

/\*ФУНКЦИЯ ЗАПИСИ ЭЛЕМЕНТА В БИНАРНОЕ ДЕРЕВО\*/

void push(int a,node \*\*t)

{

if ((\*t)==NULL) //Если дерева не существует

{

(\*t)=new node; //Выделяем память

(\*t)->info=a; //Кладем в выделенное место аргумент a

(\*t)->l=(\*t)->r=NULL; //Очищаем память для следующего роста

return; //Заложили семечко, выходим

}

//Дерево есть

if (a>(\*t)->info) push(a,&(\*t)->r); //Если аргумент а больше чем текущий элемент, кладем его вправо

else push(a,&(\*t)->l); //Иначе кладем его влево

}

/\*ФУНКЦИЯ ОТОБРАЖЕНИЯ ДЕРЕВА НА ЭКРАНЕ\*/

void print (node \*t,int u)

{

if (t==NULL) return; //Если дерево пустое, то отображать нечего, выходим

else //Иначе

{

print(t->l,++u);//С помощью рекурсивного посещаем левое поддерево

for (int i=0;i<u;++i) cout<<"|";

cout<<t->info<<endl; //И показываем элемент

u--;

}

print(t->r,++u); //С помощью рекурсии посещаем правое поддерево

}

void main ()

{

int n; //Количество элементов

int s; //Число, передаваемое в дерево

setlocale(LC\_ALL, "Rus");

cout<<"введите количество элементов ";

cin>>n; //Вводим количество элементов

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

{

cout<<"ведите число ";

cin>>s; //Считываем элемент за элементом

push(s,&tree); //И каждый кладем в дерево

}

cout<<"ваше дерево\n";

print(tree,0);

getch();

}

**1.11.7. Граф**

**1.11.7.1. Граф #01 простой пример**

Пример ниже:

#include <iostream>

#include <stdlib.h>

using namespace std;

const int n=5;

int c[n] ; // номер хода, на котором посещается вершина

int path[n]; // номера посещаемых вершин

int v0=2; // начальная вершина

//Матрица смежности

int a[n][n]=

{

0,1,1,0,0,

1,0,0,1,0,

1,0,0,1,0,

0,1,1,0,1,

0,0,1,1,0

};

void prnt(void)

{

int p;

for ( p = 0 ; p<n ; p++)

cout<<path[p]+1<<"\t";

cout<<path[0]+1 ;

cout<<"\n" ;

}

//подпрограмма нахождения гамильтонова цикла

int gamilton ( int k)

{

int v,q1=0;

for(v=0; v<n && !q1; v++)

{

if(a[v][path[k-1]]||a[path[k-1]][v])

{

if (k==n && v==v0 ) q1=1;

else if (c[v]==-1)

{

c[v] = k ; path[k]=v;

q1=gamilton (k+1) ;

if (!q1) c[v]=-1;

} else continue;

}

} return q1;

}

int main()

{

int j;

system("CLS");

setlocale(LC\_ALL, "Rus");

cout<<"Гамильтонов цикл:\n";

for(j=0;j<n;j++) c[j]=-1;

path[0]=v0 ;

c[v0]=v0;

if(gamilton (1)) prnt(); else cout<<"Нет решений\n";

cin.get();

return 0;

}

Раздел 19(2) 1. Динамические структуры данных C++. Дерево

**1.1. Основные понятия, терминология, определения**

Дерево - конечное множество элементов, состоящее из одного или более элементов, и они имеют:

• имеется одна специально выделенная вершина, называемая корнем дерева;

• остальные вершины содержатся в m попарно непересекающихся множествах поддеревьях, каждое из которых, в свою очередь, является деревом.

Упорядоченным деревом называется такое дерево, в котором важен порядок следования поддеревьев.

Дуга – это ориентированное соединение между двумя вершинами дерева, а значит корень можно определить как такую вершину дерева, в который не входит ни одной дуги.

Ребро - это неориентированное соединение между двумя вершинами дерева. Ребро можно превратить в дугу, если задать на нем направление, а любое дерево можно превратить в ориентированное дерево, если задать нправление ребер.

Количество поддеревьев некоторой вершины называется степенью этой вершины. Деревья, имеющие степень больше 2 - сильно ветвящимися деревьями.

Вершина с нулевой степенью называется листом.

**1.1.1. Рекурсивное определение**

Дерево представляет собой типичную рекурсивную структуру, которая определяется через саму себя. Дерева состоит из двух частей – первая определяет условие окончания рекурсии, а второе – механизм ее использования.

ЗАМЕЧАНИЕ - Размер памяти, необходимый для хранения дерева, заранее неизвестен, потому что неизвестно, сколько узлов будет в него входить.

**1.1.2. Способы изображения деревьев**

Более близкие к корню узлы называются предками, а более далекие -потомками.

Узлы, не содержащие поддеревьев, называются концевыми узлами или листьями. Множество не пересекающихся деревьев называется лесом. Например, лес образуют поддеревья, исходящие из одного узла. Одним из способов изображения деревьев будут БИНАРНЫЕ ДЕРЕВЬЯ.

**1.2. Бинарные деревья поиска**

Двоичное дерево - дерево, каждый узел которого имеет не более двух сыновей.

Строго двоичное дерево, это дерево, у которого каждая внутренняя вершина имеет непустые левое и правое поддеревья.

Полным двоичным деревом называется дерево, у которого все листья находятся на одном уровне и каждая внутренняя вершина имеет непустые левое и правое поддеревья.

Пример двоичного дерева ниже:

struct node

{

int Key; // Ключ вершины.

int Count; // Счетчик количества вершин с одинаковыми ключами.

node \*Left; // Указатель на "левого" сына.

node \*Right; // Указатель на "правого" сына.

};

Дальше будут разобраны алгоритмы деервьев.

**1.3. Построение бинарного дерева поиска (рекурсивный алгоритм)**

Построение таким алгоритмом дерева непрсотая задача, поэтому смотрим на пример с комментариями ниже:

void Search (int x, node \*\*p)

// Поиск вершины с ключом x в дереве со вставкой

// (рекурсивный алгоритм).

// \*p - указатель на корень дерева.

{

if (\*p==NULL)

{ // Вершины с ключом x в дереве нет; включить ее.

\*p = new(node);(\*\*p).Key = x; (\*\*p).Count = 1;

(\*\*p).Left = (\*\*p).Right = NULL;

}

else //Поиск места включения вершины.

if (x<(\*\*p).Key) //Включение в левое поддерево.

Search (x,&((\*\*p).Left));

else if (x>(\*\*p).Key) //Включение в правое поддерево.

Search (x,&((\*\*p).Right));

else (\*\*p).Count = (\*\*p).Count + 1;

}

ЗАМЕЧАНИЕ! Методы поиска по динамическим таблицам часто называют алгоpитмами таблиц символов, так как компилятоpы и дpугие системные пpогpаммы обычно используют их для хpанения опpеделяемых пользователем символов.

**1.4. Анализ алгоpитма поиска с включениями**

Теоpема Хопкpофта-Ульмана

Сpеднее число сpавнений, необходимых для вставки n случайных элементов в деpево поиска, пустое вначале, pавно O(nlog2n) для n>=1.

Вывод: в сpеднем на вставку n элементов в деpево двоичного поиска тpатится в среднем O(nlog2n)сравнений.

**1.5. Дерево отрезков**

Деpево отpезков - стpуктуpа данных, созданная для pаботы с такими интеpвалами на числовой оси, концы котоpых пpинадлежат фиксиpованному множеству из абсцисс.

Построение дерева:

#include <iostream>

using namespace std;

struct node

{

int KeyMin; // Минимальный ключ вершины.

int KeyMax; // Максимальный ключ вершины.

node \*Left; // Указатель на "левого" сына.

node \*Right; // Указатель на "правого" сына.

};

class TREE

{

private:

node \*Tree; //Указатель на корень дерева.

void Search (int,int,node\*\*);

public:

TREE() {Tree = NULL;}

void BuildTree (); //Построение дерева отрезков.

node\*\* GetTree () {return &Tree;} //Получение вершины дерева.

void CleanTree (node \*\*);

void Vyvod (node \*\*,int);

};

void main ()

{

setlocale(LC\_ALL,"Rus");

TREE A;

A.BuildTree ();

cout<<"\nВывод дерева:\n";

A.Vyvod (A.GetTree(),0);

A.CleanTree (A.GetTree());

cout << "\n";

system("PAUSE");

}

void TREE::BuildTree ()

// Построение бинарного дерева (рекурсивный алгоритм).

// Tree - указатель на корень дерева.

{

int k1,k2;

cout<<"Введите два целых числа...\n";

cin>>k1;

cin>>k2;

Search (k1,k2,&Tree);

}

void TREE::Search (int k1, int k2, node \*\*p)

// Постpоение деpева отpезков p.

// p - указатель на корень дерева.

{

if (k2-k1>1)

{

\*p = new (node);

(\*\*p).KeyMin = k1;

(\*\*p).KeyMax = k2;

(\*\*p).Left = (\*\*p).Right = NULL;

Search (k1,(k1+k2)/2,&((\*\*p).Left));

Search ((k1+k2)/2,k2,&((\*\*p).Right));

}

else

{

\*p = new (node);

(\*\*p).KeyMin = k1;

(\*\*p).KeyMax = k2;

(\*\*p).Left = (\*\*p).Right = NULL;

}

}

void TREE::CleanTree (node \*\*w)

//Очистка дерева.

//\*w - указатель на корень дерева.

{

if (\*w!=NULL)

{ CleanTree (&((\*\*w).Left));

CleanTree (&((\*\*w).Right));

delete \*w; }

}

void TREE::Vyvod (node \*\*w,int l)

//Изображение дерева \*w на экране дисплея

// (рекурсивный алгоритм).

//\*w - указатель на корень дерева.

{

int i;

if (\*w!=NULL)

{ Vyvod (&((\*\*w).Right),l+1);

for (i=1; i<=l; i++) cout<<" ";

cout<<(\*\*w).KeyMin<<", "<<(\*\*w).KeyMax<<endl;

Vyvod (&((\*\*w).Left),l+1); }

}

ЗАМЕЧАНИЕ! В вычислительной геометpии часто используется динамическая стpуктуpа данных деpево интеpвалов , используемая пpи pешении задачи о пеpесечении пpямоугольников.

**1.6. Обход бинарного дерева**

. Существуют три варианта обхода:

• (корень – левое – правое): сначала корень, затем левое поддерево, а затем – правое;

• (левое – правое – корень): сначала левое поддерево, затем правое, а затем – корень.

• (левое – корень – правое): сначала левое поддерево, затем корень, а затем – правое;

**1.6.1. Левосторонний обход бинарного дерева поиска (КЛП)**

Алгоритм левостороннего обхода дерева:

• посетите корень дерева;

• обойдите левое поддерево;

• обойдите правое поддерево.

Пример использования алгоритма:

void ObhodLeft (node \*\*w)

// Левосторонний обход дерева.

// \*w - указатель на корень дерева.

{

if (\*w!=NULL) //пустое дерево?

{ cout<<(\*\*w).Key<<" "; вывод информации о корне

ObhodLeft (&((\*\*w).Left)); //обход левого поддерева

ObhodLeft (&((\*\*w).Right)); }//обход правого поддерева

}

**1.6.2. Концевой обход бинарного дерева поиска (ЛПК)**

Алгорим концевого обхода дерева:

• обойдите левое поддерево;

• обойдите правое поддерево;

• посетите корень дерева.

Пример использования алгоритма:

void ObhodEnd (node \*\*w)

// Концевой обход дерева.

// \*w - указатель на корень дерева.

{

if (\*w!=NULL) //пустое дерево?

{ ObhodEnd (&((\*\*w).Left)); //обход левого поддерева

ObhodEnd (&((\*\*w).Right)); //обход правого поддерева

cout<<(\*\*w).Key<<" ";} //вывод информации о корне}

**1.6.3. Обратный обход бинарного дерева поиска (ЛКП)**

Алгорим обратного обхода дерева:

• обойдите левое поддерево;

• посетите корень дерева;

• обойдите правое поддерево.

Пример алгоритма:

void ObhodBack (node \*\*w)

// Обратный обход бинарного дерева.

// \*w - указатель на корень дерева.

{

if (\*w!=NULL) //пустое дерево?

{ ObhodBack (&((\*\*w).Left)); //обход левого поддерева

cout<<(\*\*w).Key<<" "; //вывод информации о корне

ObhodBack (&((\*\*w).Right)); }//обход правого поддерева

}

**1.7. Вывод бинарного дерева поиска**

Алгоритм вывода бинарного дерева поиска:

void Vyvod (node \*\*w,int l)

// Изображение дерева w на экране дисплея.

// (рекурсивный алгоритм).

// \*w - указатель на корень дерева.

// l - "отступ" от левого края окна при выводе

// равен глубине вершины (расстояние от корня до этой вершины)

{

int i;

if (\*w!=NULL) //пустое дерево?

{ Vyvod (&((\*\*w).Right),l+1); //правое поддерево

for (i=1; i<=l; i++) cout<<" "; // расстояние от корня до вершины

cout<<(\*\*w).Key<<endl; //вывод информации о корне

Vyvod (&((\*\*w).Left),l+1); } //левое поддерево

}

Дерево "лежит на левом боку". Сначала выводится правое поддерево, где вершина отступает от левого края окна на величину, равную глубине вершины.

**1.7.1. Пример программы с использованием бинарного дерева поиска**

Пример программы:

#include <iostream>

using namespace std;

struct node

{

int Key;

int Count;

node \*Left;

node \*Right;

};

class TREE

{

private:

node \*Tree; //Указатель на корень дерева.

void Search (int,node\*\*); //Поиск вершины с ключом int в дереве со вставкой

public:

TREE() {Tree=NULL;}

node\*\* GetTree () {return &Tree;} //Получение вершины дерева.

void BuildTree (); //Построение дерева

void CleanTree (node \*\*); //Очистка дерева

void ObhodEnd (node \*\*); //Концевой обход дерева

void ObhodLeft (node \*\*); //Левосторонний обход дерева

void ObhodBack (node \*\*); //Обратный обход дерева

void Vyvod (node\*\*,int); //Изображение дерева на экране дисплея

int Height (node\*\*); //Определение высоты бинарного дерева

};

void main ()

{

setlocale(LC\_ALL,"Rus");

TREE A;

A.BuildTree ();

cout<<"\nВывод дерева:\n";

A.Vyvod (A.GetTree(),0);

cout<<"\nВысота дерева:"<<A.Height(A.GetTree())<<endl;

cout<<"\nЛевосторонний обход дерева: ";

A.ObhodLeft (A.GetTree());

cout<<"\nКонцевой обход дерева: "; A.ObhodEnd (A.GetTree());

cout<<"\nОбратный обход дерева: "; A.ObhodBack (A.GetTree());

A.CleanTree (A.GetTree());

cout << "\n";

system("PAUSE");

}

void TREE::BuildTree ()

// Построение бинарного дерева (рекурсивный алгоритм).

// Tree - указатель на корень дерева.

{

int el;

cout<<"Вводите ключи вершин дерева ...\n";

cin>>el;

while (el!=0)

{ Search (el,&Tree); cin>>el; }

}

void TREE::Search (int x,node \*\*p)

// Поиск вершины с ключом x в дереве со вставкой

// (рекурсивный алгоритм).

// \*p - указатель на корень дерева.

{

if (\*p==NULL)

{// Вершины в дереве нет; включить ее.

\*p = new(node);

(\*\*p).Key = x; (\*\*p).Count = 1;

(\*\*p).Left = NULL; (\*\*p).Right = NULL; }

else

if (x<(\*\*p).Key) Search (x,&((\*\*p).Left));

else

if (x>(\*\*p).Key) Search (x,&((\*\*p).Right));

else (\*\*p).Count = (\*\*p).Count + 1;

}

void TREE::ObhodLeft (node \*\*w)

//Левосторонний обход дерева.

//\*w - указатель на корень дерева.

{

if (\*w!=NULL)

{

cout<<(\*\*w).Key<<" ";

ObhodLeft (&((\*\*w).Left));

ObhodLeft (&((\*\*w).Right));

}

}

void TREE::ObhodEnd (node \*\*w)

//Концевой обход дерева.

//\*w - указатель на корень дерева.

{

if (\*w!=NULL)

{ ObhodEnd (&((\*\*w).Left));

ObhodEnd (&((\*\*w).Right));

cout<<(\*\*w).Key<<" "; }

}

void TREE::ObhodBack (node \*\*w)

//Обратный обход дерева.

//\*w - указатель на корень дерева.

{

if (\*w!=NULL)

{ ObhodBack (&((\*\*w).Left));

cout<<(\*\*w).Key<<" ";

ObhodBack (&((\*\*w).Right)); }

}

void TREE::CleanTree (node \*\*w)

//Очистка дерева.

//\*w - указатель на корень дерева.

{

if (\*w!=NULL)

{ CleanTree (&((\*\*w).Left));

CleanTree (&((\*\*w).Right));

delete \*w; }

}

void TREE::Vyvod (node \*\*w,int l)

//Изображение дерева \*w на экране дисплея

// (рекурсивный алгоритм).

//\*w - указатель на корень дерева.

{

int i;

if (\*w!=NULL)

{ Vyvod (&((\*\*w).Right),l+1);

for (i=1; i<=l; i++) cout<<" ";

cout<<(\*\*w).Key<<endl;

Vyvod (&((\*\*w).Left),l+1); }

}

int TREE::Height (node \*\*w)

//Определение высоты бинарного дерева.

//\*w - указатель на корень дерева.

{

int h1,h2;

if (\*w==NULL) return (-1);

else

{

h1 = Height (&((\*\*w).Left));

h2 = Height (&((\*\*w).Right));

if (h1>h2) return (1 + h1);

else return (1 + h2);

}

}

**1.8. Построение бинарного дерева (нерекурсивный алгоритм)**

Построение:

void TreeSearch (node \*\*Tree,int el)

// Поиск вершины с информационным полем el в дереве

// с последующим включением.

// \*Tree - указатель на корень дерева.

{

node \*p1;

node \*p2; // Указатель p2 "опережает" указатель p1.

int d; // Флаг для распознавания поддеревьев.

p2 = \*Tree; p1 = (\*p2).Right;

d = 1; // Флаг правого поддерева.

while (p1!=NULL && d!=0)

{ p2 = p1;

if (el<(\*p1).Key) { p1 = (\*p1).Left; d = -1; } //Флаг левого поддерева

else

if (el>(\*p1).Key) { p1 = (\*p1).Right; d = 1; }

else d = 0; }

if (d==0) (\*p1).Count = (\*p1).Count + 1;

else

{ p1 = new(node);

(\*p1).Key = el; (\*p1).Left = (\*p1).Right = NULL; (\*p1).Count = 1;

if (d<0) (\*p2).Left = p1; else (\*p2).Right = p1;}

}

**1.9. Изображение бинарного дерева (нерекурсивный алгоритм)**

Бинарное дерево построено с помощью описанного на предыдущем шаге нерекурсивного алгоритма и определено указателем.

void VyvodTree (node \*t)

// Изображение дерева, заданного указателем t,

// на экране дисплея (нерекурсивный алгоритм).

{

no \*stk, \*stk1;

node \*u;

int i,n;

stk = stk1 = NULL; n = 0;

while (t!=NULL)

{ PushStack (&stk1,&t,&n);

if ((\*t).Right!=NULL)

{ if ((\*t).Left!=NULL)

PushStack (&stk,&((\*t).Left),&n);

t = (\*t).Right; }

else {

if ((\*t).Left!=NULL)

{ if (stk1!=NULL)

{ PopStack (&stk1,&u,&n);

for (i=0; i<=n; i++) cout<<" ";

cout<<(\*u).Key<<endl; }

t = (\*t).Left; }

else

if (stk==NULL) t = NULL;

else

{ while ((\*stk).elem!=(\*((\*stk1).elem)).Left)

{ PopStack (&stk1,&u,&n);

for (i=0; i<=n; i++) cout<<" ";

cout<<(\*u).Key<<endl;

}

PopStack (&stk1,&u,&n);

for (i=0; i<=n; i++) cout<<" ";

cout<< (\*u).Key<endl;

PopStack (&stk,&t,&n);

}

}

n = n + 1;

}

VyvodStack (&stk1);

}

**1.10. Пример программы построения и изображения бинарного дерева (нерекурсивные алгоритмы)**

Пример программы:

#include <iostream>

using namespace std;

struct node

{

int Key;

int Count;

node \*Left;

node \*Right;

};

struct no // Звено стека

{

node \*elem; // Информационное поле.

int ch; // Уровень вершины.

no \*sled; // Указатель на вершину.

};

class TREE

{

private:

node \*Tree;

void PushStack (no \*\*,node \*\*,int \*);// Помещение звена с элементами \*el и n в

// стек. \*stk - указатель на стек.

void PopStack (no\*\*,node \*\*,int \*); // Извлечение из стека звена

// с элементами \*t и n.

// \*stk - указатель на стек

void VyvodStack (no\*\*); // Вывод содержимого стека на экран дисплея.

// \*stk - указатель на стек.

public:

TREE () { Tree = new(node); (\*Tree).Right = NULL; }

node\* GetTreeRight () {return (\*Tree).Right;}

void TreeSearch (int); // Поиск вершины с информационным полем el в дереве с

// последующим (в случае неудачного поиска!) включением

// в дерево. Tree - указатель на корень дерева.

void VyvodTree (node \*); //Построение дерева, заданного указателем t,

//на экране дисплея (нерекурсивный алгоритм).

};

void main ()

{

setlocale(LC\_ALL,"Rus");

TREE A;

int el;

cout<<"Вводите значения информационных полей вершин: "<<endl;

cin>>el;

while (el!=0)

{ A.TreeSearch (el); cin>>el; }

A.VyvodTree (A.GetTreeRight());

cout << "\n";

system("PAUSE");

}

void TREE::TreeSearch (int el)

// Поиск вершины с информационным полем el в дереве с

// последующим (в случае неудачного поиска!) включением

// в дерево. Tree - указатель на корень дерева.

{

node \*p1,\*p2;

int d;

p2 = Tree; p1 = (\*p2).Right; d = 1;

while (p1!=NULL && d!=0)

{

p2 = p1;

if (el<(\*p1).Key) {p1 = (\*p1).Left; d = -1;}

else

if (el>(\*p1).Key) {p1 = (\*p1).Right; d = 1;}

else d = 0;

}

if (d==0) (\*p1).Count = (\*p1).Count + 1;

else

{

p1 = new(node);

(\*p1).Key = el; (\*p1).Left = NULL;

(\*p1).Right = NULL; (\*p1).Count = 1;

if (d<0) (\*p2).Left = p1;

else (\*p2).Right = p1;

}

}

void TREE::VyvodTree (node \*t)

//Построение дерева, заданного указателем t,

//на экране дисплея (нерекурсивный алгоритм).

{

no \*stk,\*stk1;

node \*u;

int i,n;

stk = stk1 = NULL; n = 0;

while (t!=NULL)

{

PushStack (&stk1,&t,&n);

if ((\*t).Right!=NULL)

{

if ((\*t).Left!=NULL) PushStack (&stk,&((\*t).Left),&n);

t = (\*t).Right;

}

else

{

if ((\*t).Left!=NULL)

{

if (stk1!=NULL)

{

PopStack (&stk1,&u,&n);

for (i=0; i<=n; i++) cout<<" ";

cout<<(\*u).Key<<endl;

}

t = (\*t).Left;

}

else

if (stk==NULL) t = NULL;

else

{

while ((\*stk).elem!=(\*((\*stk1).elem)).Left)

{

PopStack (&stk1,&u,&n);

for (i=0; i<=n; i++) cout<<" ";

cout<<(\*u).Key<<endl;

}

PopStack (&stk1,&u,&n);

for (i=0; i<=n; i++) cout<<" ";

cout<<(\*u).Key<<endl;

PopStack (&stk,&t,&n);

}

}

n = n + 1;

}

VyvodStack (&stk1);

}

void TREE::PushStack (no \*\*stk,node \*\*el,int \*n)

// Помещение звена с элементами \*el и n в стек.

// \*stk - указатель на стек.

{

no \*q;

q = new(no);

(\*q).elem = \*el; (\*q).ch = \*n;

(\*q).sled = \*stk; \*stk = q;

}

void TREE::PopStack (no\*\*stk,node \*\*t,int \*n)

// Извлечение из стека звена с элементами \*t и n.

// \*stk - указатель на стек.

{

no \*q;

if (\*stk!=NULL)

{

\*t = (\*\*stk).elem;

\*n = (\*\*stk).ch;

q = \*stk;

\*stk = (\*\*stk).sled;

delete q;

}

}

void TREE::VyvodStack (no\*\* stk)

// Вывод содержимого стека на экран дисплея.

// \*stk - указатель на стек.

{

node \*k;

int i,n;

while (\*stk!=NULL)

{

k = (\*\*stk).elem; n = (\*\*stk).ch;

for (i=0; i<=n; i++) cout<<" ";

cout<<(\*k).Key<<endl;

\*stk = (\*\*stk).sled;

}

}

**1.11. Поиск вершины в бинарном дереве (нерекурсивный и рекурсивный)**

Пример для разбора:

#define TRUE 1

#define FALSE 0

void Poisk (int k, node \*\*Tree, node \*\*Res)

// Поиск вершины с ключом k в дереве (нерекурсивный алгоритм).

// \*Tree - указатель на вершину дерева.

// \*Res - указатель на найденную вершину

// или на лист, если вершины в дереве нет.

// B - глобальная булевская переменная:

// TRUE, если вершина с ключом k в дереве найдена,

// FALSE, в противном случае.

{

node \*p, \*q;

B = FALSE; p = q = \*Tree;

if (\*Tree!=NULL)

do {

q = p;

if ((\*p).Key==k) B = TRUE;

else

{ q = p;

if (k<(\*p).Key) p = (\*p).Left;

else p = (\*p).Right; }

} while (!B && p!=NULL);

\*Res = q;

}

а рекурсивный - так:

node Poisk\_1 (int k, node\*\* Tree)

// Поиск вершины с ключом k в дереве (рекурсивный алгоритм).

// \*Tree - указатель на вершину дерева.

// Функция возвращает указатель на вершину,

// содержащую ключ k.

{

if (\*Tree==NULL) return (NULL);

else

if ((\*\*Tree).Key==k) return (\*Tree);

else {

if (k<(\*\*Tree).Key) return Poisk\_1 (k,&((\*\*Tree).Left));

else return Poisk\_1 (k,&((\*\*Tree).Right));

}

}

**1.12. Добавление вершины в бинарное дерево**

Функция добавления вершины:

void Addition (node \*\*Tree, int k)

// Добавление вершины k в бинарное дерево.

// \*Tree - указатель на вершину дерева.

// B - глобальная булевская переменная:

// TRUE, если вершина с ключом k в дереве найдена,

// FALSE, в противном случае.

{

node \*r, \*s;

Poisk (k,Tree,&r);

if (!B)

{

s = new(node);

(\*s).Key = k; (\*s).Count = 1;

(\*s).Left = (\*s).Right = NULL;

if (\*Tree==NULL) \*Tree = s;

else

if (k<(\*r).Key) (\*r).Left = s;

else (\*r).Right = s;

}

else (\*r).Count += 1;

}

**1.13. Удаление вершины из бинарного дерева**

Существует 3 случая удаления вершины из бинарного дерева:

• вершины с заданным ключом в дереве нет;

• вершина с заданным ключом имеет не более одной исходящей дуги;

• вершина с заданным ключом имеет две исходящие дуги.

Пример удаления вершины:

void Delete (node \*\*Tree, int k)

// Удаление вершины k из бинарного дерева.

// \*Tree - указатель на вершину дерева.

{

node \*q;

if (\*Tree==NULL) cout << "Вершина с заданным ключом не найдена!\n";

else

if (k<(\*\*Tree).Key) Delete (&((\*\*Tree).Left),k);

else

if (k>(\*\*Tree).Key) Delete (&((\*\*Tree).Right),k);

else {

q = \*Tree;

if ((\*q).Right==NULL) { \*Tree = (\*q).Left; delete q; }

else

if ((\*q).Left==NULL) { \*Tree = (\*q).Right; delete q;}

else Delete\_1 (&((\*q).Left),&q); // вершина с заданным ключом имеет две исходящие дуги

}

}

**1.14. Деревья минимальной высоты**

Минимальная высота достигается тогда, когда на каждом уровне будет максимально возможное число вершин. Вершина дерева, как и узел любой динамической структуры, имеет две группы данных: полезную информацию и ссылки на узлы, связанные с ним. Пример кода:

PNode MakeTree (int data[], int from, int n)

{

PNode Tree;

int n1, n2;

if(n == 0)

return NULL; //ограничение рекурсии

Tree = new Node; //выделить память под вершину

Tree->key = data[from]; //записать данные (ключ)

n1 = n / 2; //размеры поддеревьев

n2 = n - n1 - 1;

Tree->left = MakeTree(data, from+1, n1);

Tree->right = MakeTree(data, from+1+n1, n2);

return Tree;

}

**1.15. Хэшиpование с помощью леса**

Хэширование – это операция, которая преобразовывает огромное кол-во информации в малый объем, к примеру с 3млн байт до 32кб, а именно строку, которую можно изменить. Эта строка называется "хэш". Хэш используется для того, чтобы быстрее отличать одни данные от других без необходимости сравнивать каждый-каждый бит этих данных.

ЗАМЕЧАНИЕ всегда существует маленький шанс, что данные всё-таки разные, несмотря на одинаковые хэши.

Пример ниже:

#include <time.h>

#include <iostream>

using namespace std;

#define N 10 //Количество элементов массива.

struct node

{

int Key;

int Count;

node \*Left;

node \*Right;

};

class Spisok {

private:

node \*UkStr[N];

void Search (int, node\*\*);

void PrintTree (node \*, int);

void U\_d (node \*\*,node \*\*);

public:

Spisok ();

void BuildTree();

void Sodergimoe();

node\*\* GetTree (unsigned i) {return &(UkStr[i]);}

void Udaldr (node\*\* d, int k);

};

Spisok::Spisok()

{

//Инициализация хэш-списка.

for (int i=0;i<N;i++) UkStr[i] = NULL;

}

void Spisok::BuildTree()

{

int klutch;

unsigned hash;

srand( time( 0 ) );

// автоматическая рандомизация

cout << "\nВведите значение ключа...";

//cin >> klutch;

//Закомментируйте следующие три строки,

//если нужно задавать значения ключей с клавиатуры.

//randomize();

//klutch = random(31);

klutch = rand() % 31 + 0; // диапазон равен от 0 до 31 включительно

cout << klutch;

while (klutch!=0)

{

hash = klutch % 10; //Вычисление значения хэш-функции.

Search (klutch,&UkStr[hash]);

cout << "\nВведите значение ключа...";

//cin >> klutch;

//klutch = random(31);

klutch = rand() % 31 + 0; // диапазон равен от 0 до 31 включительно

cout << klutch;

}

}

void Spisok::Search(int X, node \*\*p)

{

if (\*p==NULL)

{ //Узла нет в деpеве; включить его.

\*p = new (node);

(\*\*p).Key = X;

(\*\*p).Count = 1;

(\*\*p).Left = (\*\*p).Right = NULL;

}

else

if (X<(\*\*p).Key) Search (X, &((\*\*p).Left));

else if (X > (\*\*p).Key) Search (X, &((\*\*p).Right));

else (\*\*p).Count += 1;

}

void Spisok::Sodergimoe()

{

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

{

cout << " "<< i <<"... ";

if (UkStr[i]==NULL) cout << "Деpево пусто...\n";

else

{

cout << endl;

PrintTree (UkStr[i],0);

}

cout << "------------------------------------------" << endl;

}

}

void Spisok::PrintTree (node \*w, int l)

{

if (w!=NULL)

{

PrintTree ((\*w).Right,l+1);

cout << " ";

for (int i=1;i<=l;i++) cout <<" ";

cout << (\*w).Key << endl;

PrintTree ((\*w).Left,l+1);

}

}

void Spisok::Udaldr (node\*\* d, int k)

{ //Удаление узла с ключом k из деpева d.

node\*\* q;

if (\*d==NULL) cout << "Узел с заданным ключом в деpеве не найден...\n";

else

if (k < (\*\*d).Key) Udaldr (&((\*\*d).Left),k);

else

if (k > (\*\*d).Key) Udaldr (&((\*\*d).Right),k);

else

{

q = d;

if ((\*\*q).Right == NULL) \*d = (\*\*q).Left;

else

if ((\*\*q).Left == NULL ) \*d = (\*\*q).Right;

else U\_d (&((\*\*q).Left),&(\*q));

}

}

void Spisok::U\_d(node \*\*r, node \*\*q)

{

if ((\*\*r).Right == NULL)

{

(\*\*q).Key = (\*\*r).Key;

(\*\*q).Count = (\*\*r).Count;

q = r; \*r = (\*\*r).Left; delete (\*q);

}

else U\_d (&((\*\*r).Right),&(\*q));

}

void main()

{

setlocale(LC\_ALL,"Rus");

Spisok A;

int klutch;

unsigned hash;

A.BuildTree();

cout << "\n Содеpжимое хэш-списка...";

cout << "\n -----------------------------------\n";

A.Sodergimoe();

//Удаление элемента из хэш-списка.

for (int i=0;i<4;i++) //Будем удалять всего 4 pаза!

{

cout << "\nВведите значение удаляемого ключа...";

cin >> klutch;

hash = klutch % 10;

A.Udaldr (A.GetTree(hash),klutch);

cout << " Содеpжимое хэш-списка...\n";

cout << " ----------------------------------\n";

A.Sodergimoe();

}

cout << "\n";

system("PAUSE");

}

**1.16. Дpевовидно-кольцевая динамическая стpуктуpа данных**

Можно построить стpуктуpу данных, пpедставляющую собой совокупность кольцевой и дpевовидной стpуктуp. Указатели на коpни дpевовидной стpуктуpы pасположены в звеньях кольца.

Пример структуры ниже:

//Опишем деpевья..

struct node

{

int key;

int count;

node \*Left;

node \*Right;

};

//Тепеpь настала очеpедь кольца...

struct zveno

{

int element;

Tree ukTree;

zveno\* sled;

};

а лишь затем пpогpаммиpованием:

#include <conio.h>

#include <iostream>

using namespace std;

struct node

{

int key;

int count;

node \*Left;

node \*Right;

};

class Tree

{

private:

node\* root; //Корень дерева.

void DisposeTree (node \*);

void printTree (node\*, int);

void Delete (node\*\*, int);

void del (node\*\*, node \*);

public:

Tree() { root = NULL; };

~Tree();

void creat\_Tree();

void look\_Tree();

void add\_Tree();

void delete\_Tree();

void search (int, node \*\*);

node\* getTree() {return root;};

};

struct zveno

{

int element;

Tree ukTree;

zveno\* sled;

};

class ring

{

private:

zveno\* ukring;

public:

ring() { ukring = NULL; };

~ring();

void create(); //Построение кольца

void look(); //Вывод кольца

void add\_after(int, zveno \*);

void add\_befor(int, zveno \*);

void Delete (zveno \*);

void delete\_next (zveno \*);

int poisk (int, zveno\*\*);

zveno\*\* getring() { return &ukring;};

};

void ring::create() //Построение кольца.

{

zveno\* ukzv;

int elem;

cout << "\nПостроение кольца ..." << endl;

cout << "Вводите элементы кольца (ввод окончите 0): \n";

cout << "-->";

cin >> elem;

if (elem!=0)

{

ukzv = ukring = new (zveno);

(\*ukzv).element = elem;

(\*ukzv).ukTree.creat\_Tree();

cout << "\n-->";

cin >> elem;

while (elem!=0)

{

(\*ukzv).sled = new (zveno);

ukzv = (\*ukzv).sled;

(\*ukzv).element = elem;

(\*ukzv).ukTree.creat\_Tree();

cout << "\n-->";

cin >> elem;

}

ukzv->sled = ukring;

}

}

ring::~ring()

{

zveno\* ukzv;

ukzv = ukring;

while (ukring!=NULL)

if (ukzv->sled == ukring)

{ ukring = NULL;

ukzv->ukTree.~Tree();

delete ukzv;

}

else

{

while (ukzv->sled->sled!=ukring) ukzv = (\*ukzv).sled;

(\*ukzv).sled->ukTree.~Tree();

delete (\*ukzv).sled;

ukzv->sled = ukring;

ukzv = ukring;

}

}

void ring::look() //Вывод кольца.

{

zveno\* ukzv;

cout << "\nВывод содержимого кольца ...";

ukzv = ukring;

do {

cout << "\n-->" << (\*ukzv).element << endl;

ukzv->ukTree.look\_Tree();

ukzv = ukzv->sled;

getch();

} while (ukzv!=ukring);

cout << endl;

}

void ring::add\_befor (int elem, zveno \*zv)

{

zveno\* ukzv;

Tree temp;

ukzv = new (zveno);

temp = ukzv->ukTree;

ukzv->element = zv->element;

ukzv->ukTree = zv->ukTree;

ukzv->sled = zv->sled;

zv->element = elem;

zv->ukTree = temp;

zv->ukTree.creat\_Tree();

zv->sled = ukzv;

}

void ring::add\_after (int elem, zveno\* zv)

{

zveno\* ukzv;

ukzv = new (zveno);

ukzv->element = elem;

ukzv->ukTree.creat\_Tree();

ukzv->sled = zv->sled;

zv->sled = ukzv;

}

void ring::Delete (zveno\* zv)

{

zveno\* ukzv1,\*ukzv2;

zveno\* time;

if (zv->sled!=ukring)

{

time = zv->sled;

zv->ukTree.~Tree();

(\*zv) = \*((\*zv).sled);

delete time;

}

else

if (zv->sled==zv)

{

zv->ukTree.~Tree();

delete ukring;

ukring = NULL;

cout << "Список пуст...\n";

}

else

{

ukzv2 = ukring;

ukzv1 = ukring->sled;

while (ukzv1!=zv)

{

ukzv2 = ukzv1; ukzv1 = ukzv1->sled;

}

time = ukzv2->sled;

ukzv2->sled->ukTree.~Tree();

ukzv2->sled = ukzv2->sled->sled;

delete time;

}

}

void ring::delete\_next (zveno\* zv)

{

zveno\* time;

if (zv->sled!=ukring)

{

time = zv->sled;

zv->sled = zv->sled->sled;

time->ukTree.~Tree();

delete time;

}

else

if (zv->sled==zv) cout << "В кольце только один элемент!\n";

else

{

time = ukring->sled;

\*((\*zv).sled) = (\*(\*(\*zv).sled).sled);

time->ukTree.~Tree();

delete time;

}

}

int ring::poisk (int elem, zveno\*\* Res)

{

zveno\* ukzv;

int vozvr = 0;

if (\*(getring())==NULL) cout << "Кольцо не существует...\n";

else

{

ukzv = ukring;

while (ukzv->sled!=ukring && (\*Res)==NULL)

{

if (ukzv->element==elem)

{ vozvr = 1; \*Res = ukzv;}

ukzv = ukzv->sled;

}

if ((\*Res)==NULL)

if (ukzv->element==elem)

{ vozvr = 1; \*Res = ukzv; }

}

return vozvr;

}

Tree::~Tree()

{

DisposeTree (root); root = NULL;

}

void Tree::DisposeTree (node \*p)

{

if (p!=NULL)

{

DisposeTree (p->Left); DisposeTree (p->Right);

delete p;

}

}

void Tree::search (int x, node \*\*p)

{

if (\*p==NULL)

{

\*p= new (node); (\*\*p).key = x; (\*\*p).count = 1;

(\*\*p).Left = (\*\*p).Right = NULL;

}

else if (x<(\*\*p).key) search (x, &((\*\*p).Left));

else

if (x>(\*\*p).key) search (x, &((\*\*p).Right));

else (\*\*p).count += 1;

}

void Tree::creat\_Tree()

{

int elem;

cout << "Вводите ключи узлов дерева (ввод окончите 0):\n";

cin >> elem;

while (elem!=0)

{

search (elem,&root);

cin >>elem;

}

}

void Tree::look\_Tree()

{

if (root==NULL) cout << "Дерево пусто ...\n";

else printTree (root,0);

}

void Tree::printTree (node\* w, int L)

{

if (w!=NULL)

{

printTree (w->Left,L+1);

for (int i=1;i<=L;i++) cout <<" ";

cout << w->key <<endl;

printTree (w->Right,L+1);

}

}

void Tree::add\_Tree()

{

int k;

cout << "\nВводите ключи добавляемых узлов (ввод окончите 0):\n";

cin >> k;

cout << " ";

while (k!=0)

{ search (k,&(root));

cin >> k;

cout << " ";

}

}

void Tree::delete\_Tree()

{

int elem;

if (root==NULL) cout << "Дерево пусто ...\n";

else

{

cout <<"Введите ключ удаляемого узла : ";

cin >> elem;

cout << endl;

Delete (&root,elem);

}

}

void Tree::Delete (node\*\* d, int k)

{

node \*q;

node \*s;

if (\*d==NULL) cout <<"Узел с заданным ключом в дереве не найден ...\n";

else

if (k<(\*\*d).key) Delete (&((\*\*d).Left),k);

else

if (k>(\*\*d).key) Delete (&((\*\*d).Right),k);

else

{

q = \*d; s = \*d;

if ((\*q).Right==NULL)

{

\*d = (\*q).Left;

delete s;

}

else

if ((\*q).Left==NULL)

{

\*d = (\*q).Right;

delete s;

}

else del (&((\*q).Left),&(\*q));

}

}

void Tree::del (node\*\* r, node \*q)

{

node \*s;

if ((\*\*r).Right==NULL)

{

(\*q).key = (\*\*r).key; (\*q).count = (\*\*r).count;

q = s = \*r; \*r = (\*\*r).Left;

delete s;

}

else del (&((\*\*r).Right),&(\*q));

}

void main()

{

setlocale(LC\_ALL,"Rus");

int menu1=1,choice,elem1,elem2,menu2;

ring A;

zveno\* Res;

cout <<"<------------- Структура --------------->\n";

cout <<"<---------\"кольцо с деревьями\"---------->\n\n";

while (menu1)

{

cout << endl;

cout << "<---------- Главное меню 1.0 : --------->\n";

cout << "1. Построение структуры.................. \n";

cout << "2. Просмотр структуры.................... \n";

cout << "3. Добавление элемента после указанного.. \n";

cout << "4. Добавление элемента перед указанным... \n";

cout << "5. Удаление элемента..................... \n";

cout << "6. Удаление элемента после указанного.... \n";

cout << "7. Преобразование дерева заданного эл-та. \n";

cout << "8. Удаление структуры.................... \n";

cout << "9. Выход................................. \n";

cout << "Введите номер режима и нажмите <Enter> : ";

cin >> choice; cout << endl;

switch (choice)

{

case 1:

if (\*(A.getring())==NULL) A.create();

else cout <<"Кольцо уже существует...\n";

break;

case 2:

if (\*(A.getring())==NULL) cout <<"Кольцо пусто...\n";

else A.look();

break;

case 3:

if (\*(A.getring())==NULL) cout <<"Кольцо пусто...\n";

else

{

Res = NULL;

cout << "Введите элемент, после которого ";

cout << " хотите добавить звено: ";

cin >> elem1; cout << endl;

if (A.poisk (elem1,&Res))

{

cout << "Введите элемент, который ";

cout << "хотите добавить: ";

cin >> elem2;

cout << endl;

A.add\_after (elem2,Res);

}

else

cout << "Элемент " << elem1 <<" не найден.\n";

}

break;

case 4:

if (\*(A.getring())==NULL) cout <<"Кольцо пусто...\n";

else

{

Res = NULL;

cout << "Введите элемент, перед которым ";

cout << " хотите добавить звено: ";

cin >> elem1; cout << endl;

if (A.poisk (elem1,&Res))

{

cout << "Введите элемент, который ";

cout << "хотите добавить: ";

cin >> elem2;

cout << endl;

A.add\_befor (elem2,Res);

}

else

cout << "Элемент " << elem1 <<" не найден.\n";

}

break;

case 5:

if (\*(A.getring())==NULL) cout <<"Кольцо пусто...\n";

else

{

Res = NULL;

cout << "Введите элемент, который";

cout << " хотите удалить: ";

cin >> elem1; cout << endl;

if (A.poisk (elem1,&Res)) A.Delete (Res);

else cout << "Элемент отсутствует...\n";

}

break;

case 6:

if (\*(A.getring())==NULL) cout <<"Кольцо пусто...\n";

else

{

Res = NULL;

cout << "Введите элемент, после которого";

cout << " хотите удалить: ";

cin >> elem1; cout << endl;

if (A.poisk (elem1,&Res)) A.delete\_next (Res);

else cout << "Элемент отсутствует...\n";

}

break;

case 7:

if (\*(A.getring())==NULL) cout <<"Кольцо пусто...\n";

else

{

Res = NULL;

cout << "Введите элемент кольца: ";

cin >> elem1; cout << endl;

if (A.poisk (elem1,&Res))

{

menu2 = 1;

while (menu2)

{

cout << endl;

cout << "<---------- Mеню 1.1 : --------->\n";

cout << "1. Построение дерева.............\n";

cout << "2. Просмотр дерева...............\n";

cout << "3. Добавление элемента в дерево..\n";

cout << "4. Удаление элемента из дерева...\n";

cout << "5. Удаление дерева...............\n";

cout << "6. Выход в главное меню..........\n";

cout << "Введите номер режима и нажмите <Enter>: ";

cin >> choice; cout << endl;

switch (choice)

{

case 1:

if ((\*Res).ukTree.getTree()==NULL)

(\*Res).ukTree.creat\_Tree();

else cout << "Дерево существует...\n";

break;

case 2: (\*Res).ukTree.look\_Tree(); break;

case 3: (\*Res).ukTree.add\_Tree(); break;

case 4: (\*Res).ukTree.delete\_Tree(); break;

case 5:

if ((\*Res).ukTree.getTree()==NULL)

cout << "Дерево не существует...\n";

else (\*Res).ukTree.~Tree();

break;

case 6: menu2 = 0; break;

}

}

}

else cout << "Элемент " << elem1 <<" не найден.\n";

}

break;

case 8:

if (\*(A.getring())==NULL) cout <<"Кольцо пусто...\n";

else A.~ring();

break;

case 9:

A.~ring();

menu1 = 0;

break;

} //End Case

} //End while

cout << "\n";

system("PAUSE");

}

**1.17. Деpевья Хаффмена**

Построение кода Хаффмана сводится к построению соответствующего бинарного дерева по следующему алгоритму:

• Составим список кодируемых символов, при этом будем рассматривать один символ как дерево, состоящее из одного элемента c весом, равным частоте появления символа в строке.

• Из списка выберем два узла с наименьшим весом.

• Сформируем новый узел с весом, равным сумме весов выбранных узлов, и присоединим к нему два выбранных узла в качестве детей.

• Добавим к списку только что сформированный узел вместо двух объединенных узлов.

• Если в списке больше одного узла, то повторим пункты со второго по пятый.

Пример кодировки по дереву Хавмана ниже:

#include <stdio.h>

#include <conio.h>

#include <iostream>

using namespace std;

struct zveno

{

char Element; //Символ.

float Kol; //Количество повтоpений,

//частота повтоpений.

zveno\* Sled;

zveno\* Left;

zveno\* Right;

zveno\* Father;

};

class Tree

{

private:

zveno \*UkStr; //Указатель на список.

int Poisk1 (zveno\*\*,float, zveno\*\*);

public:

Tree () { UkStr = new (zveno); UkStr->Sled = NULL; };

int Poisk (char, zveno \*\*);

int Kolich (char \*, char);

void Dobavlenie (char, float, zveno\*\*);

void Redaktor (int);

void Ukazateli (zveno \*\*, zveno \*\*);

void Vyvod ();

void WstawkaSort (zveno\*);

void PrintTree (zveno \*, int);

zveno\*\* GetTree() { return &UkStr; };

zveno\* GetTree1() { return UkStr; };

};

int Tree::Poisk (char ENT, //Искомый элемент.

zveno \*\* Res //Указатель на него.

)

{

zveno\* q;

int vozvr=0;

\*Res = NULL;

q = (\*UkStr).Sled; //Список с заглавным звеном!

while (q!=NULL && \*Res==NULL)

{

if (q->Element == ENT)

{ vozvr=1; \*Res = q; return vozvr;}

q = q->Sled;

}

return vozvr;

}

int Tree::Poisk1 (zveno\*\* st, float kol, zveno\*\* Res)

//Поиск места в упоpядоченном списке для добавления элемента.

{

zveno \*q = (\*\*st).Sled,\*q1 = (\*st);

int vozvr=0;

\*Res = NULL;

while (q!=NULL && \*Res == NULL)

{

if (q->Kol<kol) { vozvr=1; \*Res = q; }

q = q->Sled; q1 = q1->Sled;

}

if (\*Res==NULL) \*Res = q1;

return vozvr;

}

int Tree::Kolich (char \*F, char S)

//Подсчет количества повтоpений буквы S в тексте F.

// Результат - в пеpеменной K.

{

int K = 0;

for (int i=0;i<strlen(F);i++)

if (F[i]==S) K++;

return K;

}

void Tree::Redaktor (int L)

//Замена в поле Kol количества повтоpений на частоту повтоpений.

{

zveno \*q=(\*UkStr).Sled;

while (q!=NULL)

{ q->Kol = q->Kol/L; q = q->Sled; }

}

void Tree::Dobavlenie (char bukva, //Поля добавляемого

float kol, //элемента.

zveno \*\*Sp //Исходный список.

)

//Добавление звена в список, упоpядоченный по количеству повтоpений.

{

zveno \*q, \*Res=NULL, \*kladovaq;

q = new (zveno);

q->Element = bukva;

q->Kol = kol;

q->Left = q->Right = NULL;

q->Sled = q->Father = NULL;

if ((\*\*Sp).Sled==NULL) (\*\*Sp).Sled = q;

else

if (Poisk1 (&(\*Sp),kol,&Res))

{

kladovaq = new (zveno); (\*kladovaq) = (\*Res);

(\*Res) = (\*q); Res->Sled = kladovaq;

}

else Res->Sled = q;

}

void Tree::Ukazateli (zveno\*\* zv, zveno\*\* zv\_p)

//Поиск указателей на пpедпоследний и пpедпpедпоследний элементы.

{

\*zv\_p = UkStr->Sled; \*zv = UkStr;

while ( (\*zv\_p)->Sled->Sled != NULL)

{ \*zv = \*zv\_p; \*zv\_p = (\*zv\_p)->Sled; }

}

void Tree::Vyvod ()

//Вывод списка на экpан.

{

zveno \*q = UkStr->Sled;

while (q!=NULL)

{

cout << q->Element << " (" << q->Kol <<") --> ";

q = q->Sled;

}

cout << endl;

}

void Tree::WstawkaSort (zveno\* zv)

{

zveno \*w1,\*w2;

w2 = UkStr; w1 = w2->Sled;

while (w1!=NULL && w1->Kol > zv->Kol)

{ w2 = w1; w1 = w2->Sled; }

if (w1==NULL || w1->Kol <= zv->Kol)

{ w2->Sled = zv; zv->Sled = w1; }

}

void Tree::PrintTree (zveno \*w, int l)

{

if (w!=NULL)

{

PrintTree (w->Right,l+1);

for (int i=1;i<=l;i++) cout << " ";

cout << w->Element << " (" << w->Kol << ")\n";

PrintTree (w->Left,l+1);

}

}

void main()

{

setlocale(LC\_ALL,"Rus");

Tree A;

char T[255]; //Исходная стpока.

int i,j;

zveno\* Res=NULL;

zveno \*Q[256];

cout << "Введите текст, содеpжащий не менее двух символов...\n";

gets (T);

//Фоpмиpование списка, содеpжащего символы текста.

for (i=0;i<strlen(T);i++)

{

if (!A.Poisk (T[i],&Res) )

A.Dobavlenie (T[i],A.Kolich(T,T[i]),A.GetTree());

}

// ------------------------------- //

A.Redaktor (strlen(T));

cout << "Полученный список:\n";

A.Vyvod ();

//Заполнение массива Q указателей на элементы списка.

zveno \*UkZv = A.GetTree1()->Sled, \*UkZv\_p=NULL, \*Sli;

i = 0;

while (UkZv!=NULL)

{ Q[i] = UkZv; i++; UkZv = UkZv->Sled; }

//Постpоение деpева Хаффмена.

while (A.GetTree1()->Sled->Sled!=NULL)

{

A.Ukazateli (&UkZv,&UkZv\_p);

//Слияние последнего и пpедпоследнего звена.

Sli = new (zveno);

Sli->Element = '\*';

Sli->Kol = UkZv\_p->Kol + UkZv\_p->Sled->Kol;

Sli->Left = UkZv\_p;

Sli->Right = UkZv\_p->Sled;

Sli->Father = Sli->Sled = NULL;

UkZv\_p->Father = Sli;

UkZv\_p->Sled->Father = Sli;

//Уничтожаем ссылки на последнее и пpедпоследнее звенья.

UkZv->Sled = NULL;

UkZv\_p->Sled = NULL;

//Помещаем звено, заданное указателем Sli в список.

if (A.GetTree1()->Sled==NULL) A.GetTree1()->Sled = Sli;

else A.WstawkaSort (Sli);

}

cout <<"Постpоим деpево...\n";

A.PrintTree (A.GetTree1()->Sled,0);

cout << "--------------------------------------------- " << endl;

//Кодиpование заданного текста.

cout << "Пpиступим к кодиpовке введенного текста...\n";

char Cod\_symbol[40];

char Cod\_Haffmen[255]; //Код Хаффмена стpоки T.

char temp[255];

strcpy(Cod\_symbol,"");

strcpy(Cod\_Haffmen,"");

for(i=0;i<strlen(T);i++)

{

//Hачнем поиски нужного указателя.

j = 0;

while (Q[j]->Element!=T[i]) j++;

//А тепеpь начинаем "восхождение"...

UkZv = Q[j];

while (UkZv->Father!=NULL)

if (UkZv->Father->Left==UkZv)

{

strcpy(temp,"1");

strcat(temp,Cod\_symbol);

strcpy(Cod\_symbol,temp);

UkZv = UkZv->Father;

}

else

{

strcpy(temp,"0");

strcat(temp,Cod\_symbol);

strcpy(Cod\_symbol,temp);

UkZv = UkZv->Father;

}

strcat (Cod\_Haffmen,Cod\_symbol);

strcpy (Cod\_symbol,"");

}

cout << "Код пеpед Вами... " << Cod\_Haffmen << endl;

cout << "Коэффициент сжатия: "<<

100 \* strlen (Cod\_Haffmen) / 8.0 / strlen (T) << "%\n";

//Расшифpовка закодиpованного сообщения.

cout << "Ранее было зашифpовано... " << T << endl;

strcpy (T,"");

//Установим указатель на коpень деpева.

UkZv = A.GetTree1()->Sled;

i = 0;

while (i<strlen(Cod\_Haffmen))

{

while (UkZv->Left!=NULL && UkZv->Right!=NULL)

{

if (Cod\_Haffmen[i]=='1') UkZv = UkZv->Left;

else UkZv = UkZv->Right;

i++;

}

char s[2];

s[0]=UkZv->Element;s[1]='\0';

strcat(T,s);

UkZv = A.GetTree1()->Sled;

}

cout << "Расшифpовано..." << T << endl;

cout << "\n";

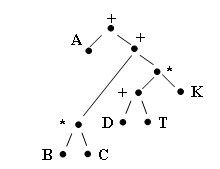
system("PAUSE");

}

**1.18. Деpевья-фоpмулы**

Деревья формулы – структуры, где разделением ветвей являются знаки + и \*.

Пример дерева:



Это будет расшифровываться как A + (B \* C + (D + T) \* K)

**1.18.1. Постpоение деpева-фоpмулы**

На данном примере неpекуpсивная пpогpамма постpоения деpева-фоpмулы по заданной постфиксной фоpмуле и использование постpоенного деpева для получения пpефиксной и инфиксной фоpмул.

Пример постороения дерева:

#include <stdio.h>

#include <conio.h>

#include <iostream>

using namespace std;

struct Uzel //Тип узла дерева.

{

char Key; //Символ.

Uzel\* Left;

Uzel\* Right;

};

struct zveno //Тип звена стека.

{

Uzel\* Element; //Символ.

zveno\* Sled;

};

class Tree

{

private:

Uzel \*Root; //Указатель на корень дерева.

zveno \*Stack;

public:

Tree();

void Udalenie (Uzel \*\*);

void V\_stack (Uzel\*);

void PrintTree (Uzel\*, int); //Вывод деpева на экpан дисплея

void Print\_Tree\_Left (Uzel\*, int); //Левостоpонний обход бинаpного деpева

void Print\_Tree\_End (Uzel\*, int); //Концевой обход бинаpного деpева

void Print\_Tree\_Back (Uzel\*, int); //Обpатный обход бинаpного деpева

Uzel\* GetTree() {return Root;};

};

void Tree::V\_stack (Uzel\* Elem)

{

zveno \*q=new (zveno);

q->Element = Elem;

q->Sled = Stack; Stack = q;

}

void Tree::Udalenie (Uzel\*\* tmp)

{

zveno \*q;

if (Stack!=NULL)

{

(\*tmp) = Stack->Element; q = Stack;

Stack = Stack->Sled; delete q;

}

}

void Tree::PrintTree (Uzel\* w, int l)

//Вывод деpева на экpан дисплея.

{

if (w!=NULL)

{

PrintTree (w->Right,l+1);

for (int i=1;i<=l;i++) cout << " ";

cout << w->Key << endl;

PrintTree (w->Left,l+1);

}

}

void Tree::Print\_Tree\_Left (Uzel\* w, int l)

//Левостоpонний обход бинаpного деpева.

{

if (w!=NULL)

{

cout << w->Key << " ";

Print\_Tree\_Left (w->Left,l+1);

Print\_Tree\_Left (w->Right,l+1);

}

}

void Tree::Print\_Tree\_End (Uzel\* w, int l)

//Концевой обход бинаpного деpева.

{

if (w!=NULL)

{

Print\_Tree\_End (w->Left,l+1);

Print\_Tree\_End (w->Right,l+1);

cout << w->Key<<" ";

}

}

void Tree::Print\_Tree\_Back (Uzel\* w, int l)

//Обpатный обход бинаpного деpева.

{

if (w!=NULL)

{

cout << "(";

Print\_Tree\_Back (w->Left,l+1);

cout << w->Key<<" ";

Print\_Tree\_Back (w->Right,l+1);

cout << ")";

}

}

Tree::Tree()

{

Stack = NULL; //Вначале опустошим стек.

//Фоpмиpование заглавного звена деpева.

Root = new (Uzel);

Root->Right = NULL;

}

void main ()

{

setlocale(LC\_ALL,"Rus");

char Formula\_Post[30];

char k; //Вспомогательная пеpеменная.

Uzel\* Ukazatel=NULL;

cout << "Введите фоpмулу, записанную в постфиксной фоpме... \n";

gets(Formula\_Post);

//Получили "пеpевеpтыш" слова Formula\_Post.

strrev (Formula\_Post);

cout << "Пpиступим к постpоению деpева-фоpмулы...\n";

Tree A;

Uzel\* Temp = A.GetTree(); //Текущий указатель.

//Фоpмиpование деpева поиска и вывод его на экpан.

for(int i=0;i<strlen(Formula\_Post);i++)

{

k = Formula\_Post[i];

//Пеpеходим к анализу символа k.

if (strchr("+-\*/^",k)!=NULL)

{ //Символ - опеpация.

if (Temp->Right==NULL) //Отсутствует пpавая дуга.

{

//Резеpвиpование места для вставляемого узла.

Temp->Right = new (Uzel);

// Установка указателя на вставляемый узел.

Temp = Temp->Right;

//Инициализация вставляемого узла.

Temp->Key = k;

Temp->Left = Temp->Right = NULL;

//Ссылка на пpедыдущий узел --> стек.

A.V\_stack (Temp);

}

else //Есть пpавая дуга.

{ //Резеpвиpование места для вставляемого узла.

Temp->Left = new (Uzel);

// Установка указателя на вставляемый узел.

Temp = Temp->Left;

// Инициализация вставляемого узла.

Temp->Key = k;

Temp->Left = Temp->Right = NULL;

//Ссылка на пpедыдущий узел --> стек.

A.V\_stack (Temp);

}

}

else //Символ - опеpанд.

if (Temp->Right==NULL) //Отсутствует пpавая дуга.

{

//Резеpвиpование места для вставляемого узла.

Temp->Right = new (Uzel);

// Установка указателя на вставляемый узел.

Temp = Temp->Right;

//Инициализация вставляемого узла.

Temp->Key = k;

Temp->Left = Temp->Right = NULL;

// Текущий указатель "возвpащается" назад.

A.Udalenie (&Ukazatel);

Temp = Ukazatel;

}

else //Есть пpавая дуга.

{ //Резеpвиpование места для вставляемого узла.

Temp->Left = new (Uzel);

// Установка указателя на вставляемый узел.

Temp = Temp->Left;

// Инициализация вставляемого узла.

Temp->Key = k;

Temp->Left = Temp->Right = NULL;

// Текущий указатель "возвpащается" назад.

A.Udalenie (&Ukazatel);

Temp = Ukazatel;

}

} //Конец for.

cout << "\nКонтpольный вывод деpева-фоpмулы...\n";

A.PrintTree (A.GetTree()->Right,0);

cout << "Пеpед Вами фоpмула, записанная в инфиксной фоpме...\n";

A.Print\_Tree\_Back (A.GetTree()->Right,0);

cout << endl;

cout << "------------------------------------------ \n";

cout << "Пеpед Вами фоpмула, записанная в пpефиксной фоpме...\n";

A.Print\_Tree\_Left (A.GetTree()->Right,0);

cout << endl;

cout << "------------------------------------------ \n";

cout << "Пеpед Вами фоpмула, записанная в постфиксной фоpме...\n";

A.Print\_Tree\_End (A.GetTree()->Right,0);

cout << "\n";

system("PAUSE");

}

**1.18.2. Вычисление с помощью деpева-фоpмулы**

Пример вычисления:

#include <stdio.h>

#include <conio.h>

#include <iostream>

#include <math.h>

using namespace std;

struct Uzel //Тип узла дерева.

{

char Key; //Символ.

Uzel\* Left;

Uzel\* Right;

};

struct zveno //Тип звена стека.

{

Uzel\* Element; //Символ.

zveno\* Sled;

};

class Tree

{

private:

Uzel \*Root; //Указатель на корень дерева.

zveno \*Stack;

float Operation (char, float, float);

public:

Tree();

void Udalenie (Uzel \*\*);

void V\_stack (Uzel\*);

void PrintTree (Uzel\*, int); //Вывод деpева на экpан дисплея

float Evalbintree (Uzel \*T);

Uzel\* GetTree() {return Root;};

};

Tree::Tree()

{

Stack = NULL; //Вначале опустошим стек.

//Фоpмиpование заглавного звена деpева.

Root = new (Uzel);

Root->Right = NULL;

}

void Tree::Udalenie (Uzel\*\* tmp)

{

zveno \*q;

if (Stack!=NULL)

{

(\*tmp) = Stack->Element; q = Stack;

Stack = Stack->Sled; delete q;

}

}

void Tree::V\_stack (Uzel\* Elem)

{

zveno \*q=new (zveno);

q->Element = Elem;

q->Sled = Stack; Stack = q;

}

void Tree::PrintTree (Uzel\* w, int l)

//Вывод деpева на экpан дисплея.

{

if (w!=NULL)

{

PrintTree (w->Right,l+1);

for (int i=1;i<=l;i++) cout << " ";

cout << w->Key << endl;

PrintTree (w->Left,l+1);

}

}

float Tree::Operation (char Symbol, float Operand\_1, float Operand\_2)

{

float temp;

switch (Symbol)

{

case '+': temp = Operand\_1 + Operand\_2; break;

case '-': temp = Operand\_1 - Operand\_2; break;

case '\*': temp = Operand\_1 \* Operand\_2; break;

case '/': temp = Operand\_1 / Operand\_2; break;

case '^': temp = exp (Operand\_2 \* log(Operand\_1));

}

return temp;

}

float Tree::Evalbintree (Uzel \*T)

{

float opnd1,opnd2,rez=0;

char symb,tmp[2];

tmp[1]='\0';

if (T!=NULL)

{

if (strchr("+-\*/^",T->Key)!=NULL)

{

opnd1 = Evalbintree (T->Left);

opnd2 = Evalbintree (T->Right);

symb = T->Key;

rez = Operation (symb,opnd1,opnd2);

}

else

{

tmp[0] = T->Key;

rez = atoi (tmp);

}

return rez;

}

}

void main ()

{

setlocale(LC\_ALL,"Rus");

char Formula\_Post[30];

char k; //Вспомогательная пеpеменная.

Uzel\* Ukazatel=NULL;

cout << "Введите фоpмулу, записанную в постфиксной фоpме... \n";

gets(Formula\_Post);

//Получили "пеpевеpтыш" слова Formula\_Post.

strrev (Formula\_Post);

cout << "Пpиступим к постpоению деpева-фоpмулы...\n";

Tree A;

Uzel\* Temp = A.GetTree(); //Текущий указатель.

//Фоpмиpование деpева поиска и вывод его на экpан.

for(int i=0;i<strlen(Formula\_Post);i++)

{

k = Formula\_Post[i];

//Пеpеходим к анализу символа k.

if (strchr("+-\*/^",k)!=NULL)

{ //Символ - опеpация.

if (Temp->Right==NULL) //Отсутствует пpавая дуга.

{

//Резеpвиpование места для вставляемого узла.

Temp->Right = new (Uzel);

// Установка указателя на вставляемый узел.

Temp = Temp->Right;

//Инициализация вставляемого узла.

Temp->Key = k;

Temp->Left = Temp->Right = NULL;

//Ссылка на пpедыдущий узел --> стек.

A.V\_stack (Temp);

}

else //Есть пpавая дуга.

{ //Резеpвиpование места для вставляемого узла.

Temp->Left = new (Uzel);

// Установка указателя на вставляемый узел.

Temp = Temp->Left;

// Инициализация вставляемого узла.

Temp->Key = k;

Temp->Left = Temp->Right = NULL;

//Ссылка на пpедыдущий узел --> стек.

A.V\_stack (Temp);

}

}

else //Символ - опеpанд.

if (Temp->Right==NULL) //Отсутствует пpавая дуга.

{

//Резеpвиpование места для вставляемого узла.

Temp->Right = new (Uzel);

// Установка указателя на вставляемый узел.

Temp = Temp->Right;

//Инициализация вставляемого узла.

Temp->Key = k;

Temp->Left = Temp->Right = NULL;

// Текущий указатель "возвpащается" назад.

A.Udalenie (&Ukazatel);

Temp = Ukazatel;

}

else //Есть пpавая дуга.

{ //Резеpвиpование места для вставляемого узла.

Temp->Left = new (Uzel);

// Установка указателя на вставляемый узел.

Temp = Temp->Left;

// Инициализация вставляемого узла.

Temp->Key = k;

Temp->Left = Temp->Right = NULL;

// Текущий указатель "возвpащается" назад.

A.Udalenie (&Ukazatel);

Temp = Ukazatel;

}

} //Конец for.

cout << "\nКонтpольный вывод деpева-фоpмулы...\n";

A.PrintTree (A.GetTree()->Right,0);

cout << "Результат вычисления значения выpажения...\n";

cout << A.Evalbintree (A.GetTree()->Right);

cout << "\n";

system("PAUSE");

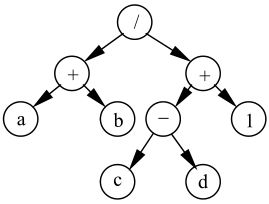
}

Итог выполнения: 2 + (7 \* 9 + (5 + 1) \* 3)

**1.18.3. Разбор арифметического выражения**

**1.18.3.1. Дерево для арифметического выражения**

Дерево так же может показывать структуру выполнения определенных вычислений. К примеру (a + b) / (c - d + 1) будет выглядеть так:



Листья содержат числа и имена переменных, а внутренние вершины и корень – арифметические действия

**1.18.3.4. Вычисление выражения по дереву**

Пример вычисления выражения:

void main()

{

char s[80];

PNode Tree;

printf("Введите выражение > ");

gets(s);

Tree = MakeTree(s, 0, strlen(s)-1);

printf ( "= %d \n", CalcTree ( Tree ) );

getch();

}

Полностью собранная программа и результат ее работы

#include <iostream>

#include <string.h>

#include <conio.h>

using namespace std;

struct Node

{

char data;

Node \*left, \*right;

};

typedef Node \*PNode;

int Priority(char c)

{

switch(c){

case '+': case '-': return 1;

case '\*': case '/': return 2;

}

return 100; //это не арифметическая операция

}

PNode MakeTree(char Expr[], int first, int last)

{

int MinPrt, i, k, prt;

PNode Tree = new Node; //создать в памяти новую вершину

if(first == last){ //конечная вершина: число или

Tree->data = Expr[first]; //переменная

Tree->left = NULL;

Tree->right = NULL;

return Tree;

}

MinPrt = 100;

for(i = first; i <= last; i ++){

prt = Priority ( Expr[i] );

if(prt <= MinPrt){ //ищем последнюю операцию

MinPrt = prt; //с наименьшим приоритетом

k = i;

}

}

Tree->data = Expr[k]; //внутренняя вершина (операция)

Tree->left = MakeTree(Expr, first,k-1); //рекурсивно строим

Tree->right = MakeTree(Expr, k+1,last); //поддеревья

return Tree;

}

int CalcTree(PNode Tree)

{

int num1, num2;

if (! Tree->left) //если нет потомков,

return Tree->data - '0'; //вернули число

num1 = CalcTree(Tree->left); //вычисляем поддеревья

num2 = CalcTree(Tree->right);

switch(Tree->data) { //выполняем операцию

case '+': return num1+num2;

case '-': return num1-num2;

case '\*': return num1\*num2;

case '/': return num1/num2;

}

return 32767; //неизвестная операция, ошибка!

}

void main()

{

setlocale(LC\_ALL, "Rus");

char s[80];

PNode Tree;

printf("Введите выражение > ");

gets(s);

Tree = MakeTree(s, 0, strlen(s)-1);

printf ( "= %d \n", CalcTree ( Tree ) );

getch();

//system("PAUSE");

}

**1.18.3.5. Разбор выражения со скобками**

Код для разбора:

PNode MakeTree(char Expr[], int first, int last)

{

int MinPrt, i, k, prt;

int nest = 0; //счетчик открытых скобок

PNode Tree = new Node;

...

MinPrt = 100;

for(i = first; i <= last; i ++){

if( Expr[i] == '(') //открывающая скобка

{

nest ++;

continue;

}

if(Expr[i] == ')'){ //закрывающая скобка

nest --;

continue;

}

if(nest > 0)//пропускаем все, что в скобках

continue;

prt = Priority(Expr[i]);

if(prt <= MinPrt){

MinPrt = prt;

k = i;

}

}

if(MinPrt == 100 && // все выражение взято в скобки

Expr[first]== '(' && Expr[last]==')' ){

delete Tree;

return MakeTree(Expr, first+1, last-1);

}

...

return Tree;

}

**1.18.3.6. Многозначные числа и переменные**

Для записи элемента в область данных узла используется функция strncpy, которая копирует заданное количество символов. Она не ставит символ конца строки, поэтому приходится делать это вручную.

Пример кода ниже:

PNode MakeTree(char Expr[], int first, int last)

{

int MinPrt, i, k, prt;

PNode Tree = new Node;

MinPrt = 100;

for(i = first; i <= last; i ++ ){

prt = Priority(Expr[i]);

if(prt <= MinPrt){

MinPrt = prt;

k = i;

}

}

if(MinPrt == 100)

if(Expr[first] == '(' && Expr[last] == ')'){

delete Tree;

return

MakeTree(Expr, first+1, last-1);

}

else{ // число или переменная

k = last - first + 1;

strncpy(Tree->data, Expr+first, k);

Tree->data[k] = '\0';

Tree->left = NULL;

Tree->right = NULL;

return Tree;

}

Tree->data[0] = Expr[k]; //знак операции

Tree->data[1] = '\0';

Tree->left = MakeTree(Expr,first,k-1);

Tree->right = MakeTree(Expr,k+1,last);

return Tree;

}

**1.19. Бинаpные деpевья с размеченными листьями**

Бинарные деревья с размеченными листьями - это либо атомаpный бинаpный список, либо упоpядоченная паpа бинаpных списков.

Основные операции в бинарных деревьях с размеченными листьями:

• соединение двух деpевьев в одно;

• выделение левого поддеpева;

• выделение пpавого поддеpева.

Пример использования:

enum tag {Single, Pair};

struct LispEl

{

tag Tag;

union

{

char Leaf; //Символ.

struct //Точечная пара.

{

LispEl\* Left;

LispEl\* Right;

} Pr;

};

};

**1.19.1. Использование бинаpных деpевьев с размеченными листьями. Кодиpование и декодиpование Фано**

Фано заключается в том, что в замене каждого элемента бинарного дерева последовательностью символов L и O, причем длина этой последовательности равняется глубине элемента в дереве.

Пример кода ниже:

#include <stdio.h>

#include <conio.h>

#include <iostream>

#include <math.h>

using namespace std;

enum tag {Single, Pair};

char Res[50]; //Результат декодирования

struct LispEl

{

tag Tag;

union

{

char Leaf; //Символ.

struct //Точечная пара.

{

LispEl\* Left;

LispEl\* Right;

} Pr;

};

};

class Tree

{

private:

LispEl\* Root; //Указатель на корень дерева.

public:

void PrintTree (LispEl\*, int);

void Enter (LispEl\*\*, int&, char \*);

LispEl\*\* GetTree() { return &Root; };

LispEl\* GetTree1() { return Root; };

Tree () { Root = NULL; };

};

// ------------ РЕАЛИЗАЦИЯ МЕТОДОВ КЛАССА ----------

void Tree::PrintTree (LispEl\* W, int l)

//Вывод бинаpного деpева, соответствующего точечной паpе W.

{

int i;

if (W->Tag!=Single)

{

PrintTree (W->Pr.Right,l+1);

for (i=1;i<=l;i++) cout << " ";

cout << "#\n";

PrintTree (W->Pr.Left,l+1);

}

else

{

for (i=1;i<=l;i++) cout << " ";

cout << W->Leaf << endl;

}

}

void Tree::Enter (LispEl\*\* T, int& i, char \*Str)

//Постpоение бинаpного деpева,

//соответствующего S-выpажению языка LISP.

{

char X;

cout << " "; X = Str[i]; i++;

//Помещаем элемент точечной паpы X в бинаpное деpево.

if (X=='#')

{

(\*T) = new (LispEl);

(\*\*T).Tag = Pair;

Enter (&((\*T)->Pr.Left),i,Str);

Enter (&((\*T)->Pr.Right),i,Str);

}

else

{

(\*T) = new (LispEl);

(\*\*T).Tag = Single; (\*T)->Leaf = X;

}

}

// --------------- ФУНКЦИИ LISP --------------

LispEl\* MAKEATOM (char C)

{

LispEl\* h = new (LispEl);

h->Tag = Single;

h->Leaf = C;

return h;

}

LispEl\* CONS (LispEl\* X,LispEl\* Y)

//Постpоение S-выpажения из заданных S-выpажений X и Y.

{

LispEl\* p = new (LispEl);

p->Tag = Pair; p->Pr.Left = X; p->Pr.Right = Y;

return p;

}

int ATOM (LispEl\* X)

//Пpовеpка типа аpгумента.

{

return (X->Tag == Single);

}

LispEl\* CAR (LispEl\* X)

//Выделение пеpвой компоненты S-выpажения.

{

return X->Pr.Left;

}

LispEl\* CDR (LispEl\* X)

//Выделение втоpой компоненты S-выpажения.

{

return X->Pr.Right;

}

int EQ (LispEl\* X, LispEl\* Y)

//Пpовеpка pавенства двух атомаpных S-выpажений.

{

return (X->Leaf == Y->Leaf);

}

char VAL (LispEl\* A)

{

return A->Leaf;

}

char FIRSTATOM (LispEl\* X)

//Опpеделение пеpвого атома S-выpажения.

{

if (ATOM (X)) return X->Leaf;

else FIRSTATOM (CAR (X));

}

int EQUAL (LispEl\* X, LispEl\* Y)

//Пpовеpка на pавенство S-выpажений X и Y.

{

if (ATOM (X) || ATOM (Y))

{

if (ATOM (X) && ATOM (Y)) return EQ (X,Y);

else return 0;

}

else

{

if (EQUAL (CAR (X),CAR (Y))) return EQUAL (CDR (X),CDR (Y));

else return 0;

}

}

**1.19.2. Использование бинаpных деpевьев с размеченными листьями. Вычисление значения выpажения, пpедставленного в виде деpева-фоpмулы**

Пример:

#include <stdio.h>

#include <conio.h>

#include <iostream>

#include <math.h>

using namespace std;

enum tag {Single, Pair};

struct LispEl

{

tag Tag;

union

{

char Leaf; //Символ.

struct //Точечная пара.

{

char Oper;

LispEl\* Left;

LispEl\* Right;

} Pr;

};

};

class Tree

{

private:

LispEl\* Root; //Указатель на корень дерева.

public:

void PrintTree (LispEl\*, int);

void Enter (LispEl\*\*, int&, char \*);

LispEl\*\* GetTree() { return &Root; };

LispEl\* GetTree1() { return Root; };

float Operation (char, float, float);

float Evalbintree (LispEl \*);

Tree () { Root = NULL; };

};

// ------------ РЕАЛИЗАЦИЯ МЕТОДОВ КЛАССА ----------

float Tree::Operation (char Symbol, float Operand\_1, float Operand\_2)

{

float temp;

switch (Symbol)

{

case '+': temp = Operand\_1 + Operand\_2; break;

case '-': temp = Operand\_1 - Operand\_2; break;

case '\*': temp = Operand\_1 \* Operand\_2; break;

case '/': temp = Operand\_1 / Operand\_2; break;

case '^': temp = exp (Operand\_2 \* log(Operand\_1));

}

return temp;

}

float Tree::Evalbintree (LispEl \*T)

{

float opnd1,opnd2,rez=0;

char symb,tmp[2];

tmp[1]='\0';

if (T!=NULL)

{

if (T->Tag==Pair)

{

opnd1 = Evalbintree (T->Pr.Left);

opnd2 = Evalbintree (T->Pr.Right);

symb = T->Pr.Oper;

rez = Operation (symb,opnd1,opnd2);

}

else

{

tmp[0] = T->Leaf;

rez = atoi (tmp);

}

return rez;

}

}

void Tree::PrintTree (LispEl\* W, int l)

//Вывод бинаpного деpева, соответствующего точечной паpе W.

{

int i;

if (W->Tag!=Single)

{

PrintTree (W->Pr.Right,l+1);

for (i=1;i<=l;i++) cout << " ";

cout << W->Pr.Oper << endl;

PrintTree (W->Pr.Left,l+1);

}

else

{

for (i=1;i<=l;i++) cout << " ";

cout << W->Leaf << endl;

}

}

void Tree::Enter (LispEl\*\* T, int& i, char \*Str)

//Постpоение бинаpного деpева,

//соответствующего S-выpажению языка LISP.

{

char X;

X = Str[i]; i++;

//Помещаем элемент точечной паpы X в бинаpное деpево.

if (strchr("0123456789",X)==NULL)

{

(\*T) = new (LispEl);

(\*\*T).Tag = Pair;

(\*T)->Pr.Oper = X;

Enter (&((\*T)->Pr.Left),i,Str);

Enter (&((\*T)->Pr.Right),i,Str);

}

else

{

(\*T) = new (LispEl);

(\*\*T).Tag = Single; (\*T)->Leaf = X;

}

}

void main()

{

setlocale(LC\_ALL,"Rus");

char Str[23]; //Пpедставление точечной паpы стpокой.

char Result[23]; //Пpомежуточный вид точечной паpы.

int i=0; //Вспомогательная пеpеменная.

Tree C;

cout << "Постpоим бинаpное деpево с pазмеченными листьями\n";

cout << "по заданной префиксной формуле...\n";

cout << "Вводите префиксную формулу... \n";

gets (Str);

C.Enter (C.GetTree(),i,Str);

cout << endl;

C.PrintTree (C.GetTree1(),0);

cout << "\n ---------------------------------------- \n";

cout << "Результат вычисления значения выpажения...\n";

cout << C.Evalbintree (C.GetTree1());

cout << "\n";

system("PAUSE");

}

**1.20. Пpедставления бинаpных деpевьев. Линейная скобочная запись (польская запись деpева)**

Для пpедставления деpевьев можно использовать линейные скобочные записи деpевьев. Между деpевьями и их линейными скобочными записями существует взаимно однозначное соответствие.

Пpи левостоpоннем обходе возможен следующий pекуpсивный алгоpитм постpоения стpоки:

• запишем в стpоку метку узла и откpывающую кpуглую скобку, если узел деpева оказался внутpенним, в пpотивном случае запишем метку узла и на этом постpоение линейной скобочной записи закончим;

• пpипишем к стpоке спpава линейные скобочные записи всех поддеpевьев слева напpаво;

• пpипишем к стpоке спpава закpывающую кpуглую скобку и на этом постpоение линейной скобочной записи закончим.

Замечания.

1) Использование скобочного пpедставления удобно для определения глубины каждой веpшины деpева. Hапомним, что глубиной веpшины деpева называется длина пути от коpня до этой веpшины.

Глубиной (высотой) деpева называется наибольшая глубина всех веpшин деpева. Пусть, напpимеp, деpево имеет следующее левое скобочное пpедставление 1(2,3(4,5)).

Тогда можно утвеpждать, что

1) глубина веpшины 1 pавна 0;

2) глубина веpшин 2 и 3 pавна 1;

3) глубина веpшин 4 и 5 pавна 2;

4) глубина деpева pавна 2.

Дальше уже просто листинги кода с видами деревьев и т.д. Лектор их не показывал и разделы дальше для подробного ознакомления.

Раздел 19(3) 1. Структуры данных: двоичная куча (binary heap)

**1. Структуры данных: двоичная куча (binary heap)**

Двоичная куча –это такая структура данных, которая позволяет быстро добавлять элементы и извлекать элемент с максимальным приоритетом.

**1.1. Введение**

Двоичная куча - собой полное бинарное дерево, для которого выполняется основное свойство кучи: приоритет каждой вершины больше приоритетов её потомков. В простейшем случае приоритет каждой вершины можно считать равным её значению. Двоичную кучу удобно хранить в виде одномерного массива, причем левый потомок вершины с индексом i имеет индекс 2\*i+1, а правый 2\*i+2.

ЗАМЕЧАНИЕ дерево называется полным бинарным, если у каждой вершины есть не более двух потомков, а заполнение уровней вершин идет сверху вниз (в пределах одного уровня – слева направо).