МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ им. М.В. ЛОМОНОСОВА
ФИЗИЧЕСКИЙ ФАКУЛЬТЕТ

Антонюк В.А., Задорожный С.С.,
Иванов А.П., Лукашёв А.А.,
Панов Н.А., Шленов С.А.

Язык программирования Си

Учебно-методическое пособие

(I семестр)

Москва, 2022 г.

Антонюк В.А., Задорожный С.С., Иванов А.П., Лукашёв А.А.,
Панов Н.А., Шленов С.А.

Язык программирования Си. Учебно-методическое пособие (I семестр).

М.: Физический факультет МГУ им. М.В. Ломоносова, 2022. — 108 с.
ISBN 978-5-8279-0247-8

Учебно-методическое пособие охватывает все основные разделы учебной программы курса по программированию на языке Си и подготовлено на основе курса, который много лет читается в первом семестре для студентов физического факультета МГУ. Каждая глава курса соответствует теме проводимого семинара и содержит пояснения к учебному материалу, разбираемому на данном семинаре.

Для контроля успеваемости в середине семестра проводится коллоквиум в форме индивидуального опроса всех без исключения студентов по уже прослушанным к этому моменту разделам курса, а в конце семестра проводится зачет.

Рассчитано на студентов младших курсов физико-математических специальностей.

Авторы — сотрудники кафедр математического моделирования и информатики и общей физики и волновых процессов физического факультета МГУ.

Рецензенты: профессор физического ф-та МГУ Голубцов П.В.,
доцент физического ф-та МГУ Митин И.В.

Подписано в печать 9 ноября 2022 г.
Формат 60x90/16. Объём 6,75 п.л.
Тираж 50 экз. Заказ №

Отпечатано в Отделе оперативной печати физического факультета МГУ
Физический факультет МГУ им. М.В. Ломоносова
119991 Москва, ГСП-1, Ленинские горы, д.1, стр. 2

ISBN 978-5-8279-0247-8

Содержание

Тема 1. Средства программирования. Структура консольного приложения, этапы компиляции, сборки и отладки программы. Первая программа. Методические рекомендации преподавателям.

1 Введение

Язык программирования Си — относительно старый язык программирования, он был создан как язык для написания максимально эффективных программ операционной системы Unix в конце 60-х годов XX века. В частности, на этом языке написан почти весь программный код самой операционной системы Unix, а также код современных операционных систем Linux и Microsoft Windows. При этом язык Си продолжает оставаться инструментом для написания высокоэффективных расчетных и научных задач, а по востребованности он уверенно занимает одну из верхних позиций в рейтинге использования различных языков программирования.

Язык Си до сих пор продолжает развиваться: появляются новые конструкции и изменения в синтаксисе языка. Время от времени это фиксируется в очередном официальном стандарте языка. К счастью, эти изменения в большинстве случаев носят характер дополнений и расширений, то есть все предыдущие конструкции языка и особенности синтаксиса сохраняются. И хотя обычно бывает полезно идти в ногу со временем, для написания программ ещё важно, чтобы они были легко переносимы с одного компьютера на другой, где могут отсутствовать свежие компиляторы с языка Си. Поэтому мы будем ориентироваться на некоторый устоявшийся вариант языка и рассмотрим именно основы его синтаксиса. Отчасти такой подход реализован в интегрированной среде программирования Microsoft Visual Studio (VS) 2022, включая ее бесплатную версию Community Edition. Несмотря на то, что эта версия среды программирования, самая свежая на момент написания пособия, допускает компиляцию программ на языке Си стандартов ISO C11, ISO C17 (2018 года), по умолчанию в ней используется более старый вариант. Все примеры, которые приведены ниже, написаны в рамках стандарта языка C99.

Литература:

1. В.В. Подбельский, С.С. Фомин. «Программирование на языке Си» – Москва, «Финансы и статистика», 2005, 600 с.

2. Б. Керниган, Д. Ритчи. «Язык программирования С» – Москва, «Вильямс», 2015, 304 с.

3. С. Прата. «Язык программирования С. Лекции и упражнения», 6-е изд.: Пер. с англ. – М.: Издательский дом «Вильямс», 2015, 928 с.

4. В.А. Антонюк, А.П. Иванов. «Программирование и информатика. Краткий конспект лекций.» – Москва, физический ф-т МГУ, 2015, 64 с.

5. В.А. Антонюк, С.С. Задорожный. «Язык программирования С/С++, часто встречающиеся ошибки при написании программ» – Москва, физический ф-т МГУ, 2021, 59 с.

6. Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. «Алгоритмы: построение и анализ» – Вильямс, 2011, 1296 с.

7. Н.Н.Калиткин, Е.А.Альшина «Численные методы. Книга 1. Численный анализ», Москва, «Академия», 2013, 303 с.

Настоящее пособие, а также пособия 4-5 из списка литературы можно найти в электронном виде в Интернет по адресу: https://cmp.phys.msu.ru/ru/study/programming, а также эти электронные пособия доступны в локальной сети компьютерного практикума на диске Q:.

Подчеркнем, что для глубокого изучения языка настоящего учебно-методического пособия недостаточно, требуется использование полноценного учебника по языку программирования Си (возможно в электронной форме), например, любой из трех первых книг списка литературы выше. При работе в практикуме рекомендуется, чтобы студенты использовали не только настоящее учебное пособие, но и полноценный учебник.

2 Интегрированная среда программирования

С первого же практического занятия студенты должны устойчиво освоить шаги, которые необходимо выполнить для создания программного проекта в интегрированной среде программирования. В этом разделе мы о них подробно расскажем. Версии интерактивной среды программирования могут различаться, но разработчик обычно обеспечивает совместимость пользовательского интерфейса среды с предыдущими версиями, так что, приведенные тут снимки экранов почти точно совпадают как со средой программирования, установленной в практикуме, так и с более современными версиями среды разработки. При понимании необходимой последовательности действий не должно возникнуть проблем и при использовании русифицированной версии Microsoft Visual Studio.

Запуск среды программирования Microsoft Visual Studio выполняется через меню «Пуск» обычным образом, как правило, это не вызывает трудностей у студентов, нужно только показать, как найти среду программирования среди многообразия установленных программ.

Дальше студенты должны выполнить заведение проекта типа «консольное приложение» (Win32 Console Application, т.е. текстовое приложение), нужно пояснить, что без этих действий будет автоматически создан проект для графической программы Microsoft Windows, что выходит за рамки изучаемого материала. Ошибки заведения проекта являются одними из наиболее часто встречаемых у студентов, нужно проследить за тем, чтобы все четко освоили эти действия.

Также нужно пояснить, что в дисплейном классе практикума от одного занятия до другого будут сохраняться только проекты, которые заводятся на диске Z: (домашний диск студента), проекты и любые другие данные и тексты программ пользователя, сохраненные в других местах, могут быть просто стерты до следующего занятия и их придется заводить и набирать заново.

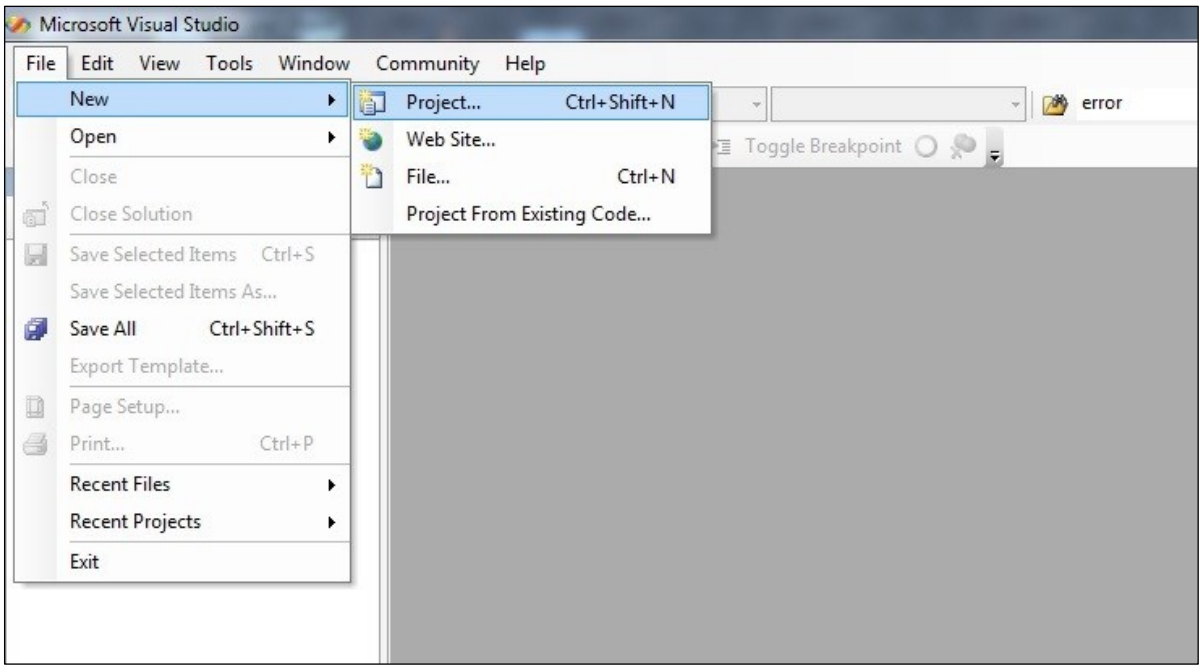
Студенты часто задают вопросы про английский язык среды программирования. Тут важно их успокоить: преподаватель в практикуме всегда поможет перевести непонятные сообщения интерактивной среды программирования, объяснит их смысл, но и сами студенты должны конспектировать и запоминать переводы наиболее частых сообщений об ошибках.

2.1 Пуск → Все программы → Microsoft Visual Studio 2008 → Microsoft Visual Studio 2008

При первом запуске Visual Studio будет предложено выбрать стартовые настройки интегрированной среды программирования. В появившемся окошке выбираем Visual C++ Development Settings, затем нажимаем кнопку Start Visual Studio. Этот запрос возникает не всегда, но если возникнет – нужно правильно на него ответить.

2.2 File → New → Project

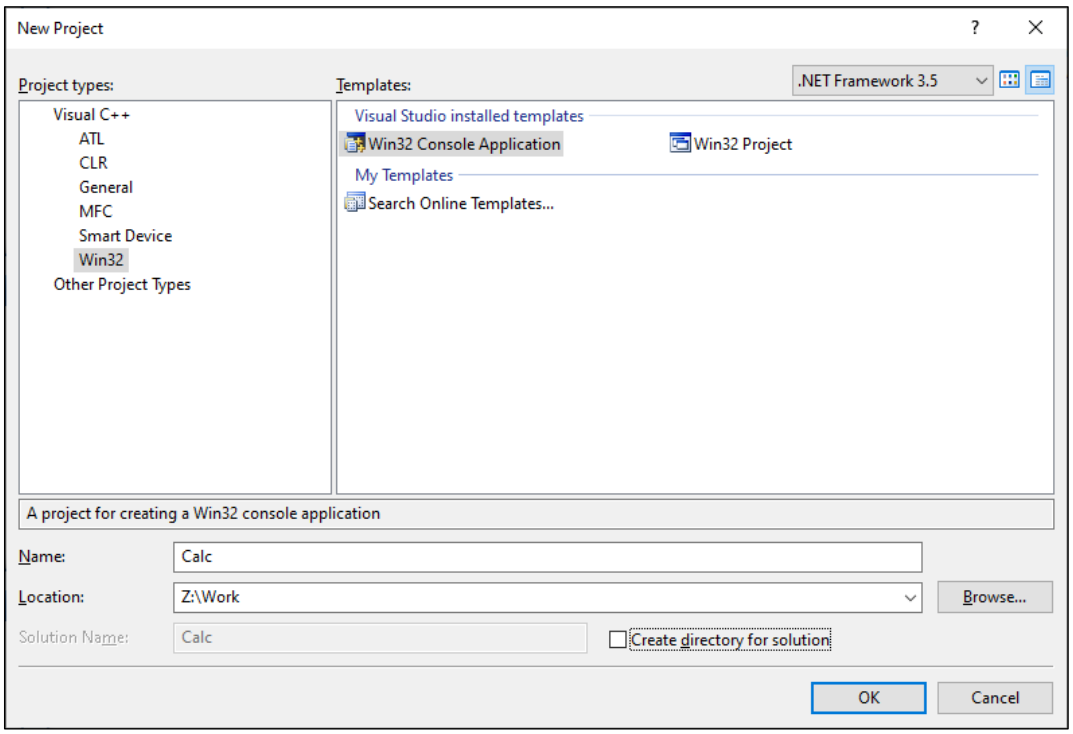
Данную последовательность действий нужно выполнить после запуска интерактивной среды программирования для заведения проекта.



2.3 Win32 → Win32 console application

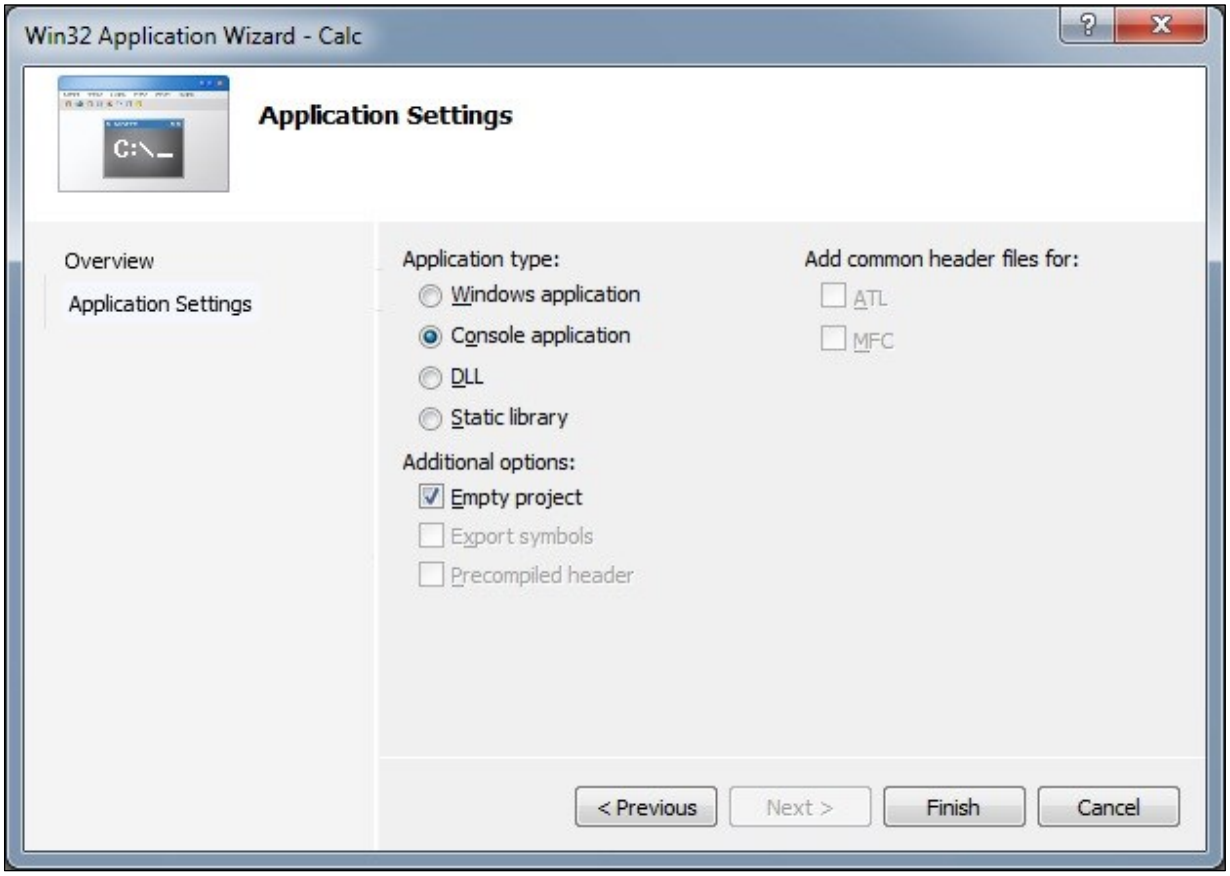
В появившейся карточке выбираем тип проекта «консольное приложение», то есть – приложение текстового режима, ввод данных в которое будет осуществляться с клавиатуры, а вывод – в текстовое окно на экране компьютера.

В этом же окне нужно задать имя проекта (для первой программы – Calc) и место на дисках, где проект будет расположен. Вместо диска C: здесь следует указать домашний диск студента в практикуме (обычно – диск Z:).



2.4 Application settings

Вслед за нажатием кнопки ОК в предыдущем диалоге появится окно настроек свойств будущего проекта. Точнее, будет раскрыто окно с описанием свойств проекта «по умолчанию», но нужно в этом же диалоге переключиться на пункт «Настройки» приложения (Application settings) и установить все органы управления так, как это показано на иллюстрации ниже:

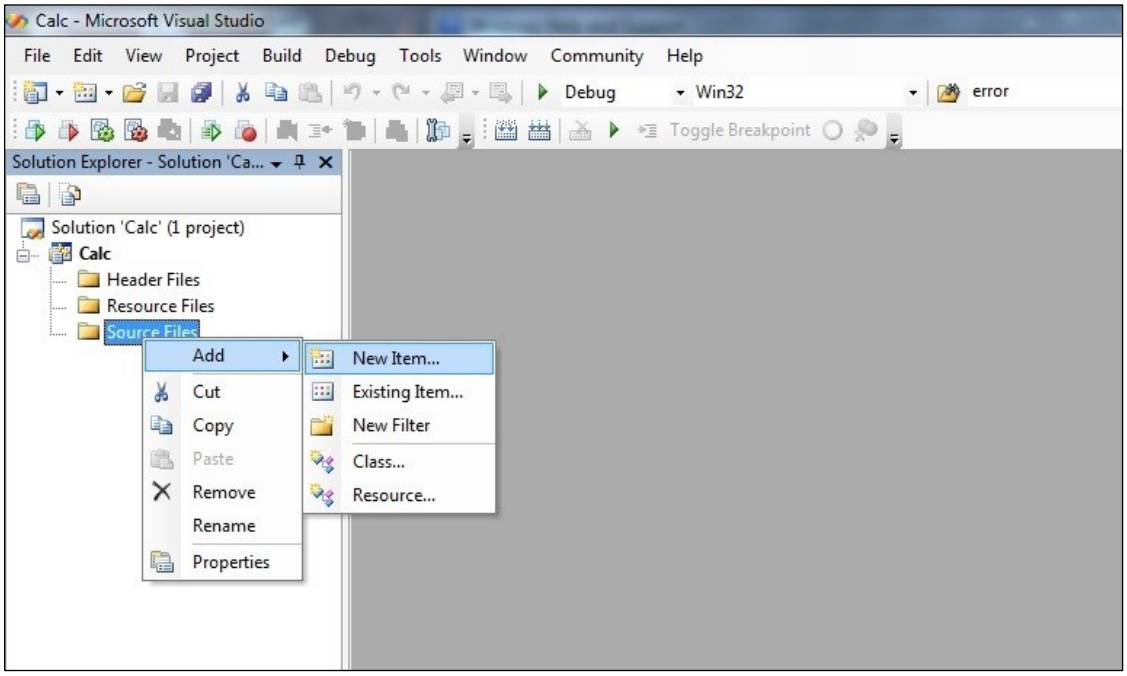


Именно здесь мы задаем тип проекта «консольное приложение» (Console application), а также отменяем использование пред-компилированных заголовочных файлов языка Си (для небольших учебных программ выигрыш в скорости сборки от них невелик, а вот затруднения с перекомпиляцией приложения у студентов возникают часто).

Последняя опция, которую надо задать здесь: «Пустой проект» (Empty project), означает, что проект будет создан пустым, без автоматической генерации шаблона приложения и других сопроводительных файлов. Делается это для того, чтобы студент не отвлекался на файлы, смысл которых ему пока будет непонятен.

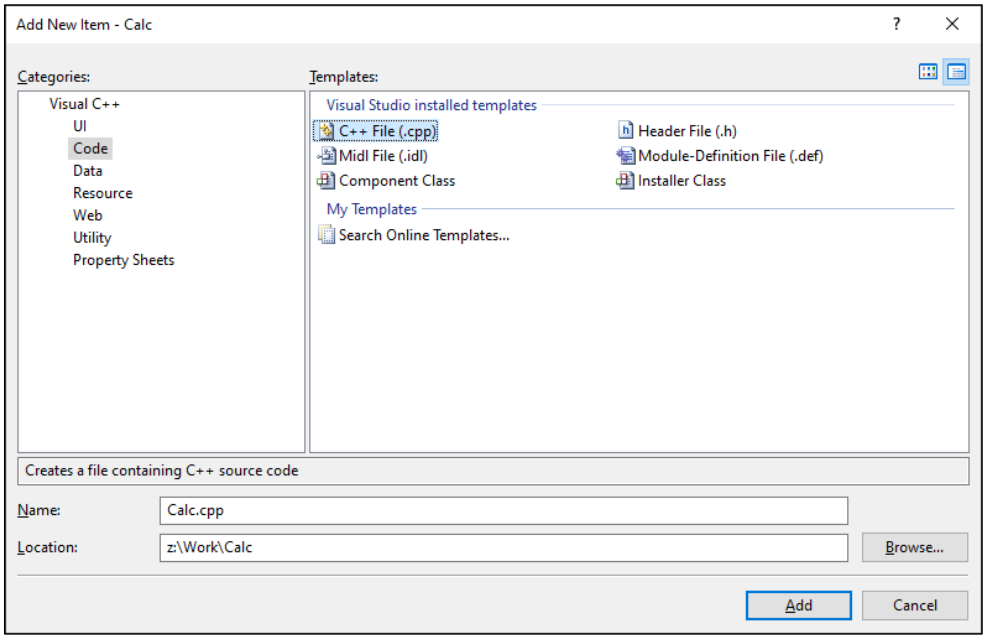
2.5 Project → Add new item → C++ source file (.cpp)

После выполнения предыдущих действий пустой проект будет заведен и в его папку «Исходные тексты» (Source Files) нужно добавить текстовый файл, который будет содержать текст будущей программы. Это можно сделать как через меню Project, так и щелчком правой клавиши мыши по папке Source Files созданного проекта.



В появившемся затем диалоге нужно указать тип добавляемого в проект файла – исходный текст языка C++ (C++ File). Нужно пояснить, что это файл именно языка программирования C++, для которого изучаемый в первом семестре язык программирования Си является почти строгим подмножеством.

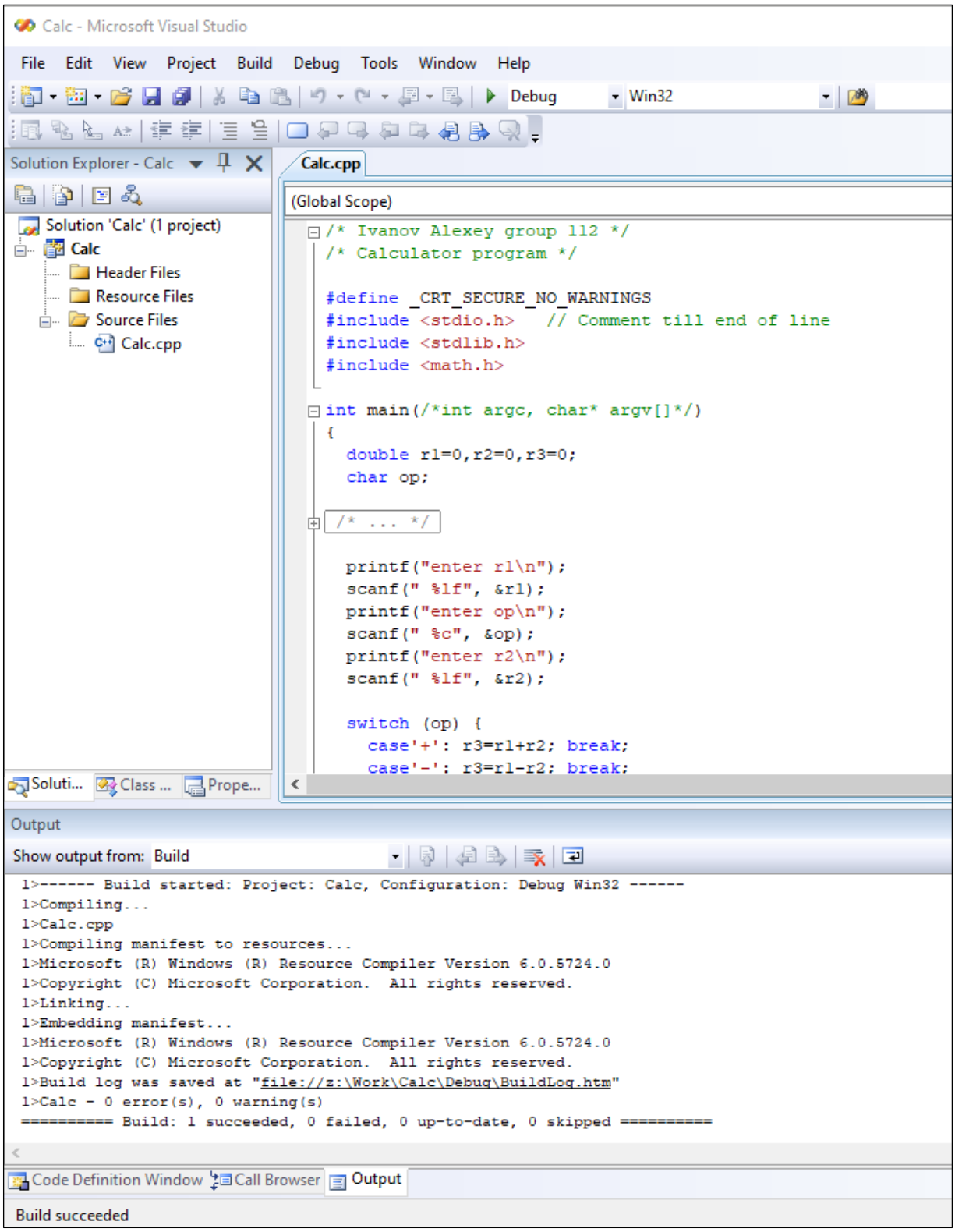
Естественно, нужно указать и имя файла (удобно, чтобы оно совпадало с именем проекта) и его месторасположение на диске компьютера (обычно – в папке, автоматически созданной для проекта при его заведении), расширение `*.cpp` будет подставлено автоматически (но, если нужно, чтобы программа компилировалась строго компилятором языка Си, без расширений языка Си++, то можно явно указать расширение имени файла `*.c`):



Рассмотрение прочих возможных компонентов проекта выходит за рамки курса (они предназначены для графических приложений среды Microsoft Windows), можно только очень кратко описать назначение заголовочных файлов (Header File (.h)), которым будет посвящена тема на одном из следующих семинарских занятий.

2.6 Окончательный вид созданного проекта

На примере данной иллюстрации нужно пояснить основные панели (зоны) интерактивной среды программирования, их назначение и иерархию сущностей проекта (Solution → Project → Source Files).



Панель слева — Solution Explorer (Обозреватель Решения), в ней будут представлены все проекты, входящие в данное решение (Solution), для одиночного проекта одноименный файл решения генерируется автоматически, но для более сложных задач может понадобится целый набор разнотипных компонентов проекта (выполняемые файлы, статические и динамические библиотеки и т.п.), которые будут зависеть друг от друга и должны собираться в строго определенной последовательности.

Иногда студент случайно закрывает панель Solution Explorer, в этом случае её можно заново отобразить выбирая в меню программы пункт View → Solution Explorer (или просто нажав на клавиатуре `Ctrl+Alt+L`).

Правая панель отведена для текстового редактора, в котором происходит написание и исправление текста программы. Изначально этот файл (открываемый двойным щелчком мыши по имени файла в Solution Explorer) пуст, студент самостоятельно пишет в нем текст своей программы.

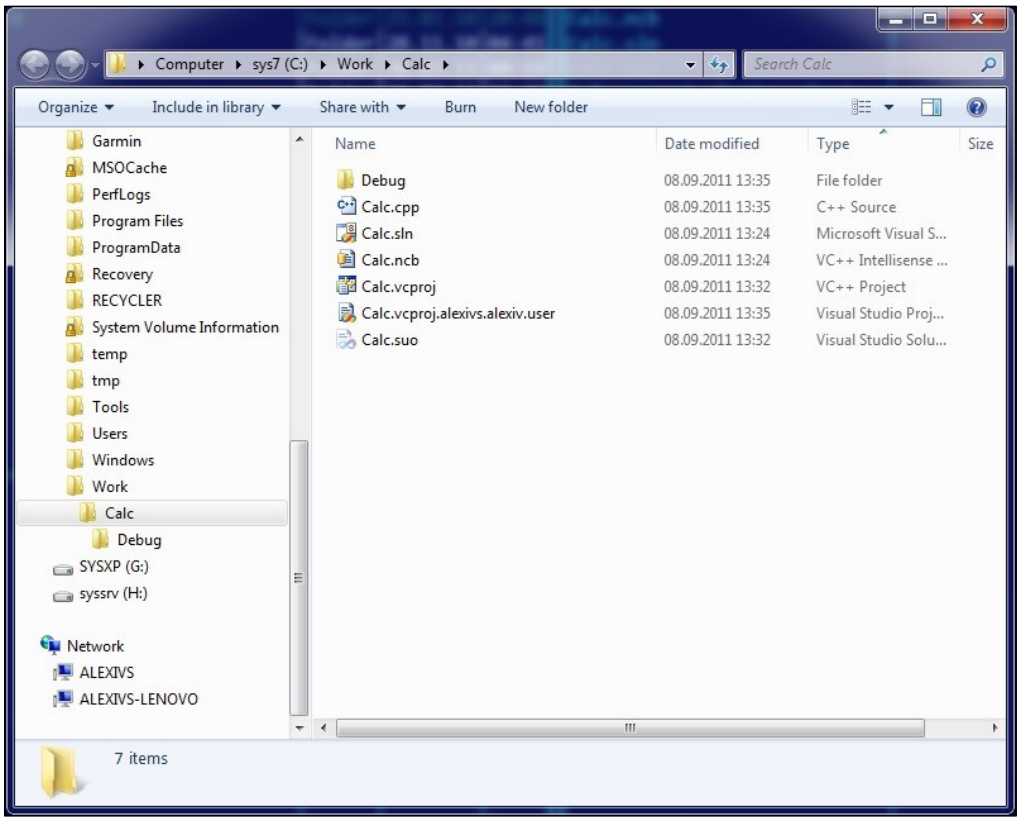
Панель снизу открывается автоматически при запуске сборки проекта (см. ниже). В эту панель выводятся все текстовые сообщения: сообщения об ошибках и об успешном окончании этапов сборки проекта (компиляция, линковка).

Если в этой панели выведено сообщение об ошибке компиляции – то нужно анализировать их сверху вниз, начиная с самой первой, так как последующие ошибки компиляции могут быть наведенными, вызванными предыдущими. Двойной щелчок мышью по строке с сообщением об ошибке позиционирует окно текстового редактора на строку, которая явилась причиной ошибки.

Если подвести текстовый курсор к номеру ошибки (`C2144` на иллюстрации выше) и нажать на клавиатуре клавишу `F1` – то справочная система интерактивной среды выдаст подробное описание данной ошибки и примеры ее возникновения.

2.7 Содержимое папок созданного и собранного проекта на диске

После создания проекта в рабочей папке образуется следующая файловая структура:



То есть, каждое решение (Solution) занимает отдельную папку, в которой расположены все файлы, необходимые для сборки всех проектов, входящих в данное решение:

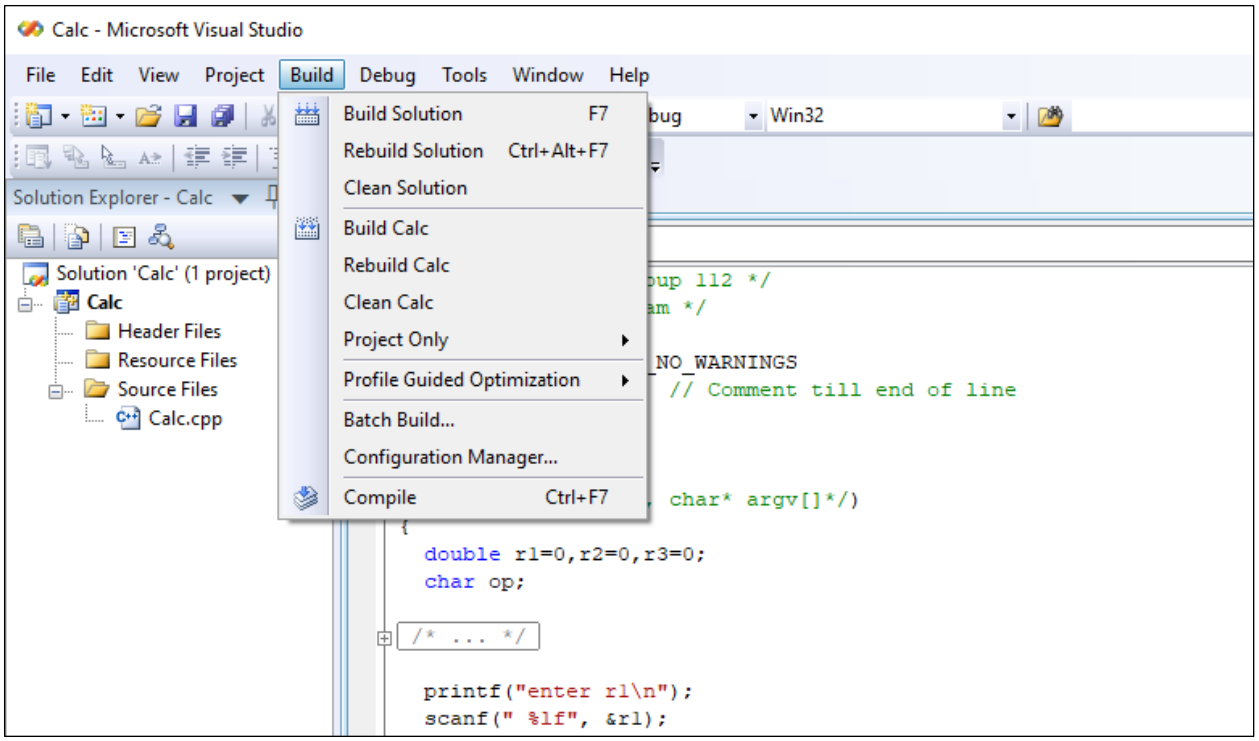
- `Calc.cpp` – исходный текст написанной программы.
- `Calc.sln` – файл, описывающий состав решения (Solution), именно этот файл следует открывать при помощи команды File/Open интегрированной среды программирования.
- `Calc.vcproj` – файл проекта (Project), описывающий состав исходных текстов одной компоненты решения, в данном случае – исполняемой программы. Его можно включить в то или иное решение (и даже в несколько решений сразу), но для простых учебных программ обычно включаться будет единственный проект в единственное соответствующее проекту решение.

Прочие файлы этой папки являются служебными, создаются они средой программирования автоматически, останавливаться на их назначении мы не будем.

Папка `Debug` содержит все результаты компиляции и сборки всех проектов решения, в частности – исполняемый файл нашей программы (`Calc.exe`). Папка `Debug` соответствует отладочному режиму сборки проекта, собираемые в ней компоненты будут содержать отладочную информацию и в них будет отключена часть оптимизаций программного кода, выполняемых компилятором.

Если в среде программирования выбрать режим сборки релизной версии (то есть, не содержащей отладочной информации) программы (это делается переключателем в панели инструментов, расположенной прямо под меню интегрированной среды), то будет создана папка `Release`. Таким образом, компоненты отладочной и окончательной версии никогда не перепутаются, каждая версия программы будет собираться в отдельной папке.

2.8 Сборка проекта (Build)



По готовности исходного текста программы ее можно попробовать собрать. Полный набор команд для этого содержится в меню Build (Сборка), показанном на иллюстрации выше. Здесь же видны сочетания «горячих клавиш», которые можно нажимать для быстрого запуска той или иной команды:

- `Ctrl+F7` или Build → Compile – скомпилировать текущий (открытый в редакторе) файл с исходным текстом программы, выдать ошибки компиляции (если будут).
- `F7` или Build → Build Solution – полностью собрать все решение, включая компиляцию всех исходных текстов и линковку всех проектов. При этом компилироваться и линковаться будут только измененные с последней сборки файлы, это очень экономит время сборки.
- `Ctrl+Alt+F7` или Build → Rebuild Solution – полная пересборка решения: перекомпилируются абсолютно все файлы, составляющие проекты решения. Иногда помогает устранить непонятные ошибки компиляции, вызываемые сбоями сетевого оборудования.

В соседнем меню Debug доступны команды запуска программы после успешной сборки:

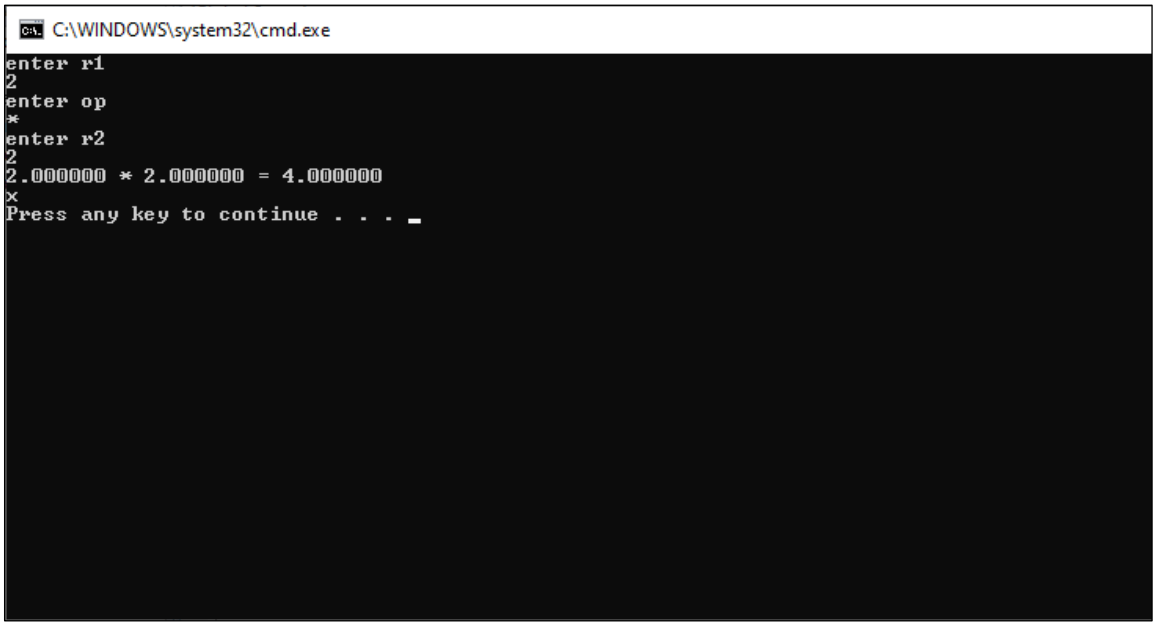
- `Ctrl+F5` или Debug → Start Without Debugging – запустить собранную программу на выполнение без отладчика.
- `F5` или Debug → Start Debugging – запустить собранную программу на выполнение под отладчиком.

Запуск под отладчиком без указания точки остановки приведет к быстрому мельканию и автоматическому закрытию окна с результатами программы, если программа не ожидает ввода от пользователя. При обычном запуске, в конце работы программы, будет выведена надпись, предлагающая нажать любую клавишу для закрытия окна, то есть, пользователь сам сможет решить, когда закрывать окно.

Все эти команды доступны не только через главное меню интегрированной среды, но и из панели инструментов. Наведение указателя мыши на ту или иную кнопку панели инструментов раскрывает подсказку по смыслу данной команды и ее горячей клавише.

2.9 Результат выполнения первой программы

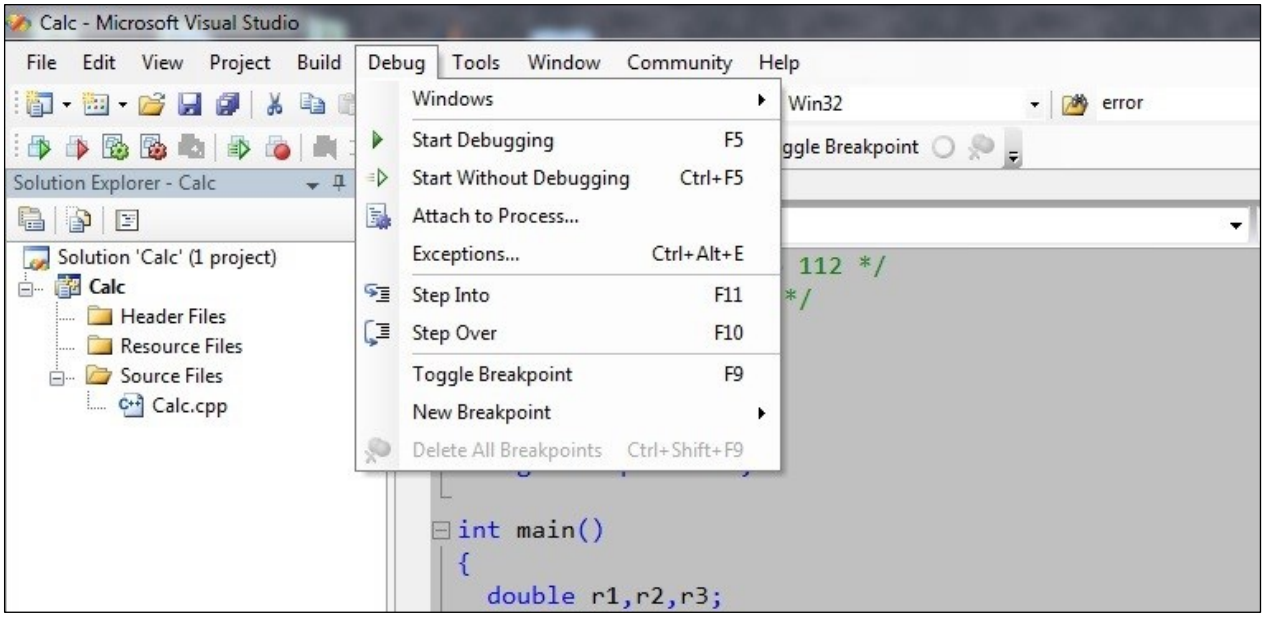
В результате сборки и запуска первой программы (см. п.4) должно появиться примерно такое окно:



Здесь сначала пользователю предлагают ввести тот или иной элемент арифметического выражения (`enter r1`), затем он его печатает на клавиатуре и этот элемент появляется на экране (`2`). В конце программа печатает результат вычисления выражения (`2 * 2 = 4`), после чего ожидается ввод любого символа (x), а далее операционная система печатает свое приглашение нажать любую клавишу, чтобы закрыть окно (`Press any key to continue…`).

2.10 Отладка (Debug)

Интерактивный отладчик помогает найти ошибки в логике работы программы, которые компилятор, в отличие от ошибок в синтаксисе программы, принципиально не может обнаружить.



Команды отладчика:

- `F11` или Debug → Step Into – пошаговое выполнение программы, при этом каждая строчка исходного кода выполняется по нажатию клавиши F11, после чего курсор перемещается к следующей строке, а у программиста появляется возможность просмотреть значения всех переменных программы к этому моменту (например, просто подведя к ним курсор мыши или в нижней панели среды), а при необходимости можно даже исправить эти значения.
- `F10` или Debug → Step Over – то же самое, что предыдущая команда (пошаговый режим), но, в отличие от нее, если в текущей строке есть вызов функции, то не происходит переход пошагового режима в тело этой функции. Функция просто вычисляется полностью, возвращается в точку вызова, после чего полностью выполняется текущая строка и отладчик переходит к следующей строке.
- `Shift+F9` или `Ctrl+Alt+Q` или Debug → Quick Watch – быстрый просмотр значения переменной. Команда в меню появляется только, когда программа запущена под отладчиком! Нужно подвести курсор к интересующей переменной, нажать «горячие клавиши», после чего появится окно, в котором можно будет просмотреть значение переменной.

- `F9` или Debug → Toggle Breakpoint – переключатель, позволяющий поставить или убрать точку остановки отладчика (Breakpoint) на текущей строке. Когда отладчик запущен командой `F5`, он будет быстро выполнять все строки программы до тех пор, пока не дойдет до точки остановки, в которой отладчик остановится. В точке остановки можно просмотреть значения переменных и постараться понять – правильно работает программа или нет.
- `F5` или Debug → Start Debugging – продолжить выполнение программы под отладчиком после точки остановки (до конца программы или до следующей точки остановки, которых может быть сколько угодно).
- `Shift+F5` или Stop Debugging – прервать выполнение программы и ее отладку. Производится возврат к редактированию исходного текста программы.

Не рекомендуется вносить изменения в исходный текст программы во время выполнения ее под отладчиком. Лучше прервать отладку, исправить и перекомпилировать программу, после чего запустить отладку заново.

2.11 Вызов справки

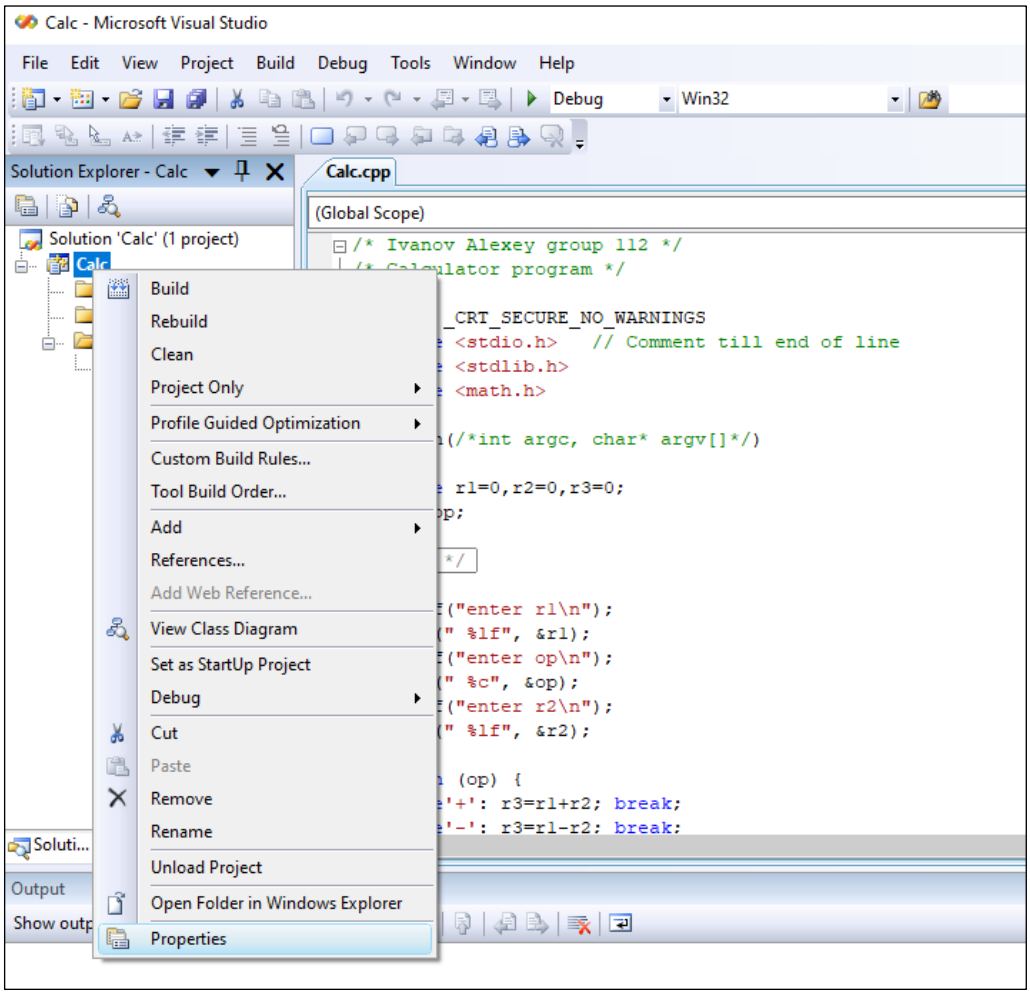
Как уже упоминалось выше, в любой момент можно получить развернутую справку не только по среде программирования и ошибкам компиляции и сборки, но и по любым элементам синтаксиса и ключевым словам языка Си, а также по типу и назначению параметров вызова всех функций стандартных библиотек Си. Для этого нужно подвести курсор (можно кликнуть мышью) к соответствующему элементу или функции прямо в тексте программы и нажать клавишу `F1` на клавиатуре.

После этого система справки может предложить показать справку онлайн (то есть, на сайте разработчика), либо выбрать справку, встроенную в саму среду разработки. В этом случае рекомендуется выбирать не онлайн, а встроенную справку.

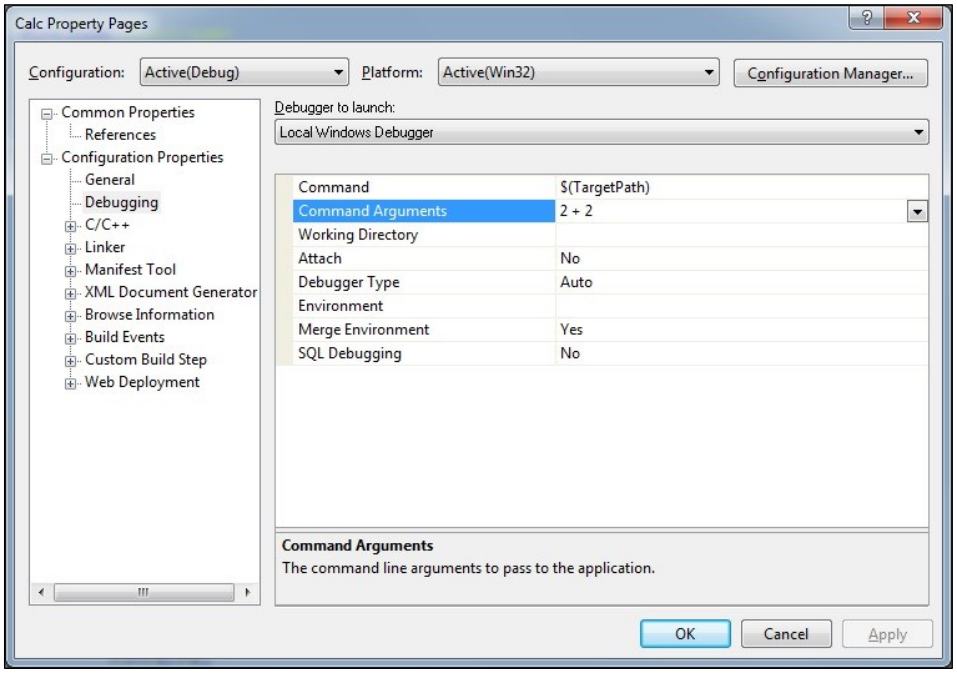
2.12 Задание параметров для командной строки программы в интерактивной среде

Иногда бывает удобно задать необходимые аргументы для выполнения программы прямо в ее командной строке при запуске. В таком случае, аргументы следует указать непосредственно в среде программирования, любой последующий запуск программы (в отладочном или релизном режиме) будет использовать именно указанные аргументы.

Для задания аргументов программы следует щелкнуть правой клавишей мыши по проекту и вызвать редактор свойств проекта:



В диалоге редактирования свойств проекта следует выбрать Configuration Properties → Debugging, а потом, в правой панели задать аргументы командной строки (Command Arguments) с пробелами между отдельными параметрами программы:



3 Первая программа

Первая программа, которую все студенты должны написать в практикуме, представляет собой простейший калькулятор для вводимых пользователем чисел.

```
/* Ivanov Alexey, group 112  - начало программы следует
 * Calculator program        - оформить таким многострочным комментарием
 */

#define _CRT_SECURE_NO_WARNINGS
// Данное макроопределение отключит лишние предупреждения при сборке

#include <stdio.h>              // подключение стандартных библиотек
#include <stdlib.h>             // это были комментарии до конца строки
#include <math.h>

int main(/*int argc, char* argv[] - внутристрочный комментарий*/)
{
  double r1=0, r2=0, r3=0;
  char op;

  printf("enter r1\n"); // вывод на терминал приглашения
  scanf(" %lf", &r1);   // ввод данных с терминала, начальный пробел
                        // в форматирующей строке – важен!
  printf("enter op\n");
  scanf(" %c", &op);
  printf("enter r2\n");
  scanf(" %lf", &r2);

  switch (op) {
    case '+': r3=r1+r2; break;
    case '-': r3=r1-r2; break;
    case '*': r3=r1*r2; break;
    case '/':
        if ( fabs(r2) < 1.0E-10 ) {
            printf("Divide by zero!\n");
            scanf(" %c", &op);
            return 1;
        }
        r3=r1/r2;
        break;
    default:
        printf("Error!\n");
        scanf(" %c", &op);
        return 2;
  }

  printf("%f %c %f = %f\n", r1, op, r2, r3); // вывод результата
  scanf(" %c", &op); // вводим что угодно, чтобы терминал не закрылся
  return 0;
}
```

Небольшая модификация программы позволит пользователю вводить аргументы выражения не с консоли, а непосредственно в командную строку программы:

```
...
int main(int argc, char* argv[])
{
  double r1=0, r2=0, r3=0;
  char op;
  if ( argc-1 != 3 ) { // проверяем, чтобы было ровно три аргумента
      printf("Invalid arguments!\n");
      return 1;
  }
```

```
  r1 = atof(argv[1]); // преобразуем первый аргумент в вещественный тип
  op = argv[2][0];    // запоминаем первый символ второго аргумента
  r2 = atof(argv[3]); // преобразуем третий аргумент в вещественный тип

  switch (op) {
  ..............
```

Здесь функция `atof()` производит преобразование строкового типа (массив символов типа **`char`**) в соответствующее вещественное значение. Для получения целых значений есть аналогичная функция `atoi()`, обе они входят в стандартную библиотеку языка Си, доступ к которой предоставляется подключением заголовочного файла `stdlib.h`.

Запустить такую программу можно будет либо прямо из интегрированной среды, как это было описано в предыдущем разделе, либо из программы Start → Accessories → Command Prompt так:

```
...\Debug> calc.exe 2 + 2
...
```

Пробелы между аргументами в данном случае важны.

Для того, чтобы программа могла выводить в консоль тексты и приглашения на русском языке, рекомендуется вставить до первого оператора вывода вызов следующей системной функции:

```
#include <locale.h>
  ..............
  setlocale(LC_ALL,"rus");
```

К сожалению, для того, чтобы обеспечить ввод строк на русском языке нужно выполнить больше действий, но для учебных программ первого семестра этого обычно не требуется.

4 Об отладке

Программы далеко не всегда работают правильно: компилятор, конечно, найдет все синтаксические ошибки, однако, в успешно скомпилированной программе может скрываться алгоритмическая ошибка, которую не так легко обнаружить и понять, почему программа работает неправильно?

Представим себе, что программист в приведенной выше программе по недосмотру два раза ввел одну и ту же переменную:

```
      printf("enter r1\n");
      scanf(" %lf", &r1);
      printf("enter op\n");
      scanf(" %c", &op);
      printf("enter r2\n");
      scanf(" %lf ", &r1);
```

Такую ошибку легко получить, если для повышения скорости набора программы скопировать строчку первого ввода на место строчки второго ввода, а потом забыть поправить один символ в этой второй строке. К каким результатам приведет такая ошибка?

```
enter r1
25
enter op
/
enter r2
5
Divide by zero!
x
Press any key to close this window . . .
```

Конечно, если программа невелика, как наша первая учебная программа — то ошибку можно будет найти методом «пристального вглядывания», но что делать, если программа длинная и писалась значительное время назад?

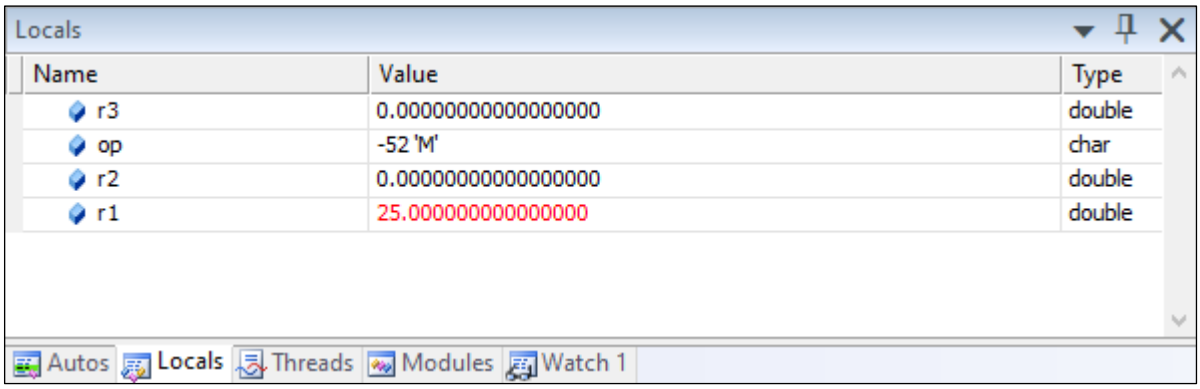
Для решения подобных проблем встроенный отладчик среды разработки является совершенно незаменимым инструментом. Давайте пройдем нашу программу построчно, в пошаговом режиме, для чего закроем окно с загадочным результатом работы программы и нажмем клавишу `F10`, которая выполнит первый «шаг» программы:

```
10  int main(/*int argc, char* argv[] - внутристрочный комментарий*/)
11  {
12    double r1=0,r2=0,r3=0;
13    char op;
14
15  /* ... */
25
26    printf("enter r1\n");
27    scanf(" %lf", &r1);
```

Стрелочка указывает на текущую позицию в программе: самое начало выполнения функции `main()`, мы находимся до первого реального оператора в этой функции. Нажмем клавишу `F10` еще несколько раз, пока не дойдем до 28-й строки нашей программы. Если мы теперь еще раз нажмем эту клавишу – то наша программа перейдет в режим ожидания ввода от пользователя. Введем число 25, после чего обнаружим программу в состоянии «перед выполнением 28-й строки»:

```
26    printf("enter r1\n");
27    scanf(" %lf", &r1);
28    printf("enter op\n");
29    scanf(" %c", &op);
30    printf("enter r2\n");
31    scanf(" %lf", &r1);
```

При этом в нижней части окна среды разработки мы увидим блок текущих значений переменных (при необходимости – переключиться на вкладку Locals):



Мы видим, что переменная `r1` сейчас имеет то самое значение, которое только что ввел пользователь. А вот значение переменной `op` – какое-то странное, что неудивительно: ведь мы не инициализировали эту переменную при объявлении и еще не успели её ввести, так что в ней содержится какое-то значение, случайно находившееся в данной ячейке памяти при запуске программы.

Идем дальше, то есть, продолжаем нажимать клавишу пошагового режима (и вводить требуемые данные) до тех пор, пока не дойдем до строки 33, то есть, не пройдем весь ввод:

```
26    printf("enter r1\n");
27    scanf(" %lf", &r1);
28    printf("enter op\n");
29    scanf(" %c", &op);
30    printf("enter r2\n");
31    scanf(" %lf", &r1);
32
33    switch (op) {
34      case'+': r3=r1+r2; break;
35      case'-': r3=r1-r2; break;
36      case'*': r3=r1*r2; break;
37      case'/':
```

Пусть мы пока не заметили ошибки в тексте программы, смотрим на значения переменных:

Locals

Name	Value	Type
r3	0.00000000000000000	double
op	47'/'	char
r2	0.00000000000000000	double
r1	5.0000000000000000	double

И тут уже сложно не заметить, что изменилось ранее введённое правильное значение переменной `r1`, за которой мы следили, хотя этого не должно было произойти! Переменная `op` имеет правильное значение (знак деления). Кроме того, можно навести курсор мыши на переменную `r2` прямо в тексте программы (либо можно подвести к нужной переменной текстовый курсор и нажать `Shift+F9`):

```
33    switch (op) {
34      case'+': r3=r1+r2; break;
35      case'-': r3=r1-r2; break;
36      case'*': r3=r1*r2; break;
37      case'/':
```

r2 0.00000000000000000

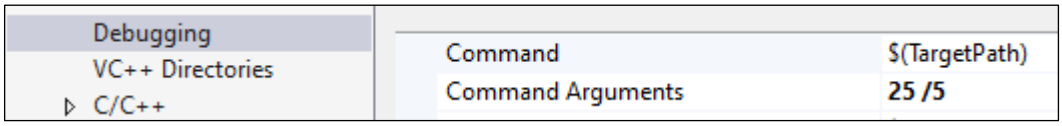
Мы видим, что значение переменной `r2` не изменилось, оно осталось точно таким же, каким оно было с самого начала выполнения нашей программы, то есть нулём. И теперь уже сложно не обратить внимание на предыдущий выполненный оператор, который должен был ввести переменную `r2`, но вместо этого испортил значение `r1` — и, наконец, понять, что же пошло неправильно именно в этой строке! Прерываем отладку (`Shift+F5`), исправляем ошибку, собираем программу и убеждаемся, что теперь она работает правильно.

Другой частой ошибкой времени выполнения является нечаянный выход индекса за границы массива. Рассмотрим вторую версию нашей программы: допустим, мы не позаботились проверить количество аргументов в командной строке:

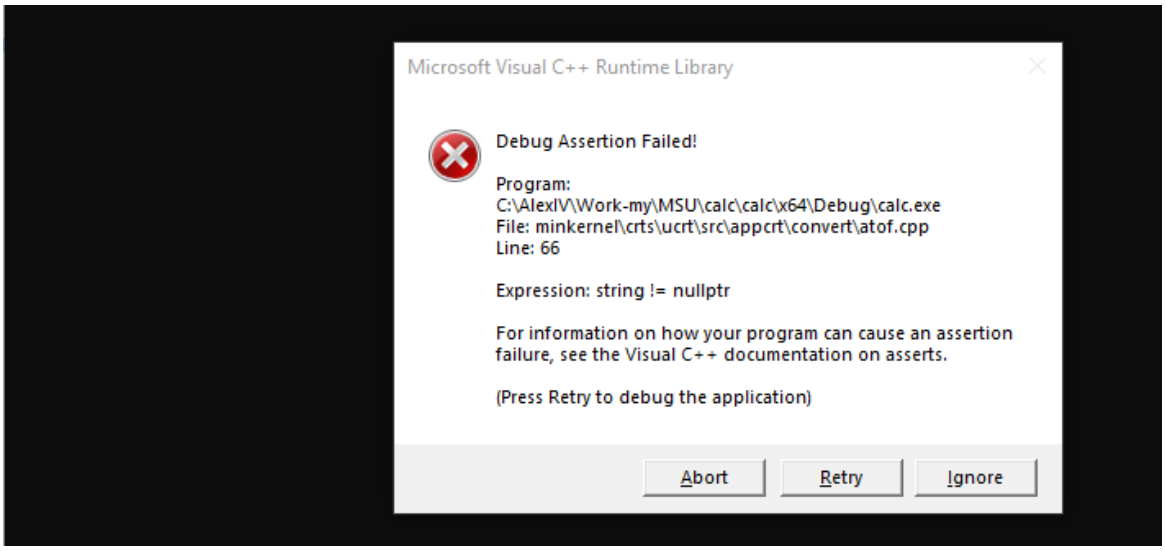
```
int main(int argc, char* argv[])
{
  double r1=0, r2=0, r3=0;
  char op;
  r1 = atof(argv[1]);
  op = argv[2][0];
  r2 = atof(argv[3]);

  // ........
```

А пользователь при вводе тоже ошибся и не разделил пробелами два последних операнда:

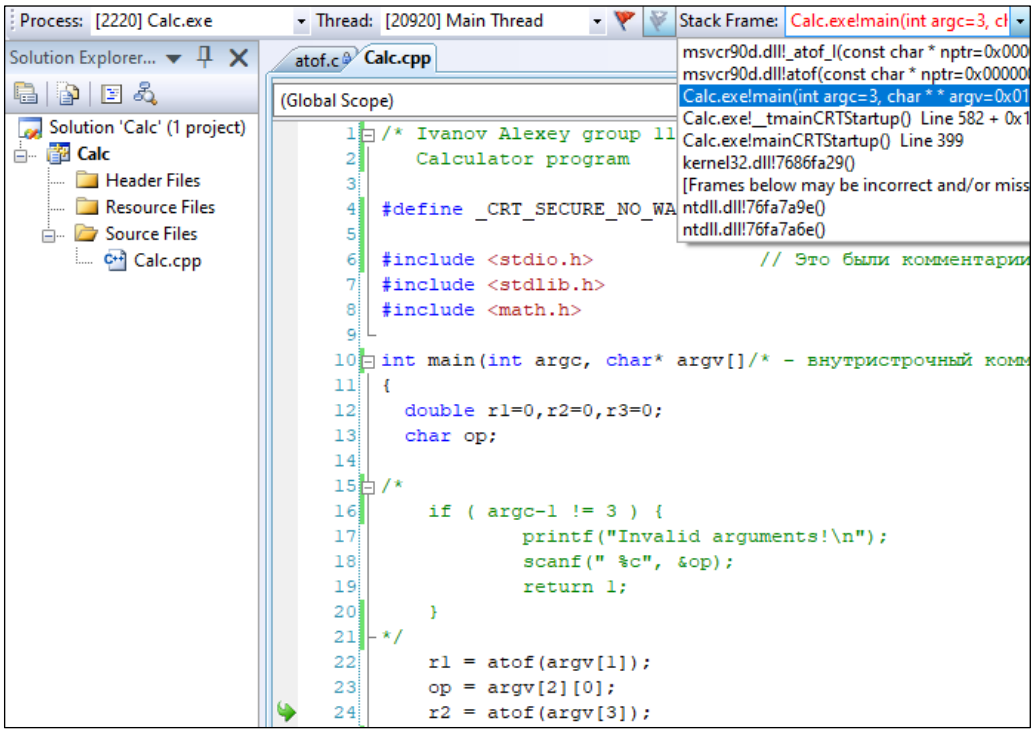


Попытка выполнить нашу программу приведет к такому результату:

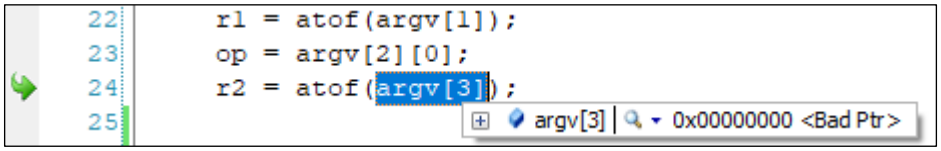


То есть, в возникшей карточке нам сообщат, что произошла какая-то ошибка, но при этом укажут строчку в непонятном файле системной библиотеки. Закроем эту карточку и консольное окно и в среде разработки нажмем `F5` (то есть, запустим программу в режиме отладки, но без остановки на первой строке программы). Теперь в возникшей той же самой карточке с сообщением об ошибке можно нажать `Retry` — после чего среда разработки откроет нам файл системной библиотеки и укажет строчку, где произошла фатальная ошибка.

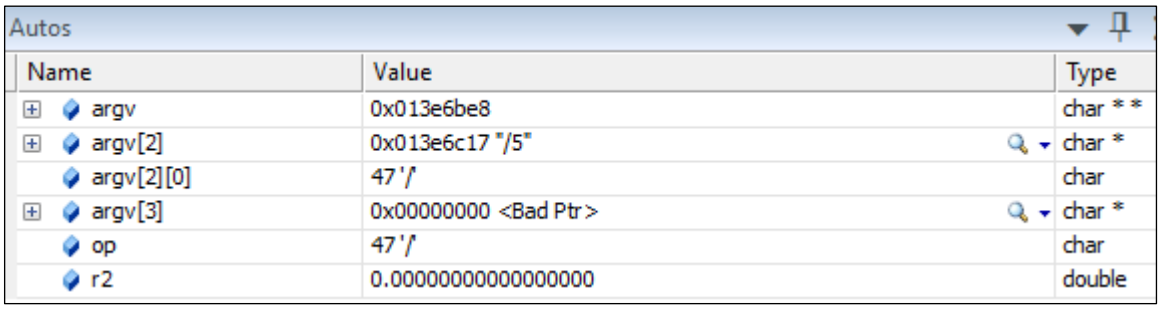
Для того, чтобы переключиться на нужную строку своей программы, необходимо в верхней панели инструментов отладчика выбрать нужный «Stack frame», то есть, нужную нам нашу собственную функцию `main()`:



Теперь мы видим, что «падение» (то есть, аварийное завершение) программы произошло при выполнении 24-й строки нашего кода, при попытке преобразовать третий операнд программы в вещественное число. Выделим мышью аргумент функции в этой строке:



Мы видим, что этот аргумент пустой, в нем нет нужного нам значения. Посмотрим, что лежит во втором операнде (можно посмотреть все сразу в нижней панели):



И тут мы видим, что помимо пустого третьего операнда во втором операнде знак деления «склеился» с третьим операндом, в результате чего третий операнд и остался пустым.

Останавливаем отладку (`Shift+F5`), исправляем ввод в командной строке и, чтобы подобного не повторилось в будущем – вставляем в самое начало программы код проверки правильности введенных аргументов:

```
    if ( argc-1 != 3 ) {
        printf("Invalid arguments!\n");
        return 1;
    }
```

Теперь, даже если пользователь ошибется снова, он получит не «падение» программы с непонятным системным сообщением об ошибке, а нашу собственную диагностику — в чём именно он ошибся при вводе аргументов.

Здесь приведены самые базовые способы поиска ошибок отладчиком, в последующих темах, по мере изучения циклов и функций могут понадобиться и другие, более продвинутые инструменты отладки, описанные выше в разделе 2.10:

- `F11` или Debug → Step Into – пошаговое выполнение с «заходом» внутрь вызываемой функции.
- `F9` или Debug → Toggle Breakpoint – переключатель, позволяющий поставить или убрать точку остановки отладчика (Breakpoint) на текущей строке.

Тема 2. Основы синтаксиса языка Си. Базовые типы данных. Запись констант и определение переменных. Области видимости. Приведение типов. Арифметические операторы. Условные операторы, циклы.

1 Базовые типы данных

Язык Си относится к статически типизированным языкам программирования. В нем переменная связывается с типом в момент её объявления и далее в программе этот тип уже не может быть изменен. Такой подход удобен для компилируемых языков, когда перед выполнением программы создается исполняемый файл. В результате многие ошибки в программе могут быть обнаружены уже на этапе компиляции. Компилятору тип переменной важен как с точки зрения количества байтов памяти, отводимых под переменную, так и их интерпретации. В момент объявления переменной её тип определяется с помощью соответствующих ключевых слов. Основные или базовые типы и их длина в байтах представлены в Таблице 2.1.

Таблица 2.1 Базовые типы данных в Си

Целые со знаком, длина в VS		Примечание
int	4 байта	Длина зависит от компилятора, но обязательно: **sizeof**(**short**) ≤ **sizeof**(**int**) ≤ **sizeof**(**long**).
short	2 байта	Беззнаковые целые типы образуются из знаковых добавлением ключевого слова **unsigned**: **unsigned int**, **unsigned short**, **unsigned long**, **unsigned char**. Вместо **unsigned int** можно использовать сокращенную форму — просто **unsigned**, а для **short** и **long** – «удлиненную» форму: **short int** и **long int**.
long	4 байта	
char	1 байт	
Вещественные с плавающей точкой		Тип **double** определяет вещественные числа с двойной точностью. По сравнению с типом **float** здесь увеличено число разрядов после плавающей точки и диапазон порядка.
float	4 байта	
double	8 байт	

2 Запись констант

Константы, так же, как и переменные и любые другие выражения, имеют тип. Целочисленные константы могут быть записаны не только в десятичной системе счисления, но и в восьмеричной и шестнадцатеричной. Признаком восьмеричной константы является лидирующий ноль, а при записи шестнадцатеричной константы после лидирующего нуля добавляется латинская буква «x» в любом регистре. Несколько примеров приведено в Таблице 2.2.

Таблица 2.2 Примеры записи констант

Целочисленные константы:

Обычная	Си	Обычная по основанию 8 и 16	Си
138	138 138L	118 = 910	011
-367	-367	1F16 = 3110	0x1F

Вещественные константы:

Обычная	Си	Обычная с плавающей запятой	Си
1,345	1.345	$178 \cdot 10^{-3}$	178.e-3 или 178E-3
-0,15	-0.15 или -.15	$0{,}367 \cdot 10^{5}$	0.367e5 или 0.367E5

По умолчанию вещественные константы в Си имеют тип **`double`**. При необходимости записать вещественную константу с одинарной точностью надо при её записи в конце добавить суффикс «`F`», например, `1.345F`.

Вариантом целочисленных констант являются символьные константы, в которых символ записывается в одинарных кавычках, например, `'A'`. Ту же самую символьную константу `'A'` можно записать, используя её целочисленный код в восьмеричной системе счисления, который в таком случае записывается после обратной косой черты: `'\101'`. Лидирующий ноль в этом случае не используется. Таким образом можно записать специальные символы, не представленные на клавиатуре. Например, `'\7'` – звонок, который прозвучит из динамика компьютера (при его наличии).

3 Объявление переменной. Область видимости переменных.

До использования в программе переменная должна быть объявлена. Объявление переменной содержит тип этой переменной и её имя, которое является комбинацией символов латинского алфавита, цифр и символа подчеркивания. При этом с цифры имя начинаться не может, как и совпадать с одним из ключевых слов. Так как язык Си является регистрозависимым, то переменные «`A`» и «`a`» — это разные переменные. Несколько переменных одного типа могут быть объявлены через запятую. Объявление переменной можно совместить с её инициализацией. Ниже приведены примеры объявлений переменных:

```
int a;
float b;
int c, d, e;              // объявление нескольких переменных одного типа
double g = 5.0, q = 7.1; // объявление с инициализацией
```

Переменная может быть объявлена как внутри функции — такая переменная называется локальной, так и вне функций — это глобальная переменная. Глобальная переменная после её объявления видна, то есть может быть использована в любой функции, следующей за её объявлением. Локальную переменную необходимо объявить в начале блока в фигурных скобках до первого исполняемого оператора. Начиная со стандартов языка C99 допускается объявление локальной переменной в любом месте функции. Область видимости этой переменной начинается с позиции её объявления и до конца текущего оператора либо блока операторов, ограниченного фигурными скобками, то есть в любом случае не выходит за пределы самой функции.

Локальная переменная может иметь такое же имя, как и глобальная переменная. Потенциальный конфликт имён в этом случае разрешается в пользу локальной переменной — в области своей видимости она перекрывает глобальную переменную с тем же именем, как в следующем примере.

```
#include <stdio.h>
int i = 10;         // глобальная переменная i

int main() {
  int i = 1;        // локальная переменная i
  printf("%d", i); // выводится значение 1 локальной переменной i
  return 0;
}
```

По умолчанию глобальная переменная инициализируется нулем. Локальную переменную перед использованием следует инициализировать в явном виде, иначе она будет содержать произвольное случайное значение.

4 Приведение типов

При присваивании переменной одного типа значения переменной или выражения другого типа возникает ситуация несовпадения типов, которая разрешается путем их приведения, которое может быть как явным, так и неявным. В приведенном ниже фрагменте программы переменной типа **float** присваивается значение переменной типа **double**:

```
float f;
double df = 1.0;
f = df;  // неявное приведение типов
```

Такое несовпадение типов может повлечь потерю точности и, в зависимости от настроек, компилятор может сгенерировать предупреждение (warning). Чтобы его не было, можно показать компилятору, что вы, как программист, контролируете ситуацию, записав явное преобразование типа:

```
f = (float) df; // явное приведение типов
```

В круглые скобки перед переменной или выражением, тип которого надо преобразовать, помещается имя нового типа.

5 Арифметические операторы и выражения

Оператор присваивания мы уже использовали в приведенных выше примерах. Изначально в языке Си он выглядит предельно просто и органично: «=». Допускается использование цепочки операторов присваивания, например:

```
i = m = k = 10;
```

Такая запись эквивалентна последовательности выражений:

```
k = 10;
m = k;
i = m;
```

Другие арифметические операторы с примерами использования приведены в Таблице 2.3.

Таблица 2.3 Основные арифметические операторы

Оператор	Пример	Оператор	Пример
Сложение +	`i = i + 1;`	Деление /	`x = 4.0 / 3.0;`
Вычитание –	`i = 5 - 2;`	Деление по модулю %	`i = 10 % 7;  /* i равно 3 */` `/*(остаток от деления на 7)*/`
Умножение *	`x = 2.4 * j;`	Изменение знака (унарный минус) –	`i = -j;`

Порядок выполнения операторов в одном выражении зависит от их приоритета. Более высокий приоритет имеет унарный минус, затем следует группа операторов *, /, % и далее операторы +, –. В рамках одного приоритета операторы выполняются слева направо.

Порядок, установленный приоритетами, можно изменить с помощью круглых скобок:

```
j = 1;
i = j+1 * j+2;      // значение i равно 4
i = (j+1) * (j+2); // значение i равно 6
```

Тип результата арифметического выражения зависит от типа операндов. Например, если делимое и делитель имеют тип **int**, то и результат деления также будет иметь тип **int**. Рассмотрим следующий пример:

```
double a;
a = 3/4;
```

Здесь выполняется деление целого на целое. Результат в этом случае также имеет целый тип, поэтому дробная часть будет отброшена и переменная a, несмотря даже на то, что она имеет вещественный тип **double**, получит значение 0. Это удобно, если нужно найти целое от деления. Если важно сохранить дробную часть, то надо в выражении справа от оператора присваивания заменить тип хотя бы одного операнда на вещественный. Для этого достаточно одну из констант записать с десятичной точкой:

```
double a;
a = 3./4;
```

Теперь тип арифметического выражения будет вещественным, т.к. если в одном выражении встречаются переменные разных типов, компилятор пытается привести их к «старшему» типу в выражении. «Младший» тип **char** приводится к типу **int**, тип **int** приводится к вещественным типам. В результате переменная a получит значение 0.75.

Альтернативой постановки десятичной точки служит явное приведение типа одного из операндов. Такой способ подходит и в случае, если операндами являются переменные:

```
a = (double)3/4;
```

В языке Си есть много конструкций, которые отчасти дублируют уже имеющиеся средства написания программы, но они позволяют писать более компактный код. К этим конструкциям, в частности, относятся операторы увеличения ++ и уменьшения -- на единицу значения переменной. Они могут стоять как после имени переменной (постфиксная запись), так и перед (префиксная запись):

```
int i = 1;
i++;
++i;
```

И в том, и в другом случае значение переменной i увеличивается на 1, т.е. каждая из последних строк фрагмента программы эквивалентна следующему выражению:

```
i = i + 1;
```

Разница в использовании постфиксной и префиксной форм этих операторов будет заметна в более сложных выражениях, содержащих несколько операторов, например, в следующем фрагменте программы:

```
int ii, i = 1;
ii = i--;
```

переменная i уменьшается на 1 и принимает значение 0, но переменной ii присваивается её старое значение, т.е. 1. При префиксной записи оператора

```
ii = --i;
```

переменной ii присваивается новое значение i, т.е. 0. Таким образом, при постфиксной записи в выражении используется старое значение переменной, при префиксной — новое.

Целая серия вариантов оператора присваивания предназначены для более краткой записи выражений, в которых слева и справа от оператора присваивания стоит одна и та же переменная. Например, пусть есть арифметическое выражение, в котором к старому значению переменной i

прибавляется число и новое значение сохраняется в той же переменной. В примере ниже слева стоит классическая запись этого выражения, а справа — краткий вариант того же самого:

```
i = i + 5;     i += 5;
```

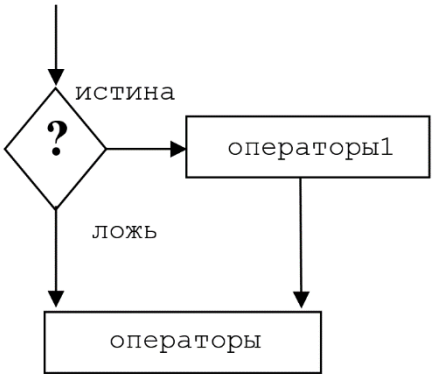
Аналогичные варианты оператора присваивания существуют и в комбинации с другими арифметическими операторами, например:

```
i = i*10;      i *= 10;
i = i%7;       i %= 7;
```

и, как мы увидим позже, не только с арифметическими операторами.

6 Условные операторы

Почти любая программа предполагает возможность выполнения или не выполнения некоторых операторов в зависимости от конкретного условия. На рис. 2.1 приведена блок-схема такого фрагмента программы. Если условие (в ромбе) истинно, то будут выполнены *операторы*1. В противном случае программа сразу перейдет к выполнению *операторов*. В Си такая конструкция реализуется с помощью условного оператора **if**.



if (условное выражение) *оператор1;*
операторы;

Пример:

```
if (i == 10 && t > 0)
{
    j = 1;
    k = i;
}
i = 11;
```

Рис. 2.1 Блок-схема фрагмента программы (слева), схема её реализации с использованием условного оператора **if** и пример (справа).

Если условие, заключаемое в круглые скобки, истинно, то будет выполнен *оператор*1. Если нужно выполнить несколько операторов вместо *оператор*1, то они заключаются в фигурные скобки (как в примере) и трактуются как один составной *оператор*1.

Проверяемое условие в общем случае представляет собой логическое выражение, которое может содержать либо одно простое логическое отношение (сравнение), либо несколько отношений, объединенных с помощью логических операторов. В примере два логических отношения «==» (сравнение на предмет равенства) и «>» (больше) объединены логическим оператором «И» («&&»). В результате операторы в фигурных скобках будут выполнены только, если значение переменной `i` равно 10 и одновременно значение переменной `t` больше 0. В Таблице 2.4 приведён синтаксис логических отношений и операторов в языке Си.

Таблица 2.4 Логические отношения и операторы

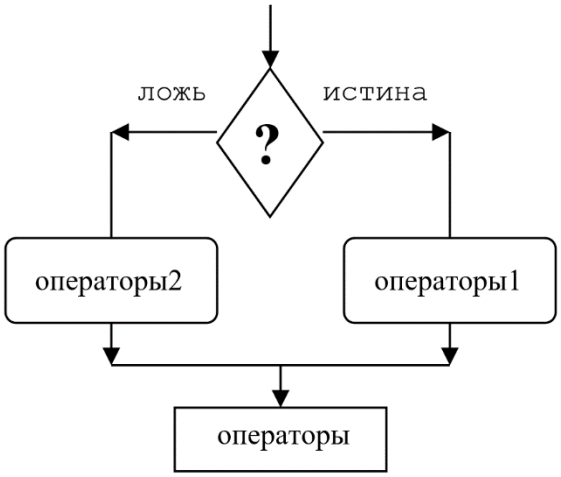
Отношение	Пояснение	Оператор	Пояснение
==	равно	&&	Логическое «И»
>, >=	больше, больше или равно	||	Логическое «ИЛИ»
<, <=	меньше, меньше или равно	!	Логическое «НЕ»
!=	не равно		

В языке Си булевский (логический) тип **bool** появился только начиная со стандарта C99, и его использование требует включение заголовочного файла <stdbool.h>.

Поэтому отдельно встаёт вопрос, какое значение имеют истина и ложь? Или другими словами, что в языке понимается под истиной, и что под ложью? Истина в языке Си трактуется очень широко – это всё, что не ноль. Соответственно, ноль – это ложь. В результате, если логическое выражение истинно, то оно получает значение 1, если нет, то 0. Например, в следующем выражении переменной i будет присвоено значение 1:

```
i = 2 > 0;
```

Рассмотрим ещё одно часто встречающееся в программах ветвление, когда при истинности некоторого условия выполняются *операторы*1, в противном случае – *операторы*2, и далее идут общие *операторы*. Такую блок-схему можно реализовать с помощью нескольких операторов **if**, но удобнее использовать его разновидность – оператор **if … else** (рис. 2.2).



if (условное выражение) *оператор1;*
else *оператор2;*
операторы;

Пример:

```
if (i == 10)
{
      j = 1;
      k = i;
}
else j = 10;

i = 11;
```

Рис. 2.2 Блок-схема фрагмента программы (слева), схема её реализации с использованием условного оператора **if … else** и пример (справа).

Так как в ветви **else** только одно выражение (`j = 10;`), то его не потребовалось заключать в фигурные скобки (а можно было бы!).

Упомянем ещё один, тернарный условный оператор «?:». В общем виде его синтаксис имеет вид:

```
Имя_переменной = Условное_выражение ? выражение1 : выражение2;
```

Если условное выражение истинно, то вычисляется *выражение1*, в противном случае – *выражение2*:

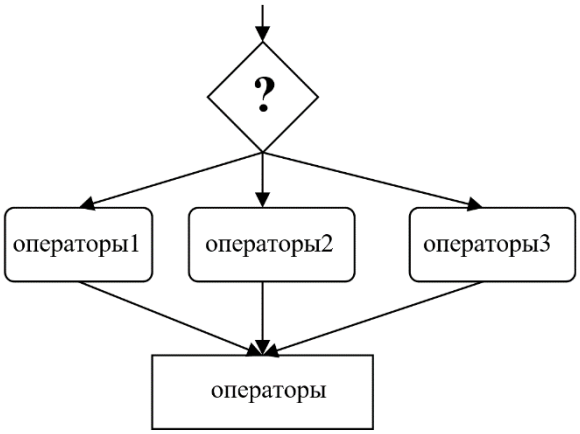
```
a = 3;
x = a > 2 ? ++a : --a;
```

В этом примере значение переменной a увеличится на 1, т.к. условное выражение `a > 2` истинно. Так как оператор инкремента – префиксный, то и x присвоится значение 4.

7 Множественный выбор

Иногда встает задача выбора одного из множества вариантов продолжения программы в зависимости от значения некоторого арифметического выражения. Это можно реализовать с помощью нескольких условных операторов, но удобнее воспользоваться оператором **switch** (рис. 2.3). Конструкция этого оператора включает заголовок в круглых скобках и тело в фигурных скобках, которое размечается с помощью ключевого слова **case** с различными целочисленными константами. В начале выполнения оператора **switch** вычисляется

арифметическое выражение в его заголовке и далее происходит переход на метку **case** с тем же значением. Если значение заголовка не совпадает ни с одним из значений меток **case**, то либо происходит переход на метку **default**, либо, если такой метки нет, ни один из операторов тела **switch** не выполняется и переход происходит на следующий за **switch** оператор.




```
switch (выражение целого типа)
{
    case Значение1:
        операторы1;
        break;
    case Значение2:
        операторы2;
        break;
    default:
        операторы3;
}
```

Рис. 2.3 Блок-схема фрагмента программы с множественным выбором (слева), и схема её реализации с использованием оператора **switch** (справа).

Обратите внимание, что каждый **case** на схеме завершается оператором **break**, который завершает выполнение оператора **switch**. Его использование не является обязательным, но именно он позволяет реализовать блок-схему слева. В противном случае после перехода на одну из меток **case** далее продолжилось бы выполнение всех оставшихся до закрывающей фигурной скобки операторов. Иными словами, вход в тело оператора **switch** происходит по месту расположения соответствующей метки **case**, а выход либо с помощью оператора **break**, либо после выполнения последнего оператора тела. Ниже приведён пример использования оператора **switch**:

```
int a, i = 2;

switch (i)
{
 case 1: a = 10;
         break;
 case 2: a = 20;
         break;
 default: a = 30;
}
```

8 Оператор перехода *goto*

Почти не используемый в языке Си оператор **goto** позволяет из текущего места программы сразу перейти к помеченному меткой оператору. Имя метки задаётся по тем же правилам, что и имя переменной; завершается метка двоеточием, например:

```
goto START; // имя метки часто набирается заглавными буквами
...
START:  a = 1;
```

Используя оператор перехода и условный оператор, можно организовать многократное повторение некоторого фрагмента программы. Но лучше это делать с помощью специальных конструкций цикла.

9 Циклы

В языке Си имеется несколько операторов для организации в программе циклов.

9.1 Цикл `while`

Цикл **`while`** имеет следующий синтаксис:

```
while (условное выражение) оператор;
```

Если внутри цикла нужно выполнить несколько операторов, то они заключаются в фигурные скобки и с точки зрения синтаксиса трактуются как один составной оператор. При входе в цикл проверяется условное выражение в круглых скобках и, если оно истинно, то выполняются операторы тела цикла. В противном случае тело цикла не выполняется, а происходит переход к следующему за циклом оператору. После завершения выполнения операторов тела цикла этот процесс повторяется заново в виде очередной итерации до тех пор, пока проверяемое условие остается истинным.

Пример: суммирование целых чисел от `1` до `total`.

```
double sum, total = 10.;
int i;

sum = 0;                // инициализация переменной sum
i = 1;                  // и счетчика цикла i
while (i <= total)      // заголовок цикла с проверяемым условием
{
      sum = sum + i;    // тело цикла, рассмотрите вариант: sum += i++;
      i = i + 1;
}
```

В цикле **`while`** часть операторов, имеющих прямое отношение к организации цикла, приходится записывать вне оператора **`while`**, нарушая тем самым единое восприятие всей конструкции. В цикле **`for`** этот недостаток устранен.

9.2 Цикл `for`

В отличие от цикла **`while`** в этом операторе расширен заголовок. Теперь он включает 3 позиции, разделяемые точкой с запятой (их наличие обязательно):

```
for (init-выражение ; условное выражение ; loop-выражение ) оператор;
```

Смысл условного выражения на второй позиции такой же, как и в цикле **`while`**, в частности, здесь можно использовать составное условие, где несколько отношений связаны логическими операторами «И», «ИЛИ», «НЕ». Операторы, на первой позиции (*init-выражение*) будут выполнены один раз до первой проверки условия. Если на этой позиции нужно разместить несколько операторов, то они отделяются друг от друга запятой (не точкой с запятой!). Операторы на третьей позиции выполняются каждый раз по окончанию очередной итерации цикла перед следующей проверкой условия. Здесь так же можно использовать запятую, если операторов несколько. На самом деле, эта запятая так же является оператором языка Си и его (её) можно использовать и без цикла, как все остальные операторы.

Нижеследующий фрагмент программы иллюстрирует использования цикла **`for`** для той же задачи, для которой ранее был использован цикл **`while`**:

```
double total = 10.;
double sum;
int i;
for (i = 1, sum = 0; i <= total; i++)
{
      sum = sum + i;
}
```

9.3 Цикл do … while

Рассмотренные выше циклы **while** и **for** относятся к циклам с предусловием. В них условие проверяется перед первым выполнением тела цикла. И если оно не верно, то тело цикла ни разу не будет выполнено. В цикле **do … while** первая проверка условия выполнятся только после первой итерации цикла. Это цикл с постусловием. Поэтому тело цикла всегда будет выполнено хотя бы один раз. Цикл **do … while** имеет следующий синтаксис:

```
do оператор; while (условное выражение);
```

Для иллюстрации использования этого цикла мы, уже традиционно, просуммируем целые числа от 1 до total:

```
double total = 10.;
double sum=0.;
int i;

sum = 0;
i = 1;
do
{
      sum = sum + i;
      i = i + 1;
} while (i <= total);
```

Заметим, что внутри любого из циклов можно в качестве оператора использовать тот же или другие операторы цикла. Получается структура вложенных циклов, которая допускается в языке Си.

10 Досрочное прекращение цикла или его итерации

Иногда необходимо прервать выполнение цикла до очередной проверки условия. Это можно сделать с помощью уже знакомого оператора **break**, который в этом случае обычно используется в сочетании с условным оператором. Еще один оператор, **continue**, используется для прерывания текущей итерации цикла и досрочному переходу к очередной проверке условия. Заметим, что в случае нескольких вложенных циклов оба эти оператора действуют только на самый внутренний цикл, в теле которого они расположены. Завершить все вложенные циклы одним оператором **break** нельзя.

Нижеприведенная программа иллюстрирует использование этих операторов на примере задачи подсчета числа вводимых по одному символов с клавиатуры. При этом пробел не подсчитывается, а при вводе символа «z» подсчет завершается.

```
#include <stdio.h>

int main()
{
      char s;
      int scount = 0;

      while (1)
      {
            scanf("%c", &s);
            if (s == ' ') continue;  // пробел не подсчитываем
            scount++;
            if (s == 'z') break;     // при вводе z завершаем подсчет
      }
      printf("%d\n", scount);

      return 0;
}
```

В программе подсчет реализован с помощью «бесконечного» цикла **while** (1 – это истина!), так как мы заранее не знаем, сколько будет введено символов. Переменная `scount` служит счетчиком введенных символов. Первый оператор тела цикла осуществляет ввод символа с клавиатуры в переменную `s`. Если это пробел, то ввод не засчитывается и с помощью оператора **continue** происходит переход к следующей итерации цикла (вводу следующего символа). При вводе символа `z` цикл завершается оператором **break**. После завершения цикла значение переменной `scount` выводится в консольное окно программы.

Тема 3. Массивы, строки, директивы препроцессора, стандартные математические функции, приоритет операторов, битовые операторы.

1 Массивы

Массивом называется поименованная последовательность однотипных элементов данных (переменные с индексом). Важным обстоятельством является то, что все элементы массива имеют один тип, а значит одинаковую длину в байтах, и располагаются в памяти последовательно друг за другом. Это позволяет, зная, где расположено в памяти начало массива, легко определить положение (адрес) любого из его элементов, т.е. к ним обеспечен быстрый доступ. Перед использованием в программе массив, как и любую другую (скалярную) переменную, необходимо объявить. Объявление массива состоит из типа его элементов, собственно имени массива и числа его элементов в квадратных скобках, например, массив из 20 элементов типа **int** можно объявить так:

```
int a[20];
```

Обратите внимание, что число элементов массива необходимо задавать целочисленной константой, все массивы в языке Си должны иметь константный размер, переменную для этого использовать нельзя. Однако, можно использовать макроопределение (см. ниже):

```
#define N 20
int a[N];
```

или перечисление (см. тему 9).

Обращение к конкретному элементу массива происходит по имени массива и порядковому номеру (индексу) элемента. Например, присвоить значение 10 пятому элементу массива:

```
a[5] = 10; // или i = 5; a[i] = 10;
```

Нумерация элементов начинается с нуля. Поэтому максимальный индекс элемента массива `a` равен 19. Одной из распространенных ошибок начинающих и не только программистов на языке Си является обращение к несуществующему элементу массива с непредсказуемыми последствиями:

```
a[20] = 10;
```

Возможные варианты имени массива и область его видимости ничем не отличается от обычных переменных. Инициализировать элементы массива можно не только поэлементно, но и списком в момент его объявления:

```
double a[4] = { 1.34, 1.48, 1.65, 2.96 };
```

На основе списка начальных данных в фигурных скобках компилятор может сам подсчитать необходимое число элементов массива. Поэтому допускается такое объявление:

```
double a[] = { 1.34, 1.48, 1.65, 2.96 };
```

2 Строки

В языке Си достаточно удобно работать с текстовыми строками, несмотря на то, что строкового типа, как такового, нет. Строкой называется массив элементов типа **char**, который завершается специальным символом конца строки `'\0'`. Такой подход, с одной стороны, требует дополнительной памяти (дополнительного элемента массива) для хранения служебной информации, но с другой стороны, упрощает работу со строками, т.к. для обработки строки

достаточно знать, где в памяти расположено её начало. Константная строка записывается в двойных кавычках. Для иллюстрации рассмотрим следующее объявление массива в программе:

```
char str[] = "Hello";
```

В этом случае компилятор выделит память на 6 элементов типа **char** – для 5 символов в слове «Hello» и 1 символа конца строки. Такой массив можно было бы объявить и так:

```
char str[6] = {'H','e','l','l','o','\0'};
```

Так как двойные кавычки используются для записи строки, то возникает вопрос: как записать сам символ «двойные кавычки»», входящий в состав строки?

Для этого используются специальные кодовые последовательности или спецсимволы, которые представляют собой символ "\" (обратный слеш), за которым следует один или несколько других символов. Компилятор вместо такой кодовой последовательности вставит в строку единственный символ, например:

```
char str[] = "Hello\n\"world!\"\n";
```

В этой строке использовано 4 кодовых последовательности: \n — это символ, который при выводе обеспечит перевод курсора на следующую строку (перевод строки), а \" — это символ двойных кавычек, в которые будет заключено второе слово.

Таблица 3.1 Обозначения специальных символов в строках

Код	Назначение
\0	Символ с кодом 0, маркер конца строки.
\a	Звуковой сигнал (звонок).
\b	Возврат курсора на одну позицию назад. Символ, находящийся на этой позиции, будет перезаписан следующей операцией вывода.
\r	Возврат курсора в начало текущей строки. Информация, находившаяся в строке к этому моменту, будет перезаписана следующей операцией вывода.
\n	Перевод строки. Следующий символ печатается с начала новой строки.
\t	Горизонтальная табуляция.
\\	Одиночный символ \ (обратный слеш).
\"	Одиночный символ " (двойная кавычка).
\'	Одиночный символ ' (апостроф). Внутри строк эту последовательность использовать не обязательно
\oct	Вывод символа по его ASCII коду, здесь oct — код символа в восьмеричной системе счисления (три цифры), например: "\041" — это альтернативная запись строки, содержащей один восклицательный знак: "!".
\xhex	Вывод символа по его ASCII коду, здесь hex — код символа в шестнадцатеричной системе счисления (две цифры), например: "\x21" — это альтернативная запись строки, содержащей один восклицательный знак: "!".

В остальных случаях последовательность из обратного слеша и любого символа представляет собой этот же самый символ: `"\Q"` — это строка из одного символа `Q`.

3 Многомерные массивы

Рассмотренные выше одномерные массивы ещё называют векторами, т.к. элементы такого массива можно трактовать как координаты вектора. Наряду с одномерными массивами в языке Си можно объявлять многомерные массивы, например, двумерный массив, который можно интерпретировать как двумерную матрицу однотипных элементов:

```
double a[2][3]; //
```

$$\begin{pmatrix} a[0][0] & a[0][1] & a[0][2] \\ a[1][0] & a[1][1] & a[1][2] \end{pmatrix}$$

При объявлении число строк матрицы указывается в первой паре квадратных скобок, число столбцов – во второй. Обращение к элементам двумерного массива происходит по имени массива и двум индексам (см. схему хранения элементов в комментарии приведенного выше примера).

Обратите внимание, что индексы в многомерном массиве записываются в отдельных парах квадратных скобок в отличие, например, от такой формы записи `a[1, 2]`, которая означает обращение к одномерному массиву и «операцию запятая». Это не случайно. Такая запись отражает тот факт, что двумерный массив трактуется как «массив массивов». В памяти элементы массива располагаются по строкам. Позднее это будет важно при передаче массива в функцию и при работе с указателями.

В языке Си допускаются массивы и большей размерности, объявление которых можно написать по аналогии с двумерными массивами. Ограничения связаны только с выделяемыми системой ресурсами.

4 Операторы `sizeof()` и `typedef`

Оператор **`sizeof`**`()` возвращает длину операнда в круглых скобках в байтах. Например, можно узнать длину массива:

```
double x[8];
printf("%d", sizeof(x));
```

Результат будет 64. Если речь идёт о переменной или, как в примере, о массиве, то заключать их имена в круглые скобки не обязательно. Часто этот оператор используется для того, чтобы понять, сколько байт отводится в программе на определённый тип данных, например, **`double`**:

```
printf("%d", sizeof(double));
```

Оператор **`typedef`** позволяет дать собственное имя-синоним типу данных. Например, можно назвать тип **`int`** именем, скажем, `disco`:

```
typedef int disco;
```

и затем использовать в программе такие объявления переменных:

```
disco x;
```

Конечно, так делать не рекомендуется, иначе вы сами рискуете не понять свою собственную программу. Оператор **`typedef`** обычно используется для того, чтобы дать краткое имя структурному типу, которое исходно в языке Си выглядит довольно громоздко. О структурах мы поговорим позже.

5 Директивы препроцессора

Формально эти директивы не являются частью языка Си, но фактически присутствуют в большинстве программ на этом языке. Эти директивы начинаются с символа «#» и могут быть использованы в любом месте программы. Идея здесь следующая. Прежде, чем текст программы будет передан компилятору, он обрабатывается препроцессором, который выполняет предназначенные для него директивы. Такой подход, в частности, позволяет повторяющуюся из одной программы в другую рутинную часть кода располагать в отдельном, так называемом заголовочном файле, содержимое которого не загромождает текст программы, но будет вставлено в неё перед компиляцией. Функциональность директив препроцессора не ограничивается вставкой файлов. Рассмотрим некоторые из них.

5.1 Директива `#include`

Эта директива предназначена для включения в месте её расположения содержимого другого текстового файла, например:

```
#include <stdio.h>
#include "my.h"
```

Часто включаемые файлы имеют расширение `.h`, что является сокращением от слова «header», то есть заголовок, поскольку включаются они обычно в самом начале — в заголовке программы. Если в директиве препроцессора имя файла заключено в угловые скобки, то ищется он в системной папке, определяемой компилятором. Типичное содержимое заголовочных файлов — это прототипы (заголовки) библиотечных функций (для контроля типов передаваемых параметров и возвращаемого значения компилятору в момент обработки вызова функции нужно знать либо её определение, либо прототип, в котором указываются эти типы), определения символьных констант, макроопределения и др.

Также можно включить пользовательский заголовочный файл. Если он располагается в предназначенной для этого папке, например, текущей папке проекта, то достаточно его имя заключить в двойные кавычки.

Обратите внимание, что директивы препроцессора не завершаются точкой с запятой — они не являются операторами самого языка Си, а составляют дополнительный язык программирования для управления текстом программы.

5.2 Директива `#define`

Директива `#define` определяет макроподстановку (макрос). После этой директивы препроцессор заменит в следующей за ней части программы первый операнд на последующую строку (второй операнд). Используется, в том числе, для определения символьных констант:

```
#define MM 100
```

После такого определения в листинге программы вместо явной записи константы `100` можно использовать символы `MM`. Заметим, что если эту директиву завершить точкой с запятой, то в программе символы `MM` будут заменены не на «`100`», а на «`100;`» что может привести к трудно диагностируемым ошибкам.

Имя символьной константы обычно содержит буквы в верхнем регистре (заглавные). Это не является обязательным требованием, но так обычно делают, чтобы в тексте программы было легче отличить константу от переменной.

В языке Си для объявления константы можно использовать ключевое слово `const`, например:

```
const int MM=100;
```

Такой способ является предпочтительным, т.к. в этом случае константа получает тип, который компилятор может контролировать.

Директиве **#define** можно передавать набор параметров, которые при этом заключаются в круглые скобки. Например, для возведения в квадрат можно задать следующий макрос:

```
#define SQ(x) x*x
```

Вызов в программе макроса с параметром внешне напоминает вызов функции:

```
	a = SQ(4);
```

однако им не является. Вместо этого препроцессор заменит приведенное выше выражение на

```
	a = 4*4;
```

Отсюда следует важное замечание. Если мы воспользуемся макросом для возведения в квадрат, например, суммы двух чисел

```
	a = SQ(2+2); // a = 2+2*2+2  = 8 !
```

то после препроцессора на вход компилятора попадёт выражение, записанное в комментарии к примеру, и при выполнении программы переменная `a` вместо ожидаемого значения 16 получит 8! Чтобы минимизировать такого сорта недоразумения, лучше при определении макроса использовать круглые скобки:

```
#define SQ(x) ((x)*(x))
```

Здесь внутренние скобки решают вышеприведенную проблему, а внешние нужны для того, чтобы проблемы не возникли, если макрос будет частью более сложного выражения.

5.3 Условные директивы препроцессора

Они напоминают условный оператор **if** языка Си, но начинаются с символа «#». В качестве примера рассмотрим следующую задачу. Пусть мы хотим в одном проекте написать две разные программы и иметь возможность быстро собирать либо одну программу, либо другую. Можно поступить следующим образом: в начале программы определяем символьную константу, значение которой будет служить своеобразным переключателем между двумя ветвями условной компиляции. Далее пишем две разные функции `main()`, окружая их условными директивами препроцессора:

```
#define MAIN 1

#if MAIN == 1
	int main()
	{
		операторы1;
	}
#else
	int main()
	{
		операторы2;
	}
#endif
```

Если мы всё оставляем как в примере, то на компиляцию после препроцессора попадет только первая функция `main()`. Но стоит нам просто закомментировать директиву **#define** (в этом случае неопределённый `MAIN` будет считаться нулём) или определить константу `MAIN` отличным от `1` значением, как на компиляцию отправится только вторая функция `main()`.

Условная директива **#ifdef** позволяет узнать, определена какая-либо символьная константа или нет. Например, в вышеприведенном примере её можно было бы использовать вместо директивы **#if**:

```
#ifdef MAIN
```

В этом случае, чтобы перейти к компиляции второй функции `main()`, надо именно закомментировать директиву `#define MAIN 1`, т.к. любое значение символьной константы `MAIN` (даже без конкретного значения -- по директиве `#define MAIN` без второго операнда) даст истину при выполнении директивы `#ifdef MAIN`. В сложных булевских выражениях с этой же целью можно использовать эквивалентный оператор препроцессора

```
#if defined(MAIN) && C > 2
```

Ещё одна директива условной компиляции `#ifndef` считает выражение истинным, если её операнд не определен. Обратите внимание, что любая условная директива препроцессора `#if`, `#ifdef` или `#ifndef` должна завершаться директивой `#endif`. Ветвь `#else` в условной компиляции возможна, но не является обязательной.

5.4 Стандартные математические функции

Формально в языке Си нет математических функций, но они поставляются вместе с компилятором в виде библиотеки стандартных функций. Поэтому вы всегда можете на них рассчитывать. В текст программы, использующей функции из этой библиотеки, требуется включить заголовочный файл:

```
#include <math.h>
```

В Таблице 3.2 приведены прототипы (краткие заголовки) некоторых функций, которые дают представление о названии функции, о типе и количестве аргументов, а также о типе возвращаемого значения. Часть менее очевидных функций снабжены комментариями.

Таблица 3.2 Стандартные математические функции

Заголовок функции	Комментарий/пример
`double sin(double x);`	`y = sin(x);`
`double cos(double x);`	`y = cos(x);`
`double tan(double x);`	`y = tan(x);`
`double sqrt(double x);`	корень квадратный
`double exp(double x);`	$y = e^x$
`double log(double x);`	натуральный логарифм
`double log10(double x);`	десятичный логарифм
`double fabs(double x);`	модуль вещественного числа
`int abs(int k);`	модуль целого числа
`double fmin(double x, double y);`	минимум из двух вещественных чисел, только начиная со стандарта C99
`double fmax(double x, double y);`	максимум из двух вещественных чисел, только начиная со стандарта C99
`double cosh(double x);`	косинус гиперболический: $\cosh(x) = \frac{e^x + e^{-x}}{2}$

`double sinh(double x);`	синус гиперболический: $\sinh(x) = \frac{e^x - e^{-x}}{2}$
`double tanh(double x);`	тангенс гиперболический: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
`double atan2(double y, double x);`	Арктангенс с двумя аргументами. Возвращает в радианах угол α между осью x и линией, проведенной из начала координат (0, 0) в точку с координатами (x, y). Диапазон изменения угла: $-\pi < \alpha \leq \pi$.
`double pow(double y, double z);`	`x = pow(y, 3.12); // x = y^3,12`

В отличие от некоторых других языков Си не имеет оператора возведения в степень. Функция `pow()` решает эту проблему. Аргументы функции `atan2()` можно рассматривать как пропорциональные синусу угла (первый аргумент) и косинусу (второй аргумент). При этом анализируются знаки синуса и косинуса, что позволяет определить угол в диапазоне 2π.

5.5 Генератор псевдослучайных чисел

Функция `rand()` возвращает целое псевдослучайное число в диапазоне от `0` до `RAND_MAX`. Прототип функции имеет вид `int rand(void);` ключевое слово `void` (переводится как пустой) в круглых скобках означает, что функция не имеет аргументов. Функция находится в отличной от математической библиотеке стандартных функций, и её использование предполагает включение в программу заголовочного файла `stdlib.h`. При последовательном вызове этой функции генерируется новое псевдослучайное число.

В программе полезно предусмотреть смену начальной точки генератора с помощью однократного вызова функции `srand()`, чтобы при новом запуске программы не получить тот же самый ряд псевдослучайных чисел. В свою очередь, чтобы сдвиг начальной точки от запуска к запуску программы отличался, значение аргумента функции `srand()` должно быть разным. Часто для этого используется функция `time()`, которая выдает различное время, если только программы не запускаются на исполнение одновременно.

Следующий пример иллюстрирует генерацию двух псевдослучайных чисел с сохранением их в массиве и предварительной сменой начальной точки генератора:

```
#include <stdlib.h>
#include <time.h> // здесь задан прототип функции time()

int main()
{
      int i[2];
      srand((unsigned)time(NULL));
      i[0] = rand();
      j[1] = rand();
}
```

Аргумент `NULL` функции `time()` является указателем, о которых речь пойдёт в одной из следующих тем.

5.6 Приоритет операторов

Если в одном выражении встречается несколько операторов, то они выполняются в порядке убывания приоритета. О приоритете основных арифметических операторов мы уже говорили в

прошлой теме. Но на самом деле число уровней приоритета гораздо больше и касаются они не только арифметических операторов, но и логических и некоторых других. Эти уровни сгруппированы в Таблице 3.3. Некоторые операторы вам могут быть не знакомы пока, о них пойдет речь позже. Чем меньше номер уровня оператора в таблице, тем выше приоритет этой операции. В рамках одной группы, за исключением 2, 3, 15 и 16 операторы выполняются слева направо.

Таблица 3.3 Группы уровней приоритетов операторов

№	Операции	Пояснение
1	`(), [], x.y, p->y, x++, x--`	скобки, выделение элемента, выделение элемента по указателю, постфиксное увеличение и уменьшение
2	`sizeof(), &x, *p, -x, ~x, !x, ++x, --x`	определение размера, получение адреса, разыменование указателя, унарный минус, побитовое и логическое отрицание, префиксное увеличение и уменьшение
3	`(any_type)`	приведение типа
4	`x*y, x/y, x%y`	умножение, деление, деление по модулю (взятие остатка)
5	`x+y, x-y`	сложение, вычитание
6	`x>>y, x<<y`	побитовый сдвиг
7	`x < y, x <= y, x > y, x >= y`	сравнение «меньше», «меньше или равно», «больше», «больше или равно»
8	`x == y, x != y`	сравнение «равно», «не равно»
9	`x & y`	побитовое «И»
10	`x ^ y`	побитовое исключающее «ИЛИ»
11	`x | y`	побитовое «ИЛИ»
12	`x && y`	логическое «И»
13	`x || y`	логическое «ИЛИ»
14	`q ? x : y`	тернарный условный оператор
15	`=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=`	варианты оператора присваивания
16	`x , y`	операция «запятая» (последовательное вычисление)

Заметим, что, согласно этой таблице, наибольший приоритет имеют, в частности, операторы постфиксного увеличения «++» и уменьшения «--», т.е. при наличии в выражении других операторов с меньшим приоритетом постфиксные операторы будут выполнены первыми. Однако это не меняет их смысл — при дальнейшем вычислении данного выражения будут использованы старые значения переменных, которые вернут постфиксные операторы.

Ещё один вопрос, косвенно касающийся приоритета операторов, связан с определением того, какой именно оператор записан в выражении. Рассмотрим следующий пример:

```
int x = 2, y = 2, z;
z = x+++y;
```

Как следует трактовать последнюю строчку фрагмента программы с большим числом подряд идущих символов? Какие тут записаны операторы? В соответствие с правилами компилятор связывает подряд идущие символы слева направо, пытаясь получить наиболее длинный (по количеству символов) оператор. Поскольку оператора «+++» не существует, то наиболее длинным является оператор «++», который должен стоять рядом с переменной. В итоге получается постфиксное увеличение `x` и оставшийся символ «+» трактуется как оператор сложения. В результате переменная `x` примет значение `3`, переменная `y` останется равной `2`, а `z` будет присвоено значение `4`. На самом деле, таких неочевидных выражений лучше избегать при написании программы, более активно используя скобки для ясного понимания программы как компилятором, так и человеком:

```
z = (x++)+y;
```

5.7 Побитовые операторы

Если записать байт в двоичной системе счисления, то двоичные разряды, представляющие коэффициенты при соответствующей степени двойки, и есть биты. Бит может быть «выставлен», т.е. равен 1, или нет — тогда его значение равно 0. С помощью битовых операторов можно манипулировать битами любого целочисленного выражения или переменной. Для простоты рассмотрим беззнаковые однобайтовые переменные, например,

```
unsigned char c=8;
```

В двоичном представлении эта переменная имеет только один выставленный в единицу бит:

0	0	0	0	1	0	0	0

2^7 2^0

Побитовый сдвиг. Операторы «<<» и «>>» сдвигают в числе биты в первом операнде по направлению стрелок на число разрядов, указанном во втором операнде, например:

```
c = c << 3; // смещение битов влево на три разряда, что
            // эквивалентно умножению на 23 ≡ 8

c = c >> 3; // смещение битов вправо на три разряда, что
            // эквивалентно целочисленному делению на 23 ≡ 8
```

При сдвиге биты, уходящие за правую или левую границу байта, теряются, а приходящие имеют значение 0.

Побитовое «**И**» (&) выполняет логическую операцию «И» над каждой парой одинаковых двоичных разрядов двух операндов. В примерах справа в рамочке показана иллюстрация выполнения операции в двоичном представлении:

```
unsigned char a = 3;

unsigned char b = 6;

c = a & b;  // c = 2
```

```
  00000011
& 00000110
----------
  00000010
```

Побитовое «И» часто используется для ответа на вопрос: «выставлен» ли в переменной n-ый бит, например:

```
unsigned flag = 1;
if (c & (flag << n)) оператор; // выставлен n-ый бит в c!
```

Побитовое «**ИЛИ**» (|) выполняет логическую операцию «ИЛИ» над каждой парой одинаковых двоичных разрядов двух операндов.

```
c = a | b; // c = 7
```

```
  00000011
| 00000110
----------
  00000111
```

Пример: «выставить» в переменной c n-ый бит, не затрагивая остальные биты:

```
unsigned char flag = 1;
c = c | (flag << n)
```

Побитовое «исключающее **ИЛИ**» (^) выполняет логическую операцию «исключающего ИЛИ» над каждой парой одинаковых двоичных разрядов двух операндов:

```
c = a ^ b; // c = 5
```

```
  00000011
^ 00000110
----------
  00000101
```

Побитовое «**НЕ**» (~) выполняет логическое «НЕ» над каждым битом по отдельности:

```
c = ~b; // c = 249
```

```
~ 00000110
----------
  11111001
```

Приведём пример использования побитовых операторов для выяснения, является ли значение целочисленной переменной x целой степенью двойки:

```
unsigned int x = 128;
printf("%d", !(x & (x-1)));
```

Выражение во втором аргументе функции `printf()` принимает значение 1 (истина), если x есть целая степень двойки (как в примере), и 0 (ложь) в противном случае.

Все побитовые операторы с двумя операндами могут использоваться в краткой записи оператора присваивания, например, `c <<= 3;` (`c = c << 3;`).

Тема 4. Функции, глобальные переменные, модульный подход в программировании.

1 Функции. Передача параметров по значению.

Функция — самостоятельная единица программы, спроектированная для решения конкретной задачи. Изначально функции в основном предназначались для многократного использования кода и сокращения объёма программ. Обычно сейчас так применяются функции, предоставляемые разработчиком компилятора [например, `rand()` или `sin()`]. В настоящее время, когда объемы оперативной памяти даже мобильных телефонов достигают нескольких гигабайт, функции в большей степени предназначены для разделения кода на относительно независимые блоки. Они повышают уровень модульности программы, облегчают ее чтение, внесение изменений и исправление ошибок. Определение любой функции имеет следующий вид:

```
<тип_результата> <имя_функции>(<тип1 аргумент1> [, другие аргументы...] )
{
  // тело функции
  <выполняемые операторы>;
}
```

Одну функцию внутри другой определять нельзя — все функции глобальны в программе.

Первая строка, в которой описывается тип возвращаемого значения, имя функции и принимаемые функцией аргументы[1] называется *заголовком функции*. Функция может не возвращать значение. В таком случае она должна иметь тип **`void`**, что переводится на русский язык как «пустой». Блок в фигурных скобках, который следует за заголовком функции и описывает то, что именно функция делает, называется *телом функции*. Например, создадим функцию, вычисляющую модуль числа, а в `main()` устроим проверку её работы:

```
#include <stdio.h> // Пример 1

int iabs(int x)     //функция описывается до своего вызова
// Здесь x – формальный параметр, его значение неизвестно,
// известен только тип этого аргумента.
{
  if (x < 0)
    return -x;
  return x;
}

int main()
{
  int a = 10, b = 0, c= -2;
  int d, e, f;
  d = iabs(a); // вызываем функцию iabs для фактического входного параметра a
  e = iabs(b);
  f = iabs(c);
  printf("d = %d, e = %d, f = %d\n", d, e, f);
  return 0;     // этот оператор здесь необязателен
}
```

Оператор **`return`** завершает выполнение функции и передает управление вызывающей функции. Значение выражения, стоящее после **`return`**, будет возвращено в качестве результата работы функции. Если функция имеет **`void`** в качестве типа возвращаемого значения,

[1] Количество аргументов функции не всегда определено изначально. Например, функции `printf()` и `scanf()` принимают переменное число параметров. В этой главе мы будем рассматривать только функции с определенным заранее числом параметров.

оператор **return** указывается без аргумента, причем, если он самый последний в теле функции, то его можно не указывать вовсе:

```
void print_bcd(int b, int c, int d) // Пример 2
{
  printf("b = %d c = %d d = %d\n", b, c, d);
}
```

Функция `main()` — точно такая же функция, как и все остальные, ее отличие лишь в том, что после компиляции всех файлов проекта, компоновщик начинает сборку программы именно с этой функции, и при старте программы она вызывается самой первой. Обратите внимание, в примере 1 функция `main()` описана как возвращающая значение тип **int**. Этот возврат значения реализован с помощью оператора **return**, аргументом которого указано нулевое значение, оно обычно свидетельствует о корректном завершении программы. Значение, возвращаемое `main()` по завершению программы, можно увидеть в окне сообщений отладчика. Следует отметить, что `main()` — единственная функция возвращающая значение, в которой можно в конце не писать оператор **return**. В этом случае она по умолчанию возвращает ноль.

Функцию можно вызывать только после того, как она объявлена, то есть, заголовок и тело функции в программе должны идти перед ее первым вызовом из другой функции (см. Пример 1). Иногда оказывается удобным или даже необходимым описать вызываемую функцию после вызывающей. Для этого следует перед вызывающей функцией описать заголовок (прототип) вызываемой функции. Прототип функции, описанный перед вызывающей функцией, «сообщает» компилятору, что где-то ниже будет реализована вызываемая функция. Прототипы функций содержат ее имя, тип возвращаемого значения и аргументы с советующими типами, но не содержат тела функции. Используя прототип функции `iabs()`, модифицируем Пример 1 так, чтобы она была реализована после функции `main()`:

```
#include <stdio.h> // Пример 3

int iabs(int x);   // заголовок (прототип) функции

int main()
{
  int a = 10, b = 0, c= -2;
  int d, e, f;
  d = iabs(a); // вызываем функцию iabs для фактического входного параметра a
  e = iabs(b);
  f = iabs(c);
  printf("d = %d, e = %d, f = %d", d, e, f);
  return 0;    // этот оператор необязателен здесь
}

int iabs(int x)
{
  if (x < 0)
    return -x;
  return x;
}
```

Заметим, что имена аргументов в прототипе указывать не обязательно, то есть прототип функции `iabs()` в Примере 3 может быть корректно записан как

```
int iabs (int);   // заголовок (прототип) функции iabs()
```

1.1 Локальные переменные

Все переменные, объявленные в функции, являются ее внутренними переменными и вызывающая функция о них ничего не знает, следовательно, никак не может с ними работать (ни прочитать значение, ни изменить). Локальные переменные создаются при вызове функции и уничтожаются при ее завершении. Переменные вызывающей функции не известны вызываемой функции — все объявленные в функциях переменные локальны, то есть с ними можно работать только в пределах данной функции, начиная от места их объявления и ниже.

1.2 Передача параметров по значению

Все аргументы функции, объявленные в ее заголовке (формальные параметры) — это такие же локальные переменные данной функции, вызывающая функция их инициализирует (присваивает им начальное значение), но дальше никак на них повлиять не может:

```
#include <stdio.h> // Пример 4

int sum(int x, int y)     // x, y - аргументы функции sum()
{
  int a = y;              // a - локальная переменная функции sum()
  x += a;
  printf("x = %d\n", x); // будет напечатано: x = 3
  return x;
}

int main()
{
  int b = 1, c = 2, d;    // локальные переменные функции main()
  d = sum(b,c);
  printf("b = %d c = %d d = %d\n", b, c, d);
  // Вместо предыдущей строки можно вызвать функцию print_bcd() из примера 2
}

// Будет напечатано: b = 1 c = 2 d = 3
```

Поэтому такой способ передачи параметров называется «*передача по значению*», что подчеркивает копирование значений при вызове функции.

1.3 Передача массивов в функции

Массивы в функции тоже можно передавать в качестве аргументов, однако тут есть существенная особенность: для достижения большей эффективности работы массив в языке Си определяется как машинный адрес начального элемента. Следовательно, если в функцию передать массив (адрес начального элемента), то любые изменения элементов этого массива в вызываемой функции отразятся также и на вызывающей стороне. Такая возможность языка реализована в функции `cumulative()`, см. следующий пример. Единственный способ для функции гарантировать вызывающей стороне, что переданный массив в ходе ее выполнения не будет изменен, — это объявить массив константным, как это сделано в функции `print_array()`:

```
#include <stdio.h> // Пример 5

void cumulative(int size, int A[])
{
  int i;
  for (i = 1; i < size; i++)
    A[i] += A[i-1];
}
```

```
void print_array(int size, const int A[])
{
  int i;
  for (i = 0; i < size; i++)
    printf("%d\n", A[i]);
}

int main()
{
  int X[5] = {10, 20, 30, 40, 50};
  cumulative(5,X);
  print_array(5,X);
}

// Будет напечатано в столбик: 10 30 60 100 150
```

1.4 Рекурсия

Функция может прямо либо опосредованно вызываться из самой себя. Такой вызов называется рекурсивным. Прямой вызов соответствует случаю вызова функции `A()` из функции `A()`. При опосредованном вызове из функции `A()` вызывается функция `B()`, из `B()` — `C()` и т.д., а из последней в этом ряду функция `A()`. Многократные рекурсивные вызовы функции могут быть использованы вместо циклов, и, наоборот, циклы могут заменить последовательность рекурсивных вызовов.

В качестве классического примера рекурсии рассмотрим функцию, вычисляющую факториал числа:

```
#include <stdio.h> // Пример 6

double factorial(int n)
// Здесь используется double, чтобы не допустить переполнение типа int
{
  if (n <= 0) return 1.0;     // блок выхода
  return n * factorial(n-1); // повторяемая часть
}

int main()
{
  int a = 5, b = 20;
  printf("%d! = %e\n%d! = %e\n", a, factorial(a), b, factorial(b));
}

/* Будет напечатано:
5! = 1.200000e+002
20! = 2.432902e+018
*/
```

Рекурсивная функция делится на две части: повторяемую, в которой происходит повторный вызов этой функции, и блок выхода, в котором записано условие прерывания рекурсии. Рекурсивная функция не является корректной, если такого условия нет, или оно никогда не бывает истинным — в этом случае произойдет ошибка стадии исполнения «переполнение стека[2]». Эта же ошибка возникает и при слишком большом числе последовательных рекурсивных вызовов.

Преимущество рекурсии по сравнению с циклами проявляется в относительной краткости и простоте реализации ряда алгоритмов. Рассмотрим задачу, решение которой может быть элегантно получено с использованием рекурсии. Пусть имеется автомат, который на вход

[2] Стек — специальная область памяти, куда, в частности, записывается адрес точки возврата из функции.

получает число и может совершать с ним некоторые действия, например, увеличивать его на 2 и на 5. Требуется найти число «программ» (т.е. последовательностей действий), которыми этот автомат может перевести число `s` в число `f`. Рекурсивное решение этой задачи представлено в следующем примере:

```
#include <stdio.h> // Пример 7

int count(int s, int f)
{
  if (s >  f) return 0; // блок выхода
  if (s == f) return 1; // блок выхода
  return count(s + 2, f) + count(s + 5, f); // повторяемая часть
}

int main()
{
  int s = 1, f = 20;
  printf("%d\n", count(s, f));
}

// Будет напечатано 18
```

Рассмотрим теперь опосредованный рекурсивный вызов:

```
#include <stdio.h> // Пример 8

int B(int n); // прототип или заголовок функции B()

int A(int n)
{
  if (n > 2) return A(n-1) + B(n-2);
  return n;
}

int B(int n)
{
  if (n > 2) return B(n-1) + A(n-2);
  return n + 1;
}

int main()
{
  printf("%d\n", A(25));
}

// Будет напечатано 158905
```

Ключевым элементом данной программы является прототип функции `B()` — здесь он необходим, поскольку в случае опосредованной рекурсии одна из вызывающих функций заведомо «ничего не знает» о вызываемой. Отсутствие прототипа в Примере 8 приведет к ошибке или предупреждению на стадии компиляции.

2 Глобальные переменные. Правила видимости переменных. Статические переменные.

2.1 Глобальные переменные

До настоящего времени мы использовали локальные переменные, существующие только во время исполнения функции. Они «видны» лишь в соответствующей функции, в разных функциях имена переменных могут совпадать. Так, в Примере 7 функции `count()` и `main()` содержат переменные `s` и `f` с совпадающими именами. Иногда бывает нужно создать глобальную переменную, то есть такую переменную, которая существует не только в пределах той или иной функции, но доступна в пределах всей программы (то есть нужно, чтобы любая функция программы могла с ней работать). Такие переменные объявляются вне тела любой функции и инициализируются до начала работы самой первой функции. Глобальные переменные автоматически инициализируются нулями. Поэтому, можно не заметить ошибки в случае, если предполагались другие начальные значения, так как компилятор об этом не сообщит.

Для демонстрации работы с глобальными переменными модифицируем Пример 4 так, чтобы в глобальной переменной `N` записывалось число вызовов функции `sum()`:

```
#include <stdio.h> // Пример 9

int y, N; // глобальные переменные

int sum(int x, int y) // x, y – аргументы функции sum()
{
// Переменная y - второй параметр функции, это локальная переменная, ее
// значение не имеет никакого отношения к одноименной глобальной переменной
  int a = y; // a - локальная переменная функции sum()
  x += a;
  N++; // подсчитываем число вызовов функции sum()
       // через глобальную переменную N
  y++; // данный оператор будет бесполезным, т.к.
       // он меняет значение локальной переменной
  printf("x = %d\n", x); // будет напечатано: x = 3
  return x;
}

int main()
{
  int x, d; // локальные переменные функции main()
  N = 0;    // присвоение значения глобальной переменной N
  y = 2;    // присвоение значения глобальной переменной y
  x = 1;    // присвоение значения локальной переменной x
  d = sum(y, x);
  // Параметру x функции присваивается значение глобальной переменной y
  // Параметру y функции присваивается значение локальной  переменной x
  d = sum(y, x);
  d = sum(y, x);
  d = sum(y, x);
  // При каждом вызове функции sum() значение переменной N возрастает на 1
  printf("x = %d y = %d d = %d N = %d\n", x, y, d, N);
}
// Будет напечатано: x = 1 c = y d = 3 N = 4
```

В Примере 9 имеются две глобальных переменных `N` и `y`, имя последней из которых совпадает с аргументом (т.е. локальной переменной) функции `sum()`. При совпадении имен глобальных и локальных переменных в некоторой функции будут использоваться локальные, то есть приоритет глобальных переменных ниже, чем локальных.

2.2 Статические переменные

Применение глобальных переменных может вести к ошибкам, которые трудно обнаружить. Для долговременного хранения данных, в том числе между вызовами функции, можно использовать локальные статические переменные, которые создаются в момент запуска программы, а не в момент вызова функции. Статические локальные переменные не уничтожаются между вызовами функции. Их инициализация происходит один раз при первом вызове функции, затем их значение сохраняется до следующего вызова. При последующих вызовах функции, в которой объявлены статические переменные, эта инициализация будет проигнорирована. Таким образом, статические локальные переменные ведут себя почти так же, как и глобальные, но работает с ними только одна единственная функция, та самая, в которой они объявлены.

Внутри любой функции их можно описать следующим образом:

```
static <тип> <имя> = <знач>;
```

Здесь **static** — ключевое слово, <тип> — это тип статической переменной <имя>, которая только в момент первого вызова функции инициализируется значением <знач>.

Рассмотрим пример, который подсчитывает число вызовов функции `trystat()`, используя статическую переменную `stay`, сопоставляя ее с обычной локальной переменной `fade`:

```
#include <stdio.h> // Пример 10

void trystat()
{
  int fade = 1;         // создание и инициализация локальной переменной
  static int stay = 1; // создание и инициализация локальной статической
                        // переменной
  printf("fade = %d & stay = %d\n", fade, stay);
  fade++; stay++; // увеличиваем значение переменных fade и stay на единицу
}

int main()
{
  int count;

  printf("Try 1: ");
  trystat(); // при этом вызове происходит инициализация локальной
             // статической переменной

  for (count = 2; count <= 5; count++)
  {
    printf("Try %d: ", count);
    trystat(); // здесь инициализация локальной статической переменной
               // не происходит
  }
}
```

Функция `trystat()` увеличивает на единицу каждую переменную после печати её значения. Но результаты работы этой функции для двух её переменных будут различны:

```
Try 1: fade = 1 & stay = 1
Try 2: fade = 1 & stay = 2
Try 3: fade = 1 & stay = 3
Try 4: fade = 1 & stay = 4
Try 5: fade = 1 & stay = 5
```

Статическая переменная `stay` «помнит», что её значение было увеличено на 1, в то время как для переменной `fade` начальное значение равное единице устанавливается каждый раз заново.

3 Модульный подход в программировании и раздельная компиляция

Как правило, программы на языке Си состоят из большого числа небольших функций. Рано или поздно становится неудобно помещать все функции в один файл — удобнее разбить связанные между собой функции по разным файлам исходных текстов. Эти файлы будут компилироваться раздельно, но потом будут объединены в одну программу на последнем этапе сборки проекта. При необходимости на этом последнем этапе сборки будут добавлены и функции стандартных системных библиотек языка Си.

Предположим, что компоненты программы имеют большие размеры, и мы хотим распределить их по нескольким файлам. Пусть функция `main()` находится в отдельном файле, и каждая функция также находится в отдельном файле. Создадим проект и присоединим все файлы с функциями к нему. Однако в этом случае возникнет ошибка компиляции — к моменту вызова какой-либо функции она еще не будет нигде определена. Чтобы эта ошибка не возникала, в файле с исходным текстом вызывающей функции достаточно поместить заголовок вызываемой функции:

```
#include <stdio.h> // Пример 11

void trystat(); // прототип функции trystat() из примера 10
                // функция реализована в отдельном файле

int main()
{
  int count;

  printf("Try 1: "); trystat();
  for (count = 2; count <= 5; count++)
  {
    printf("Try %d: ", count); trystat();
  }
}
```

Однако, это неудобно: в каждом файле с вызывающими функциями придётся дублировать все заголовки вызываемых функций. Поэтому обычно поступают так: выносят все заголовки функций в отдельный заголовочный файл (традиционно он имеет расширение имени `.h` от английского слова «header», заголовок), который затем включают во все исходные тексты программы при помощи оператора препроцессора **`#include`**. В этот же заголовочный файл выносят определения констант и общие для всей программы операторы препроцессора.

Заголовочные файлы системных библиотек типа `math.h` ничем не отличаются от ваших собственных заголовочных файлов — в них точно также содержатся все общие для данной библиотеки объявления. Разница заключается только в том, что свои файлы вы указываете оператору **`#include`** в двойных кавычках, а системные заголовочные файлы указываются в угловых скобках.

Используя заголовочный файл перепишем Пример 11:

```
#ifndef _TRSTT_H_    // Пример 12, файл "trstt.h"
#define _TRSTT_H_

#include <stdio.h>  // подключаем необходимые системные заголовочные файлы

void trystat();

#endif // конец файла "trstt.h"
```

```c
#include "trstt.h" // файл "trstt.c"

void trystat()
{
  int fade = 1;
  static int stay = 1;

  printf("fade = %d & stay = %d\n", fade, stay);
  fade++; stay++;
} // конец файла "trstt.c"
```

```c
#include "trstt.h" // файл "main.c"

int main()
{
  int count;

  printf("Try 1: "); trystat();
  for (count = 2; count <= 5; count++)
  {
    printf("Try %d: ", count); trystat();
  }
} // конец файла "main.c"
```

Обратите внимание, в файле `trstt.h` блок операторов препроцессора применяется для того, чтобы все определения, которые есть в этом заголовочном файле, реализовались бы во включающем файле строго один раз вне зависимости от того, сколько раз этот заголовочный файл будет включен в него. Это довольно распространенный прием в языке Си.

Заголовочные файлы удобно включать в проект Microsoft Visual Studio в папку «Header files», поскольку это позволит быстро открывать их в редакторе. Однако, это делать необязательно, так как в текст программы они вставляются директивой препоцессора **`#include`**.

3.1 Внешние переменные

Пусть в некотором программном модуле (файле с исходным текстом) определены глобальные переменные. Для того, чтобы получить возможность воспользоваться ими в другом программном модуле, необходимо в этом модуле объявить их внешними глобальными переменными, используя ключевое слово **`extern`**.

Сохранив функциональность Примера 11, модифицируем его таким образом, что переменная `stay` будет объявлена как глобальная в файле `main.c` и там же будет инициализирована единицей. Для работоспособности программы объявим в файле `trstt.c` переменную `stay` как внешнюю (файл `trstt.h` остается без изменений):

```c
#include "trstt.h" // Пример 13, файл "main.c"

int stay = 1;       // объявляем глобальную переменную

int main()
{
  int count;

  printf("Try 1: "); trystat();
  for (count = 2; count <= 5; count++)
  {
    printf("Try %d: ", count); trystat();
  }
} // конец файла "main.c"
```

```
#include "trstt.h" // файл "trstt.c"

extern int stay;   // объявляем внешнюю глобальную переменную

void trystat()
{
  int fade = 1;

  printf("fade = %d & stay = %d\n", fade, stay);
  fade++; stay++;
} // конец файла "trstt.c"
```

Если нет желания использовать объявленные глобальные переменные во всем файле (например, в других функциях должны быть переменные с теми же именами), то можно объявить внешние переменные локально внутри нужной функции:

```
#include "trstt.h" // Пример 14, файл "trstt.c"

void trystat()
{
  int fade = 1;
  extern int stay; // объявляем внешнюю глобальную переменную
                   // внутри функции trystat()
  printf("fade = %d & stay = %d\n", fade, stay);
  fade++; stay++;
} // конец файла "trstt.c"
```

3.2 Статические глобальные переменные

Наконец, бывает необходимость «закрыть» глобальную переменную, объявленную в данном модуле от любого использования в других модулях одной и той же программы. Для этого используется такое же ключевое слово **static**, что и для объявления локальных статических переменных, но смысл его в данном контексте другой: это позволяет объявить глобальную переменную с областью видимости, ограниченной отдельным программным модулем. Рассмотрим это на примере:

```
#include <stdio.h> // Пример 15, файл "f.h"

double f1(double x);
double f2(double x);
// конец файла "f.h"
```

```
#include "f.h" // файл "f1.c"

static double eps = 1e-5; // глобальная переменная, которая "видна"
                          // только внутри файла "f1.c"
double f1(double x)
{
  if (x < eps) return 0.0;
  return 1 / x;
} // конец файла "f1.c"
```

```
#include "f.h" // файл "f2.c"

static double eps = 1e-3; // глобальная переменная, которая "видна"
                          // только внутри файла "f2.c"
double f1(double x)
{
  if (x < eps) return 0.0;
  return 1 / x / x;
} // конец файла "f2.c"
```

```c
#include "file.h" // файл "main.c"

int main()
{
  printf("%lf\n", f1(1e-4));
  printf("%lf\n", f2(1e-4));
} // конец файла "main.c"
```

Будет напечатано:

```
10000.000000
0.000000
```

В каждом из двух файлов исходных текстов переменная `eps` — своя, она может меняться совершенно независимо от другой и ни одна из функций одного файла никак не может получить доступ к переменной `eps` из другого файла. Попытка использовать оператор **`extern`** в данном случае также ни к чему не приведет — доступ будет получен только к своей собственной переменной.

Тема 5. Указатели, динамическая память.

1 Указатели. Основные действия с указателями.

Указатель — это переменная, которая содержит адрес (то есть целое число — порядковый номер первого байта той области памяти, в которой она расположена) другой переменной. Указатели широко применяются в языке Си, потому что во многих случаях без них невозможно обойтись, и в некоторых случаях они существенно ускоряют работу программы. Любая переменная имеет значение и располагается по некоторому адресу в оперативной памяти. Указатель также имеет значение и расположен по какому-то адресу, отличие тут в том, что значением указателя является адрес другой переменной.

Как и любую другую переменную, указатель необходимо объявить перед использованием. Объявляется указатель следующим образом[3]:

```
<тип> *<имя>;
```

<тип> — это тип переменной, адрес которой будет содержать указатель <имя>.

Объявленный и не инициализированный указатель будет указывать «в никуда» (указатель содержит случайный адрес) и, если использовать такой указатель, можно выйти за пределы адресного пространства, отведённого программе операционной системой, в результате чего программа будет аварийно завершена. Для начальной инициализации указателя в тот момент, когда ещё неизвестно, какой именно адрес он будет содержать, определено специальное значение NULL, то есть такое значение указателя, для которого гарантируется, что он не является адресом каких-либо данных (или функций) в доступной программе оперативной памяти. Целым числом, соответствующим указателю NULL, является ноль. Для использования значения NULL нужно подключить любой из заголовочных файлов: stdio.h, stdlib.h, stddef.h, locale.h, string.h.

Основные действия с указателями и переменными, адреса которых могут храниться в них, выполняются двумя следующими операторами:

& (амперсанд)	оператор взятия адреса, с помощью которого можно получить адрес переменной любого типа, результатом будет указатель на эту переменную
* (звёздочка)	оператор разыменования указателя (оператор получения значения по адресу), позволяющий получить значение по адресу, хранящемуся в указателе-операнде

Рассмотрим пример работы с этими операторами:

```
#include <stdio.h>  // Пример 1

int main()
{
  int a = 0x11223344; // объявляем и инициализируем переменную типа int
  int *p = NULL;      // объявляем и инициализируем указатель на int
  printf("Address of p: %p \n", &p); // Выводим адрес p

  p = &a; // присваиваем указателю p значение – адрес переменной a
  printf("Address of a: %p    Content of p: %p\n", &a, p);
  // Выводим адрес переменной a и значение переменной p
  printf("a: %x  *p: %x\n", a, *p);
  // Выводим значение переменной a и значение, хранящееся по адресу p
```

[3] Расстановка пробелов здесь не играет роли, допустима, например, и такая запись:

```
<тип>* <имя>;
```

```
  a = 0x11223388;    // меняем значение переменной a
  printf("a: %x  *p: %x\n", a, *p); // снова выводим a и *p
  *p = 0x112233dd;   // меняем значение, хранящееся по адресу p
  printf("a: %x  *p: %x\n", a, *p); // еще раз выводим a и *p
  printf("Address of p: %p\n", &p);
  // Выводим адрес p, убеждаемся, что он не изменился
}
```

Будет выведено:

```
Address of p: 000000000023FE40
Address of a: 000000000023FE4C    Content of p: 000000000023FE4C
a: 11223344  *p: 11223344
a: 11223388  *p: 11223388
a: 112233dd  *p: 112233dd
Address of p: 000000000023FE40
```

Отметим, что в языке Си имеется тип указателей **void***, который хранит только адрес области памяти безотносительно типа данных, размещающихся по этому адресу. Тем самым, оператор разыменования неприменим к переменным типа **void***. Для работы с данными, которые хранятся по адресу, который хранит переменная типа **void***, ее необходимо явно привести к указателю, соответствующему обрабатываемым данным, см. разделы 3 и 6. Обратное преобразование указателя любого иного типа к **void*** осуществляется автоматически.

2 Арифметика указателей. Указатели и массивы.

Кроме уже упомянутого выше оператора разыменования «*» к указателям применимы следующие операторы:

- присваивания =
- сложения/вычитания (в том числе совмещенного с присваиванием) с целыми числами +, –, +=, –=, ++, --
- вычитания указателей –
- сравнения ==, !=, >, <, >=, <=
- индексирования []

Оператор присваивания одного указателя другому работает подобно присваиванию целых чисел.

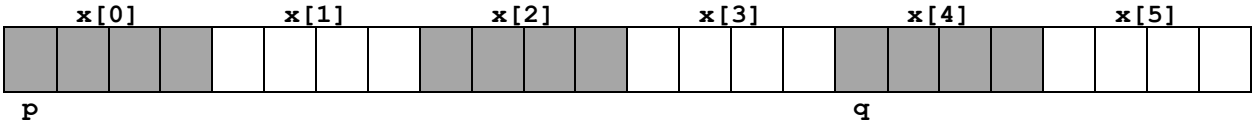
При сложении или вычитании указателя с целыми числами его значение увеличивается/уменьшается на величину этого числа, умноженного на размер в байтах переменной, на которую он указывает. В частности, использование оператора инкремента «++» увеличивает указатель на размер в байтах переменной, на которую этот указатель указывает, а использование оператора декремента «--» уменьшает его на этот размер. Если указатель указывает на элемент массива, то при этом происходит переход соответственно к последующему либо предыдущему элементу.

Для демонстрации операции сложения указателя (на переменную типа **int**, которую будем полагать четырехбайтной) с целым числом рассмотрим Пример 2. Чтобы результат этой операции также указывал на некоторую целую переменную, создадим массив[4] x из шести элементов типа **int**. Напомним, что элементы массива хранятся в памяти последовательно, поэтому Пример 2 будет корректным.

```
    int x[6];      // Пример 2
    int *p, *q;    // полагаем, что переменная типа int занимает 4 байта
    p = &x[0];     // инициализация указателя p, см. пример 3 и его описание
    q = p + 4;
```

[4] Подробности о связи массивов и указателей см. ниже.

Сопоставим данный фрагмент кода его со следующим рисунком:



В первой его строке схематически изображен массив `x`. Каждая клетка во второй строке соответствует одному байту, а группа по четыре клетки — переменной типа **`int`**. После выполнения конструкции `q = p + 4` указатель `q` становится больше `p` на 16 байт, что эквивалентно четырем переменным типа **`int`**.

Оператор вычитания указателей является обратным к оператору сложения указателя с целым числом. Результатом вычитания указателей является целое число (оно может быть положительным, нулём или отрицательным). Так для указателей `p` и `q` из Примера 2 значение `q - p` равно 4, а `(q - p) * sizeof(int)` — 16 (те самые шестнадцать байт, см. рисунок выше), при этом `p - q` будет равно –4. Сравнение указателей осуществляется подобно сравнению целых чисел. Для указателей `p` и `q` из рассматриваемого примера значение `p <= q` является истиной, а конкретно в Си единицей, а `p == q` — ложью, то есть нулем в языке Си. Любой указатель, хранящий адрес переменной в памяти компьютера, больше `NULL`.

Применительно к массивам операция индексирования `[]` предполагает доступ к некоторому элементу массива. При работе с указателями эта операция эквивалентна сложению или вычитанию указателя с целым числом с последующим разыменованием. Так, для указателей из примера 2 конструкция `p[4]` является тем же самым, что и `*(p + 4)`. К элементам массива `x` можно обращаться, используя синтаксис указателей: `*(x + 2) = 3`, что эквивалентно конструкции `x[2] = 3`.

В языке Си указатели и массивы тесно связаны между собой. Переменная — имя массива может рассматриваться как константный указатель на нулевой элемент массива. Если имя массива связано с выделенной на этапе компиляции областью памяти и никуда кроме этой области указывать не может, то указатель может содержать адрес любой переменной и менять своё значение по ходу выполнения программы. Указатель представляет собой переменную, которая расположена в физической оперативной памяти, под имя массива место в памяти не отводится (в памяти находится сам массив).

Для иллюстрации связи массивов и указателей, а также для демонстрации операторов, рассмотренных выше, приведем следующий пример:

```
#include <stdio.h>  // Пример 3

#define LEN  16

int main()
{
  double x[LEN];
  double *p, *q;
  int k;
  for (k = 0; k < LEN; k++) x[k] = (double)k;
  // Заполняем массив числами от 0 до 15

  p = x; // указатель p указывает на адрес нулевого элемента массива x
  printf("*p = x[0]: %.2f\n", *p);

  q = p + 4;  // q указывает на x[4]
  p++;        // p указывает на x[1]
  printf("*p = x[1]: %.2f  *q = x[4]: %.2f\n", *p, *q);
  printf("*(q - 2) = x[2]: %.2f\n", *(q - 2));
  // Выводим значение по адресу q - 2, т.е. x[2]
```

```
  printf("p == q: %d  p < q: %d\n", p == q, p < q); // сравниваем p и q
  printf("p == NULL: %d  p > NULL: %d\n", p == NULL, p > NULL);
  // Сравниваем p со значением NULL

  p = &x[2]; // p указывает на x[2]
  q = x + 8; // q указывает на x[8]
  printf("Distance between x[8] and x[2] is\n");
  printf("%ld * sizeof(double) bytes.\n", q - p); // разность указателей
  printf("p[3]:  %.2f  x[5]:  %.2f\n", p[3], x[5]);
  // Применяем индексирование к указателю p: выводим p[3], т.е. x[5]
}
```

Будет выведено:

```
*p = x[0]: 0.00
*p = x[1]: 1.00  *q = x[4]: 4.00
*(q - 2) = x[2]: 2.00
p == q: 0  p < q: 1
p == NULL: 0  p > NULL: 1
Distance between x[8] and x[2] is
6 * sizeof (double) bytes.
p[3]:  5.00  x[5]:  5.00
```

3 Приведение указателей разных типов друг к другу

Указатели разных типов можно приводить друг к другу (однако здесь требуется определенная осторожность, поскольку такие действия могут привести к ошибкам стадии исполнения). Следующая программа выполнялась на компьютере, на котором переменная типа **long** занимает 8 байт, а переменная типа **int** — 4 байта (это зависит от архитектуры и компилятора), и порядок хранения байт в переменной от младшего к старшему[5]. Эта программа выводит сначала число типа **long**, а затем его старшие 4 байта и младшие 4 байта:

```
#include <stdio.h>  // Пример 4
int main()
{
  long a = 0x1122334455667788L;  // переменная типа long, 8 байт
  long *p = &a;                  // указатель p хранит адрес переменной a
  int  *q = (int *)p;
  // Указатель q на int будет указывать на младшие 4 байта переменной a
  printf("long: %lx\n", *p);     // выводим значение переменной a типа long
  printf("high part: %x\n", *(q + 1)); // выводим ее старшие 4 байта
  printf("low part:  %x\n", *q); // выводим ее младшие 4 байта
}
```

Будет выведено:

```
long: 1122334455667788
high part: 11223344
low part:  55667788
```

[5] В архитектуре Intel целые числа хранятся от младшего байта к старшему. Например, в первом байте переменной a из примера 4 будет храниться шестнадцатеричное число 88_{16}, равное двоичному числу 10001000_2, во втором — $77_{16} = 01110111_2$, в восьмом — $11_{16} = 00010001_2$. В других архитектурах хранение целых чисел может осуществляться и в более привычном формате — от старшего к младшему. Архитектура ARM, часто используемая в мобильных устройствах, допускает оба варианта хранения целых чисел.

4 Передача параметра в функцию по указателю

Указатели используются при передаче параметров в функции в том случае, когда нужно, чтобы функция изменила значение переданного ей параметра. Иначе это сделать невозможно, потому что язык Си допускает передачу параметров только по значению, то есть функция получает копию переменной и изменение копии не сказывается на оригинале. При передаче параметра по указателю изменение значения переданного параметра становится возможным. Приведём пример функции, которая меняет местами значения своих аргументов, получая их по указателю (но сами указатели, как и обычные переменные, передаются по значению). Обратите внимание, при вызове функции используется оператор взятия адреса переменной `&`.

```
#include <stdio.h> // Пример 5

void Swap(int *q, int *r) // Функция получает два указателя на int
{
  int tmp = *q; // в tmp записываем значение, которое находится по адресу q
  *q = *r;
// В переменную, находящуюся по адресу q, записываем значение,
// находящееся по адресу r
  *r = tmp; // в переменную, находящуюся по адресу r, записываем значение tmp
}

int main()
{
  int a = 3, b = 5;
  printf ("Before: a: %d  b: %d\n", a, b);
  Swap(&a, &b); // вызываем функцию, передаём ей адреса переменных a и b
  printf("After:  a: %d  b: %d\n", a, b);
}
```

Будет выведено:

```
Before: a: 3  b: 5
After:  a: 5  b: 3
```

Передавать параметры по указателю необходимо, например, если надо передать в функцию массив. Рассмотрим программу, в которой реализована функция `average()`, возвращающая значение среднего арифметического чисел, хранящихся в массиве:

```
#include <stdio.h> // Пример 6

#define LEN 16

double average(double *w, int n)
// w - адрес передаваемого в функцию массива длиной n
{
  double s = 0.;
  int k;
  for (k = 0; k < n; k++) s += w[k]; // применяем операцию индексирования
  return s / n;
}

int main()
{
  double x[LEN];
  int k;
  for (k = 0; k < LEN; k++) x [k] = (double) k; // заполняем массив
  printf("average: %f\n", average (x, LEN));
}
```

Будет выведено:

```
average: 7.500000
```

Несколько сложнее приходится действовать, когда возникает необходимость, чтобы функция изменила сам указатель. В таком случае можно передать функции адрес указателя (указатель на указатель или адрес адреса). Реализуем функцию `increasePointer()`, которая увеличивает значение указателя на единицу, и изучим ее работу:

```
#include <stdio.h> // Пример 7

#define LEN 16

void increasePointer(double **w) // функция принимает указатель на указатель
{
  (*w)++; // изменяем значение по адресу, на который указывает w
  // Мы поставили *w в скобки, поскольку оператор инкремента ++ имеет
  // более высокий приоритет по сравнению с разыменованием указателя *
}

int main()
{
  double x[LEN];
  double *p = x;
  int k;

  for (k = 0; k < LEN; k++) x[k] = (double) k;
  printf ("Try 0: p: %p,  *p: % 6.2f\n", p, *p);

  increasePointer(&p); // передаём функции адрес указателя
  printf ("Try 1: p: %p,  *p: % 6.2f\n", p, *p);

  increasePointer(&p);
  increasePointer(&p);
  printf("Try 3: p: %p,  *p: % 6.2f\n", p, *p);
}
```

Будет выведено:

```
Try 0: p: 000000000023FDC0,  *p:   0.00
Try 1: p: 000000000023FDC8,  *p:   1.00
Try 3: p: 000000000023FDD8,  *p:   3.00
```

Обратите внимание, что при переходе от `Try 0` к `Try 1` значение указателя возрастает на 8 байт, то есть на размер переменной типа **`double`**. Аналогично переход от `Try 1` к `Try 3` соответствует возрастанию адреса на 16 — это размер двух переменных типа **`double`**.

5 Указатель на функцию

Указатель может содержать адрес не только переменной, но и функции («вызвать функцию» означает не что иное, как передать управление по адресу, начиная с которого расположен программный код функции). Поэтому можно объявить указатель на функцию и делается это следующим образом:

```
<тип_результата > (*<имя>)(<тип1 аргумент1> [, другие аргументы...]);
```

Эта запись означает указатель `<имя>` на функцию, которая возвращает значение типа `<тип_результата>`. Круглые скобки вокруг звёздочки и имени функции необходимы, без них запись будет интерпретирована как «функция, возвращающая указатель на `<тип>`». В следующих скобках указаны аргументы функции (`<аргумент1>` и т.д.) вместе с их типами (`<тип1>` и т.д.).

Рассмотрим пример объявления и использования указателя на функцию:

```
#include <stdio.h> // Пример 8

double increase(int a) { return 2.5 * a; }

double decrease(int a) { return 0.75 * a; }

double callFunc(int w, double (*f)(int))
// Второй параметр - указатель на функцию, принимающую один аргумент типа int
// и возвращающую значение типа double
{
  return (*f)(w);  // вызов функции по адресу f
  // В последних стандартах языка Си разрешено вызывать функции по указателям
  // и обычным образом, т.е. использовать f(w) вместо (*f)(w)
}

int main()
{
  int b = 2;
  double w;
  double (*funcPointer)(int) = NULL; // объявляем указатель на функцию

  funcPointer = increase; // здесь оператор взятия адреса & не записан явно
  w = callFunc(b, funcPointer);
  // Первый аргумент - переменная типа int, которая обрабатывается
  // в функции, адрес которой хранится во втором аргументе

  printf("Call 1: %.2f\n", w);
  funcPointer = &decrease; // допустимо явное использование оператора &
  w = callFunc(b, funcPointer);
  printf("Call 2: %.2f\n", w);
}
```

Будет выведено:

```
Call 1: 5.00
Call 2: 1.50
```

Обратите внимание, как в Примере 8 указатель на функцию `funcPointer` получает адрес функции `increase()` — имя функции без круглых скобок трактуется в языке Си как её адрес.

Конструкции, определяющие указатель на функцию (см. `double (*f)(int)`, которая используется в Примере 8), выглядят достаточно громоздко. При многократном использовании таких определений в коде представляется удобным объявить тип[6] — указатель на функцию, применяя ключевое слово **`typedef`**. Перепишем Пример 8, создав такой тип:

```
#include <stdio.h> // Пример 9

typedef double (*ptrFunc)(int);
// Тип - указатель на функцию с одним параметром типа int и
// возвращаемым значением типа double

double increase(int a) { return 2.5 * a; }
double decrease(int a) { return 0.75 * a; }
double callFunc(int w, ptrFunc f) { return (*f)(w); }

int main()
{
  int b = 2;
  double w;
```

[6] См. Тему 3.

```
  ptrFunc funcPointer = NULL;
  funcPointer = increase;
  w = callFunc(b, funcPointer);
  printf("Call 1: %.2f\n", w);

  funcPointer = decrease;
  w = callFunc(b, funcPointer);
  printf("Call 2: %.2f\n", w);
}
```

Иногда может быть удобно использовать массив указателей на функции. Сохранив функциональность, модифицируем Пример 9, включив в него такой массив:

```
#include <stdio.h> // Пример 10

typedef double (*ptrFunc)(int);

double increase(int a) { return 2.5 * a; }
double decrease(int a) { return 0.75 * a; }
double callFunc(int w, ptrFunc f) { return (*f)(w); }

int main()
{
  int b = 2, k;
  double w;
  ptrFunc funcPointer[2];    // создаем массив указателей на функцию
  funcPointer[0] = increase; // инициализируем массив функциями increase
  funcPointer[1] = decrease; // и decrease

  for (k = 0; k < 2; k++)
  {
    w = callFunc(b, funcPointer[k]);
    printf("Call %d: %.2f\n", k + 1, w);
  }
}
```

Отметим, что без создания пользовательского типа `ptrFunc` массив функций в Примере 10 пришлось бы определить как **`double`** `(*funcPointer [2])(`**`int`**`)`.

6 Функции работы с динамической памятью

При проведении сложных расчётов достаточно часто возникают ситуации, когда заранее неизвестно, какой объём данных потребуется обрабатывать в процессе вычислений. В этих случаях можно на этапе работы программы запросить у операционной системы дополнительный блок памяти. Это называется *выделить память динамически*. Также динамическое выделение памяти необходимо, если размер обрабатываемого массива известен, но он превышает размер стека программы, в котором, как правило, компилятор выделяет статическую память для локальных массивов.

В стандартную библиотеку языка Си включены функции для работы с динамической памятью, их прототипы объявлены в заголовочном файле `stdlib.h`:

```
      void *malloc(size_t size);
      void *calloc(size_t nmemb, size_t size);
      void *realloc(void *ptr, size_t size);
      void free(void *ptr);
```

Описание их работы представлено в Таблице 5.1.

Таблица 5.1 Функции для работы с динамической памятью

Функция	Основная функциональность	Примечания
`malloc()`	выделяет блок памяти размером `size` байтов и возвращает указатель на него. Память не инициализируется. В случае ошибки возвращается `NULL`.	Если `size` равен нулю, функция возвращает значение `NULL`.
`calloc()`	выделяет память для массива из `nmemb` элементов размером `size` каждый и возвращает указатель на него. Память обнуляется. В случае ошибки возвращается `NULL`.	Если `nmemb` либо `size` равны нулю, функция возвращает значение `NULL`.
`realloc()`	изменяет размер ранее выделенного [функцией `malloc()`, `calloc()` или `realloc()`] блока памяти, на который указывает указатель `ptr` до размера `size` байтов. Содержимое исходного блока памяти сохраняется вплоть до последнего элемента в случае, если выделено больше памяти, или вплоть до последнего, умещающегося в новый массив памяти. Если новый размер больше старого, добавленная память не будет инициализирована. В случае ошибки возвращается `NULL`, исходный блок памяти при этом остаётся неизменным.	Если `ptr` равен `NULL`, вызов `realloc()` эквивалентен вызову `malloc(size)`. Если `size` равен нулю, а `ptr` не равен `NULL`, происходит очистка исходного блока памяти, то есть вызов эквивалентен `free(ptr)`.
`free()`	освобождает место в памяти, на которое указывает `ptr`. Указатель `ptr` должен указывать на блок памяти, выделенный ранее с помощью `malloc()`, `calloc()` или `realloc()`.	Если функция вызывается повторно для уже освобождённого блока, возникает неопределённое поведение. Чтобы избежать такого поведения программы, после вызова функции `free()` разумно присвоить указателю значение `NULL`. В случае, когда `ptr` равен `NULL`, функция не делает ничего.

Функции `malloc()`, `calloc()` и `realloc()` возвращают значение типа **`void`**`*`, поэтому необходимо явное приведение типа к желаемому[7]. Однако такое приведение не требуется при передаче аргументов в функции `realloc()` и `free()`, поскольку преобразование любого указателя к **`void`**`*` осуществляется автоматически.

[7] В языке Си++ явное преобразование типа указателя является обязательным. Компилятор языка Си преобразует тип автоматически.

Рассмотрим программу, демонстрирующую применение функций `malloc()` и `free()`:

```
#include <stdio.h>  // Пример 11
#include <stdlib.h> // Данный заголовочный файл необходим для подключения
                    // функций, работающих с динамической памятью
int main()
{
  double *x;
  size_t n = 8, k;

  x = (double*) malloc(sizeof(double) * n);
  // Выделяем память вызовом malloc() под массив из n элементов типа double.
  // Одна переменная типа double имеет размер sizeof(double),
  // т.е. мы запрашиваем у операционной системы sizeof(double) * n байт.
  // (double*) используется для явного преобразования типов.

  if (x == NULL) // при ошибке выводим сообщение и завершаем программу
  {
    printf("Cannot allocate memory.\n");
    return 1; // программа завершилась некорректно
  }

  for (k = 0; k < n; k++)   // работаем с выделенной памятью
  {
    x[k] = 0.25 * k;
    printf("x [%lu]: %f\n", k, x [k]);
  }

  free(x);  // возвращаем память операционной системе
  x = NULL; // чтобы избежать неопределенного поведения программы,
            // присваиваем x значение NULL

  free(x);  // теперь вызов free(NULL) безопасен
  return 0; // программа завершилась корректно
}
```

Теперь применим функцию `calloc()` для выделения динамической памяти под целочисленный массив из n элементов. Заполним созданный массив числами Фибоначчи.

Также разработаем функцию

```
      void expand_array(int** f, size_t size_in, size_t* size_out);
```

которая:

- принимает указатель на массив `f`, из `size_in` элементов,
- используя функцию `realloc()`, изменяет размер массива до `*size_out` (если изменение размера массива невозможно, то записывает в переменную `*size_out` значение `size_in`) и
- возвращает указатель на измененный массив через тот же указатель на указатель[8] `f`.

```
#include <stdio.h> // Пример 12
#include <stdlib.h>

void expand_array(int** f, size_t size_in, size_t* size_out)
{
  int *y;
  size_t k;
```

[8] Об указателях на указатель см. раздел 4.

```
  y = (int*) realloc(*f, sizeof(int) * *size_out);
  // Изменяем размер блока памяти так, что массив y будет содержать *size_out
  // элементов, причем элементы с нулевого по седьмой совпадают с таковыми
  // у исходного массива f. Параметр size_out передается через указатель,
  // что позволяет изменять значение переменной в вызывающей функции.
  // Две звездочки перед size_out соответствуют последовательности операторов
  // умножения и разыменования. Приоритет операторов в Си таков, что сначала
  // выполняется разыменование, а потом умножение.

  if (y != NULL) *f = y;
  else *size_out = size_in;
  // В дальнейшем мы снова будем работать с массивом, который хранится по
  // адресу *f, однако содержит *size_out элементов. Если выделение памяти
  // для *size_out элементов прошло успешно, то запишем значение указателя y
  // в *f.
  // В случае ошибки (значение y равно NULL) значение указателя *f
  // не изменяется, но в переменную *size_out записывается size_in.

  for (k = size_in; k < *size_out; k++)
    (*f)[k] = (*f)[k - 2] + (*f)[k - 1];
  // Дополняем массив до *size_out чисел Фибоначчи
  // *f должно быть в скобках, поскольку приоритет оператора индексации []
  // выше, чем оператора разыменования *.
}

int main()
{
  size_t n = 8, k, m = 20;
  int *fib;
  fib = (int*) calloc(n, sizeof(int));
  // Выделяем память вызовом calloc() под массив из n = 8 элементов типа int.

  if (!fib) // равенство NULL и нуля позволяет проделывать проверку
            // на ошибки при выделении памяти и таким образом
  {
    printf ("Cannot allocate memory.\n");
    return 1; // программа завершилась некорректно
  }

  fib[0] = 0;
  fib[1] = 1;
  for (k = 2; k < n; k++)
    fib[k] = fib[k - 2] + fib[k - 1]; // заполняем массив числами Фибоначчи

  expand_array(&fib, n, &m);
  // Вызываем функцию, которая расширяет массив fib до m = 20 элементов

  for (k = 0; k < m; k++)
    printf("%2lu:  %d\n", k, fib [k]);

  free(fib); // возвращаем память операционной системе

  // free(fib);
  // Повторный вызов free() для указателя со старым значением приведёт к
  // аварийному завершению программы. Поэтому он находится под комментарием.

  return 0;  // программа завершилась корректно
}
```

Тема 6. Форматированный ввод и вывод в языке Си.

В языке программирования Си нет специальных конструкций, предназначенных для ввода и вывода (input/output, I/O). Все операции ввода и вывода производятся с помощью функций, которые находятся в стандартной библиотеке языка Си. Для их использования необходимо подключить к программе заголовочный файл:

```
#include <stdio.h>
```

Этот файл содержит все необходимые объявления для того, чтобы можно было воспользоваться возможностями ввода/вывода стандартной библиотеки.

Ввод и вывод происходят так, как будто программа работает с файлами. Ввод представляется как чтение из файла, называемого стандартным потоком (stream) ввода, вывод — как запись в файл, называемый стандартным потоком вывода. При старте программы операционная система (ОС) всегда открывает для неё три стандартных потока: `stdin` — стандартный поток ввода (по умолчанию связан с клавиатурой терминала), `stdout` — стандартный поток вывода (по умолчанию связан с экраном терминала) и `stderr` — стандартный поток сообщений об ошибках или отладочной информации (по умолчанию также связан с экраном терминала).

Операции ввода/вывода бывают буферизованные и небуферизованные (прямые). В случае небуферизованного ввода введённый символ становится доступен ожидающей программе немедленно. Если ввод буферизован, символы накапливаются в специальной области памяти — буфере и становятся доступны программе после нажатия клавиши `Enter` или же когда буфер будет полностью заполнен. При выводе ситуация аналогична: в случае небуферизованного вывода символ возникает на экране немедленно, если вывод буферизован, символы, предназначенные для вывода, накапливаются в буфере и только потом отображаются на экране. Информация из буфера будет выведена на экран (или в файл, если поток связан с файлом) либо когда в буфер поступает символ перевода строки, либо буфер заполнен, либо программа ожидает ввод.

Наличие буфера позволяет согласовывать скорости работы устройств, которые его используют. Например, если программа выводит данные по одному символу, жёсткий диск не будет успевать их записывать. Или, если при вводе данных допущена ошибка, её можно исправить до того, как будет нажата клавиша `Enter`. По умолчанию стандартные потоки ввода (`stdin`) и вывода (`stdout`) буферизованы, стандартный поток сообщений об ошибках (`stderr`) не буферизован.

Для вывода информации на экран нужно уметь преобразовывать значения переменных из их внутреннего представления, с которым имеет дело программа, в текстовую форму, с которой имеет дело человек, то есть представлять информацию в форматированном виде. А при вводе надо делать обратное преобразование. Для этого в стандартной библиотеке существуют функции форматированного ввода и вывода:

```
int scanf(const char *format, ...);
int printf(const char *format, ...);
```

Первая предназначена для ввода информации, вторая для её вывода. Эти функции принимают переменное число параметров, в качестве первого параметра должна идти строка форматирования, ввод и вывод осуществляется в соответствии с этой строкой. Функция `scanf()` возвращает количество успешно считанных и преобразованных (присвоенных переменным в её параметрах) значений, оно может быть меньше предусмотренного или даже нулём в случае ошибки чтения или преобразования.

Если до первого преобразования будет достигнут конец ввода или произойдёт ошибка, функция возвращает `EOF` (от End Of File — это константа, определённая в `stdio.h`, её значение — отрицательное число, как правило -1). Аргументами функции `scanf()` должны быть указатели на переменные, в которые будет помещена прочитанная информация.

Функция `printf()` возвращает количество выведенных символов, не включая завершающий нулевой символ. В случае возникновения ошибки функция возвращает отрицательное значение. Аргументами функции `printf()` должны быть переменные, значения которых будут выведены на экран.

Рассмотрим форматированный вывод с помощью функции `printf()`:

```
#include <stdio.h>

int main(void)
{
    int res;
    res = printf("Hello, world!\n");
    printf("res: %d\n", res);
    return 0;
}
```

Результат работы этой программы:

```
Hello, world!
res: 14
```

Первый вызов `printf()` выводит строку длиной 14 символов, последний из которых — перевод строки, второй выводит текст из строки форматирования, а на месте сочетания символов `%d` подставляет значение переменной `res`.

1 Функция форматированного вывода `printf()`

Функция `printf()` выводит в стандартный поток вывода символы строки форматирования до тех пор, пока не встретит в ней специальный символ (типа перевода строки или табуляции) или спецификацию формата для очередного параметра. Спецификация формата — это последовательность символов, начинающаяся символом `%` (процент), она имеет вид:

```
%[флаги][ширина][.точность][размер]тип
```

Обязательными в этой спецификации являются символы начала спецификации формата (`%`) и типа выводимой переменной. Символ типа определяет форму вывода переменной данного типа. В Таблице 6.1 приведены символы типа выводимой переменной.

Таблица 6.1 Символы типов в строке форматирования функции `printf()`

Тип	Форма представления
`%d` или `%i`	Целое число со знаком в десятичной системе счисления. Число выводится с правым выравниванием, знак при выводе указывается только для отрицательных чисел. Тип аргумента по умолчанию считается **`int`**.
`%u`	Беззнаковое целое число в десятичной системе счисления. Число выводится с правым выравниванием. Тип аргумента по умолчанию считается **`unsigned int`**.
`%x` или `%X`	Беззнаковое целое число в шестнадцатеричной системе счисления, `x` использует маленькие буквы (`abcdef`), `X` — большие (`ABCDEF`). Число выводится с правым выравниванием. Тип аргумента по умолчанию считается **`unsigned int`**.
`%o`	Беззнаковое целое число в восьмеричной системе счисления. Число выводится с правым выравниванием. Тип аргумента по умолчанию считается **`unsigned int`**.
`%f` или `%F`	Число с плавающей точкой в виде `[-]dd.ddd`, по умолчанию выводится 6 знаков после точки с округлением, если число по модулю меньше единицы, перед десятичной

	точкой выводится 0. Число выводится с правым выравниванием, знак указывается только для отрицательных чисел. Тип аргумента может быть **float** или **double**.
%e или %E	Число с плавающей точкой в экспоненциальной форме записи в виде [-]dd.ddde±dd, знак экспоненты будет представлен строчной или прописной буквой «e», в зависимости от использованного символа типа. По умолчанию выводится 6 знаков после точки с округлением, если число по модулю меньше единицы, перед десятичной точкой выводится 0. Число выводится с правым выравниванием, знак будет указан только для отрицательных чисел. Тип аргумента может быть **float** или **double**.
%g или %G	Число с плавающей точкой в виде, зависящем от величины числа (если значение d находится в диапазоне $10^{-5} < d < 10^{6}$, используется форма представления f или F, иначе e или E), по умолчанию выводится 6 значащих цифр числа с округлением, если число по модулю меньше единицы, перед десятичной точкой выводится 0, незначащие нули справа от десятичной точки не выводятся. Число выводится с правым выравниванием, знак указывается только для отрицательных чисел. Тип аргумента может быть **float** или **double**.
%a или %A	Число с плавающей точкой в шестнадцатеричном формате в экспоненциальной форме записи в виде [-]0xhh.hhhp±hh, показатель степени обозначается символом «p» для спецификатора a и символом «P» для спецификатора A. Число выводится с правым выравниванием, знак указывается только для отрицательных чисел. Тип аргумента может быть **float** или **double**.
%c	Вывод символа, указанного в аргументе. Тип аргумента по умолчанию считается **unsigned char**.
%s	Вывод строки до нулевого завершающего байта, на которую ссылается указатель в аргументе. Строка должна быть объявлена как **char** *.
%p	Вывод указателя. Результат зависит от архитектуры и компилятора, как правило, выводится адрес в шестнадцатеричном представлении.
%%	Вывод самого символа % (процент).

Приведём пример вывода переменных различных типов:

```
#include <stdio.h>

int main(void)
{
    int a1 = 65;
    double d1 = 0.123456789e+6;
    char c1 = 65;
    char s1[] = "A string to output.";

    /* Выводим целое число в разных системах счисления */
    printf("oct: %o  dec: %d  hex: %x\n", a1, a1, a1);

    printf("%f  %e  %g\n", d1, d1, d1); // выводим число с плавающей точкой
    printf("%d %c\n", c1, c1); // выводим char как целое и как символ
    printf("%s\n", s1);        // выводим строку
    printf("Address of a1: %p\n", &a1); // выводим указатель
    printf("Value is: %d%%\n",a1);      // выводим символ %

    return 0;
}
```

Эта программа выведет:

```
oct: 101  dec: 65  hex: 41
123456.789000  1.234568e+05  123457
65 A
A string to output.
Address of a1: 0x7ffdcf469070
Value is: 65%
```

Рассмотрим необязательные поля спецификации формата. Поля `[флаги]`, `[ширина]`, `[.точность]` и `[размер]` не являются обязательными, но во многих случаях бывают очень важны.

В поле `[флаги]` могут находиться значения, указанные в Таблице 6.2.

Таблица 6.2 Возможные значения флагов в строке форматирования `printf()`

Флаг	Назначение флага
− (минус)	Выводимое значение выравнивается по левому краю (по умолчанию выводимое поле выравнивается по правому краю).
+ (плюс)	Всегда выводить знак числа (по умолчанию выводится только знак минус перед отрицательными числами).
пробел	Выводить перед положительным числом пробел, если не указан флаг плюс.
# (решётка)	Используется альтернативная форма представления числа. При выводе чисел в шестнадцатеричной системе счисления (тип `x` или `X`) перед числом будет выведено `0x` или `0X`. При выводе чисел в восьмеричной системе счисления перед числом будет выведен `0` (ноль). При выводе чисел с плавающей точкой в представлении числа всегда будет содержаться точка. При использовании с типом `g` или `G` незначащие нули справа будут отображаться.
`0` (ноль)	Слева от выводимого значения выводятся символы `0`, чтобы дополнить ширину числа до величины, указанной в поле `[ширина]` (по умолчанию выводится символ пробел). Если для типов `d`, `i`, `o`, `u`, `x` или `X` указано поле `[.точность]`, или если указан флаг «−» (минус), то флаг `0` игнорируется.

Поле `[ширина]` определяет минимальный размер выводимого значения в символах. Если количество символов в выводимом числе меньше указанной ширины, недостающее количество символов заполняется нулями или пробелами слева или справа от числа в зависимости от значений поля `[флаги]`. Если представление числа больше ширины поля, запись выходит за его пределы. Ширина указывается либо целым числом, либо символом `*`, тогда ширина поля вывода берётся из значения аргумента целого типа, указанного перед аргументом для вывода. Если он имеет отрицательное значение, считается, что ширина равна его модулю, а спецификация формата должна быть обработана, как будто в ней выставлен флаг минус (выравнивание по левому краю).

Например, в результате вызова:

```
      printf("{%*x}{%*x}\n", 4, 0x55, -4, 0x55);
```

на экран будет выведено:

```
{  55}{55  }
```

Поле [.точность] определяет количество цифр в выводимом значении. Его действие зависит от значения поля тип:

Таблица 6.3 Действие поля [.точность] в строке форматирования printf()

Тип	Действие поля [.точность]
d, i, o, u, x, X	Определяет минимальное число выводимых цифр. Если количество цифр меньше, чем указано, то при выравнивании по правому краю число будет дополнено нулями слева.
a, A, e, E, f, F	Определяет минимальное число знаков после десятичной точки. Если в числе знаков после точки меньше, чем указано, число дополняется нулями, если больше — число выводится с указанным количеством знаков с округлением.
g и G	Определяет максимальное число выводимых цифр.
s	Определяет максимальное число выводимых символов.

Значение поля указывается символом «.» (точка) и следующим за ним целым числом или символом «*», тогда значение поля берётся из значения аргумента целого типа, указанного перед аргументом для вывода. Если этот аргумент имеет отрицательное значение, считается, что поле отсутствует.

Например, в результате вызова:

```
printf("{%.*x}{%.*f}{%.*f}\n", 4, 0x55, 2, 3.1415, -2, 3.1415);
```

на экран будет выведено:

```
{0055}{3.14}{3.141500}
```

Поле [размер] уточняет размер аргументов, переданных функции. Функция printf() принимает произвольное количество параметров и не может определить их тип и размер, поэтому информация о типе и размере должна передаваться функции явно. Действие поля [размер] зависит от значения поля тип, как это указано в Таблице 6.4.

Таблица 6.4 Действие поля [размер] в строке форматирования printf()

Поле [размер]	Назначение поля
h	Предназначено для вывода значения типа **short int** или **unsigned short int** или для приведения аргумента к одному из этих типов. Используется с типами d, i, o, u, x и X.
hh	Предназначено для вывода значения типа **char** или **unsigned char** или для приведения аргумента к одному из этих типов. Используется с типами d, i, o, u, x и X.
l	Предназначено для вывода значения типа **long int** или **unsigned long int** или для приведения аргумента к одному из этих типов. Используется с типами d, i, o, u, x и X. Также может использоваться для вывода значения с плавающей точкой,

	когда ожидается переменная типа **double**, а не **float**. В этом случае такое уточнение используется с типами a, A, e, E, f, F, g и G.
ll	Предназначено для вывода значения типа **long long int** или **unsigned long long int** или для приведения аргумента к одному из этих типов. Используется с типами d, i, o, u, x и X.
L	Предназначено для вывода значения типа **long double** или для приведения аргумента к этому типу. Используется с типами a, A, e, E, f, F, g и G.

Пример использования дополнительных полей в спецификации формата:

```
#include <stdio.h>

int main(void)
{
    int a1 = 65;
    int a2 = -65;
    double d4 = 0.123;
    long double d5 = 0.567;
    char s1[] = "A string to output.";

    /* Выводим целые числа с разными флагами */
    printf("%+d\n%d\n% d\n", a1, a2, a1);
    printf("%#x  %#o\n", a1, a1);

    /* Число с плавающей точкой */
    printf("%f  %lf\n", d4, d4);

    /* Число с плавающей точкой с флагом # */
    printf("%#f  %#e  %#g\n", d4, d4, d4);

    /* Устанавливаем поля ширина и точность */
    printf("%0*.*f\n", 6, 2, d4);

    /* Ширина, точность и выравнивание для строки */
    printf("|%.8s|\n", s1);
    printf("|%20.8s|\n", s1);
    printf("|%-20.8s|\n", s1);

    /* Переменная типа long double */
    printf("%Lf\n", d5);

    return 0;
}
```

Результат работы этой программы:

```
+65
-65
 65
0x41  0101
0.123000  0.123000
0.123000  1.230000e-01  0.123000
000.12
|A string|
|            A string|
|A string            |
0.567000
```

Количество и типы аргументов, переданных функции `printf()`, должны совпадать с указанными в строке форматирования, в противном случае возникает неопределённое поведение программы (Неопределённое поведение, undefined behavior — ситуация, когда программа синтаксически правильна, однако её поведение не определено. При этом вовсе не обязательно, что программа сразу же завершится или выдаст ошибку, но на её поведение полагаться уже нельзя).

2 Функция форматированного ввода `scanf()`

Теперь рассмотрим ввод данных с помощью функции `scanf()` и приведём пример её использования:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

int main(void)
{
    int a1;
    double d1;
    char c1;
    char s1[5];

    printf("Enter int, double, char, string: ");

    // Читаем переменные типа int, double, пропускаем пробельные символы,
    // читаем переменные типа char и char* (не более 4 символов)
    scanf(" %d %lf %c%4s", &a1, &d1, &c1, s1);

    printf("You have entered: ");
    printf("%d  %f  %c  %s\n", a1, d1, c1, s1);

    return 0;
}
```

Результат работы этой программы может выглядеть так (ввод пользователя выделен полужирным шрифтом):

```
Enter int, double, char, string: 1 2.5 A qwertyuiop
You have entered: 1  2.500000  A  qwer
```

При попытке откомпилировать эту программу без первой строки в среде Microsoft Visual Studio компилятор выдаст предупреждение:

```
Warning C4996: 'scanf': This function or variable may be unsafe. Consider using
scanf_s instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS.
```

То есть компилятор предлагает вместо функции `scanf()` использовать функцию `scanf_s()`. Функция `scanf_s()` считается более безопасной, чем `scanf()`, но она включена в стандарт как рекомендуемая и её реализуют не все производители компиляторов. Для наших целей эти тонкости не важны, поэтому эту диагностику лучше отключить, воспользовавшись макроопределением `_CRT_SECURE_NO_WARNINGS` в самом начале программы обязательно до первого подключаемого заголовочного файла.

Функция `scanf()` читает информацию из стандартного потока ввода, преобразует её в соответствии со строкой форматирования и присваивает введенные значения аргументам, чьи адреса указаны в последующих параметрах функции. Аналогично `printf()`, функция `scanf()` также принимает произвольное количество аргументов, первым из которых должна быть форматирующая строка.

Спецификации формата функции `scanf()` имеют вид, почти такой же, как для функции `printf()`:

```
%[*][ширина][размер]тип
```

Однако, символы форматирующей строки вне спецификации формата интерпретируются специальным образом, а именно: если в строке встречается любой пробельный символ (пробел, перевод строки, табуляция и т. д.) или их последовательность, то при вводе будут пропущены (проигнорированы) все пробельные символы в потоке ввода, сколько бы их ни было. При этом допускается, что пробельных символов в потоке ввода не будет вообще.

Любой другой символ, отличный от пробельного или символа `%`, употребленный в строке форматирования вне спецификаций формата, должен точно соответствовать следующему введённому символу, иначе функция `scanf()` немедленно завершится, пропустив обработку всех последующих элементов форматирующей строки.

Как и для функции `printf()`, в спецификации формата для `scanf()` обязательными являются символ начала спецификации формата (`%`) и тип очередного параметра. Поле `тип` действует аналогично описанному для функции `printf()`, но есть различия, указанные в Таблице 6.5.

Таблица 6.5 Символы типов в строке форматирования функции `scanf()`

Тип	Действие поля
`%%`	Комбинации `%%` в строке формата соответствует один входной символ `%` — то есть, требуется, чтобы в потоке ввода в этот момент находился именно этот символ. Преобразование не выполняется, присваивания не происходит, параметр не требуется.
`%[...]`	Ожидается последовательность любых символов из набора, указанного в скобках, соответствующий аргумент должен быть указателем на начало строки, достаточно длинной для хранения вводимой последовательности и завершающего нулевого символа, который будет добавлен автоматически. Для того, чтобы ожидалась последовательность любых других символов, за исключением указанных, нужно начать описание множества с символа `^`. Чтение прекращается при появлении символа не из определенного тут набора или при достижении максимальной ширины, в зависимости от того, что произойдёт раньше.
`%d`	Ожидается целое число в десятичной системе счисления со знаком, соответствующий аргумент должен быть указателем на **`int`**.
`%i`	Ожидается целое число со знаком. Если ввод начинается символами `0x` или `0X`, считается, что число записано в шестнадцатеричной системе счисления, если первым символом является `0` (ноль), число считается восьмеричным, иначе читается значение в десятичной системе счисления, аргумент должен быть указателем на **`int`**.
`%o`	Ожидается целое число в восьмеричной системе счисления без знака, соответствующий аргумент должен быть указателем на **`unsigned int`**.
`%u`	Ожидается целое число в десятичной системе счисления без знака, соответствующий аргумент должен быть указателем на **`unsigned int`**.

`%x`	Ожидается целое число в шестнадцатеричной системе счисления без знака, соответствующий аргумент должен быть указателем на **`unsigned int`**.
`%F`, `%f`, `%E`, `%e`, `%G`, `%g`	Ожидается число с плавающей точкой со знаком, соответствующий аргумент должен быть указателем на **`float`**. Учтите, что функция `scanf()` при вводе значения с плавающей точкой ведёт себя не так, как функция `printf()`. Когда `scanf()` встречает в строке форматирования спецификатор `%f`, она ожидает, что соответствующий аргумент будет указателем именно на переменную типа **`float`**. Для присваивания значения переменной типа **`double`** необходимо использовать спецификатор `%lf` (см. описание поля `[размер]` далее). Если этого не сделать, поведение программы непредсказуемо.
`%A`, `%a`	При вводе числа в десятичной системе счисления работает так же, как `%f`, но может читать и вещественные числа в шестнадцатеричной системе счисления. Например, число `5.0` может быть введено как `0x1.4p+2` (в качестве знака экспоненты при вводе должны использоваться символы `p` или `P`). При вводе переменных типа **`double`** необходимо использовать спецификацию `%la`.
`%s`	Ожидается последовательность непробельных символов, соответствующий аргумент должен быть указателем на начальный элемент символьного массива (строки), достаточно длинного для хранения входной последовательности и завершающего нулевого символа, который будет добавлен автоматически. Чтение прекращается при достижении пробельного символа или при достижении максимальной ширины, в зависимости от того, что произойдёт раньше.
`%c`	Ожидается символ, соответствующий аргумент должен быть указателем на **`char`**. Ведущие пробельные символы не подавляются, чтобы их пропустить, в форматной строке используется символ «пробел».
`%p`	Ожидается указатель, соответствующий аргумент должен быть указателем на **`void`**.

Поле `[*]` предназначено для подавления присваивания. Данные считываются, преобразуются в соответствии с полем `[тип]`, но присвоения не происходит — это нужно для того, чтобы аккуратно пропустить описанную последовательность символов в потоке ввода. Соответствующий аргумент не требуется.

Поле `[ширина]` представляет собой целое десятичное число. Функция `scanf()` прочтёт не более этого числа символов.

Поле `[размер]` определяет, на какой вариант типа данных указывает соответствующий аргумент. Действие поля `[размер]` зависит от значения поля `тип`, как это указано в Таблице 6.6.

Таблица 6.6 Действие поля `[размер]` в строке форматирования `scanf()`

Поле `[размер]`	Назначение поля
`h`	Соответствующий аргумент является указателем на **`short int`** или **`unsigned short int`**. Используется с типами `d`, `i`, `o`, `u`, `x` и `X`.
`hh`	Соответствующий аргумент является указателем на **`char`** или **`unsigned char`**. Используется с типами `d`, `i`, `o`, `u`, `x` и `X`.
`l`	При использовании с типами `d`, `i`, `o`, `u`, `x` и `X` соответствующий аргумент является указателем на **`long int`** или **`unsigned long int`**. При использовании с типами `f`, `e`, `g`, `E`, `a` соответствующий аргумент является указателем на **`double`**.
`L`	При использовании с типами `d`, `i`, `o`, `u`, `x` и `X` соответствующий аргумент является указателем на **`long long int`** или **`unsigned long long int`**. При использовании с типами `f`, `e`, `g`, `E`, `a` соответствующий аргумент является указателем на **`long double`**.

Пример использования форматирующих строк функции `scanf()`:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

int main(void)
{
    int a1, a2, a3;
    double d1;
    char c1, c2;
    char s1[256];

    /* Ввод целого в различных системах счисления */
    printf("Enter dec, oct, hex: ");
    scanf(" %i %i %i", &a1, &a2, &a3);
    printf("%d  %#o  %#x\n", a1, a2, a3);

    /* Ввод переменной типа double */
    printf("Enter double: ");
    scanf(" %lf", &d1); // при вводе ОБЯЗАТЕЛЬНО использовать %lf
    printf("%f\n", d1); // при выводе можно использовать %f

    /* Ввод переменных типа char */
    printf("Enter two char: ");
    scanf(" %c %c", &c1, &c2);
    printf("%c  %c\n", c1, c2);

    /* Ввод строки символов */
    printf("Enter string with spaces: ");
    /* Читаем строку не более 255 символов длиной, ведущие пробельные
       символы подавляются, чтение прекращается, когда в потоке
       встретится символ перевода строки '\n' */
    scanf(" %255[^\n]s", s1);
    printf("%s\n", s1);
```

```
    /* Ввод точно соответствующего символа и использование
       поля подавления присваивания */
    printf("Enter int, point, int, any symbol, int: ");
    scanf(" %d.%d%*c%d", &a1, &a2, &a3);
    printf("%d  %d  %d\n", a1, a2, a3);

    return 0;
}
```

В результате её работы на экране можно увидеть (ввод пользователя выделен полужирным шрифтом):

```
Enter dec, oct, hex: 65 0101 0x41
65  0101  0x41
Enter double: 1e+2
100.000000
Enter two char: A B
A  B
Enter string with spaces: A string with spaces.
A string with spaces.
Enter int, point, int, any symbol, int: 5.6:8
5  6  8
```

Количество и типы аргументов, переданные функциям, должны точно соответствовать строке форматирования, в противном случае может возникнуть неопределённое поведение программы или даже её аварийное завершение.

Во всех приведённых примерах мы молчаливо предполагали, что ввод информации происходит без ошибок. Теперь поговорим о том, как можно обрабатывать ошибки ввода. В приведённой ниже программе показано, как можно обработать неверный ввод пользователя:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int a1 = 12345, res = 0;
    printf("Enter int: ");
    while (1) {
        res = scanf(" %d", &a1);
        if (res == EOF) {
            /* Конец файла или ошибка ввода. */
            /* Выводим диагностическое сообщение и завершаем программу. */
            printf("\nEnd of file or input error.\n");
            return 1;
        }
        else if (res == 0) {
            /* scanf() не сумела сопоставить введённое */
            /* значение переменной типа int. */
            /* Данные остались в буфере, очищаем буфер */
            /* и повторно запрашиваем ввод. */
            int ch;
            while ((ch = getchar()) != '\n') ;
            printf("Conversion error. Reenter int: ");
        }
        else {
            /* Успешный ввод */
            printf("Success, number is %d\n", a1);
            break;
        }
    }
```

```
    /* Работа с введёнными данными */
    /* ........................... */

    return 0;
}
```

Если при вводе `scanf()` достигает конца файла (напомним, ввод представляется как чтение из файла), диалог с пользователем будет таким:

```
Enter int: ^Z
End of file or input error.
```

Чтобы имитировать условие конца файла при вводе с клавиатуры, в большинстве ОС на базе Unix надо нажать комбинацию клавиш `Ctrl+D`, в ОС Windows — `Ctrl+Z` (отображается в терминале как `^Z`).

Если пользователь допустит ошибку, диалог может быть, например, таким:

```
Enter int: q
Conversion error. Reenter int: 5
Success, number is 5
```

В случае правильного ввода диалог может быть таким:

```
Enter int: 8
Success, number is 8
```

3 Функции преобразования форматов для строк

Упомянем ещё о нескольких функциях стандартной библиотеки, которые бывают полезны. Строго говоря, они не являются функциями ввода/вывода, они преобразуют данные из их внутреннего представления в строку и наоборот, из строки во внутреннее представление, но используют те же правила преобразования, что и функции `printf()` и `scanf()`. Это функции:

```
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t sz, const char *format, ...);
int sscanf(const char *str, const char *format, ...);
```

Первые две функции делают то же самое, что и функция `printf()`, но не выводят результат в стандартный поток вывода, а помещают его в строку `str`. Функция `sprintf()` не проверяет, достаточно ли выделено под строку места, об этом должен позаботиться программист. В случае переполнения строки поведение программы непредсказуемо. Функция возвращает количество символов, помещённых в строку (исключая завершающий нулевой байт) в случае успеха и отрицательное значение в случае ошибки.

Функция `snprintf()` записывает в строку не более `sz` байт (включая завершающий нулевой байт). Она также возвращает количество символов, помещённых в строку (исключая завершающий нулевой байт), если оно не превышает величины `sz` и количество символов, которое было бы помещено в строку (исключая завершающий нулевой байт), если бы в ней было достаточно места. Таким образом, если возвращаемое значение больше или равно `sz`, вывод был усечён. Если возникла ошибка, функция возвращает отрицательное значение.

Функция `sscanf()` подобна `scanf()`, но не считывает информацию из стандартного потока ввода, а читает её из строки `str`. Возвращает она то же самое, что и функция `scanf()`.

Функции `sprintf()` и `snprintf()` удобно использовать для предварительного форматирования выводимой информации, например:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

#define LEN1 64
#define LEN2 24
```

```
int main(void)
{
    char s1[LEN1], s2[LEN2], *p;
    int res = 0, t;
    size_t len;

    /* Формируем строку для вывода, в строке s1 достаточно места */
    p = s1;
    t = sprintf(p, "char:   %lu\n", sizeof(char));
    res += t; p += t;
    t = sprintf(p, "int:    %lu\n", sizeof(int));
    res += t; p += t;
    t = sprintf(p, "double: %lu\n", sizeof(double));
    res += t;

    printf("String 1:\n%s", s1);
    printf("Characters: %d\n", res);

    /* Записываем информацию в короткую строку s2 */
    res = 0; len = LEN2; p = s2;
    t = snprintf(p, len, "char:   %lu\n", sizeof(char));
    res += t; p += t;

    /* Проверяем, сколько места в строке осталось */
    len = (len >= t ? len - t : 0);
    t = snprintf(p, len, "int:    %lu\n", sizeof(int));
    res += t; p += t;

    /* Проверяем, сколько места в строке осталось */
    len = (len >= t ? len - t : 0);
    t = snprintf(p, len, "double: %lu\n", sizeof(double));
    res += t;
    printf("String 2:\n%s\n", s2);

    res = (res < LEN2 ? res : LEN2 - 1);
    printf("Characters: %d\n", res);
    return 0;
}
```

Результат работы этой программы:

```
String 1:
char:   1
int:    4
double: 8
Characters: 30
String 2:
char:   1
int:    4
dou
Characters: 23
```

Функция `sscanf()` выполняет обратные действия. Вот как её можно использовать в одном из примеров первой темы:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <math.h>

int main(int argc, char *argv[])
{
    double r1=0, r2=0, r3=0;
    char op;
    int res;
```

```
    /* Проверяем количество аргументов командной строки */
    if ((argc - 1) != 3) {
        printf("Invalid arguments.\n");
        return 1;
    }

    /* Преобразуем первый аргумент из строки во внутреннее представление */
    res = sscanf(argv[1], "%lf", &r1);
    if (res != 1) {
        printf("'%s': conversion error.\n", argv[1]);
        return 1;
    }

    /* Читаем символ из второго аргумента */
    sscanf(argv[2], " %c", &op);

    /* Преобразуем третий аргумент из строки во внутреннее представление */
    res = sscanf(argv[3], "%lf", &r2);
    if (res != 1) {
        printf("'%s': conversion error.\n", argv [3]);
        return 1;
    }
    switch (op) {
        case '+': r3 = r1 + r2; break;
        case '-': r3 = r1 - r2; break;
        case '*': r3 = r1 * r2; break;
        case '/':
            if (fabs(r2) < 1.0E-10) {
                printf("Divide by zero.\n");
                return 1;
            }
            r3 = r1 / r2;
            break;

        default:
            printf("Unknown operator '%c'.\n", op);
            return 1;
    }

    printf("%f %c %f = %f\n", r1, op, r2, r3);

    return 0;
}
```

При запуске этой программы из командной строки:

```
calc.exe 2.5 * 3.7
```

получим:

```
2.500000 * 3.700000 = 9.250000
```

В операционной системе на базе Unix символ «*» (звёздочка) надо экранировать символом «\» (обратный слеш):

```
./a.out 2.5 \* 3.7
```

Тема 7. Работа с файлами. Бесформатный ввод и вывод в языке Си.

В языке Си существуют средства для работы с дисковыми файлами. Для того, чтобы программа могла записать информацию в файл или прочитать её из файла, необходимо:

1. Создать поток ввода или вывода (или одновременного ввода и вывода) и связать его с файлом на диске (открыть файл).
2. Провести операции чтения из потока, записи в поток или чтения/записи.
3. Закрыть поток, созданный в п. 1 (закрыть файл).

Для работы с потоками в файле `stdio.h` объявлен тип данных `FILE`, он предназначен для управления потоками ввода/вывода и представляет собой структуру, реализация которой системно зависима.

1 Функции для работы с файлами

Для открытия файла предназначена функция

```
FILE *fopen(const char *pathname, const char *mode);
```

Она открывает файл, то есть создаёт поток ввода, вывода или ввода/вывода и связывает его с файлом на диске. Если операция открытия файла прошла успешно, функция возвращает указатель на тип данных `FILE`, в случае ошибки возвращается `NULL`. Аргумент `pathname` представляет собой строку, которая является именем файла. Имя файла может быть абсолютным, например, `"c:\\users\\ivan\\file.txt"` в операционной системе (ОС) Windows, или `"/home/ivan/file.txt"` в ОС на базе Unix или относительным (относительно текущей рабочей папки), например `"../file.txt"`.

Аргумент `mode` представляет собой строку, которая указывает режим, в котором будет открыт файл. Строка должна начинаться с одного из символов `r` (read — чтение), `w` (write — запись) или `a` (append — добавление), за которым могут следовать символы `+` (плюс), `t` (text — текстовый файл) или `b` (binary — двоичный или бинарный файл). По умолчанию файлы открываются в текстовом режиме, наличие символа `t` в строке не обязательно. В Таблице 7.1 перечислены возможные режимы открытия файла.

Таблица 7.1 Обозначения режимов открытия файла

`mode`	Режим открытия файла
`r`	Открыть текстовый файл для чтения. Если файла с таким именем на диске нет, возникает ошибка открытия несуществующего файла на чтение и `fopen()` возвращает `NULL`.
`r+`	Открыть текстовый файл для чтения и записи. Если файла с таким именем на диске нет, возникает ошибка открытия несуществующего файла и `fopen()` возвращает `NULL`.
`w`	Открыть текстовый файл для записи. Если файла с таким именем на диске нет, он будет создан, если такой файл уже есть, его размер усекается до нулевой длины (то есть содержимое файла пропадает).
`w+`	Открыть текстовый файл для чтения и записи. Если файла с таким именем на диске нет, он будет создан, если такой файл уже есть, его размер усекается до нулевой длины.

a	Открыть текстовый файл для добавления (записи в конец файла). Если файла с таким именем на диске нет, он будет создан.
a+	Открыть текстовый файл для чтения и добавления (записи в конец файла). Если файла с таким именем на диске нет, он будет создан.

К строке mode можно добавить символ b. В ОС на базе Unix (вернее, в ОС, соответствующих стандарту POSIX) этот символ игнорируется, поскольку эти системы не делают различий между текстовыми и бинарными файлами, в ОС Windows работа с текстовыми и бинарными файлами производится по-разному. При записи строки в текстовый файл символ перехода на новую строку "\n" транслируется в два символа "\r\n", при чтении текстового файла концы строк преобразуются их двух символов "\r\n" в один символ "\n". При чтении или записи бинарного файла никаких преобразований не производится. Добавлять символ b имеет смысл всегда, когда программа работает с бинарными файлами, это поможет в том случае, если она будет перенесена на не-Unix ОС.

Для чтения из файла и записи в файл в стандартной библиотеке есть функции:

```
int fscanf(FILE *stream, const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
```

Они предназначены для форматированного ввода из файла и вывода в файл. От описанных в предыдущей теме функций scanf() и printf() эти функции отличаются наличием ещё одного аргумента — потока ввода или вывода stream. Функции fscanf() и fprintf() делают то же самое, что и функции scanf() и printf(), но работают с любыми потоками ввода/вывода, а не только с потоками stdin и stdout.

Для закрытия файла используется функция

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

Она разрывает связь между программой и файлом. Если файл был открыт для чтения, читать информацию из него становится невозможно. Если же он был открыт для записи, данные, которые находились в буфере, будут выведены в файл принудительно (буфер будет сброшен) и дальнейшая запись в файл становится невозможна. Если программа завершается штатно, ОС сама сбрасывает все буферы и закрывает все открытые программой файлы, в случае аварийного завершения возможна потеря не записанных к этому моменту данных, поэтому закрывать открытые файлы считается хорошим тоном. При успешном завершении функция возвращает 0, при ошибке — EOF (от End Of File — это константа, определённая в stdio.h, её значение — отрицательное число, как правило -1).

Кроме вышеперечисленных, полезным бывает использование функций:

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

Для потока вывода она инициирует вывод в указанный поток данных, находящихся к моменту её вызова в буфере. Применение этой функции к потоку ввода приводит к неопределённому поведению программы. Если вызвать функцию с аргументом NULL, будут сброшены все буферы. При успешном завершении функция возвращает 0, в случае возникновения ошибки — EOF.

```
int remove(const char *pathname);
```

Эта функция удаляет с диска файл, имя которого передано ей в аргументе pathname. При успешном завершении функция возвращает 0, в случае возникновения ошибки она возвращает значение -1.

Приведём пример записи данных в файл и чтения их из файла. Эта программа открывает файл на запись, записывает в него данные и закрывает его, затем открывает его же на чтение, читает из него данные и выводит их на экран.

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define LEN 16

int main(void)
{
    char fn[] = "data1.txt";
    FILE *fout, *fin;
    int k;
    double x;

    /* Открываем файл для записи */
    fout = fopen(fn, "w");
    if (fout == NULL) {
        printf("Cannot open output file \"%s\"\n", fn);
        return 2;
    }

    /* Записываем в него данные */
    for (k = 0; k < LEN; k++) {
        fprintf(fout, "%3d %f\n", k, sin(0.1 * k));
    }

    /* Закрываем файл */
    fclose(fout);

    /* Открываем этот же файл для чтения */
    fin = fopen(fn, "r");
    if (fin == NULL) {
        printf("Cannot open input file \"%s\"\n", fn);
        return 2;
    }

    /* Читаем из него данные */
    while (fscanf(fin, "%d%lf", &k, &x) != EOF) {
        printf("%3d %f\n", k, x);
    }
    fclose(fin); /* Закрываем файл */

    return 0;
}
```

Результат её работы:

```
  0 0.000000
  1 0.099833
  2 0.198669
...
 14 0.985450
 15 0.997495
```

(Часть вывода пропущена).

При попытке откомпилировать эту программу без первой строки в среде Microsoft Visual Studio компилятор выдаст предупреждения:

```
warning C4996: 'fopen': This function or variable may be unsafe. Consider using
fopen_s instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS.
```

```
warning C4996: 'fscanf': This function or variable may be unsafe. Consider using
fscanf_s instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS.
```

Причина их появления описана в предыдущей теме.

Как понять, что произошло в случае, когда функция вернула значение `EOF` — был достигнут конец файла (строго говоря, это не ошибка, просто больше нечего читать) или возникла ошибка ввода? Понять это можно с помощью функций стандартной библиотеки:

```
int feof(FILE *stream);
int ferror(FILE *stream);
```

Первая функция возвращает ненулевое значение в случае достижения конца файла, вторая возвращает ненулевое значение в случае возникновения ошибки. Эти функции не обращаются к файлу, они только проверяют состояние последней операции ввода/вывода.

Узнать, какая именно произошла ошибка, можно, например, с помощью функции

```
void perror(const char *s);
```

Она выводит в стандартный поток сообщений об ошибках строку `s`, двоеточие, пробел и описание последней возникшей ошибки. Заключительную часть нашей программы можно переписать с использованием этих функций:

```
while (fscanf(fin, "%d%lf", &k, &x) != EOF) {
    printf("%3d %f\n", k, x);
}

if (feof(fin)) {
    /* Достигнут конец файла */
    printf("\nEnd of file.\n");
}

if (ferror(fin)) {
    /* Произошла ошибка ввода */
    perror("Input error");
}

/* Закрываем файл */
fclose(fin);
```

При работе с файлами в режиме чтения/записи бывают полезны функции контроля текущей позиции в файле:

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

Первая функция перемещает указатель чтения/записи в файле, связанном с потоком `stream` на расстояние `offset` (измеряется в байтах) относительно позиции `whence`. Возможные значения аргумента `whence` приведены в Таблице 7.2.

Таблица 7.2 Значения отсчета позиции в функции `fseek()`

Значение `whence`	Относительно какой позиции отсчитывается смещение
`SEEK_SET`	Относительно начала файла
`SEEK_CUR`	Относительно текущей позиции указателя
`SEEK_END`	Относительно конца файла

В случае успешного вызова `fseek()` очищает индикатор достижения конца файла (если он был установлен предыдущей операцией чтения/записи), отменяет действие функции `ungetc()` для текущего потока и возвращает значение 0. В случае ошибки возвращается значение -1.

Вторая функция (`ftell()`) возвращает текущее положение указателя чтения/записи для потока `stream` в случае успешного вызова и -1 в случае ошибки.

Вызов третьей функции (`rewind()`) эквивалентен вызову

```
fseek(stream, 0L, SEEK_SET);
```

Функция `rewind()` не возвращает значения.

Приведём пример использования этих функций:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char fn[] = "data2.txt";
    char data[] = "ABCDefgh";
    FILE *f;
    long pos;
    int ch1, ch2;

    /* Открываем файл для чтения и записи */
    f = fopen(fn, "w+b");
    if (f == NULL) {
        printf("Cannot open file \"%s\"\n", fn);
        return 2;
    }

    /* Записываем в него строку текста */
    fputs(data, f);

    /* Получаем и выводим на экран текущую позицию указателя */
    pos = ftell(f);
    printf("File size: %ld bytes.\n", pos);

    /* Перемещаем указатель на начало файла */
    rewind(f);

    /* Выводим содержимое файла на экран */
    printf("Before replacement:\n");
    while ((ch1 = fgetc(f)) != EOF) {
        fputc(ch1, stdout);
    }
    printf("\n");

    /* Перемещаем указатель на нулевой байт */
    fseek(f, 0L, SEEK_SET);

    /* Считываем символ, находящийся на этой позиции */
    ch1 = fgetc(f);

    /* Перемещаем указатель на последний байт */
    fseek(f, -1L, SEEK_END);

    /* Считываем символ, находящийся на этой позиции */
    ch2 = fgetc(f);
```

```
    /* Меняем символы местами */
    fseek(f, 0L, SEEK_SET);
    fputc(ch2, f);
    fseek(f, -1L, SEEK_END);
    fputc(ch1, f);

    /* Перемещаем указатель на начало файла */
    rewind(f);

    /* Выводим содержимое файла на экран */
    printf("After replacement:\n");
    while ((ch1 = fgetc(f)) != EOF) {
        fputc(ch1, stdout);
    }
    printf("\n");

    /* Закрываем файл */
    fclose(f);

    return 0;
}
```

Эта программа открывает файл для чтения и записи, пишет в него строку символов, выводит содержимое файла на экран, меняет местами первый и последний символы, снова выводит содержимое файла на экран и закрывает файл. Результат её работы:

```
File size: 8 bytes.
Before replacement:
ABCDefgh
After replacement:
hBCDefgA
```

2 Бесформатный ввод и вывод

До сих пор мы говорили о форматированном вводе и выводе. Иногда возникают ситуации, в которых достаточно читать и выводить информацию непосредственно в бинарном машинном представлении или в виде символов строк или просто потока байтов, то есть пользоваться функциями бесформатного ввода/вывода. Для чтения и записи блоков данных в машинном представлении предназначены функции:

```
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);
```

Тип `size_t` определяется как целочисленный тип без знака и широко используется для представления размера или длины объектов. Первая функция записывает `nmemb` элементов данных длиной `size` байт каждый из буфера `ptr` в поток `stream`. Возвращается количество записанных элементов (оно равно числу записанных байт, если аргумент `size` равен 1), в случае ошибки это количество меньше `nmemb` (или ноль).

Вторая функция читает `nmemb` элементов данных длиной `size` байт каждый из потока `stream` и помещает их в буфер `ptr`. Возвращается количество прочитанных элементов, если оно меньше `nmemb`, был достигнут конец файла или произошла ошибка чтения. Чтобы определить, что произошло — ошибка или был достигнут конец файла, надо использовать функции `feof()` и `ferror()`. Пример использования всех этих функций:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

```c
#define LEN 16
#define LEN2 7

int main(void)
{
    char fn[] = "data3.bin";
    double x[LEN], y[LEN2];
    int k;
    size_t res, m;
    FILE *f;

    /* Открываем файл для записи */
    f = fopen(fn, "wb");
    if (f == NULL) {
        printf("Cannot open file \"%s\"\n", fn);
        return 2;
    }

    for (k = 0; k < LEN; k++) {
        x[k] = sin(0.1 * k);
    }

    /* Пишем в него данные */
    res = fwrite(x, sizeof(double), LEN, f);
    printf("%lu items written.\n", res);
    fclose(f);

    /* Открываем файл для чтения */
    f = fopen(fn, "rb");
    if (f == NULL) {
        printf("Cannot open file \"%s\"\n", fn);
        return 2;
    }

    /* Читаем данные блоками */
    res = fread(y, sizeof(double), LEN2, f);
    while (res > 0) {
        printf("%lu items read:\n", res);
        for (m = 0; m < res; m++) {
            printf("%f ", y [m]);
        }
        printf("\n");
        res = fread(y, sizeof(double), LEN2, f);
    }

    /* Проверяем, достигнут конец файла или произошла ошибка */
    if (feof(f)) printf("End of file.\n");
    if (ferror(f)) perror("Read error");
    fclose(f); /* Закрываем файл */

    return 0;
}
```

Программа записывает блок данных в файл и затем считывает их из файла блоками меньшего размера. В конце работы проверяется состояние потока. Результат её работы:

```
16 items written.
6 items read:
0.000000 0.099833 0.198669 0.295520 0.389418 0.479426
6 items read:
0.564642 0.644218 0.717356 0.783327 0.841471 0.891207
4 items read:
0.932039 0.963558 0.985450 0.997495
End of file.
```

Рассмотрим функции строкового и символьного ввода/вывода. Самыми простыми из них являются функции односимвольного ввода и вывода:

```
int getchar(void);  // ввод символа
int putchar(int c); // вывод символа
```

Первая читает символ из стандартного потока ввода. Функция читает символ как **unsigned char** (беззнаковый тип), но возвращает его, приведённый к типу **int** (знаковый тип). Так сделано потому, что в случае достижения конца файла или возникновения ошибки функция возвращает значение `EOF`, что гарантирует его несовпадение с кодом любого обычного символа. Вторая выводит символ, приведённый к типу **unsigned char**, в стандартный поток вывода и возвращает его же, приведённый к типу **int** или значение `EOF` в случае возникновения ошибки.

Приведём пример использования этих функций:

```
#include <stdio.h>

int main(void)
{
    int c;
    while ((c = getchar()) != EOF) {
        putchar(c);
    }
    return 0;
}
```

Эта программа читает символы из стандартного потока ввода и выводит их в стандартный поток вывода. Чтобы имитировать условие конца файла при вводе с клавиатуры, в большинстве ОС на базе Unix надо нажать комбинацию клавиш `Ctrl+D`, в ОС Windows — `Ctrl+Z` (отображается в терминале как `^Z`). Вводимые символы накапливаются во входном буфере и после нажатия клавиши `Enter` поступают на вход программы, читаются функцией `getchar()` и выводятся на экран функцией `putchar()`.

Диалог с пользователем может быть примерно таким (ввод пользователя выделен полужирным шрифтом):

```
Line 1.<Enter>
Line 1.
String 2.<Enter>
String 2.
[^Z]
```

Кроме `getchar()` и `putchar()` в стандартной библиотеке есть ещё функции, которые работают с одиночными символами и позволяют их читать или записывать в произвольный файл:

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

Эти функции считывают символ из потока `stream` и возвращают его как **unsigned char**, приведённый к типу **int**. В случае достижения конца файла или возникновения ошибки возвращается значение `EOF`. Обе эти функции эквивалентны, за исключением того, что `getc()` может быть реализована как макроопределение. Для вывода символов существуют функции:

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

Они выводят символ, приведённый к типу **unsigned char**, в поток `stream`. Если вывод прошёл успешно, функции возвращают выведенный символ, приведённый к типу **int**, в случае ошибки возвращается `EOF`. Функции эквивалентны, за исключением того, что `putc()` может быть реализована как макроопределение.

Приведём пример использования этих функций:

```
#include <stdio.h>

int main(void)
{
    int c;
    while ((c = fgetc(stdin)) != EOF) {
        fputc(c, stdout);
    }
    return 0;
}
```

Эта программа эквивалентна предыдущей, за исключением того, что функции `getchar()` и `putchar()` работают только со стандартными потоками ввода и вывода соответственно, а функциям `fgetc()` и `fputc()` потоки надо указать явно.

Для работы со строками существуют функции чтения

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

и записи строк

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

Функция `fgets()` читает не более `size-1` символов из потока `stream`. Чтение завершается, когда будет достигнут конец файла или из потока будет прочитан символ перехода на новую строку (`'\n'`) или прочитано `size-1` символов, в зависимости от того, что произойдёт раньше. Если прочитан символ перехода на новую строку, он сохраняется в строке. К строке добавляется завершающий нулевой байт. Функция возвращает указатель на прочитанную строку в случае успеха и `NULL` в случае возникновения ошибки или достижения конца файла в момент, когда не было прочитано ни одного символа.

Функция `fputs()` выводит строку в поток `stream` (без завершающего нулевого байта). Функция `puts()` выводит строку, дополняя её символом перехода на новую строку, в стандартный поток вывода. Обе функции возвращают неотрицательное значение, если вывод успешен и `EOF` в случае возникновения ошибки.

Пример использования функций строкового ввода/вывода:

```
#include <stdio.h>

#define LEN 256

int main(void)
{
    char s[LEN];
    while (fgets(s, LEN - 1, stdin) != NULL) {
        fputs(s, stdout);
    }
    if (ferror(stdin)) {
        perror("Input error");
    }
    return 0;
}
```

Диалог с пользователем может быть примерно таким:

```
string #1.
string #1.
string #2.
string #2.
^Z
```

Иногда бывает полезна функция

```
      int ungetc(int ch, FILE *stream);
```

Она помещает символ с кодом `ch`, приведённый к типу **`unsigned char`**, обратно в поток ввода после чтения из него, откуда этот символ может быть прочитан следующей операцией чтения, гарантируется возврат только одного символа. Возвращаемое значение функции — помещённый в поток символ в случае успеха или `EOF` в случае ошибки.

Пример использования функции `ungetc()`:

```
#include <stdio.h>

int main(void)
{
    int c, ret = 65, k = 0;
    while ((c = fgetc(stdin)) != EOF) {
        k++;
        if (k % 3 == 1) ungetc(ret, stdin);
        fputc(c, stdout);
    }

    return 0;
}
```

Результат работы этой программы может быть таким:

```
qwert
qAweArtA
^Z
```

Стандартные потоки ввода и вывода могут быть переадресованы и связаны с дисковыми файлами средствами ОС. Например, программу посимвольного чтения из стандартного потока ввода с помощью функции `fgetc()` и записи в стандартный поток вывода с помощью `fputc()` (пример приведён выше) можно запустить из командной строки следующим образом:

```
./a.out < infile.txt > outfile.txt
```

в ОС на базе Unix или

```
a.exe < infile.txt > outfile.txt
```

в ОС Windows. Стандартный поток ввода будет связан с файлом `infile.txt` (файл должен существовать), стандартный поток вывода связывается с файлом `outfile.txt` (если такого файла нет, он будет создан, если есть, его размер будет усечён до нулевой длины). При таком запуске программа просто скопирует входной файл.

Тема 8. Библиотеки для работы с символами и строками.

Символьная и строковая информация в программировании используется достаточно часто. В стандартной библиотеке языка Си для работы с ней существует множество функций.

1 Работа с одиночными символами

В заголовочном файле `ctype.h` объявлены прототипы функций, которые работают с символами. Они проверяют, попадает ли их аргумент в определённый класс символов. Аргумент должен быть значением типа **`unsigned char`** или `EOF` (`EOF` имеет отрицательное значение для несовпадения с кодом любого символа, именно поэтому все описанные ниже функции для работы с символами принимают значение типа **`int`**). Для того, чтобы функции корректно обрабатывали русские символы, при работе в операционной системе Microsoft Windows необходимо предварительно выполнить региональные настройки (locale — их настройка была описана в первой теме): `setlocale(LC_ALL, "rus");`

Таблица 8.1 Основные функции для работы с отдельными символами

Функция	Что функция делает
`int isalpha(int c);`	Проверяет, является ли символ буквой.
`int isblank(int c);`	Проверяет, является ли символ пробелом или символом табуляции.
`int isdigit(int c);`	Проверяет, является ли символ десятичной цифрой.
`int isxdigit(int c);`	Проверяет, является ли символ шестнадцатеричной цифрой (символом из набора 0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F).
`int isalnum(int c);`	Проверяет, является ли символ буквой или десятичной цифрой. Эквивалентна вызову `isalpha(c) || isdigit(c).`
`int islower(int c);`	Проверяет, является ли символ строчной буквой.
`int isupper(int c);`	Проверяет, является ли символ заглавной буквой.
`int isprint(int c);`	Проверяет, является ли символ печатным (включая пробельные символы).
`int isgraph(int c);`	Проверяет, является ли символ графическим, то есть, отображаемым при печати и при этом — не пробельным.
`int ispunct(int c);`	Проверяет, является ли символ символом пунктуации.
`int isspace(int c);`	Проверяет, является ли символ пробельным (обычно это пробел, `'\f'`, `'\n'`, `'\r'`, `'\t'`, `'\v'`).

Все функции возвращают ненулевое значение (не обязательно единицу!), если проверяемый символ попадает в соответствующий класс символов и ноль в противном случае.

И есть ещё две функции, которые преобразуют символы из одного регистра в другой:

```
int toupper(int c);
```

Если переменная `c` содержит строчную букву, то функция возвращает её эквивалент в верхнем регистре, иначе возвращает сам символ.

```
int tolower(int c);
```

Если переменная `c` содержит заглавную букву, то функция возвращает её эквивалент в нижнем регистре, иначе возвращает сам символ.

Пример использования некоторых из этих функций:

```
#include <stdio.h>
#include <ctype.h>
#include <locale.h>

int main(void)
{
    int s[] = {'A', (unsigned char) 'ё', '!', '5'};
    int n, k;
    setlocale(LC_ALL, "rus");

    n = sizeof(s) / sizeof(int);
    for (k = 0; k < n; k++) {
        printf("Symbol '%c': ", s [k]);
        if (!isalpha(s[k])) fputs("not ", stdout);
        fputs("is letter, ", stdout);
        if (!isdigit(s[k])) fputs("not ", stdout);
        fputs("is digit, ", stdout);
        if (!ispunct(s[k])) fputs("not ", stdout);
        fputs("is punctuation char.\n", stdout);
    }

    printf("Lowercase of '%c' is '%c'\n", s[0], tolower(s[0]));
    printf("Uppercase of '%c' is '%c'\n", s[1], toupper(s[1]));
    return 0;
}
```

Результат работы этой программы:

```
Symbol 'A': is letter, not is digit, not is punctuation char.
Symbol 'ё': is letter, not is digit, not is punctuation char.
Symbol '!': not is letter, not is digit, is punctuation char.
Symbol '5': not is letter, is digit, not is punctuation char.
Lowercase of 'A' is 'a'
Uppercase of 'ё' is 'Ё'
```

2 Работа со строками

В заголовочном файле `string.h` объявлены прототипы функций для работы со строками.

Функция, вычисляющая длину строки:

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

Вычисляет длину переданной строки, завершающий нулевой байт в длину не включается. Возвращает количество символов в строке. Пример её использования:

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

/* Пример использования функции strlen() (хотя такую простую */
/* функцию несложно написать самостоятельно) */

size_t strLength(const char *s)
{
    size_t k = 0;
    while (s[k] != 0) k++;
    return k;
}
```

```
int main(void)
{
    char s1[] = "String to count string length.";
    printf("strlen: %lu\n", strlen(s1));
    printf("strLength: %lu\n", strLength(s1));
    return 0;
}
```

Результат работы программы:

```
strlen: 30
strLength: 30
```

Функции копирования строк:

```
      char *strcpy(char *dest, const char *src);

      char *strncpy(char *dest, const char *src, size_t n);
```

Функция `strcpy()` копирует строку, расположенную по адресу `src`, включая завершающий нулевой байт, в буфер `dest`. Приёмный буфер должен быть достаточно большим, чтобы исходная строка поместилась в него. В случае переполнения буфера поведение программы непредсказуемо, чаще всего в таких случаях программа аварийно завершается. Функция `strncpy()` копирует не более `n` байт исходной строки. Если среди её первых `n` байт нет нулевого байта, строка в буфере-приёмнике не будет заканчиваться нулевым байтом. Если длина строки-источника меньше `n` байт, функция дописывает дополнительные нулевые байты в приёмный буфер. Обе функции возвращают указатель на результирующую строку.

Пример их использования:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <string.h>

#define LEN1 32
#define LEN2 7

int main (void)
{
    char s1[LEN1] = "String to copy.";
    char d1[LEN1], d2[LEN1], char d3[LEN2];

    /* Копируем строку в буфер, в котором достаточно места */
    strcpy(d1, s1);
    printf("strcpy: %s\n", d1);

    /* Исходная строка короче, чем ограничитель количества */
    /* байт для копирования, нулевой символ добавляется */
    strncpy(d2, s1, LEN1);
    printf("strncpy 1: %s\n", d2);

    /* Исходная строка длиннее, чем ограничитель количества */
    /* байт для копирования, нулевой символ не добавляется */
    strncpy(d3, s1, LEN2 - 1);

    /* Добавляем нулевой символ */
    d3[LEN2 - 1] = 0;

    printf("strncpy 2: %s\n", d3);

    return 0;
}
```

Результат работы программы:

```
strcpy: String to copy.
strncpy 1: String to copy.
strncpy 2: String
```

При попытке собрать вышеприведенную программу без первой строки в среде Microsoft Visual Studio будет выдано предупреждение или ошибка:

```
error C4996: 'strcpy': This function or variable may be unsafe. Consider using
strcpy_s instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS.
```

Ее причина заключается в том, что некоторое время назад в стандартную библиотеку языка Си были включены более безопасные версии (с дополнительными параметрами, контролирующими переполнение буфера) функций работы со строками (они обозначены суффиксом «_s» в имени) и в стандарте прямо декларируется, что «классические» версии этих функций более не рекомендованы к использованию. Для наших целей эти тонкости не важны, поэтому эту диагностику лучше отключить, для чего определить `_CRT_SECURE_NO_WARNINGS` в самом начале программы (до первого включаемого заголовочного файла!), либо зайдя в настройки проекта и указав для всех конфигураций сборки проекта в графе C/C++ → Preprocessor → Preprocessor definitions константу `_CRT_SECURE_NO_WARNINGS`.

Функции объединения строк:

```
char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src, size_t n);
```

Функция `strcat()` добавляет строку `src` к строке `dest` (перезаписывая её завершающий нулевой байт) и добавляет к результату завершающий нулевой байт. В приёмном буфере должно быть достаточно места. В случае переполнения буфера поведение программы непредсказуемо. Функция `strncat()` использует не более `n` байт из `src`, если `src` содержит `n` и более байт, она не обязательно должна заканчиваться нулевым байтом. В этом случае функция дописывает к `dest` `n` байт из `src` и добавляет завершающий нулевой байт, поэтому размер приёмного буфера должен быть как минимум `strlen(dest) + n + 1`. Обе функции возвращают указатель на результирующую строку.

Пример их использования:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <string.h>

#define LEN1 32
#define LEN2 17

int main(void)
{
    char s1[LEN1] = " + second string.";
    char d1[LEN1] = "First string";
    char d2[LEN1] = "First string";
    char d3[LEN2] = "First string";
    size_t n;

    /* Объединяем строки в буфере, в котором достаточно места */
    strcat(d1, s1);
    printf("strcat: %s\n", d1);

    /* Исходная строка короче, чем ограничитель количества */
    /* байт для копирования, места в буфере достаточно */
    strncat(d2, s1, LEN1);
    printf("strncat 1: %s\n", d2);
```

```
    /* Приёмный буфер короткий, вычисляем, сколько байт можно добавить */
    n = LEN2 - strlen (d3) - 1;
    strncat(d3, s1, n);
    printf("strncat 2: %s\n", d3);

    return 0;
}
```

Результат работы программы:

```
strcat: First string + second string.
strncat 1: First string + second string.
strncat 2: First string + s
```

Функции сравнения строк:

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

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

Функция `strcmp()` сравнивает две строки и возвращает значение, меньшее нуля, если первая строка меньше второй, ноль, если строки равны и значение, большее нуля, если первая строка больше второй. Под «меньше» и «больше» имеется в виду лексикографическое сравнение строк, то есть меньше будет та строка, которая раньше записывается в словаре. Например, строка `"alpha"` больше строки `"aleph"`. Функция `strncmp()` аналогична функции `strcmp()`, за исключением того, что она сравнивает не более `n` первых символов строк.

Пример их использования:

```
#include <stdio.h>
#include <string.h>
/* Вспомогательная функция, выводит не более, чем n */
/* символов строки в кавычках и завершает вывод символом ch */
void printStr(char *s, size_t n, char ch)
{
    size_t k = 0;
    putchar('"');
    for (k = 0; k < n && s [k] != 0; k++) {
        putchar(s[k]);
    }
    putchar('"'); putchar(ch);
}
/* Вспомогательная функция, выводит результат сравнения в текстовой форме */
void printRes(int res)
{
    if (res < 0) printf("less than ");
    else {
        if (res > 0) printf("greater than ");
        else         printf("equal to ");
    }
}

int main(void)
{
    char s1[] = "Brown";
    char s2[] = "Brownie";
    char s3[] = "Brow";
    int res;

    /* Сравниваем две строки */
    res = strcmp(s1, s2);
    printf("strcmp: ");
    printf("\"%s\" ", s1);
    printRes(res);
    printf("\"%s\".\n", s2);
```

```
    /* Сравниваем не более 5 первых символов строк */
    res = strncmp(s1, s2, 5);
    printf("strncmp 1: ");
    printStr(s1, 5, ' ');
    printRes(res);
    printStr(s2, 5, '.');
    printf("\n");

    /* Сравниваем не более 5 первых символов строк */
    res = strncmp(s2, s3, 5);
    printf("strncmp 2: ");
    printStr(s2, 5, ' ');
    printRes(res);
    printStr(s3, 5, '.');
    printf("\n");

    return 0;
}
```

Результат работы программы:

```
strcmp: "Brown" less than "Brownie".
strncmp 1: "Brown" equal to "Brown".
strncmp 2: "Brown" greater than "Brow".
```

Функции поиска символа в строке:

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

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

Функция `strchr()` возвращает указатель на первое вхождение символа в строке, функция `strrchr()` возвращает указатель на последнее вхождение символа в строке. Если символ не найден, функции возвращают `NULL`. Завершающий нулевой байт считается частью строки, так что если ищется символ `'\0'`, функции возвращают указатель на него.

Пример их использования:

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char s1[] = "A string to search for a character in it.";
    int ch = 'i';
    char *res;
    /* Ищем символы в строке */
    res = strchr(s1, ch);
    if (res != NULL) {
        printf("strchr {'%c'}: %s\n", ch, res);
    }
    res = strrchr(s1, ch);
    if (res != NULL) {
        printf("strrchr {'%c'}: %s\n", ch, res);
    }
    /* Ищем последовательно все вхождения символа в строку */
    ch = 'a';
    res = strchr(s1, ch);
    while (res != NULL) {
        printf("{'%c'}: %s\n", ch, res);
        res = strchr(res + 1, ch);
    }

    return 0;
}
```

Результат работы программы:

```
strchr {'i'}: ing to search for a character in it.
strrchr {'i'}: it.
{'a'}: arch for a character in it.
{'a'}: a character in it.
{'a'}: aracter in it.
{'a'}: acter in it
```

Функция поиска подстроки в строке:

```
      char *strstr(const char *haystack, const char *needle);
```

Функция находит первое вхождение подстроки `needle` в строке `haystack`. Завершающие нулевые байты не сравниваются. Возвращает указатель на начало найденной подстроки или `NULL`, если подстрока не найдена.

Пример её использования:

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char s1[] = "A string to search for a substring in it.";
    char s2[] = "string";
    char *res;
    /* Ищем последовательно все вхождения подстроки в строку */
    res = strstr(s1, s2);
    while (res != NULL) {
        printf("{\"%s\"}: %s\n", s2, res);
        res = strstr(res + 1, s2);
    }

    return 0;
}
```

Результат работы программы:

```
{"string"}: string to search for a substring in it.
{"string"}: string in it.
```

Функции поиска сегмента строки по условию:

```
      size_t strspn(const char *s, const char *accept);

      size_t strcspn(const char *s, const char *reject);
```

Функция `strspn()` находит длину максимального начального сегмента строки `s`, состоящего только из символов, находящихся в строке `accept`. Функция `strcspn()` находит длину максимального начального сегмента строки `s`, состоящего только из символов, не находящихся в строке `reject`.

Пример их использования:

```
#include <stdio.h>
#include <string.h>
/* Вспомогательная функция, выводит не более, чем n */
/* символов строки в кавычках */
void printStr(char *s, size_t n)
{
    size_t k = 0;
    putchar('"');
    for (k = 0; k < n && s [k] != 0; k++) {
        putchar(s [k]);
    }
    putchar('"'); putchar('\n');
}
```

```
int main(void)
{
    char s1[] = "0123456789ABCDEF";
    char accept[] = "052341";
    char reject[] = "FADE";
    size_t len;

    len = strspn(s1, accept);
    printf("accept: {\"%s\"}: %lu bytes, ", accept, len);
    printStr(s1, len);

    len = strcspn(s1, reject);
    printf("reject: {\"%s\"}: %lu bytes, ", reject, len);
    printStr(s1, len);

    return 0;
}
```

Результат работы программы:

```
accept: {"052341"}: 6 bytes, "012345"
reject: {"FADE"}: 10 bytes, "0123456789"
```

Функция поиска символа из набора в строке:

```
      char *strpbrk(const char *s, const char *accept);
```

Функция находит первое вхождение любого символа из набора в `accept` в строке `s` и возвращает указатель на него. Если ничего не найдено, функция возвращает `NULL`.

Пример её использования:

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

int main(void)
{
    char s1[] = "A string to search for a char in it.";
    char accept[] = "wcty";
    char *res;
    /* Ищем последовательно все вхождения символов из набора в строку */
    res = strpbrk(s1, accept);
    while (res != NULL) {
        printf("{\"%s\"}: %s\n", accept, res);
        res = strpbrk(res + 1, accept);
    }

    return 0;
}
```

Результат работы программы:

```
{"wcty"}: tring to search for a char in it.
{"wcty"}: to search for a char in it.
{"wcty"}: ch for a char in it.
{"wcty"}: char in it.
{"wcty"}: t.
```

Функция, разбивающая строку на лексемы (в лингвистике лексема — это слово как основная единица языка, в программировании лексема — последовательность допустимых символов языка программирования, имеющая смысл для компилятора):

```
      char *strtok(char *str, const char *delim);
```

Функция разбивает строку `str` на лексемы, используя разделители, содержащиеся в `delim`. После вызова строка изменяется. Для поиска следующей лексемы в качестве первого параметра передаётся `NULL`. Функция возвращает указатель на первую найденную лексему, если ничего не найдено, возвращается `NULL`.

Пример её использования:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <string.h>

int main(void)
{
    char s1[] = "Ave, Caesar, morituri te salutant!";
    char delim[] = " ,!";
    char *res;

    /* Ищем последовательно все вхождения лексем в строку */
    res = strtok(s1, delim);
    while (res != NULL) {
        printf("%s\n", res);
        res = strtok(NULL, delim);
    }
    return 0;
}
```

Результат работы программы:

```
Ave
Caesar
morituri
te
salutant
```

Функция, создающая копию строки:

```
      char *strdup(const char *s);
```

Функция создаёт копию строки и возвращает указатель на неё. В случае ошибки возвращается `NULL`. Память для копии получается с помощью вызова функции `malloc()` и может быть освобождена с помощью вызова функции `free()`. Пример её использования:

```
#define _CRT_NONSTDC_NO_DEPRECATE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char s1[] = "String to duplicate.";
    char *p;

    p = strdup(s1);
    if (p == NULL) {
        perror("strdup");
        return 1;
    }
    p[7] = 'T';
    p[8] = 'O';
    printf("Original string: %s\n", s1);
    printf("Copied string:   %s\n", p);
    free(p);
    p = NULL;
    return 0;
}
```

Результат работы программы:

```
Original string: String to duplicate.
Copied string:   String TO duplicate.
```

Функция, заполняющая область памяти:

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

Функция заполняет байтом `c` первые `n` байт области памяти, на которую указывает `s`, и возвращает указатель на эту область. Пример её использования:

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char s1[] = "This string will be changed.";
    memset(s1, 'A', 19);
    printf("%s\n", s1);
    return 0;
}
```

Результат работы программы:

```
AAAAAAAAAAAAAAAAAAA changed.
```

Функция, копирующая область памяти:

```
      void *memmove(void *dest, const void *src, size_t n);
```

Функция копирует `n` байт из области памяти `src` в область `dest`. Области памяти могут перекрываться (копирование происходит через промежуточный буфер). Функция возвращает указатель на `dest`.

Пример её использования:

```
#include <stdio.h>
#include <string.h>
int main(void)

{
    char s[] = "0123456789";
    printf("Before: %s\n", s);
    memmove(&s[4], &s[2], 5);
    printf("After:  %s\n", s);
    return 0;
}
```

Результат работы программы:

```
Before: 0123456789
After:  0123234569
```

Кроме перечисленных выше функций работы со строками есть ещё не входящие в стандарт и реализованные не всеми производителями компиляторов функции:

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

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

Они делают то же самое, что и функции `strcmp()` и `strncmp()`, но при сравнении не учитывают регистр символов.

```
      char *strset(char *s, int c);

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

Приведенные выше две функции заполняют строку указанным символом (похоже на работу функции `memset()`).

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

Эти две функции переводят все символы строки в нижний или верхний регистр.

Функция

```
char *strrev(char *s);
```

меняет порядок следования символов строки на противоположный (разворачивает строку задом наперёд).

3 Функции преобразования чисел в строки и строк в числа

Также полезными могут оказаться функции, определённые в заголовочном файле `stdlib.h`:

```
int atoi(const char *nptr);
long atol(const char *nptr);
double atof(const char *nptr);
```

Они преобразуют начальную часть строки, на которую указывает `nptr`, в число. Первая функция возвращает значение типа **`int`**, вторая — **`long int`**, третья — **`double`**. Если преобразовать строку в число не удалось, функции возвращают нулевое значение.

Пример их использования:

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

int main(void)
{
    char s1[] = "123.45ab678";
    char s2[] = "ab123";
    int a1;
    long int a2;
    double d1;

    /* Преобразуем начальную часть строки в число типа int */
    a1 = atoi(s1);
    /* типа long int */
    a2 = atol(s1);
    /* типа double */
    d1 = atof(s1);

    printf("int:      %d\n", a1);
    printf("long int: %ld\n", a2);
    printf("double:   %f\n", d1);

    /* Начальная часть строки не может быть */
    /* преобразована в число, результат равен 0 */
    a1 = atoi(s2);
    d1 = atof(s2);
    printf("int:      %d\n", a1);
    printf("double:   %f\n", d1);
    return 0;
}
```

Результат работы программы:

```
int:      123
long int: 123
double:   123.450000
int:      0
double:   0.000000
```

Для обратного преобразования чисел в строки рекомендуется использовать функции `sprintf()` или `snprintf()`.

Кроме них существуют не включённые в стандарт и, соответственно, поддерживаемые не всем компиляторами функции:

```
void* itoa(int val, char *buf, int radix);
```

Функция переводит число `val` типа **`int`** в его строковое представление по основанию системы счисления `radix` (должно быть в диапазоне от 2 до 36: 10 десятичных цифр и 26 латинских букв).

```
char *ltoa(long val, char *buf, int radix);
```

Функция делает то же, что и `itoa()`, но работает с переменной типа **`long`**.

```
char *gcvt(double val, int precision, char *buf);
```

Функция переводит число `val` типа **`double`** в его строковое представление с округлением. Параметр `precision` определяет число цифр в строке.

В зависимости от реализации функции возвращают либо указатель на `buf`, либо указатель на завершающий нулевой символ буфера, в результате чего можно вычислить длину получившейся строки.

Пример их использования:

```
#define _CRT_NONSTDC_NO_DEPRECATE // При работе в среде Microsoft Visual
#define _CRT_SECURE_NO_WARNINGS   // Studio следует добавить эти строки.
#include <stdio.h>
#include <stdlib.h>

#define LEN 256

int main(void)
{
    int a1 = 12345;
    long int a2 = 0x11223344;
    double d1 = 123.456;
    char buf[LEN];

    /* Переводим число в его строковое представление */
    itoa(a1, buf, 10);
    printf("buf: %s\n", buf);

    ltoa(a2, buf, 16);
    printf("buf: %s\n", buf);

    gcvt(d1, 5, buf);
    printf("buf: %s\n", buf);

    return 0;
}
```

Результат работы программы:

```
buf: 12345
buf: 11223344
buf: 123.46
```

При работе в ОС Microsoft Windows часто возникают проблемы при вводе из консоли и выводе в консоль информации на русском языке. Один из возможных способов решения этой проблемы состоит в использовании системно-зависимых функций `SetConsoleCP()` и

`SetConsoleOutputCP()` (при этом шрифт, которым выводится текст в консоль, обязательно должен быть Lucida Console):

```
#include <stdio.h>
#include <locale.h>
#include <windows.h>

#define LEN 256

int main(void)
{
    char buf[LEN];

    /* Устанавливаем русскую региональную настройку */
    setlocale(LC_ALL, "rus");

    /* Устанавливаем кодовую страницу вывода в кодировку CP1251 */
    SetConsoleOutputCP(1251);

    /* Устанавливаем кодовую страницу ввода в кодировку CP1251 */
    SetConsoleCP(1251);

    printf ("Введите строку: ");
    /* Запрашиваем и выводим строку */
    fgets(buf, LEN, stdin);
    printf("Строка: %s\n", buf);

    return 0;
}
```

Результат работы программы (ввод пользователя выделен полужирным шрифтом):

```
Введите строку: Русский текст.
Строка: Русский текст.
```

В ОС на базе UNIX таких проблем, как правило, не возникает.

Тема 9. Перечисления, структуры, объединения.

Помимо базовых типов Си (встроенных) в языке предусмотрено создание так называемых производных типов (из которых в нашем курсе изучаются только массивы и указатели), а также имеется возможность употребления определяемых пользователем типов: перечислений, структур и объединений.

1 Перечисления

Иногда в программе необходимы величины, у которых нет явного (или общепринятого) численного значения. Можно, конечно, объявить для этой цели специальные переменные и придумать им какие-то конкретные значения, однако существует более элегантный способ: можно объявить в программе некоторый набор идентификаторов (символьных имён) с автоматическим присваиванием им уникальных целочисленных значений.

Синтаксис определения перечисления: после ключевого слова **enum** указывается имя, выбранное для обозначения всего набора величин, а далее в фигурных скобках (через запятую) перечисляются имена самих отдельных величин; завершается определение точкой с запятой:

```
enum ИмяНабора {Величина1, Величина2, ...};
```

Предположим, что нам необходимо реализовать программу какой-то карточной игры или пасьянса. Как быть при этом с мастями карт, какие числовые значения им сопоставить? Для подобных ситуаций в языке предусмотрен весьма простой способ: достаточно перечислить названия карточных мастей в некотором «правильном» порядке в рамках определения:

```
enum Suit {Spades, Clubs, Diamonds, Hearts};
```

После этого можно использовать приводимые имена (а здесь это реальные названия мастей карт в английском языке) в программе: сравнивать масти различных карт на совпадение между собой или с конкретной мастью, а также учитывать старшинство мастей, поскольку теперь мастям соответствуют определённые значения: 0, 1, 2 и 3 — для указанного выше порядка следования (они присваиваются так по умолчанию). Имея такие значения, можно далее сопоставить им массив со строками их названий в русском языке (например, для удобного вывода).

Если какие-то идентификаторы в перечислении должны иметь вполне определённые значения, то их можно указывать после идентификатора и знака равенства. Тем идентификаторам, при которых желательное значение не указано, присваиваются последовательно увеличивающиеся (на единицу) целочисленные значения:

```
enum RomanDigit {I = 1, V = 5, X = 10};
```

Отметим также, что так как каждый элемент перечисления является константой, то этот элемент можно использовать в качестве объявления размерности массива:

```
int A[V][X];
```

Резюмируя, можно сказать, что фактически с помощью перечислений определяются величины с каким-то небольшим набором (целочисленных) значений, причём чаще всего программисту даже не нужно заботиться о том, чтобы сопоставить этим величинам конкретные значения, потому что компилятор сам позаботится об этом; программисту же будет достаточно лишь придумать имена этим величинам.

В следующем простом примере определяются численные значения для английских слов — названий дней недели — в соответствии с принятыми в нашей стране традициями следования дней недели в календарях.

```
#include <stdio.h>

enum WeekDay {
    Monday = 1, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
};
```

```
int main(void)
{
    printf("%d %d %d %d %d %d %d\n",
           Monday, Tuesday, Wednesday, Thursday,
           Friday, Saturday, Sunday);

    return 0;
}
```

Легко убедиться, что определяемые символьные целочисленные константы получили правильные значения.

Результат работы программы:

```
1 2 3 4 5 6 7
```

2 Структуры

Ранее читатели уже познакомились с таким типом данных как массив; он является набором однотипных величин. Структура с этой точки зрения — это набор *разнотипных* величин, объединённых вместе под одним именем и лежащих смежно в памяти компьютера. Точнее говоря, имён будет как минимум два: одно имя — при определении, имя нового типа, своеобразного «проекта» для набора будущих величин, другое — имя конкретного экземпляра структуры со своими значениями величин.

Использование такой синтаксической конструкции позволяет структурировать данные, то есть, организовать в них некоторую иерархию, где на каждом уровне имеется имя компоненты той или иной степени сложности, а далее сама компонента, в свою очередь, может оказаться более сложной, чем просто величина некоторого базового типа. Кстати, определение любой структуры может включать в себя другие (в том числе и сложные) структуры, однако не может содержать саму себя.

Синтаксис определения структуры: поскольку все компоненты структуры (называемые в языке Си полями) имеют чаще всего разные типы, а потому и различающиеся размеры, понадобятся имена для всех полей, составляющих структуру, — для того, чтобы к ним можно было обратиться. Напомним, что в случае массивов никакие дополнительные имена для элементов были не нужны: положение любого из них можно было найти по его номеру (индексу), так как все элементы имели одинаковый размер в силу их однотипности.

Поля структуры перечисляются (в виде пар `тип-имя` с точками с запятой в конце) в фигурных скобках, перед которыми имеется ключевое слово **`struct`** и имя, выбранное для этого типа структур; всё завершается также точкой с запятой, например:

```
struct Pack {
   int i;
   char c;
   float f;
};
```

Такое определение вводит в программу новый тип данных; величины этого типа появятся тогда, когда имя нового типа (**`struct`** `Pack`) или его определение будет предшествовать имени переменной.

Проинициализировать (то есть придать им конкретные значения) поля любой структуры можно либо при её объявлении с помощью синтаксиса, аналогичного инициализации элементов массива (это происходит при компиляции кода),

```
struct Pack ICF1 = {10, 'A', 3.14159};
```

либо в процессе работы готовой программы — с помощью выражения обращения к каждому полю структуры в левой части оператора присваивания.

Обратиться к любому из полей структуры (для определённости предположим, что мы будем работать с полями структуры `ICF1`, определённой выше) можно с помощью специальной операции обращения к отдельному полю (бинарная операция, обозначаемая символом «точка»). Значения полей можно как извлекать (считывать):

```
printf("%d %c %f\n", ICF1.i, ICF1.c, ICF1.f);
```

так и изменять (перезаписывать):

```
ICF1.i = 20; ICF1.c = 'B'; ICF1.f = 2.71828;
```

Если предстоит пользоваться структурой, имея на неё указатель, то можно воспользоваться существующей в языке Си специальной операцией: обращением к полю структуры по указателю на неё. Можно, конечно, сначала разыменовывать указатель (оператор «`*`», звёздочка), а затем, применяя к получаемой структуре операцию обращения к отдельному полю, получать значение этого поля. Однако из-за того, что приоритет операции разыменования ниже, чем приоритет операции обращения к полю, всегда понадобятся круглые скобки для того, чтобы операция с более низким приоритетом (разыменование) была выполнена до операции с более высоким приоритетом (обращение к полю).

```
(*УказательНаСтруктуру).ПолеСтруктуры

УказательНаСтруктуру->ПолеСтруктуры
```

Эти фрагменты выражений эквивалентны в языке Си, но при многократном применении в рамках выражения (это бывает тогда, когда полями структур являются указатели на структуры или аналогичные им конструкции) вариант со «стрелочкой» выглядит предпочтительнее — из-за отсутствия многочисленных обязательных скобок.

Структура может содержать и указатели на какие-то величины или на другие структуры. В этом случае в тип поля будет входить символ «звёздочка», а само поле — будет содержать адрес величины или другой структуры (сама величина или другая структура при этом должны располагаться где-то в другом месте), синтаксис работы с таким полем не будет отличаться от работы с указателем.

Хотя, как говорилось выше, определение структуры и не может включать в качестве поля её саму, указатель на такую же структуру может быть полем самой структуры. Рассмотрим пример с использованием подобной структуры (здесь это **`struct`** `Node` — дословно: узел). Видно, что каждый экземпляр структуры (конкретный узел) будет содержать какое-то целочисленное значение `value` и указатель `next` на некоторый следующий узел. Таким образом, появляется возможность образовать из таких узлов последовательность («цепочку») целых величин, порядок значений в которой будет определяться не расположением узлов, а «направленностью» их указателей; а в конце «цепочки» (где следующего узла со значением уже нет) в поле `next` будем размещать специальное значение «невозможного» указателя (`NULL`) — чтобы показать, что далее узлов нет.

```
#include <stdio.h>

struct Node {
     int value;
     struct Node *next;
};

struct Node b = {2, NULL}, a = {1, &b};
```

```
int main(void)
{
     struct Node *p = &a;

     while (p != NULL) {
         printf("%d\n", p->value);
         p = p->next;
     }

     return 0;
}
```

В данном примере, после определения структуры узла, объявляются (здесь — глобально, но в действительности они чаще всего размещаются в некотором отдельном участке динамической памяти) два узла: `b` и `a`. Они объявлены здесь в таком порядке потому, что поле `next` у экземпляра узла `a` должно быть инициализировано адресом расположения объявленного ранее экземпляра узла `b`. А у узла `b` в поле `next` записывается значение `NULL` — так как этот узел будет последним в этой маленькой «цепочке», начинающейся в узле `a` и заканчивающейся в узле `b`.

Программа содержит цикл прохождения организованной подобным образом «цепочки» (односвязного списка) с помощью указателя `p` на отдельный составляющий её узел. Первоначально этот указатель получает значение адреса начала «цепочки» (в нашем примере это адрес узла `a`). После вывода целочисленного значения в узле (`p->value`) на каждой итерации цикла указатель `p` получает новое значение: адрес следующего узла (`p = p->next`). Так происходит до тех пор, пока значение `p` не станет равным `NULL`, что означает: обработан последний узел «цепочки». Результат работы программы — вывод целых значений всех по порядку узлов в «цепочке»:

```
1
2
```

В следующем примере определена безымянная структура, описывающая иррациональную константу и состоящая из указателя на строку с её названием, её вещественной величины с двойной точностью и двухэлементного массива целых чисел: числителя и знаменателя дроби, приближающей величину иррациональной константы. Имя этой структуре не нужно, поскольку она больше нигде не будет использоваться для объявлений, а её величина, точнее, массив таких структур, объявляется здесь же и сразу инициализируется. Двухэлементный вложенный массив предназначен для хранения двух таких приближений: одного малой точности и другого — большей.

```
#include <stdio.h>

struct {
    char *valname;
    double value;
    struct {
        int numerator, denominator;
    } approx[2];
} irrationals[] = {
  "pi", 3.141592654, 22, 7, 355, 113, "e", 2.718281828, 19, 7, 87, 32
};

#define NIRRS (sizeof(irrationals) / sizeof(irrationals[0]))

int main(void)
{
    int n;
```

```
    for (n = 0; n < NIRRS; n++)
        printf("%s = %lf ~ %d/%d ~ %d/%d\n",
               irrationals[n].valname, irrationals[n].value,
               irrationals[n].approx[1].numerator,
               irrationals[n].approx[1].denominator,
               irrationals[n].approx[0].numerator,
               irrationals[n].approx[0].denominator);

    return 0;
}
```

В этой программе перебираются подряд все структуры, хранящие сведения об имеющихся иррациональных константах (здесь это знакомые всем со школы числа **π** и **e**), и выводится вся хранимая информация о конкретной константе: её имя, вещественное значение с двойной точностью и два приближения в виде рациональных дробей. Но поскольку размер массива структур `irrationals[]` не указан явно (это позволяет добавлять новые константы, не заботясь об изменении размера массива), он вычисляется при каждой компиляции программы с помощью стандартного для таких случаев выражения, определяемого директивой препроцессора.

Приближения выводятся в порядке, обратном порядку хранения: сначала более точное в виде дроби, затем менее точное — тоже в виде дроби. Поскольку в программе присутствуют несколько массивов (массив структур и два вложенных в каждую структуру массива дробей), то полезно обратить внимание на то, как они индексируются.

Сначала в выражениях (исполняемых слева направо) формируется нужная структура `irrationals[n]`, далее идёт обращение к содержащемуся в ней массиву дробей `irrationals[n].approx`, потом выбирается интересующая нас дробь в массиве (скажем, `irrationals[n].approx[0]`), от которой берётся числитель `irrationals[n].approx[0].numerator` или знаменатель `irrationals[n].approx[0].denominator`.

Обратите внимание, что инициализирующие величины в программе перечислены все подряд через запятую. Это допустимо, но поскольку в таком варианте легко ненароком сделать ошибку, удобнее использовать дополнительные фигурные скобки, как разделяющие структуры в массиве, так и отделяющие элементы вложенной структуры (друг от друга и их массив — от остальных полей), например, вот так:

```
...
} irrationals[] = {  {"pi", 3.141592635, {{22, 7}, {355, 113}}},
  {"e", 2.718281828, {{19, 7}, {87, 32}}}
};
```

Результат работы программы:

```
pi = 3.141593 ~ 355/113 ~ 22/7
e = 2.718282 ~ 87/32 ~ 19/7
```

3 Объединения

В отличие от структуры, где для всех ее полей (элементов) отводится отдельная память, в объединении все поля разделяют одну и ту же область памяти. Поля объединения используются не для раздельного хранения данных, а для возможности обращения к одним и тем же данным разным способом (в зависимости от типа поля).

Синтаксис определения объединения: после ключевого слова **union** следует имя объединений определяемого типа, а в фигурных скобках перечисляются пары `тип-имя` возможных величин, которые могут быть сохранены в объединении; завершается всё точкой с запятой.

```
union Pack3 {
    int i;
    char c;
    double f;
};
```

Как и в случае определения структуры, такое определение объединения просто вводит в программу новый тип величин; самих величин этого типа (**union** `Pack3`) пока ещё нет. Они появятся только после объявления величин такого типа по обычной схеме `тип-имя`, например, после строки

```
union Pack3 MyICFPack;
```

Можно сделать объявление величины и при определении объединения, употребляя выбранное для экземпляра такого объединения имя прямо перед завершающей наше определение точкой с запятой. Правда, это имеет смысл делать, если других таких объединений в программе не будет; если же другие экземпляры таких объединений нужны, то сначала определяется объединение (как это сделано выше), а затем объявляются (по мере необходимости) его конкретные экземпляры.

```
union Pack3 MyICFPack2;
```

Для чего можно использовать объединение? Во-первых, для хранения различных (разнотипных) величин, которые в любой момент времени исполнения программы используются поодиночке. Это может понадобиться, скажем, при ограниченном количестве памяти для хранения величин (см. Пример 1 ниже).

Во-вторых, для извлечения отдельных компонент какой-либо величины (в Примере 2 из представления вещественного значения могут быть считаны байты этого представления). В любом случае с помощью объединений одна и та же область памяти может быть использована в программе для хранения величин разного типа, а компилятор языка Си (как языка со строгой типизацией) не будет иметь к этому никаких претензий.

Пример 1: поочерёдное хранение разнотипных величин. Здесь используется определённое выше объединение **union** `Pack3` и его конкретный экземпляр `MyICFPack`.

```
MyICFPack.i = 10;  // Сейчас в объединении хранится целое значение 10
printf("%d\n", MyICFPack.i);
...
MyICFPack.c = 'A'; // Теперь в объединении хранится символ 'A'
printf("%c\n", MyICFPack.c);
...
MyICFPack.f = 3.14159;
                   // Теперь в объединении хранится вещественное число
printf("%f\n", MyICFPack.f);
```

Здесь демонстрируется заполнение объединения величиной некоторого (заранее предусмотренного) типа с последующим считыванием и выводом этой величины, чтобы убедиться, что там сохраняется именно она.

Пример 2: изучение представления вещественных величин. Определим такое объединение, в рамках которого хранится вещественная величина `f` одинарной точности (занимающая, как известно, четыре байта). Одновременно будем считать, что там же хранятся те 4 байта, которые она занимает (в виде массива `b`).

```
    union BytesOfFloat {
        float f;
        unsigned char b[4];
    } BF;
```

Такое определение позволяет записывать в объединение вещественное значение, а потом исследовать, что содержится в отдельных байтах его представления в памяти, верифицируя приводимые иногда схемы хранения вещественных чисел в компьютере. В эксперименте далее используется объявленный здесь же экземпляр `BF` этого объединения. Сначала в объединение записывается исследуемое вещественное число (здесь это 1.0, имеющее довольно простое представление), затем на экран выводятся шестнадцатеричные значения байтового представления этого числа.

```
    BF.f = 1.0;
    printf("%02x %02x %02x %02x\n", BF.b[0], BF.b[1], BF.b[2], BF.b[3]);
```

Следующий пример иллюстрирует определение синонима типа для объединения, инициализацию объединения, анонимные структуры, а также возможность вариантного (т.е., осуществляемого разными способами) обращения к объединению и его компонентам.

```
#include <stdio.h>

typedef union {
   short  v2;
   char   s[2];
   struct { char x, y; };
   struct { char s0, s1; };
} char2;

char2 C2 = { 0x1234 };

int main(void)
{
    printf("%04x\n", C2.v2);
    printf("%02x %02x\n", C2.s[0], C2.s[1]);
    printf("%02x %02x\n", C2.x, C2.y);
    printf("%02x %02x\n", C2.s0, C2.s1);

    return 0;
}
```

Одновременно эта программа демонстрирует (необычный на первый взгляд) порядок хранения байт в двухбайтовом целом значении. Инициализация переменной `C2` типа `char2` производится целым значением, шестнадцатеричное представление которого выглядит весьма специфично. Стоит обратить внимание на то, что это значение будет использовано для самого первого в определении объединения способа его трактовки (как будто бы присваивание значения осуществлялось так: `C2.v2 = 0x1234;`). Однако потом обращение к отдельным байтам возможно как с помощью массива `s` (`C2.s[0]` и `C2.s[1]`), так и с помощью имён полей безымянных структур (`C2.x` и `C2.y`; `C2.s0` и `C2.s1`). Если бы у структур были имена, их пришлось бы включить в «цепочку» обращения к полям структур, используя дополнительную операцию, обозначаемую символом «точка». А раз имён нет, все выражения выглядят проще, поскольку обращение идёт сразу к полям безымянных структур.

Результат работы этой программы:

```
1234
34 12
34 12
34 12
```