Существуют различные классификации языков программирования.

Одна из них разделяет ЯП на:

1. ЯП общего назначения,
2. Специализированные ЯП,
3. ЯП сценария.

Классификация по парадигме программирования (парадигма – сложившийся набор методов для проектирования, программирования и сопровождения программных систем):

1. процедурное программирование,
2. структурное программирование,
3. модульное программирование,
4. объектно-ориентированное программирование,
5. компонентный подход к программированию.

Создание языка С приписывают фирме AT&T Bell Labs (Денис Ритчи). Язык был разработан в начале 1970-х годов для реализации ОС Unix и планировался для замещения Ассемблера.

Основные стандарты:

1989 – стандарт C'89 ANSI

1990 – стандарт C'90 ISO

1999 – стандарт C'99

Место языка С в различных классификациях

1. **С – язык системного программирования.**
2. **С – язык общего назначения.**
3. **С’99 поддерживает парадигму процедурного программирования (до С’99 язык медленно контролировал типы данных).**
4. **С – компилируемый язык программирования.**

Преимущества

1. **Язык стандартизирован.**
2. **Практически для любой платформы (даже для микроконтроллеров) существует компилятор С. Следствие – высокая переносимость между платформами на уровне технических средств и исходных текстов.**
3. **Компактный ЯП (нет операторов ввода/вывода – идей на уровне стандартных функций).**
4. **Содержит низкоуровневые средства (можно осуществлять доступ к произвольным адресам памяти и портам).**
5. **Наличие побитовых операций.**
6. **Изначально компиляторы С развивались как оптимизируемые.**
7. **Широкий набор стандартных библиотек.**

Недостатки

1. **Язык консервативен (отсутствие средств работы с мультимедиа, с сетью и поддержки многозадачности).**
2. **Язык содержит низкоуровневые средства, поэтому программы, написанные на С, могут представлять опасность для других программ и ОС.**
3. **Ослабленный контроль за типами данных.**
4. **Не поддерживаются новые парадигмы программирования.**

Алфавит состоит из первых 128 символов кодов таблицы ASCII. На его основе создаются строковые константы, идентификаторы, зарезервированные слова, знаки операций, знаки для обозначения комментариев. Дополнительный алфавит представлен широким набором символов (чаще всего из UTF-16). Символы из доп. алфавита могут встречаться в константах, либо в виде строк или отдельных символов. Язык Си был создан уже после внедрения стандарта ASCII, поэтому использует почти все его графические символы (нет только $, @, `).

Текст, заключённый в служебные символы /* и */ в этом порядке, полностью игнорируется компилятором. Компиляторы, совместимые со стандартом C99, также позволяют использовать комментарии, начинающиеся с символов // и заканчивающиеся переводом строки.
Примеры:

```
/* Это комментарий */
//Тоже комментарий
```

Нельзя вкладывать блочные комментарии друг в друга:

```
/* Это комментарий /* Второй комментарий*/ Третий комментарий */
```

В этом случае третий комментарий не будет считаться таковым.

Константы

Константы в С бывают: целочисленные, действительные, строковые.

Целочисленные константы – это целые числа и символы (каждый символ определяется своим числовым кодом).

Представление символов:

- **непосредственно – ‘A’,**
- **десятичным кодом – 65,**
- **восьмеричным кодом – \ddd, d – восьмеричная цифра (от 0 до 7). Например, \016 или \214;**
- **шестнадцатеричным кодом – \xhh или \Xhh, h – шестнадцатеричная цифра (от 0 до F). Например, \x0f или \x3B**

Представление целочисленных констант:

- **в десятичной системе счисления – 65,**
- **в восьмеричной системе счисления – 071,**
- **в шестнадцатеричной системе счисления – 0x41.**

Целочисленные константы могут иметь суффикс, который указывает количество памяти для хранения:

- **long int – 65L**

Основные типы: char, short int, int, long int

Размер каждого типа в байтах определяется компилятором. Узнать его заранее, не почитав описание, невозможно.

Для определения размера пользуются функцией sizeof:

sizeof(char) – обычно 1 байт,

sizeof(short int**)** – обычно 2 байта,

sizeof(int) – обычно 4 байта,

sizeof(long int) – обычно 4 байта.

Действительные константы

Основные типы: **float** (4 байта), **double** (8 байт), **long double** (≥8, для процессоров Intel – 10)

Задание констант:

- 1.28 – для базового типа **double**,
- 1.28f – для **float**,
- 1.28L – для **long double**.

Научная нотация – представление действительного числа в виде мантиссы и степени числа 10. Пример: 1.28e2 ($1{,}28 \cdot 10^2$).

Чтобы отличить действительные константы от целочисленных, для них всегда ставится разделитель целой и дробной части. **Пример:**

- 0, 1 – целочисленные константы,
- 0.0, 1.0 – действительные константы.

Строковые константы

В С строки могут иметь произвольную длину. Признаком конца строки является символ с кодом 0 – так называемый ***ноль-терминатор*.** В записи используется как **\0 .**

Пример строковой константы: “Hello” (ноль-терминатор добавляется в конец строки автоматически,а в памяти строка выглядит как "Hello\0").

Идентификаторы

Правила составления идентификаторов

Могут использоваться первые 128 символов таблицы кодов ASCII. Лучше всего использовать латинские буквы, арабские цифры и символ подчёркивания. Начинаться идентификатор должен либо с латинской буквы, либо с “_”.

Язык С чувствителен к регистру!!! Поэтому, к примеру, идентификаторы n и N будут считаться различными.

Длина идентификаторов не ограничена, но, как правило, значащими являются только первый 31 символ.

Зарезервированные слова

auto – класс памяти (автоматическая память), встречается в описании переменных и указывает на то, что переменная является локальной.

break – используется для прерывания выполнения операторов циклов и выхода из оператора-переключателя **switch**.

char – описание переменных символьного типа.

const – объявляет неизменяемую переменную или параметр.

continue – прерывает текущую итерацию цикла и инициирует начало новой итерации этого же цикла.

case – используется в операторе-переключателе.

default – используется в операторе-переключателе.

do – используется в операторе цикла с постусловием.

double – объявление переменной с плавающей точкой двойной точности.

else – используется в операторе условного перехода.

Зарезервированные слова

enum – для объявления перечислимых типов.

extern – класс памяти, указывающий на то, что переменная является глобальной и внешней, т.е. определена в другом месте исходного текста программы.

float – объявление переменной с плавающей точкой одинарной точности.

for – цикл с параметром.

goto – оператор безусловного перехода.

int – объявление переменной целочисленного типа.

if – оператор условного перехода.

long – тип для описания переменной целочисленного и вещественного типов с увеличением выделяемой памяти для соответствующего типа (напр., long int, long double).

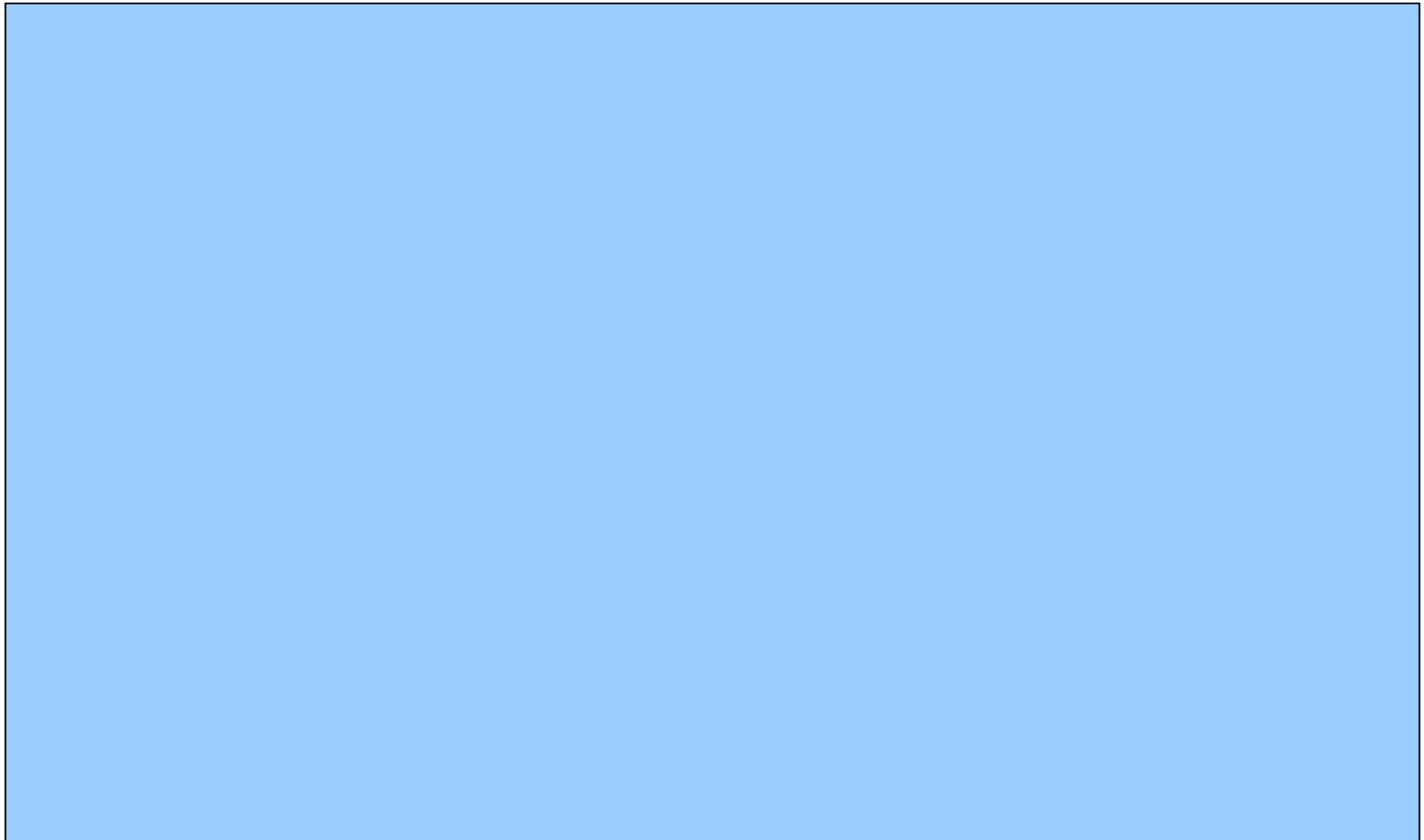
register – используется для описания локальной регистровой переменной – для размещения целочисленных переменных не в ОЗУ, а в регистре общего назначения процессора.

return – оператор передачи управления из вызванной функции в функцию, её вызвавшую.

Зарезервированные слова

short – для описания переменных типа short int,
signed – задаёт знаковый тип для всех целочисленных типов, кроме перечислимых.
sizeof – применение этой операции к имени типа, константе, выражению, переменной возвращает количество байт, которое операнд занимает в памяти.
struct – структурированный тип.
switch – оператор-переключатель.
static –класс памяти, использующийся при описании переменной или функции; влияет на область видимости, а для переменной также и на время жизни.
typedef – позволяет задавать имя для типа (синоним для существующего типа).
union – тип объединения, специальный случай структуры, у которой поля размещаются по одному и тому же адресу памяти.
unsigned – беззнаковый модификатор типа.
void – пустой тип данных.
while – для построения циклов с пред- и постусловием.
volatile – модификатор типа, запрещающий компилятору оптимизацию.

Ниже приведены операции в порядке убывания приоритета. Операции, приведённые на одной строчке, имеют одинаковый приоритет. Операции, помеченные как R->L, имеют правую ассоциативность (то есть при сочетании равноприоритетных операций без скобок они вычисляются справа налево; при этом порядок вычисления аргументов большинства операций не специфицирован и зависит от реализаций):



Общая структура программы на С

Программа на С – последовательности процессорных директив, описаний и определений глобальных объектов и функций.

// директивы препроцессора

#include <stdio.h>

#define Pi 3.14169265

```
double s,c;   // объявление глобальных переменных
double length_c(double);   // описание прототипов
double area(double);       // функций
int main()                 // главная функция
{
   double r;           // объявление локальной переменной
   printf("введите r ");    // вывод на экран
   scanf("%lf",&r);
```

```
   c = length_c(r);
   s = area(r);
   // аргумент функции printf определяет формат вывода
   // l – long, f – float, \n – перевод строки
   printf("\n c=%lf\n s=%lf\n",c,s);
   return 0;
   // если программа успешно завершилась, то ставится 0
}
// описываем пользовательские функции
double length_c(double r)
{
   return 2*Pi*r;
}
double area(double r)
{
   return Pi*r*r;
}
```

С – модульный язык. Программу на С можно (и нужно) писать в виде отдельных подпрограмм, которые называются функциями.
Функция **– независимый замкнутый фрагмент кода, созданный для выполнения определённой задачи. С помощью функций можно добиться эффективной и экономичной работы программы.**
Обязательной в каждой программе является функция main. С неё начинается выполнение программы. Она может быть записана в одной из нескольких форм:

```
int main()
void main()
void main(int argc, char* argv[], char* envp[])
```

В последнем определении argv – массив строк (количество его элементов задаёт argc), envp – массив строк переменных системного окружения.
argc всегда ≥ 1. Первый элемент – argv[0] – имя самой программы. Начиная с argv[1] идут параметры командной строки.
Кроме main, в программе может быть определено любое число функций. Из любой функции может быть вызвана любая другая функция (кроме main – её вызывать из других функций нельзя). Вложение функций, т.е. определение одной функции внутри другой, недопустимо. Хорошим стилем программирования считается написание небольших по объёму функций.
Лексема – единица текста программы, которая при компиляции воспринимается как единое целое и не может быть разделена на составные части.

Директивы препроцессора

#include **– указывает компилятору подставить на это место текст из указанного в директиве файла.**

#define **– определение макроса (замена одной последовательности символов на другую) или препроцессорного идентификатора.**

Пример:

```
#define REAL long double
#define min(x;y) (x<y)?((x):(y))
```

Поточные эффекты в макросах: пусть указана директива

```
#define sq(x) x*x
```

Тогда строка a = sq(j+2) будет преобразована в j+2*j+2. Чтобы этого избежать, используемые переменные в макросах нужно заключать в скобки.

#undef **– отмена определения макроса.**

Формат: *#undef <идентификатор>*

Директивы ветвления:

#if **– проверка условия**

Формат: *#if <целочисленное константное выражение>*

#ifdef **– проверка определённости идентификатора**

Формат: *#ifdef <идентификатор>*

#ifndef **– проверка неопределённости идентификатора**

Формат: *#ifndef <идентификатор>*

#else **– альтернативная ветвь («иначе») для директив ветвления**

#elif **– составная директива (#else if)**

#endif **– окончание условной директивы**

Структура применения директив условной компиляции.

Условную компиляцию можно с успехом применять при отладке программы, включая или выключая компиляцию наборов строк программы.

#if
<текст 1>
#else
<текст 2>
#endif

необязательная часть

Пример:

```
#define DEBUG
#ifdef DEBUG
printf("отладочная печать");
#endif
```

Так можно проверить, определён ли идентификатор.

Файлы, предназначенные для препроцессорного включения, обычно снабжаются защитой от повторного включения, которое может произойти, если имеется несколько файлов, в каждом из которых, в свою очередь, запланировано препроцессорное включение одного и того же файла, объединяются в общий текст программы.

Пример:

// __FILE_NAME – зарезервированное имя для файла
#ifndef __FILE_NAME
#define __FILE_NAME
<текст>
#endif
***#line* – смена номера следующей ниже строки,**
***#error* – оформление текстового сообщения об ошибке трансляции,**
***#pragma* – действия, предусмотренные реализацией,**
 # – пустая директива.

Препроцессорные операции:

***defined* – проверка определённосте операнда,**
– конкатенация препроцессорных лексем,
 # – преобразование операнда в строку символов.


исходный текст программы (*.c)
включаемые файлы
препроцессорная обработка
полный текст программы
выполняется преобразование текста программы. Препроцессор ищет строки, начинающиеся с #, и преобразует текст на основе их инструкций, т.е. заменяет одни выражения на другие простой подстановкой.
лексический анализ
текст преобразуется в поток лексем
синтаксический анализ
формальная проверка синтаксиса языка (правил, определяющих допустимые конструкции языка из букв алфавита)
семантический анализ
текст преобразуется в поток конструкций, осуществляется перевод каждой языковой конструкции в машинный код
объектный код программы (*.obj)
стандартные библиотеки
компоновка (сборка), редактирование связей
Компоновщик выбирает нужные функции, которые исполняются в программе и подключает к объектному коду программы коды библиотек функций
исполняемая программа (*.exe)
Исполняемый файл можно сохранить в памяти. Он имеет самостоятельное значение и может работать под управлением ОС.

Типы данных

Реальные данные, которые обрабатывает программа, – целые и вещественные числа, символы и логические величины. Эти простые типы данных называются базовыми. Тип определяет:

1. способ записи информации в ячейки памяти,
2. необходимый объём памяти для её хранения,
3. набор операций обработки данных, размещённых в памяти.

Объём памяти для каждого типа определяется таким образом, чтобы в него можно было поместить любое значение из допустимого диапазона данного типа. Например, для типа **char** допустимый диапазон от –128 до 127; значение занимает в памяти 1 байт


младший адрес
code segment (команды программы)
data segment (статические данные)
heap (куча) – динамическая память
Свободная память
stack segment (стек)
старший адрес
Обычно в этот сегмент не обращаются, но в С можно использовать адреса «входов» в функцию: начальный адрес функйий (или указатель на неё)
Неизменяемая память. Хранение глобальных и статических переменных (2части: инициализированные и неиниц. переменные). Для неинициализированных происходит обнуление памяти.
Куча – динамическая память: она выделяется и очищается в ходе выполнения программы с использованием указателей.
Хранение локальных переменных, фактических параметров функций, автоматических переменные функции, адреса возврата функций, временных переменных. Размеры стека изменяются в ходе выполнения программы.

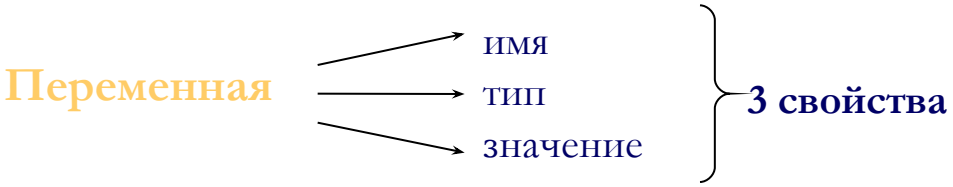
Память используется для 4 целей:
Размещение программного кода.
Размещение данных.
Для динамического использования.
Резервирование компилятором на время выполнения программы.

Модели памяти

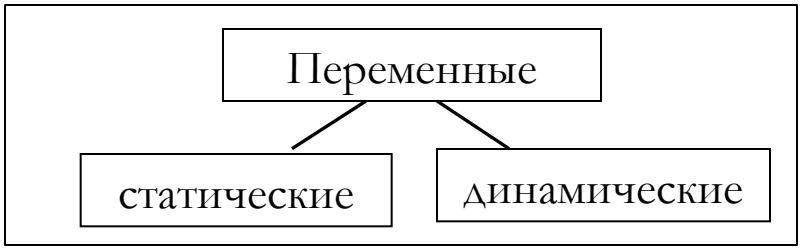
Существует 6 моделей памяти:
tiny – крошечная,
small – маленькая,
medium – средняя,
compact – компактная,
large – большая,
huge – огромная.
Модель памяти выбирается в меню Options->Compiler->Code generation.

Понятие переменной

Переменная – именованный объект, который может изменять своё значение.
Переменная характеризуется именем (идентификатором), типом и значением:

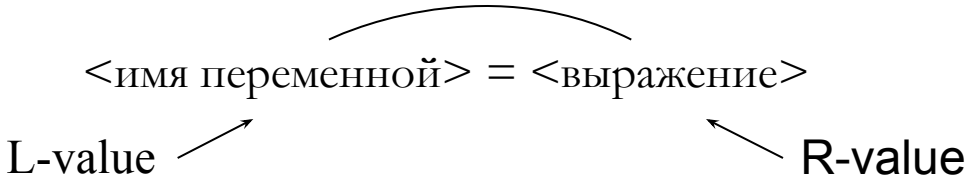


Имя – идентификатор (позволяет отвлечься от адреса в памяти)
Тип определяет способ хранения, размер памяти и операции, которые можно применять.
Переменные бывают двух видов:



Статические присутствуют в программе на протяжении всего времени работы программы (это, например, глобальные переменные).
Динамические создаются и удаляются на разных этапах выполнения программы (к примеру, локальные переменные внутри функции).

Операция присваивания



Пример:
a = b; // присваивание выполняется справа налево, в свою очередь слева от знака = может быть только модифицируемое леводопустимое значение.
Поэтому операцию a++ можно выполнить, а 10++ нельзя, т.к. 10 – константа.

Область действия переменных

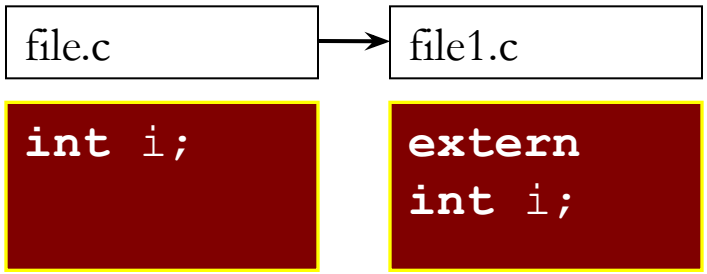
Определение переменной – это объявление объекта, которое сообщает компилятору его имя и тип. Говорят, что объявление связывает идентификатор с атрибутами типа. И заставляет компилятор выделить память для его хранения.
Для переменной возможно одно определение, но произвольное количество описаний.
Пример 1.

```
int i = 5;
int j = i + 3;
```

Пример 2.

```
int i;
int i; // здесь будет ошибка
```

Пример 3.



повторное описание для одноразового выделения памяти

extern означает, что переменная внешняя

Область действия переменной определяет права доступа разных частей программы к этой переменной, т.е. в каких частях программы эта переменная видима, а в каких – нет. В С термины ***«область доступности»*** и ***«область видимости»*** взаимозаменяемы.
Область действия влияет на ***время жизни*** переменной, т.е. на время, в течение которого переменная хранится в памяти.
Способ хранения переменной в памяти при выполнении программы определяется ***классом памяти.***

При выборе класса памяти предпочтение отдаётся автоматическим переменным. Другие классы памяти используется в случае необходимости.

Класс auto

Переменная класса **auto** имеет локальное время жизни и видима только в блоке, в котором определена. Память для неё выделяется при входе в блок и освобождается при выходе из блока. При повторном входе в блок для этой переменной может быть выделен другой участок памяти. Переменная этого класса автоматически не инициализируется (считается неопределённой).

Класс register

Указывает компилятору хранить значение переменной в регистре, если это возможно. Использование регистровой памяти сокращает время доступа к переменной. число регистров, которое можно использовать, ограничено возможностями процессора; если компилятор не имеет в распоряжении свободных регистров, переменная использует память как для класса **auto** (переменная становится автоматической). Этот класс памяти может быть указан только для переменных целочисленного типа или для указателей с размером, равным размеру целочисленной переменной

Класс static

Обеспечивает возможность С хранить значение переменной при выходе из блкоа и использовать его при повторном входе в блок. В отличие от переменных класса **auto**, память для которых выделяется в стеке, для статических переменных память выделяется в сегменте данных, поэтому их значение сохраняется при выходе из числа. Переменные этого класса могут быть инициализированы константным выражением. Если явной инициализации нет, такой переменной присваивается значение 0. Инициализация выполняется один раз при первом входе в блок.

Пример:

```
void func(void)
{
  // x будет инициализирована
  // только при первом вызове
  static int x = 1;
  int y = 1;
  // cout – поток вывода в C++
  cout << x++ << ‘ ‘ << y++ << ’\n’;
}
```

Внешняя память

Если переменная объявлена вне функции, она является глобальной по отношению к каждой функции. При вызове каждой функции глобальная переменная видна внутри неё. Глобальная переменная может быть скрыта внутри блока (говорят «затенена»), если в этом блоке определена переменная с тем же именем, что и у глобальной. При этом такая переменная не влияет на значение глобальной переменной, потому что этим переменным выделяются разные участки памяти: глобальным – в сегменте данных, автоматическим – в стеке.

Пример.

```
#include …
int N = 5;
void f(void)
{
  cout << N--;
}
int main()
{
  int N;
  for(N=0;N<5;N++)
    f();
  return 0;
}
```

Результат выполнения программы:

Если переменная объявленна локально (внутри функции) с классом памяти **extern**, то это означает ссылку на переменную с тем же именем, определённую глобально в одном из исходных файлов программы. Цель такого объявления – сделать определение переменной глобального уровня видимым внутри блока.
Цель этого объявления — сделать определеную переменную глобального уровня видимой внутри блока

Пример:

Объявление a[] как extern делает её видимой внутри функции func.
Определение этой переменной находится в файле file1.c на глобальном уровне и должно быть сделано только один раз, в то время как ***объявление*** с классом памяти extern может быть сделано несколько раз.
Также объявление с классом памяти extern требуется при необходимости использовать переменную, описанную в текущем исходном файле, но ниже по тексту программы, т.е. до выполнения её глобального определения.

Пример:

```
main()
{
  {
    extern int x[];
  }
  static int x[100];
  f() { ... }
}
```

Объявление со спецификатором **extern** информирует компилятор о том, что память для переменной выделять не требуется, т.к. это действие выполнено где-то в другом месте программы.

При объявлении переменных на глобальном уровне может быть использован спецификатор класса памяти **static** или **extern**, а также можно объявить переменную без указания класса памяти. Классы памяти **auto** и **register** для глобальных объявлений недопустимы. Объявление переменных на глобальном уровне – это или определение переменных, или ссылки на определения, сделанные в другом месте программы. Объявление глобальной переменной, которое инициализирует эту переменную (явно или неявно), является определением переменной. Определение на глобальном уровне можно задавать в следующих формах:

1. Переменная объявляется с классом памяти static:
   static int i = 0;
2. Переменная объявлена без указания класса памяти, но с явной инициализацией. Такой переменной по умолчанию присваивается класс **static**:
   int x = 5; // ***равносильно static int x=5*** ;

Переменная, объявленная глобально, видима в пределах остатка исходного файла, в котором она определена. Выше своего описания и в других исходных файлах эта переменная невидима, если только не была объявлена с классом **extern**. Глобальная переменная может быть определена только 1 раз (в пределах своей области видимости). В другом исходном файле может быть объявлена другая глобальная переменна с таким же именем и классом памяти **static**.

Конфликта не возникает, т.к. каждая из этих переменных будет видима только в своём исходном файле. Спецификатор класса памяти **extern** для глобальных переменных используют как и для локального объявления в качестве ссылки на переменную, объявленную в другом месте программы, т.е. для расширения области видимости переменной. При таком объявлении область видимости переменной расширяется до конца исходного файла, в котором она объявлена. В объявлениях с классом памяти **extern** не допускается инициализация, т.к. эти объявления ссылаются на уже существующие и определённые ранее переменные. Переменная, на которую делается ссылка с помощью спецификатора **extern**, может быть определена только один раз в одном из исходных файлов.

Определение функции

Функцией называется независимый фрагмент кода, имеющий собственное имя, предназначенный для выполнения определённой задачи и возвращающий значений в вызывающую программу (последнее не обязательно).

Имя функции. У каждой функции есть **имя**. Используя это имя в другой части программы выполняются операторы, содержащиеся в функции (это называется **вызовом функции**). Любую функцию можно вызывать из другой функции (кроме main).

Независимость. Функция выполняет свою задачу без вмешательства других функций.

Возвращаемое значение позволяет передать определённую информацию в вызывающую программу:

```
#include <iostream.h>
double cub(double x); // прототип функции
int main()
{
  double x;
  cin >> x;
  cout << cub(x);
  return 0;
}
double cub(double x)
{ return x*x*x; }
```

Прототип функции необходим , чтобы при компиляции вызовам функции был поставлен в соответствие формат её определения. Он состоит из **спецификатора** памяти (определённый тип возвращаемого значения), **имени** функции, **стека** передаваемых в неё параметров. Имена параметров указывать не обязательно, достаточно перечислить их типы.

Синтаксис прототипа функции:

тип_возвращаемого_значения имя_функции(тип_аргумента1, тип_аргумента2...);

(наличие имён аргументов необязательно)

Пример:

```
int func(int, double, int); // описание прототипа функции
```

Сама функция и её полный текст называется **определением** функции.

Если прототип содержит имена аргументов, тогда первая строка определения (заголовок функции) должен полностью совпадать с прототипом (за исключением “;” после прототипа). Кроме того, в отличие от прототипа, имена аргументов в определении функции необходимы.

При определении функции после заголовка открывается операторная скобка.

Тело фукции — все её выполняемые операторы описывают метод, реализуемый данной функцией.

Возвращаемое значение передаётся оператором **return**, за которым следует выражение, определяющее это возвращаемое значение. Когда программа подходит к оператору **return**, вычисляется выражение, стоящие за ним, и его значение передаётся в вызывающую программу с одновременной передачей управления в неё. Функция может содержать несколько операторов **return**.
Можно объявить функцию, не возвращающую никакого значения, задавая пустой тип **void**.

Пример:

```
void f1(...)
{ ... } // функция ничего не возвращает
```

Операторы цикла

Цикл — разновидность управляющей конструкции в высокоуровневых языках программирования, предназначенная для организации многократного исполнения набора инструкций. Также циклом может называться любая многократно исполняемая последовательность инструкций, организованная любым способом (например, с помощью условного перехода).

В языке Си цикл с параметром и цикл с предусловием построены по общей схеме:

```
  заголовок_цикла
  <тело цикла>
```

Формат цикла с предусловием:

```
while ( условное_выражение )
{
  <тело цикла>
}
```

Параметрический цикл (цикл со счётчиком):

```
for(выражение1; условное_выражение; выражение2)
{
  <тело цикла>
}
```

Выражение1 — определяет действие, выполняемое до начала цикла, т.е. задаёт начальное условие цикла. Как правило, оно является выражением присваивания.

Операция "запятая" - ","

Запятая может быть использована в качестве операции и в качестве разделителя. Разделитель используется для разделения элементов списка, например:

int x[]={1,2,3,6};

Разделитель может использоваться и в заголовке оператора цикла **for**

Пример:

```
for (i=1, j=1, s=0; i<n; i++, j--, s+=i*j);
```

Условное выражение

Определяет условие окончания работы цикла. Цикл выполняется до тех пор, пока значение цикла выражения истинно (!=0)

После определения истинности условного выражения выполняется тело цикла. А затем вычисляется **выражение 2,** которое задаёт правила изменения параметров или любых переменных тела цикла.

Цикл с постусловием

Цикл с постусловием — цикл, в котором условие проверяется после выполнения тела цикла. Отсюда следует, что тело всегда выполняется хотя бы один раз.

Формат цикла с постусловием:

```
do
{
  <тело цикла>
}while(условное_выражение);
```

При таком подходе задача разбивается на отдельные подзадачи, каждая из которых выполняется независимыми фрагментами кода

Преимущества структурного программирования

1)Структурную программу легче понимать, т. к. сложная задача разбивается на много мелких простых. Каждая задача выполняется функцией, в которой и код, и переменные изолированы от остальной части программы.

2)Структурную программу легче отлаживать. Структурированность локализует ошибки.

Обработка вызова функции

При обработке вызова функции компилятор вставляет в код программы последовательность машинных команд, выполняющих следующие действия:
Выделение в стеке ячеек памяти, в которые записываются копии значений фактических параметров, адрес возврата и другие локальные переменные.
Если фактические параметры являются выражениями, то эти выражения будут вычислены и при необходимости приведены к соответствующему виду.
Пример 1.

```
int f(int, float) { ... }
void main()
{
  x = f(a+2, b+a); // параметры будут вычислены
                   // и в функцию будут переданы их значения
  x = f(1, 2); // 2 будет приведено к типу float
}
```

Пример 2.

```
i = 0;
f(i, ++i)  // f(0,1) или f(1,1)?
```

В С принят такой порядок передачи аргументов, при котором первым вычисляется и передаётся последний в списке фактический параметр.
Т.к. порядок вычисления выражений фактических аргументов играет существенную роль, во избежание ошибок лучше вначале вычислять эти выражения, а затем подставлять в вызов функции.

Формальные и фактические параметры

Формальными параметрами (аргументами) принято называть параметры, которые используются при описании функции. С их помощью определяется место в стэке, так называемом frame функции. На их основе описывается алгоритм обработки фактических параметров.

Фактическими параметрами (параметрами) называются те параметры, которые используются при вызове функции. Именно их копии значений будут записаны в frame функции для дальшейшего использования. На их же основе будет вычислено возвращаемое значение функции. На место аргументов копируются значения передаваемых фактических параметров. Изменяя значение передаваемых параметров, мы изменяем копии объектов вызывающей функции.

Время жизни аргументов — время выполнения функции. Оно начинается с вызова функции (аргументы получают значения фактических параметров). Как только функция выполнена, аргументы теряют и свои значения и фрэйм.

Передача в функцию адресов переменных

Чтобы с помощью функций изменять значение аргументов надо в качестве параметров передавать не переменные, а их *адреса*. Тогда фрэйм функции будет содержать адрес переменной, по которому можно будет обратиться к самому аргументу и изменить его по своему усмотрению.

Указатель – это переменная, которая содержит адрес памяти, т.е. сообщает о том, где размещён объект, и не говорит о самом объекте. Символ «*» используется для задания указателя на объект.

```
int* x; // x – указатель на целое
char *p, a='+';
p=&a; //адрес
printf("%c %c", a, *p); // *p - содержимое
```

Унарная операция * – обращение по адресу, т.е. раскрытие указателя или **операция разадресации** (доступ по адресу к значению того объекта, на который указывает операнд). Операндом должен быть указатель (*p, где p – указатель) (у этой операции ранг 2).

Объявление указателей

тип* имя_указателя;

Типом может быть любой из типов данных Си. Он изменяет тип переменной, на который указывает указатель. Указатель можно объявить вместе с переменными:

```
char *ch, c;
int *p,x,v;
```

Инициализация указателей

Использование неинициализированных указателей потенциально опасно, хотя и возможно. Пока указатель не содержит адреса переменной, он бесполезен. Адрес в указатель помещается с помощью **операции взятии адреса** – &.

Для обозначения указателя часто используют такие имена: pointer, ptr, p.

Указатель на тип **void** совместим с любым указателем:

```
void * y;
int *x;
x = y;
```

Пример.

```
main() {
  int *x,*w,y,z;
  *x = 16; // здесь допущена серьёзная ошибка, распространённая и опасная
  y = –16;
  w = &y;
}
```

По адресу, задаваемому в x, помещается значение 16. Ошибка – не инициализирован указатель. В 1 строке – компилятор резервирует память (*x, *w – под указатель, y, z – для переменных типа int).

Для выделения памяти существуют специальные функции, с помощью которых запрашивается и выделяется память из кучи.

Передача в функцию адресов переменных:

```
void change(int* px, int* py) // * означает указатель
{
  // * означает получение значения по адресу
  (*px)++; // увеличиваем значение
  (*py)—; // уменьшаем значение
}
// Если не использовать скобки, то мы будем изменять  не значение, а адрес
void main()
{
  int x=1,*px=&x, y=2;
  change(px,&y); // x=2,y=1
}
```

В программах на С широко используются библиотечные функции. Их прототипы находятся в специальных заголовочных файлах, поставляемых вместе с библиотеками в составе систем программирования и включающихся с помощью директивы #include
Имя функции (или идентификатор, объявляемый как функция) представляет указатель, значение которого является адресом функции, возвращающий значение определённого типа.
Адрес функции не изменяется во время выполнения программы, меняются только возвращаемые значения. Идентификаторы функций — константы, соответственно они не могут стоять в левой части операции присваивания, то есть они являются праводопустимыми выражениями.
Спецификатор класса памяти для функции необязателен, он задаёт класс памяти функции и может быть только **static** или **extern**, т.е функция всегда глобальна.

Правила определения области видимости функции

Функция, объявляемая как **static**, видима в пределах того файла, в котором она определена. Любая функция может вызвать другую функцию с классом памяти **static** из этого же файла, но не может вызвать функцию, определённую с классом **static** в другом файле. Разные функции с классом памяти static, имеющие одинаковые имена, могут быть определены в разных исходных файлах, и это не ведёт к ошибке (конфликту).

Функция, объявленная с классом **extern**, видима в пределах всех исходных файлов программы. Любая функция может вызывать функции с классом памяти **extern**. Если в объявлении отсутствует спецификатор класса памяти, то по умолчанию устанавливается класс **extern**. Все объекты с классом памяти **extern** компилятор помещает в объектном файле в специальную таблицу внешних ссылок, которые используются редактором связей для разрешения внешних ссылок. Часть внешних ссылок перёдается компилятором при обращении к библиотечным функциям С, поэтому для разрешения этих ссылок редактору связи должны быть доступны соответствующие библиотечные функции.

Имеется возможность объявить в одном прототипе несколько функций, если эти функции возвращают значения одного типа и имеют одинаковые списки формальных параметров. указав имя одной из функций в качестве имени функции, а все другие поместить в список других имён функций, причём каждая функций должна сопровождаться списком формальных параметров.

double area (double);
double len(double); ⟶ double area (double), len (double);

Если прототип не задан и встречается вызов функции, то строится неявный прототип из анализа формы вызова функции с типом возвращаемого значения **int.** А список формальных параметров формулируется из вызова.

Прототипы функциям необходимо задавать, если:
1) Функция возвращает значение типа, отличное от **int**
2) Требуется проинициализировать некоторый указатель на функцию до того, как эта функция будет определена.

Рекомендуется всегда задавать прототипы функций.
Вызов функции имеет следующую форму:
1)Любую ф-ю можно вызвать, указав её имя со списком фактических параметров.
имя_функции(параметр1, параметр2);
Функция выполняется, а возвращаемое значение игнорируется.
2)Функции можно использовать в выражениях. Так можно вызывать функции, которые возвращают значения НЕ void.
a=f(x)+f(x+y);
if(func(x)!=2){..}
При попытке использовать в выражении ф-ю, возвращающую void, компилятор выдаст ошибку.

Операции над указателями

Массив – совокупность однотипных элементов, отличающихся номером элементов и имеющихся общее имя.
Объявление массива имеет 2 формы:
спецификатор_типа идентификатор[константное выражение]
спецификатор_типа идентификатор[]
Спецификатор типа служит для определения, какого типа элементы будут храниться в памяти.
Идентификатор – это имя массива.
[константное выражение] – количество элементов массива.
Элементами массива не могут быть функции и элементы типа **void**.
При объявлении массива константное выражение может быть опущено, если:
1)при объявлении массив инициализируется,
2)массив объявлен как формальный параметр,
3)массив объявлен как ссылка на массив, явно определённый в другом файле.
В С определены только одномерные массивы, но, т.к. элементом массива может быть массив, можно определить и многомерные массивы.
Пример объявления символьного массива с инициализацией:
char str[] = “объявление символьного массива”;
Идентификатор объявления массива представляет собой указатель – константу, значением которой является адрес первого элемента массива. Тип, адресуемый указателем, это тип элементов массива. Значение указателя изменить нельзя, т.к. идентификатор массива не является L-value выражением.

Пример:

```
int arr[25];
int *p;
p = arr; // указателю p присвоить адрес 0-го элемента массива
*arr = 2;   // нулевому элемент массива присвоить 2
arr[0] = 2;    // то же самое
*p = 2;        // - * -
*(arr+0) = 2;  // - * -
p[0] = 2;      // - * -
*(arr+16) = 3; // 16-му элементу массива присвоить 3
int i = 5;
*(arr+i+2) = 1; // 7-му элементу присвоить 1
[7]arr = 1;     // - * -
```

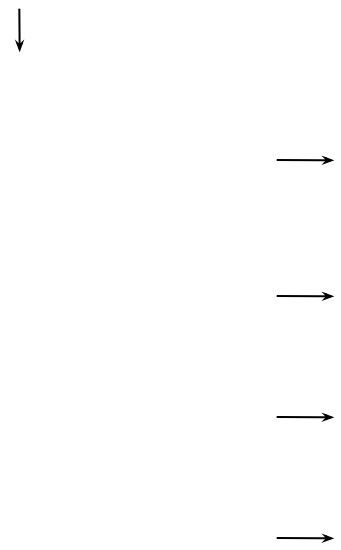
Указатели на многомерные массивы

Указатели на многомерные массиве в языке С являются по сути массивами массивов. При объявлении таких массивов в памяти создаётся несколько разных объектов. Например, объявление

int arr[4][3];

порождает 3 разных объекта:

1) указатель с идентификатором arr,
2) безымянный массив из 4 указателей,
3) безымянный массив из 12 чисел типа int.



Пример:

```
int a[ ] = {10, 11, 12, 13, 14};
int* p[ ] = {a, a+1, a+2, a+3, a+4};
int** pp = p; // pp ссылается на 0-й элемент
pp += 3; // pp теперь ссылается на 3-й элемент
pp –= 1; // pp теперь ссылается на 2-й элемент
```

Массивы указателей типа char (указатели и строки)

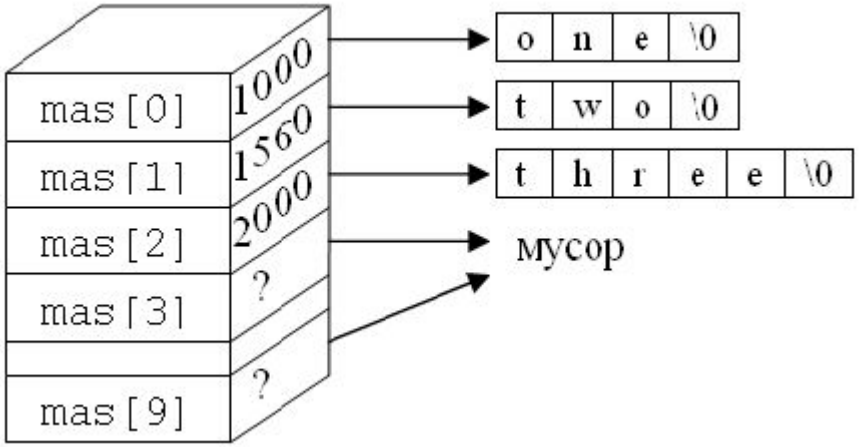
Инициализация символьного массива:

```
char b[8] = “student”;
char b[8] = {‘s’, ‘t’, ‘u’, ‘d’, ‘e’, ‘n’, ‘t’, ‘\0’};
```

Если при объявлении массива происходит его инициализация, компилятор сам вычисляет его длину по длине строки.

Следующий оператор объявляет массив из 10 указателей на строки:

```
char* mas[10] = {“one”, “two”, “three”, …}; // (*)
```



Оператор (*) выполняет следующие действия:
Размещает в памяти массив из 10 элементов с именем mas. Каждый элемент массива является указателем типа char.
Выделяет пространство в памяти (где именно – определяет компилятор) и размещает в нём заданные строки, каждая из которых завершается 0-терминатором.
Инициализируются указатели mas[0], mas[1], mas[2] – адреса первых символов соответствующих строк. Элементы массива mas[3..9] не инициализируются никакими адресами

Указатели на функцию

Указатели на функцию являются одним из дополнительных средств вызова функции. При выполнении программы для каждой её функцией определяется конкретный адрес. Указатель на функцию содержит начальный адрес функции, т.е. её точку входа.

Объявление указателя на функцию

Как и любая переменная, указатель на функцию должен быть объявлен, чтобы его можно было использовать.
Общая форма объявления указателя на функцию:

тип_ф-ции(*ptr_func) (список параметров);

// ↑ идентификатор

Этот оператор объявляет переменную ptr_func указателем на функцию, возвращающую значение заданного типа и принимающую указанный список параметров.
Примеры объявлений:

```
int (*f1)(int a);
double (*f2)(double x, double y);
void (*f3)(char* p);
```

Зачем нужны скобки? Рассмотрим другой пример.

```
int* f(int a);
```
Приоритет операции * ниже, чем приоритет скобок, поэтому приведённый пример без скобок вокруг имени указателя является объявлением функции f, возвращающей указатель типа int и не является указателем на функцию.

Инициализация и использование

Пример объявления и инициализации указателя на функцию:

```
int sqr(int i);
int (*ptr)(int i);
int sqr(int i)
{
  return i*i;
}
int main()
{
  ptr = sqr; // инициализация происходит, т.к.
             // ptr и sqr имеют одинаковые
             // список параметров и тип
             // возвращаемого значения
  int x;
  x = ptr(7); // вызов функции sqr через ptr
  return x;
}
```

Передача указателя как аргумента в функцию

```
#include <stdio.h>
void one(void);
void two(void);
void other(void);
void func(void (*p)(void));
int main()
{
  void (*ptr)(void);
  int n;
  for(;;) // бесконечный цикл
  {
    puts("Введите целое число между 1 и 30, 0 – выход: ");
    scanf("%d", &n);
    if(!n) break; // если 0, то выход
    switch(n)
    {
      case 1: ptr = one; break;
      case 2: ptr = two; break;
      default: ptr = other;
    }
    func(ptr);
  }
}
```

```
void one(void)
{
  puts("you entered 1");
}
void two(void)
{
  puts("you entered 2");
}
void other(void)
{
  puts("you entered some other than 1 or 2");
}
void func(void (*p)(void))
{
  p();
}
```

Встраиваемые функции (inline)

Встраиваемые функции обычно небольшие по размеру. Компилятор постарается выполнить самым быстрым возможным способом.

Термин «встраиваемая» возник от того, что вместо вызова функции выполняется её прямая вставка (встраивание) в тело вызывающей функции. Добавив к объявлению функции ключевое слово inline делам функцию встраиваемой. При этом компилятор постарается максимально оптимизировать её выполнение.

Следует знать, что вместо вызова функции скорее всего, но не обязательно, будет подставлен код в вызывающую функцию. Единственное, что гарантирует компилятор, это оптимизация выполнения функции. Вызов встраиваемой функции не отличается от вызова любых других функций.

Пример объявления: **inline int** f(**int** x) { …; }

Передача массивов в функцию

Единственный способ передать в функцию массив – это воспользоваться указателем. Известно, что аргумент функции обязательно должен быть единичным значением. Указатель на массив представляет собой одиночное числовое значение – адрес нулевого элемента массива. Если передать его в функцию, функция будет знать адрес массива и сможет обратиться к нему с помощью операции ссылки по указателю.

Если функция принимает аргумент-массив, то желательно, чтобы она работала с массивами различной длины. Существует два способа сообщить функции размер массива:

Можно поместить в последний элемент массива специальное терминирующее значение. Недостаток: зарезервированное значение служит единственной целью – указывает конец массива. Это ухудшает гибкость обработки информации в массиве.

Способ более гибкий и непосредственный – передача длины массива в функцию в явном виде. Функция получит минимум 2 элемента: указатель на нулевой элемент массива и целое число, обозначающее количество элементов этого массива.

Пример: нахождение max элемента массива.

Описать функцию, возвращающую индекс первого наибольшего элемента массива.

```
int index_max(int arr[], int len)
{
  int i, imax=0;
  for (i=1;i<len;i++)
    if (arr[i]>arr[imax])
     imax = i;
  return imax;
}
```

Форма записи параметра int arr[] эквивалентна int* arr и означает указатель на int. Первая форма часто бывает предпочтительней, т.к. напоминает, что мы имеем дело с указателем на массив.

Массивы и динамическая память

Формирование массив с переменными размерами можно организовать с помощью указателей и средств для динамического выделения памяти.

Начнём рассмотрение указанных средств с библиотечных функций, которые описаны в файле **stdlib.h**

Функции, использующиеся для работы с динамической памятью

malloc()

calloc()

realloc()

malloc()- функция выделения динамической памяти, входящая в стандартную библиотеку языка Си.

Прототип:

```
void *malloc (size_t size);
```

size – размер распределяемой области памяти

принимает в качестве аргумента размер выделяемой области в байтах; возвращает нетипизированный указатель (void*) на область памяти заявленного размера или NULL в случае, если выделить память невозможно. Содержимое выделяемой области памяти не определено.

calloc() работает как и malloc(), но она так же предназначена для очищения памяти.

Прототип:

```
void *calloc (size_t num, size_t size);
```

num – количество распределяемых элементов

size – размер каждого элемента

calloc принимает в качестве аргумента количество элементов и размер каждого элемента в байтах; возвращает нетипизированный указатель (**void***) на область памяти заявленного размера или **NULL** в случае, если выделить память невозможно. Значения элементов устанавливаются в ноль. **malloc** работает быстрее, чем **calloc**, в связи с отсутствием функции обнуления выделяемой памяти.

realloc() изменяет размер блока памяти, ранее созданного функцией **malloc** или **calloc**

Прототип:

```
void *realloc (void* ptr, size_t size);
```

ptr — указатель на исходный блок памяти

size — новый размер в байтах.

При вызове realloc возможен один из следующих исходов:

1) Если для расширения блока, находящегося по адресу **ptr**, имеется достаточно памяти, то происходит её выделение в нужном количистве и функция возвращает **ptr**.

2) Если памяти недостаточно для расширения по текущему адресу, создаётся новый блок размера size и имеющиеся данные копируются из старого блока в начало нового. Старый блок освобождается. Ф-я возвращает указатель на новый блок памяти.

3) Если **ptr==NULL**, то функция действует так же, как и **malloc**

4) Если аргумент **size == 0**, тогда блок памяти **ptr** освобождается, а функция возвращает **NULL**.

5) Если для перераспределения памяти недостаточно места: нельзя расширить старый блок или создать новый — функция возвращает **NULL**, а исходный блок остаётся неизменным.

Функция free() решает вопрос освобождения памяти, которая перед этим была выделена одной из трёх вышеприведённых функций. Сведения об этом участке памяти передаются в функцию free с помощью указателя – параметра типа void*.

Пример создания динамического массива

```
int i,n,m,**A; // ** - двумерный массив
scanf("%d %d",&n,&m);
A =(int**)malloc(n*sizeof(int*));
for(i=0;i<n;i++)
{
  A[i] = (int*)malloc(m*sizeof(int));
}
```

```
for(i=0;i<n;i++)
free(a[i]);
free(a );
```

Манипулирование блоками памяти.

Инициализация памяти с помощью функции **memset**(); Эта функция используется для того, чтобы сделать все байты определённого блока равными одному и тому же значению.

Прототип:

```
void *memset(void *dest, int c, size_t n);
```

dest – указатель на блок памяти

c – значение, которое нужно поместить в байты блока (от 0 до 255).

n – количество этих байт, начиная с **dest**

Так как функция **memset** инициализирует блок памяти побайтно значениями типа **char**, то её не имеет смысла использовать для работы с блоками других типов. Исключение – обнуление блока с помощью этой функции. Так массив чисел типа **int** нельзя заполнить, но можно обнулить.

Копирование блоков памяти функцией memcpy();

Функция копирует байты данных из одного блока памяти в другой. Такие блоки называются **буферами**. Функция не различает тип копируемых данных, а выполняет перенос байт за байтом.

Прототип:

```
void *memcpy(void *s1, conts void *s2, size_t n);
```

s1 — начало блока памяти, куда производить копирование

s2 — начало блока памяти, откуда начинать копирование

n — количество копируемых байт

Функция возвращает значение s1.

Если два блока памяти накладываются один на другой, функция может работать некорректно. Часть данных блока-источника окажется затёртой ещё до копирования. Для работы с перекрывающимися блоками используют функцию **memmove**();

Функция **memmove**(); по назначению напоминает функцию **memcpy**(); Она так же производит копирование побайтно, но она обладает большей гибкостью, так как конкретно справляется с ситуацией наложения блоков.

Прототип:

```
void *memmove(void *s1, conts void *s2, size_t n);
```

Аргументы такие же, как и в **memcpy**();

Операции над строками

Библиотечные функции обработки строк можно использовать, подключив заголовочный файл **string.h**

Определение длины строки

Функция **strlen**();

Прототип:

```
size_t strlen(const char *s);
```

Функция возвращает целое число без знака, равное количеству символов в строке s (без \0)

Копирование строк

Функция **strcpy**();

Прототип:

```
char *strcpy(char *s1, const char *s2);
```

Выполняет копирование строки, находящейся по адресу **s2** вместе с \0 в участок памяти, начинающийся по адресу в указателе **s1**. Функция возвращает указатель на новую строку **s1**.

Функция **strncpy**();

Прототип:

```
char *strncpy(char *s1, const char *s2, size_t n);
```

Эта функция аналогична **strcpy**() за тем исключением, что с её помощью можно копировать определённое число символов. Число копируемых символов — переменная **n**.

Если строка **s2** короче, чем n символов, то к ней добавляется достаточное количество символов '**\0**', чтобы всего в **s1** их копировалось ровно **n**. Если строка **s2** длиннее, чем **n** символов, то к **s1** не добавляется завершающий ноль-терминатор. Функция **strncpy**(); возвращает указатель на **s1**.

Замечание:

При копировании строк необходимо убедиться, что количество копируемых символов не превосходит длины буфера, в который они копируются. К тому же необходимо помнить, что strncpy не добавляет завершающий ноль-терминатор в конец строки.

Функция **strdup**();

Ещё одна функция копирования По назначению аналогична **strcpy**(), но она сама выполняет распределение памяти для буфера, в который копируется строка с помощью вызова функции **malloc**(), а затем копирует строки, используя **strcpy**(). Если выделение памяти закончилось неудачей, то создания копии не происходит и функция возвращает **NULL**. Эта функция не определена в стандарте ANSII, но входит в библиотеки множества компиляторов.

Прототип:

```
char *strdup(char *src);
```

Функция возвращает указатель на начало созданного ею буфера памяти, куда скопирована строка **src**.

Внимание: так как функция выполняет распределение памяти, то после использования строки, в которую производили копирование, её необходимо очистить с помощью **free**().

Пример использования функции **strdup**():

```
#include <iostream.h>
#include <string.h>
#include <alloc.h>
char source[]="Test string";

int main(){
    char *dest;
    if((dest=strdup(source))==NULL)
    {
        cout << "Error!";
        return 1;
    }
    cout << dest;
    free(dest);
    return 0;
}
```

Конкатенация строк

Прототип:

```
char *strcat(char *s1, const char *s2);
```

Эта функция помещает копию строки **s2** непосредственно после конца строки **s1** и ставит **\0** в конец. Предварительно необходимо позаботиться о том, чтобы в строке **s1**

было достаточно места для хранения результата сцепления двух строк. Сама функция **strcat**(); возвращает указатель на **s1**.
Пример использования функции **strcat**():

```
#include <iostream.h>
#include <string.h>

void main(){
    char str1[27]="a";
    char str2='\0';
    for(str2='b'; str2<='z'; str2++)
    {
       strcat(str1, str2);          //a
        cout << str1 << '\n';       //ab
    }                               //abc ...
}
```

Функция **strncat**();
Выполняет конкатенацию строк, но при этом позволяет указать сколько символов из исходной строки будет добавлено в конец строки назначения.
Прототип:

```
char *strncat(char *s1, const char *s2, size_t n);
```

В любом случае (длина **s2** > или < **n**) к результирующей строке будет добавлен '**\0**'.
Функция возвращает указатель на **s1**.

Сравнение строк

Строки сравниваются для выяснения их равенства. Если строки не равны, то определение "больше" или "меньше" основывается на кодах символов ASCII. В случае букв порядок следования кодов полностью совпадает с алфавитным порядком. Коды заглавных букв меньше, чем строчных.

Библиотека ANSI языка С содержит функции для 2х видов сравнения строк: сравнение двух целых строк и сравнение фрагментов строк определённой длины.

Функция **strcmp**();

Предназначена для посимвольного сравнения 2х строк

Прототип:

```
int strcmp(const char *s1, const char *s2);
```

Возвращаемые значения:

* **отрицательное число** — если **s1** < **s2**
* **ноль** — если **s1**==**s2**
* **положительное число** — если **s1** > **s2**

Функция **strncmp**();

Предназначена для сравнения фрагментов заданной длины из одной строки с другой строкой.

Прототип:

```
int strncmp(const char *s1, const char *s2, size_t n);
```

Функция сравнивает **n** символов **s1** со строкой **s2**. Сравнение идёт до тех пор, пока не исчерпаются все **n** символов или не будет достигнут конец **s1**. Способ сравнения и возвращаемые значения такие же, как и у **strcmp**(). Сравнение идёт с учётом регистров.

Сравнение строк без учёта регистра:
ANSI — функции нет.
Symantec — strcmpl();
Microsoft — stricmp();
Borland — stricmp(); и strcmpi();

Поиск в строках
Функция **strchr**();
Прототип:

```
char *strchr(const char *s, inc c);
```

Функция производит поиск в строке **s** слева направо, пока не найдёт символ **c** или пока строка не закончится ноль-терминатором. Если символ найден, функция возвращает указатель на него, иначе — **NULL**

Функция **strrchr**();
Прототип:

```
char *strrchr(const char *s, inc c);
```

По своему назначению аналогична предыдущей функции за исключением того, что она начинает поиск справа.

Функция strcspn();

Прототип:

```
size_t strcspn(const char *s1, const char *s2);
```

Функция ищет с первого символа строки **s1** первое вхождение любого из символов строки **s2**. Важно запомнить, что функция ищет в **s1** отдельные символы из **s2**, а не всю строку целиком. Как только совпадение найдено, функция возвращает смещение от начала строки **s1**, указывающее местонахождение найденного символа. Если совпадения не обнаруживается, то функция возвращает значение **strlen(s1)**. Это означает, что первое найденное соответствие — ноль терминатор (Т.к. \0 есть и в строке **s2**, ведь мы задаём строку, а значит она оканчивается ноль-терминатором).

Функция strspn();

Прототип:

```
size_t strspn(const char *s1, const char *s2);
```

Функция перебирает строку **s1** символ за символом, пока не встретится с символом, которого нет в строке **s2**. Она возвращает номер позиции первого символа в строке **s1**, не входящего в **s2**. Другими словами, функция возвращает длину начального фрагмента строки **s1**, состоящего только из символов, которые встречаются в **s2**.

Функция strpbrk();

Прототип:

```
char *strpbrk(const char *s1, const char *s2);
```

Функция **strpbrk**() возвращает указатель на символ в **s1**, соответствующий одному из символов в наборе **s2**, или **NULL**, если такого символа нет в строке.

Функция **strstr**();
Прототип:

```
char *strstr(const char *s1, const char *s2);
```

Она ищет первое появление одной строки внутри другой (всю строку). Функция возвращает указатель на первую позицию, с которой начинается строка **s2** внутри строки **s1**. Если между строками нет соответствия, функция возвращает **NULL**.Если вторая строка имеет длину 0, функция возвращает указатель на строку **s1**. Поиск и сравнение осуществляется с учётом регистров символов.

Функции преобразования символов в строках

Имеется две функции для изменения регистров символов внутри строки. Они не определены в стандарте ANSI.

```
char *strlwr(char *s);
char *strupr(char *s);
```

Первая функция преобразует все буквенные символы строки в маленькие. Вторая — в большие.

Другие функции преобразования строк

Функция **strrev**();
Прототип:

```
char *strrev(char *s1);
```

Изменяет порядок следования символов на противоположный.

Функции **strset**() и **strnset**()
Прототипы:

```
char *strset(char *s, int c);
char *strnset(char *s, int c, size_t n);
```

Первая функция заменяет все символы в строке **s**, а вторая только **n** символов, на один и тот же заданный символ **c**.

Преобразование строк в числа

Для использования следующих функций необходимо подключить заголовочный файл **stdlib.h**
Функция **atoi**();
Прототип:

```
int atoi(const char *nptr);
```

Функция преобразует строку **nptr** в целое число типа **int**, соответствующее строковому определителю. Кроме цифр строка может содержать пробелы в начале и знаки + и -. Преобразование начинается с первого символа строки и продолжается до тех пор, пока не встретится недопустимый в строке символ. Если функция не находит пригодных для преобразования в число символов, то возвращает 0.

Функция **atol**();
Прототип:

```
long atol(const char *nptr);
```

Ведёт себя точно так же, как и **atoi**(), только для значений типа **long**.

Функция **atof**();
Прототип:

```
double atof(const char *nptr);
```

Преобразует строку **nptr** в число с плавающей точкой. Она может содержать перед началом числа пробелы и знаки +, - и . Число может состоять из цифр от 0 до 9, десятичного разделителя, а так же знака показателя степени E или e.
Примеры:
"12" → 12.0
"-0.123" → -0.123
"12E+3" → 12000
"123.1e-5" → 0.001231

Функции анализа символов

Определение этих функций-макросов лежит в заголовочном файле **ctype.h**
Они анализируют значение, возвращая 1 или 0, в зависимости от того, удовлетворяет ли они определённым условиям.
Общий вид этих функций можно описать как

```
int is<xxxx>(int c);
```

Таблица макросов расположена на следующем слайде.

Таблица макросов анализа символов

Таблица функций, проверяющих тип символа

функция	проверяемое условие
isalpha(c)	c-латинская буква
isdigit(c)	c-десятичная цифра
isxdigit(c)	c- шестнадцатиричная цифра
isalnum(c)	c-латинская буква или цифра
iscntrl(c)	c-управляющий символ (0-0x1 F)
isprint(c)	c-печатаемый символ, включая пробел
ispunct(c)	c-знак пунктуации
isspace(c)	c- пробел, новая строка, табуляция
isupper(c)	c-буква верхнего регистра
islower(c)	c-буква нижнего регистра

Функции с переменным числом параметров

В Си допустимы функции, в которых не определено число параметров.
Прототип:

```
тип_возвр_значения имя_функции(список_явных_параметров,...);
```

Любая функция с переменным числом параметров должна иметь хотя бы 1 обязательный параметр. Такие функции должны иметь механизм определения количества параметров и их типов.

Первый способ:

Предполагает добавление в список обязательных параметров сведений о реальном количестве используемых фактических параметров и их типов.

Второй способ:

Предполагает добавление в конец списка необязательных параметров специального параметра — индикатора с уникальным значением, сигнализирующем об окончании списка.

Во всех случаях переход от первого фактического параметра к другому выполняется с помощью указателей.

Пример функции суммирования целых чисел:

```
#include <iostream.h>
long sum(int k, ...)
{
    int *pick = &k;
    long total=0;
    for(; k; k--) total+=*(++pick);
    return total;
}
void main(){
    cout << "Sum(2,1,3)" << sum(2,1,3);
    cout << "\nSum(5,1,2,3,4,5)" << sum(5,1,2,3,4,5);
}
```

Опасность — выход за предел адресов при неправильно заданных параметрах.

Обязательный параметр необходим, чтобы выяснить адрес нахождения необязательных параметров. Чтобы функция с переменным количеством параметров могла определять их типы, необходимо в качестве исходных данных передавать ей эту информацию.

Исправим предыдущий пример так, чтобы функция могла суммировать не только целые, но и вещественные числа.

После некоторых преобразований функция изменит вид на

```
…
double sum(char t, int k, ...){
    if (t=='i') int *pick = &k;
    else  double *p = &k;
…
```

Макро-средства для функций с переменным числом параметров

Для использования этих макросов необходимо подключить заголовочный файл **stdarg.h**
В нём определены три макро-средства:

```
void va_start(va_list param, <последний_явн_параметр>);
type va_arg(va_list param, type);
void va_end(va_list param);
```

В этом заголовочном файле определён специальный тип **va_list** для обработки переменных списков параметров.
Для обращения к макро-командам их первые параметры имеют тип **va_list**. В качестве второго параметра макроса **va_arg** используется обозначение типа очередного параметра, к которому выполняется доступ.
Порядок использования макросов:
1) В теле функции обязательно определение переменной типа **va_list**.
Например,
va_list ptr;
С помощью макроса va_start переменная ptr связывается с первым необязательным параметром (началом списка необязательных параметров).
va_start (ptr, последний_явный_параметр);
Дальше с помощью указателя ptr можно получить значение первого фактического параметра из списка переменной длины. Остаётся неизвестным тип этого параметра. Этот тип необходимо каким-либо образом передать в функцию, используя явные параметры.
Определив тип очередного параметра, обращаемся к макросу **va_arg(ptr, type)**, который вернёт значение искомого параметра.

va_end(ptr) предназначена для организации возврата из функции с переменным количеством параметров. Она должна быть вызвана после того, как будет обработан весь список параметров. Функция удаляет указатель для того, чтобы нельзя было использовать его повторно без вызова **va_start()**.

Пример функции с переменным числом параметров

```
#include <iostream.h>
#include <stdarg.h>

void miniprint(char *format, ...)
{
    va_list ap;
    char *p;
    int ii;
    double dd;
    va_start(ap, format);
    for(p=format; *p; p++)
    {
      if(*p!='%')
         printf("%c", *p);
      else
      {
```

```
            switch(*++p)
            {
              case 'd':
                 ii = va_arg(ap, int);
                 printf("%d", ii);
                 break;
              case 'f':
                 dd = va_arg(ap, double);
                 printf("%f", dd);
                 break;
              default:
                 printf("%c", *p);
            }
         }
    }
    va_end(ap);
}

void main()
{
    int k=123;
    double e=2.718282;
    miniprint("\nЦелое k= %d,\tчисло e= %f", k, e);
}
```

Структуры, объединения и сложные типы данных.

Структура — составной объект, в который входят элементы нескольких типов.

Объявление структуры:

```
struct <имя_структуры>{<определение_элементов>};
```

Пример

```
struct date
{
    int year;
    char month;
    char day;
};
```

Обращение к элементам структуры производится через точку. Например:

str.name="Иванов";

Определение структуры не создаёт объекта, а определяет новый тип, который может использоваться в будущем для определения объектов.

Пример:

```
#include ...

struct point
{
    int x,y;
};
```

```
double dist(struct point A, struct point B)
{
  dobule d;
  d = sqrt((A.x-B.x)*(A.x-B.x)+(A.y-B.y)*(A.y-B.y));
  return d;
}

void main()
{
  struct point A,B;
  cin >> A.x >> A.y;
  cin >> B.x >> B.y;
  cout << "AB=" << dist (A,B);
}
```

Создание синонимов структур с помощью оператора typedef

```
typedef struct coord point;
```

typedef используется для присвоения нового имени существующему типу данных. Фактически с помощью typedef создаётся синоним типа.

Сложные структуры

Структурный тип данных может включать в себя другие структуры или массивы в качестве элементов.

Например:

```
struct rectangle
{
    struct point topleft;
    struct point bottomright;
} mybox;

mybox.topleft.x=10;
mybox.topleft.y=10;
mybox.bottomright.x=100;
mybox.bottomright.y=120;
```

Инициализация структур

Структуры можно инициализировать при объявлении аналогично массивам.

После объявление структуры ставится знак = и в фигурных скобках записывается список значений, разделённый запятыми

```
struct student
{
    char name[25];
    int score;
    group[8];
} stud1 = {"Ivanov I.I.", 34, "VIS-21"};
```

Структуры и указатели

Указатель, как поле структуры.

Указатели — поля структуры объявляются так же, как и в других местах

```
struct data
{
    int *value;
    int *date;
} first;
```

Инициализирование можно выполнить, присвоив полям структуры адреса подходящих переменных

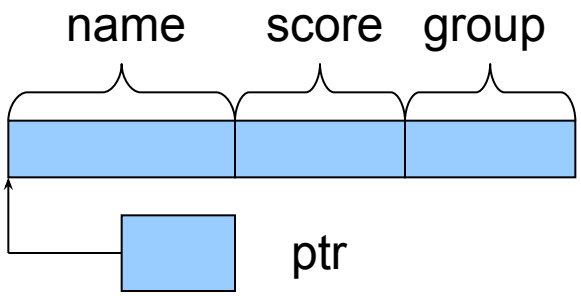
Пусть есть две переменных:

```
int a,b;
first.value = &a;
first.data = &b;
```

Создание указателей на структуры

В языке Си можно объявлять и использовать указатели на структуры точно так же, как и другие типы данных

```
struct student st;
struct student *ptr;
```



ptr = &st;

ptr — указатель на st

*ptr — сама структура st

Обращение к полям: **(*ptr).score = 1000;**

Другой способ доступа к полям, используя указатель — **стрелка**. Так называемое косвенное обращение к элементам структуры.
Вышеуказанный пример будет:
ptr->score = 1000;

Объединения (union)

Внешне похожи на структуры

```
union box
{
    int i;
    char c;
    long l;
    ...
}
```

Всё вышеперечисленное для структур справедливо и для объединений. Но для обращение к полю каждый раз доступен только один элемент. Памяти под структуру выделяется столько, сколько занимает самый большой её элемент.

Потоковый ввод-вывод

Основная концепция ввода-вывода в Си — концепция потоков.
Любая операция по вводу-выводу рассматривается как операция с последовательностью байт. Причём нет никакой разницы откуда поступают эти байты или куда они направлены. При выводе мы направляем некоторые данные в поток, а этот поток в разное время может быть связан с разлиными устройствами.

Двоичные и текстовые режимы доступа к файлам

Для стандартных и системных функций ввода-вывода возможны два режима доступа к файлам — **текстовый** и **двоичный**.
При осуществлении операции ввода-вывода в текстовом режиме
1. При записи информации в файл символ новой строки преобразуется в пару символов **CR** и **LF**. При чтении из файла эта пара символов преобразуется обратно в символ **\n**. В конце файла записывается **EOF (0x1A)**. При считывании информации прочитать находящуюся после **EOF** не удаётся.
При выполнении ввода-вывода в двоичном режиме никакого преобразования символов не происходит и все остальные рассматриваются как не имеющие особого значения символы.

Открытие и закрытие потоков

Функции работы с файлами определены в заголовочном файле **stdio.h**
Указатель на поток
FILE *fp;
Открытие файла
```
fp=fopen(имя_файла, режим_открытия);
```

Пример:

```
fp=fopen("t.txt", "r");
```

При открытии файла он связывается со структурой, определённой в типе FILE. В ней содержатся компоненты, с помощью которых ведётся работа с потоком.
Указатель на буфер, указатель текущей позиции в потоке, флаги состояния файла, размер внутреннего буфера и т.п.
При открытии потока в программу возвращается указатель на поток, являющийся указателем на объект структурного типа FILE.

Режимы открытия:

"w" - Новый текстовый файл открывается для записи. Если файл уже существовал, то старый стирается и создаётся новый.
"r" — Существующий текстовый файл открывается для чтения.
"a" — Текстовый файл открывается для добавления новой информации в конец файла.
"w+" - Новый текстовый файл открывается для записи и последующих многократных исправлений. Стирает файл, если он уже есть и создаёт новый.
"r+" - Существующий файл открывается как для чтения, так и для записи в любое место файла
"a+" - Текстовый файл открывается или создаётся и становится доступным для изменений.
Для использования этих же режимов, но для двоичных файлов, к ним добавляется буква b.

Закрытие файла производится с помощью функции **fclose**

```
int fclose(FILE *stream);
```

Перемотка файла

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

В потоке *stream эта функция смещает указатель чтения записи в начало файла.

Позиционирование указателя чтения-записи

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

Смещение задаётся переменной или выражением типа **long**. Может быть меньше нуля, т.е. Можно "ходить" по файлу в разные стороны. **offset**- смещение, **whence** — начало отсчёта. Начало отсчёта задаётся одной из определённых констант

SEEK_SET(имеет значение 0) – начало файла
SEEK_CUR(имеет значение 1) – текущая позиция
SEEK_END(имеет значение 2) – конец файла

```
fseek(fp, 0L, SEEK_SET); // перемещение к началу файла из произвольной позиции
```

Так мы можем перейти к началу файла из произвольной позиции.

Посимвольный ввод

```
int fgetc(FILE *stream);
```

Функция считывает один символ с текущей позиции потока и перемещает указатель файла на следующий символ. Ошибка, если конец файла - **EOF**.

Посимвольный вывод

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

Ввод строки из файла

```
char *fgets(char *s, int n, FILE *stream);
```

Читает из файла не более **n-1** символов и размещает их по адресу **s**. Символов может быть меньше, чем **n-1**, если встретился **\n** или **EOF**.

Запись строки в файл

```
int fputs(const char *s, FILE *stream);
```

Записывает строку, ограниченную **\0**, в файл и возвращает неотрицательное целое число. При неудаче возвращает **EOF**. Не заменяет ноль-терминатор на перевод новой строки.

Запись данных в поток и чтение из потока

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE
*stream);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE
*stream);
```

В файлах можно хранить потоки данных, состоящих из фиксированного числа последовательных байт. Именно вышеуказанные функции fread и fwrite осуществляют последовательные запись\чтение блоков.

Функция **fread** считывает элементы данных **nmemb** (с размером каждого **size** байтов) с потока, на который указывает **stream**, и сохраняет их в позиции, на которую указывает **ptr**.

Функция **fwrite** записывает элементы данных **nmemb** (с размером каждого **size** байтов) в поток, на который указывает **stream**, при получении элементов с той позиции, на которую указывает **ptr**.

Потоки ввода-вывода

Процесс перемещения данных с внешних носителей в оперативную память называется **вводом данных**, а из памяти на внешние носители – **выводом**.

Поток – последовательность байт данных. Поток байт, принимаемый программой, называется **потоком ввода**, а поток, посылаемый программой на устройство, называется **потоком вывода**. Основное преимущество работы с потоками состоит в том, что потоки являются независимыми от устройства. Программа рассматривает ввод-вывод как работу с непрерывным потоком байт, независимо от их происхождения или места назначения. Каждый поток в С связан с файлом. В этом смысле термин «файл» ассоциируется с различными устройствами и не относится только к файлам, хранящимся на диске.

Файл понимают как промежуточный объект между потоком и реальным физ. устройством, выполняющим ввод-вывод данных. В С не приходится беспокоиться о деталях работы с файлами, т.к. автоматическое взаимодействие между потоками, файлами и устройствами обеспечивают библиотечные функции С и операционная система.

Потоки бывают двух видов: текстовые и двоичные.