Лекция 14

10 марта 2020 г. 12:59

14. Использование функций.

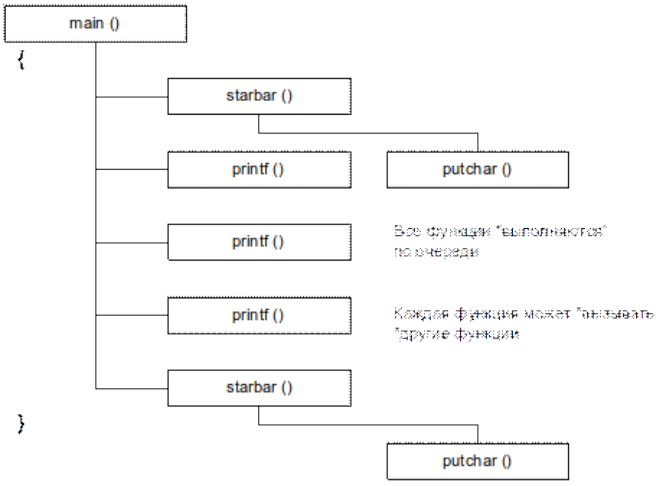
14.1. Создание и использование простой функции.
14.2. Прототипы функций.
14.3. Вызов по значению и вызов по ссылке.
14.4. Использование указателей для связи между функциями
14.5. Рекурсия.
14.6. Параметры и аргументы функций
14.7. Возвращение значения функцией: оператор return.
14.8. Типы функций.
14.9. Важные возможности C++.
14.10. Аргументы функции main().
14.11. Области видимости. Локальные и глобальные переменные
14.12. Сложности в правилах области действия (scope rules).
14.13. Математические функции
14.14. Указатель на функцию
14.15. Массив указателей на функции
14.16. Шаблоны функций в C++. Основные понятия
14.17. Параметры шаблонов функций.
14.18. Шаблоны функций. Аргументы по умолчанию
14.19. Функции округления
14.20. Компиляция программ, состоящих из двух или более функций.

14.1. Создание и использование простой функции.

void starbar(); - прототип функции

starbar(); - вызов функции

Мы вызвали функцию starbar() из функции main(), используя только ее имя.



14.2. Прототипы функций.

Согласно стандарту ANSI C все функции должны иметь прототипы. Прототипы могут располагаться либо в самой программе на C или C++, либо в заголовочном файле. Большинство прототипов функций находятся в самих программах. Объявление функции в C и C++ начинается с ее прототипа. Прототип функции достаточно прост; обычно он включается в начало программы для того, чтобы сообщить компилятору тип и количество аргументов, используемых некоторой функцией. Использование прототипов обеспечивает более строгую проверку типов по сравнению с той, которая была при прежних стандартах C.

Сама функция содержит в себе некий фрагмент программного кода на C или C++ и обычно следует за описанием функции main(). Функция может иметь следующий вид:

возвращаемый_тип имя_функции(типы_аргументов и имена_аргументов)

```
{
.
.
(объявления данных и тело функции)
.
.
return();
}
```

14.3. Вызов по значению и вызов по ссылке.

В предыдущих примерах аргументы передаются функциям по значению. Когда переменные передаются по значению, в функцию передается копия текущего значения этой переменной. Поскольку передается копия переменной, сама эта переменная внутри вызывающей функции не изменяется. Вызов по значению — наиболее распространенный способ передачи информации (параметров) в функцию, и этот метод в C и C++ задан по умолчанию. Главным ограничением вызова по значению является то, что функция обычно возвращает только одно значение.

При вызове по ссылке в функцию передается не текущее значение, а адрес аргумента. Программе требуется меньше памяти, чем при вызове по значению. Кроме того, переменные в вызывающей функции могут быть изменены. Еще одним достоинством этого метода является то, что функция может возвращать более одного значения.

14.4. Использование указателей для связи между функциями

Ниже приводится программа, в которой указатели служат средством, обеспечивающим правильную работу функции, которая осуществляет обмен значениями переменных. #include<stdio.h>

```
void interchange(int * ,int * );

void main()
{
int x = 5, y = 10;
printf(" At the beginning x = %d and y = %d.\n" , x, y);
interchange(&x,&y); /* передача адресов функции */
printf(" And now x = %d and y = %d.\n" , x, y);
}
void interchange(int *u, int *v) /* u и v являются указателями */
{
int temp;
temp = *u; /* temp присваивается значение, на которое указывает u */
*u = *v;
*v = temp;
}
```

14.5. Рекурсия.

Функция может вызывать сама себя. При этом говорят, что возник рекурсивный вызов. Рекурсия бывает:

простой - если функция в теле содержит вызов самой себя;

косвенной – если функция вызывает другую функцию, а та в свою очередь вызывает первую.

При выполнении рекурсии программа сохраняет в стеке значения всех локальных переменных функции и ее аргументов с тем, чтобы в дальнейшем по возвращении из рекурсивного вызова восстановить их сохраненные значения.

Применять рекурсию следует с осторожностью, так как ее использование для функции, содержащих количество переменных или слишком большое вызовов, может вызвать переполнение стека, также помнить, что при использовании

рекурсивного вызова разработчик обязан предусмотреть механизм возврата в вызывающую процедуру, чтобы не произошло образования бесконечного цикла.

Некоторые задачи на практике могут быть проще и нагляднее решены именно с использованием рекурсивных функций.

14.6 Параметры и аргументы функции

Гибкость языка Си во многом объясняется тем, что пользователь практически свободен от применения аргументов функции.

Обязательным для использования функции является описание прототипа и самой функции.

Прототип является необязательным.

Прототип функции расположен прямо перед описанием функции (но это плохой тон). При описании алгоритма могут использоваться формальные и фактические параметры .

```
Int soup(int, int);
Double max(double par1, double par2);
Void func();
```

Хороший стиль программирования предполагает описание прототипов функции в заголовочном файле. В нем описывают константы, внешние и исходные файлы.

В проект достаточно один раз включить заголовочный файл.

Обязательным условием для компилятора является:
1) Совпадение типов
2) Типы аргументов должны совпадать

Язык Си содержит значительно меньшее кол-во функций, чем другие языки. Для подключения большинства математических функций требуется подключить библиотеку <math.h>. Все функции работают с радианами.

Существует множество задач, в которых значения предопределены. Такая возможность реализуется по умолчанию. В этом случае аргумент функции при определении типа указывается значение по умолчанию.

Использование значения по умолчанию делает возможным вызов функции по умолчанию.

При использовании значения по умолчанию необходимо учитывать, что первым записываются те значения, которые мы не задаем по умолчанию.

14.7 Возвращение значений функций: оператор return

Функция по умолчанию может возвращать значении несколькими способами:
1) return
2) применение ссылки или указателей

Применение return использует обязательное требование: возвращаемый тип данных должен соответствовать типу самой функции.

14.8 Типы функций

Типы функций могут любыми, которые используются в С.

14.9 Важные возможности С++

Возможности С++ расширяются за счет
1) Встраивания (inline)
Встраивание во многом напоминает применение макросов. Принципиальное отличие в том, что inline обрабатывается компилятором, а не препроцессором.

Основное правило к inline function - минимальный размер и минимальное количество операторов. Это связано с тем, что куча компилятора ограничена. Для описания функции inline к прототипу добавляются ключевое слово inline.

Inline int sum(int, int);

2) Перегрузка (ovreloading)
Перегрузка является одним из основных атрибутов ООП. Именно эффект перегрузки дает полиморфизм - возможность по разному обрабатывать одни и те же события.

А) имена перегружаемых функций должны быть одинаковы
Б) Количество аргументов должно быть разным или тип хотя-бы одного из аргументов должен отличаться

При формировании объектного модуля компилятор создает функции, перегружаемые с учетом

А) отличие типов функции
Б) обязательное совпадение типа
В) создаются префиксы

Функции с переменным числом параметров

В качестве последнего аргумента ставится многоточие.

В конечном итоге компилятор знает количество параметров. Всегда при использовании многоточия компилятор решает три задачи:
1) как установится на список параметров стеке
2) как перебирать параметры
3) как закончить перебор

Использование специальных макросов для работы с неопределенным количеством параметров.

Макросы описаны в библиотеке stdath.h. Для работы с неопределенным кол-вом параметров используются три макроса:

Void va_start(va_list param, <последний явный параметр>) // связывание переменной param с первым параметром
Type va_arg(va_list param, type);
Void va_end(va_list param);

Наиболее часто используется неопределенное кол-во параметров функции, когда аргументами функции является строка.

Т.к. строка является фундаментальным типов при создании интерфейса, то неопределённое кол-во параметров требует знать:
1) размер строки
2) указатель на строку (или на массив указателей на строки)

В самом простом случае функция, работающая в неопределенным кол-вом строк используется синтаксис:
Char *f(char*s1, …);

14.10 Аргументы функции main()

Все функции языка Си равноправны. Main() имеет свои особенности:
1) определяет стартовый адерс работы программы
2) может принимать произвольное кол-во аргументов
3) аргументы могут быть заданы только из командной строки

В соответствии с правилами записи синтаксических конструкций в квадратных скобка записывается необязательная информация.

Первой в командной строке передается имя исполняемого файла. Все аргументы программы (функции main()) записываются через tab или space. После имени исполняемого файла следующим идет кол-во передаваемых файлов, а далее массив указателей.

Для преобразования строк Си предоставляет множество функций.

Main() имеет два аргумента, но с их помощью можно передать произвольное кол-во данных.

14.11 Области видимости

Классы памяти позволяют манипулировать идентификаторами под одним именинам, но они являются или клонами, или самостоятельными переменными.

Класс памяти может использоваться не только для идентификатора, но и для функции. Класс памяти может быть объявлен двумя способами:
1) ключевым словом
2) местом написания

14.12 Сложности в правилах области действия

Если используются переменные с различной областью действия, то можно столкнуться с совершенно неожиданными результатами программирования, называемыми побочными эффектами. Например, как вы уже знаете, может существовать переменная (вернее две переменные с одинаковым именем) как с файловой, так и с локальной областью действия. Правила области действия констатируют, что переменная с локальной областью действия (называемая локальной переменной) имеет приоритет по сравнению с переменной с файловой областью действия (называемой глобальной переменной).

В следующем примере четыре переменные имеют локальную область действия внутри функции main(). Копии переменных il и im передаются в функцию iproduct(). Это не нарушает правила области действия. Однако, когда функция iproduct() пытается обратиться к переменной in, она ее не находит. Почему? Потому что область действия этой переменной локальна для функции main().

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <stdio.h>
#include <process.h>
#include <ctype.h>
#include <stdlib.h>
#include <math.h>
#include <stdarg.h>
#include <string.h>
using namespace std;

int iproduct(int iw,int ix);
/* int in=10; */

main ()
{
int il=3;
int im=7;
int in=10;
int io;
io=iproduct(il,im);
/* Произведение чисел равно */
printf("The product of the numbers is: %d\n", io);
printf ("\n\nPress any key to finish\n");
_getch();
return(0);
}

int iproduct(int iw,int ix)
{
```

```
int iy;
iy=iw*ix;
/* iy=iw*ix*in; */
return (iy);
}
```

Глобальная видимость переменной in во всем файле позволяет ее использовать как функции main(), так и функции iproduct(). Также надо заметить, что обе функции, main() и iproduct(), могут изменять значение переменной. Существует хорошее программистское правило: если функции должны быть действительно переносимыми, не следует позволять им менять глобальные переменные.

Правила области действия констатируют, что у переменной, имеющей как локальную, так и файловую область действия, используется ее локальное, а не глобальное значение.

Оператор cout правильно распечатывает значения переменных il и im. Но при обращении к переменной in он выбирает глобальную переменную с файловой областью действия. Результат, выдаваемый программой, 3*7*10 = 42, является явной ошибкой. Как вы знаете, в подобной ситуации функция iproduct() использует локальное значение переменной in.

Как было показано выше, объявление локальной переменной скрывает глобальную переменную с таким же именем. Таким образом, все обращения к имени глобальной переменной в пределах области действия локального объявления вызывают обращение к локальной переменной. Однако C++ позволяет обращаться к глобальной переменной из любого места программы с помощью использования операции разрешения области видимости. Для этого перед именем переменной ставится префикс в виде двойного двоеточия (::):

14.13 Математические функции

Прототипы стандартных математических функций определены в заголовочном файле math.h.

```
double log(double); // натуральные
float logf(float);
long double logl(long double);
double log10(double); // десятичные
float log10(float);
long double log10 (long double);
double sqrt(double); // корень числа
float sqrtf(float);
long double sqrtl(long double);
int abs(int); // целые
double f abs (double); // двойной точности
long labs (long); // длинные
float fabsf (float); // с плавающей точкой
long double fabsl (long double);// длинные
// Арккосинус:
double acos(double);
float acosf(float);
// Арксинус:
double asin(double);
float asinf(float);
// Арктангенс:
double atan(double);
```

```
float atanf(float);
// Арктангенс отношения y/x:
double atan2(double x, double y);
float atan2f(float, float);
// Косинус:
double cos(double);
float cosf(float);
// Гиперболический косинус:
double cosh(double);
float coshf(float);
// Синус:
double sin(double);
float sinf(float);
// Гиперболический синус:
double sinh(double);
float sinhf(float);
// Тангенс:
double tan(double);
float tanf(float);
// Гиперболический тангенс:
double tanh(double);
float tanhf(float);
```

14.14 Указатель на функцию

Указатели на функции задаются следующим образом:

```
void f() { }
void (*pf)() = &f;
pf();
```

В этом коде f - некоторая функция.

Во второй строчке определяется переменная pf, которая является *указателем на функцию, которая ничего не возвращает и не принимает ни одного аргумента*. В определении pf ей присваивается адрес функции f.

В третьей строке мы вызываем функцию по указателю pf(в данном случае будет вызвана функция f).

Во время присваивания указателю на функцию адреса функции амперсанд(&) можно опустить. В этом случае будет выполнено автоматическое привидение имени функции к адресу этой функции.

Модификатор const после имени функции означает, что функция не может изменять члены-переменные того объекта, к которому она принадлежит. Более того, из этой функции можно вызывать только такие же константные функции. Зачем все это нужно - используя это, можно защитить самого себя от случайного изменения информации, которая не должна меняться

Указатель на функцию - переменная, которая содержит адрес некоторой функции. Соответственно, косвенное обращение по этому указателю представляет собой вызов функции.

Определение указателя на функцию имеет вид:

```
int (*pf)(); // без контроля параметров вызова
int (*pf)(void); // без параметров, с контролем по прототипу
```

```
int (*pf)(int, char*); // с контролем по прототипу
```

Как и обычный указатель, указатель на функцию имеет размерность адреса, т.е. на уровне архитектуры является машинным словом, содержащим адрес функции в соответствии с системой формирования адресов процессором (системой адресации памяти). Это позволяет, в свою очередь, опускаться до таких машинно-зависимых действий как вызов функции по заданному адресу памяти, преобразуя целую константу к указателю на функцию.

```
void (*pf)(void) = (void(*)(void))0x1000; // Константа 1000 – шестнадцатеричный адрес
void main() { (*pf)(); } // функции, которую вызывает main
// void(*)(void) –абстрактный тип данных –
// указатель на функцию
```

14.15 Массив указателей на функции

Массив указателей на функции можно использовать в разных целях. Один из вариантов такого использования — это если, к примеру, у вас в коде есть одна куча разных функций, но схожих по смыслу и другая куча разных функций, не похожих на первые функции, но похожих друг на друга. Вот можно похожие функции собрать в массив и потом уже вызывать из массива. Не возьмусь утверждать, но скорее всего так будет проще не запутаться.

14.16 Шаблоны функций в C++. Основные понятия

Ключевое слово template обозначает Шаблон.

template <class T> обозначает Шаблон функции с одним параметром T

После написания template и указания в угловых скобках всех параметров (В приведенном примере один параметр T) в примере написана функция MyFunc. Вместо явного указания типа, тип для функции и тип для параметра этой функции был указан как тип по шаблону. Для такого указания типа используются имена параметров, указанных в угловых скобках шаблона.

Проще говоря — этот описанный тип T можно изменить на привычный тип (int, double или другой).

Сейчас тот этап когда нужно все четко расставить на свои места и понять, что вместо указания явного типа для описываемой функции были использованы возможности шаблонов C++. Можно сказать, что шаблон функции вытеснил описание типов и встал на их место.

Дальше проще. Функция MyFunc возвращает некоторое значение. Для того, чтоб было максимально просто понять материал,- что функция принимает, то и возвращает, используется только один параметр. Так как функция возвращает один параметр, то и в шаблоне для этой функции должен быть указан один параметр. Если в шаблоне указан один параметр — то и внутри функции по этому шаблону нужен один, если два — то два и т.д.

Теперь остается функция main. Внутри функции main три раза происходит вызов функции MyFunc и каждый раз в нее передается и в ней принимается один параметр. При этом каждый раз этот параметр воспринимается компилятором как параметр некоторого типа, причем типы различны. (Это указано в комментариях кода). Тут как раз немного отображается смысл использования шаблонов. Без шаблонов нужно писать отдельные функции, которым нужно явно задать тип, который они возвращают и в нашем случае — это три отдельные функции с уникальными именами, выполняющие одно и то же с единственным отличием — разница результирующего типа.

14.17 Параметры шаблонов функций

Проблемным моментом в шаблонной функции предыдущего раздела был тип возвращаемого значения. Это просто логическое продолжение того материала.

В main происходит вызов функции max, в которую передаем 2 значения, причем типы этих значений различны (int и double). Функция принимает эти параметры, как a и b. Принимает она их по ссылке, благодаря чему не создается нутренних локальных копий этих переменных. А указанием const, мы обозначаем, что изменять мы эти значения не намерены. Типы принимаемых параметров, мы оставляем на потом. Т.е. когда нам надо будет вызвать эту функцию, тогда мы и скажем, что подставить в T1 и что подставить в T2.

Этот код имеет 2 недостатка. Первый, наиболее очевидный, в том, что возвращаемое значение из функции max, может принимать оба типа. Какой она тип вернет зависит от порядка передаваемых в нее аргументов.

Другая проблема не столь очевидная в том, что T2 может неявно приводиться к T1. Если немного подумать, можно легко понять, что функция возвращает тип T1. А тип T2 может не совпадать с типом T1. Что если переменная типа T2 больше чем переменная типа T1 ? Возвращать надо T2, но так типы не совпадают, происходит приведение типа из T2 в T1. Для этого приведения внутри функции создается внутренняя локальная переменная. Живет она внутри функции и умирает вместе с завершением работы функции. В свою очередь из-за этого нельзя вернуть результат по ссылке.

14.18 Шаблоны функций. Аргументы по умолчанию

Этот материал показывает маленький фокус с обработкой матриц. До $C++_{11}$ обработка двумерных и более мерных матриц внутри функций требовала того, чтобы программисты C++ явно указывали саму матрицу и ее размеры. Либо использования двумерного массива как одномерного, либо другого извращенного варианта. Язык C++ развивается и дополняется различными возможностями. С одной стороны это плохо, с другой удобства — это удобства. Поэтому простой прием обработки двумерного статически создаваемого массива внутри функции.

Из-за того, что размеры строк и колонок одинаковые, нам не нужно дополнительно узнавать сколько там у массива получилось в пустых скобках. Мы ведь итак знаем, что у него число строк и колонок одинаковое. Удивительно, но этот код прекрасно работает благодаря тому, что мы использовали шаблон, в котором указали аргумент по умолчанию.

Но что делать, если массив не такой "ровный", а прямоугольный, где число строк и колонок разнится. То, что массив способен распознать часть своего размера не обозначает, что это сможем сделать мы, да и этот показанный прием даст сбой.

14.19 Функции округления

Зачастую требуется воспользоваться округленным значением той или иной переменной. C++ предлагает набор функций для решения этой задачи. В зависимости от конкретной ситуации может понадобиться функция, округляющая значение аргумента в большую или меньшую сторону. Рассмотрим наиболее часто используемые варианты вызовов.

Для округления числа в меньшую сторону используется функция floor() и ее разновидности для различных типов аргументов и возвращаемых параметров. Данная функция имеет следующий синтаксис:

```
double floor(double x);
long double floorl(long double x);
```

Округление в большую сторону производится с помощью функции ceil():

```
double ceil(double x);
long double ceil(long double x);
```

Однако в реальности проблема выбора, в какую же сторону производить округление, возлагается на разрабатываемую программу. Ниже предлагается два варианта решения этой задачи.

14.20 Компиляция программ, состоящих из двух или более функций.

Для создания больших программ вы должны использовать функции в качестве «строительных блоков». Каждая функция должна выполнять одну вполне определенную задачу. Используйте аргументы для передачи значений функции и ключевое слово return для передачи результирующего значения в вызывающую программу. Если возвращаемое функцией значение не принадлежит типу int, вы должны указать тип функции в ее определении и в разделе описаний вызывающей программы. Типичное определение функции имеет следующий вид:

имя (список аргументов)

описание аргументов

тело функции

Пример

```
diff(x, y) /* имя функции и список аргументов */
int x, y; /* описание аргументов */
{ /* начало тела функции */
int z; /* описание локальной переменной */
z = x - y;
return(z);
} /* конец тела функции */
```

Лекция 15

17 марта 2020 г. 16:24

15. Классы памяти

15.1 "Зоопарк" классов памяти.
15.2 Объявление переменных на внутреннем уровне
15.3 Объявление переменных на внешнем уровне.
15.4 Переменные класса volatile
15.5 Ключевое слово mutable
15.6 Классы памяти и область действия.
15.7 Пространства имен
16.8 Функции и классы памяти
16.9 Функция получения случайных чисел
15.10 Игра в кости
15.11 Функция получения целых чисел getint()
15.12 Сортировка

Одно из достоинств языка Си состоит в том, что он позволяет управлять ключевыми механизмами программы. Классы памяти языка Си — пример такого управления; они дают возможность определить, с какими функциями связаны какие переменные и как долго переменная сохраняется в программе.

Ранее уже упоминалось о таком важном свойстве переменной, как время жизни. Существует четыре модификатора переменных, определяющих область видимости и время действия переменных.

15.1 "Зоопарк" классов памяти

Модификатор auto используется при описании локальных переменных. Поскольку для локальных переменных данный модификатор используется по умолчанию, на практике его чаще всего опускают.

Модификатор auto применяется только к локальным переменным, которые видны только в блоке, в котором они объявлены. При выходе из блока такие переменные уничтожаются автоматически.

По умолчанию переменные, описанные внутри функции, являются автоматическими.

Модификатор register предписывает компилятору попытаться разместить указанную переменную в регистрах процессора. Если такая попытка оканчивается неудачно, переменная ведет себя как локальная переменная типа auto. Размещение переменных в регистрах, оптимизирует программный код по скорости, так как процессор оперирует с переменными, находящимися в регистрах, гораздо быстрее, чем с памятью. Но в связи с тем, что число регистров процессора ограничено, количество таких переменных может быть очень небольшим.

Статические переменные во многом похожи на глобальные переменные. Для описания статических переменных используется модификатор static. Если такая переменная объявлена глобально, то она инициализируется при запуске программы, а ее область видимости совпадает с областью действия и простирается от точки объявления до конца файла. Если же статическая переменная объявлена внутри функции или блока, то она инициализируется при первом входе в соответствующую функцию или блок. Значение переменной сохраняется от одного вызова функции до другого. Таким образом, статические переменные можно использовать для хранения значений переменных на протяжении времени работы программы.

Программа состоит из нескольких модулей, некоторые переменные могут использоваться для передачи значений из одного файла в другой. При этом некоторая переменная объявляется глобальной в одном модуле, а в других файлах, в которых она должна быть видима, производится ее объявление с использованием модификатора extern. Если объявление внешней переменной производится в блоке, она является локальной.

Вы можете также описать статические переменные вне любой функции. Это создаст «внешнюю статическую» переменную. Разница между внешней переменной и внешней статической переменной заключается в области их действия. Обычная внешняя переменная может использоваться функциями в любом файле, в то время как внешняя статическая переменная может использоваться только функциями того же самого файла, причем после определения переменной.

15.2 Объявление переменных на внутреннем уровне.

Любой из четырех спецификаторов класса памяти можно использовать для объявления переменных на внутреннем уровне. (По умолчанию используется auto.) Спецификатор auto объявляет переменную с локальным временем жизни. Она видна только в том блоке, в котором описана, и может иметь инициирующее значение.

Спецификатор класса памяти register указывает компилятору на необходимость размещения переменной в регистре (если это возможно). При использовании этого спецификатора увеличивается скорость доступа и уменьшается размер

кода программы. Видимость переменной такая же, как при использовании спецификатора auto. Если компилятор встречает объявление register, но отсутствуют доступные регистры, переменной назначается класс памяти auto и она запоминается в оперативной памяти.

В ANSI C нельзя использовать адрес объекта типа register. Это ограничение, однако, не распространяется на C++. Если используется операция определения адреса (&) вместе с регистровой переменной в C++, компилятор помещает ее в оперативной памяти, так как он должен сохранить переменную в таком месте, которое имеет действительный адрес.

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.

15.3 Объявление переменных на внешнем уровне

В объявлении переменных на внешнем уровне можно использовать только класс памяти static или extern (auto или register использовать нельзя). Это могут быть либо описания переменных, либо ссылки на переменные, описанные в другом месте. Внешнее объявление переменной, инициализирующее переменную (неявно или явно), является объявлением-описанием:

```
static int ivalue1; // по умолчанию подразумевается 0
static int ivalue1 = 10 // явная инициализация
int ivalue2 =20; // явная инициализация
```

Если переменная описывается на внешнем уровне, она видима в оставшейся части того исходного файла, в котором она объявлена, и не видима в этом файле до точки ее описания. Кроме того, она не видима в других исходных файлах программы, если ее видимость не обеспечена ссылочным объявлением.

Внутри исходного файла можно только один раз описать переменную на внешнем уровне. Если использовать спецификатор класса памяти static, в другом исходном файле можно описать еще одну переменную с таким же именем и спецификатором static. Поскольку каждое статическое описание видимо только в соответствующем исходном файле, конфликтов не будет. При помощи спецификатора extern объявляется ссылка на переменную, которая описана в другом месте. Объявление external можно использовать для обеспечения видимости описания из другого файла или для того, чтобы расширить видимость переменной в том же самом файле. Переменная видима на протяжении оставшейся части того исходного файла, в котором находится объявленная ссылка.

Для того чтобы внешняя ссылка была действительной, связанная с ней переменная должна быть объявлена только один раз на внешнем уровне. Описание может располагаться в любом исходном файле программы.

15.4 Переменные класса volatile

В тех случаях, когда необходимо предусмотреть возможность модификации переменной периферийным устройством или другой программой, используют модификатор volatile. В связи с этим компилятор не пытается оптимизировать программу путем размещения переменной в регистрах.

Пример объявления таких переменных приведен ниже:

```
volatile short sTest;
volatile const int vciTest;
```

Несмотря ни на что, volatile таки используется для доступа к переменной из разных потоков.

Итак, volatile в языке Си - 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.

Есть три основных типа ошибок, касающихся квалификатора volatile:

- неиспользование volatile там, где нужно

обычно совершается программистами, которые не знают про существование volatile, или видели, но не понимают, что это такое;

- использование volatile там, где нужно, но не так, как нужно

присуща программистам, знающим, насколько важен volatile при программировании параллельных процессов или при доступе к периферийным регистрам, но не учитывающие некоторые его нюансы;

- использование volatile там, где не нужно (бывает и такое)

такое делают те, кто однажды обжегся на первых двух ошибках. Это не ошибка и она не приведет к неправильному поведению программы, но создаст свои неприятности.

15.5 Ключевое слово mutable

Иногда есть необходимость изменить некий объект внутри класса, гарантируя неприкосновенность остальных элементов. Неприкосновенность можно гарантировать при помощи const, однако const запрещает изменение всего.

Помочь в данном случае может определение переменной a с ключевым словом mutable. Внесём исправление в приведённый чуть выше пример:

```
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 a
b = i; // Ошибка! Переменная b по прежнему не доступна для изменения.
}
}
```

15.6 Классы памяти и область действия

Большой заблуждение: использовать внешний класс памяти, который дает возможность использовать. При выборе класса памяти следует руководствоваться золотым правилом программирования: программа должна знать то, что должна знать.

При выборе класса памяти следует пользоваться ответами на два вопроса:

1) Время жизни
2) Видимость переменных

15.7 Пространства имен

В структурном программировании может осуществляться разными способами.

Пространство имен содержит описатели идентификаторов констант, прототипы функций и тд. Пространство имен позволяет применять одни и те же идентификаторы в коде, причем они могут быть глобальными и локальными, что дает возможность использовать одновременно несколько пространств.

Широко используется пространство имен в ООП, тк пространство имен включает классы, структуры и тд.

Для описания пространства имен используется namespace.

Существует набор систем стандартных пространств имен. Пользователь может манипулировать со своим и стандартным пространством имен.

Идентификаторы, объявленные в пространстве имен могут присутствовать и в других пространствах имен. Для доступа к данным используется операция расширенного доступа.

Синтаксис доступа к идентификаторам может быть упрощен. Для этого объявляется пространство имен, после чего имя пространства имен указывать необязательно.

Не путать понятия перегрузки и переопределения функции.

Можно переопределять имя пространства имен.

```
namespace A_Very_Long_Name_Of_NameSpace
    {
    float y;
    }

A_Very_Long_Name_Of_NameSpace:: y = 0.0;
namespace Neo = A Very_Long_Name_Of_NameSpace;
Neo::y = 13.4;
```

Код в виде функции может быть размещен в различных структурных единицах программы (в файле, структуре, классе... и пространстве имен). Хорошим стилем записи функции является:

1) Определение прототипа
2) Сам код функции располагается вне

```
namespace Nspace
{
    char c;
    int i;
    void Fund (char Flag);
}

void Nspace::Fund(char Flag)
{
    // тело функции
    …
}
```

Допускается отсутствие имени пространства имен.

```
#include <iostream>

namespace CL { // пространство
    int var = 0;
}

using namespace CL; // отбрасываем префикс CL

int main() {
    std::cout << "var= " << var << "\n";
    var = 1; // используем переменную var
    std::cout << "var= " << var << "\n";

    system("PAUSE");
    return 0;
}
```

15.8 Функции и классы памяти

Как и к любому идентификатору, к идентификатору функции может быть применен класс памяти. Существует ограничение по объявлению классов функции: функция может использовать класс памяти extern и static.

Невозможно обратиться к функции класса static из другого файла, где она не описана. Для функции по умолчанию используется класс extern.

15.9 Функция получения случайных чисел

Вы не можете обойтись без функции получения случайных чисел. Когда кто-либо требует от вас какое-нибудь число, нужно обратиться к этому полезному средству, вместо того чтобы, заикаясь, каждый раз оправдываться. Вы можете использовать ее во многих машинных играх, что менее практично.

```
rand()
{
static int randx = 1;
randx = (randx * 25173 + 13849) % 65536; /* магическая формула */
return( randx);
}
```

15.10 Игра в кости

```
/* 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");
}
```

15.11 Функция получения целых чисел: getint()

Содержание getint()

В общих чертах на псевдокоде выглядит примерно так:

- читаем на входе информацию в виде символов
- помещаем символы в строку, пока не встретим символ EOF
- если встретился символ EOF, устанавливаем состояние в STOP в противном случае
- проверяем строку, преобразуем символы в целое число, если возможно,
- и выдаем сообщение о состоянии (YESNUM или NONUM)

```
/* getint() */
#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 */
}
```

15.12 Сортировка

```
/* сортировка массива целых чисел в порядке убывания */
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] );
}
```

Лекция 16

31 марта 2020 г. 16:51

16. Дополнительные приемы программирования.

16.1 Совместимость типов
16.2 Идентичный тип
16.3 Макроопределения
16.4 Директивы препроцессор
16.5 Оператор defined
16.6 Условная компиляция
16.7 Дополнительные операции препроцессора
16.8 Обрабока ошибок: prerror()
16.9 Модель памяти
16.10 Модификаторы функций
16.11 Модификаторы cdecl и pascal
16.12 Динамическое распределение памяти. Связанные списки.

16.1 Совместимость типов

С — слабо типизированный язык, а C++ имеет чуть большую типизацию (например, перечисляемые типы). Вы видели, как язык С может выполнять автоматические и явные преобразования типов, используя оператор явного приведения типа.

16.2 Идентичный тип

Составной тип — это общий тип, образованный двумя совместимыми типами. Два любых совпадающих типа являются совместимыми, и их составной тип принадлежит к тому же типу.

Два арифметических типа идентичны, если они — одного типа. Краткие объявления одного и того же типа также идентичны. В следующем примере обе переменные, shivalue1 и shivalue2, имеют идентичный тип:

```
short shivalue 1;
short int shivalue2;
```

В следующем примере тип int совпадает с типом unsigned int:

```
int sivalue1;
unsigned int sivalue2;
```

Однако, типы int, short и unsigned отличаются друг от друга; при работе с символьными данными большое различие существует между типами char, signed char и unsigned char.

Как постановил комитет ANSI С, любой тип, которому предшествует какой-либо модификатор доступа, порождает несовместимые типы. Например, следующие два объявления относятся к несовместимым типам:

```
int ivalue1;
const int ivalue2;
```

Комитет ANSI С установил, что каждый перечисляемый тип совместим с зависящим от реализации интегральным типом; однако, в С++ перечисляемые типы с интегральными несовместимы.

Если два массива содержат элементы совместимых типов, то сами массивы считаются совместимыми. Если размер указан только для одного массива или же не указан совсем, то совместимость типов остается. Однако в случае, когда размеры указаны для обоих массивов, для того, чтобы они были совместимыми, размеры должны совпадать.

Для того чтобы считать совместимыми две функции, имеющие прототипы, должны выполниться три условия. Две функции должны иметь один и тот же возвращаемый тип и одинаковое число параметров; при этом соответствующие параметры должны быть совместимых типов. Однако, имена параметров могут не совпадать.

16.3 Макроопределения

Директиву препроцессора #define для объявления символьных констант. Директиву препроцессора #define Ту же самую директиву

можно использовать для определения макросов. Макрос — это фрагмент кода, который выглядит и работает так же, как функция.

Преимущество правильно написанного макроса заключается в скорости выполнения. Макрос раскрывается (заменяется его определением #define) во время работы препроцессора, при этом создается так называемый встраиваемый код. Именно поэтому макросы не вызывают дополнительных затрат времени, как это обычно случается при вызове функций. Однако, при каждой подстановке макроса увеличивается общий размер исполняемой программы.

Обратная ситуация — определения функций раскрываются только один раз и не имеет значения сколько раз они вызываются. Оптимальный выбор между скоростью выполнения и общим размером программы поможет вам решить, каким образом записывать конкретную процедуру.

Имеются и другие, более тонкие различия между макросами и функциями, касающиеся момента раскрытия кода. Их можно разбить на три категории. В С имя функции имеет значение адреса, по которому эта функция располагается. Поскольку макрос является встраиваемым средством и может раскрываться много раз, не существует единого адреса, связанного с ним. Поэтому макрос нельзя использовать в контексте, где требуется указатель на функцию. Кроме того, можно объявить указатель на функцию, но невозможно объявить указатель на макрос.

Компилятор С по-разному анализирует объявление функции и описание макроса #define, поэтому он не выполняет никаких проверок типов в макросе. В результате этого при передаче макросу неправильного количества аргументов или при ошибке в типе аргумента никаких сообщений от компилятора получено не будет.

Так как макросы раскрываются до начала фактической компиляции программы, некоторые макросы могут неправильно обрабатывать аргументы, если этот аргумент вычисляется в макросе несколько раз.

Макрос это фрагмент кода, который выглядит и работает так же, как функция. Однако функцией он не является. Имеется несколько различий между макросами и функциями.

Макросы определяются аналогично символьным константам. Единственное отличие в том, что строка подстановки substitution_string обычно содержит несколько значений:

```
#define search_string substitution_string
```

Для того чтобы показать сходство описаний, в следующем примере директива препроцессора используется для определения символьной константы и макроса:

```
/* #define — символьная константа */
#define iMAX_ROWS 100
/* #define — макрос */
#define NL putchar('\n')

/* макроопределение с аргументами */
#include<stdio.h>
#define SQUARE(x) x*x
#define PR(x) printf("x = %d.\n" , x)
void main()
{
int x = 4;
int z;
z = SQUARE(x);
PR(z);
z = SQUARE(2);
PR(z);
PR(SQUARE(x));
PR(SQUARE(x + 2));
PR(100/SQUARE(2));
PR(SQUARE(++ x));
}
```

При использовании макросов одни символы или лексемы буквально заменяются на другие. Фактический синтаксический анализ объявления макроса, выражения внутри него и оператора, где он вызывается, выполняется после процесса раскрытия макроса.

Многие задачи можно решать, используя макроопределение с аргументами или функцию. Что из них следует применять нам? На

этот счет нет строгих правил, но есть некоторые соображения.

Макроопределения должны использоваться скорее как хитрости, а не как обычные функции: они могут иметь нежелательные побочные эффекты, если вы будете неосторожны. Некоторые компиляторы ограничивают макроопределения одной строкой, и, по-видимому, лучше соблюдать такое ограничение, даже если ваш компилятор этого не делает.

Выбор макроопределения приводит к увеличению объема памяти, а выбор функции — к увеличению времени работы программы. Так что думайте, что выбрать! Макроопределение создает «строчный» код, т. е. вы получаете оператор в программе. Если макроопределение применить 20 раз, то в вашу программу вставится 20 Строк кода. Если вы используете функцию 20 раз, у вас будет только одна копия операторов функции, поэтому получается меньший объем памяти. Однако управление программой следует передать туда, где находится функция, а затем вернуться в вызывающую Программу, а на это потребуется больше времени, чем при работе со «строчными» кодами.

Преимущество макроопределений заключается в том, что при их использовании вам не нужно беспокоиться о типах переменных (макроопределения имеют дело с символьными строками, а не с фактическими значениями). Так наше макроопределение SQUARE(X) можно использовать одинаково хорошо с переменными типа int или float.

Обычно программисты используют макроопределения для простых действий, подобных следующим:

```
#define MAX(X,Y) ( (X) > (Y) ? (X) : (Y))
#define ABS(X) ( (X) < 0 ? - (X) : (X))
#define ISSIGN(X) ((X) == ' +' || (X) == ' ' ? 1 : 0
```

16.4 Директивы препроцессора

Препроцессор С обрабатывает исходный файл перед тем, как компилятор транслирует программу в объектный код. Выбирая правильные директивы, можно создать более эффективные заголовочные файлы, решить некоторые проблемы программирования и предотвратить наложения объявлений при наличии множества файлов.

Эти директивы обычно используются при программировании больших модулей. Они позволяют приостановить действие более ранних определений и создать файлы, каждый из которых можно компилировать по-разному.

Директивы препроцессора представляют собой команды компилятору, которые позволяют управлять компиляцией программы и сделать ее код более понятным. Все директивы начинаются с символа #. Перед начальным символом # и вслед за ним могут располагаться пробелы. Директивы обрабатываются во время первой фазы компиляции специальной программой - препроцессором.

Директива #define позволяет определить некоторый идентификатор или связать его с некоторой последовательностью лексем. В последнем случае он служит для определения макроса. Макросы мы рассмотрим чуть позже, а здесь приведем пример использования директивы #define для определения некоторого идентификатора или символической константы:

```
#define _cplusplus
```

Когда препроцессор «распознает» директиву #include, он ищет следующее за ней имя файла и включает его в текущий файл. Директива имеет вид:

```
#include <header_name>
#include "header name"
#include macro identifier
#include <stdio.h> имя файла в угловых скобках
#include "mystuff.h" имя файла в двойных кавычках
```

Здесь header_name должно быть именем файла с расширением или именем заголовка в новом стиле. Традиционно используется расширение .h или .hpp. Кроме того, в header name может указываться путь к файлу. Препроцессор удаляет директиву #include из текста файла с исходным кодом и направляет содержимое указанного в ней файла для обработки компилятором. Если указан путь к файлу, компилятор ищет его в указанном каталоге. Первая и вторая форма синтаксиса различаются используемым компилятором алгоритмом поиска файла, если полный путь к нему не задан. Форма <header_name>, в которой имя заголовочного файла задается в угловых скобках, указывает компилятору, что заголовочный файл следует искать в подкаталоге \include компилятора. Форма "header_name", в которой имя заголовочного файла задается в двойных кавычках, указывает, что заголовочный файл следует искать в текущем каталоге, затем - в подкаталоге \include компилятора.

Первая и вторая версии не подразумевают никакого раскрытия макроса. В третьей версии предполагается, что существует макроопределение, которое раскрывается в допустимое имя заголовочного файла.

Многие программисты разрабатывают свои стандартные заголовочные файлы, чтобы использовать их в программах. Некоторые файлы могут создаваться для специальных целей, другие можно использовать почти в каждой программе. Так как включенные

файлы могут содержать директивы #include, можно, если хотите, создать сжатый, хорошо организованный заголовочный файл. Рассмотрим этот пример:

```
/* заголовочный файл*/
#include < stdio.h>
#include "bool.h"
#include "funct.h"
#define YES 1
#define NO 0
```

Во-первых, мы хотим напомнить вам, что препроцессор языка Си распознает комментарии, помеченные /* и */, поэтому мы можем включать комментарии в эти файлы.

Во-вторых, мы включили три файла. По-видимому, третий содержит некоторые макрофункции, которые используются часто.

В-третьих, мы определили YES как 1, тогда как в файле bool.h мы определили TRUE как 1. Но здесь нет конфликта, и мы можем использовать слова YES и TRUE в одной и той же программе. Каждое из них будет заменяться на 1.

Может возникнуть конфликт, если мы добавим в файл строку

```
#define TRUE 2
```

Директива #ifdef позволяет проверить, определена ли символическая константа и, если – да, следующие за ней строки будут направлены для обработки компилятором. Она относится к так называемым условным директивам, поскольку проверяет выполнение некоторого условия и в зависимости от результата проверки изменяет процесс компиляции. Используемая совместно с данной (и другими условными директивами), директива #endif сообщает препроцессору о конце условного блока кода.

Директивы #ifdef и #endif представляют собой две из нескольких условных директив препроцессора. Их можно использовать для выборочного включения в программу некоторых операторов. Директива #endif применяется со всеми условными командами препроцессора и означает конец условного блока. Например, если ранее было определено имя LARGE_CLASSES, то в следующем фрагменте программы описывается новое имя MAX_SEATS:

```
#ifdef LARGE_CLASSES
#define MAX_SEATS 100
#endif
```

Директива #undef сообщает препроцессору об отмене всех предыдущих определений для указанного идентификатора. В следующем примере для изменения значения константы MAX_SEATS используется уже знакомая директива #ifdef и #undef:

```
#ifdef LARGE_CLASSES
#undef MAX_SEATS 30
#define MAX_SEATS 100
#endif
```

Сходной с директивой #ifdef является директива #ifndef. Эта директива также проверяет существование указанного в ней идентификатора, однако следующие за ней строки кода передаются компилятору, если этот идентификатор не определен. Приведем пример ее использования:

```
#ifndef WINVER
#define WINVER 0x0400
#endif
```

В директиве #if также присутствует лексема defined. В следующем фрагменте показано как при помощи директивы #if и конструкции defined выполняются те же действия, которые требовали вложенных директив #ifndef и #ifdef:

```
#if defined(LARGE_CLASSES) && !defined (PRIVATE_LESSONS)
#define MAX_SEATS 30
#endif

#ifdef LARGE_CLASSES
```

```
#ifndef PRIVATE_LESSONS
#define MAX_SEATS 30
#endif
```

Директива #else применяется совместно с условными директивами и позволяет отделить часть кода, которая будет обрабатываться, если предыдущее условие не выполняется. Например:

```
#ifdef __FLAT__
#include <win32\windowsx.h>
#else
#include <win16\windowsx.h>
#endif
```

Применение директивы #else достаточно очевидно. Предположим, что некоторая программа переносится с компьютера VAX на PC. Для целых чисел VAX отводит 4 байта или 32 бита, а в PC выделяется только 2 байта или 16 бит. Следующий фрагмент кода, использующий директиву #else, обеспечивает одинаковый размер целых чисел в обеих системах:

```
#ifdef VAX_SYSTEM
#define INTEGER short int
#else
#define INTEGER int
#endif
```

Две другие условные директивы - #if и #elif - также используются для проверки условия. Имя последней из них является сокращением английских слов “else if”. Она позволяет проверить условие, альтернативное установленному в директиве #if. Приведем пример их использования:

```
#if _MSC_VER < 901
//…
#elif _MSC_VER < 1001
//...
#endif
```

Директива #line изменяет внутренний счетчик строк компилятора. Синтаксис ее использования следующий:

```
#line целая_константга <"имя_файла">
```

Директива #error указывает компилятору, что нужно напечатать сообщение об ошибке и прекратить компиляцию. Обычно ее используют внутри условных директив. Вот типичный пример ее использования:

```
#if!defined(MYNAME)
#error Должна быть определена \
константа MYNAME
#endif
```

```
#include <stdio.h>
#include <iostream>
using namespace std;

//Макроопределение
#define MU 13
int main()
{
```

```
setlocale(LC_ALL, "Rus");
int a = 2, b = 3;
printf("%d", MU);
#undef MU
#if!defined(MU)
#error Должна быть определена константа MU
#endif

getchar(); getchar();
return 0;
}
```

Директива #pragma позволяет управлять возможностями компилятора. Синтаксис ее использования следующий:

```
#pragma имя_директивы
```

16.5 Оператор defined

В директивах #if и #elif может применяться оператор defined. Он позволяет проверить, был ли перед этим определен идентификатор или макрос с указанным в этом операторе именем. Можно также применять операцию логического отрицания (!) для проверки того, что идентификатор или макрос не определен. Следующий пример демонстрирует это:

```
#if defined(_MFC_COMPACT_)
#if !defined(_BORLANDC_)
#undef _ANONYMOUS_STRUCT
#else
#define _ANONYMOUS_STRUCT
#endif
#endif
```

16.6 Условная компиляция

Не всегда команды препроцессора располагаются только в заголовочных файлах; их можно использовать в исходном коде для более эффективной компиляции.

```
/* компилируемый оператор if */
if(DEBUG_ON) {
printf("Entering Example Function");
printf("First argument passed has a value of %d",ifirst_arg);
}
/* оператор сравнения */
#if defined(DEBUG_ON)
printf("Entering Example Function");
printf("First argument passed has a value of %d",ifirst_arg);
#endif
```

Первый оператор if компилируется всегда — это означает, что отладочная информация постоянно присутствует в исполняемом файле программы, отражаясь на его размере.

Во второй части фрагмента показана выборочная компиляция кода с использованием директивы #if defined.

16.7 Дополнительные операции препроцессора

Существуют три операции, которые можно использовать только в директивах препроцессора: подстановка строки (#), конкатенация (##) и подстановка символа (#@).

Если перед параметром макроса используется символ #, то компилятор вместо значения этого параметра подставляет его имя. В результате имя аргумента преобразуется в строку. Эта операция необходима потому, что параметры не заменяются, если они входят в литеральную строку, записанную в макросе в явном виде. Следующий пример иллюстрирует синтаксис операции подстановки строки:

```
#define STRINGIZE(ivalue) printf(#ivalue " is: %d",ivalue)
...
...
...
int ivalue = 2;
STRINGIZE(ivalue);
```

Результат работы макроса:

ivalue is: 2

Одно из применений операции конкатенации — динамическое создание имен переменных и макроопределений. Операция объединяет синтагмы, и, удаляя все пробелы, образует новое имя. Если операция ## используется в макроопределении, то она обрабатывается после подстановки параметров макроса, перед тем, как этот макрос проверяется на наличие каких-либо дополнительных операций. Например, в следующем примере показано, как создавать имена переменных с помощью команд препроцессора:

```
#define IVALUE_NAMES(icurrent_number) ivalue ## icurrent_number;
...
...
...
int IVALUE_NAMES(1);
```

Компилятор получит следующее объявление:

```
int ivalue1;
```

Операция подстановки символа предшествует формальным параметрам в макроопределении. При этом фактический параметр рассматривается как отдельный символ, заключенный в простые кавычки. Например:

```
#define CHARIZEIT(cvalue) #@cvalue
...
...
...
cletter = CHARIZEIT(z);
```

Компилятор получит следующее выражение:

```
cletter = 'z';
```

16.8 Обработка ошибок: perror()

Среди многих интересных функций, имеющих прототипы в файле stdio.h, есть функция с именем perror(). Эта функция выводит в поток stderr системное сообщение о последней ошибке, произошедшей при вызове библиотечной процедуры. Для этого используются переменные errno и _sys_errlist, предопределенные в файле stdlib.h. Массив _sys_erriist представляет собой масси в строк сообщений об ошибках. Переменная errno — это индекс массива строковых сообщений, который автоматически устанавливается равным номеру возникшей ошибки. Число элементов в массиве определяется другой константой, _sys_nerr, также определенной в файле stdlib.h.

Функция perror() имеет единственный параметр — строку символов. Обычно передаваемый аргумент — это строка, идентифицирующая файл или функцию, где произошла ошибка. Использовать эту функцию просто; это показано в следующем примере:

```
/*14PERROR.C
Программа на С, демонстрирующая функцию perror(), имеющую прототип
```

```
в файле STDIO.H*/
#include <stdio.h>
void main(void)
{
FILE *fpinfile;
fpinfile = fopen("input.dat", "r");
if (!fpinfile)
/* Невозможно открыть input.dat в файле main() : */
perror("Could not open input.dat in file main() :");
}
```

16.9 Модель памяти

Компилятор Visual C/C++ поддерживает шесть стандартных моделей памяти — tiny (миниатюрная), small (малая), medium (средняя), compact (компактная), large (большая) и huge (гигантская) — а также пользовательскую модель, которая используется в тех приложениях, где имеются особые требования к хранению данных и кода. Тщательно выбирая подходящую модель памяти для приложения, можно улучшить степень использования системных ресурсов и скорость выполнения программы. Ниже кратко описываются отличительные особенности каждой из шести стандартных моделей памяти.

При использовании модели памяти tiny создаются программные файлы с расширением .COM. Такие программы содержат один 64 Кб сегмент для кода и данных. Доступ ко всем объектам кода и данных осуществляется при помощи near (ближних) адресов. Эти программы не могут использовать библиотеки, содержащие far-функции, например, библиотеки графики. Файлы с расширением .COM можно запускать только под DOS. Кроме этого, компилятор Microsoft Visual C/C++ не создает р-код для модели tiny. Программы с этой моделью памяти используют память аналогично описанным в следующем параграфе программам с моделью памяти small. Однако, при компоновке tiny-приложений объектный файл связывается с библиотекой CRTCOM.LIB; в результате исполняемый файл получается с расширением .COM, а не .EXE.

Когда выбрана опция small, программа может иметь два сегмента: один для данных и один для кода. После компиляции программы с моделью памяти small получают расширение .EXE. Эта опция используется по умолчанию, когда не указана другая модель памяти. Размер обоих сегментов — данных и кода — ограничен 64 Кб. Длина программы с моделью памяти small не может превышать 128 Кб. По умолчанию в таких программах используются near-адреса как для кода, так и для данных.

Программа с моделью памяти medium может иметь один сегмент для данных, но множество сегментов — для кода. Поэтому такие программы могут содержать более 64 Кб кода, но не более 64 Кб данных. По мере необходимости программный код может занимать столько места, сколько необходимо, однако при этом размер всего блока данных не должен превышать 64 Кб. По умолчанию в программах с моделью памяти medium используются far (дальние) адреса для кода и near (ближние) адреса — для данных. Установки по умолчанию можно отменить при помощи ключевого слова _near.

При использовании модели памяти compact программы могут иметь несколько сегментов данных, но только один сегмент кода. Эта модель памяти наиболее подходит для тех приложений С, у которых большой объем данных, но короткий код. Модель памяти compact позволяет данным занимать столько места и сегментов, сколько нужно. По умолчанию в программах с этой моделью памяти используются near-адреса для кода и far-адреса для доступа к данным. В приложении можно переопределить эти установки при помощи ключевого слова _near или _huge для данных и ключевого слова _far для кода.

Как нетрудно догадаться, приложения с моделью памяти large могут иметь по несколько сегментов для кода и данных. Однако, размер отдельного объекта данных не должен превышать 64 Кб. Эта модель памяти подходит для основных программ, работающих с большими объемами данных. По умолчанию в программах с моделью памяти large far-адреса используются как для кода, так и для данных. Эти установки можно переопределить в приложении при помощи ключевого слова _near или _huge для данных и ключевого слова _far для кода.

Модели памяти huge и large схожи. Главное отличие заключается в том, что при использовании модели huge снимаются ограничения на размер отдельных объектов данных. Однако, имеются ограничения на размер элементов массива, если этот массив больше, чем 64 Кб. Элементы массива не могут пересекать границы сегментов — это гарантирует эффективную адресацию для каждого элемента. Поэтому размер отдельного элемента массива не может превышать 64 Кб. Кроме того, если массив больше 128 Кб, то размер каждого элемента в байтах должен быть равным степени числа 2. Однако, если размер массива равен 128 Кб или меньше, то его элементы могут иметь любую длину до 64 Кб максимум. Необходимую модель памяти можно устанавливать непосредственно в среде разработки. Для этого достаточно выбрать пункт главного меню Option Language Options, а затем указать C or C++ Compiler options.... Следующее диалоговое окно будет иметь пункт Memory Model. Для выбора нужно щелкнуть по стрелке вниз.

16.10 Модификаторы функций

В зависимости от того, какая модель памяти применяется, функции и указатели на функцию являются ближними или дальними. Как и в случае с указателями на переменные, размер указателя на функцию можно изменить с помощью модификаторов near и far. Так, например, функции, обращение к которым осуществляется операционной системой или драйвером какого-либо устройства, всегда должны быть объявлены как дальние (far) и, если используется модель памяти с короткими указателями для кода (Tiny, Small или Compact), такие функции должны специфицироваться с помощью модификатора far.

16.11 Модификаторы cdecl и pascal

Как уже доминалось ранее, при вызове некоторой функции ее параметры обычно помещаются в стек, а при возврате из функции параметры из стека удаляются. Помещение и удаление параметров из стека осуществляется в соответствии с определенными правилами. Существует два варианта помещения аргументов в стек; слева направо (соглашение о вызове языка Pascal) и справа налево (соглашение о вызове языка C). По умолчанию компилятор использует соглашение, указанное в параметрах (при вызове из командной строки или в опциях интегрированной среды разработки приложения), однако программист всегда может явно указать ятя каждой функции, какое из соглашений вызова необходимо применять. Это осуществляется с помощью модификаторов cdecl и pascal.

Оба модификатора соответствующим образом влияют на внутреннее имя функции при декорировании имен (будет рассматриваться позже), тем самым сообщая компилятору использовать то или иное соглашение о вызове.

Модификатор cdecl указывает компилятору на то, что параметры вызываемой функции должны помещаться в стек в порядке, обратном следованию при вызове, то есть справа налево (первым передается аргумент, стоящий последним в списке параметров, затем предыдущий и т.д.). При этом не делается никаких предположений об ответственности функции за очистку стека. Внутреннее имя функции эквивалентно объявляемому имени с добавлением символа подчеркивания и использованием соглашения языка C о различении верхнего и нижнего регистров:

MyFunction - имя из прототипа функции

_MyFunction - внутреннее имя функции

Модификатор pascal наоборот, требует прямой передачи параметров в стек, слева направо. Кроме того, данный спецификатор сигнализирует, что именно вызываемая функция ответственна за очистку стека. Данное соглашение используют на практике, если функция вызывается много раз из разных мест. Однако функции с переменным числом параметров не могут использовать данного соглашения. Внутреннее имя при использовании данного вида соглашения совпадает с именем прототипа, но с преобразованием в верхний регистр:

MyFunction - имя из прототипа функции

_MYFUNCTION - внутреннее имя функции

16.12 Динамическое распределение памяти. Связанные списки

Связанный список — это набор структур, каждая из которых имеет некоторый элемент, или указатель, ссылающийся на другую структуру в этом списке. Указатель служит для связи между структурами. Эта концепция напоминает массив, однако она позволяет динамически увеличивать список.

```
struct stboat
{
char sztype[15];
char szmodel[15];
char sztitle[20];
char szcomment[80];
int iyear;
long int lmotor_hours;
float fretail;
float fholesale;
struct stboat *nextboat;
} Nineveh, *firstboat, *currentboat;
```

Лекция 17

14 апреля 2020 г. 18:33

17. Библиотека языка С и ввод-вывод.

17.1 Стандартные библиотеки С и С++
17.2 Доступ в библиотеку языка Си
17.3 Потоковый ввод-вывод
17.4 Связь с файлами
17.5 Понятие файла
17.6 Потоковые функции
17.7 Низкоуровневый ввод и вывод в С
17.8 Ввод и вывод символов
17.9 Определение строк в программе
17.10 Ввод и вывод строки
17.11 Функции, работающие со строками
17.12 Создание собственных функций ввода-вывода
17.13 Проверка и преобразование символов
17.14 Преобразования символьных строк: atoi(), atof()
17.15 Ввод и вывод целых чисел
17.16 Форматированный вывод
17.17 Использование фкнкций fread и fwrite, fseek, ftell и rewind
17.18 Форматированный ввод
17.19 Использование функций scanf(), fscanf() и sscanf()
17.20 Распределение памяти: malloc() и calloc()
17.21 Другие библиотечные функции

17.1 Стандартные библиотеки С и С++

assert.h	Макрос отладчика assert
bios.h	Сервисные функции BIOS
cderr.h	Коды ошибок окон диалога
colordlg.h	Идентификационные (ID) номера цветов управляющих элементов окон диалога
commdlg.h	Функции, типы и описания окон диалога
conio.h	Функции консоли и портов ввода/вывода
cpl.h	Описание панели управления и DLL
ctype.h	Классификация символов
custcntl.h	Библиотека пользовательских элементов управления
dde.h	Динамический обмен данными
ddeml.h	Интерфейс прикладного программирования DDEML
direct.h	Управление каталогами
dlgs.h	Идентификационные номера элементов окон диалога
dos.h	Интерфейс MS-DOS
drivinit.h	Устаревший — используйте print.h
errno.h	Описания переменной errno (код ошибки)
excpt.h	Структурированная обработка исключительных ситуаций
fcntl.h	Флаги, используемые в _open и _sopen
float.h	Константы, используемые автоматическими функциями
fpieee.h	Обработка исключительных ситуаций для чисел с плавающей точкой в стандарте IEEE
fstream.h	Функции для классов filebuf и fstream
graph.h	Низкоуровневая графика и шрифты
io.h	Управление файлами и низкоуровневый ввод/вывод
iomanip.h	Параметрические манипуляторы для iostream
ios.h	Функции для класса ios
iostream.h	Функции для классов iostream
istream.h	Функции для класса istream
limits.h	Диапазоны целых чисел и символов
locale.h	Функции локализации
lzdos.h	Устаревший — заменен при помощи #define lib на #include<lzexpand.h>
lzexpand.h	Интерфейс Public для lzexpand.dll
malloc.h	Функции выделения памяти

math.h	Математические функции для чисел с плавающей точкой
memory.h	Функции работы с буферами
mmsystem.h	Интерфейс прикладного программирования для мультимедиа
new.h	Функции выделения памяти C++
ntimage.h	Структуры изображений
ntsdexts.h	Расширения отладчика NTSD и KD
ole.h	Функции, типы и описания OLE
ostream.h	Функции для класса ostream
penwin.h	Функции, типы и описания для оконных перьев
penwoem.h	Интерфейсы API для распознавания оконных перьев
pgchart.h	Презентационная графика
print.h	Функции печати, типы и описания
process.h	Управление процессами
rpc.h	Приложения RPC (Remote Procedure Call — вызов удаленных процедур)
rpcdce.h	Интерфейсы API для выполнения RPC в среде распределенных вычислений (DCE)
rpcdcep.h	Интерфейсы API для выполнения частных (private) RPC
rpcndr.h	Преобразование чисел с плавающей точкой и чисел двойной длины для RPC
rpcnsi.h	Данные интерфейса API, независимого от службы имен
rpcnsip.h	Описания типов и функций для средств автоматической настройки при выполнении RPC
rpcnterr.h	Коды ошибок компилятора и выполнения RPC
scmsave.h	Директивы define и описания хранителей экрана Windows 3.1
search.h	Функции поиска и сортировки
setjmp.h	Функции setjmp и longjmp
share.h	Флаги, используемые в _sopen
shellapi.h	Функции, типы и описания для shell.dll
signal.h	Константы, используемые функцией signal
stdarg.h	Макросы для функций со списком аргументов переменной длины
stddef.h	Общие типы данных и значения
stdio.h	Стандартный ввод/вывод
stdiostr.h	Функции для классов stdiostream и stdiobuf
stdlib.h	Общие библиотечные функции
streamb.h	Функции для класса streambuf
stress.h	Функции ударения
string.h	Обработка строк
strstrea.h	Функции для классов strstream и strstreambuf
tchar.h	Общие интернациональные функции
time.h	Функции времени
toolhelp.h	Функции, типы и описания для toolhelp.dll
varargs.h	Функции списка аргументов переменной длины
ver.h	Функции, типы и описания для управлениями версиями
vmemory.h	Виртуальная память
wchar.h	Широкие символы
wfext.h	Расширения для Диспетчера Файлов Windows
winbase.h	Базовые 32-разрядные Windows интерфейсы API
wincon.h	Подсистема консоли NT
windef.h	Базовые типы окон
windows.h	Функции, типы и описания Windows
windowsx.h	Макросы интерфейсов API
winerror.h	Коды ошибок для интерфейсов API win32
wingdi.h	Данные компоненты интерфейса графического устройства GDI
winioctl.h	Коды управления вводом/выводом для 32-разрядных устройств Windows
winmem32.h	Прототипы и общие определения для winmem32.dll
winmm.h	Приложения мультимедиа
winnetwk.h	Стандартный заголовочный файл winnet для nt-win32

winnls.h	Процедуры, константы и макросы для компоненты NLS (National Language Support — поддержка национальных языков)
winnt.h	32-разрядные типы и константы Windows
winperf.h	Данные для утилиты Performance Monitor
winreg.h	Данные интерфейса API 32-разрядных регистров Windows
winsock.h	Используется для winsock.dll
winspool.h	API-интерфейсы печати
winsvc.h	Диспетчер служб
winuser.h	Процедуры, константы и макросы для пользовательских компонентов
winver.h	Для использования с ver.dll
sys/locking.h	Установка флагов функций
sys/stat.h	Статус файлов
sys/timeb.h	Функция времени
sys/types.h	Типы статусов файла и времени
sys/utime.h	Функция времени выполнения

17.2 Доступ в библиотеку языка Си

Получение доступа к библиотеке зависит от системы, поэтому вам нужно посмотреть в своей системе, как применять наиболее распространенные операторы.

Во многих больших системах UNIX вы только компилируете программы, а доступ к более общим библиотечным функциям выполняется автоматически.

Если функция задана как макроопределение, то можно директивой #include включить файл, содержащий ее определение. Часто подобные функции могут быть собраны в соответствующим образом названный заголовочный файл.

На некотором этапе компиляции или загрузки программы вы можете выбрать библиотеку. В нашей системе, например, есть файл lc.lib, содержащий скомпилированную версию библиотечных функций, и мы предлагаем редактору связей IBM PC использовать эту библиотеку. Даже система, которая автоматически контролирует свою стандартную библиотеку, может иметь другие библиотеки редко применяемых функций, и эти библиотеки следует запрашивать явно, указывая соответствующий признак во время компиляции.

17.3 Потоковый ввод-вывод

Библиотека стандартных функций ввода/вывода С позволяет считывать данные и записывать их в файлы и устройства. Однако в самом языке С отсутствуют какие-либо предопределенные файловые структуры. В С все данные обрабатываются как последовательность байт. Имеется три основных типа функций ввода/вывода: потоковые, работающие с консолью и портами, низкоуровневые.

В потоковых функциях ввода/вывода файлы или объекты данных рассматриваются как поток отдельных символов. Выбирая соответствующую потоковую функцию, вы можете обрабатывать данные любого необходимого размера или формата, начиная от отдельных символов и заканчивая большими, сложными структурами данных.

На техническом уровне, когда программа открывает файл для ввода/вывода при помощи потоковых функций, открытый файл связывается с некоторой структурой типа FILE (предопределенной в stdio.h), содержащей базовую информацию об этом файле. После открытия потока возвращается указатель на файловую структуру. Указатель файла, иногда называемый указателем потока или потоком, используется для ссылки к файлу при всех последующих операциях ввода/вывода.

Все потоковые функции ввода/вывода обеспечивают буферизированный, форматированный или неформатированный ввод и вывод. Буферизированный поток обеспечивает место для промежуточного хранения всей информации, вводимой из потока или записываемой в поток.

Поскольку дисковый ввод/вывод занимает довольно много времени, буферизация потока разгружает приложение. Вместо того чтобы вводить данные из потока по одному символу или по одному элементу данных, потоковые функции ввода/вывода получают данные поблочно. Когда приложению необходимо обработать введенную информацию, оно просто обращается к буферу, что гораздо быстрее. Когда буфер становится пустым, выполняется считывание с диска другого блока.

Во многих языках высокого уровня существует одна проблема с буферизированным вводом/выводом, которую нужно принимать во внимание. Например: если ваша программа выполнила несколько операторов вывода, которые не заполнили буфер вывода, и запись на диск не произошла, то по завершении программы эта информация будет потеряна. Для решения этой проблемы обычно выполняется вызов соответствующей функции для очистки буфера. В отличие от других языков высокого уровня, в языке С данная проблема с буферизированным вводом/выводом решается путем автоматической очистки содержимого буфера по завершении программы. Конечно, хорошо написанное приложение не должно рассчитывать на эти автоматические действия; все действия программы должны описываться в явном виде. Дополнительное замечание: если вы используете потоковый ввод/вывод и приложение заканчивается с аварийным остановом, то буферы вывода могут оказаться неочищенными, что приведет к потере данных.

Аналогичным образом выглядят процедуры, работающие с консолью и портами; их можно рассматривать как расширенные потоковые функции. Они позволяют читать и писать на терминал (консоль) или в порт ввода/вывода (например, в порт принтера). Функции портов ввода/вывода выполняют простое побайтное считывание и запись. Функции ввода/вывода на консоль обеспечивают несколько дополнительных возможностей. Например, можно определить: введен ли с консоли символ или имеют ли вводимые символы эхо-отображение на экране. Последним типом ввода и вывода является низкоуровневый. Функции низкоуровнего ввода/вывода не

выполняют никакой буферизации и форматирования; они непосредственно обращаются к средствам ввода и вывода операционной системы. Эти функции позволяют обращаться к файлам и периферийным устройствам на более низком уровне, чем это делают потоковые функции. При открытии файла на этом уровне возвращается описатель файла (file handle), представляющий собой целое число, использующееся затем для обращения к этому файлу при последующих операциях. В общем случае не рекомендуется смешивать функции потокового ввода/вывода с низкоуровневыми. Поскольку потоковые функции являются буферизированными, а низкоуровневые — нет, при обращении к файлу или устройству при помощи двух разных способов возможны рассогласование или даже потеря данных в буферах. Поэтому для каждого конкретного файла необходимо использовать либо потоковые, либо низкоуровневые функции.

17.4 Связь с файлами

Часто нам бывает нужна программа получения информации от файла или размещения результатов в файле. Один способ организации связи программы с файлом заключается в использовании операций переключения < и >. Этот метод прост, но ограничен. Например, предположим, вы хотите написать диалоговую программу, которая спрашивает у вас названия книг (звучит фамильярно?), и вы намерены сохранить весь список в файле. Если вы используете переключение как, например, в

books > bklist

то ваши диалоговые приглашения также будут переключены на bklist. И тогда не только нежелательная чепуха запишется в bklist, но и пользователь будет избавлен от вопросов, на которые он, как предполагалось, должен отвечать.

Си предоставляет и более мощные методы связи с файлами. Один подход заключается в использовании функции fopen(), которая открывает файл, затем применяются специальные функции ввода-вывода для чтения файла или записи в этот файл и далее используется функция fclose() для закрытия файла.

17.5 Понятие файла

Файл -- именованная область на жестком диске.

Файл -- это всё, что предназначено для ввода или вывода информации.

С этой точки зрения файлы бывают разными: принтер может только выводить информацию, а клавиатура -- только вводить. У такого рода файлов есть много особенностей. У файла на жестком диске есть понятие конца файла. Мы можем его считывать до тех пор, пока он не кончится. Тогда как у клавиатуры нет конца.

Неправильно думать, что между сущностями "файл" и "название файла" есть взаимно однозначное соответствие.

Можно привести аналогию из жизни: если представить, что файл -- это банка с некоторым содержимым, то название файла -- это этикетка на этой банке. Логично предположить, что у банки может быть несколько этикеток.

Язык C++ унаследовал от языка C библиотеку стандартных функций ввода-вывода. Функции ввода-вывода объявлены в заголовочном файле <stdio.h>. Операции ввода-вывода осуществляются с файлами. Файл может быть текстовым или бинарным (двоичным). Различие между ними заключается в том, что в текстовом файле последовательности символов разбиты на строки. Признаком конца строки является пара символов CR (возврат каретки) и LF (перевод строки) или, что то же самое - '\r' + '\n'. При вводе информации из текстового файла эта пара символов заменяется символом CR, при выводе, наоборот, - символ CR заменяется парой символов CR и LF. Бинарный (или двоичный) файл - это просто последовательность символов. Обычно двоичные файлы используются в том случае, если они являются источником информации, не предполагающей ее непосредственного представления человеку. При вводе и выводе информации в бинарные файлы никакого преобразования символов не производится.

open("file1.txt", "wt") -- откроет файл как текстовый файл;

fopen("file1.txt", "wb") -- откроет файл как бинарный файл.

Различие между "wt" и "wb" объясняется тем, что в разных операционных системах символы перевода строки разные. При чтении файла, т.е. при открытии файла с параметрами "rt" или "rb", проблема следующая. Если мы поставим параметр "rb", то при чтении файла символ \10 будет восприниматься как перевод строки. А если поставим параметр "rt", то при чтении файла пара символов \10\13 будет восприниматься как символ перевода строки.

17.6 Потоковые функции

Для того чтобы использовать потоковые функции, в программу должен быть включен файл stdio.h. В этом файле содержатся описания констант, типов и структур, используемых в потоковых функциях, а также — прототипы и макроопределения этих функций.

Вес потоковые функции ввода-вывода обеспечивают буферизированный, форматированный или не форматированный ввод и вывод.

Когда начинается выполнение программы, автоматически открываются следующие потоки:

- stdin - стандартное устройство ввода;
- stdout - стандартное устройство вывода;
- stderr - стандартное устройство сообщений об ошибках;
- stdprn - стандартное устройство печати;
- stdaux - стандартное вспомогательное устройство.

Все они называются стандартными (или предопределенными) потоками ввода-вывода. По умолчанию стандартным устройством ввода, вывода и сообщений об ошибках является пользовательский терминал. Поток стандартного устройства печати относится к принтеру, а поток стандартного вспомогательного устройства - к вспомогательному порту компьютера. По умолчанию при открытии все стандартные потоки, за исключением потоков stderr и stdaux, буферизуются.

Перед тем, как выполнять операции ввода и вывода в файловый поток, нужно его открыть с помощью функции fopen(). Эта функция имеет следующий прототип:

FILE *fopen(const char *filename, const char *mode);

Она открывает файл с именем filename и связывает с ним поток. Функция fopen() возвращает указатель, используемый для идентификации потока в последующих операциях. Параметр mode является строкой, задающей режим, в котором открывается файл.

По завершении работы с потоком он должен быть закрыт. Это осуществляется с помощью функции fclose(), которая имеет следующий прототип:

int fclose(FILE *stream);

Пример:

```
#include <stdio.h>
using namespace std;


int main()
{
setlocale(LC_ALL, "Rus");
FILE *in, *out;
if ((in = fopen("C:\\AUTOEXEC. BAT", "rt")) == NULL){
fprintf(stderr,
"Cannot open input file.\n");
getchar(); getchar();
return 1;
}
if ((out = fopen("C:\\AUTOEXEC.BAK", "wt")) == NULL){
fprintf(stderr,
"Cannot open output file.\n");
getchar(); getchar();
return 1;
}
while (!feof(in))
fputc(fgetc(in), out);
fclose(in);
fclose(out);
fprintf(stderr,
"The file is copied successfully.\n");


getchar(); getchar();
return 0;
}
```

17.7 Низкоуровневый ввод и вывод в С.

В следующем списке перечислены наиболее часто используемые в программах функции низкоуровневого ввода и вывода:

Функция Описание

close()	Закрывает дисковый файл
lseek()	Поиск указанного байта в файле
open()	Открывает дисковый файл
read()	Читает данные в буфер
unlink()	Удаляет файл из подкаталога
write()	Записывает буфер данных

Функции низкоуровневого ввода и вывода не буферизируют и не форматируют данные. Доступ к файлам, открываемых функциями нижнего уровня, осуществляется при помощи описателя файла (handle — целое число, используемое операционной системой для обращения к файлу). Для открытия файлов используется функция open(). Для открытия файла с атрибутами совместного использования (sharing) можно использовать макрос sopen().

17.8 Ввод и вывод символов

Во всех компиляторах С имеются несколько описанных в стандарте ANSI С функций, предназначенных для ввода и вывода символов. Эти функции обеспечивают стандартный ввод и вывод и рассматриваются как высокоуровневые процедуры.

Функция getc() преобразует целое число в беззнаковый символ. Использование беззнакового символа вместо целого числа гарантирует, что ASCII-символы, имеющие значение > 127, не будут представляться как отрицательные значения. Следовательно, отрицательные значения можно использовать для представления нестандартных ситуаций, например, ошибок и конца входного файла. К примеру: конец файла обычно представляется значением -1, хотя в стандарте ANSI С записано, что только константа EOF может иметь некоторое отрицательное значение.

Функция getc() обычно буферизируется, и когда приложение запрашивает символ, управление не передается программе до тех пор, пока в стандартном входном файловом потоке не будет введен символ возврата каретки. Все символы, введенные до символа возврата каретки, хранятся в буфере и передаются последовательно в программу, которая повторно вызывает функцию getc() до тех пор, пока буфер не опустеет.

17.9 Определение строк в программе

Символьные строки представляют один из наиболее полезных и важных типов данных языка Си. Хотя до сих пор все время применялись символьные строки, мы еще не все знаем о них. Конечно, нам уже известно самое главное: символьная строка является массивом типа char, который заканчивается нуль-символом ('\0'). Здесь мы больше узнаем о структуре строк, о том, как описывать и инициализировать строки, как их вводить или выводить из программы, как работать со строками.

Ниже представлена работающая программа, которая иллюстрирует несколько способов создания строк, их чтения и вывода на печать. Мы .используем две новые функции: gets (), которая получает строку, и puts (), которая выводит строку.

17.10 Ввод и вывод строк

Во многих приложениях более естественно выполнять ввод и вывод информации не посимвольно, а большими последовательностями.

Функция gets() имеет следующий прототип:

```
char *gets(char *s);
```

Она выполняет считывание строки символов из стандартного входного потока и помещает их в переменную s. Символ перехода на новую строку '\n' заменяется символом '\0' при помещении в строку s. При использовании этой функции следует соблюдать осторожность, чтобы число символов, считанных из входного потока, не превысило размер памяти, отведенной для строки s.

Функция puts() имеет следующий прототип:

```
int puts(const char *s);
```

Она выводит строку в стандартный выводной поток и добавляет символ перевода на новую строку '\n'. В случае успеха puts () возвращает неотрицательное значение. В противном случае она возвращает EOF.

Приведем пример использования функции вывода строк:

```
#include <stdio.h>
using namespace std;
int main ()
{
setlocale(LC_ALL, "Rus");
char* string = "Выводимая строка";
puts(string);
```

```
getchar(); getchar();
return 0;
}
```

17.11 Функции, работающие со строками.

Большинство библиотек языка Си снабжено функциями, работающими со строками. Рассмотрим четыре наиболее полезных и распространенных: strlen (), strcat (), strcmp () и strcpy().

17.12 Создание собственных функций ввода/вывода

Не ограничивайте себя при вводе и выводе только этими библиотечными функциями. Если у вас нет нужной функции, или она вам не нравится, можно создавать свои собственные версии, используя для этого getchar() и putchar().

Предположим, у вас нет функции puts(). Вот один из путей ее создания:

```
/* put1---- печатает строку */
put1 (string);
char *string;
{
while(*string != ' \0')
putchar(*string++ );
putchar(' \n');
}

/* put2-----печатает строку и считывает символы */
put2 (string);
char *string;
{
int count = 0;
while (*string != ' \0')
{
putchar(*string ++);
count ++;
}
putchar(' \n');
return(count);
}
```

Вызов

```
put2("пицца");
```

печатает строку пицца, в то время как оператор

```
num = puts("пицца");
```

передаст, кроме того, количество символов в num; в данном случае это число 5. Вот несколько более сложный вариант, показывающий вложенные функции:

```
/* вложенные функции */
#include <stdio.h>
main ()
{
put1 ("Если бы я имел столько денег, сколько могу потратить,");
printf("Я считаю %d символа\n",
put2 ("Я никогда бы не жаловался, что приходится чинить старые стулья."));
}
```

Исходный текст программы будет иметь следующий вид:

```
/* вложенные функции */
#include <stdio.h>
/* put1---- печатает строку */
void put1 (char *string)
{
while(*string != '\0')
putchar(*string++ );
putchar(' \n');
}
/* put2-----печатает строку и считывает символы */
int put2 (char *string)
{
int count = 0;
while (*string != '\0')
{
putchar(*string ++);
count ++;
}
putchar(' \n');
return(count);
}

void main ()
{
put1 ("If I had so much money, how much I could spent,");
printf("I count %d simbols\n", put2 ("I would never complain about needs to repair old chairs."));
}
```

17.13 Проверка и преобразование символов.

Заголовочный файл ctype.h содержит несколько функций макроопределений, которые проверяют, к какому классу принадлежат символы. Функция isalpha(c), например, возвращает ненулевое значение (истина), если с является символом буквы, и нуль (ложь), если символ не является буквой. Таким образом,

isalpha('S') != 0, но isalpha('#') ==0

Ниже перечислены функции, чаще всего находящиеся в этом файле. В каждом случае функция возвращает ненулевое значение, если с принадлежит к опрашиваемому классу, и нуль в противном случае.

ФУНКЦИЯ ПРОВЕРЯЕТ, ЯВЛЯЕТСЯ ЛИ с

isalpha(c) буквой

isdigit(c) цифрой

islower(c) строчной буквой

isspace(c) пустым символом (пробел, табуляция или новая строка)

isupper(c) прописной буквой

Ваша система может иметь дополнительные функции, такие как

ФУНКЦИЯ ПРОВЕРЯЕТ, ЯВЛЯЕТСЯ ЛИ с

isalnum(c) алфавитно-цифровым (буква или цифра)

isascii(c) кодом ASCII (0—127)

iscntrl(c) управляющим символом

ispunct(c) знаком пунктуации

Еще две функции выполняют преобразования:

toupper(c) преобразует с в прописную букву

tolower(c) преобразует с в строчную букву

17.14 Преобразования символьных строк: atoi(), atof()

Использование scanf() для считывания цифровых значений не является самым надежным способом, поскольку scanf() легко ввести в заблуждение ошибками пользователей при вводе чисел с клавиатуры. Некоторые программисты предпочитают считывать даже числовые данные как символьные строки и преобразовывать строку в соответствующее числовое значение. Для этого используются функции atoi() и atof(). Первая преобразует строку в целое, вторая — в число с плавающей точкой.

Функция atoi() игнорирует ведущие пробелы, обрабатывает ведущий алгебраический знак, если он есть, и обрабатывает цифры вплоть до первого символа, не являющегося цифрой. Поэтому наш пример «3 - 4 + 2» был бы превращен в значение 3.

Функция atof() выполняет подобные действия для чисел с плавающей точкой. Она возвращает тип double, поэтому должна быть описана как double в использующей ее программе.

Простые версии atof() будут обрабатывать числа вида 10.2, 46 и —124.26. Более мощные версии преобразуют также экспоненциальную запись, т. е. числа, подобные 1.25E — 13.

Ваша система может также иметь обратные функции, работающие в противоположном направлении. Функция itoa() будет преобразовывать целое в символьную строку, а функция ftoa() преобразовывать число с плавающей точкой в символьную строку.

В то время как фунция fgets читеат обычную строку, функция scanf может читать и различные другие типы (целые, вещественные числа).

В языке С есть семейство функций - atoi (a -- ASCII, i -- integer):

N = atoi(string);

Функция принимает единственный параметр строку и пытается ее привести в типу int. Надо заметить, что функция atoi безопасная, но не очень удобная. Безопасная в том смысле, что не сломается: atoi("25a") == 25. "Неудобства" заключаются в том, то если мы передаем в качестве параметра строку, в которой есть не только числа, нужно быть очень внимательным и знать, как работает эта функция. Функция atoi никак не проинформирует нас, если преобразование прошло неудачно.

Например, atoi("abc") == 0, что на самом деле не совсем соответствует действительности. Использовать функцию atoi нужно лишь в том случае, когда вы уверены, что в строке есть число.

17.15 Ввод и вывод целых чисел

В некоторых программах возникает необходимость ввода и вывода в поток (или в буфер) целочисленной информации. Для этого в С имеется две функции: getw() и putw().

Дополняющие друг друга функции getw() и putw() весьма похожи на функции getc() и putc(); отличие в том, что они работают с целыми числами, а не с символами. Функции getw() и putw() можно использовать только с файлами, открытыми в двоичном режиме.

17.16 Форматированный вывод

Богатый ассортимент средств управления форматом вывода в С позволяет легко создавать печатные графики, отчеты или таблицы. Двумя основными функциями, выполняющими этот форматированный вывод, являются printf() и эквивалентная функция для файлов — fprintf().

Мы уже обсуждали функцию printf() довольно основательно. Подобно puts(), она использует указатель строки в качестве аргумента. Функция printf() менее удобна, чем puts(), но более гибка. Разница заключается в том, что printf() не выводит автоматически каждую строку текста с новой строки. Вы должны указать, что хотите выводить с новых строк. Так,

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

дает то же самое, что и

puts(string);

Вы можете видеть, что первый оператор требует ввода большего числа символов и большего времени при выполнении на компьютере. С другой стороны, printf () позволяет легко объединять строки для печати их в одной строке. Например:

printf(" Хорошо, %s, %s\n", name, MSG);

объединяет " Хорошо" с именем пользователя и с символьной строкой MSG в одну строку.

17.17 Использование функций fread, fwrite, fseek, ftell и rewind

Функция ftell() возвращает текущее значение счетчика связанного с файлом. Она имеет следующий прототип:

long int ftell(FILE *stream);

В случае ошибки она возвращает -1L.

Функция fseek() имеет следующий прототип:

```
int fseek(FILE *stream, long offset,
int from);
```

Эта функция изменяет позиционирование файлового потока stream (изменяя значение указанного счетчика) на offset относительно позиции, определяемой параметром from. Для потоков в текстовом режиме параметр offset должен быть равен 0 или значению, возвращаемому функцией ftell().

Функция rewind() имеет следующий прототип:

```
void rewind(FILE *stream);
```

Она устанавливает файловый указатель позиции в начало потока.

Функция fseek() позволяет нам обрабатывать файл подобно массиву и непосредственно достигать любого определенного байта в файле, открытом функцией fopen().

17.18 Форматированный ввод

В программах на С форматированный ввод обеспечивают весьма гибкие функции scanf() и fscanf(). Главное отличие между ними заключается в том, что для последней функции необходимо явно указывать входной файл, из которого считываются данные. В табл. 11.3 перечислены все возможные управляющие символы, которые можно использовать с функция-ми scanf(), fscanf() и sscanf().

17.19 Использование функций scanf(), fscanf() и sscanf()

Все три функции, scanf(), fscanf() и sscanf(), можно использовать для ввода чрезвычайно сложных данных. Взгляните, например, на следующий оператор:

```
scanf("%2d%5s%4f",&ivalue,psz,&fvalue);
```

Этот оператор вводит целое число из двух цифр, строку из пяти символов и вещественное число, занимающее максимально четыре позиции (2.97, 12.5 и так далее). Сможете ли вы определить, что выполняет следующий оператор:

```
scanf ("%*[ \t\n] \"%[^A-Za-z] %[^\"] \"",ps1,ps2);
```

17.20 Распределение памяти: malloc() и calloc()

Ваша программа должна предоставить достаточный объем памяти для запоминания используемых данных. Некоторые из этих ячеек памяти распределяются автоматически. Например, мы можем объявить

```
char place [] = " Залив Свиной печенки" ;
```

и будет выделена память, достаточная для запоминания этой строки.

Или мы можем быть более конкретны и запросить определенный объем памяти:

```
int plates[100];
```

Это описание выделяет 100 ячеек памяти, каждая из которых предназначена для запоминания целого значения.

Язык Си не останавливается на этом. Он позволяет вам распределять дополнительную память во время работы программы. Предположим, например, вы пишете диалоговую программу и не знаете заранее, сколько данных вам придется вводить. Можно выделить нужный вам (как вы считаете) объем памяти, а затем, если понадобится, потребовать еще. Ниже дан пример, в котором используется функция malloc(), чтобы сделать именно это. Кроме того, обратите внимание на то, как такая программа применяет указатели.

Сначала давайте посмотрим, что делает функция malloc(). Она берет аргумент в виде целого без знака, которое представляет количество требуемых байтов памяти. Так, malloc(BLOCK) требует 100 байт. Функция возвращает указатель на тип char в начало нового блока памяти. Мы использовали описание

```
char *malloc();
```

чтобы предупредить компилятор, что malloc() возвращает указатель на тип char. Поэтому мы присвоили значение этого указателя элементу массива starts [index] при помощи оператора

```
starts[index] = malloc(BLOCK);
```

17.21 Другие библиотечные функции

Большинство библиотек будут выполнять и ряд дополнительных функций в тех случаях, которые мы рассмотрели. Кроме функций, распределяющих память, есть функции, освобождающие память после работы с нею. Могут быть другие функции, работающие со строками, например такие, которые ищут в строке определенный символ или сочетание символов.

Некоторые функции, работающие с файлами, включают open(), close(), create(), fseek(), read() и write(). Они выполняют почти те же самые задачи, что и функции, которые мы обсудили, но на более фундаментальном уровне. Действительно, функции, подобные fopen(), обычно пишутся с применением этих более общих функций. Они немного более трудны в использовании, но могут работать с двоичными файлами так же, как и с текстовыми.

Лекция 18

21 апреля 2020 г. 12:30

18. Структуры и другие типы данных

18.1. Структуры С и С++.
18.2. Массив структур.
18.3. Вложенные структуры.
18.4. Использование указателей на структуры.
18.5. Структуры и функции.
18.6. Структуры и битовые поля.
18.7. Структуры. Их дальнейшее использование.
18.8. Объединения.
18.9. Вспомогательные средства.
18.10. Сложные формы данных.
18.11. Отличие struct от class.
18.12 Функции работы с датой и временем.

Кроме простых типов данных Си позволяет манипулировать со сложными (самый простой сложный - массив): ассоциативные массивы, запись, структуры, объединения, класс... Более того, пользователь может создавать собственные типы данных.

Чрезмерное использование собственного типа данных затрудняет чтения кода.

18.1 Структуры в С и С++

В отличие от массива, структура позволяет хранить множество значений разного типа. Более того, сама структура может быть объявлением структуры. Структура - тип данных. Тип этой структуры определяется с помощью шаблона, которой соответствует переменная. Сам шаблон при его описании место в оперативной памяти не занимает. Память выделяется только для структурных переменных. Для объявления нового типа данных struct используется синтаксис:

```
struct поле_тега {
    тип_элемента элемент 1;
    тип_элемента элемент 2;
    тип_элемента элемент 3;
    ...
    ...
    ...
    тип_элемента_элемент N;
};
```

В качестве элемента может выступать любой тип данных, включая структуру:

```
struct stuff {
    int numb;
    char code [4];
    float cost;
};
```

Для работы со структурными переменными активно используется указатель. Указатель манипулируя со структурными данными изменяется на значение равное самому шаблону.

Структуры являются оптимальным типом данных для манипулирования с базами данных. Необязательным при описании шаблона является его имя (имя тега). (Нецелесообразно)

Для описания структурных переменных:
1) Описать шаблон (даже без имени) и сразу за ним указать имена переменных
2) Описать шаблон (с указанием имени). Используя короткий синтаксис описать переменные.

Язык С++ является более типизированным и позволяет применять самое короткое описание структуры.

```
struct book {
            char title [MAXTIT];
            char author [MAXAUT];
            float value;
    } libry; /* присоединяет имя переменной к шаблону */
```

★ Обычно шаблоны выносятся в заголовочные файлы.

Инициализация:

```
struct MyStruct
{
	int iVariable;
	long iValue;
	char Str[10];
} mystruct = {10, 300L, "Hello"};

struct HOUSE
{
	unsigned short RegNum;
	char Street[51]; // с учетом '\0'
	char HouseNum[6];
	unsigned short MaxFloorNum;
	unsigned short MaxFlatNum;
	bool Parking;
};

HOUSE House;
```

Определить значение элементов структуры можно несколькими способами. Одной из наиболее распространенных является ".":

```
House.RegNum = 524;
strcpy(House.Street, "ул. Гоголя");
strcpy(House.HouseNum, "2-а");
House.MaxFloorNum = 7;
House.MaxFlatNum = 84;
House.Parking = true;
```

Для использования указателя, чтобы вычислить размер шаблона используется функция sizeof():

```
int i = sizeof(HOUSE);
```

Для работы с системными данными существует широкий спектр стандартных структур, описанных в заголовочном файле:

```
struct DATETIME
{
	unsigned short Year; // год
	unsigned short Month; // месяц
	unsigned short Date; // дата
	unsigned short Hour; // часы
	unsigned short Minute; // минуты
	unsigned short Second; // секунды
}
```

Для экономии памяти минимальным размером памяти может быть не байт, а произвольное количество бит:

```
struct DATETIME2
{
	unsigned Year: 7 // год
	unsigned Month: 4 // месяц
	unsigned Date:5 // дата
	unsigned Hour: 6 // часы
	unsigned Minute: 6 // минуты
	unsigned Second: 6 // секунды
}
```

Пример:

```
/* 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);
}
```

Во многих случаях С можно считать подклассом языка С++. В общем, это означает, что все программы на С будут работать в среде С++.

Если использовать методы проектирования С в программах на С++, то зачастую можно упустить более развитые и простые возможности С++.

Описанные выше примеры объявлений структуры работают с компиляторами С и С++. В С++, однако, имеется еще один способ объявления переменной некоторого структурного типа. В этой краткой записи, имеющейся только в С++, отсутствует необходимость повторять ключевое слово struct. Это тонкое различие отражено в следующем примере:

Для обращения к отдельным элементам структуры можно использовать "точку" — операцию обращения к члену структуры (.).

- Элементы структуры обрабатываются так же, как и любые другие переменные С или С++; необходимо только всегда использовать операцию "точка".

Если структурная переменная будет внешней или статической. Здесь следует иметь в виду, что принадлежность структурной переменной к внешнему типу зависит от того, где определена переменная, а не где определен шаблон

Структуру можно передать в функцию при помощи, следующей синтаксической конструкции:

fname(stvariable);

Если передавать полные копии структур в функции, то нередко можно снизить эффективность программы. В приложениях, критичных ко времени, лучше использовать указатели. Если стоит вопрос об экономии памяти при обработке связанных списков, то вместо статически выделяемой памяти зачастую используется функция malloc(), динамически выделяющая память для структур.

18.2 Массив структур

Структуру можно рассматривать как аналогию отдельной карточки в картотеке. Реальные достоинства структур проявляются тогда, когда используются набор структур, называемый массивом структур. На основе такого массива можно разработать информационную базу данных, содержащую самые разные объекты.

Сама по себе одна единственная запись типа структуры в большинстве случаев вряд ли может вызывать повышенный интерес. Однако в случае, когда структурированные данные объединяются в массивы, речь идет уже не о единичном объекте, а о целой базе данных.

Объявление массива структур отличается от объявления обычных массивов лишь тем, что в качестве типа создаваемого массива указывается вновь образованный тип структуры.

Доступ к элементам такого массива осуществляется обычным способом, например, по индексу (индексация ведется, начиная с нуля). Следующий фрагмент кода осуществляет в цикле упорядоченное заполнение номеров домов структуры HOUSE, а затем выводит записанные данные в столбик на экран:

```
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 ';
```

18.3 Вложенные структуры

Иногда бывает удобно, чтобы одна структура содержалась или была «вложена» в другую.

```
/* 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");
}
```

Во-первых, следует рассказать о том, как поместить вложенную структуру в шаблон. Она просто описывается точно так же, как это делалось бы для переменной типа int:

```
struct names handle;
```

Во-вторых, следует рассказать, как мы получаем доступ к элементу вложенной структуры. Нужно дважды использовать операцию ".":

fellow.handle.first = = "Франко" ;

18.4 Использование указателей на структуры

Указатели можно использовать и для структур. Это хорошо по крайней мере по трем причинам. Во-первых, 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. Во-вторых, структура не может использоваться в качестве аргумента функции, а указатель на структуру может. В-третьих, многие удобные представления данных являются структурами, содержащими указатели к другим структурам.

Вот самое простое описание, какое только может быть:

```
struct guy *him;
```

Первым стоит ключевое слово struct, затем слово guy, являющееся именем структурного шаблона, далее * и за нею имя указателя. Синтаксис тот же, как для описаний других указателей, которые мы видели.

Теперь можно создать указатель him для ссылок на любые структуры типа guy. Мы инициализируем him, заставляя его ссылаться на fellow[0]; заметим, что мы используем операцию получения адреса:

```
him = &fellow[0];
```

Первый способ, наиболее общий, использует новую операцию - >. Она заключается в вводе дефиса (-) и следующего за ним символа «больше чем» (>). Пример помогает понять смысл сказанного:

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

Второй способ определения значения элемента структуры вытекает из последовательности:

если him == &fellow[0], то *him == fellow[0].

Это так, поскольку & и * — взаимообратные операции. Следовательно, после подстановки

fellow[0]. income = = (*him). income

Круглые скобки необходимы, поскольку операция "." имеет приоритет выше, чем *.

Отсюда можно сделать вывод, что если him является указателем на структуру fellow [0], то следующие обозначения эквивалентны:

fellow[0]. income == (*him).income == him-> income

18.5 Структуры и функции

Зачастую в тело функции необходимо передать информацию, структурированную по определенному принципу. При этом в качестве параметра в прототипе функции указывается пользовательский тип данных, сформированный при объявлении структуры. Так, если была объявлена структура

```
struct ALLNUMB
{
int nVar;
long lVar;
short shVar;
unsigned int uiVar;
}
```

Разработчик ПО на C++ имеет возможность обращаться к элементам структуры через указатели. Для этого должна быть объявлена соответствующая переменная типа указателя на структуру, синтаксис которой может быть представлен в виде:

тип структуры* идентификатор_указателя;

Помимо использования указателей возможно применение ссылок на структуры. Объявление такой ссылки имеет следующий синтаксис:

тип структуры &имя_ссылки = имя_переменной;

Как и ссылка на обычную переменную, ссылка на структуру должна быть инициализирована именем объекта, на который она указывает (в данном случае это имя_переменной).

Ссылки и указатели на структуры данных могут быть переданы в качестве аргументов в тело функции. При этом значительно снижается время (в сравнении с передачей по значению), за которое данный параметр передается в функцию, а также экономится стековая память.

Синтаксис прототипа функции при передаче структур посредством указателей и ссылок идентичен синтаксису обычной передачи параметров через указатели и ссылки:

```
// 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);
```

Таким образом, в функцию Func2 будут переданы не сами значения структур, а соответствующие адреса, что в значительной степени экономит стековую память.

Поскольку элемент структуры является переменной с единственным значением (т. е. типа int или одного из его «родственников» — char, float, double или указатель), он может быть передан как аргумент функции.

18.6 Структуры и битовые поля

При описании полей структуры в явном виде можно указывать размер полей. Минимальный размер поля структуры – один бит. Общий синтаксис объявления структуры с явным указанием размеров полей имеет вид:

```
struct имя{
тип_поля1 имя_поля1: размер1;
тип_поля2 имя_поля2: размер2;
...
тип_поляN имя_поляN: размерN;
};
```

При этом для части полей можно указывать размер, а для других – нет. Если поле имеет размер в один бит, в качестве его типа указывается unsigned (у числа, реализованного с помощью одного бита, не может быть знака).

Путем явного указания размеров полей структуры удалось добиться экономии системных ресурсов: для записи значений используется минимально необходимое количество бит.

18.7 Структуры. Их дальнейшее использование.

Использование структур: создание новых типов данных. Пользователи компьютеров разработали новые типы данных, гораздо более эффективные для определенных задач, чем массивы и простые структуры, которые мы описали.

Эти типы имеют такие названия, как очереди, двоичные деревья, неупорядоченные массивы, рандомизированные таблицы и графы. Многие из этих типов создаются из «связанных» структур. Обычно каждая структура будет содержать один или два типа данных плюс один или два указателя на другие структуры такого же типа. Указатели служат для связи одной структуры с другой и для обеспечения пути, позволяющего вам вести поиск по всей структуре. Например, на рисунке показано двоичное дерево, в котором каждая отдельная структура (или «узел») связана с двумя, расположенными ниже.

18.8 Объединения.

Объединение — еще один тип данных, который можно использовать различными способами. К примеру, некоторое объединение может рассматриваться как целое значение при выполнении одной операции и как число с плавающей точкой или двойной точности — при выполнении другой. По виду объединения напоминают структуры; однако, они сильно отличаются. Объединение, так же как и структура, может содержать несколько различных типов данных. Однако в объединении эти данные занимают одну и ту же область памяти! Таким образом, в отдельный момент времени объединение может хранить информацию только об одном типе данных.

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

```
union поле_тега {
тип поле1;
тип поле2;
тип поле3;
...
…
…
тип полеN;
};
```

Объявление объединения начинается с ключевого слова union, за которым следует идентификатор и блок описания элементов различного типа, например:

```
union MyData
{
int iVar1;
unsigned long ulFreq;
char Symb[10];
}
```

При этом в памяти компилятором резервируется место для размещения самого большого элемента объединения, в данном случае - 10 байт под символьный массив. Чаше всего объединения включают в состав структур, а также служат для преобразования данных одного типа в другой. Доступ к элементам объединения производится с помощью операции 'точка' (.). Ниже приводится пример такого объединения.

В отличие от структур, переменная типа объединения может быть проинициализирована только значением первого объявленного члена (в данном случае - целочисленной переменной iVar1).

В каждый момент времени запоминается только одно значение; нельзя записать char и int одновременно, даже если для этого достаточно памяти.

18.9 Вспомогательные средства.

Помимо структур и объединений разработчик программного обеспечения на C++ имеет возможность моделирования новых типов данных на базе уже имеющихся в его распоряжении типов. Формирование пользовательских типов осуществляется с использованием ключевого слова typedef, за которым указывается какой-либо из имеющихся типов данных и следующий за ним идентификатор, назначающий новое имя для выбранного типа. Фактически, таким образом, определяется синоним типа данных. Например, выражение

```
typedef unsigned char byte;
```

Определяет новый тип данных byte, суть которого такая же, как и типа unsigned char. Следовательно, в программе возможно определение переменных типа byte, которые в памяти будут занимать один байт и смогут принимать значения от 0 до 255.

Будьте внимательны и не используйте слишком много новых типов; это может вызвать обратный эффект и ухудшить читаемость программы и усложнить ее. Используйте typedef с осторожностью.

Кроме того, можно использовать тип COMPLEX для представления комплексных чисел.

Одна из причин использования typedef заключается в создании удобных, распознаваемых имен для часто встречающихся типов. Например, многие пользователи предпочитают применять STRING или его эквивалент, как это мы делали выше.

Вторая причина: имена typedef часто используются для сложных типов. Например, описание

```
typedef char *FRPTC () [5];
```

приводит к тому, что FRPTC объявляет тип, являющийся функцией, которая возвращает указатель на пятиэлементный массив типа Char.

Третья причина использования typedef заключается в том, чтобы сделать программы более мобильными.

Перечисляемый тип данных enum имеет единственное назначение — сделать текст программы более читаемым.

```
enum поле_тега { значение1,. . .значениеN } переменная;
```

```
enum eweekdays { Monday, Tuesday, Wednesday, Thursday, Friday };
/* Объявление переменной типа enum в C */
enum eweekdays ewToday;
/* То же объявление в C++ */
eweekdays ewToday;
/* Без перечисляемого типа */
for(i = 0; i <= 4; i++)
…
…
…
/* С использованием перечисляемого типа */
for(ewToday = Monday; ewToday <= Friday; ewToday++)
```

18.10 Сложные типы данных.

Язык Си позволяет вам создавать сложные формы данных. Обычно мы придерживаемся более простых форм, но считаем своим долгом указать на потенциальные возможности языка. При создании описания мы используем имя (или «идентификатор»), которое можно изменять при помощи модификатора:

модификатор значение

* указатель

() функция

[] массив

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 */
```

Для распутывания этих описаний нужно понять, в каком порядке следует применять модификаторы. Три правила помогут вам справиться с этим.

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 struct от class.

Отличий class от struct несколько (основных всего два):

- *Member access control [class.access]*

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.

- *Accessibility of base classes and base class members [class.access.base]*

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.

- *отличие между структурами и классами, связанное с инициализацией по умолчанию.*

```
struct Foo {
int a;
};

class Bar {
int a;
};

class Tester {
Foo m_Foo = Foo();
Bar m_Bar = Bar();

public:
Tester() {}
};

int main() {
auto myTester = Tester();
}
```

18.12 Функция работы с датой и временем.

Функции и типы данных, необходимые для работы с датой и временем объявлены в заголовочном файле time.h. В частности, этот файл содержит определения типа данных time_t:

```
typedef long time_t;
```

и структуры tm, которая объявлена следующим образом:

```
struct tm
{
```

```
int tm_sec; // секунды
int tm_min; // минуты
int tm_hour; // часы (0-23)
int tm_mday; // дата (1-31)
int tm mon; // месяц (0-11)
// год (текущий год минус 1900)
int tm_year;
// день недели (0-6; Воскр = 0)
int tm wday;
int tm_yday; // день в году (0-365)
int tm_isdst; // 0 если зимнее время
}
```

В качестве обобщающего примера рассмотрим программу работы с датой и временем:

```
#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);
```

Лекция 19.1

28 апреля 2020 г. 21:27

19.1 Динамические структуры данных

19.1.1 Динамические структуры данных
19.1.2 Линейный список
19.1.3 Барьеры
19.1.4 Двусвязные списки
19.1.5 Циклические списки
19.1.6 Стек
19.1.7 Очередь
19.1.8 Дек
19.1.9 Деревья
19.1.10 Графы
19.1.11 Примеры динамических структур

19.1.1 Динамические структуры данных

Часто в серьезных программах надо использовать данные, размер и структура которых должны меняться в процессе работы. Динамические массивы здесь не выручают, поскольку заранее нельзя сказать, сколько памяти надо выделить – это выясняется только в процессе работы.

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

19.1.2 Линейный список

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

```
struct Node{
char word[40]; //область данных
int count;
Node *next; //ссылка на следующий узел
};
typedef Node *PNode; //тип данных: указатель на узел
```

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

```
PNode Head = NULL;
```

Для того, чтобы добавить узел к списку, необходимо создать его, то есть выделить память под узел и запомнить адрес выделенного блока. Будем считать, что надо добавить к списку узел, соответствующий новому слову, которое записано в переменной NewWord. Составим функцию, которая создает новый узел в памяти и возвращает его адрес. Обратите внимание, что при записи данных в узел используется обращение к полям структуры через указатель.

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.

Добавление узла перед заданным. Эта схема добавления самая сложная. Проблема заключается в том, что в простейшем линейном списке (он называется односвязным, потому что связи направлены только в одну сторону) для того, чтобы получить

адрес предыдущего узла, нужно пройти весь список сначала. Задача сведется либо к вставке узла в начало списка (если заданный узел – первый), либо к вставке после заданного узла.

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

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

```
PNode p = Head; //начали с головы списка
while( p != NULL ){ //пока не дошли до конца
//делаем что-нибудь с узлом p
p = p->next; //переходим к следующему узлу
}
```

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

19.1.3 Барьеры

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

19.1.4 Двусвязный список

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

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

```
struct Node
{
char word[40]; //область данных
int count;
Node *next, *prev; //ссылки на соседние узлы
};
typedef Node *PNode; //тип данных «указатель на узел»
```

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

```
PNode Head = NULL, Tail = NULL;
```

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

Дан адрес NewNode нового узла и адрес p одного из существующих узлов в списке. Требуется вставить в список новый узел после p. Если узел p является последним, то операция сводится к добавлению в конец списка (см. выше). Если узел p – не последний, то операция вставки выполняется в два этапа:

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.

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

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

19.1.5 Циклические списки

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

19.1.6 Стек

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

Стек называют структурой типа LIFO (Last In – First Out) – последним пришел, первым ушел. Стек похож на стопку с подносами, уложенными один на другой – чтобы достать какой-то поднос надо снять все подносы, которые лежат на нем, а положить новый поднос можно только сверху всей стопки.

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.

На стек выделяется ограниченная область памяти. При каждом вызове процедуры в стек добавляются новые элементы (параметры, локальные переменные, адрес возврата). Поэтому если вложенных вызовов будет много, стек переполнится. Очень опасной в отношении переполнения стека является рекурсия, поскольку она как раз и предполагает вложенные вызовы одной и той же процедуры или функции. При ошибке в программе рекурсия может стать бесконечной, кроме того, стек может переполниться, если вложенных вызовов будет слишком много.

Если максимальный размер стека заранее известен, его можно реализовать в программе в виде массива. Удобно объединить в одной структуре сам массив и его размер. Объявим новый тип данных – стек на 100 элементов-символов.

```
const int MAXSIZE = 100;
struct Stack
{
char data[MAXSIZE];
int size;
};
```

Для работы со стеком надо определить, как выполняются две операции – добавление элемента на вершину стека (Push) и снятие элемента с вершины стека (Pop).

```
void Push(Stack &S, char x)
{
if(S.size == MAXSIZE){
printf("Стек переполнен");
return;
}
S.data[S.size] = x;
S.size ++;
}
```

19.1.7 Очередь

Очередь – это упорядоченный набор элементов, 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.

Хорошо знакомой моделью является очередь в магазине. Очередь называют структурой типа FIFO (First In – First Out) – первым пришел, первым ушел. На рисунке изображена очередь из 3-х элементов.

Если максимальный размер очереди заранее известен, его можно реализовать в программе в виде массива. Удобно объединить в одной структуре сам массив и его размер. Объявим новый тип данных – очередь на 100 элементов (целых чисел).

```
const int MAXSIZE = 100;
struct Queue
{
int data[MAXSIZE];
int size, head, tail;
};
```

Добавление элемента в конец очереди. Фактически это добавление нового элемента в конец двусвязного списка. 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.

19.1.8 Дек

Дек (deque) - это упорядоченный набор элементов, в котором добавление новых и удаление существующих элементов допустимо с любого конца.

Дек может быть реализован на основе массива или двусвязного списка. Для дека разрешены четыре операции:

- добавление элемента в начало;
- добавление элемента в конец;
- удаление элемента с начала;
- удаление элемента с конца.

19.1.9 Деревья

Дерево – это совокупность узлов (вершин) и соединяющих их направленных ребер (дуг), причем в каждый узел (за исключением одного - корня) ведет ровно одна дуга.

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

Предком для узла x называется узел дерева, из которого существует путь в узел x.

Потомком узла x называется узел дерева, в который существует путь (по стрелкам) из узла x.

Родителем для узла x называется узел дерева, из которого существует непосредственная дуга в узел x.

Сыном узла x называется узел дерева, в который существует непосредственная дуга из узла x.

Уровнем узла x называется длина пути (количество дуг) от корня к данному узлу. Считается, что корень находится на уровне 0.

Листом дерева называется узел, не имеющий потомков.

Внутренней вершиной называется узел, имеющий потомков.

Высотой дерева называется максимальный уровень листа дерева.

Упорядоченным деревом называется дерево, все вершины которого упорядочены (то есть имеет значение последовательность перечисления потомков каждого узла).

Дерево представляет собой типичную рекурсивную структуру (определяемую через саму себя). Как и любое рекурсивное определение, определение дерева состоит из двух частей – первая определяет условие окончания рекурсии, а второе – механизм ее использования.

Вершина дерева, как и узел любой динамической структуры, имеет две группы данных: полезную информацию и ссылки на узлы, связанные с ним. Для двоичного дерева таких ссылок будет две – ссылка на левого сына и ссылка на правого сына. В результате получаем структуру, описывающую вершину (предполагая, что полезными данными для каждой вершины является одно целое число):

```
struct Node
{
```

```
int key; //полезные данные (ключ)
Node *left, *right; //указатели на сыновей
};
typedef Node *PNode; // указатель на вершину
```

Одной из необходимых операций при работе с деревьями является обход дерева, во время которого надо посетить каждый узел по одному разу и (возможно) вывести информацию, содержащуюся в вершинах.

Пусть в результате обхода надо напечатать значения поля данных всех вершин в определенном порядке. Существуют три варианта обхода:

- КЛП (корень – левое – правое): сначала посещается корень (выводится информация о нем), затем левое поддерево, а затем – правое;
- ЛКП (левое – корень – правое): сначала посещается левое поддерево, затем корень, а затем – правое;
- ЛПК (левое – правое – корень): сначала посещается левое поддерево, затем правое, а затем – корень.

19.1.10 Графы

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

Если дуги имеют направление (вспомните улицы с односторонним движением), то такой граф называется направленным или ориентированным графом (орграфом).

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.

19.1.11 Примеры динамических структур данных

Односвязный список:

```
struct list * init(int a) // a- значение первого узла
{
 struct list *lst;
 // выделение памяти под корень списка
 lst = (struct list*)malloc(sizeof(struct list));
 lst->field = a;
 lst->ptr = NULL; // это последний узел списка
 return(lst);
}
```

Двусвязный список:

```
struct list * init(int a)  // a- значение первого узла
{
 struct list *lst;
 // выделение памяти под корень списка
 lst = (struct list*)malloc(sizeof(struct list));
 lst->field = a;
 lst->next = NULL; // указатель на следующий узел
 lst->prev = NULL; // указатель на предыдущий узел
 return(lst);
}
```

Стек:

```
struct comp { //Структура с именем comp int Data; //Какие то данные(могут быть любими, к примеру можно написать int key; char Data; или добавить еще какие то данные) comp *next;//Указатель типа comp на следующий элемент };
```

Очередь:

```
typedef struct QUEUE {
QUEUE * next;
int info;
};
```

Дерево:

```
struct node
{
int info; //Информационное поле
node *l, *r;//Левая и Правая часть дерева
};
```

Граф:

```
#include <iostream>
#include <queue> // очередь
using namespace std;
int main()
{
  queue<int> Queue;
  int mas[7][7] = { { 0, 1, 1, 0, 0, 0, 1 }, // матрица смежности
  { 1, 0, 1, 1, 0, 0, 0 },
  { 1, 1, 0, 0, 0, 0, 0 },
  { 0, 1, 0, 0, 1, 0, 0 },
  { 0, 0, 0, 1, 0, 1, 0 },
  { 0, 0, 0, 0, 1, 0, 1 },
  { 1, 0, 0, 0, 0, 1, 0 } };
  int nodes[7]; // вершины графа
  for (int i = 0; i < 7; i++)
    nodes[i] = 0; // исходно все вершины равны 0
  Queue.push(0); // помещаем в очередь первую вершину
  while (!Queue.empty())
  { // пока очередь не пуста
    int node = Queue.front(); // извлекаем вершину
    Queue.pop();
    nodes[node] = 2; // отмечаем ее как посещенную
    for (int j = 0; j < 7; j++)
    { // проверяем для нее все смежные вершины
      if (mas[node][j] == 1 && nodes[j] == 0)
      { // если вершина смежная и не обнаружена
        Queue.push(j); // добавляем ее в очередь
        nodes[j] = 1; // отмечаем вершину как обнаруженную
      }
    }
    cout << node + 1 << endl; // выводим номер вершины
  }
  cin.get();
  return 0;
}
```

Лекция 19.2

7 мая 2020 г. 22:20

19.2 Динамические структуры данных C++. Однонаправленны список, очередь, стек, дек, двунаправленный список, двунаправленный кольцевой список., дек на двунаправленном списке.

19.2.1 Линейные однонаправленные списки
19.2.2 Операции над списками с заглавным звеном
19.2.3 Ортогональные списки
19.2.4 Кольцевые списки
19.2.5 Списки магазинного типа
19.2.6 Списки магазинного типа. Очереди
19.2.7 Стек
19.2.8 Дек
19.2.9 Линейные двунаправленные списки
19.2.10 Двунаправленные кольцевые списки
19.2.11 Деки на базе двунаправленных списков

19.2.1 Линейные однонаправленные списки

Список - это совокупность объектов, называемых элементами списка, в которой каждый объект содержит информацию о местоположении связанного с ним объекта.

Если список располагается в оперативной памяти, то, как правило, информация для поиска следующего объекта - это адрес (указатель) в памяти. Если список хранится в файле на диске, то информация о следующем элементе может включать смещение элемента от начала файла, положение указателя записи/считывания файла, ключ записи и любую другую информацию, позволяющую однозначно отыскать следующий элемент списка.

Каждый элемент списка представим структурой языка C++ с двумя полями:

- информационное поле, которое в общем случае может содержать произвольное количество полей разных типов. Ясно, что если значением переменной p является ссылка на элемент списка, то присоединяя к обозначению (*p) с помощью точки имя соответствующего поля, можно манипулировать со значением любого поля информационной части;
- ссылка на следующий элемент списка.

```
struct node
{
int elem; //Информационный элемент звена списка
node *sled; // Указатель на следующее звено списка
};
```

Чтобы сделать действия, выполняемые при включении элемента в список (исключении элемента из списка), единообразными, обычно применяется следующий прием. В начало каждого списка добавляется заглавное звено (заголовок списка). Оно никогда не исключается из списка и перед ним в список никогда не включаются новые элементы.

Информационная часть заглавного звена или не используется вовсе, или используется для специальных целей. Например, в случае списка целых чисел она может содержать число, равное количеству звеньев в списке. Добавление заглавного звена в список приводит к тому, что теперь у всех элементов, в том числе и у первого, имеется предшественник, и действия по включению новых элементов в список (или исключение элементов из списка) проводятся единым способом.

Алгоритм удаления можно сформулировать следующим образом:

- заводим два указателя q и q1, один из которых будет "опережать" другой (пусть q1 опережает q);
- начальное значение q - это адрес расположения в памяти заглавного звена, а q1 - адрес расположения первого элемента списка;
- организуем цикл, пока q1 != NULL, то есть пока q1 "не доберется" до указателя последнего элемента списка;
- в теле цикла переместим указатели на следующую пару элементов списка (то есть для первого случая q будет содержать адрес первого звена списка, а q1 - второго), и удалим элемент, который адресуется значением q;
- после выполнения цикла у нас останется только заглавное звено, адресуемое указателем phead. Его также нужно удалить.

19.2.2 Операции над списками с заглавным звеном

Приведем алгоритм последовательного поиска звена с заданным значением информационного поля в однонаправленном списке, записанный в виде функции языка C++:

```
void POISK (node **phead, int el, node **Res)
// Поиск звена с элементом el в списке, заданном указателем *phead.
// В случае успешного поиска в *Res находится адрес
// звена списка, содержащего элемент el, в случае неуспеха в *Res помещается NULL.
```

```
{
node *t;


*Res = NULL;
t = *phead; t = (*t).sled;
while (t != NULL && *Res == NULL)
if ((*t).elem == el) *Res = t; else t = (*t).sled;
}
```

Рассмотрим алгоритм включения звена в список после звена, на которое указывает заданная ссылка.

Предположим, что имеется однонаправленный список с заглавным звеном, и необходимо вставить звено с заданным информационным полем после звена, на которое указывает ссылка Res. Вставка звена осуществляется по следующему алгоритму.

```
void VSTAV (node **Res, int el)
// Включение звена с информационным полем el
//после звена, на которое указывает ссылка *Res.
{
node *q;
q = new (node);
(*q).elem = el; (*q).sled = (**Res).sled;
(**Res).sled = q;
}
```

На этом шаге мы рассмотрим алгоритм удаления звена, расположенного после звена, на которое указывает ссылка Res.

```
void YDALE (node **Res)
// Удаление звена, расположенного после
// звена, на которое указывает ссылка *Res.
{
node *q;
q = (**Res).sled;
if (q!=NULL)
// Если звено, после которого нужно удалять,
// не является последним, то...
{ (**Res).sled = (*(**Res).sled).sled; delete q; }
else
cout<<"Звено с заданным элементом - последнее!\n";
}
```

19.2.3 Ортогональные списки

До сих пор мы рассматривали линейные структуры динамических переменных. Добавление к динамической переменной двух и более полей указателей создает возможность построения нелинейных структур. Дело в том, что при решении практических задач обычно не удается обойтись только линейными динамическими структурами данных (список, очередь, стек, дек и т.д.): приходится создавать структуры данных, максимально отражающие существо выполняемой исполнителем задачи.

Мы рассмотрим только простейшие нелинейные динамические структуры, которые называются ортогональными списочными структурами (ортогональными списками, многосвязными списками).

Более точно, ортогональными списками называется списочная структура данных, в которой узлы могут принадлежать более чем одному списку и содержать более одного указателя.

Горизонтальный линейный однонаправленный список с заглавным звеном мы будем называть гирляндой. Каждое звено этого списка содержит три поля, причем, если указатель P указывает на звено гирлянды, то:

- поле (*P).Key является информационным полем узла гирлянды;
- поле (*P).Next содержит указатель на следующее звено гирлянды;
- поле (*P).Trail содержит указатель на линейный однонаправленный список с заглавным звеном, который называется висюлькой

(английское слово Trail переводится как "тащиться, свисать, волочиться").

19.2.4 Кольцевые списки

Структура в виде линейного списка является весьма полезной, у нее имеется ряд недостатков. Сейчас мы рассмотрим другие методы организации списков и использование их с целью устранения этих недостатков.

Один из недостатков линейных списков заключается в том, что, зная указатель p на звено списка, мы не имеем доступа к предшествующим ему звеньям. Если производится просмотр списка, то для повторного обращения к нему исходный указатель на начало списка должен быть сохранен.

Предположим теперь, что в структуре линейного списка было сделано изменение, при котором поле sled последнего элемента содержит указатель "назад" или на заглавное звено, или на элемент, следующий за заглавным звеном.

Под кольцевым (циклическим) списком понимается список, в котором указатель из некоторой ячейки направлен на такое место в списке, откуда данная ячейка может быть достигнута снова [1]. Очевидно, что теперь мы можем из любого звена списка, "перемещаясь" по указателям достичь любого другого звена.

Кольцевым списком (кольцом) на базе линейного однонаправленного списка называется линейный список, в котором указатель из некоторого звена направлен на такое звено в списке, из которого данное звено может быть достигнуто вновь.

```
void POSTROENIE (node **phead)
//Построение кольцевого списка с заглавным звеном.
//*phead - указатель на заглавное звено.
{
int el;
struct node *t;
// Вначале сформируем заглавное звено.
*phead = new (node);
t = *phead; (*t).sled = NULL;
cout<<"Вводите элементы кольца: "; cin>>el;
while (el!=0)
{ (*t).sled = new (node); t = (*t).sled; (*t).elem = el;
cin>>el;
}
(*t).sled = (*(*phead)).sled;
}
```

19.2.5 Списки магазинного типа

Списки магазинного типа подразделяются на очереди, стеки и деки.

Очередь - список магазинного типа, в котором все включения производятся на одном конце списка, а все исключения делаются на другом его конце.

Очередь иногда называют циклической памятью или списком типа FIFO ("First In - First Out" - "первым включается - первым исключается").

Стек - список магазинного типа, в котором все включения и исключения звеньев делаются в одном конце списка.

Из стека мы всегда исключаем "младший" элемент из имеющихся в списке (тот элемент, который был включен позже других). Для очереди справедливо в точности противоположное правило: исключается всегда самый "старший" элемент; элементы "покидают" список в том порядке, в котором они в него вошли.

Существуют и другие названия стека: магазин, список типа LIFO ("Last In - First Out" - "последним включается - первым исключается"); "пуш-даун" список ("push-down"), реверсивная память, гнездовая память.

Дек ("Double-Ended Queue" - "двухсторонняя очередь") - список магазинного типа, в котором все включения и исключения звеньев делаются на обоих концах списка.

19.2.6 Списки магазинного типа. Очереди

Очередь - динамическая структура данных, так как с течением времени длина очереди (количество входящих в нее звеньев) изменяется. Например, все мы знакомы с очередью людей у кассы в магазине самообслуживания или очередью автомобилей у бензозаправочной станции. Вновь прибывшие становятся в один конец очереди и покидают ее после оплаты покупок или заправки с другого.

Другой, возможно более уместный, пример очереди может быть обнаружен в вычислительной системе с разделением времени, с которой одновременно работает несколько пользователей. Поскольку такая система обычно имеет единственный центральный

процессор и одну основную память, то эти ресурсы должны разделяться среди пользователей путем выделения короткого интервала времени на выполнение программы одного пользователя, за которым следует выполнение программы другого пользователя и так далее до тех пор, пока не будет вновь выполняться программа первого пользователя. Программы пользователей, ожидающие своего выполнения, образуют очередь ожидания. Управление такой очередью необязательно должно основываться на принципе "первым пришел" - "первым вышел", а можно использовать некоторую сложную приоритетную схему, учитывающую такие факторы, как используемый компилятор, требуемое время выполнения, желаемое число строк, выводимых на печать и так далее.

19.2.7 Стек

Стек - это специально организованная память, выборка и занесение данных в которую подчиняется дисциплине LIFO ("последним вошел - первым обслужен").

```
struct comp { //Структура с именем comp int Data; //Кикие то данные(могут быть любими, к примеру можно написать int key; char Data; или добавить еще какие то данные) comp *next;//Указатель типа comp на следующий зелемент };
```

19.2.8 Дек

Дек ("двухсторонняя очередь") на базе однонаправленного линейного списка - список магазинного типа на базе однонаправленного линейного списка, в котором все включения и исключения звеньев делаются на обеих концах очереди.

```
void Postroenie (node **nsp, node **ksp)
// Построение двунаправленного списка с заглавным звеном:
// *nsp - указатель на начало списка,
// *ksp - указатель на конец списка.
{
node *rsp;
int el;
*nsp = new(node);
rsp = *nsp;
(**nsp).pred = (**nsp).sled = NULL;
cout<<"Вводите последовательность:\n"; cin>>el;
while (el!=0)
{ (*rsp).sled = new(node); (*((*rsp).sled)).pred = rsp;
rsp = (*rsp).sled; (*rsp).sled = NULL; (*rsp).elem = el;
cin>>el; }
*ksp = rsp;
}
```

19.2.9 Линейные двунаправленные списки

Приступим к построению алгоритма формирования двунаправленного списка с заглавным звеном. Опишем необходимые переменные:

```
node *nsp; // Указатель на заглавное звено списка.
node *ksp; // Указатель на последнее звено списка.
node *rsp; // Рабочий указатель для перемещения по списку.
```

1. Построим заглавное звено:

```
nsp = new(node);
rsp = nsp;
(*nsp).pred = NULL;
(*
```

2. Создаем элемент списка:

```
cin>>el;
(*rsp).sled = new(node);
```

3. Заполняем поля элемента:

```
(*((*rsp).sled)).pred = rsp;
rsp = (*rsp).sled;
(*rsp).sled = NULL;
(*rsp).elem = el;
```

4. "Настраиваем" указатель на последний элемент списка:

```
ksp = rsp;
```

19.2.10 Двунаправленные кольцевые списки

В программировании двунаправленные списки часто преобразовывают следующим образом: "обычный" линейный двунаправленный список замыкают в своеобразное "кольцо": при движении по списку в прямом направлении можно от последнего звена переходить к звену, следующему прямо за заглавным звеном, а при движении в обратную сторону - от заглавного звена переходить сразу к последнему звену. В связи с этим мы будем называть двунаправленные списки подобного типа кольцевыми двунаправленными списками (двунаправленными кольцами или просто кольцами).

Подобная организация списка упрощает процедуру поиска или перебора звеньев с любого места списка с автоматическим переходом от конца к началу или наоборот.

19.2.11 Деки на базе двунаправленных списков

Дек на базе двунаправленного списка:

```
void BuiltDeck (node **nd, node **kd)
// Построение дека на базе двунаправленного
// списка с заглавным звеном.
// *nd - указатель на начало дека.
// *kd - указатель на конец дека.
{
node *q, *z;
int el;
// Построение заглавного звена.
*nd = new(node);
z = *nd;
(**nd).pred = (**nd).sled = NULL;
cout<<"Введите последовательность: \n";
cin>>el;
while (el!=0)
{ (*z).sled = new(node);
(*((*z).sled)).pred = z; z = (*z).sled;
(*z).sled = NULL; (*z).elem = el; cin>>el; }
if ((**nd).sled!=NULL)
{ q = *nd; *nd = (**nd).sled; (**nd).pred = NULL; *kd = z; delete q; }
else
{ delete *nd; *nd = *kd = NULL;}
}
```

Добавление звена в начало дека:

```
void InsLeft (node **nd, node **kd,int el)
// Вставка звена, содержащего элемент el, в дек слева.
// *nd - указатель на начало дека.
// *kd - указатель на конец дека.
{
node *q;
q = new(node);
```

```
(*q).elem = el;
if (*nd==NULL)
{ // Если дек пуст, то...
*nd = q; (*q).sled = (*q).pred = NULL; *kd = q;}
else
{ (*q).sled = *nd; (*q).pred = NULL; (**nd).pred = q; *nd = q;}
}
```

Добавление звена в конец дека:

```
void InsRight (node **nd, node **kd,int el)
// Добавление звена, содержащего элемент el, в дек справа.
// *nd - указатель на начало дека.
// *kd - указатель на конец дека.
{
node *q;


q = new(node);
(*q).elem = el;
if (*kd==NULL)
{// Если дек пуст, то...
*nd = q; (*q).sled = (*q).pred = NULL; *kd = q;}
else
{ (*q).sled = NULL; (*q).pred = *kd; (**kd).sled = q; *kd = q;}
}
```

Удаление звена из дека слева:

```
void DelLeft (node **nd, node **kd,int *el)
// Удаление звена из дека слева с помещением элемента
// удаляемого звена в переменную el.
// *nd - указатель на начало дека.
// *kd - указатель на конец дека.
{
node *q;
if ((**nd).sled!=NULL)
{ q = *nd; *el =(*q).elem; *nd = (**nd).sled; (**nd).pred = NULL; delete q;}
else
{ //В деке находится один элемент.
q = *nd; *el =(*q).elem; *nd = *kd = NULL;
delete q; cout<<"Дек пуст!\n";}
}
```

Удаление звена из дека справа:

```
void DelRight (node **nd, node **kd,int *el)
// Удаление звена из дека справа с помещением элемента
// удаляемого звена в переменную el.
// *nd - указатель на начало дека.
// *kd - указатель на конец дека.
{
node *q;
if ((**kd).pred!=NULL)
{ q = *kd; *el =(*q).elem; *kd = (**kd).pred; (**kd).sled = NULL; delete q;}
else
{ // В деке находится один элемент.
```

```
q = *kd; *el =(*q).elem; *nd = *kd = NULL;
delete q; cout<<"Дек пуст!\n";}
}
```

Лекция 19.3

12 мая 2020 г. 22:49

19.3 Динамические структуры данных C++. Дерево

19.3.1 Основные понятие, терминология, определения
19.3.2 Бинарные понятия поиска
19.3.3 Построение бинарного дерева поиска (рекурсивный алгоритм)
19.3.4 Анализ алгоритма поиска включениями
19.3.5 Дерево отрезков
19.3.6 Обход бинарного дерева
19.3.7 Вывод бинарного дерева
19.3.8 Построения бинарного дерева (нерекурсивный алгоритм)
19.3.9 Изображение бинарного дерева (нерекурсивный алгоритм)
19.3.10 Пример программы построения изображения бинарного дерева
19.3.11 Поиск вершины в бинарном дереве (нерекурсивный и рекурсивный)
19.3.12 Добавление вершины в бинарное дерево
19.3.13 Удаление вершины из бинарного дерева
19.3.14 Деревья минимальной высоты
19.3.15 Хеширование с помощью леса
19.3.16 Древовидно-кольцевая динамическая структура данных
19.3.17 Деревья Хаффмена
19.3.18 Деревья-формулы
19.3.19 Бинарные деревья с размеченными листьями
19.3.20 Представление бинарных деревьев Линейная скобочная запись (польская запись дерева)
19.3.21 Представление бинарных деревьев. Код Прюфера
19.3.22 Представление бинарных деревьев списками степеней исхода
19.3.23 Представление деревьев с помощью массивов
19.3.24 Идеально сбалансированные бинарные деревья
19.3.25 Балансированные по высоте деревья (АВЛ-деревья)
19.3.26 Математический анализ АВЛ-деревьев
19.3.27 Деревья Фибоначчи
19.3.28 Алгоритмы балансировки. Общие положения.
19.3.29 Построение АВЛ-дерева.
19.3.30 Поиск с помощью дерева
19.3.31 Разбор арифметического выражения
19.3.32 Дерево игр

19.3.1 Основные понятия, терминология, определения

Древовидные структуры получили широкое распространение при решении различных задач, связанных не только с генеалогией. Однако терминология, принятая при описании генеалогических деревьев, сохраняется и при решении другого рода задач. В дальнейшем древовидные структуры мы будем называть просто деревьями.

Определение

Вершина Y, которая находится непосредственно под узлом X, называется (непосредственным) потомком (сыном) X, вершина X в данном случае называется (непосредственным) предком (отцом) Y.

В этом случае, если вершина X находится на уровне i, то говорят, что вершина Y находится на уровне i+1. Мы будем считать, что корень дерева расположен на уровне 0. Максимальный уровень какой-либо вершины дерева называется его глубиной или высотой.

Максимальная степень всех вершин дерева называется степенью дерева.

Предком для узла x называется узел дерева, из которого существует путь в узел x.

Потомком узла x называется узел дерева, в который существует путь (по стрелкам) из узла x.

Родителем для узла x называется узел дерева, из которого существует непосредственная дуга в узел x.

Сыном узла x называется узел дерева, в который существует непосредственная дуга из узла x.

Дерево представляет собой типичную рекурсивную структуру (определяемую через саму себя). Как и любое рекурсивное определение, определение дерева состоит из двух частей – первая определяет условие окончания рекурсии, а второе – механизм ее использования.

19.3.2 Бинарные деревья поиска

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.

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

На рисунке выше только дерево а) является полным двоичным деревом.

19.3.3 Построение бинарного дерева (рекурсивный алгоритм)

```
void BuildTree (node **Tree)
// Построение бинарного дерева.
// *Tree - указатель на корень дерева.
{
int el;


*Tree = NULL; // Построено пустое бинарное дерево.
cout<<"Вводите ключи вершин дерева...\n";
cin>>el;
while (el!=0)
{ Search (el,Tree); cin>>el;}
}
```

В функции BuildTree() используется функция поиска вершины с данным ключом x - Search ():

```
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;
}
```

19.3.4 Анализ алгоритма поиска с включениями

Теорема Хопкрофта-Ульмана

Среднее число сравнений, необходимых для вставки n случайных элементов в дерево поиска, пустое вначале, равно O(nlog2n) для n>=1.

19.3.5 Дерево отрезков

На этом шаге мы рассмотрим дерево отрезков.

Дерево отрезков (впервые введенное Дж.Бентли в 1977 г.) - это структура данных, созданная для работы с такими интервалами на числовой оси, концы которых принадлежат фиксированному множеству из N абсцисс. Поскольку множество абсцисс фиксировано, то дерево отрезков представляет собой статическую структуру по отношению к этим абсциссам, т.е. структуру, на которой не разрешены

вставки и уделения абсцисс; кроме того эти абсциссы можно нормализовать, заменяя каждую из них ее порядковым номером при обходе их слева направо. Не теряя общности, можно считать эти абсциссы целыми числами в интервале [1,N].

Дерево отрезков - это двоичное дерево с корнем. Для заданных целых чисел l и r таких, что l<r, дерево отрезков T(l,r) строится рекурсивно следующим образом: оно состоит из корня v с параметрами B[v]=l и E[v]=r (B и E мнемонически соответствуют словам "Beginning" (начало) и "End" (конец), а если r-l>1, то оно состоит из левого поддерева T(l,(B[v]+E[v]) DIV 2) и правого поддерева T((B[v]+E[v]) DIV 2),r). Параметры B[v] и E[v] обозначают интервал [B[v],E[v]], включенный в [l,r], связанный с узлом v .

19.3.6 Обход бинарного дерева

Одной из необходимых операций при работе с деревьями является обход дерева, во время которого надо посетить каждый узел по одному разу и (возможно) вывести информацию, содержащуюся в вершинах.

Пусть в результате обхода надо напечатать значения поля данных всех вершин в определенном порядке. Существуют три варианта обхода:

- КЛП (корень – левое – правое): сначала посещается корень (выводится информация о нем), затем левое поддерево, а затем – правое;
- ЛПК (левое – правое – корень): сначала посещается левое поддерево, затем правое, а затем – корень.
- ЛКП (левое – корень – правое): сначала посещается левое поддерево, затем корень, а затем – правое;

19.3.7 Вывод 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
}
```

Само дерево "лежит на левом боку". Сначала выводится правое поддерево, причем очередная вершина "отступает" от левого края окна на величину, равную глубине вершины (расстояние от корня до этой вершины). Этот отступ реализуется циклом:

```
for (i=1; i<=l; i++) cout<<" ";
```

19.3.8 Построение бинарного дерева (нерекурсивный алгоритм)

```
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;}
}
```

19.3.9 Изображение бинарного дерева (нерекурсивный алгоритм)

```
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);
}
```

19.3.10 Пример программы построения изображения бинарного дерева (нерекурсиные алгоритмы)

```
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);
}
```

19.3.11 Поиск вершины в бинарном дереве (нерекурсивный и рекурсивный)

Нерекурсивный поиск проводится следующим образом:

```
#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));
}
}
```

19.3.12 Добавление вершины в бинарное дерево

```
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;
}
```

19.3.13 Удаление вершины из бинарного дерева

Алгоритм удаления из бинарного дерева вершины с заданным ключом различает три случая:

- 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.

19.3.14 Деревья минимальной высоты

Для большинства практических задач наиболее интересны такие деревья, которые имеют минимально возможную высоту при заданном количестве вершин n. Очевидно, что минимальная высота достигается тогда, когда на каждом уровне (кроме, возможно, последнего) будет максимально возможное число вершин.

Предположим, что задано n чисел (их количество заранее известно). Требуется построить из них дерево минимальной высоты. Алгоритм решения этой задачи предельно прост.

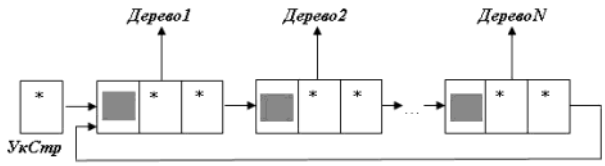
- Взять одну вершину в качестве корня и записать в нее первое нерассмотренное число.
- Построить этим же способом левое поддерево из n1 = n/2 вершин (деление нацело!).
- Построить этим же способом правое поддерево из n2 = n-n1-1 вершин.

19.3.15 Хеширование с помощью леса

Хэширование - это процесс, в котором вы подаёте на вход некоторого хэширующего алгоритма некоторые достаточно большие по объёму данные (допустим миллион байт) и получаете на выходе относительно короткую (допустим 32 байта), но при этом достаточно уникальную строку, которая позволяет отличить эти ваши данные (что были на входе) от каких-то других данных. Эта строка называется "хэш".

19.3.16 Древовидно-кольцевая динамическая структура данных

Используя построенные ранее процедуры для работы с древовидными структурами данных, можно построить структуру данных, представляющую собой совокупность кольцевой и древовидной структур. Указатели на корни древовидной структуры расположены в звеньях кольца. Само кольцо представим с помощью однонаправленного списка без заглавного звена.



19.3.17 Деревья Хаффмена

Построение кода Хаффмана сводится к построению соответствующего бинарного деревапо следующему алгоритму:

- Составим список кодируемых символов, при этом будем рассматривать один символ как дерево, состоящее из одного элемента с 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.

19.3.18 Деревья-формулы

ема арифметических и логических выражений проходит красной нитью через большую часть программирования, так как с ней связаны синтаксис и семантика языков программирования, компиляция, формальные языки, структуры данных, логика, рекурсия и вычислительная сложность. Поскольку эти выражения являются неотъемлемой частью фактически всех вычислительных программ, нужно иметь алгоритмы, распознающие и вычисляющие их как можно быстрее и эффективнее.

Чаще всего арифметические и логические выражения описываются при помощи бинарного дерева, которое в этом случае называется деревом-формулой. Все листья дерева-формулы соответствуют переменным или операндам, а все внутренние вершины соответствуют арифметическим операциям.

19.3.19 Бинарные деревья с размеченными листьями

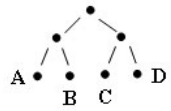
Абстрактным типом данных, представляющим значительный теоретический и практический интерес, является бинарный (двоичный) список, который рекурсивно определяется следующим образом:

бинарный список - это либо атомарный бинарный список (символ), либо упорядоченная пара бинарных списков.

Графическим представлением бинарного списка служат бинарное дерево с размеченными листьями:

((A . B) . (C . D))

Бинарный список



Бинарное дерево с размеченными листьями

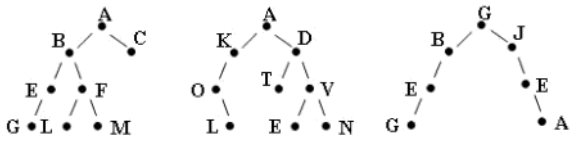
19.3.20 Представления бинарных деревьев. Линейная скобочная запись (польская запись дерева)

Для представления деревьев можно использовать линейные скобочные записи деревьев, т.е. представление деревьев в виде строк, содержащих символы, помечающие узлы дерева, а также открывающие и закрывающие круглые скобки. Между деревьями и их линейными скобочными записями существует взаимно однозначное соответствие.

Заметим, что линейная скобочная запись строится в результате того или иного обхода дерева.

Например, при левостороннем обходе возможен следующий рекурсивный алгоритм построения строки, представляющей линейную скобочную запись бинарного дерева:

- запишем в строку метку узла и открывающую круглую скобку, если узел дерева оказался внутренним, в противном случае запишем метку узла и на этом построение линейной скобочной записи закончим;
- припишем к строке справа линейные скобочные записи всех поддеревьев слева направо;
- припишем к строке справа закрывающую круглую скобку и на этом построение линейной скобочной записи закончим.



A(B(E(G)F(LM))C) A(K(O(L))D(TV(EN))) G(S(T(C))J(E(A)))

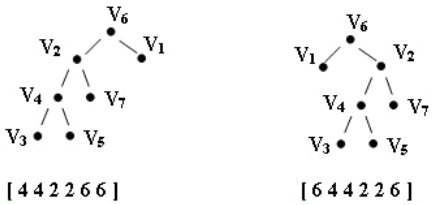
19.3.21 Представления бинарных деревьев. Код Прюфера

Пусть T - дерево с множеством вершин $\{V_1,V_2,...,V_N\}$. Будем считать, что номер вершины V_i равен i. Сопоставим дереву T последовательность $[a_1,a_2, ..., a_{N-1}]$ по следующему алгоритму [1]:

1. Полагаем i равным 1.
2. В последовательности
3. 1, 2, ..., N (*)

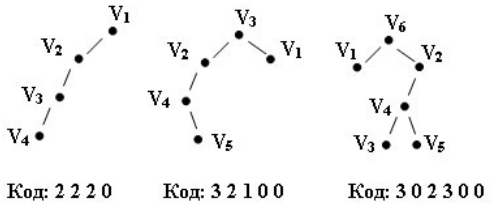
путем просмотра слева направо ищем номер *первого слева* листа. Пусть это будет b_i.

1. Ищем предка листа b_i. Пусть это будет узел a_i. Запоминаем его.
2. В последовательности (*) вычеркиваем b_i.
3. Из дерева T удаляем лист b_i.
4. Полагаем i := i + 1.
5. Если i<=N-1, то переходим к шагу 2.



19.3.22 Представления бинарных деревьев списками степеней исхода

Способ кодирования деревьев, рассмотренный ниже, позволяет однозначно кодировать любое *бинарное дерево*.



19.3.23 Представление бинарных деревьев с помощью массивов

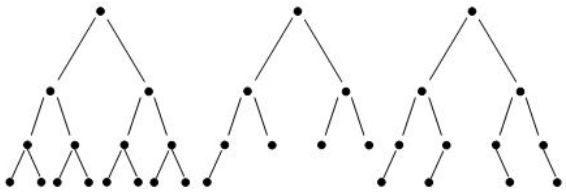
```
class Sort
{
private:
int A[N+1];
void Initialize(int (*)[], const int);
void Readjust (int (*)[], unsigned short &);
public:
void Tourn ();
void Vvod();
void Vyvod();
};
```

19.3.24 Идеально сбалансированные бинарные деревья

Определение.

Бинарное дерево назовем *идеально сбалансированным*, если для *каждой его вершины количество вершин в левом и правом поддереве различаются не более чем на 1*.

Изобразим несколько идеально сбалансированных деревьев:



Теорема.

Длина внутреннего пути в идеально сбалансированном дереве, содержащем n вершин, не превосходит величины:

$$(n+1)[\log_2 n] - 2 * 2^{[\log_2 n]} - 2$$

19.3.25 Балансированные по высоте деревья (АВЛ-деревья)

Бинарное дерево поиска называется *балансированным по высоте*, если *для каждой его вершины высота ее двух поддеревьев различается не более, чем на 1*. Деревья, удовлетворяющие этому условию, часто называют *АВЛ-деревьями* (по первым буквам фамилий их изобретателей *Г.М.Адельсона-Вельского* и *Е.М.Ландиса*).

19.3.26 Математический анализ АВЛ-деревьев

Теорема 1.

Обозначим T_h - АВЛ-дерево высотой h с количеством узлов N_h. Тогда

$$\log_2(N_h+1)=h+1\le\frac{1}{\log_2\frac{1+\sqrt{5}}{2}}\log_2\left\{\frac{\sqrt{5}(3-\sqrt{5})}{2}(N_h+1)+9-4\sqrt{5}\right\}$$

Теорема 2

Пусть T_h - АВЛ-дерево высоты h, имеющее N_h узлов. Тогда для *средней длины ветвей дерева* S_h при

имеет место следующая асимптотическая оценка:

$$S_h\sim\frac{1+\sqrt{5}}{2\sqrt{5}}\frac{\log_2 N_h}{\log_2\frac{1+\sqrt{5}}{2}}\sim 1.04229776903821\log_2 N_h$$

19.3.27 Деревья Фибоначчи

Дерево Фибоначчи порядка k определяется следующим образом [2, с.493; 6, с.274].

- если k=0, то дерево Фибоначчи *пусто*;
- если k=1, то дерево Фибоначчи состоит из единственного узла, ключ которого содержит 1;
- если k >=2, то корень дерева Фибоначчи содержит ключ F_k, левое поддерево есть *дерево Фибоначчи порядка* k-1, правое поддерево есть *дерево Фибоначчи порядка* k-2 с ключами в узлах, увеличенными на F_k.

19.3.28 Алгоритмы балансировки. Общее положение.

Однократный LL-поворот

```
p1 = (*p).Left;
(*p).Left = (*p1).Right; (*p1).Right = p;
(*p).bal = 0; p = p1;
```

Однократный RR-поворот

```
p1 = (*p).Right;
(*p).Right = (*p1).Left;
(*p1).Left = p;
(*p).bal = 0; p = p1;
```

Двухкратный LR-поворот

```
p1 = (*p).Left; p2 = (*p1).Right;
(*p1).Right = (*p2).Left; (*p2).Left = p1;
(*p).Left = (*p2).Right; (*p2).Right = *p;
p = p2;
```

Двухкратный RL-поворот

```
p1 =(*p).Right; p2 = (*p1).Left;
(*p1).Left = (*p2). Right; (*p2). Right = p1;
(*p).Right = (*p2). Left; (*p2). Left = *p;
p = p2;
```

19.3.29 Построение АВЛ-дерева

```
void Search (int x, node **p)
// x - ключ вершины, помещаемой в АВЛ-дерево.
// *p - указатель на корень АВЛ-дерева.
// h - флаг, сигнализирующий об увеличении высоты поддерева:
```

```
// TRUE - высота поддерева увеличилась,
// FALSE - высота поддерева не увеличилась.
// При первом обращении к функции Search() h=FALSE.
{
node *p1, *p2;
h = FALSE;
if (*p==NULL)
{ // Вершины в дереве нет; включить ее...
*p = new(node);
h = TRUE; (**p).Key = x;
(**p).Count = 1; (**p).Left = (**p).Right = NULL;
(**p).bal = 0; // Вершине присвоили нулевой баланс.
}
else
if (x<=(**p).Key)
{
Search (x,&((**p).Left)); // Вершина уже включена в дерево.
if (h==TRUE)
// Если высота поддерева увеличилась,
// то выросла левая дуга.
switch ((**p).bal)
{ case 1: (**p).bal = 0; h = FALSE; break;
// Предыдущая несбалансированность уравновесилась.
case 0: (**p).bal = -1; break; // Вес "склонился" влево.
case -1:
//Балансировка.
p1 = (**p).Left;
if ((*p1).bal==-1)
{//Однократный LL-поворот.
(**p).Left = (*p1).Right;
(*p1).Right = *p;
(**p).bal = 0; *p = p1;
}
else
{//Двукратный LR-поворот.
p2 = (*p1).Right;
(*p1).Right = (*p2).Left;
(*p2).Left = p1;
(**p).Left = (*p2).Right;
(*p2).Right = *p;
//Пересчет баланса вершины с указателем p.
if ((*p2).bal==-1) (**p).bal = 1;
else (**p).bal = 0;
// Пересчет баланса вершины с указателем p1.
if ((*p2).bal==1) (*p1).bal = -1;
else (*p1).bal = 0;
*p = p2;
}
(**p).bal = 0; h = FALSE;
break;
}
}
```

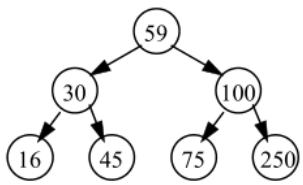
```
else //... иначе выросла правая дуга.
if (x>(**p).Key)
{
Search (x,&((**p).Right));
// Вершина уже включена в дерево.
if (h==TRUE)
// Если высота поддерева увеличилась,
// то выросла правая дуга.
switch ((**p).bal)
{ case -1: (**p).bal = 0; h = FALSE; break;
case 0: (**p).bal = 1; break;
case 1:
//Балансировка.
p1 = (**p).Right;
if ((*p1).bal==1)
{ //Однократный RR-поворот.
(**p).Right = (*p1).Left;
(*p1).Left = *p; (**p).bal = 0; *p = p1;
}
else
{ //Двухкратный RL-поворот.
p2 = (*p1).Left; (*p1).Left = (*p2).Right;
(*p2).Right = p1; (**p).Right = (*p2).Left;
(*p2).Left = *p;
// Пересчет баланса вершины с указателем p.
if ((*p2).bal==1) (**p).bal = -1;
else (**p).bal = 0;
//Пересчет баланса вершины с указателем p1.
if ((*p2).bal==-1) (*p1).bal = 1;
else (*p1).bal = 0; *p = p2;
}
(**p).bal = 0; h = FALSE; break;
}
}
}
```

19.3.30 Поиск с помощью дерева

Деревья очень удобны для поиска в них информации. Однако для быстрого поиска требуется предварительная подготовка – дерево надо построить специальным образом.

Предположим, что существует массив данных и с каждым элементом связан ключ - число, по которому выполняется поиск. Пусть ключи для элементов таковы:

59, 100, 75, 30, 16, 45, 250



Для этих данных нам надо много раз проверять, есть ли среди ключей заданный ключ x, и если есть, то вывести всю связанную с этим

элементом информацию.

Если данные организованы в виде массива (без сортировки), то для поиска в худшем случае надо сделать n сравнений элементов (сравнивая последовательно с каждым элементом пока не найдется нужный или пока не закончится массив).

Теперь предположим, что данные организованы в виде дерева, показанного на рисунке. Такое дерево (оно называется дерево поиска) обладает следующим важным свойством:

Значения ключей всех вершин левого поддерева вершины x меньше ключа x, а значения ключей всех вершин правого поддерева x больше или равно ключу вершины x.

Для поиска нужного элемента в таком дереве требуется не более 3 сравнений вместо 7 при поиске в списке или массиве, то есть поиск проходит значительно быстрее. С ростом количества элементов эффективность поиска по дереву растет.

19.3.31 Разбор арифметического выражения

трансляторах широко используется постфиксная запись выражений, которая получается в результате обхода в порядке ЛПК (левое – правое – корень). В ней знак операции стоит после обоих операндов:

a b + c d - 1 /

Порядок выполнения такого выражения однозначно определяется следующим алгоритмом, который использует стек:

Пока в постфиксной записи есть невыбранные элементы,

- взять очередной элемент;
- если это операнд (не знак операции), то записать его в стек;
- если это знак операции, то
- выбрать из стека второй операнд;
- выбрать из стека первый операнд;
- выполнить операцию с этими данными и результат записать в стек.

19.3.32 Дерево игр

Одно из применений деревьев - игры с компьютером. Рассмотрим самый простой пример – игру в крестики-нолики на поле 3 на 3.

Программа должны анализировать позицию и находить лучший ход. Для этого нужно определить оценочную функцию, которая получая позицию на доске и указание, чем играет игрок (крестики или нолики) возвращает число – оценку позиции. Чем она выше, тем более выгодна эта позиция для игрока. Примером такой функции может служить сумма строк, столбцов и диагоналей, которые может занять игрок минус такая же сумма для его противника.

Однако, в этой ситуации программа не ведет просчет вперед и не оценивает позиции, которые могут возникнуть из текущей. Это недостаточно для предсказания исхода игры. Хотя для крестиков-ноликов можно перебрать все варианты и найти выигрышную позицию, большинство игр слишком сложно, чтобы допускать полный перебор.

Выбор хода может быть существенно улучшен, если просматривать на несколько ходов вперед. Уровнем просмотра называется число будущих рассматриваемых ходов. Начиная с любой позиции можно построить дерево возможных позиций, получающихся после каждого хода игры. Для крестиков-ноликов ниже приведено дерево с уровнем просмотра 3 для того случая, когда крестики сделали первый ход в центр доски.

Лекция 19.4

14 мая 2020 г. 0:09

19.4 Граф

19.4.1. Представление и обход графов. Основная терминология
19.4.2. Представления графов. Список ребер
19.4.3. Представления графов. Списки смежности
19.4.4. Реализация простейших операций над графами, представленными списками смежности
19.4.5. Представления графов. Ортогональные списки смежности
19.4.6. Представления графов. Структуры Вирта
19.4.7. Реализация простейших операций над графом, представленным структурой Вирта
19.4.8. Пример программы, реализующей простейшие операции над графом, представленным структурой Вирта
19.4.9. Модифицированные структуры Вирта
19.4.10. Представление отношений
19.4.11. Топологическая сортировка
19.4.12. Первый пример использования топологической сортировки
19.4.13. Второй пример использования топологической сортировки
19.4.14. Третий пример использования топологической сортировки
19.4.15. Четвертый пример использования топологической сортировки
19.4.16. Понятие о методе PERT
19.4.17. Представление грамматики
19.4.18. Обход графов (общие сведения)
19.4.19. Обход графов в глубину
19.4.20. Обход графов в ширину
19.4.21. Путь между фиксированными вершинами
19.4.22. Эйлеровы пути и циклы
19.4.23. Алгоритмы на графах. Кратчайшие пути между всеми парами вершин. Алгоритм Уоршалла
19.4.24. Применение алгоритма Уоршалла. Вычисление длин кратчайших путей между вершинами
19.4.25. Применение алгоритма Уоршалла. Отыскание компонент сильной связности
19.4.26. Применение алгоритма Уоршалла. Определение рекурсивности подпрограммы
19.4.27. Кратчайшие пути между всеми парами вершин. Контуры в ориентированных графах
19.4.28. Связность. Вычисление компонент связности
19.4.29. Связность. Нахождение компонент двусвязности
19.4.30. Остовы. Построение остова
19.4.31. Остовы. Построение остова наименьшей стоимости
19.4.32. Остовы. Построение фундаментального множества циклов
19.4.33. Алгоритмы с возвратом (общие сведения)
19.4.34. Построение алгоритмов с возвратом
19.4.35. Гамильтоновы циклы
19.4.36. Клики
19.4.37. Независимые множества вершин графа
19.4.38. Раскраски
19.4.39. Алгоритмы раскраски графа
19.4.40. Изоморфизм

19.4.1 Граф

Определение.

Простой путь в неориентированном графе - это последовательность смежных ребер (v_1,v_2), (v_2,v_3),..., (v_{k-1},v_k), таких, что все v_i кроме, быть может, v_1 и v_k различны. *Простой путь в ориентированном графе* - это последовательность смежных одинаково ориентированных дуг (v_1,v_2), (v_2,v_3),..., (v_{k-1},v_k), таких, что все v_i , кроме, быть может, v_1 и v_k различны. Число ребер, составляющих простой путь, называется *длиной простого пути*.

Простой путь называется:

- *в неориентированном графе - цепью*, а
- *в ориентированном графе - путем*.

Простой путь, начинающийся и заканчивающийся в одной и той же вершине, называется:

- *в неориентированном графе - циклом*, а
- *в ориентированном графе - контуром*.

Расстояние между вершинами u *и* v в связном неориентированном графе - это минимальная длина цепи между u и v.

Теорема.

Для графа G, имеющего p вершин и q ребер, следующие утверждения эквивалентны:

- G - дерево;
- любые две вершины в G соединены единственной цепью;
- G - связный граф и p=q+1;
- G - ациклический граф и p=q+1;
- G - ациклический граф, и если любую пару несмежных вершин соединить ребром x, то в графе G+xбудет точно один цикл;
- G - связный граф, отличный от K_p для p>=3, и если любую пару несмежных вершин соединить ребром x, то в графе G+x будет точно один цикл;
- G - граф, отличный от K_3UK_1 и K_3UK_2, p=q+1, и если любую пару несмежных вершин соединить ребром x, то в графе G+x будет точно один цикл.

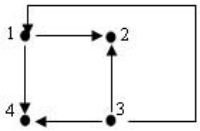
19.4.2 Представление графов. Список ребер

Более экономным в отношении памяти (особенно в случае так называемых неплотных графов, когда |E| гораздо меньше $|V|^2$) по сравнению с матрицей смежностей является метод представления графа с помощью *структуры смежности*, которая является в простейшем случае списком пар, соответствующих его ребрам [1, с.354].

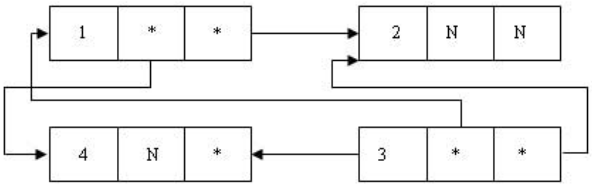
Пара <x,y>, входящая в список ребер, соответствует ребру {x,y} в случае неориентированного графа и дуге (x,y), если граф ориентированный.

19.4.3 Представление графов. Списки смежности

Например, если ориентированный граф имеет следующий вид:



то без всяких ухищрений представим его в памяти следующим образом (символ N обозначает NULL):



19.4.4 Реализация простейших операций над графами, представленными списками смежности

1. Построение списков смежности, соответствующих данному ориентированному графу. Перед первым обращением к функции MakeGraph (создание графа) необходима инициализация списков смежности:

```
for (i=0;i<N;i++) beg[i] = NULL;
void MakeGraph (svqz beg[N])
// Построение списков смежности beg графа.
{
int x,y;
svqz ukzv,uzel; //Рабочие указатели.


cout<<"Вводите начало дуги: ";
cin>>x;
while (x!=0)
```

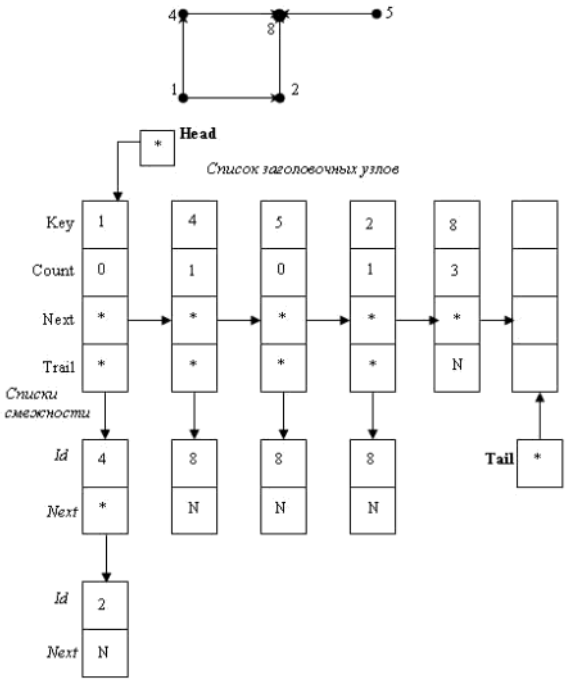
```
{
cout<< "Вводите конец дуги: ";
cin>>y;
AddGraph (x,y,beg);
cout<< "Вводите начало дуги: "; cin>>x;
}
}
```

2. Вывод содержимого списков смежности, соответствующих ориентированному графу.

```
void PrintGraph (svqz beg[N])
{
int i;
svqz ukzv; //Рабочий указатель.
for (i=1;i<N;i++)
{
cout<<i<<" ...";
ukzv = beg[i];
if (ukzv==NULL) cout<<"Пустой список!\n";
else {
while (ukzv!=NULL)
{ cout<< (*ukzv).Key; ukzv = (*ukzv).Sled; }
cout<<endl; }
}
}
```

19.4.5 Представление графов. Ортогональные списки смежности

На рисунке, приведенном ниже, изображен граф и его представление с помощью структуры данных, которую мы будем называть *ортогональными списками смежности*.



Если указатель P указывает на заголовочный узел, представляющий вершину Q графа, то:

- поле (*P).Key содержит информацию, связанную с этой вершиной Q;
- поле (*P).Count содержит количество вершин, "предшествующих" данной;
- поле (*P).Next содержит указатель на заголовочный узел, представляющий следующую вершину графа (если такая вершина есть) в списке заголовочных узлов;
- каждый заголовочный узел является заглавным звеном списка узлов второго типа, называемых *дуговыми узлами*. Такой список мы также будем называть *списком смежности*. Каждый узел списка смежности представляет конечную вершину некоторой дуги графа. Поле (*P).Trail указывает на список смежности, представляющий дуги, выходящие из вершины Q графа (английское слово "Trail" переводится как *"тащиться, свисать, волочиться"*).

19.4.6 Представление графов. Структуры Вирта

Если указатель P указывает на заголовочный узел, представляющий вершину Q графа, то:

- поле (*P).Key содержит информацию, связанную с вершиной Q;
- поле (*P).Count содержит количество вершин, "предшествующих" данной;
- поле (*P).Next содержит указатель на заголовочный узел, представляющий следующую вершину графа (если такая вершина есть) в списке заголовочных узлов;
- каждый заголовочный узел является заглавным звеном списка узлов второго типа, называемых *дуговыми узлами*. Такой список мы будем называть *списком смежности*. Поле (*P).Trail указывает на список смежности, представляющий дуги, выходящие из вершины Q графа (английское слово "Trail" переводится как *"тащиться, свисать, волочиться"*).

19.4.7 Реализация простейших операций над графом, представленным структурой Вирта.

1. Поиск в структуре Вирта заголовочного узла с заданным значением поля Key *и добавление его в список заголовочных узлов в случае отсутствия.*

```
Lref SearchGraph (int w, Lref Head, Lref *Tail)
//Функция возвращает указатель на заголовочный узел
//с ключом w. Если заголовочный узел отсутствует,
//то он добавляется в список. Head - указатель на
//структуру Вирта.
{
Lref h;


h = Head; (**Tail).Key = w;
while ((*h).Key!=w) h = (*h).Next;
if (h==*Tail)
{//В списке заголовочных узлов нет узла с ключом w.
//Поместим его в конец списка Head.
*Tail = new (Leader); (*h).Count = 0;
(*h).Trail = NULL; (*h).Next = *Tail;}
return h;
}
```

2. Построение структуры Вирта, соответствующей данному ориентированному графу. Перед самым первым обращением к функции создания графа MakeGraph () инициализируем список заголовочных узлов:

```
Head = new (Leader); Tail = Head;.

void MakeGraph (Lref *Head, Lref *Tail)
//Функция возвращает указатель *Head на структуру
// Вирта, соответствующую ориентированному графу.
{
int x,y;
Lref p,q; //Рабочие указатели.
Tref t,r; //Рабочие указатели.
```

```
Boolean Res; //Флаг наличия дуги.


cout<<"Вводите начальную вершину дуги: ";
cin>>x;
while (x!=0)
{ cout>>"Вводите конечную вершину дуги: "; cin>>y;
//Определим, существует ли в графе дуга (x,y)?
p = SearchGraph (x,*Head,Tail); q = SearchGraph (y,*Head,Tail);
r = (*p).Trail; Res = FALSE;
while ((r!=NULL) && (!Res))
if ((*r).Id==q) Res = TRUE;
else r = (*r).Next;
if (!Res) //Если дуга не существует, то поместим её в граф.
{ t = new (Trailer); (*t).Id = q; (*t).Next = (*p).Trail;
(*p).Trail = t; (*q).Count++; }
cout<< "Вводите начальную вершину дуги: "; cin>>x; }
}
```

19.4.8 Пример программы, реализующей простейшие операции над графом, представленным структурой Вирта.

```
Lref Spisok::SearchGraph (int w)
//Функция возвращает указатель на заголовочный узел
//с ключом w. Если заголовочный узел отсутствует, то он
//добавляется в список. Head - указатель на структуру Вирта.
{
Lref h;


h = Head; (*Tail).Key = w;
while ((*h).Key!=w) h = (*h).Next;
if (h==Tail)
//В списке заголовочных узлов нет узла с ключом w.
//Поместим его в конец списка Head.
{ Tail = new (Leader); (*h).Count = 0;
(*h).Trail = NULL; (*h).Next = Tail; }
return h;
}


Lref Spisok::Search (int w)
//Функция возвращает указатель на заголовочный узел
//ключом w. Если узел отсутствует, то функция возвращает NULL .
{
Lref h = Head;
(*Tail).Key = w; //Поиск "с барьером".
while ((*h).Key!=w) h = (*h).Next;
if (h==Tail) //В списке заголовочных узлов нет узла с ключом w.
```

```
h = NULL;
return h;
}


void Spisok::MakeGraph ()
//Функция возвращает указатель Head на структуру
//Вирта, соответствующую ориентированному графу.
{
int x,y;
Lref p,q; //Рабочие указатели.
Tref t,r; // Рабочие указатели.
Boolean Res; //Флаг наличия дуги.


cout<<"Вводите начальную вершину дуги: ";
cin>>x;
while (x!=0)
{
cout<<"Вводите конечную вершину дуги: "; cin>>y;
//Определим, существует ли в графе дуга (x,y)?
p=SearchGraph (x); q=SearchGraph (y);
r = (*p).Trail; Res = FALSE;
while ((r!=NULL)&&(!Res))
if ((*r).Id==q) Res = TRUE;
else r = (*r).Next;
if (!Res) //Если дуга отсутствует, то поместим её в граф.
{ t = new (Trailer); (*t).Id = q;
(*t).Next = (*p).Trail; (*p).Trail = t; (*q).Count++; }
cout<<"Вводите начальную вершину дуги: "; cin>>x;
}
}
```

19.4.9 Модифицированные структуры Вирта

```
typedef struct L *Lref; //Тип: указатель на заголовочный узел.
typedef struct T *Tref; //Тип: указатель на дуговой узел.
//Описание типа заголовочного узла.
typedef struct L
{
int Key; //Имя заголовочного узла.
int Count; //Количество предшественников.
int Count1; //Количество последующих вершин.
Tref Pred; //Указатель на список смежности, содержащий "предшественников".
Tref Trail; //Указатель на список смежности, содержащий "последователей".
Lref Next; //Указатель на следующий узел в списке заголовочных узлов.
} Leader;
```

```
//Описание типа дугового узла.
typedef struct T
{
Lref Id;
Tref Next;
} Trailer;
```

19.4.10 Представление отношений

Определение.

Под *бинарным отношением на множестве* M мы понимаем произвольное подмножество E множества MxM.

Рассмотрим ориентированный граф, в котором любая дуга (v,w) имеется только в том случае, если элементы v и w, представляемые вершинами v и w, находятся в *данном бинарном отношении* r, т.е. vrw. Такой граф является исчерпывающей и наглядной формой представления отношения r, так как он полностью перечисляет все упорядоченные пары вершин-элементов, для которых отношение r имеет место.

Графы отношений могут обладать специальными свойствами: рефлексивностью, симметричностью, антисимметричностью, транзитивностью и т.д., отражающими соответствующие свойства отношений.

19.4.11 Топологическая сортировка

Множество M называется *частично упорядоченным*, если над его элементами определено отношение, которое мы назовем *"x предшествует y"* и обозначим x<<y, удовлетворяющее следующим свойствам для любых элементов x, y и z из M:

- не x<<x (*антирефлексивность*),
- если x<<y, то не y<<x (*антисимметричность*),
- если x<<y и y<<z, то x<<z (*транзитивность*).

19.4.12 Первый пример использования топологической сортировки

Закодируем сорта вин следующим образом:

1 - белое сухое,

2 - белое бархатистое,

3 - белое сладкое,

4 - красное легкое,

5 - красное крепкое.

Тогда система отношений порядка примет вид:

1<2, 2<3, 4<5, 1<4, 1<5, 2<4, 2<5, 4<3, 5<3,

а программа топологической сортировки выдаст следующий результат:

1 2 4 5 3

Но так как не принято подавать за обедом более *четырех вин*, а шампанское в нашем перечне сортов вин отсутствует, то возможны лишь следующие варианты:

2 4 5 3

1 4 5 3

1 2 5 3

1 2 4 3

1 2 4 5

19.4.13 Второй пример использования топологической сортировки

```
void Spisok::Sorting (Lref Head)
// Сортировка списка элементов по полю CountF
```

```
// в порядке убывания.
// 21.10.93 (c) Коврижных Д. & Швецкий М.В.
{
Lref UkZv_1,UkZv_2;
int A,B,C;
Tref D;
Lref UkZv_3; // Рабочий указатель.
Tref UkZv_4; // Рабочий указатель.


UkZv_1 = Head;
while ( UkZv_1!=NULL )
{
UkZv_2 = UkZv_1->Next;
while (UkZv_2!=NULL)
{
if (UkZv_1->CountF < UkZv_2->CountF)
{
A = UkZv_1->Key;
B = UkZv_1->Count;
C = UkZv_1->CountF;
D = UkZv_1->Trail;


UkZv_1->Key = UkZv_2->Key;
UkZv_1->Count = UkZv_2->Count;
UkZv_1->CountF = UkZv_2->CountF;
UkZv_1->Trail = UkZv_2->Trail;


UkZv_2->Key = A;
UkZv_2->Count = B;
UkZv_2->CountF = C;
UkZv_2->Trail = D;


UkZv_3 = Head;
while (UkZv_3!=NULL)
{
UkZv_4 = UkZv_3->Trail;
while (UkZv_4!=NULL)
{
if (UkZv_4->Id==UkZv_1)
UkZv_4->Id = UkZv_2;
else
if ( UkZv_4->Id==UkZv_2 ) UkZv_4->Id = UkZv_1;
UkZv_4 = UkZv_4->Next;
}
UkZv_3 = UkZv_3->Next;
```

```
}
}
UkZv_2 = UkZv_2->Next;
}
UkZv_1 = UkZv_1->Next;
}
}
```

19.4.14 Третий пример использования топологической сортировки

- для системы отношений частичного порядка:
- 8>5, 2>3, 2>1, 5>2, 4>1, 3>4, 4>6, 6>9, 9>5

программа обнаруживает цикл вида: 2,5,9,6,4,3,2.

- для системы отношений частичного порядка [1, с.648]:
- 9>2, 3>7, 7>5, 5>8, 8>6, 4>6, 1>3, 7>4, 9>5, 2>8, 1>9, 10>1, 6>10.

программа обнаруживает цикл вида: 2,9,1,10,6,8,2.

19.4.15 Четвертый пример использования топологической сортировки

В произвольном ориентированном бесконтурном графе вершины можно перенумеровать так, что каждая дуга будет иметь вид (v_i,v_j), где i<j.

Для *доказательства леммы* предложим алгоритм, конструирующий такую нумерацию (в виде программы на языке С++).

Алгоритм основывается на следующем простом факте: в произвольном бесконтурном графе существует вершина, в которую не заходит ни одна дуга.

Чтобы убедиться в этом, выберем произвольную вершину w_1 графа, затем некоторую вершину w_2, такую что w_1<-w_2, и т.д. Через конечное число шагов мы должны дойти до некоторой вершины w_i, в которую не заходит ни одна дуга, ибо в силу бесконтурности ни одна вершина не может повторяться в последовательности $w_1,w_2,w_3,\ldots$

19.4.16 Понятие о методе PERT

Сетевое планирование - совокупность методов, использующих сетевую модель как основную форму представления информации об управляемом комплексе работ. Использование сетевого планирования позволяет повысить качество планирования и управления при реализации комплекса работ, например, дает возможность четко координировать деятельность всех сторон (организаций), участвующих в реализации, выделять наиболее важные задачи, определять сроки реализации, а также координировать план его реализации.

Сетевая модель - информационная модель реализации некоторого комплекса взаимосвязанных работ, рассматриваемая как ориентированный граф без контуров, отображающий естественный порядок выполнения этих работ во времени; может содержать некоторые дополнительные характеристики (например, время, стоимость, ресурсы), относящиеся к отдельным работам и (или) к комплексу в целом.

Предположим, что шеф-повар получил заказ приготовить яичницу из одного яйца. Вся процедура ее приготовления может быть разбита на ряд отдельных подзадач:

Взять яйцо ----> Разбить яйцо ----> Взять жир ---->

----> Положить жир на сковороду ----> Растопить жир ---->

----> Вылить яйцо на сковороду ---->

----> Ждать, пока яичница не изжарится ----> Снять яичницу

Некоторые из этих подзадач должны предшествовать другим (например, задача "взять яйцо" должна предшествовать задаче "разбить яйцо"). Ряд подзадач может выполняться параллельно (например, задачи "взять яйцо" и "растопить жир"). Шеф-повар хотел бы выполнить заказ как можно быстрее, при этом предполагается, что число его помощников не ограничено.

19.4.17 Представление грамматики

Плексы (сплетения) - структуры данных, служащие для представления графов, имеющих ребра, принадлежащих нескольким

различным семействам. *И.Флорес* называет плексы *ветвящимися списками*, а *Д.Кнут - Списками* (с большой буквы!).

Отметим, что плексы широко применяются в *трансляторах* для отображения деревьев и графов в памяти ЭВМ, так как плексы лучше других типов структур отображают *многоуровневые структуры данных* в памяти ЭВМ.

19.4.18 Обход графов (общие сведения)

Два наиболее распространенных алгоритма обхода графов называются:

- *обход графа в глубину (поиск в глубину* (англ. Depth First Search)) и
- *обход графа в ширину (поиск в ширину* (англ. Breadth First Search)).

19.4.19 Обход графов в глубину

Обход в глубину (называемый иногда *стандартным обходом*), есть обход графа по следующим правилам:

- находясь в вершине x, нужно двигаться в любую другую, ранее не посещенную вершину (если таковая найдется), одновременно запоминая дугу, по которой мы впервые попали в данную вершину;
- если из вершины x мы не можем попасть в ранее не посещенную вершину или таковой вообще нет, то мы возвращаемся в вершину z, из которой впервые попали в x, и продолжаем обход в глубину из вершины z.

19.4.20 Обход графов в ширину

Перейдем теперь к другому алгоритму обхода графа, известному под названием *обход в ширину* (*поиск в ширину*). Прежде чем описать его, отметим, что при обходе в глубину чем *позднее* будет посещена вершина, тем *раньше* она будет использована. Это прямое следствие того факта, что просмотренные, но еще не использованные вершины накапливаются в стеке.

Обход графа в ширину, грубо говоря, основывается на *замене стека очередью*. После такой модификации чем раньше посещается вершина (помещается в очередь), тем раньше она используется (удаляется из очереди). Использование вершины происходит с помощью просмотра сразу всех еще непросмотренных вершин, смежных этой вершины. Таким образом, "поиск ведется как бы во всех возможных направлениях одновременно"

19.4.21 Путь между фиксированными вершинами

Оба вида обхода графа - в глубину и в ширину могут быть использованы *для нахождения пути (цепи) между фиксированными вершинами* u *и* v. Достаточно начать обход графа с вершины v и вести его до момента посещения вершины u.

Преимуществом обхода графа в глубину является тот факт, что в момент посещения вершины u стек содержит последовательность вершин, определяющую путь (цепь) из v в u. Это становится очевидным, если отметить, что каждая вершина, помещаемая в стек, является смежной вершиной верхнего элемента стека.

Однако *недостатком* использования алгоритма обхода графа в глубину для поиска пути между данными вершинами является то, что полученный таким образом путь в общем случае *не будет кратчайшим путем (кратчайшей цепью)*.

19.4.22 Эйлеровы пути и циклы

Эйлеровым путем в графе называется путь, проходящий через каждое ребро графа только один раз, т.е. путь $v_1, v_2 ..., v_{m+1}$, такой, что каждое ребро e, принадлежащее E, появляется в последовательности v_1, v_2..., v_{m+1} в точности один раз как $e=\{v_i, v_{i+1}\}$. 0 Иногда граф, обладающий эйлеровым путем, называют *полуэйлеровым* [2, с.43].

Если $v_1=v_{m+1}$, то такой путь называется *эйлеровым циклом*, а граф, обладающий эйлеровым циклом, называют *эйлеровым графом* [2, с.43].

Задача существования эйлерова пути в заданном графе была решена Л.Эйлером в 1736 г., и представленное им необходимое и достаточное условие существования такого пути считается первой в истории теоремой теории графов.

19.4.23 Алгоритмы на графах. Кратчайшие пути между всеми парами вершин. Алгоритм Уоршалла

Обозначим *булевскую сумму* C двух матриц A и B размера n*n как

$C = A \cup B$

, а *булевское произведение* D -

$D = A \cap B$

.

Элементы матриц C и D задаются соотношениями

$$\forall i, j = 1,2,\ldots,n$$
$$c_{ij} = a_{ij} OR\, b_{ij}$$
$$d_{ij} = \bigcup_{k=1}^{n} (a_{ik} \cap b_{kj})$$

Заметим, что элемент d_{ij} легко получается путем просмотра i-й строки матрицы A слева направо и одновременно j-го столбца матрицы B сверху вниз. Если k-й элемент в строке матрицы A и k-й элемент в столбце матрицы B равны 1 для какого-нибудь k, то d_{ij}=1. В противном случае d_{ij}=0.

Булевы матрицы более экономичны в вычислительном отношении, чем целочисленные. Действительно, запоминание булевой матрицы требует *меньшего объема* оперативной памяти ЭВМ *по сравнению с целочисленной матрицей той же размерности*. Кроме того, выполнение на компьютере логических операций над булевыми матрицами требует меньшего объема вычислений, чем над целочисленными матрицами тех же размерностей.

Матрица смежностей, так же как и путевая матрица, является булевской матрицей. Заметим, что

$$\forall r = 2, 3, \ldots \; A \cap A^{(r-1)} = A^{(r)}$$

.

Единственная разница между A^2 и $A^{(2)}$ заключается в том, что $A^{(2)}$ является булевской матрицей и элемент на пересечении i-й строки и j-го столбца $A^{(2)}$ равен 1 в том случае, когда существует по крайней мере один путь длины 2 из v_i в v_j. Аналогичное положение имеет место для A^3 и $A^{(3)}$ и в общем случае для A^r и $A^{(r)}$ при любом целом положительном r.

19.4.24. Применение алгоритма Уоршалла. Вычисление длин кратчайших путей между вершинами

Алгоритм Уоршалла может быть модифицирован с целью получения матрицы, содержащей *длины кратчайших путей между вершинами* [1, с.441].
Идея этого алгоритма следующая [2,с.131]. Обозначим через $d^{(m)}_{ij}$ длину кратчайшего из путей из v_i в v_j с промежуточными вершинами в множестве $\{v_1, \ldots, v_m\}$.
Тогда имеем следующие уравнения, объединенные в систему:

$d^{(0)}_{ij} = a_{ij}$

$d^{(m)}_{ij} = \min(d^{(m)}_{ij}, d^{(m)}_{im} + d^{(m)}_{mj})$

Обоснование второго уравнения достаточно простое. Рассмотрим кратчайший путь из v_i в v_j с промежуточными вершинами из множества $\{v_1, \ldots, v_m, v_{m+1}\}$. Если этот путь не содержит v_{m+1}, то деля путь на отрезки от v_i до v_{m+1} и от v_{m+1} до v_j, получаем равенство

$d^{(m+1)}_{ij} = d^{(m)}_{im} + d^{(m)}_{mj}$.

Вышеприведенные уравнения дают возможность легко вычислить расстояния

$d(v_i,v_j) = d^{(n)}_{ij}$, $1 <= i, j <= n$.

19.4.25. Применение алгоритма Уоршалла. Отыскание компонент сильной связности

Определение.

Ориентированный граф называется *сильно связным*, если для любой пары вершин каждая из них достижима из другой [1, с.11].

Сильно связный *подграф* ориентированного графа называется *зоной*.

Зона, максимальная относительно включения вершин, называется *компонентой сильной связности (бикомпонентой).*

Знание матрицы достижимости позволяет выявить все компоненты сильной связности в ориентированном графе.
Определим *поэлементное (адамарово) произведение* матриц $B=(b_{ij})$ и $C=(c_{ij})$ по правилу: $B * C=(b_{ij} * c_{ij})$.
Тогда вершины бикомпоненты ориентированного графа, содержащей вершину x_i, определяются единичными элементами i-й строки матрицы $R * R^T$, где R^T - транспонированная матрица достижимости.
Отсюда следует, что, вычислив поэлементное произведение матриц R и R^T и разбив все ненулевые строки этого произведения на группы одинаковых строк, мы можем определить множество вершин каждой бикомпоненты, поскольку номера строк *однозначно* определяют номера вершин, входящих в бикомпоненту.

19.4.26. Применение алгоритма Уоршалла. Определение рекурсивности подпрограммы

В некоторых языках программирования программист может задать рекурсивность процедуры в явной форме. В других языках для того, чтобы определить, какие из процедур являются рекурсивными, можно использовать идеи, вытекающие из теории графов.
Вначале необходимо заметить, что *рекурсивная процедура необязательно вызывает себя непосредственно*. Если процедура P_1 вызывает P_2, процедура P_2 вызывает P_3, ..., процедура P_{n-1} вызывает P_n и процедура P_n вызывает P_1, то

процедура P_1 является *рекурсивной*. Подобная форма рекурсии называется *косвенной*.
Пусть $P=\{P_1, P_2, P_3, ..., P_n\}$ - набор процедур в программе. Построим ориентированный граф, состоящий из вершин, соответствующих элементам P. Ребро из P_i в P_j проведем в графе в том случае, когда процедура P_i вызывает процедуру P_j.

19.4.27. Кратчайшие пути между всеми парами вершин. Контуры в ориентированных графах

Алгоритм отыскания множества вершин, принадлежащих контуру заданной длины
Алгоритм использует матрицу смежности A(G) и матрицу A^k, если длина контура равна k. Выберем некоторое i, такое, что $a_{ii}^{(k)}=1$. Это означает, что вершина v_i принадлежит контуру длины k.
Тогда вершина v_j принадлежит тому же контуру, если выполняются следующие три условия:

- $a_{jj}^{(k)}=1$;
- для любого n $a_{ij}^{(n)}=1$, т.е. существует путь длины n из v_i в v_j;
- $a_{ji}^{(k-n)}=1$, т.е. существует путь длины k-n из v_j в v_i.

19.4.28. Связность. Вычисление компонент связности

Определение.

Рассмотрим некоторое семейство подграфов графа G.
Граф H_{max} из этого семейства называется *максимальным*, если он не содержится ни в каком другом графе из рассматриваемого семейства.
Максимальный связный подграф графа G называется *связной компонентой* G *(компонентой связности* G). В связном графе имеется единственная связная компонента, совпадающая с самим графом.

Для вычисления компонент связности графа на языке LISP воспользуемся уже написанной ранее функцией.
Рассмотрим еще раз работу функции DEFI. Заметим, что когда к списку просмотренных вершин VISITEDдобавляется новая вершина, список PATH представляет собой путь из этой вершины в начальную. Поэтому по окончанию работы в списке VISITED содержатся только те вершины, которые можно соединить с начальной. Можно показать [2], что в этом списке содержатся все вершины, обладающие этим свойством.
Таким образом, можно сказать, что функция DEPTHFIRST вычисляет список вершин связной компоненты вершины ROOT, и теперь мы готовы решить задачу о построении всех компонент связности данного графа.

19.4.29. Связность. Нахождение компонент двусвязности

Определение.

Вершину a неориентированного графа G=(V,E) будем называть *точкой сочленения*, если удаление этой вершины и всех инцидентных ей ребер ведет к увеличению числа компонент связности графа.
Неориентированный граф называется *двусвязным*, если он связный и не содержит точек сочленения.
Произвольный максимальный двусвязный подграф графа G называется *компонентой двусвязности* или*блоком* этого графа.

19.4.30. Остовы. Построение остова

Определение.

Для произвольного связного неориентированного графа G=(V,E) каждое дерево (V,T), где T - подмножество E, будем называть *стягивающим деревом (остовом, остовным деревом)* графа G. Ребра такого дерева будем называть *ветвями*, а все остальные ребра графа будем называть *хордами*.

19.4.31. Остовы. Построение остова наименьшей стоимости

Пусть G=(V,E) - связный неориентированный граф, для которого задана *функция стоимости*, отображающая ребра в вещественные (целые) числа. Напомним, что *остовным деревом* для данного графа называется неориентированное дерево, содержащее все вершины графа. *Стоимость остовного дерева* определяется как сумма стоимостей его ребер.

Теорема.

Пусть G - связный граф с n вершинами, и пусть каждому его ребру приписано неотрицательное действительное число m(e), называемое его *мерой*.
Тогда следующая процедура приводит к решению задачи о минимальном остовном графе:

- выберем ребро e_1, обладающее в G наименьшей мерой;
- определим по индукции последовательность ребер $e_2, e_3, ..., e_{n-1}$, выбирая на каждом шаге ребро (отличное от предыдущих) с

наименьшей мерой, обладающее тем свойством, что оно не образует циклов с предыдущими ребрами e_i. Полученный подграф T графа G, ребрами которого являются $e_2, e_3, ..., e_{n-1}$ и есть требуемое остовное дерево.

19.4.32. Остовы. Построение фундаментального множества циклов

Определение.

Множество $J=\{C_e$: e *принадлежит* E\T$\}$ будем называть *фундаментальным множеством циклов* графа G(относительно стягивающего дерева (V,T)).

Теорема.

Пусть G=(V,E) - связный неориентированный граф, а (V,T) - его стягивающее дерево. Произвольный цикл графаG можно однозначно представить как симметрическую разность некоторого числа фундаментальных циклов. В общем случае произвольный псевдоцикл C графа G можно однозначно представить в виде симметрической разности вида:

$$C = \sum_{e \in C \setminus T} C_e$$

19.4.33. Алгоритмы с возвратом (общие сведения)

Проблема существования гамильтонова пути принадлежит к классу так называемых 2NP-*полных задач* [2, с.109]. Это широкий класс задач, включающий фундаментальные задачи из теории графов, логики, теории чисел, дискретной оптимизации и других дисциплин, ни для одной из которых неизвестен полиномиальный алгоритм (т.е. с числом шагов, ограниченным полиномом от размерности задачи), причем существование полиномиального алгоритма для хотя бы одной из них автоматически влекло бы за собой существование полиномиальных алгоритмов для всех этих задач. Именно факт фундаментальности многих NP-полных задач в различных областях и то, что, несмотря на независимые друг от друга усилия специалистов в этих областях, не удалось найти полиномиального алгоритма ни для одной из этих задач, склоняет к предположению, что такого алгоритма не существует.

19.4.34. Построение алгоритмов с возвратом

Основная идея метода состоит в том, что мы строим решение последовательно, начиная с пустой последовательности e (длины 0). Вообще, имея данное частичное решение $(x_1, x_2, ..., x_i)$, мы стараемся найти такое допустимое значение x_{i+1}, относительно которого мы не можем сразу заключить, что $(x_1, x_2, ..., x_{i+1})$ можно расширить до некоторого решения (либо $(x_1, x_2, ..., x_{i+1})$ уже является решением). Если такое предполагаемое, но еще не использованное решение x_{i+1} существует, то мы добавляем его к нашему частичному решению и продолжаем процесс для последовательности $(x_1, x_2, ..., x_{i+1})$. Если его не существует, то мы возвращаемся к нашему частичному решению $(x_1, x_2, ..., x_{i-1})$ и продолжаем наш процесс, отыскивая новое, еще не использованное допустимое значение x_i' - отсюда название *"алгоритм с возвратом"* (англ. Backtracking) [1, с.110].

19.4.35. Гамильтоновы циклы

Определение.

Простой цикл в графе - это замкнутый путь, все вершины которого, кроме v_0 и v_n, попарно различны.

Гамильтонов цикл - это простой цикл, содержащий все вершины графа. Заметим, что гамильтонов цикл есть далеко не в каждом графе. Граф называется *гамильтоновым*, если в нем имеется гамильтонов цикл.

Граф, который содержит простой путь, проходящий через каждую его вершину, называется *полугамильтоновым*. Ясно, что всякий гамильтонов граф является полугамильтоновым.

19.4.36. Клики

Клика графа - это любой максимальный полный подграф.

19.4.37. Независимые множества вершин графа

Определение 1.

Рассмотрим неориентированный граф G=(V,E).

Независимое множество вершин (известное также как *внутренне устойчивое множество*) есть множество вершин графа G, такое, что любые две вершины в нем не смежны (никакая пара вершин не соединена ребром).

19.4.38. Раскраски

Определение [1, с.73].

Граф G называют r-*хроматическим*, если его вершины могут быть раскрашены с использованием r цветов (красок) так, что не найдется двух смежных вершин одного цвета.
Наименьшее число r, такое, что граф G является r-хроматическим, называется *хроматическим числом* графа G и обозначается g(G).

Теорема [1, с.80].

Если граф является r-хроматическим, то он может быть раскрашен с использованием r (или меньшего числа) красок с помощью следующей процедуры: сначала в один цвет окрашивается некоторое максимальное независимое множество $S_1[G]$, затем окрашивается в следующий цвет множество $S_1[X-S_1[G]]$ и т.д. до тех пор, пока не будут раскрашены все вершины.

19.4.39. Алгоритмы раскраски графа

Алгоритм последовательной раскраски

1. Произвольной вершине v_1 графа G припишем цвет 1.
2. Если вершины $v_1, v_2, ...,v_i$ раскрашены l цветами 1, 2, ..., l, l<=i, то новой произвольно взятой вершине v_{i+1}припишем минимальный цвет, не использованный при раскраске смежных вершин.

Алгоритм прямого неявного перебора

1) окрасить v_1 в цвет 1;
2) каждую из оставшихся вершин окрашивать последовательно в соответствии с заданным упорядочением: вершина v_i окрашивается в цвет с наименьшим возможным "номером" (т.е. выбираемый цвет должен быть первым в данном упорядочении цветом, не использованным при окраске какой-либо вершины, смежной с v_i).

19.4.40. Изоморфизм

Определение 1.

Пусть G и H - графы, f - взаимно однозначное отображение множества V(G) на множество V(H) и g - взаимно однозначное отображение E(G) на E(H). Обозначим Q упорядоченную пару (f,g). Будем говорить, что Q есть*изоморфное отображение (изоморфизм)* графа G на граф H, если выполняется следующее условие: вершина хинцидентна ребру A в графе G тогда и только тогда, когда вершина fx инцидентна ребру gA в графе H.
Если такой изоморфизм Q существует, то будем говорить, что графы G и H *изоморфны*. Ясно, что в этом случае

$|V(G)| = |V(H)|$ и $|E(G)| = |E(H)|$.

Алгоритм, распознавания изоморфизма графов

1) Он находит, как правило, хорошие, хотя и не обязательно оптимальные решения.
2) Его легче и быстрее реализовать, чем любой из известных точных алгоритмов (т.е. алгоритмов, гарантирующих оптимальное решение).

Лекция 19.5

19 мая 2020 г. 13:34

19.5 Сеть

19.5.1. Сети (общие сведения)
19.5.2. Кратчайшие пути в сети. Алгоритм Форда-Беллмана
19.5.3. Кратчайшие пути в сети. Алгоритм Дейкстры
19.5.4. Пути в бесконтурной сети
19.5.5. Потоки в сетях. Система дорог
19.5.6. Описание алгоритма Дейкстры
19.5.7. Алгоритм Флойда
19.5.8. Задача о максимальном потоке
19.5.9. Алгоритм нахождения максимального потока
19.5.10. Нахождение потока наименьшей стоимости (общие замечания)
19.5.11. Нахождение потока наименьшей стоимости. Сетевая модель
19.5.12. Нахождение потока наименьшей стоимости. Сетевая модель как задача линейного программирования
19.5.13. Нахождение потока наименьшей стоимости. Симплексный алгоритм для сетей с ограниченной пропускной способностью
19.5.14. Методы сетевого планирования (общие сведения)
19.5.15. Методы сетевого планирования. Построение сети проекта
19.5.16. Методы сетевого планирования. Метод критического пути
19.5.17. Методы сетевого планирования. Построение временного графика
19.5.18. Приложение "Сетевые модели"

19.5.1. Сети (общие сведения)

Наиболее общий вид *многосвязной структуры* - многосвязная структура, которая характеризуется следующими свойствами.

1. Каждый элемент структуры содержит произвольное число направленных связей с другими элементами (или ссылок на другие элементы).
2. С каждым элементом может связываться произвольное число других элементов (т.е. каждый элемент может быть объектом ссылки произвольного числа других элементов).
3. Каждая связь в структуре имеет не только направление, но и вес.

Такую многосвязную структуру называют *сетевой* структурой или *сетью*

19.5.2. Кратчайшие пути в сети. Алгоритм Форда-Беллмана

Вычислить кратчайшее расстояние между вершинами s и t в сети:

(s,a,4), (s,d,6), (a,d,-3), (a,b,8), (b,t,2), (a,e,5),

(d,e,2), (e,c,3), (d,c,11), (c,b,-4), (c,t,7), (b,d,-6).

Перенумеруем вершины:

(s,1), (a,2), (d,3), (b,4), (c,5), (e,6), (t,7),

выпишем матрицу весов дуг

0 4 6 0 0 0 0

0 0 -3 8 0 5 0

0 0 0 0 11 2 0

0 0 -6 0 0 0 2

0 0 0 -4 0 0 7

0 0 0 0 3 0 0

0 0 0 0 0 0 0

и применим *алгоритм Форда-Беллмана* к этой матрице. В результате получим:

Введите источник: 1

Матрица расстояний 0: 0 4 -4 2 1 -2 4

Введите конечную вершину пути: 7

Кратчайший путь: ...

Компьютер "зависает", ибо *в графе имеется контур отрицательной длины* 3-6-5-4 (его длина -6+2+3-4=-5), что приводит к

появлению бесконечного множества путей, каждый из которых "короче" предыдущего, например:

путь 1-2-3-6-5-4-7 с длиной 4,

путь 1-2-3-6-5-4-3-6-5-4-7 с длиной -1,

путь 1-2-3-6-5-4-3-6-5-4-3-6-5-4-7 с длиной -6 и т.д.

19.5.3. Кратчайшие пути в сети. Алгоритм Дейкстры

```
// Нахождение кратчайшего пути из S в T с использованием
// построенной матрицы расстояний.
// -----------------------------------------------------
cout << "Введите конечную вершину пути: ";
cin >> T; T--;
W_S (&Stack,T); v = T;
while ( v!=S )
{
for (i=0;i<MaxNodes;i++)
if ( D[v]==D[i]+A[i][v] ) u = i;
W_S (&Stack,u);
v = u;
}
//Вывод кратчайшего пути на экран дисплея.
cout << "Кратчайший путь: ";
UkZv = Stack;
while ( UkZv != NULL )
{ cout << (UkZv->Element+1) << " ";
UkZv = UkZv->Sled; }
cout << endl;
}
```

19.5.4. Пути в бесконтурной сети

Займемся теперь вторым случаем, для которого известен алгоритм нахождения расстояний от фиксированной вершины за время $O(n^2)$, а именно случаем, когда граф является бесконтурным (*веса дуг могут быть произвольными*).

При описании алгоритма нахождения путей в бесконтурном графе мы можем предположить, что каждая дуга идет из вершины с меньшим номером в вершину с большим номером.

19.5.5. Потоки в сетях. Система дорог

Система дорог - это размеченный мультиграф (без петель), который отличается от графа тем, что в нем одна и та же пара (различных) вершин может быть связана более чем одним ребром [3, с.97-98]. При этом вершины соответствуют городам, а ребра - дорогам. Односторонним дорогам соответствуют дуги, а двусторонним дорогам - ребра.

Каждая дорога имеет некоторую *длину* - положительное вещественное число. Понятие пути, достижимости и замкнутого пути определяются для системы дорог аналогично подобным понятиям для графа и ориентированного графа.

Длина пути в системе дорог - это сумма длин дорог этого пути.

Расстояние между двумя городами - это длина минимального пути между этими городами.

19.5.6. Описание алгоритма Дейкстры

Шаг 0. Исходному узлу (узел 1) присваивается метка [0, -]. Полагаем i = 1.

Шаг i. а) Вычисляются временные метки $[u_i + d_{ij}, i]$ для всех узлов j, которые можно достичь непосредственно из узла i и которые не имеют постоянных меток. Если узел j уже имеет метку $[u_j, k]$, полученную от другого узла k, и если $u_i + d_{ij} < u_j$, тогда метка $[u_j, k]$ заменяется на $[u_i + d_{ij}, i]$.

19.5.7. Алгоритм Флойда

Обозначим длину кратчайшего пути между вершинами u и v, содержащего, помимо u и v, только вершины из множества $\{1\,..\,i\}$ как $d_{uv}^{(i)}$, $d_{uv}^{(0)} = \omega_{uv}$.

На каждом шаге алгоритма, мы будем брать очередную вершину (пусть её номер — i) и для всех пар вершин u и v вычислять $d_{uv}^{(i)} = \min(d_{uv}^{(i-1)}, d_{ui}^{(i-1)} + d_{iv}^{(i-1)})$. То есть, если кратчайший путь из u в v, содержащий только вершины из множества $\{1\,..\,i\}$, проходит через вершину i, то кратчайшим путем из u в v является кратчайший путь из u в i, объединенный с кратчайшим путем из i в v. В противном случае, когда этот путь не содержит вершины i, кратчайший путь из u в v, содержащий только вершины из множества $\{1\,..\,i\}$ является кратчайшим путем из u в v, содержащим только вершины из множества $\{1\,..\,i-1\}$.

19.5.8. Задача о максимальном потоке

Рассмотрим сеть трубопроводов для транспортировки сырой нефти от буровых скважин до нефтеперегонных заводов. Для перекачки нефти предусмотрены магистральные насосные станции. Каждый сегмент трубопровода имеет свою пропускную способность. Сегменты трубопровода могут быть как однонаправленные (осуществляют перекачку нефти только в одном направлении), так и в двунаправленные. В однонаправленных сегментах положительная пропускная способность предполагается в одном направлении и нулевая - в другом. На рис. 1 показана типовая сеть нефтепроводов. Как определить оптимальную пропускную способность (т.е. максимальный поток) между нефтяными скважинами и нефтеперегонными заводами?

Шаг 1. Для всех ребер (i, j) положим остаточную пропускную способность равной первоначальной пропускной способности, т.е. приравняем (c_{ij}, c_{ji}) = (C_{ij}, C_{ji}). Назначим a_1 = *бесконечности* и пометим узел 1 меткой [*бесконечность*, -]. Полагаем i = 1 и переходим ко второму шагу.

Шаг 2. Определяем множество S_i как множество узлов j, в которые можно перейти из узла i по ребру с положительной остаточной пропускной способностью (т.е. $c_{ij} > 0$ для всех j, принадлежащих S_i). Если S_i не пустое множество, выполняем третий шаг, в противном случае переходим к шагу 4.

Шаг 3. В множестве S_i находим узел k, такой, что $c_{ik} = \max\{c_{ij}\}$ для всех j, принадлежащих S_i. Положим $a_k = c_{ik}$ и пометим узел k меткой [a_k, i]. Если последней меткой помечен узел стока (т.е. если k = n), сквозной путь найден, и мы переходим к пятому шагу.

Шаг 4 (Откат назад). Если i = 1, сквозной путь невозможен, и мы переходим к шагу 6. Если i не равно 1, находим помеченный узел r, непосредственно предшествующий узлу i, и удаляем узел i из множества узлов, смежных с узлом r. Полагаем i = r и возвращаемся ко второму шагу.

Шаг 5 (Определение остаточной сети). Обозначим через $N_p = \{1, k_1, k_2, \ldots, n\}$ множество узлов, через которые проходит p-й найденный сквозной путь от узла источника (узел 1) до узла стока (узел n). Тогда максимальный поток, проходящий по этому пути, вычисляется как $f_p = \min\{a_1, a_{k1}, a_{k2}, \ldots, a_n\}$.

19.5.9. Алгоритм нахождения максимального потока

Шаг 1. Для всех ребер (i, j) положим остаточную пропускную способность равной первоначальной пропускной способности, т.е. приравняем (c_{ij}, c_{ji}) = (C_{ij}, C_{ji}). Назначим a_1 = *бесконечности* и пометим узел 1 меткой [*бесконечность*, -]. Полагаем i = 1 и переходим ко второму шагу.

Шаг 2. Определяем множество S_i как множество узлов j, в которые можно перейти из узла i по ребру с положительной остаточной пропускной способностью (т.е. $c_{ij} > 0$ для всех j, принадлежащих S_i). Если S_i не пустое множество, выполняем третий шаг, в противном случае переходим к шагу 4.

Шаг 3. В множестве S_i находим узел k, такой, что $c_{ik} = \max\{c_{ij}\}$ для всех j, принадлежащих S_i. Положим $a_k = c_{ik}$ и пометим узел k меткой [a_k, i]. Если последней меткой помечен узел стока (т.е. если k = n), сквозной путь найден, и мы переходим к пятому шагу.

Шаг 4 (Откат назад). Если i = 1, сквозной путь невозможен, и мы переходим к шагу 6. Если i не равно 1, находим помеченный узел r, непосредственно предшествующий узлу i, и удаляем узел i из множества узлов, смежных с узлом r. Полагаем i = r и возвращаемся ко второму шагу.

Шаг 5 (Определение остаточной сети). Обозначим через $N_p = \{1, k_1, k_2, \ldots, n\}$ множество узлов, через которые проходит p-й найденный сквозной путь от узла источника (узел 1) до узла стока (узел n). Тогда максимальный поток, проходящий по этому пути, вычисляется как $f_p = \min\{a_1, a_{k1}, a_{k2}, \ldots, a_n\}$.

19.5.10. Нахождение потока наименьшей стоимости (общие замечания)

Задача нахождения потока наименьшей стоимости в сети с ограниченной пропускной способностью обобщает задачу *определения максимального потока по следующим направлениям.*

1. Все ребра допускают только одностороннее направление потока, т.е. являются (ориентированными) дугами.
2. Каждой дуге поставлена в соответствие (неотрицательная) стоимость прохождения единицы потока по данной дуге.
3. Дуги могут иметь положительную нижнюю границу пропускной способности.

4. Любой узел сети может выступать как в качестве источника, так и стока.

19.5.11. Нахождение потока наименьшей стоимости. Сетевая модель

Рассмотрим сеть G = (N, A) с ограниченной пропускной способностью, где N - множество узлов, A - множество дуг. Обозначим:

- x_{ij} - величина потока, протекающего от узла i к узлу j,
- u_{ij} (l_{ij}) - верхняя (нижняя) граница пропускной способности дуги (i, j),
- c_{ij} - стоимость прохождения единицы потока по дуге (i, j),
- f_i - величина "чистого" результирующего потока, протекающего через узел i.

19.5.12. Нахождение потока наименьшей стоимости. Сетевая модель как задача линейного программирования

Нижнюю границу пропускной способности l_{ij} можно удалить из ограничений с помощью подстановки $x_{ij} = x'_{ij} - l_{ij}$. Для нового потока верхней границей пропускной способности будет величина $u_{ij} - l_{ij}$. В этом случае результирующий поток через узел i будет равен $[f_i] - l_{ij}$, а через узел j - $[f_j] + l_{ij}$. На рисунке 1 показаны преобразования характеристик дуги (i, j) после исключения ее нижней границы пропускной способности.

19.5.13. Нахождение потока наименьшей стоимости. Симплексный алгоритм для сетей с ограниченной пропускной способностью

Шаг 0. Определяем для данной сети начальное базисное допустимое решение (множество дуг). Переходим к шагу 1.

Шаг 1. С помощью условия оптимальности симплекс-метода определяем вводимую в базис переменную (дугу). Если на основе условия оптимальности определяем, что последнее решение оптимально, вычисления заканчиваются. В противном случае переходим к шагу 2.

Шаг 2. На основе условия допустимости симплекс-метода определяем исключаемую из базиса переменную (дугу). Изменив базис, возвращаемся к шагу 1.

19.5.14. Методы сетевого планирования (общие сведения)

На основе сетевых моделей разработано множество методов планирования, составление временных расписаний и управления проектами. Наиболее известные - *метод критического пути* (Critical Path Method, сокращенно CPM), а также система планирования и руководства программами разработок (Program Evaluation and Review Technique, сокращенно PERT). В этих методах проекты рассматриваются как совокупность некоторых взаимосвязанных процессов (видов деятельности, этапов или фаз выполнения проекта), каждый из которых требует определенных временных и других ресурсов. В методах CPM и PERT проводится анализ проектов для составления временных графиков распределения фаз проектов. На рисунке 1 в обобщенной форме показаны основные этапы выполнения этих методов. На первом этапе определяются отдельные процессы, составляющие проект, их отношения предшествования (т.е. какой процесс должен предшествовать другому) и их длительность. Далее проект представляется в виде сети, показывающей отношения предшествования среди процессов, составляющих проект. На третьем этапе на основе построенной сети выполняются вычисления, в результате которых составляется временной график реализации проекта.

19.5.15. Методы сетевого планирования. Построение сети проекта

Правило 1. Каждый процесс в проекте представим одной и только одной дугой.

Правило 2. Каждый процесс идентифицируется двумя концевыми узлами.

Правило 3. Для поддержания правильных отношений предшествования при включении в сеть любого процесса необходимо ответить на следующие вопросы.

19.5.16. Методы сетевого планирования. Метод критического пути

Проход вперед. Здесь вычисления начинаются в узле 1 и заканчиваются в последнем узле n.

Начальный шаг. Полагаем A_1=0; это указывает на то, что проект начинается в нулевой момент времени. *Основной шаг* j. Для узла j определяем узлы p, q, ..., v, непосредственно связанные с узлом j процессами (p, j), (q, j), ..., (v, j), для которых уже вычислены самые ранние времена наступления соответствующих событий. Самое раннее время наступления события j вычисляется по формуле

$A_j = \max \{ A_p + D_{pj}, A_q + D_{qj}, ..., A_v + D_{vj}\}$.

Проход вперед завершается, когда будет вычислена величина A_n для узла n. По определению величина j равна самому

длинному пути (длительности) от начала проекта до узла (события) j.

Проход назад. В этом проходе вычисления начинаются в последнем узле n и заканчивается в узле 1.

Начальный шаг. Полагаем $B_n = A_n$; это указывает, что самое раннее и самое позднее времена для завершения проекта совпадают.

Основной шаг j. Для узла j определяем узлы p, q, …, v, непосредственно связанные с узлом j процессами (j, p), (j, q), …, (j, v), для которых уже вычислены самые поздние времена наступления соответствующих событий. Самое позднее время наступления события j вычисляется по формуле

$B_j = \min \{B_p - D_{jp}, B_q - D_{jq}, \ldots, B_v - D_{jv}\}$.

19.5.17. Методы сетевого планирования. Построение временного графика

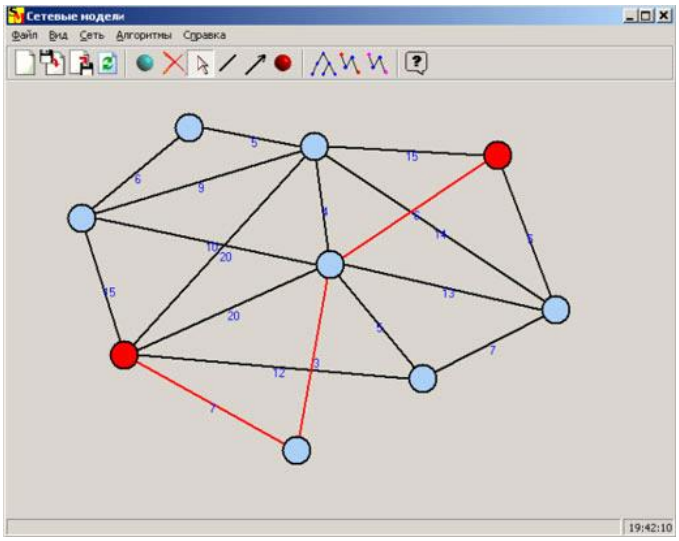
Для некритического процесса (i, j)

- *если* $FF_{ij} = TF_{ij}$, *тогда данный процесс может выполняться в любое время внутри максимального интервала* (A_i, B_j) *без нарушения отношений следования*;
- *если* $FF_{ij} < TF_{ij}$, *тогда без нарушения отношений следования данный процесс может начаться со сдвигом, не превышающим* FF_{ij}, *относительно самого раннего момента начала процесса* i. *Сдвиг начала процесса на величину времени, превышающую* FF_{ij} *(но не более* TF_{ij}*), должен сопровождаться равным сдвигом относительно* j *всех процессов, начинающихся с события* j.

19.5.18. Приложение "Сетевые модели"

Работу программы можно разделить на две части: *построение сети* и *применение указанных алгоритмов к этой сети*.

Для всех кнопок панели инструментов и команд главного меню в строке состояния отображается подсказка. При наведении мыши на кнопку панели инструментов появляется всплывающая подсказка. К некоторым командам есть горячие клавиши, которые отображаются справа от команды в главном меню или во всплывающей подсказке.



Лекция 19.6

23 мая 2020 г. 14:02

19.6 Структуры данных: двоичная куча (binary heap)

19.6.1 Введение
19.6.2 Реализация двоичной кучи (binary heap)
19.6.3 Примеры бинарной кучи

19.6.1 Введение

Бинарная куча (пирамида, сортирующее дерево, binary heap) – это двоичное дерево, удовлетворяющее следующим условиям:

- Приоритет любой вершины не меньше (≥), приоритета ее потомков
- Дерево является полным двоичным деревом (complete binary tree) – все уровни заполнены слева направо (возможно за исключением последнего)

Двоичная куча представляет 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. В таком случае структура называется max-heap, поскольку корень поддерева является максимумом из значений элементов поддерева. Здесь для простоты используется именно такое представление. Напомним также, что 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).

19.6.2 Реализация двоичной кучи (binary heap)

Двоичная куча представляет 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. В таком случае структура называется max-куча, поскольку корень поддерева является максимумом из значений элементов поддерева.

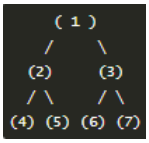
В качестве альтернативы, если сравнение перевернуть, то наименьший элемент будет всегда корневым узлом, такие кучи называют min-кучами.

Двоичную кучу удобно хранить в виде одномерного массива.

```
//Реализация класса кучи
class Heap {
static const int SIZE = 100; // максимальный размер кучи
int *h; // указатель на массив кучи
int HeapSize; // размер кучи
public:
Heap(); // конструктор кучи
void addelem(int); // добавление элемента кучи
void outHeap(); // вывод элементов кучи в форме кучи
void out(); // вывод элементов кучи в форме массива
int getmax(); // удаление вершины (максимального элемента)
void heapify(int); // упорядочение кучи
};
```

Метод heapify восстанавливает основное свойство кучи для дерева с корнем в i-ой вершине при условии, что оба поддерева ему удовлетворяют. Для этого необходимо «опускать» i-ую вершину (менять местами с наибольшим из потомков), пока основное свойство не будет восстановлено (процесс завершится, когда не найдется потомка, большего своего родителя)

19.6.3 Примеры бинарной кучи



Нам дано дерево с корнем в вершине с номером 1. У вершины 1 два ребёнка: 2 и 3. У вершины 2 и у вершины 3 тоже по два

ребёнка (4;5 и 6;7 соответственно). Каждый элемент кучи состоит из двух составляющих: номер элемента и его значение. Пускай для простоты в нашей куче номера и будут являться значениями. Итак, основные правила кучи:

- У каждой вершины должно быть ровно по 2 ребёнка
- Каждый ребёнок “хуже” своего отца.

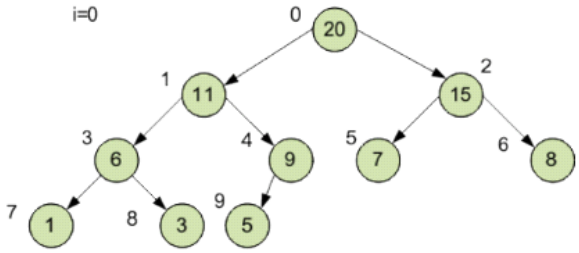
Двоичная куча представляет собой полное бинарное дерево, для которого выполняется основное свойство кучи: приоритет каждой вершины больше приоритетов её потомков.

В простейшем случае приоритет каждой вершины можно считать равным её значению. В таком случае структура называется max - куча, поскольку корень поддерева является максимумом из значений элементов поддерева.

В качестве альтернативы, если сравнение перевернуть, то наименьший элемент будет всегда корневым узлом, такие кучи называют min-кучами.

Двоичную кучу удобно хранить в виде одномерного массива, причем

- левый потомок вершины с индексом i имеет индекс 2*i+1,
- правый потомок вершины с индексом i имеет индекс 2*i+2.



Лекция 20

25 мая 2020 г. 14:09

20 Рекурсия и рекурсивные алгоритмы

20.1. Рекуррентные соотношения. Рекурсия и итерация
20.2. Сущность рекурсии
20.3. Виды рекурсии
20.4. Имитация работы цикла с помощью рекурсии
20.5. Произвольное количество вложенных циклов
20.6. Графы
20.7. Фракталы
20.8. Деревья
20.9. Примеры рекурсивных алгоритмов
20.10. Рекурсия или цикл. Избавление от рекурсии

20.1. Рекуррентные соотношения. Рекурсия и итерация

Рекурсией называется ситуация, когда алгоритм вызывает сама себя.

Следует понимать, что вызов функций влечет за собой некоторые дополнительные накладные расходы, поэтому первый вариант вычисления факториала будет несколько более быстрым. Вообще итерационные решения работают быстрее рекурсивных.

Прежде чем переходить к ситуациям, когда рекурсия полезна, обратим внимание еще на один пример, где ее использовать не следует.

Рассмотрим частный случай рекуррентных соотношений, когда следующее значение в последовательности зависит не от одного, а сразу от нескольких предыдущих значений. Примером может служить известная последовательность Фибоначчи, в которой каждый следующий элемент есть сумма двух предыдущих:

$$x_n = x_{n-1} + x_{n-2},\ x_0 = 1,\ x_1 = 1 \qquad (5)$$

Любые рекурсивные процедуры и функции, содержащие всего один рекурсивный вызов самих себя, легко заменяются итерационными циклами. Чтобы получить что-то, не имеющее простого нерекурсивного аналога, следует обратиться к процедурам и функциям, вызывающим себя два и более раз. В этом случае множество вызываемых процедур образует уже не цепочку, а целое дерево. Существуют широкие классы задач, когда вычислительный процесс должен быть организован именно таким образом. Как раз для них рекурсия будет наиболее простым и естественным способом решения.

20.2. Сущность рекурсии

Процедура или функция может содержать вызов других процедур или функций. В том числе процедура может вызвать саму себя. Никакого парадокса здесь нет – компьютер лишь последовательно выполняет встретившиеся ему в программе команды и, если встречается вызов процедуры, просто начинает выполнять эту процедуру. Без разницы, какая процедура дала команду это делать.

Предыдущий вызов еще не завершился, поэтому можете представить себе, что создается еще одна процедура и до окончания ее работы первая свою работу не заканчивает. Процесс вызова заканчивается, когда параметр a = 0. В этот момент одновременно выполняются 4 экземпляра процедуры. Количество одновременно выполняемых процедур называют *глубиной рекурсии*.

20.3. Виды рекурсии

Рекурсия - метод определения класса объектов или методов предварительным заданием одного или нескольких (обычно простых) его базовых случаев или методов, а затем заданием на их основе правила построения определяемого класса, ссылающегося прямо

или косвенно на эти базовые случаи.

Другими словами, рекурсия - способ общего определения объекта или действия через себя, с использованием ранее заданных частных определений. Рекурсия используется, когда можно выделить само подобие задачи.

Рекурсивный алгоритм (процедура, функция):

- алгоритм называется рекурсивным, если в его определении содержится прямой или косвенный вызов этого же алгоритма;
- рекурсивная функция - одно из математических уточнений интуитивного понятия вычислимой функции.

Базис рекурсии - это предложение, определяющее некую начальную ситуацию или ситуацию в момент прекращения. Как правило, в этом предложении записывается некий простейший случай, при котором ответ получается сразу, даже без использования рекурсии.

Шаг рекурсии - это правило, в теле которого обязательно содержится, в качестве подцели, вызов определяемого предиката.

Подпрограмма - все, что находится внутри рекурсивной функции.

Глубина рекурсии - количество одновременно выполняемых процедур.

Прямая рекурсия - непосредственный вызов алгоритма (функции, процедуры, метода) из текста самого метода.

При косвенной рекурсии имеется циклическая последовательность вызовов нескольких алгоритмов.

В данном случае функция r1( ) вызывает функцию r2( ), которая вызывает r3( ).

Функция r3( ) в свою очередь снова вызывает r1( ).

Линейная рекурсия - если исполнение подпрограммы приводит только к одному вызову этой же самой подпрограммы, то такая рекурсия называется линейной.

Ветвящаяся рекурсия - если каждый экземпляр подпрограммы может вызвать себя несколько раз, то рекурсия называется нелинейной или "ветвящейся".

Бесконечная рекурсия (на самом деле это условное обозначение так как при переполнении памяти компьютера программа выдаст ошибку и/или завершит ее в аварийном режиме).

Одна из самых больших опасностей рекурсии - бесконечный вызов функцией самой себя.

Возможна чуть более сложная схема: функция A вызывает функцию B, а та в свою очередь вызывает A. Это называется *сложной рекурсией*. При этом оказывается, что описываемая первой процедура должна вызывать еще не описанную. Чтобы это было возможно, требуется использовать опережающее описание.

20.4. Имитация работы цикла с помощью рекурсии

Если процедура вызывает сама себя, то, по сути, это приводит к повторному выполнению содержащихся в ней инструкций, что аналогично работе цикла. Некоторые языки программирования не содержат циклических конструкций вовсе, предоставляя программистам организовывать повторения с помощью рекурсии (например, Пролог, где рекурсия - основной прием программирования).

Если представить себе цепочку из рекурсивно вызванных процедур, то в примере 1 мы проходим ее от раньше вызванных процедур к более поздним. В примере 2 наоборот от более поздних к ранним.

20.5. Произвольное количество вложенных циклов

Разместив рекурсивные вызовы внутри цикла, по сути, получим вложенные циклы, где уровень вложенности равен глубине рекурсии.

Для примера напишем процедуру, печатающую все возможные сочетания из k чисел от 1 до n. Числа, входящие в каждое сочетание, будем печатать в порядке возрастания. Сочетания из двух чисел (k=2) печатаются так:

```
#include <iostream>
using namespace std;
int main()
{
// переходим на русский язык в консоли
setlocale(LC_ALL, "Rus");
```

```
int n = 5;
for (int i1 = 1; i1 <= n; i1++)
for (int i2 = i1+1; i2 <= n; i2++)
cout << i1 << " " << i2 << "\n";
system("PAUSE");
return 0;
}
```

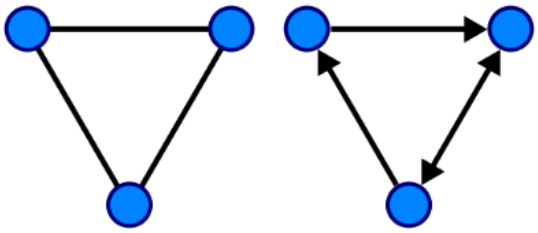
20.6. Графы

Графом называют графическое изображение, состоящее из *вершин* (*узлов*) и соединяющих некоторые пары вершин *ребер* (рис. 11а).

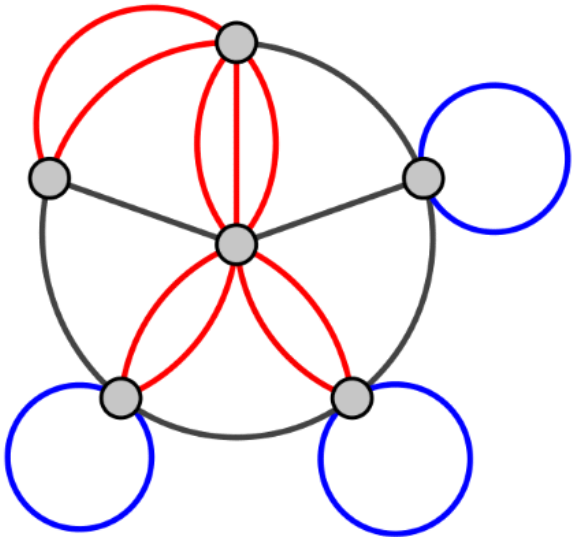
Более строго: граф – совокупность множества вершин и множества ребер. Множество ребер – подмножество евклидова квадрата множества вершин (то есть ребро соединяет ровно две вершины).

Графы – фундаментальное понятие как в математике, так и в информатике. Проще всего объяснить его с помощью аналогии с дорожной системой. Существует определённый набор городов, некоторые из которых связаны дорогами, которые могут быть как односторонними, так и двухсторонними. Вся эта структура и называется графом.

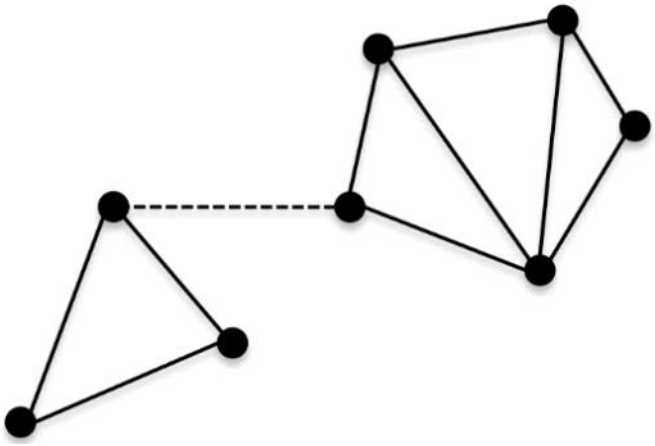
Граф, в котором все рёбра неориентированные, также называют неориентированным, а граф с ориентированными рёбрами, соответственно, ориентированным.



Существует множество разновидностей графов, и среди них встречаются довольно специфические. В частности, так называемые мультиграфы разрешают наличие между двумя вершинами нескольких рёбер (называемых кратными рёбрами), а также наличие петель. Петля – ребро, входящее в ту же вершину, из которой исходит. Выглядят они следующим образом:



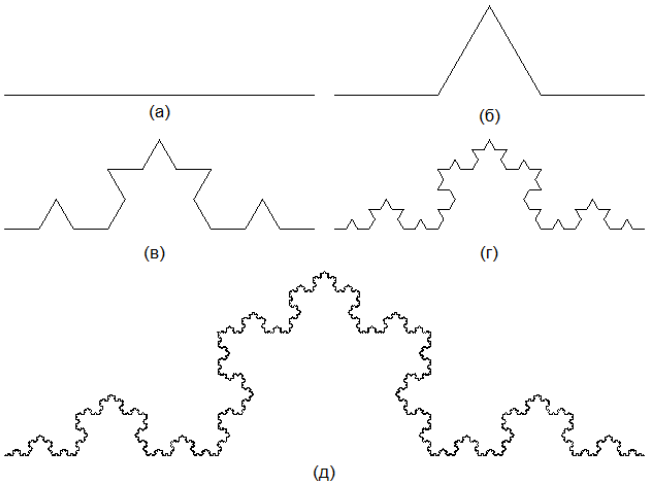
Граф называется связным, если между любой парой вершин существует хотя бы один путь. Как пример рассмотрим следующий граф:



20.7. Фракталы

Фракталами называют геометрические фигуры, обладающие свойством самоподобия, то есть состоящие из частей, подобных всей фигуре.

Классическим примером является кривая Коха, построение которой показано на рис. 12. Изначально берется отрезок прямой (рис. 12а). Он делится на три части, средняя часть изымается и вместо нее строится угол (рис. 12б), стороны которого равны длине изъятого отрезка (то есть 1/3 от длины исходного отрезка). Такая операция повторяется с каждым из получившихся 4-х отрезков (рис. 12в). И так далее (рис. 12г). Кривая Коха получается после бесконечного числа таких итераций. На практике построение можно прекратить, когда размер деталей окажется меньше разрешения экрана.

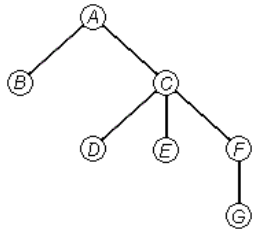


20.8. Деревья

Определение: Деревом будем называть конечное множество *T*, состоящее из одного или более узлов, таких что:

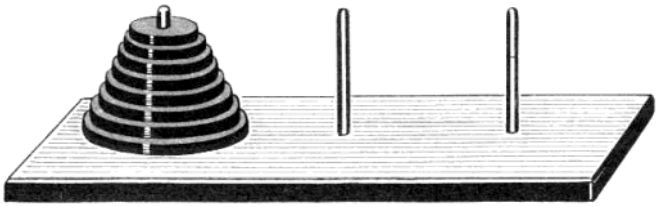
- Имеется один специальный узел, называемый корнем данного дерева.
- Остальные узлы (исключая корень) содержатся в $m \geq 0$ попарно непересекающихся подмножествах $T_1, T_2, \ldots, T_m$, каждое из которых в свою очередь является деревом. Деревья $T_1, T_2, \ldots, T_m$ называются *поддеревьями* данного дерева.

Это определение является рекурсивным. Если коротко, то дерево это множество, состоящее из корня и присоединенных к нему поддеревьев, которые тоже являются деревьями. Дерево определяется через само себя. Однако данное определение осмысленно, так как рекурсия конечна. Каждое поддерево содержит меньше узлов, чем содержащее его дерево. В конце концов, мы приходим к поддеревьям, содержащим всего один узел, а это уже понятно, что такое.



20.9. Примеры рекурсивных алгоритмов

Согласно легенде в Великом храме города Бенарас, под собором, отмечающим середину мира, находится бронзовый диск, на котором укреплены 3 алмазных стержня, высотой в один локоть и толщиной с пчелу. Давным-давно, в самом начале времен монахи этого монастыря провинились перед богом Брамой. Разгневанный, Брама воздвиг три высоких стержня и на один из них поместил 64 диска из чистого золота, причем так, что каждый меньший диск лежит на большем. Как только все 64 диска будут переложены со стержня, на который Бог Брама сложил их при создании мира, на другой стержень, башня вместе с храмом обратятся в пыль и под громовые раскаты погибнет мир.
В процессе требуется, чтобы больший диск ни разу не оказывался над меньшим. Монахи в затруднении, в какой же последовательности стоит делать перекладывания? Требуется снабдить их софтом для расчета этой последовательности.



Предположим, что существует решение для *n*-1 диска. Тогда для перекладывания *n* дисков надо действовать следующим образом:

- Перекладываем *n*-1 диск.
- Перекладываем *n*-й диск на оставшийся свободным штырь.
- Перекладываем стопку из *n*-1 диска, полученную в пункте (1) поверх *n*-го диска.

20.10. Рекурсия или цикл. Избавление от рекурсии

Головная рекурсия— рекурсивный вызов выполняется ближе к началу метода и является одним из первых обрабатываемых объектов. Поскольку он вызывает сам себя, ему приходится полагаться на результаты предыдущей операции, помещенной в стек вызовов. Из-за использования стека вызовов существует вероятность переполнения стека, если стек вызовов недостаточно велик.

Концевая рекурсия— рекурсивный вызов выполняется в конце и является последней строкой обрабатываемого кода. Этот метод не использует стек вызовов независимо от глубины рекурсии.

Цикл — это многократное исполнение нескольких операторов. Циклы не заносят данные в стек вызовов независимо от числа исполнений цикла. Важным отличием циклов от рекурсивных функций является тот факт, что циклы используют для подсчета числа исполнений итератор, а рекурсивные функции для определения момента выхода должны выполнять сравнение результатов. Другим важным отличием является возможность применения в циклах фильтров и прочих селекторов. Примером такой ситуации может служить цикл foreach.

Рекурсия является очень мощным инструментом программирования. Ее можно использовать для решения задач, которые более эффективно представляются рекурсией и могут использовать стек вызовов. Как правило, это относится к задачам сортировки, которые эффективно решаются с помощью рекурсии, обеспечивающей более высокую скорость, чем циклы.