Курс «Основы Языка С»

Курс предназначен для специалистов о. 16 первого года работы и студентов, проходящие практику в отделе.

Целью курса является:

* углубление понимания основ программирования на языке C;
* ускорение освоения слушателями языка C на профессиональном уровне;
* внесение ясности в представление слушателей о возможностях языка C и месте языка C среди других ЯП;
* внесение ясности в понимание взаимосвязи и отличий языков C и C++;
* улучшение понимания общих подходов и технологий программирования.

Курс имеет практическую направленность и включает практические задания, выполняемые слушателями в свободное время (домашние задания) и финальный тест для оценки качества усвоения знаний и навыков.

Примерное содержание теоретической части курса:

[1 Обзор языка C](#_Toc402986478)

[1.1 Аннотация к разделу](#_Toc402986479)

[1.2 Краткая история появления и развития языка C](#_Toc402986480)

[1.3 Общая характеристика языка Си](#_Toc402986481)

[1.4 Си – структурированный язык](#_Toc402986482)

[1.5 Трансляция, компиляция, компоновка, распределение памяти](#_Toc402986483)

[1.6 Сравнительная характеристика языков C и C++](#_Toc402986484)

[2 Переменные и типы данных](#_Toc402986485)

[2.1 Базовые типы данных](#_Toc402986486)

[2.2 Порядок байт](#_Toc402986487)

[2.3 Модификация типов данных](#_Toc402986488)

[2.4 Переполнение диапазона целых чисел](#_Toc402986489)

[2.5 Преобразование типов](#_Toc402986490)

[2.6 Переменные и константы](#_Toc402986491)

[2.7 Инициализация переменных](#_Toc402986492)

[2.8 Область видимости и время жизни переменной](#_Toc402986493)

[2.9 Квалификаторы типа (const и volatile)](#_Toc402986494)

[2.10 Спецификаторы класса памяти](#_Toc402986495)

[2.11 Создание пользовательских типов данных](#_Toc402986496)

[2.12 Структуры](#_Toc402986497)

[2.13 Объединения](#_Toc402986498)

[2.14 Битовые поля](#_Toc402986499)

[2.15 Перечисления](#_Toc402986500)

[3 Выражения](#_Toc402986501)

[3.1 Операции](#_Toc402986502)

[3.2 Выражения](#_Toc402986503)

[4 Операторы](#_Toc402986504)

[4.1 Термины операция и оператор](#_Toc402986505)

[4.2 Условные операторы](#_Toc402986506)

[4.3 Операторы цикла](#_Toc402986507)

[4.4 Операторы перехода](#_Toc402986508)

[4.5 Оператор break](#_Toc402986509)

[4.6 Оператор continue](#_Toc402986510)

[5 Массивы и строки](#_Toc402986511)

[5.1 Одномерные массивы](#_Toc402986512)

[5.2 Строки](#_Toc402986513)

[5.3 Двумерные и N-мерные массивы](#_Toc402986514)

[5.4 Индексация указателей](#_Toc402986515)

[5.5 Инициализация массивов](#_Toc402986516)

[5.6 Массивы структур](#_Toc402986517)

[6 Указатели и динамическое распределение памяти](#_Toc402986518)

[6.1 Что такое указатели](#_Toc402986519)

[6.2 Операции для работы с указателями](#_Toc402986520)

[6.3 Выражения с указателями и адресная арифметика](#_Toc402986521)

[6.4 Указатели и массивы](#_Toc402986522)

[6.5 Массивы указателей](#_Toc402986523)

[6.6 Инициализация указателей](#_Toc402986524)

[6.7 Многоуровневая адресация](#_Toc402986525)

[6.8 Указатели на функции](#_Toc402986526)

[6.9 Динамическое распределение памяти](#_Toc402986527)

[6.10 Трудности при работе с указателями и динамически-распределяемой памятью.](#_Toc402986528)

[7 Функции](#_Toc402986529)

[7.1 Понятие функции](#_Toc402986530)

[7.2 Аргументы функции](#_Toc402986531)

[7.3 Возврат значений из функции](#_Toc402986532)

[7.1 Функция main()](#_Toc402986533)

[7.2 Рекурсия](#_Toc402986534)

[7.3 Прототипы функций](#_Toc402986535)

[7.4 Функции с переменным количеством аргументов](#_Toc402986536)

[8 Ввод/вывод](#_Toc402986537)

[8.1 Ввод/вывод на консоль](#_Toc402986538)

[8.2 Чтение и запись символов](#_Toc402986539)

[8.3 Чтение и запись строк](#_Toc402986540)

[8.4 Форматированный ввод/вывод на консоль (printf() и scanf())](#_Toc402986541)

[8.5 Файловый ввод/вывод](#_Toc402986542)

[8.6 Потоки и файлы](#_Toc402986543)

[8.7 Основные средства работы с файлами](#_Toc402986544)

[8.8 Указатель файла](#_Toc402986545)

[8.9 Открытие, закрытие и стирание файлов](#_Toc402986546)

[8.10 Чтение и запись символов и строк](#_Toc402986547)

[8.11 Чтение и запись двоичных данных (fread() и fwrite())](#_Toc402986548)

[8.12 Ввод/вывод при прямом доступе (fseek())](#_Toc402986549)

[8.13 Форматированный ввод-вывод при работе с файлами (fprintf() и fscanf())](#_Toc402986550)

[8.14 Стандартные потоки](#_Toc402986551)

[8.15 Перенаправление стандартных потоков](#_Toc402986552)

[9 Структуры данных](#_Toc402986553)

[9.1 Статические структуры данных](#_Toc402986554)

[9.2 Динамические структуры данных (динамические массивы, стеки, списки, деревья, таблицы)](#_Toc402986555)

[9.3 Очереди](#_Toc402986556)

[9.4 Стеки](#_Toc402986557)

[9.5 Связные списки](#_Toc402986558)

[9.6 Бинарные деревья](#_Toc402986559)

[10 Обзор стандартной библиотеки языка C](#_Toc402986560)

[10.1 Функции ввода/вывода](#_Toc402986561)

[10.2 Строковые и символьные функции](#_Toc402986562)

[10.3 Математические функции](#_Toc402986563)

[10.4 Функции времени, даты и локализации](#_Toc402986564)

[10.5 Функции динамического распределения памяти](#_Toc402986565)

[10.6 Служебные функции](#_Toc402986566)

[11 Функции стандарта POSIX](#_Toc402986567)

[12 Критерии качества программ на языке C](#_Toc402986568)

[Список литературы](#_Toc402986569)

[13 Список литературы](#_Toc402986570)

# Обзор языка C

## Аннотация к разделу

Первый раздел является вводным, его цель – знакомство с базовыми теоритическими понятиями и особенностями языка Си.

В данном разделе рассматриваются:

1. краткая история развития языка C;
2. основные особенности C и сравнение его с другими языками программирования;
3. определения основных понятий применяемых в классификации языков программирования;
4. общая структура программы на языке Си;
5. особенности трансляции и компоновки программ на языке Си;
6. особенности распределения памяти в программах на языке Си.

## Краткая история появления и развития языка C

Язык C был изобретен и реализован Деннисом Ритчи (Dennis Ritchie) для компьютера DEC PDP-11 в операционной системе Unix. Этот язык был разработан на основе «более старого» языка BCPL, созданного в свое время Мартином Ричардсом (Martin Richards). BCPL оказал определенное влияние на язык B, разработанный Кеном Томпсоном (Ken Thompson). В свою чередь развитие языка B привело к созданию в 1970 г. языка C.

На протяжении многих лет де-факто стандартом языка C была версия, поставляемая вместе с операционной системой Unix. Эта версия впервые была описана Брайаном Керниганом (Brian Kernighan) и Деннисом Ритчи в книге «The С Programming Language».

В декабре 1989 года организацией «Американский национальный институт стандартизации» (ANSI) был принят первый стандарт языка C. Версию C, определенную стандартом 1989 года, обычно называют C89. Данная версия языка является базовой для языка C++.

Первоначально C использовался для решения задач *системного программирования*. Системная программа – это часть ОС или утилита, как, например, транслятор, компоновщик, драйвер принтера и т.п. По мере роста популярности C, многие программисты стали использовать его для решения других задач благодаря его переносимости и эффективности. Этот язык стал долгожданным и впечатляющим в области языков программирования.

С появлением языка C++ многим стало казаться, что C прекратил свое существование как отдельный язык программирования. Однако это не так. Во-первых, не для всех программ нужны отличительные возможности C++, такие как объектно-ориентированность и обобщенное программирование. Во-вторых, в настоящее время во всем мире работает чрезвычайно много программ, написанных на C, причем разработчики продолжают модернизировать и поддерживать эти программы. В-третьих, язык C часто является незаменимым и единственным доступным (кроме ассемблера) языком программирования на некоторых «специальных» платформах, вроде микроконтроллеров и «специального» оборудования. В-четвертых, разработка нового стандарта C99 убеждает в том, что развитие и совершенствование языка С продолжается. То, что язык C стал базисом для многих современных языков, вроде C++, Objective-C и многих других навсегда останется его неоспоримой заслугой, и в то же время язык C сам остается одним из лучших языков программирования.

## Общая характеристика языка Си

Язык Си относят к языкам так называемого «среднего уровня». Как язык среднего уровня Си позволяет манипулировать битами, байтами и адресами, то есть теми базовыми элементами данных, с которыми работает компьютер. Поэтому Си хорошо приспособлен для системного программирования. Несмотря на это, программы, написанные на Си, обладают высокой переносимостью. Переносимость – это свойство программного обеспечения, созданного для одного типа компьютера или ОС, позволяющее легко переделать его для использования, т.е. перенести в другую вычислительную среду. Например, если программу, работающую под управлением Windows, легко переделать так, чтобы она работала под управлением FreeBSD или Linux.

В таблице 1.1 показано, какое место занимает Си среди других языков программирования. (Шилдт, 2008)

Таблица 1.1 – Место Си среди других языков программирования

|  |  |
| --- | --- |
| Языки высокого уровня | ADA  Delphi (Pascal)  FORTRAN  Python  JavaScript  Visual Basic |
| Языки среднего уровня | Java  C  C++  FORTH |
| Языки низкого уровня | Макроассемблер  Ассемблер |

Язык C (далее – Си) является типизированным процедурным компилируемым языком с явной слабой статической типизацией.

Язык Си является процедурным. Это означает, что последовательно выполняемые операторы программы на языке Си можно собрать в подпрограммы с помощью механизмов самого языка.

*Компилируемость* подразумевает, что программа, написанная на Си, должна быть оттранслирована в эквивалентную программу на низкоуровневом машинном языке.

Термин *типизированный* означает, что язык Си использует понятие *типы данных*. Все языки высокого уровня придерживаются концепции типов данных. *Тип данных* описывает набор значений, хранящихся в переменных, а также набор операций, выполнение которых допускается над этими переменными.

Язык Си является *явно-типизированным*. Это означает, что тип новых переменных, функций и их аргументов нужно задавать явно. Языки с неявной типизацией перекладывают эту задачу на компилятор, который автоматически выводит тип выражений.

Язык Си имеет статическую типизацию. Типизация языков программирования может быть *статической* или *динамической*. Статическая типизация определяется тем, что конечные типы переменных и функций устанавливаются на этапе компиляции. Примеры статической типизации: Pascal, C, C++, Java и т.д. динамической типизации типы определяются во время выполнения программы. Примеры динамической типизации: JavaScript, Python, Ruby, PHP, Objective C и т.д.

Язык Си имеет слабую типизацию. Языки программирования неформально делятся на сильно и слабо типизированные (иногда говорят «строгая» / «нестрогая»). *Сильная типизация* выделяется тем, что язык не позволяет смешивать в выражениях различные типы и не выполняет автоматические неявные преобразования, например нельзя вычесть из строки множество. Языки со *слабой типизацией* выполняют множество неявных преобразований автоматически, даже если может произойти потеря точности или преобразование неоднозначно. Примеры языков с сильной типизацией ADA, Java, Python, Haskel и т.д. Примеры языков со слабой типизацией: C, C++, JavaScript, PHP, Visual Basic, PHP и т.д.

В отличие от большинства языков высокого уровня, в Си отсутствует контроль ошибок в процессе выполнения программы. Например, не проверяется нарушение границ массивов. Ответственность за подобные ошибки полностью возлагается на программиста.

## Си – структурированный язык

Язык Си хорошо структурирован. В программировании часто используется понятие *блочной структурированности* языка. В блочно-структурированных языках программирования программа записывается как множество иерархически вложенных блоков определенной структуры. Хотя формально язык Си не может быть причислен к блочно-структурированным языкам, так как он не допускает определение функций внутри других функций, Си очень похож на такие языки (Pascal, ALGOL, Modula-2). Поэтому часто Си называют просто *структурированным* языком программирования.

Отличительной особенностью структурированного языка является отдельное размещение различных частей кода программы и данных. Таким образом, программист может «скрыть» часть информации, используемой для выполнения специфической задачи от тех участков программы, где эта информация не нужна. Один из способов достижения этого – использование подпрограмм и блоков с локальными переменными (подробнее в пункте 2.8). В этом случае любые действия внутри программы не вызовут побочных эффектов в других ее частях.

Структурированные языки предоставляют программисту много различных возможностей. Например, несколько типов операторов цикла, таких как while, do-while и for. В структурированных языках использование оператора goto запрещено или не рекомендуется.

В таблице 1.2 приведены примеры структурированных и неструктурированных языков.

Таблица 2.1

|  |  |
| --- | --- |
| **Неструктурированные** | **Структурированные** |
| FORTRAN  BASIC  COBOL | Pascal  Delphi  C  C++  Java  PHP  JavaScript  Modula-2 |

Главная конструкция структурного программирования на языке Си – функция, являющаяся здесь единственным видом подпрограммы. В функциях осуществляются все действия программы на языке Си. Правильное использование функций позволяет писать код, не вызывающий побочных эффектов в других частях программы. При работе над большим проектом, когда особенно важно, чтобы одна часть кода ни в коем случае не могла непредвиденно подействовать на другую часть, умение создать функцию для программиста приобретает исключительное значение.

Другой способ структурирования программы на языке Си, заключается в использовании программных блоков. *Программный блок* (далее просто – блок) – это логически связанная группа операторов программы, которую можно рассматривать как отдельную программную единицу. В языке Си блок представляет собой последовательность операторов (в т.ч. других блоков), заключенную в фигурные скобки.

if (index == -1)

{

printf(“Ошибка. Элемент не найден”);

return 0;

}

Использование блоков позволяет сделать программу более понятной и эффективной.

Важная особенность Си – малое количество ключевых слов. В стандарте C89 (который мы будем рассматривать) определено 32 ключевых слова. Языки высокого уровня обычно имеют значительно больше ключевых слов, например, в большинстве версий языка Basic их количество превышает сотню!

В таблице 1.3 приведены ключевые слова, определенные стандартом C89 (который мы будем рассматривать на протяжении всего курса).

Таблица 3.1 – Ключевые слова стандарта C89

|  |  |  |  |
| --- | --- | --- | --- |
| auto  break  case  char  const  continue  default  do | double  else  enum  extern  float  for  goto  if | int  long  register  return  short  signed  sizeof  static | struct  switch  typedef  union  unsigned  void  volatile  while |

В языке Си (в отличие от FORTRAN и Pascal) различаются верхний и нижний регистр символов. else – ключевое слово, ELSE – нет. Переменные с именами name и Name – разные переменные.

## Трансляция, компиляция, компоновка, распределение памяти

### Библиотеки и компоновка

В Си нет ключевых слов для выполнения многих операций, например ввод/вывод, вычисление математических функций, обработка строк и т.п. Поэтому в большинстве программ присутствуют вызовы различных функций, хранящихся в библиотеке стандартных функций Си. Все компиляторы Си поставляются вместе с *библиотекой стандартных функций*, предназначенных для выполнения наиболее общих задач (обзор функций стандартной библиотеки языка Си приведен в разделе 9.1).

Кроме стандартной библиотеки программы на языке Си могут использовать много других функций из разных внешних библиотек. К таким функциям относятся функции работы с графикой, специальные функции операционных систем, функции для работы с матрицами и векторами, геодезические функции и т.д.

При вызове библиотечной функции компилятор «запоминает» ее имя. Потом *компоновщик* связывает код исходной программы с объектным кодом, уже найденным в библиотеке. Этот процесс называется *компоновкой* (или *редактированием связей*).

Библиотеки содержат большое количество функций, необходимых для написания программы. Это своего рода кирпичики, из которых программист может собрать программу. Кроме того программист может написать свою функцию и поместить ее в библиотеку.

### Раздельная компиляция

Короткая программа на языке Си может состоять из одного файла исходного текста. Однако, при увеличении длины программы увеличивается также и время компиляции (которое может быть весьма долгим – от нескольких минут до нескольких часов). Кроме того, размещение всего исходного кода большой программы в одном текстовом файле является неудобным и снижает эффективность работы программиста. Поэтому программа на Си может состоять из двух или более файлов, компилируемых отдельно. Скомпилированные файлы программы компонуются с процедурами из библиотек, формируя, таким образом, исходный код программы. Схематично процесс раздельной компиляции и компоновки описан на рисунке 1.1.

**Файлы исходных текстов**

**Файлы исходных текстов**

**Файлы исходных текстов**

**Исполняемый файл или библиотека**

**Файлы исходных текстов**

**Файлы исходных текстов**

**Объектные файлы**

**Компоновка**

**Трансляция**

Рисунок 1.1 – Схема раздельной компиляции программ на языке Си

*Объектным кодом* называется результат трансляции исходного текста программы в машинный код, который может быть прочитан и выполнен компьютером.

Преимущество раздельной компиляции состоит в том, что при изменении одного файла нет необходимости заново перекомпилировать всю программу. При работе со сложными проектами это экономит много времени. Раздельная компиляция позволяет также нескольким программистам работать над одним проектом, т.е. она служит средством организации исходного текста программы для большого проекта.

Конкретный способ компиляции программы зависит от типа используемого компилятора. Например, компоновка может выполняться компилятором, а может и отдельной программой. Эти вопросы освещаются в документации компилятора.

### Карта памяти программы на языке Си

Скомпилированная программа на языке Си имеет четыре логически обособленные области памяти (приведенные на рисунке 1.2):

1. область памяти, содержащая выполнимый код программы;
2. область памяти, где хранятся *глобальные переменные*;
3. стек;
4. динамически распределенная область памяти (другие названия: динамическая память, куча и неупорядоченный массив).

|  |
| --- |
| Стек |
| Динамически распределенная область памяти |
| Глобальные переменные |
| Код программы |

Рисунок 1.2 – Распределение памяти (карта памяти) при выполнении программы на языке Си

*Стек* используется для хранения вспомогательных переменных во время выполнения программы. Здесь размещаются адреса возвратов функций, аргументы функций, локальные переменные и т.п. Текущее состояние процессора также хранится в стеке. Динамически распределяемая память – это такая свободная область памяти, для получения участков которой программа вызывает *функции динамического распределения памяти* (подробно работа с динамической памятью описана в подразделе 6.9).

Конкретное распределение памяти может быть разным в зависимости от типа процессора, операционной системы и компилятора.

## Сравнительная характеристика языков C и C++

В заключение необходимо сказать несколько слов о языке C++. Начинающие программисты не всегда ясно представляют, что такое C++ и чем именно он отличается от C. В нескольких словах, язык C++ это язык, поддерживающий объектно-ориентированную парадигму и парадигму обобщенного программирования. Фундаментом языка C++ является язык C. Язык C++ поддерживает обработку исключений, абстракцию данных, объявление типов (классов), виртуальные функции. Стандартная библиотека C++ включает, в том числе, общеупотребительные контейнеры и алгоритмы.

Язык C – подмножество C++ и, следовательно, C++ – надмножество C.

В общем случае компилятор C++ для компиляции программы, написанной на C. Но поскольку C++ основан на стандарте C89, при написании программы C, рассчитанной на компилятор C++, допускается использование только тех возможностей языка, которые предусмотрены в C89.

# Переменные и типы данных

## Базовые типы данных

Как вызнаете, *Тип данных* описывает набор значений, хранящихся в переменных, а также набор операций, выполнение которых допускается над этими переменными.

Стандарт С89 определяет пять фундаментальных типов данных:

1. char – символьные данные,
2. int – целые,
3. float – с плавающей точкой,
4. double – двойной точности,
5. void – без значения.
6. На основе этих типов формируются другие типы данных. Размер (объем занимаемой памяти) и диапазон значений этих типов данных для разных процессоров и компиляторов могут быть разными. Однако объект типа char всегда занимает 1 байт. Размер объекта int обычно совпадает с размером слова в конкретной среде программирования. В большинстве случаев в 16-разрядной среде (DOS или Windows 4.1) int занимает 16 битов, а в 32-разрядной (Windows 95/98/NT/2000) — 32 бита. Однако полностью полагаться на это нельзя, особенно при переносе программы в другую среду. Необходимо помнить, что **стандарт Си не определяет размер в байтах для типов данных**.

Конкретный формат числа с плавающей точкой зависит от его реализации в трансляторе. Переменные типа char обычно используются для обозначения набора символов стандарта ASCII, символы, не входящие в этот набор, разными компиляторами обрабатываются по-разному.

## Порядок байт

Раз уж мы затронули тему хранения целочисленных значений в памяти, стоит упомянуть про понятие *порядок байт*.

Когда число число не может быть представлено одним байтом, имеет значение в каком порядке байты записываются в памяти компьютера или передаются по линиям связи.

Наиболее распространенными правилами записи многобайтных целых чисел являются:

* порядок от старшего к младшему – Big-endian;
* порядок от младшего к старшему – Little-endian.

Отличия в порядке байт схематично представлены на рисунке 2..



Рисунок 2. – Отличия порядков байт Big-endian и Little-endian

Порядок от старшего к младшему – big-endian (часто называют – тупоконечный) является стандартным для протоколов TCP/IP, он используется в заголовках пакетов данных и во многих протоколах более высокого уровня, разработанных для использования поверх TCP/IP. Поэтому, порядок байтов от старшего к младшему часто называют сетевым порядком байтов. Порядок байтов от старшего к младшему применяется во многих форматах файлов — например, PNG, FLV, EBML.

Порядок от младшего к старшему – little-endian (остроконечный) принят в памяти персональных компьютеров с x86-процессорами, в связи с чем иногда его называют интеловский порядок байт (по названию фирмы-создателя архитектуры x86).

Запись многобайтового числа из памяти компьютера в файл или передача по сети требует соблюдения соглашений о том, какой из байтов передается первым. Прямая запись в том порядке, в котором байты расположены в ячейках памяти приводит к проблемам при переносе приложения с платформы на платформу.

## Модификация типов данных

Базовые типы данных (кроме void) могут иметь различные спецификаторы[1], предшествующие им в тексте программы. Спецификатор типа так изменяет значение базового типа, чтобы он более точно соответствовал своему назначению в программе. Полный список спецификаторов типов:

signed

unsigned

long

short

Базовый тип int может быть модифицирован каждым из этих спецификаторов. Тип char модифицируется с помощью unsigned и signed, double – с помощью long.

В таблице 2. приведены все допустимые комбинации типов данных с их минимальным диапазоном значений и типичным размером. Обратите внимание, в таблице приведены минимально возможные, а не типичные диапазоны значений.

Таблица 2. – Все типы данных, определенные стандартом C

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **Тип** | **Типичный размер в битах** | **Минимально допустимый диапазон значений** |
|  | char | 8 | от -127 до 127 |
|  | unsigned char | 8 | от 0 до 255 |
|  | signed char | 8 | от -127 до 127 |
|  | int | 16 или 32 | от -32767 до 32767 |
|  | unsigned int | 16 или 32 | от 0 до 65535 |
|  | signed int | 16 или 32 | то же, что int |
|  | short int | 16 | от -32767 до 32767 |
|  | unsigned short int | 16 | от 0 до 65535 |
|  | signed short int | 16 | то же, что short int |
|  | long int | 32 | от -2 147 483 647 до 2 147 483 647 |
|  | signed long int | 32 | то же, что long int |
|  | unsigned long int | 32 | от 0 до 4 294 967 295 |
|  | float | 32 | от 1E-37 до 1E+37, с точностью не менее 6 значащих десятичных цифр |
|  | double | 64 | от 1E-37 до 1E+37, с точностью не менее 10 значащих десятичных цифр |
|  | long double | 80 | от 1E-37 до 1E+37, с точностью не менее 10 значащих десятичных цифр |

При объявлении целого спецификатор signed как правило предполагается по умолчанию, однако char в некоторых реализациях может быть беззнаковым.

Целые числа со знаком и без знака отличаются интерпретацией нулевого бита числа. Если целое объявлено со знаком, компилятор считает, что нулевой бит содержит знак числа. Если в нулевом бите записан 0, число считается положительным, а если 1 — отрицательным.

В большинстве реализаций отрицательные числа представлены в двоичном дополнительном коде. Это значит, что для отрицательного числа все биты, кроме нулевого, инвертируются, к полученному числу добавляется 1, а нулевой бит устанавливается в 1.

Целые числа со знаком используются почти во всех алгоритмах, но абсолютная величина наибольшего из них составляет примерно только половину максимального целого без знака. Например, знаковое целое число 32767 в двоичном коде имеет вид

01111111 11111111

Если в нулевой бит записать 1, то оно будет интерпретироваться как -1. Однако если полученную запись рассматривать как представление числа, объявленного как unsigned int, то оно будет интерпретироваться как 65535.

Если спецификатор типа записать сам по себе (без следующего за ним базового типа), то предполагается, что он модифицирует тип int.

## Переполнение диапазона целых чисел

Переполнение – специфичная для компьютерной арифметики ситуация, когда при арифметическом действии результат становится больше максимально возможного значения для переменной, использующейся для хранения результата.

Переполнение может привести к серьёзным проблемам: обнулению и потере данных, возможностям взлома программы, трудноуловимым ошибкам, которые будут накапливаться с течением времени, и т.д.

При этом, не произойдёт ни какой исключительной ситуации, ни завершения программы с ошибкой (это верно для языков Си и Си++). получится конкретно, зависит от особенностей представления чисел на конкретной системе.

## Преобразование типов

В операторах и выражениях, вообще говоря, должны использоваться переменные и константы только одного типа. Язык Си имеет слабую типизацию. Поэтому если все же вы смешаете типы в одном выражении, то компилятор с языка Си не считает программу неправильной, как это произошло бы при программировании на Паскале. Вместо этого компилятор использует набор правил для автоматического преобразования типов. Это очень удобно, но может оказаться и опасным, особенно если вы допустили смешение типов нечаянно. Рассмотрим несколько основных правил, неявного преобразования типов:

1. если операция выполняется над данными двух различных типов, обе величины приводятся к "высшему" из двух типов. Этот процесс называется "повышением" типа;
2. последовательность имен типов, упорядоченных от "низшего" к "высшему", выглядит так: char, short, int, long, float и double. Применение ключевого слова unsigned повышает ранг соответствующего типа данных со знаком;
3. в операторе присваивания конечный результат вычисления выражения в правой части приводится к типу переменной, которой должно быть присвоено это значение. Данный процесс может привести к "повышению" типа, как описано выше, или к "понижению, при котором величина приводится к типу данных, имеющему более низкий приоритет.

Повышение типа обычно происходит гладко, в то время как понижение" может привести к затруднениям. Причина этого проста: все число целиком может не поместиться в элементе данных низшего типа. Например, для платформ с размером байта – 8 бит переменная типа char может иметь целое значение 101, но не 22334.

Неявные преобразования типов опасны и могут привести к ошибкам, поэтому часто компиляторы выдают предупреждения при выполнении соответствующих преобразования. Для того, чтобы избежать проблем с неявными преобразованиями, рекомендуется применять т.н. явное преобразование типов.

Механизм неявных преобразований может быть отключён посредством явного указания в тексте программы требуемого преобразования типов.

MyIntU = MyIntU + (int)3.14159;

## Переменные и константы

Переменная – это именованный участок памяти, в котором хранится значение, которое может быть изменено программой. Все переменные перед их использованием должны быть объявлены. Общая форма объявления имеет такой вид:

тип список\_переменных;

Здесь тип означает один из базовых или объявленных программистом типов, а список\_переменных состоит из одного или более идентификаторов, разделенных запятыми. Ниже приведены примеры объявлений:

int i;

double x, y, z;

unsigned int clients\_count;

float balance, profit, loss;

В языке С имена переменных, функций и т.п. называются идентификаторами. Первый символ идентификатора должен быть буквой или символом подчеркивания, последующие символы должны быть буквами, цифрами или символами подчеркивания.

Константы используются в тексте программы для обозначения числовых значений, строк, символов или логических констант. Другими словами константа представляет собой постоянное значение, у которого нет имени. Константы также называются литералами.

Символьные константы заключаются в одинарные кавычки. Например, 'а' и '%' — это символьные константы.

В языке С определены многобайтовые (состоящие из одного или более байт) и широкие (обычно длиной 16 бит) символы. Они используются для представления символов языков, имеющих в своем алфавите много букв. Многобайтовый символ записывается в одинарных кавычках, например, 'ху', а широкий – с предшествующим символом L, например:

wchar\_t wc = L'A';

Целые константы определяются как числа без дробной части. Например, 10 и -100 — это целые константы. Константы в плавающем формате записываются как числа с десятичной точкой, например, 11.123. Допускается также экспоненциальное представление чисел (в виде мантиссы и порядка): 111.23е-1.

По умолчанию компилятор приписывает константе тип наименьшего размера, в ячейку которого может уместиться константа. Таким образом, если целые числа обычно являются 16-разрядными, то константа 10 по умолчанию имеет тип int, a 103000 — тип long int. Число 10 может поместиться в типе char, однако компилятор не нарушит границы типов и поместит ее в int. Но это правило имеет исключение: всем константам в плавающем формате, даже самым маленьким, приписывается тип double.

Если после числа в плавающем формате стоит суффикс F, то считается, что константа имеет тип float, а если L, то long double. Для целых типов суффикс U означает unsigned, a L – long. Тип суффикса не зависит от регистра, например, как F, так и f определяют константы типа float.

Иногда удобнее использовать не десятичную, а восьмеричную или шестнадцатеричную систему.

int hex = 0x80; /\* 128 в десятичной системе \*/

int oct = 012; /\* 10 в десятичной системе \*/

Язык С поддерживает еще один тип констант, а именно — строковые. Строка – это последовательность символов, заключенных в двойные кавычки. Не следует путать понятия строки и символа. Символьная константа заключается в одинарные кавычки, например, 'а'. Соответственно запись "а" означает строку, состоящую из одного символа.

Чтобы представить большинство символьных констант, достаточно заключить соответствующий символ в одинарные кавычки. Но некоторые символы, например, символ возврата каретки, требуют специального представления. В языке С определены специальные символьные константы, приведенные в табл. 2.2.

Таблица 2.

|  |  |
| --- | --- |
| ***Код*** | ***Назначение*** |
| \b | Удаление предыдущего символа |
| \n | Новая строка |
| \r | Возврат каретки |
| \t | Горизонтальная табуляция |
| \" | Двойные кавычки |
| \' | Одинарная кавычка |
| \\ | Обратный слэш |

## Инициализация переменных

При объявлении переменной она может быть инициализирована. Для этого нужно после ее объявления поставить знак равенства и константу, т.е. общая форма инициализации имеет следующий вид:

тип имя\_переменной = литерал;

Приведем несколько примеров инициализации переменных:

char ch = 'a';

int first = 0;

double balance = 123.23;

## Область видимости и время жизни переменной

*Областью видимости* переменной называется фрагмент программы, в котором переменная доступна для использования. За пределами области видимости обращение к переменной невозможно.

*Временем жизни* переменной называется интервал выполнения программы, в течение которого она существует.

Переменные различаются на *глобальные* и *локальные*.

Переменные, объявленные вне функций, называют глобальными переменными.

Глобальную переменную можно использовать в любом месте файла (ниже строки объявления этой переменной), в котором она объявлена.

Переменные, объявленные внутри функций, называются локальными переменными.

Локальную переменную можно использовать только внутри блока, в котором она объявлена. Иными словами, область видимости локальной переменной ограничена блоком, в котором она объявлена. Переменная, объявленная в одном блоке, не имеет никакого отношения к переменной с тем же именем, объявленной в другом блоке. Это означает, что в разных функциях совершенно независимо друг от друга могут использоваться переменные с одинаковыми именами. Это позволяет писать код, не имеющих побочных эффектов в других частях программы.

Рассмотрим, например, следующую функцию:

int a = 0; // глобальная переменная

int main()

{

printf("%d", a); // будет выведено число 0

{

int a = 1; // объявлена локальная переменная а, глобальная переменная a не видна

printf("%d", a); // будет выведено число 1

{

int a = 2; // еще локальная переменная в блоке, глобальная переменная a не видна, не видна и предыдущая локальная переменная

printf("%d", a); // будет выведено число 2

}

}

}

Локальная переменная при каждом входе в блок инициализируется заново. Локальные переменные существуют только во время выполнения программного блока, в котором они объявлены, создаются они при входе в блок, а разрушаются – при выходе из него.

Глобальные переменные создаются и инициализируются один раз при старте программы. Время жизни глобальной переменной совпадает со временем выполнения программы.

Неинициализированные локальные переменные до первого присвоения имеют произвольное значение. Неинициализированные глобальные в начале работы программы автоматически обнуляются.

## Квалификаторы типа (const и volatile)

В языке С определяются квалификаторы типа[1], указывающие на доступность и модифицируемость переменной. Стандарт С89 определяет два квалификатора: const и volatile. Квалификатор типа должен предшествовать имени типа, который он квалифицирует (уточняет).

Переменная, к которой в объявлении (декларации) применен квалификатор const, не может изменять свое значение. Ее можно только инициализировать, то есть присвоить ей значение в начале выполнения программы. Например:

const int a=10;

Компилятор может применять к константам более эффективные методы оптимизации, основанные на предположении о неизменности их значения.

Квалификатор const часто используется для того, чтобы предотвратить изменение функцией объекта, на который указывает аргумент функции. Без него при передаче в функцию указателя эта функция может изменить объект, на который он указывает. Однако если в объявлении параметра-указателя применен квалификатор const, функция не сможет изменить этот объект, например:

void sp\_to\_dash(const char \*str)

{

\*str = “строка”4 // ошибка - невозможно изменить значение переменной

}

Квалификатор const позволяет улучшить понимание назначения функции, наглядно разделяя параметры, передаваемые через указатель на изменяемые и неизменяемые.

Квалификатор volatile указывает компилятору на то, что значение переменной может измениться независимо от программы, т.е. вследствие воздействия еще чего-либо, не являющегося оператором программы. Например, адрес глобальной переменной можно передать в подпрограмму операционной системы, следящей за временем, и тогда эта переменная будет содержать системное время. В этом случае значение переменной будет изменяться без участия какого-либо оператора программы. Знание таких подробностей важно потому, что большинство компиляторов С автоматически оптимизируют некоторые выражения, предполагая при этом неизменность переменной, если она не встречается в левой части оператора присваивания. В этом случае при очередной ссылке на переменную может использоваться ее предыдущее значение. Некоторые компиляторы изменяют порядок вычислений в выражениях, что может привести к ошибке, если в выражении присутствует переменная, вычисляемая вне программы. Квалификатор volatile предотвращает применение таких оптимизаций программы компилятором.

Квалификаторы const и volatile могут применяться и совместно. Например, если 0x30 – адрес порта, значение в котором может задаваться только извне, то следующее объявление предотвратит всякую возможность нежелательных побочных эффектов:

const volatile char \*port = (const volatile char \*) 0x30;

## Спецификаторы класса памяти

Стандарт С поддерживает четыре спецификатора класса памяти:

extern

static

register

auto

Эти спецификаторы сообщают компилятору, как он должен разместить соответствующие переменные в памяти. Общая форма объявления переменных при этом такова:

*спецификатор\_класса\_памяти тип имя переменой;*

**Типы связывания переменных**

Прежде чем приступить к рассмотрению спецификатора extern, необходимо коротко остановиться на компоновке программы.

В языке С при редактировании связей к переменной может применяться одно из трех связываний: *внутреннее*, *внешнее* или же не относящееся ни к одному из этих типов. (В последнем случае редактирование связей к ней не применяется.) В общем случае к именам функций и глобальных переменных применяется внешнее связывание. Это означает, что после компоновки они будут доступны во всех файлах, составляющих программу. К объектам, объявленным со спецификатором static и видимым на уровне файла, применяется внутреннее связывание, после компоновки они будут доступны только внутри файла, в котором они объявлены. К локальным переменным связывание не применяется и поэтому они доступны только внутри своих блоков.

Спецификатор auto уведомляет компилятор о том, что локальная переменная, перед именем которой он стоит, создаётся при входе в блок и разрушается при выходе из блока. Все переменные, определённые внутри функции, являются автоматическими по умолчанию, и по этому ключевое слово auto используется крайне редко.

**Спецификатор extern**

Спецификатор extern указывает на то, что к объекту применяется внешнее связывание, именно поэтому они будут доступны во всей программе. Если нужно сослаться на переменную, определенную в другой части программы, необходимо объявить ее как внешнюю (extern). Пример:

#include <stdio.h>

int main(void)

{

extern int first, last; /\* используются глобальные переменные \*/

printf("%d %d", first, last);

return 0;

}

/\* описание глобальных переменных first и last \*/

int first = 10, last = 20;

При компиляции выполняются следующие правила. Если компилятор находит переменную (со спецификатором extern), не объявленную внутри блока, он ищет ее объявление во внешних блоках. Если не находит ее и там, то ищет среди объявлений глобальных переменных. В предыдущем примере, если бы не было объявления extern, компилятор не нашел бы first и last среди глобальных переменных, потому что они объявлены после main().

Спецификатор extern играет большую роль в программах, состоящих из многих файлов. Как вы знаете, в языке С программа может быть записана в нескольких файлах, которые компилируются раздельно, а затем компонуются в одно целое. В этом случае необходимо как-то сообщить всем файлам о глобальных переменных программы. Для этого необходимо описать глобальные переменные в одном файле и объявить их со спецификатором extern в остальных файлах, как показано на рис. 2..

**Файл 1** **Файл 2**

int x, y; extern int x, y;

char ch; extern char ch;

int main(void) void func22(void)

{ {

/\* ... \*/ x = y / 10;

} }

void func1(void) void func23(void)

{ {

x = 123; y = 10;

} }

Рисунок 2.2 – Использование спецификатора extern

На практике программисты обычно включают объявления extern в заголовочные файлы, которые просто подключаются к каждому файлу исходного текста программы. Это более легкий путь, который к тому же приводит к меньшему количеству ошибок, чем повторение этих объявлений вручную в каждом файле.

**Спецификатор static**

Переменные, объявленные со спецификатором static, хранятся постоянно внутри своей функции или файла. В отличие от глобальных переменных они невидимы за пределами своей функции или файла, но они сохраняют свое значение между вызовами.

Пример:

int series(void)

{

static int series\_num = 100;

series\_num = series\_num+23;

return series\_num;

}

В этом примере переменная series\_num продолжает существовать между вызовами функций, в то время как обычная локальная переменная создается заново при каждом вызове, а затем уничтожается. Статическую локальную переменную можно инициализировать. Это значение присваивается ей только один раз — в начале работы всей программы.

Спецификатор static в объявлении глобальной переменной заставляет компилятор создать глобальную переменную, видимую только в том файле, в котором она объявлена. Статическая глобальная переменная, таким образом, подвергается внутреннему связыванию, как описано ранее в пункте "Спецификатор extern".

Поскольку для функций по умолчанию используется внешнее связывание, спецификатор static рекомендуется использовать с функциями, которые должны использоваться только внутри одного файла исходного текста во избежание конфликта имен.

static void localFunc()

{

…

}

**Спецификатор register**

В первых версиях компиляторов С спецификатор register сообщал компилятору, что переменная должна храниться в регистре процессора, а не в оперативной памяти, как все остальные переменные. Это приводит к тому, что операции с переменной register осуществляются намного быстрее, чем с обычными переменными, потому такая переменная уже находится в процессоре и не нужно тратить время на выборку ее значения из оперативной памяти (и на запись в память).

В настоящее время определение спецификатора register существенно расширено. Стандарты С89 и С99 попросту декларируют "доступ к объекту так быстро, как только возможно". Практически при этом символьные и целые переменные по-прежнему размещаются в регистрах процессора. Конечно, большие объекты (например, массивы) не могут поместиться в регистры процессора, однако компилятор получает указание "позаботиться" о быстродействии операций с ними. В зависимости от конкретной реализации компилятора и операционной системы переменные register обрабатываются по-разному. Иногда спецификатор register попросту игнорируется, а переменная обрабатывается как обычная, однако на практике это бывает редко.

Спецификатор register можно применить только к локальным переменным и формальным параметрам функций. В объявлении глобальных переменных применение спецификатора register не допускается.

## Создание пользовательских типов данных

В языке С имеется пять способов создания пользовательских типов данных. Пользовательские типы данных можно создать с помощью:

* *структуры* – группы переменных, имеющей одно имя и называемой агрегатным типом данных;
* *объединения*, которое позволяет определять один и тот же участок памяти как два или более типов переменных;
* *битового поля*, которое является специальным типом элемента структуры или объединения, позволяющим легко получать доступ к отдельным битам;
* *перечисления* – списка поименованных целых констант;
* ключевого слова typedef, которое определяет новое имя для существующего типа.

## Структуры

Структура – это совокупность переменных, объединенных под одним именем. С помощью структур удобно размещать в смежных полях связанные между собой элементы информации. Объявление структуры создает шаблон, который можно использовать для создания ее объектов (то есть экземпляров этой структуры). Переменные, из которых состоит структура, называются членами. (Члены структуры еще называются элементами или полями.)

struct PortInfo

{

PortState port\_state; // Состояние порта в текущий момент

int port\_time; // Суммарная продолжительность соединения

int error\_num; // Число сбоев порта

};

Как правило, члены структуры связаны друг с другом по смыслу. Ключевое слово struct сообщает компилятору, что объявляется (еще говорят, "декларируется") структура.

Обратите внимание, что объявление завершается точкой с запятой.

Когда вы определяете структуру, вы определяете ее тип, а не конкретную переменную. Чтобы объявить переменную (то есть физический объект) типа PortInfo, необходимо написать следующее:

struct PortInfo info;

В этом операторе объявлена переменная типа PortInfo, которая называется info. Когда объявляется переменная-структура, компилятор автоматически выделяет количество памяти, достаточное, чтобы разместить все ее члены.

Одновременно с определением структуры можно объявить одну или несколько ее переменных:

struct PortInfo

{

PortState port\_state; // Состояние порта в текущий момент

int port\_time; // Суммарная продолжительность соединения

int error\_num; // Число сбоев порта

} port\_in, port\_out;

Если нужна только одна переменная-структура, то тег структуры является лишним. В этом случае наш пример объявления можно переписать следующим образом:

struct

{

PortState port\_state; // Состояние порта в текущий момент

int port\_time; // Суммарная продолжительность соединения

int error\_num; // Число сбоев порта

} port\_info;

Доступ к отдельным членам структуры осуществляется с помощью оператора “.” (который обычно называют *оператором точка* или *оператором доступа к члену структуры*)

port\_info.port\_state = 1;

Информация, которая находится в одной структуре, может быть присвоена другой структуре того же типа при помощи единственного оператора присваивания. Нет необходимости присваивать значения каждого члена в отдельности.

struct PortInfo port\_in, port\_in\_default;

...

port\_in = port\_in\_default;

## Объединения

*Объединение* – это место в памяти, которое используется для хранения переменных, разных типов. Объединение дает возможность интерпретировать один и тот же набор битов не менее чем двумя разными способами. Объявление объединения (начинается с ключевого слова union) похоже на объявление структуры, например:

union UType {

int i;

char ch[2];

};

Объявление переменной – объединения аналогично объявлению переменной – структуры.

union UType cnvt;

Когда переменная объявляется с ключевым словом union, компилятор автоматически выделяет столько памяти, чтобы в ней поместился самый большой член нового объединения. Например, при условии, что целые значения занимают по 4 байта, для размещения i в cnvt необходимо, чтобы длина этого объединения составляла 4 байта, даже если для ch требуется только 2 байта.

Для получения доступа к члену объединения используйте тот же синтаксис, что и для структур.

cnvt.i = 10;

Объединения часто используются тогда, когда нужно выполнить специфическое преобразование типов, потому что хранящиеся в объединениях данные можно обозначать совершенно разными способами. Например, используя объединения, можно манипулировать байтами, составляющими значение типа double, и делать так, чтобы менять его точность или выполнять какое-либо необычное округление.

## Битовые поля

В отличие от некоторых других компьютерных языков, в языке С имеется встроенная поддержка *битовых полей*, которая дает возможность получать доступ к единичному биту.

struct device

{  
 unsigned active : 1;  
 unsigned ready : 1;  
 unsigned xmt\_error : 1;  
} dev\_code;

Рисунок 2. показывает, как выглядит переменная dev\_code в памяти.

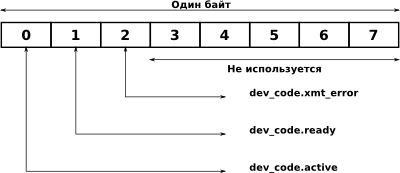


Рисунок 2. – Размещение битового поля в памяти

Битовые поля могут быть полезны по разным причинам, а именно:

* если память ограничена, то в одном байте можно хранить несколько булевых переменных (принимающих значения ИСТИНА и ЛОЖЬ);
* некоторые устройства передают информацию о состоянии, закодированную в байте в одном или нескольких битах;
* для некоторых процедур шифрования требуется доступ к отдельным битам внутри байта.

Битовые поля длиной 1 должны объявляться как unsigned, поскольку 1 бит не может иметь знака. Битовые поля могут иметь длину от 1 до16 бит для 16-битных сред и от 1 до 32 бит для 32-битных сред.

## Перечисления

*Перечисление* – это набор именованных целых констант. Вот, например, перечисление, в котором приведены названия дней недели:

enum DayOfWeek {

sunday,

monday,

thuesday,

wednesday,

thoursday,

frieday,

saturday

};

Главное, что нужно знать для понимания перечислений – каждый их элемент представляет целое число. В таком виде элементы перечислений можно применять везде, где используются целые числа. Каждому элементу дается значение, на единицу большее, чем у его предшественника. Первый элемент перечисления имеет значение 0. Например, при выполнении кода:

printf("%d %d", sunday, thuesday);

на консоль будет выведено: «0 2».

Однако для одного или более элементов можно указать значение, используемое как инициализатор. Для этого после перечислителя надо поставить знак равенства, а затем — целое значение. Перечислителям, которые идут после инициализатора, присваиваются значения, большие предшествующего. Например, следующий код присваивает thuesday значение 100:

enum DayOfWeek {

sunday,

monday,

thuesday = 100,

wednesday,

thoursday,

frieday,

saturday

};

Вот какие значения появились у этих элементов:

|  |  |
| --- | --- |
| sunday | 0 |
| monday | 1 |
| thuesday | 100 |
| wednesday | 101 |
| thoursday | 102 |
| frieday | 103 |
| saturday | 104 |

# Выражения

В настоящем разделе рассматриваются выражения и их составляющие: *операторы* (operators) и аргументы операторов. Английскому слову operator соответствуют два понятия: *оператор* и *операция*. Оператором обычно называют наименьшую автономную часть языка программирования, *команду*, *инструкцию* (statement). Путаница усугубилась тем, что в C присваивание и инкремент/декремент являются и операторами, и операциями.

Операторы рассматриваются в следующем разделе. В настоящем разделе рассматривается *операции* – конструкции языка программирования, аналогичные математическим операциям.

Язык С содержит большое количество встроенных операций. Существует четыре основных класса операций: арифметические, логические, поразрядные и операции сравнения. Кроме них, есть также некоторые специальные операторы, например, оператор присваивания.

Операции как правило состоят из знаков операции и аргументов.

Операции делятся по количеству аргументов:

* унарные – один аргумент (отрицание, унарный минус, взятие адреса);
* бинарные – два аргумента (сложение, вычитание, умножение);
* тернарные – три аргумента (в языке C есть только один тернарный оператор «условие ? выражение1 : выражение2», который часто так и называется «тернарный оператор»).

Для бинарных и тернарных (с 3 операндами) операций в подавляющем большинстве случаев используют инфиксную нотацию — по той простой причине, что она принята в математике и наиболее привычна большинству людей. Для унарных операций инфиксной нотации не существует, и, как правило, используется префиксная.

Аргументами операций могут выступать переменные, константы, литералы, значения, возвращаемые функциями и другими операциями.

Изучение данного раздела начнем с переменных

## Операции

Язык С содержит большое количество встроенных операций. Их роль в С значительно больше, чем в других языках программирования. Существует четыре основных класса операций: *арифметические*, *логические*, *поразрядные* и *операции сравнения*. Кроме них, есть также некоторые специальные операторы, например, оператор присваивания.

### Оператор присваивания

Оператор присваивания может присутствовать в любом выражении языка С. Этим С отличается от большинства других языков программирования (Pascal, BASIC и FORTRAN), в которых присваивание возможно только в отдельном операторе. Общая форма оператора присваивания:

*имя\_переменной=выражение;*

Выражение может быть просто константой или сколь угодно сложным выражением. Адресатом (получателем), т.е. левой частью оператора присваивания должен быть объект, способный получить значение, например, переменная.

В книгах по С и в сообщениях компилятора часто встречаются термины lvalue (left side value) и rvalue (right side value). Попросту говоря, lvalue — это объект в левой части оператора присваивания, получающий значение, чаще всего этим объектом является переменная. Термин rvalue означает значение выражения в правой части оператора присваивания.

В одном операторе присваивания можно присвоить одно и то же значение многим переменным. Для этого используется оператор множественного присваивания, например:

x = y = z = 0;

Составное присваивание — это разновидность оператора присваивания, в которой запись сокращается и становится более удобной в написании. Например, оператор

x = x+10;

можно записать как

x += 10;

Оператор "+=" сообщает компилятору, что к переменной х нужно прибавить 10.

"Составные" операторы[6] присваивания существуют для всех бинарных операций (то есть операций, имеющих два операнда):

–=

\*=

/=

&=

|=

и т.д.

Составное присваивание значительно компактнее, чем соответствующее простое присваивание, поэтому этот оператор широко используется в программах на языке Си.

### Арифметические операции

В табл. 3.1 приведены арифметические операции С.

Таблица 3. – Арифметические операции

|  |  |
| --- | --- |
| – | Вычитание, унарный минус |
| + | Сложение |
| \* | Умножение |
| / | Деление |
| % | Остаток от деления |
| -- | Декремент, или уменьшение |
| ++ | Инкремент, или увеличение |

### Операции увеличения (инкремента) и уменьшения (декремента)

В языке С есть два полезных оператора, значительно упрощающие широко распространенные операции. Это инкремент ++ и декремент --. Оператор ++ увеличивает значение операнда на 1, а — уменьшает на 1. Иными словами,

x = x+1;

можно записать как

++x;

Аналогично оператор

x = x-1;

равносилен оператору

--x;

Как инкремент, так и декремент могут предшествовать операнду (префиксная форма) или следовать за ним (постфиксная форма). Например

++x;

x++;

Префиксная и постфиксная формы отличаются при использовании их в выражениях. Если оператор инкремента или декремента предшествует операнду, то сама операция выполняется до использования результата в выражении. Если же оператор следует за операндом, то в выражении значение операнда используется до выполнения операции инкремента или декремента. То есть для выражения эта операция как бы не существует, она выполняется только для операнда. Например,

x = 10;

y = ++x; // y == 11

x = 10;

y = x++; // y == 10

Приоритет арифметических операций:

**Наивысший** ++ --

- (унарный минус)

\* / %

**Наинизший** + -

### Операции сравнения и логические операции

Операции сравнения — это операции, в которых значения двух переменных сравниваются друг с другом. Логические же операции реализуют средствами языка С операции формальной логики. Между логическими операциями и операциями сравнения существует тесная связь: результаты операций сравнения часто являются операндами логических операций.

В операциях сравнения и логических операциях в качестве операндов и результатов операций используются значения ИСТИНА (true) и ЛОЖЬ (false). В языке С значение ИСТИНА представляется любым числом, отличным от нуля. Значение ЛОЖЬ представляется нулем. Результатом операции сравнения или логической операции являются ИСТИНА (true, 1) или ЛОЖЬ (false, 0).

В табл. 3.3 приведены операции сравнения и логические операции С.

Таблица 3. – Операции сравнения и логические операции

|  |  |
| --- | --- |
| **Оператор** | **Операция** |
| *Операторы сравнения* | |
| > | Больше чем |
| >= | Больше или равно |
| < | Меньше чем |
| <= | Меньше или равно |
| == | Равно |
| != | Не равно |
| *Логические операции* | |
| && | И |
| || | ИЛИ |
| ! | НЕ, отрицание |

Как операции сравнения, так и логические операции имеют низший приоритет по сравнению с арифметическими. То есть, выражение 10>1+12 интерпретируется как 10>(1+12). Результат, конечно, равен ЛОЖЬ. Ниже приведен приоритет логических операций:

**Наивысший** !

> >= < <=

== !=

&&

**Наинизший** ||

Как и в арифметических выражениях, для изменения порядка выполнения операций сравнения и логических операций можно использовать круглые скобки.

### Поразрядные операции

В отличие от многих других языков программирования, в С определен полный набор поразрядных операций. Это обусловлено тем, что С был задуман как язык, призванный во многих приложениях заменить ассемблер, который способен оперировать битами данных. Поразрядные операции — это тестирование (проверка), сдвиг или присвоение значений отдельным битам данных. Эти операции осуществляются над ячейками памяти, содержащими данные типа char или int. Данные типа float, double, long double или другие более сложные не могут участвовать в поразрядных операциях.

В табл. 3.5 приведены поразрядные операции С.

Таблица 3. – Операции поразрядные операции

|  |  |
| --- | --- |
| **Оператор** | **Операция** |
| & | Поразрадное И |
| | | Поразрядное ИЛИ |
| ^ | Поразрядное исключающее ИЛИ |
| ~ | НЕ (отризание) |
| >> | Сдвиг вправо |
| << | Сдвиг влево |

Наиболее часто поразрядные операции применяются при программировании драйверов устройств, таких как модемы, а также процедур, выполняющих операции над файлами, стандартных программ обслуживания принтера и т.д. В них поразрядные операции используются для чтения и записи определенных битов.

Поразрядные операторы сдвига >> и << сдвигают все биты переменной вправо или влево. Общая форма операторjd сдвига вправо и влево:

переменная >> количество\_разрядов

переменная << количество\_разрядов

Во время сдвига битов в один конец числа, другой конец заполняется нулями. Но если число типа signed int отрицательно, то при сдвиге вправо левый конец заполняется единицами, так что знак числа сохраняется.

Поразрядные операции сдвига очень полезны при декодировании выходов внешних устройств, например таких, как цифро-аналоговые преобразователи, а также при считывании информации о статусе устройств. Побитовые операторы сдвига могут быстро умножать и делить целые числа (на 2, 4 и т.д.).

Поразрядная операция отрицания (дополнения) ~ инвертирует состояние каждого бита операнда. То есть, 0 преобразует в 1, а 1 – в 0. В следующем примере оператор отрицания используется в функции шифрования символа:

char encode(char ch) // простейшая процедура шифрования

{

return(~ch); /\* оперция отрицания \*/

}

Конечно, взломать такой шифр не представляет труда.

### Операция ? (тернарный оператор)

В языке С определен мощный и удобный оператор, который часто можно использовать вместо оператора вида if-then-else. Речь идет о тернарном операторе ?, общий вид которого следующий:

Выражение1 ? Выражение2 : Выражение3;

Обратите внимание на использование двоеточия. Оператор ? работает следующим образом: сначала вычисляется *Выражение1*, если оно истинно, то вычисляется *Выражение2* и его значение присваивается всему выражению; если *Выражение1* ложно, то вычисляется *Выражение3* и всему выражению присваивается его значение. В примере

x = 10;

y = x>9 ? 100 : 200;

переменной у будет присвоено значение 100.

### Операции получения адреса и разыменования

Указатель – это адрес объекта в памяти. Переменная типа "указатель" (или просто переменная-указатель) — это специально объявленная переменная, в которой хранится указатель на переменную определенного типа.

В языке С указатели служат мощнейшим средством создания программ и широко используются для самых разных целей. Например, с их помощью можно быстро обратиться к элементам массива или дать функции возможность модифицировать свои аргументы. Указатели широко используются для связи элементов в списках, в двоичных деревьях и в других динамических структурах данных. Раздел 6 полностью посвящен указателям и работе с динамической памятью.

Оператор &, это унарный оператор, возвращающий адрес операнда в памяти. (Унарной операцией называется операция, имеющая только один операнд.) Например, оператор

int count;

int \* m = &count;

записывает в переменную m адрес переменной count.

Допустим, переменная count расположена в памяти в ячейке с адресом 2000, а ее значение равно 100. Тогда в предыдущем примере переменной m будет присвоено значение 2000.

Второй рассматриваемый оператор \* является двойственным (дополняющим) по отношению к &. Оператор \* является унарным оператором, он возвращает значение объекта, расположенного по указанному адресу. Операндом для \* служит адрес объекта (переменной). Например, если переменная m содержит адрес переменной count, то оператор

int q = \*m;

Операторы \* и & имеют более высокий приоритет, чем любая арифметическая операция, кроме унарного минуса, имеющего такой же приоритет.

### Операция определения размера sizeof

Унарная операция sizeof, выполняемая во время компиляции программы, позволяет определить длину операнда в байтах. Например, если компилятор для чисел типа int отводит 4 байта, а для чисел типа double — 8, то следующая программа напечатает 8 4.

double f;

printf("%d ", sizeof f);

printf("%d", sizeof(int));

В языке С определяется (с помощью спецификатора класса памяти typedef) специальный тип size\_t, приблизительно соответствующий целому числу без знака. Результат операции sizeof имеет тип size\_t. Но практически его можно использовать везде, где допустимо использование целого числа без знака.

Оператор sizeof очень полезен для улучшения переносимости программ, так как переносимость существенно зависит от размеров встроенных типов данных. Не рекомендуется самостоятельно делать предположения о длине операндов в байтах. Всегда используйте sizeof. Особенно учитывая, что оператор sizeof выполняется во время трансляции, его результат в программе рассматривается как константа.

### Операторы доступа к членам структуры

В языке С операторы . (точка) и -> (стрелка) обеспечивают доступ к элементам структур и объединений. Структуры и объединения — это составные типы данных, в которых под одним именем хранятся многие объекты.

Оператор точка используется для прямой ссылки на элемент структуры или объединения, т.е. перед точкой стоит имя структуры, а после — имя элемента структуры. Оператор стрелка используется с указателем на структуру или объединение, т.е. перед стрелкой стоит указатель на структуру. Например, во фрагменте программы

struct employee

{

char name[80];

int age;

float wage;

} emp;

struct employee \*p = &emp; /\* адрес emp заносится в p \*/

для присвоения члену wage значения 123.33 необходимо записать

emp.wage = 123.23;

То же самое можно сделать, использовав указатель на структуру:

p->wage = 123.23;

что то же самое

(\*p).wage = 123.23;

### Операторы [] и ()

Круглые скобки являются оператором, повышающим приоритет выполнения операций, которые в них заключены. Квадратные скобки служат для индексации массива (массивы подробно рассматриваются в разделе 5). Если в программе определен массив, то выражение в квадратных скобках представляет собой индекс массива. Например, в программе

#include <stdio.h>

char s[80];

int main(void)

{

s[3] = 'X';

printf("%c", s[3]);

return 0;

}

значение 'Х' сначала присваивается четвертому элементу массива (в С элементы массива нумеруются с нуля), затем этот элемент выводится на экран.

### Приоритеты операций

|  |  |
| --- | --- |
| Наивысший | () [] -> |
| Средний | ! ~ ++ -- - (type) \* & sizeof \* / % + - << >> < <= > >= == != & ^ | && || ?: = += -= \*= /= и т.д. |
| Наинизший | , |

## Выражения

Выражения состоят из операторов, констант, функций и переменных. В языке С выражением является любая правильная последовательность этих элементов. Большинство выражений в языке С по форме очень похожи на алгебраические, часто их и пишут, руководствуясь правилами алгебры. Однако здесь необходимо быть внимательным и учитывать специфику выражений в языке С.

### Порядок вычисления

Порядок вычисления подвыражений в выражениях языка С не определен. Компилятор может самостоятельно перестроить выражение с целью создания оптимального объектного кода. Это значит, что программист не может полагаться на определенную последовательность вычисления подвыражений. Например, при вычислении выражения

х = f1() + f2();

нет никаких гарантий того, что функция f1() будет вызвана перед вызовом f2().

### Пробелы и круглые скобки

Для повышения удобочитаемости программы при записи выражений можно использовать пробелы и символы табуляции. Например, следующие два оператора эквивалентны:

x=10/y~(127/x);

x = 10 / y ~(127/x);

Лишние скобки, если они не изменяют приоритет операций, не приводят к ошибке и не замедляют вычисление выражения. Дополнительные скобки часто используют для прояснения порядка вычислений.

x = y/3-34\*temp+127;

x = (y/3) - (34\*temp) + 127;

# Операторы

## Термины операция и оператор

Оператор – это часть программы, которая может быть выполнена отдельно. Это означает, что оператор определяет некоторое действие. В языке С существуют 6 групп операторов:

1. условные операторы;
2. операторы цикла;
3. операторы-выражения;
4. блоки;
5. операторы безусловного перехода (goto на самостоятельное обучение);
6. метки (полностью на самостоятельное обучение);

К условным относятся операторы if и switch. Иногда их также называют *операторами условного перехода*. Операторы цикла – это while, for и do-while. К операторам безусловного перехода относятся break, continue, goto и return. К меткам относятся операторы case, default(рассматриваются в разделе "Оператор выбора — switch") и собственно метки (рассматриваются в разделе "Оператор goto"). Операторы-выражения – это операторы, состоящие из допустимых выражений. Блок представляет собой фрагмент текста программы, обрамленный фигурными скобками {}. Блок иногда называют *составным оператором*.

## Условные операторы

В языке С существуют два условных оператора: if и switch. При определенных обстоятельствах оператор ? является альтернативой оператора if.

### Оператор if

Общая форма оператора if следующая:

if *(выражение) оператор;*

[else *оператор];*

Здесь оператор может быть только одним оператором, блоком операторов или отсутствовать (пустой оператор). Фраза else может вообще отсутствовать.

Если выражение истинно (т.е. принимает любое значение, отличное от нуля), то выполняется оператор или блок операторов, следующий за if. В противном случае выполняется оператор (или блок операторов), следующий за else (если эта фраза присутствует).

Условное выражение, входящее в if, должно иметь скалярный результат. Это значит, что результатом должно быть целое число, символ, указатель или число с плавающей точкой, но им не может быть массив или структура.

### Вложенные операторы if

Оператор if является вложенным, если он вложен, т.е. находится внутри другого оператора if или else. В практике программирования вложенные условные операторы используются довольно часто. Во вложенном условном операторе фраза else всегда ассоциирована с ближайшим if в том же блоке, если этот if не ассоциирован с другой фразой else. Например:

if(i)

{

if(j) statement 1;

if(k) statement 2; /\* этот if \*/

else statement 3; /\* ассоциирован с этим else \*/

}

else statement 4; /\* ассоциирован с if(i) \*/

Последняя фраза else не ассоциирована с if(j) потому, что она находится в другом блоке. Эта фраза else ассоциирована с if(i). Внутренняя фраза else ассоциирована с if(k), потому что этот if — ближайший.

### Лестница if-else-if и оператор выбора

В программах часто используется конструкция, которую называют *лестницей if-else-if*. Общая форма лестницы имеет вид

if (*выражение*) *оператор*;

else

if (*выражение*) *оператор*;

else

if (*выражение*) *оператор*;

... .

else *оператор*;

Работает эта конструкция следующим образом. Условные выражения операторов if вычисляются сверху вниз. После выполнения некоторого условия, т.е. когда встретится выражение, принимающее значение ИСТИНА, выполняется ассоциированный с этим выражением оператор, а оставшаяся часть лестницы пропускается. Если все условия ложны, то выполняется оператор в последней фразе else, а если последняя фраза else отсутствует, то в этом случае не выполняется ни один оператор.

### Оператор выбора – switch

Оператор выбора switch (часто его называют переключателем) предназначен для выбора ветви вычислительного процесса исходя из значения управляющего выражения.

switch (*выражение*) {

case *постоянная1*:

*последовательность операторов*

break;

...

case *постояннаяN*:

*последовательность операторов*

break;

default:

*последовательность операторов*;

}

В управляющем выражении можно использовать переменные целого или символьного типа, но только не с плавающей точкой. Если оператор break отсутствует, выполнение последовательности операторов продолжается до тех пор, пока не встретится break (в другой метке) или не кончится тело оператора switch.

Оператор break – это один из операторов безусловного перехода. Он может применяться не только в операторе switch, но и в циклах (рассмотрено далее).

Оператор switch отличается от if тем, что в нем управляющее выражение проверяется только на равенство с постоянными, в то время как в if проверя ется любой вид отношения или логического выражения.

## Операторы цикла

В языке С, как и в других языках программирования, операторы цикла служат для многократного выполнения последовательности операторов до тех пор, пока выполняется некоторое условие. Условие может быть установленным заранее (как в операторе for) или меняться при выполнении тела цикла (как в while или do-while).

### Цикл for

Во всех процедурных языках программирования циклы for очень похожи. Однако в С этот цикл особенно гибкий и мощный. Общая форма оператора for следующая:

for (*инициализация*; *условие*; *приращение*) *оператор*;

В наиболее общем виде принцип работы цикла for следующий. *Инициализация* – это присваивание начального значения переменной, которая называется параметром цикла. *Условие* представляет собой условное выражение, определяющее, следует ли выполнять *оператор* цикла (*тело цикла*) в очередной раз. Оператор *приращение* осуществляет изменение параметра цикла при каждой итерации. Цикл for выполняется, если выражение *условие* принимает значение ИСТИНА. Если оно хотя бы один раз примет значение ЛОЖЬ, то программа выходит из цикла и выполняется оператор, следующий за телом цикла for.

Один из распространенных способов усиления мощности цикла for – применение оператора "запятая" для создания двух параметров цикла. Оператор "запятая" связывает несколько выражений, заставляя их выполняться вместе:

for(x=0, y=0; x+y<10; ++x)

{

...

}

Для создания бесконечного цикла можно использовать любой оператор цикла, но чаще всего для этого выбирают оператор for. Так как в операторе for может отсутствовать любая секция, бесконечный цикл проще всего сделать, оставив пустыми все секции. Это хорошо показано в следующем примере:

for( ; ; ) printf("Этот цикл крутится бесконечно.\n");

Если условие цикла for отсутствует, то предполагается, что его значение – ИСТИНА.

Тело цикла м.б. пустым. Это может пригодиться, например, чтобы решить задачу удаления пробелов из входного потока.

for( ; \*str == ' '; str++) ;

### Объявление переменных внутри цикла

В стандарте С89 НЕ допускается объявление переменных в секции инициализации цикла for (в отличии от C++). В С++ объявленная таким образом переменная является локальной переменной цикла и ее область действия распространяется на тело цикла.

### Цикл while

Обшая форма цикла while имеет следующий вид:

while (*условие*) *оператор*;

Оператор выполняется, если условие принимает значение ИСТИНА.

### Цикл do-while

В отличие от циклов for и while, которые проверяют свое условие перед итерацией, do-while делает это после нее.

do {

*оператор*;

} while (*условие*);

В следующем примере в цикле do-while числа считываются с клавиатуры, пока не встретится число, меньшее или равное 100:

do {

scanf("%d", &num);

} while(num > 100);

## Операторы перехода

В языке С определены четыре оператора перехода: return, goto, break и continue. Операторы return и goto можно использовать в любом месте внутри функции. Операторы break и continue можно использовать в любом из операторов цикла. Как указывалось ранее, break можно также использовать в операторе switch.

### Оператор return

Общий вид:

return [*выражение]*;

Оператор return используется для выхода из функции. Отнесение его к категории операторов перехода обусловлено тем, что он заставляет программу перейти в точку вызова функции. Оператор return может иметь ассоциированное с ним значение, тогда при выполнении данного оператора это значение возвращается в качестве значения функции. В функциях типа void используется оператор return без значения.

Внутри функции может присутствовать произвольное количество операторов return. Выход из функции происходит тогда, когда встречается один из них.

## Оператор break

Когда внутри цикла встречается оператор break, выполнение цикла безусловно (т.е. без проверки каких-либо условий.) прекращается и управление передается оператору, следующему за ним. Например:

#include <stdio.h>

int main(void)

{

int t;

for(t=0; t<100; t++) {

printf("%d ", t);

if(t==10) break;

}

return 0;

}

Следует помнить, что оператор break вызывает выход только из внутреннего цикла.

## Оператор continue

Пример:

/\* Подсчет количества пробелов \*/

#include <stdio.h>

int main(void)

{

char s[80], \*str;

int space;

printf("Введите строку: ");

gets(s);

str = s;

for(space=0; \*str; str++) {

if(\*str != ' ') continue;

space++;

}

printf("%d пробелов\n", space);

return 0;

}

Можно сказать, что оператор continue немного похож на break. Оператор break вызывает прерывание цикла, a continue – прерывание текущей итерации цикла и осуществляет переход к следующей итерации. При этом все операторы до конца тела цикла пропускаются. В цикле for оператор continue вызывает выполнение операторов приращения.

# Массивы и строки

Массив – это структура данных, представляющая собой набор однотипных элементов, расположенных в памяти непосредственно друг за другом, доступ к которым осуществляется по индексу. В отличие от списка, массив является структурой данных с произвольным доступом к элементам.

В языке С все массивы располагаются в отдельной непрерывной области памяти. Первый элемент массива располагается по самому меньшему адресу, а последний – по самому большому. Массивы могут быть одномерными и многомерными.

Массивы и указатели тесно связаны. То, что может быть сказано о массивах, чаще всего непосредственно относится и к указателям, и наоборот. В этом разделе рассматриваются массивы, указатели будут подробно рассмотрены в разделе 6.

*Размерность массива* – это количество индексов, необходимое для однозначного доступа к элементам массива.

## Одномерные массивы

В данном подразделе рассматриваются одномерные массивы (имеющие один индекс).

Общая форма объявления одномерного массива имеет следующий вид:

*тип имя\_переменной* [*размер*];

Как и другие переменные, массив должен быть объявлен явно, чтобы компилятор выделил для него определенную область памяти (т.е. разместил массив). Здесь *тип* обозначает базовый тип массива, являющийся типом каждого элемента. *Размер* задает количество элементов массива. Например, следующий оператор объявляет массив из 100 элементов типа double под именем balance:

double balance[100];

Согласно стандарту С89 размер массива должен быть указан явно с помощью выражения-константы. Таким образом, размер массива определяется во время компиляции и впоследствии остается неизменным.

Доступ к элементу массива осуществляется с помощью имени массива и индекса. Индекс элемента массива помещается в квадратных скобках после имени. Например, оператор

balance[3] = 12.23;

присваивает 3-му элементу массива balance значение 12.23.

Индекс первого элемента любого массива в языке С равен нулю. Поэтому оператор

double balance[100];

объявляет массив символов из 10 элементов – от balance[0] до balance[9].

Объем памяти, необходимый для хранения массива, непосредственно определяется его типом и размером. Для одномерного массива количество байтов памяти вычисляется следующим образом:

количество\_байтов = sizeof(базовый\_тип) × длина\_массива

Во время выполнения программы на С не проверяется ни соблюдение границ массивов, ни их содержимое.

На рис. 5. показано, как хранится в памяти массив а, начинающийся по адресу 1000 и объявленный как

char a[7];

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Элемент | a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] |
| Адрес | 1000 | 1001 | 1002 | 1003 | 1004 | 1005 | 1006 | 1007 |

Рисунок 5. – Массив из семи символов, размещающийся по адресу 100

Указатель на 0-й элемент массива можно создать путем присваивания ему имени массива без индекса. Например:

int sample[10];

int \* p = sample;

Адрес первого элемента можно также получить, используя оператор получения адреса &. Любопытно, что выражения sample и &sample[0] имеют одно и то же значение.

В языке С нельзя передать весь массив как аргумент функции. Однако можно передать указатель на массив, т.е. имя массива без индекса. Например, в представленной программе в func1() передается указатель на массив i:

int main(void)

{

int i[10];

func1(i);

/\* ... \*/

}

в самой функции его можно объявить одним из трех вариантов: как указатель, как массив определенного размера и как массив без определенного размера, например:

void func1(int \*x) /\* указатель \*/

{

/\* ... \*/

}

или как

void func1(int x[10]) /\* массив определенного размера \*/

{

/\* ... \*/

}

и наконец как

void func1(int x[]) /\* массив без определенного размера \*/

{

/\* ... \*/

}

Эти три объявления тождественны.

## Строки

Строки символов в C представляются массивами целы типа char, заканчивающиеся символом окончания строки (нулевым символом) '\0'. Таким образом, строка содержит символы, составляющие строку, а также нулевой символ. Это единственный вид строки, определенный в С.

Объявляя массив символов, предназначенный для хранения строки, необходимо предусмотреть место для нуля, т.е. указать его размер в объявлении на один символ больше, чем наибольшее предполагаемое количество символов. Например, объявление массива str, предназначенного для хранения строки из 10 символов, должно выглядеть так:

char str[11];

Последний, 11-й байт предназначен для нулевого символа.

Записанная в тексте программы строка символов, заключенных в двойные кавычки, является строковой константой, например,

"строка"

В конец строковой константы компилятор автоматически добавляет нулевой символ.

Для обработки строк в С определено много различных библиотечных функций, многие из которых будут рассмотрены в разделе 9.1.

Рассмотрим реализацию стандартной функции strlen().

int strlen(const char \* c)

{

int i;

for (i = 0; c[i] != '\0'; ++i)

return i;

}

## Двумерные и N-мерные массивы

Стандартом С определены многомерные массивы. Простейшая форма многомерного массива — двухмерный массив. Двухмерный массив — это массив одномерных массивов. Объявление двухмерного массива d с размерами 10 на 20 выглядит следующим образом:

int d[10][20];

В следующем примере элементам двухмерного массива присваиваются числа от 1 до 12 и значения элементов выводятся на экран построчно:

#include <stdio.h>

int main(void)

{

int t, i, num[3][4];

for(t=0; t<3; ++t)

for(i=0; i<4; ++i)

num[t][i] = (t\*4)+i+1;

/\* вывод на экран \*/

for(t=0; t<3; ++t) {

for(i=0; i<4; ++i)

printf("%3d ", num[t][i]);

printf("\n");

}

return 0;

}

Наглядно двухмерный массив num из примера можно представить так:

num[t][i]

i

| 0 1 2 3

--+-----------

0 | 1 2 3 4

t 2 | 5 6 7 8

3 | 9 10 11 12

Когда к элементам массива обращаются в том порядке, в котором они размещены в памяти, правый индекс изменяется быстрее, чем левый. Размещение элементов двумерного массива Когда к элементам массива обращаются в том порядке, в котором они размещены в памяти, правый индекс изменяется быстрее, чем левый. Размещение элементов двумерного массива char ch[3][4] в схематично изображено на рисунке 5..

|  |  |  |  |
| --- | --- | --- | --- |
| Правый индекс определяет номер столбца | | | |
| Левый индекс определяет номер строки | ch[0][0] | ch[0][1] | ch[0][2] |
|  | | |
| ch[1][0] | ch[1][1] | ch[1][2] |
|  | | |
| ch[2][0] | ch[2][1] | ch[2][2] |
|  | | |
| ch[3][0] | ch[3][1] | ch[3][2] |

Рисунок 5.2 – Размещение двумерного массива в памяти

При передаче двумерного массива в функцию в соответствующем параметре функции обязательно должен быть указан размер правого измерения:

void func1(int x[][10])

{

/\* ... \*/

}

Компилятор должен знать длину строки массива, чтобы внутри функции правильно вычислить адрес элемента массива. Если при компиляции функции это неизвестно, то невозможно определить, где начинается следующая строка, и вычислить, например, адрес элемента

x[2][4]

В языке С можно пользоваться массивами, размерность которых больше двух. Общая форма объявления многомерного массива следующая:

*тип имя\_массива* [*Размер1*][*Размер2*]...[*РазмерN*];

Массивы, у которых число измерений больше трех, используются довольно редко, потому что они занимают большой объем памяти. Например, четырехмерный массив символов размерностью 10x6x9x4 занимает 2160 байтов. Если бы массив содержал 2-байтовые целые, потребовалось бы 4320 байтов. Если бы элементы массива имели типdouble, причем каждый элемент (вещественное число двойной точности) занимал бы 8 байтов, то для хранения массива потребовалось бы 17280 байтов. Объем требуемой памяти с ростом числа измерений растет экспоненциально.

При обращении к многомерным массивам компьютер много времени затрачивает на вычисление адреса, так как при этом приходится учитывать значение каждого индекса. Поэтому доступ к элементам многомерного массива происходит значительно медленнее, чем к элементам одномерного.

Передавая многомерный массив в функцию, в объявлении параметров функции необходимо указать все размеры измерений, кроме самого левого. Например, если массив m объявлен как

int m[4] [3] [6] [5];

то функция, принимающая этот массив, должна быть объявлена примерно так:

void func1(int d[][3][6][5])

{

/\* ... \*/

}

## Индексация указателей

Указатели и массивы тесно связаны друг с другом. Имя массива без индекса – это указатель на первый (начальный) элемент массива. Рассмотрим, например, следующий массив:

char p[10];

Следующие два выражения идентичны:

p

&p[0]

Как уже указывалось, имя массива без индекса представляет собой указатель. И наоборот, указатель можно индексировать как массив. Рассмотрим следующий фрагмент программы:

int \*p, i[10];

p = i;

p[5] = 100; /\* в присваении используется индекс \*/

\*(p+5) = 100; /\* в присвоении используется адресная арифметика \*/

Оба оператора присваивания заносят число 100 в 6-й элемент массива i.

Двухмерный массив может быть представлен как указатель на массив одномерных массивов. Добавив еще один указатель, можно с его помощью обращаться к элементам отдельной строки массива. Этот прием демонстрируется в функцииpr\_row(), которая печатает содержимое конкретной строки двухмерного глобального массива num:

int num[10][10];

/\* ... \*/

void pr\_row(int j)

{

int \*p, t;

p = (int \*) &num[j][0]; /\* вычисление адреса 1-го

элемента строки номер j \*/

for(t=0; t<10; ++t) printf("%d ", \*(p+t));

}

Эту функцию можно обобщить, включив в список аргументов номер строки, длину строки и указатель на 1-й элемент:

void pr\_row(int j, int row\_dimension, int \*p)

{

int t;

p = p + (j \* row\_dimension);

for(t=0; t<row\_dimension; ++t)

printf("%d ", \*(p+t));

}

/\* ... \*/

void f(void)

{

int num[10][10];

pr\_row(0, 10, (int \*) num); /\* печать 1-й строки \*/

}

## Инициализация массивов

В языке С массивы при объявлении можно инициализировать. Общая форма инициализации массива аналогична инициализации переменной:

*тип имя\_массива*[*размер1*]...[*размер№*] = {*список\_значений*};

*список\_значений* представляет собой список констант, разделенных запятыми. Типы констант должны быть совместимыми с *типом* массива. Первая константа присваивается первому элементу массива, вторая — второму и так далее. После закрывающейся фигурной скобки точка с запятой обязательна.

В следующем примере массив целых из 10 элементов инициализируется числами от 1 до 5:

int i[5] = {1, 2, 3, 4, 5};

Символьные массивы, содержащие строки, можно инициализировать строковыми константами:

char *имя\_массива*[*размер*] = "*строка*";

Например:

char str[7] = "Язык C";

Это объявление можно записать так:

char str[7] = {'Я', 'з', 'ы', 'к', ' ', 'C', '\0'};

Строка кончается нулевым символом, поэтому при объявлении необходимо задавать размер массива, достаточный для того, чтобы этот символ поместился в нем. Если строка инициализируется строковой константой, компилятор автоматически добавляет нулевой символ в конце строки.

Многомерные массивы инициализируются так же, как и одномерные. В следующем примере массив sqrs инициализируется числами от 1 до 5 и их квадратами:

int sqrs[5][2] = {

1, 1,

2, 4,

3, 9,

4, 16,

5, 25

}

Инициализируя многомерный массив, для улучшения наглядности элементы инициализации каждого измерения можно заключать в фигурные скобки.

int sqrs[5][2] = {

{1, 1},

{2, 4},

{3, 9},

{4, 16},

{5, 25}

}

При такой записи, если внутри группы недостаточно констант инициализации, то оставшиеся элементы группы автоматически заполняются нулями.

В языке С есть конструкция, благодаря которой компилятор автоматически определяет необходимую длину строки при *инициализации* *безразмерного массива*, например:

char str[7] = "Язык C";

Тогда оператор

printf("%s имеет длину %d\n", str, sizeof str);

В многомерном массиве размер самого левого измерения также можно не указывать (размеры по остальным измерениям обязательно должны быть указаны, так как это нужно компилятору для определения длины подмассивов, составляющих массив).

int sqrs[][2] = {

{1, 1},

{2, 4},

{3, 9},

{4, 16},

{5, 25}

}

Преимущество безразмерного объявления массива состоит в том, что можно изменять длину таблицы, не заботясь о размере массива.

## Массивы структур

Структуры часто образуют массивы. Чтобы объявить массив структур, вначале необходимо определить структуру (то есть определить агрегатный тип данных), а затем объявить переменную массива этого же типа. Например, чтобы объявить 100-элементный массив структур типа addr, который был определен ранее, напишите следующее:

struct addr

{

char name[30];

char street[40];

char city[20];

char state[3];

unsigned long int zip;

};

struct addr addr\_list[100];

Чтобы получить доступ к определенной структуре, указывайте имя массива с индексом. Например, чтобы вывести ZIP-код из третьей структуры, напишите следующее:

printf("%d", addr\_list[2].zip);

# Указатели и динамическое распределение памяти

## Что такое указатели

Правильное понимание и использование указателей необходимо для составления хороших программ на языке С по следующим причинам:

1. указатели являются единственным средством, с помощью которого функция может изменять значения передаваемых в нее аргументов;
2. указатели позволяют повысить эффективность передачи аргументов в функции;
3. с помощью указателей выполняется динамическое распределение памяти;
4. указатели обеспечивают поддержку динамических структур данных, таких, например, как двоичные деревья и связные списки.

Указатель – это переменная, значением которой является адрес некоторого объекта (обычно другой переменной) в памяти компьютера. Например, если одна переменная содержит адрес другой переменной, то говорят, что первая переменная указывает (ссылается) на вторую (рис 6.).

Адрес Значение

ячейки переменной в

памяти памяти

+----------+

1000 | 1003 |--.

+----------+ |

1001 | | |

+----------+ |

1002 | | |

+----------+ |

1003 | |<-'

+----------+

1004 | |

+----------+

1005 | |

+----------+

1006 | |

+----------+

.

.

.

Память

Рисунок рис 6. – Одна переменная ссылается на другую

Объявление указателя состоит из имени базового типа, символа \* и имени переменной. Общая форма объявления указателя следующая:

*тип \* имя*;

Базовый тип указателя определяет тип объекта, на который указатель будет ссылаться. Фактически указатель любого типа может ссылаться на любое место в памяти. Однако выполняемые с указателем операции существенно зависят от его типа. Например, если объявлен указатель типа int \*, компилятор предполагает, что любой адрес, на который он ссылается, содержит переменную типа int, хоть это может быть и не так. Следовательно, объявляя указатель, необходимо убедиться, что его тип совместим с типом объекта, на который он будет ссылаться.

## Операции для работы с указателями

Операции для работы с указателями рассматривались в подразделе 3.1. В языке С определены две операции для работы с указателями: \* и &. Оператор & – это унарный оператор, возвращающий адрес своего операнда. (Напомним, что унарный оператор имеет один операнд). Оператор, \* выполняет действие, обратное по отношению к &.

## Выражения с указателями и адресная арифметика

В общем случае выражения с указателями подчиняются тем же правилам, что и обычные выражения. В этом подразделе рассматривается применение указательных выражений в операциях присваивания, преобразования типов, а также в операциях "указательной" арифметики.

Указатель можно использовать в правой части оператора присваивания для присваивания его значения другому указателю.

Указатель можно преобразовать к другому типу. Эти преобразования бывают двух видов: с использованием указателя типа void \* и без его использования. В языке С допускается присваивание указателя типа void \* указателю любого другого типа (и наоборот) без явного преобразования типа указателя. Тип указателя void \* используется, если тип объекта неизвестен. Например, использование типа void \* в качестве параметра функции позволяет передавать в функцию указатель на объект любого типа, при этом сообщение об ошибке не генерируется. Также он полезен для ссылки на произвольный участок памяти, независимо от размещенных там объектов. Например, функция размещения malloc() (рассматривается далее в этой главе) возвращает значение типа void \*, что позволяет использовать ее для размещения в памяти объектов любого типа.

В отличие от void \*, преобразования всех остальных типов указателей должны быть всегда явными (т.е. должна быть указана операция приведения типов). Однако следует учитывать, что преобразование одного типа указателя к другому может вызвать непредсказуемое поведение программы. Например, в следующей программе делается попытка присвоить значение х переменной у посредством указателя р. При компиляции программы сообщение об ошибке не генерируется, однако результат работы программы неверен.

#include <stdio.h>

int main(void)

{

double x = 100.1, y;

/\* В следующем операторе указателю на целое p

(присваивается значение, ссылающееся на double. \*/

int \* p = (int \*) &x;

/\* Следующий оператор работает не так, как ожидается. \*/

y = \*p; /\* attempt to assign y the value x through p \*/

/\* Следующий оператор не выведет число 100.1. \*/

printf("Значение x равно: %f", y);

return 0;

}

*Примечание:* В языке C++ требуется явно указывать преобразование типа указателей, в том числе указателей типа void \*. Поэтому рекомендуется использовать в языке С явное преобразование для совместимости с C++.

В языке С допустимы только две арифметические операции над указателями: суммирование и вычитание. Предположим, текущее значение указателя p1 типа int \* равно 2000. Предположим также, что переменная типа int занимает в памяти 2 байта. Тогда после операции увеличения

p1++;

указатель p1 принимает значение 2002, а не 2001. То есть, при увеличении на 1 указатель p1 будет ссылаться на следующее целое число. Это же справедливо и для операции уменьшения. Например, если p1 равно 2000, то после выполнения оператора

p1--;

значение p1 будет равно 1998.

Операции адресной арифметики подчиняются следующим правилам. После выполнения операции увеличения над указателем, данный указатель будет ссылаться на следующий объект своего базового типа. После выполнения операции уменьшения – на предыдущий объект. Применительно к указателям на char, операций адресной арифметики выполняются как обычные арифметические операции, потому что длина объекта char всегда равна 1. Для всех указателей адрес увеличивается или уменьшается на величину, равную размеру объекта того типа, на который они указывают.

Кроме суммирования и вычитания указателя и целого, разрешена еще только одна операция адресной арифметики: можно вычитать два указателя. Благодаря этому можно определить количество объектов, расположенных между адресами, на которые указывают данные два указателя; правда, при этом считается, что тип объектов совпадает с базовым типом указателей. Все остальные арифметические операции запрещены. А именно: нельзя делить и умножать указатели, суммировать два указателя, выполнять над указателями побитовые операции, суммировать указатель со значениями, имеющими тип float или double и т.д.

Стандартом С допускается сравнение двух указателей. Например, если объявлены два указателя р и q, то следующий оператор является правильным:

if(p < q) printf("p ссылается на меньший адрес, чем q\n");

Как правило, сравнение указателей может оказаться полезным, только тогда, когда два указателя ссылаются на общий объект, например, на массив. В качестве примера рассмотрим программу с двумя стековыми функциями, предназначенными для записи и считывания целых чисел. Стек – это список, использующий систему доступа (LIFO – “last in – first out”) "последним вошел – первым вышел". Иногда стек сравнивают со стопкой тарелок на столе: первая, поставленная на стол, будет взята последней. Стеки часто используются в компиляторах, интерпретаторах, программах обработки крупноформатных таблиц и в других системных программах. Для создания стека необходимы две функции: push() и pop(). Функция push() заносит числа в стек, a pop() – извлекает их.

#define SIZE 50

int \* base, \* top, stack[SIZE];

void push(int i)

{

if(top == (base+SIZE+1)) {

printf("Переполнение стека.\n");

return;

}

top++;

\*top = i;

}

int pop()

{

if (top == base) {

printf("Стек пуст.\n");

return 0;

}

p1--;

return \*(p1+1);

}

/\*Использование\*/

int main(void)

{

int value;

top = base = stack; /\* инициализация стека \*/

do {

printf("Введите значение: ");

scanf("%d", &value);

if(value != 0) push(value);

else printf("значение на вершине равно %d\n", pop());

} while(value != -1);

return 0;

}

## Указатели и массивы

Понятия указателей и массивов тесно связаны. Рассмотрим следующий фрагмент программы:

char str[80], \*p1;

p1 = str;

Здесь p1 указывает на первый элемент массива str. Обратиться к пятому элементу массива str можно с помощью любого из двух выражений:

str[4]

\* (p1+4)

В языке С существуют два метода обращения к элементу массива: адресная арифметика и индексация массива. Пример:

/\* Индексация указателя s как массива. \*/

void putstr(char \*s)

{

int t;

for(t=0; s[t]; ++t) putchar(s[t]);

}

/\* Использование адресной арифметики. \*/

void putstr(char \*s)

{

while(\*s) putchar(\*s++);

}

Большинство профессиональных программистов сочтут вторую версию более наглядной и удобной, поэтому в процедурах такого типа приемы адресной арифметики используются довольно часто.

## Массивы указателей

Как и объекты любых других типов, указатели могут быть собраны в массив. В следующем операторе объявлен массив из 10 указателей на объекты типа int:

int \*x[10];

Для присвоения, например, адреса переменной var третьему элементу массива указателей, необходимо написать:

x[2] = &var;

В результате этой операции, следующее выражение принимает то же значение, что и var:

\*x[2]

Массивы указателей часто используются при работе со строками. Например, можно написать функцию, выводящую нужную строку с сообщением об ошибке по индексу num:

void syntax\_error(int num)

{

static char \*err[] = {

"Нельзя открыть файл\n",

"Ошибка при чтении\n",

"Ошибка при записи\n",

"Некачественный носитель\n"

};

printf("%s", err[num]);

}

## Инициализация указателей

После объявления нестатического локального указателя до первого присвоения он содержит неопределенное значение. Если попытаться использовать указатель перед присвоением ему нужного значения, то скорее всего он мгновенно разрушит программу или всю операционную систему. Это очень досадная ошибка.

При работе с указателями большинство программистов придерживаются следующего важного соглашения: указатель, не ссылающийся в текущий момент времени должным образом на конкретный объект, должен содержать нулевое значение. Нуль используется потому, что С гарантирует отсутствие чего-либо по нулевому адресу. Следовательно, если указатель равен нулю, то это значит, во-первых, что он ни на что не ссылается, а во-вторых — что его сейчас нельзя использовать.

Указателю можно задать нулевое значение, присвоив ему 0. Например, следующий оператор инициализирует р нулем:

char \*p = 0;

Дополнительно к этому во многих заголовочных файлах языка С, например, в<stdio.h> определен макрос NULL, являющийся нулевой указательной константой. Поэтому в программах на С часто можно увидеть следующее присваивание:

p = NULL;

В программах на С указатель типа char \* часто инициализируют строковой константой (как в предыдущем примере). Рассмотрим следующий пример:

char \*p = "строка";

Переменная р является указателем, а не массивом. Поэтому возникает логичный вопрос: где хранится строковая константа "тестовая строка"? Так как p не является массивом, она не может храниться в p, тем не менее, она где-то записана. Чтобы ответить на этот вопрос, нужно знать, что происходит, когда компилятор встречает строковую константу. Компилятор создает так называемую *таблицу строк*, в ней он сохраняет строковые константы, которые встречаются ему по ходу чтения текста программы. Следовательно, когда встречается объявление с инициализацией, компилятор сохраняет строку "тестовая строка" в таблице строк, а в указатель p записывает ее адрес. Дальше в программе указатель p может быть использован как любая другая строка.

## Многоуровневая адресация

Иногда указатель может ссылаться на указатель, который ссылается на обычную переменную. Это называется *многоуровневой адресацией*. Иногда применение таких указателей существенно усложняет программу, делает ее плохо читаемой и подверженной ошибкам. Рисунок 6.X иллюстрирует концепцию многоуровневой адресации. На рисунке видно, что значением "нормального" указателя является адрес объекта, содержащего нужное значение. В случае двухуровневой адресации первый указатель содержит адрес второго указателя, который содержит адрес объекта с нужным значением.

Адрес

Указатель

Адрес

Указатель

Значение

Объект

**Многоуровневая адресация**

Адрес

Указатель

Значение

Объект

**Одноуровневая адресация**

Рисунок 6.X – Одноуровневая и многоуровневая схемы адресации

Многоуровневая адресация может иметь сколько угодно уровней, однако уровни глубже второго, т.е. указатели более глубокие, чем "указатели на указатели" применяются крайне редко. Дело в том, что при использовании таких указателей часто встречаются концептуальные ошибки из-за того, что смысл таких указателей представить трудно.

Пример указателя на указатель:

float \*\*newbalance;

При двухуровневой адресации для доступа к значению объекта нужно поставить перед идентификатором две звездочки:

int x, \*p, \*\*q;

x = 10;

p = &x;

q = &p;

printf("%d", \*\*q); /\* печать значения x \*/

## Указатели на функции

*Указатели на функции* – очень мощное средство языка С. Хотя нельзя не отметить, что это весьма трудный для понимания термин. Функция располагается в памяти по определенному адресу, который можно присвоить указателю в качестве его значения. Адресом функции является ее точка входа. Именно этот адрес используется при вызове функции. Так как указатель хранит адрес функции, то она может быть вызвана с помощью этого указателя. Он позволяет также передавать ее другим функциям в качестве аргумента.

#include <stdio.h>

#include <string.h>

void check(char \*a, char \*b,

int (\*cmp)(const char \*, const char \*));

int main(void)

{

char s1[80], s2[80];

int (\*p)(const char \*, const char \*);

/\* указатель на функцию \*/

p = strcmp;

/\* присваивает адрес функции strcmp указателю p \*/

printf("Введите две строки.\n");

gets(s1);

gets(s2);

check(s1, s2, p); /\* Передает адрес функции strcmp

посредством указателя p \*/

return 0;

}

void check(char \*a, char \*b,

int (\*cmp)(const char \*, const char \*))

{

printf("Проверка на совпадение.\n");

if(!(\*cmp)(a, b)) printf("Равны");

else printf("Не равны");

}

Может возникнуть вопрос: какая польза от вызова функции с помощью указателя на функцию? Ведь в данном случае никаких преимуществ не достигнуто, этим мы только усложнили программу. Тем не менее, во многих случаях оказывается более выгодным передать имя функции как параметр или даже создать массив функций. Например, в программе интерпретатора синтаксический анализатор (программа, анализирующая выражения) часто вызывает различные вспомогательные функции, такие как вычисление математических функций, процедуры ввода-вывода и т.п. В таких случаях чаще всего создают список функций и вызывают их с помощью индексов.

Альтернативный подход – использование оператора switch с длинным списком меток case – делает программу более громоздкой и подверженной ошибкам.

## Динамическое распределение памяти

Указатели используются для динамического выделения памяти компьютера для хранения данных. *Динамическое распределение* означает, что программа выделяет память для данных во время своего выполнения.

Память для глобальных переменных выделяется во время компиляции, а для нестатических локальных переменных – в стеке. Локальные переменные существуют на стеке, до тех пор, пока мы не вышли из функции (или из блока). С одной стороны, это позволяет автоматически очищать память, с другой стороны, существует необходимость в переменных, время жизни которых мы можем контролировать самостоятельно.

Кроме того, нам необходимо динамическое распределение памяти, когда размер используемого пространства заранее неизвестен. Такое случается, например, при использовании динамических структур данных, таких как связные списки или двоичные деревья. Такие структуры данных при выполнении программы расширяются или сокращаются по мере необходимости. Для реализации таких структур в программе нужны средства, способные по мере необходимости выделять и освобождать для них память.

Память, выделяемая в С функциями динамического распределения данных, находится в т.н. *динамически распределяемой области памяти (heap)*. Динамически распределяемая область памяти – это свободная область памяти, не используемая программой, операционной системой или другими программами. Размер динамически распределяемой области памяти заранее неизвестен, но как правило в ней достаточно памяти для размещения данных программы. Большинство компиляторов поддерживают библиотечные функции, позволяющие получить текущий размер динамически распределяемой области памяти, однако эти функции не определены в Стандарте С. Хотя размер динамически распределяемой области памяти очень большой, все же она конечна и может быть исчерпана.

Для работы с динамической памятью в Си существуют функции:

malloc()

calloc()

realloc()

free()

Функции malloc() и calloc () выделяют память, а free() – освобождает ее. Функция realloc() предназначена для перераспределения (увеличения или уменьшения) имеющегося объема памяти.

Прототип функции malloc() следующий:

void \* malloc(size\_t *количество\_байтов*);

Здесь *количество\_байтов* — размер памяти, необходимой для размещения данных. (Тип size\_t определен в <stdlib.h> как некоторый целый без знака.) Функция malloc() возвращает указатель типа void \*, поэтому его можно присвоить указателю любого типа. При успешном выполнении malloc() возвращает указатель на первый байт непрерывного участка памяти, выделенного в динамически распределяемой области памяти. Если в динамически распределяемой области памяти недостаточно свободной памяти для выполнения запроса, то память не выделяется и malloc() возвращает нуль. Например:

char \* p = malloc(100 \* sizeof(int)); /\* выделение памяти для 100 int’ов \*/

if(!p)

{

printf("Нехватка памяти.\n");

exit(1);

}

Как показано в примере, поскольку динамически распределяемая область памяти не бесконечна, при каждом размещении данных необходимо проверять, состоялось ли оно.

Прототип функции сalloc() следующий:

void \* сalloc(size\_t *num, size\_t size*);

Функция calloc() выделяет num объектов размером size и заполняет их нулями. В остальном, она аналогична функции malloc(). Обычно calloc() используется для выделения памяти под массивы.

Функция free() противоположна функции malloc() в том смысле, что она возвращает системе участок памяти, выделенный ранее с помощью функции malloc().Функция free() имеет следующий прототип:

void free(void \**p*)

Здесь р – указатель на участок памяти, выделенный перед этим функцией malloc(). Функцию free() ни в коем случае нельзя вызывать с неправильным аргументом, это может разрушить всю систему распределения памяти.

Еще одна важная функция – realloc() (re-allocation). Она позволяет изменить размер ранее выделенной области памяти и получает в качестве аргументов старый указатель и новый размер памяти в байтах. Ее прототип:

void \* realloc(void \* ptr*, size\_t size*);

Функция realloc() возвращает указатель на измененный участок памяти. Функция realloc() может использовать как ранее выделенный участок памяти, так и новый. При этом не важно, меньше или больше новый размер – менеджер памяти сам решает, где выделять память.

Довольно часто возникает необходимость выделить память динамически, используя malloc(), но работать с этой памятью удобнее так, будто это массив, который можно индексировать. В этом случае нужно создать *динамический массив*. Сделать это несложно, потому что каждый указатель можно индексировать как массив. В следующем примере одномерный динамический массив содержит строку:

/\* Динамическое распределение строки, строка вводится

пользователем, а затем распечатывается справа налево. \*/

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

int main(void)

{

char \*s;

register int t;

s = malloc(80);

if(!s) {

printf("Требуемая память не выделена.\n");

exit(1);

}

gets(s);

for(t=strlen(s)-1; t>=0; t--) putchar(s[t]);

free(s);

return 0;

}

Некоторые особенности функций динамического распределения памяти:

* вызов realloc(0, size) эквивалентен вызову malloc(size);
* вызов free(0) не производит никаких действий;
* поведение функций malloc(), calloc(), realloc() при вызове их с нулевым размером буфера не определено.

При использовании этих функций следует учитывать, что операции распределения/освобождения памяти обычно являются затратными по времени выполнения, поэтому не следует выполнять их чрезмерно часто.

## Трудности при работе с указателями и динамически-распределяемой памятью.

Возможности указателей огромны, однако, их использование потенциально позволяет допустить серьезные ошибки, которые будет весьма трудно обнаружить.

Ошибочный указатель трудно найти потому, что ошибка в самом указателе никак себя не проявляет. Проблемы возникают при попытке обратиться к объекту с помощью этого указателя. Если значение указателя неправильное, то программа с его помощью обращается к произвольной ячейке памяти. При чтении в программу попадают неправильные данные, а при записи искажаются другие данные, хранящиеся в памяти, или портится участок программы, не имеющий никакого отношения к ошибочному указателю. В обоих случаях ошибка может не проявиться вовсе или проявиться позже в форме, никак не указывающей на ее причину.

/\* Это программа содержит ошибку. \*/

int main(void)

{

int x, \*p;

x = 10;

\*p = x; /\* ошибка, p не инициализирован \*/

return 0;

}

В таком простом случае большинство компиляторов выводят предупреждение о том, что используется неинициализированный указатель. Однако подобная ошибка может произойти и в более завуалированном виде, тогда компилятор не сможет распознать ее.

Неприятные ошибки возникают при *повторном освобождении памяти*, также распространенный вид ошибок возникает, когда теряется указатель на неосвобожденную область памяти (при этом возникает т.н. *утечка памяти*). Утечка памяти как правило возникает, когда программист забывает вызвать функцию free() при выходе из подпрограммы. Пример:

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

int main(void)

{

char \*s;

register int t;

s = malloc(80);

if(!s) return 1;

gets(s);

if (s == ‘\0’)

{

return 2; // memory leak!!!

}

for(t=strlen(s)-1; t>=0; t--) putchar(s[t]);

free(s);

return 0;

}

Утечки памяти опасны тем, что они никак не проявляются до тех пор, пока впустую занята вся доступная память. Порой такие ошибки могут не проявляться годами.

Похожий вид ошибок возникает, когда производится обращение по указателю, после освобождения памяти (т.е. после вызова free()). Когда мы вызываем функцию free, то мы освобождаем память, т.е. говорим компьютеру, что эта память м.б. использована кем-то другим. Он может использовать эту память, а может и нет. Но теперь у нас нет гарантии, что эта память наша. При этом сам указатель не зануляется. Он продолжает хранить адрес, которым мы ранее пользовались. Это очень похоже на съем комнаты. Мы получаем дубликат ключа от комнаты, живем в ней, а потом сдаем комнату обратно. Но дубликат ключа у нас остается. Всегда можно зайти в эту комнату, но в ней уже кто-то может жить. Так что наша обязанность – удалить дубликат.

Чтобы избежать таких ошибок, принято обнулять указатель после вызова free(). В следующем примере обнуление указателя используется, для указания того, что стек не инициализирован:

static int \* p = 0;

static int count = 0;

void initStack(int stack\_size) /\*удаление стека\*/

{

if (p) free(p);

p = malloc(stack\_size \* sizeof(int));

count = 0;

}

void push(int value)

{

if (count == stack\_size) return; // переполнение

p[count] = value;

++count;

}

int pop()

{

if (count == 0) return -1; // стек пуст

--count;

return p[count];

}

void freeStack() /\*Удаление стека\*/

{

if (p)

{

free(p);

p = 0; // обнуляем указатель

}

}

# Функции

## Понятие функции

Функции — это строительные элементы языка С и то место, в котором выполняется вся работа программы. В этой главе изучаются свойства функций, в том числе их аргументы, возвращаемые значения, прототипы, а также рекурсия.

В общем виде функция выглядит следующим образом:

*возвр-тип имя-функции(список параметров)*

{

*тело функции*

}

*возвр-тип* определяет тип данного, возвращаемого функцией. Функция может возвращать любой тип данных, за исключением массивов *список параметров* — это список, элементы которого отделяются друг от друга запятыми. Каждый такой элемент состоит из имени переменной и ее типа данных. При вызове функции параметры принимают значения аргументов. Функция может быть и без параметров, тогда их список будет пустым. Такой пустой список можно указать в явном виде, поместив для этого внутри скобок ключевое слово void.

## Аргументы функции

**Формальный  
параметр**

Код

Функция

**Аргумент**

**(фактический параметр)**

В общем виде список объявлений параметров должен выглядеть следующим образом:

*f(тип имя\_переменной1, тип имя\_переменной2,..., тип имя\_переменнойN*)

Код, который составляет тело функции, скрыт от остальной части программы, и если он не использует глобальных переменных, то не может воздействовать на другие части программы или, наоборот, подвергаться воздействию с их стороны. Иначе говоря, код и данные, определенные внутри одной функции, без глобальных переменных не могут воздействовать на код и данные внутри другой функции, так как у любых двух разных функций разные области видимости.

Переменные, определенные внутри функции, являются локальными. Локальная переменная создается в начале выполнения функции, а при выходе из этой функции она уничтожается. Таким образом, локальная переменная не может сохранять свое значение в промежутках между вызовами функции.

Формальные параметры функции также как и локальные переменные находятся в ее области видимости. Это значит, что параметр доступен внутри всей функции. Параметр создается в начале выполнения функции, и уничтожается при выходе из нее. Хотя параметры выполняют специальную задачу – принимают значения аргументов, передаваемых функции, – они все равно ведут себя так, как и другие локальные переменные.

В языках программирования имеется два способа передачи значений подпрограмме: *по значению* и *по ссылке*.

При применении передачи аргумента *по значению* в формальный параметр подпрограммы копируется значение аргумента. В таком случае изменения параметра на аргумент не влияют.

При применении передачи аргумента *по ссылке* в параметр копируется адрес аргумента. Это значит, что, в отличие от вызова по значению, изменения значения параметра приводят к точно таким же изменениям значения аргумента.

Пример передачи аргумента по значению:

int sqr(int x)

{

x = x\*x;

return(x);

}

int main(void)

{

int t=10;

printf("%d %d", sqr(t), t);

return 0;

}

Пример передачи аргумента по ссылке:

void swap(int \*x, int \*y)

{

int temp = \*x; /\* сохранить значение по адресу x \*/

\*x = \*y; /\* поместить y в x \*/

\*y = temp; /\* поместить x в y \*/

}

int main(void)

{

int i = 10, j = 20;

printf("i и j перед обменом: %d %d\n", i, j);

swap(&i, &j); /\* передать адреса переменных i и j \*/

printf("i и j после обмена: %d %d\n", i, j);

return 0;

}

*Примечание 1:* Язык C++ при помощи параметров-ссылок дает возможность полностью автоматизировать вызов по ссылке. А в языке С параметры-ссылки не поддерживается

*Примечание 2:* Когда в качестве аргумента функции используется массив, то функции передается его адрес. В этом и состоит исключение по отношению к правилу, которое гласит, что при передаче параметров используется вызов по значению.

## Возврат значений из функции

Все функции, кроме тех, которые относятся к типу void, возвращают значение. Это значение указывается выражением в операторе return. Стандарт С89 допускает выполнение оператора return без указания выражения внутри функции, тип которой отличен от void. В этом случае все равно происходит возврат какого-нибудь произвольного значения, но так делать не следует.

Если функция не объявлена как имеющая тип void, она может использоваться как операнд в выражении. Например:

x = power(y);

if(max(x,y) > 100) printf("больше");

for(ch=getchar(); isdigit(ch); ) ... ;

## Функция main()

Функция main() служит в качестве начальной точки выполнения программы. Она обычно управляет выполнением программы, вызывая другие ее функции. Выполнение программы, как правило завершается в конце блока функции main().

Функция main() возвращает целое число, которое принимает вызывающий процесс – обычно этим процессом является операционная система. Возврат значения из main() эквивалентен вызову функции exit() с тем же самым значением.

Иногда при запуске программы бывает полезно передать ей какую-либо информацию. Обычно такая информация передается функции main() с помощью аргументов командной строки. Аргумент командной строки – это информация, которая вводится в командной строке операционной системы вслед за именем программы. Например, чтобы запустить компиляцию программы, необходимо в командной строке набрать примерно следующее:

cc *имя\_программы*

*имя\_программы* представляет собой аргумент командной строки, он указывает имя той программы, которую вы собираетесь компилировать.

*имя\_программы* представляет собой аргумент командной строки, он указывает имя той программы, которую вы собираетесь компилировать.

Чтобы принять аргументы командной строки, используются два специальных встроенных аргумента: argc и argv. Параметр argc содержит количество аргументов в командной строке и является целым числом, причем он всегда не меньше 1, потому что первым аргументом считается имя программы. А параметр argv является массивом указателей на строки. В этом массиве каждый элемент указывает на какой-либо аргумент командной строки. Все аргументы командной строки являются строковыми, поэтому преобразование каких бы то ни было чисел в нужный двоичный формат должно быть предусмотрено в программе при ее разработке.

Вот простой пример использования аргумента командной строки. На экран выводятся слово Привет и ваше имя, которое надо указать в виде аргумента командной строки.

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char \*argv[])

{

if(argc!=2) {

printf("Вы забыли ввести свое имя.\n");

exit(1);

}

printf("Привет %s", argv[1]);

return 0;

}

Если вы назвали эту программу name и ваше имя «Вася», то для запуска программы следует ввести в командную строку: name Вася.

В некоторых компиляторах для main() могут поддерживаться-дополнительные аргументы, чтобы в них разобраться необходимо изучить документацию к вашему компилятору.

Когда для программы не требуются параметры командной строки, то чаще всего декларируют функцию main() как не имеющую параметров.

## Рекурсия

В языке С функция может вызывать сама себя. В этом случае такая функция называется *рекурсивной*. Рекурсия — это процесс определения чего-либо на основе самого себя, из-за чего рекурсию еще называют *рекурсивным определением*.

Простым примером рекурсивной функции является factr(), которая вычисляет факториал целого числа.

/\* рекурсивная функция \*/

int factr(int n)

{

int answer;

if(n==1) return(1);

return factr(n-1)\*n; /\* рекурсивный вызов \*/

}

Рекурсия, когда рекурсивный вызов проводится один раз в конце функции, называется *хвостовой рекурсией*. Она не приносит никаких выгод, от нее следует избавиться, переписав функцию с использованием цикла:

/\* неркурсивная функция \*/

int fact(int n)

{

int t, answer = 1;

for(t=1; t<=n; t++)

answer = answer\*(t);

return(answer);

}

Когда функция вызывает сама себя, новый набор локальных переменных и параметров размещается в памяти в стеке, а код функции выполняется с самого своего начала, причем используются именно эти новые переменные. При рекурсивном вызове функции новая копия ее кода не создается. Новыми являются только значения, которые использует данная функция. При каждом возвращении из рекурсивного вызова старые локальные переменные и параметры извлекаются из стека, и сразу за рекурсивным вызовом возобновляется работа функции. При использовании рекурсивных функций стек работает подобно "телескопической" трубе, выдвигающейся вперед и складывающейся обратно.

Хотя и кажется, что рекурсия предлагает более высокую эффективность, но на самом деле такое бывает достаточно редко. Использование рекурсии в программах зачастую не очень сильно уменьшают их размер кода и обычно только незначительно увеличивает эффективность использования памяти. Кроме того, рекурсивные версии большинства программ могут выполняться несколько медленнее, чем их итеративные варианты, потому что при рекурсивных вызовах функций расходуются дополнительные ресурсы. Кроме того, большое количество рекурсивных вызовов функции может вызвать переполнение стека. Из-за того, что память для параметров функции и локальных переменных находится в стеке и при каждом новом вызове создается еще один набор этих переменных, то для переменных места в стеке может рано или поздно не хватить. Переполнение стека — вот обычная причина аварийного завершения программы, когда функция утрачивает контроль над рекурсивными обращениями.

Главным преимуществом рекурсивных функций является то, что с их помощью упрощается реализация некоторых алгоритмов, а программа становится понятнее. Например, алгоритм быстрой сортировки трудно реализовать итеративным способом. Работу с рекурсивными структурами данных гораздо проще организовать с помощью рекурсии, например, обход бинарного дерева (сверху-вниз и справа-налево):

В следующем примере обход нагруженного целыми числами дерева производится с целью вычисления суммы всех элементов дерева.

struct BinTree // Элемент бинарного дерева

{

int value;

struct BinTree \* left;

struct BinTree \* right;

};

int calcSum(struct BinTree \* tree, int sum)

{

sum += tree->value;

if (tree->left) sum += calcSum(tree->left);

if (tree->right) sum += calcSum(tree->right);

return sum;

}

## Прототипы функций

Прототипом функции называется предварительное объявления функции, содержащее только ее заголовок, без определения тела функции.

void sqrIt(int \*i); /\* прототип \*/

int main(void)

{

int x;

x = 10;

sqrIt(x); /\* несоответствие типов \*/

return 0;

}

void sqrIt(int \*i)

{

\*i = \*i \* \*i;

}

## Функции с переменным количеством аргументов

Можно вызвать функцию, которая имеет переменное количество параметров. Самым известным примером является printf(). Чтобы сообщить компилятору, что функции будет передано заранее неизвестное количество аргументов, объявление списка ее параметров необходимо закончить многоточием. Например, следующий прото тип указывает, что у функции func() будет как минимум два целых параметра и после них еще некоторое количество (в том числе и 0) параметров:

int func(int a, int b, ...);

В любой функции, использующей переменное количество параметров, должен быть как минимум один реально существующий параметр. Например, следующее объявление неправильное:

int func(...); /\* ошибка \*/

Для передачи функции переменного числа аргументов совместно используются макросы va\_arg, va\_start и va\_end. Самым распространенным примером функции, которая принимает переменное число аргументов, является функция printf(). Тип va\_list определен в заголовке <stdarg.h>.

Общая процедура создания функции, которая может принимать переменное количество аргументов, такова:

Функция должна иметь по крайней мере один известный параметр (может и больше), указываемый до переменного списка параметров. (Такие параметры называются также обязательными, а параметры, следующие за ними — необязательными.) Крайний правый известный параметр называется *last\_parm*. (Он предшествует первому необязательному параметру.) Его имя используется в качестве второго параметра в обращении к макросу va\_start(). Чтобы получить доступ к любому дополнительному параметру, сначала необходимо инициализировать указатель-аргумент *argptr*, обратившись к макросу va\_start(). (Иными словами, необходимо выполнить вызов va\_start(*argptr*, <имя *last\_parm*>).) После этого значения параметров возвращаются в результате вызова макроса va\_arg(). В качестве второго аргумента этого макроса (соответствующего параметру *type*), нужно указать тип следующего параметра. Наконец, после прочтения всех параметров до возвращения из функции необходимо вызвать макрос va\_end(), чтобы гарантировать корректное восстановление стека.

Пример:

double sumSeries(int num, ...)

{

double sum=0.0;

va\_list argptr;

/\* Инициализация argptr \*/

va\_start(argptr, num);

/\* сумма последовательности \*/

for( ; num; num--)

sum += va\_arg(argptr, double); /\* прибавить следующий аргумент \*/

/\* выполнение корректного выхода \*/

va\_end(argptr);

return sum;

}

# Ввод/вывод

## Ввод/вывод на консоль

В языке С не определено никаких ключевых слов, с помощью которых можно выполнять ввод/вывод. Вместо них используются библиотечные функции. Система ввода/вывода языка С обеспечивает гибкий и в то же время слаженный механизм передачи данных от одного устройства к другому. Впрочем, эта система достаточно большая и состоит из нескольких различных функций. Заголовочным файлом для функций ввода/вывода является <stdio.h>.

В стандарте языка С не определены никакие функции, предназначенные для выполнения различных операций управления экраном (например, позиционирования курсора) или вывода на него графики? потому, что эти операции на разных машинах очень сильно отличаются.

Функции ввода-вывода делятся на *консольные* и *файловые*, хотя, между ними и нет теоритических отличий. Сначала рассмотрим функции ввода-вывода на консоль, те функции, которые выполняют ввод с клавиатуры и вывод на консоль.

В действительности же эти функции работают со стандартным потоком ввода и *стандартным потоком вывода*. Более того, *стандартный ввод*[D:\doc\Артем\Язык C\Spr\_po\_C\08\08.htm - 33](file:///D:\\doc\\Артем\\Язык%20C\\Spr_po_C\\08\\08.htm" \l "33) и стандартный вывод могут быть перенаправлены на другие устройства. Таким образом, "консольные функции" не обязательно должны работать только с консолью. Перенаправление ввода/вывода будет рассказано в конце раздела. Пока предполагается, что ни стандартный ввод, ни стандартный вывод на другие устройства не перенаправляются.

## Чтение и запись символов

Функции чтения и ввода отдельных символов:

int getchar(void);

int putchar(int c);

Первая из этих функций ожидает, пока не будет нажата клавиша, а затем в младшем байте результата getchar() возвращается код символа. Если конец файла достигнут или в процессе чтения происходит ошибка, функция возвращает EOF (Макрос EOF определяется в <stdio.h>).

Вторая функция putchar() отображает символ на экране в текущей позиции курсора.

Как и в случае getchar() из ее аргумента на экран выводится только младший байт. Функция putchar() возвращает записанный символ или, в случае ошибки, EOF.

Пример:

#include <stdio.h>

#include <ctype.h>

int main(void)

{

char ch;

printf("Введите какой-нибудь текст

(для завершения работы введите точку).\n");

do {

ch = getchar();

if(islower(ch)) ch = toupper(ch);

else ch = tolower(ch);

putchar(ch);

} while (ch != '.');

return 0;

}

Использование getchar() может быть связано с определенными трудностями. Во многих библиотеках компиляторов эта функция реализуется таким образом, что она заполняет буфер ввода до тех пор, пока не будет нажата клавиша <ENTER>. Это называется *построчно буферизованным вводом*. Чтобы функция getchar() возвратила какой-либо символ, необходимо нажать клавишу <ENTER>. Кроме того, эта функция при каждом ее вызове вводит только по одному символу.

## Чтение и запись строк

Функции gets() и puts(), которые позволяют считывать и отображать строки символов.

char \*gets(char \**cmp*);

int puts(const char \**cmp*);

Функция gets() читает строку символов, введенную с клавиатуры, и записывает ее в память по адресу, на который указывает ее аргумент. Символы можно вводить с клавиатуры до тех пор, пока не будет введен символ возврата каретки. Он не станет частью строки, а вместо него в ее конец будет помещен символ конца строки ('0'), после чего произойдет возврат из функции gets().

Здесь cmp — это указатель на массив символов, в который записываются символы, вводимые пользователем, gets() также возвращает cmp. Следующая программа читает строку в массив str и выводит ее длину:

#include <stdio.h>

#include <string.h>

int main(void)

{

char str[80];

gets(str);

printf("Длина в символах равна %d", strlen(str));

return 0;

}

Необходимо очень осторожно использовать gets(), потому что эта функция не проверяет границы массива, в который записываются введенные символы. Таким образом, может случиться, что пользователь введет больше символов, чем помещается в этом массиве. Хотя функция gets() прекрасно подходит для программ-примеров и простых утилит, предназначенных только для вас, но в профессиональных программах ею лучше не пользоваться. Ее альтернативой, позволяющей предотвратить переполнение массива, будет функция fgets(), которая описана далее.

Функция puts() отображает на экране свой строковый аргумент, после чего курсор переходит на новую строку.

Функция puts() в случае успешного завершения возвращает неотрицательное значение, а в случае ошибки — EOF. Однако при записи на консоль обычно предполагают, что ошибки не будет, поэтому значение, возвращаемое puts(), проверяется редко.

## Форматированный ввод/вывод на консоль (printf() и scanf())

Функции printf() и scanf() выполняют форматный ввод и вывод, то есть они могут читать и писать данные в разных форматах. Данные на консоль выводит printf(). А ее "дополнение", функция scanf(), наоборот — считывает данные с клавиатуры. Обе функции могут работать с любым встроенным типом данных, а также с символьными строками, которые завершаются символом конца строки ('0').

### Функция printf()

Вот прототип функции printf():

int printf(const char \**управляющая\_строка*, ...);

Функция printf() возвращает число выведенных символов или отрицательное значение в случае ошибки.

*Управляющая\_строка* состоит из элементов двух видов. Первый из них – это символы, которые предстоит вывести на экран; второй – это *спецификаторы преобразования*, которые определяют способ вывода стоящих за ними аргументов. Каждый такой спецификатор начинается со знака процента, за которым следует код формата. Аргументов должно быть ровно столько, сколько и спецификаторов, причем спецификаторы преобразования и аргументы должны попарно соответствовать друг другу в направлении слева направо.

Например:

printf ("Hi %с %d %s", 'с', 10, "there!");

приведет к выводу «Hi с 10 there!».

Спецификаторы формата приведены в таблице 8.1.

Таблица 8.1 – Спецификаторы формата функции printf()

| ***Код*** | ***Формат*** |
| --- | --- |
| %c | Символ |
| %d | Десятичное целое со знаком |
| %i | Десятичное целое со знаком |
| %e | Экспоненциальное представление ('е' на нижнем регистре) |
| %E | Экспоненциальное представление ('Е' на верхнем регистре) |
| %f | Десятичное с плавающей точкой |
| %g | В зависимости от того, какой вывод будет короче, используется %е или %f |
| %G | В зависимости от того, какой вывод будет короче, используется %Е или %F |
| %o | Восьмеричное без знака |
| %s | Строка символов |
| %u | Десятичное целое без знака |
| %x | Шестнадцатеричное без знака (буквы на нижнем регистре) |
| %X | Шестнадцатеричное без знака (буквы на верхнем регистре) |
| %p | Выводит указатель |
| %n | Аргумент, соответствующий этому спецификатору, должен быть указателем на целочисленную переменную. Спецификатор позволяет сохранить в этой переменной количество записанных символов (записанных до того места, в котором находится код %n) |
| %% | Выводит знак % |

Целое число, расположенное между знаком % и кодом формата, играет роль *модификатора минимальной ширины поля*. Если указан модификатор минимальной ширины поля, то чтобы ширина поля вывода была не меньше указанной минимальной длины, при необходимости вывод будет дополнен пробелами. Если же выводятся строки или числа, которые длиннее указанного минимума, то они все равно будут отображаться полностью. По умолчанию для дополнения используются пробелы. А если для этого надо использовать нули, то перед модификатором ширины поля следует поместить 0. Например, %05d означает, что любое число, количество цифр которого меньше пяти, будет дополнено таким количеством нулей, чтобы число состояло из пяти цифр. Например:

double item = 10.12;

printf("%f\n", item);

printf("%7f\n", item);

printf("%07f\n", item);

Вот что выводится при выполнении этой программы:

10.12

10.12

0010.12

Модификатор минимальной ширины поля чаще всего используется при создании таблиц, в которых столбцы должны быть выровнены по вертикали.

*Модификатор точности* следует за модификатором минимальной ширины поля (если таковой имеется). Он состоит из точки и расположенного за ней целого числа. Значение этого модификатора зависит от типа данных, к которым его применяют.

Когда модификатор точности применяется к данным с плавающей точкой, для преобразования которых используются спецификаторы преобразования %f, %e или %E, то он определяет количество выводимых десятичных разрядов. Например, %10.4f означает, что ширина поля вывода будет не менее 10 символов, причем для десятичных разрядов будет отведено четыре позиции.

Примененный к строкам, модификатор точности определяет максимальную длину поля. Например, %5.7s означает, что длина выводимой строки будет составлять минимум пять и максимум семь символов. Если строка окажется длиннее, чем максимальная длина поля, то конечные символы выводиться не будут.

Например:

printf("%.4f\n", 123.1234567);

printf("%3.8d\n", 1000);

printf("%10.15s\n", "Это простая проверка.");

return 0;

}

Вот что выводится при выполнении этой программы:

123.1235

00001000

Это простая про

По умолчанию весь вывод выравнивается по правому краю. То есть если ширина поля больше ширины выводимых данных, то эти данные располагаются по правому краю поля. Вывод по левому краю можно назначить принудительно, поместив знак минус прямо за %. Например, %-l0.2f означает, что число с плавающей точкой и с двумя десятичными разрядами будет выровнено по левому краю 10-символьного поля.

В следующей программе показано, как применяется выравнивание по левому краю:

printf("по правому краю: %8d\n", 100);

printf(" по левому краю: %-8d\n", 100);

И вот что получилось:

по правому краю: 100

по левому краю: 100

Некоторые модификаторы в вызове функции printf() позволяют отображать целые числа типа short и long. Такие модификаторы можно использовать для следующих спецификаторов типа: d, i, o, u и x. Модификатор l (эль) в вызове функции printf() указывает, что за ним следуют данные типа long.

Модификатор L может находиться перед спецификаторами преобразования с плавающей точкой e, f и g, и указывать этим, что преобразуется значение long double.

### Функция scanf()

Функция scanf() – это программа ввода общего назначения, выполняющая ввод с консоли. Она может читать данные всех встроенных типов и автоматически преобразовывать числа в соответствующий внутренний формат, scanf() во многом выглядит как обратная к printf().

Эта функция возвращает количество тех элементов данных, которым было успешно присвоено значение. В случае ошибки scanf() возвращает EOF.

Вот прототип функции scanf():

int scanf(const char \**управляющая\_строка*, ...);

Управляющая строка состоит из символов трех видов:

* спецификаторов преобразования,
* разделителей,
* символов, не являющихся разделителями.

Каждый спецификатор формата ввода начинается со знака %, причем спецификаторы формата ввода сообщают функции scanf() тип считываемых данных. Перечень этих кодов (т.е. литер-спецификаторов) приведен в табл. 8.2.

Таблица 8.2 – Спецификаторы преобразования функции scanf()

|  |  |
| --- | --- |
| ***Код*** | ***Значение*** |
| %c | Читает одиночный символ |
| %d | Читает десятичное целое число со знаком |
| %i | Читает целое число как в десятичном, так и восьмеричном или шестнадцатеричном формате |
| %e | Читает число с плавающей точкой |
| %f | Читает число с плавающей точкой |
| %g | Читает число с плавающей точкой |
| %о | Читает восьмеричное число |
| %s | Читает строку |
| %x | Читает шестнадцатеричное число |
| %p | Читает указатель |
| %n | Принимает целое значение, равное количеству уже считанных символов |
| %u | Читает десятичное целое число без знака |
| %[] | Читает набор сканируемых символов |
| %% | Читает знак процента |

Следующий фрагмент программы читает восьмеричное и шестнадцатеричное число:

scanf("%o%x", &i, &j);

printf("%o %x", i, j);

Для чтения из входного потока строки можно использовать функцию scanf() со спецификатором преобразования %s. Использование спецификатора преобразования %s заставляет scanf() читать символы до тех пор, пока не встретится какой-либо разделитель. Читаемые символы помещаются в символьный массив, на который указывает соответствующий аргумент, а после введенных символов еще добавляется символ конца строки ('0'). Что касается scanf(), то таким разделителем может быть пробел, разделитель строк, табуляция, вертикальная табуляция или подача страницы. В отличие от gets(), которая читает строку, пока не будет нажата клавиша <ENTER>, scanf() читает строку до тех пор, пока не встретится первый разделитель.

Чтобы увидеть, как действует спецификатор %s, попробуйте при выполнении этой программы ввести строку "привет всем":

#include <stdio.h>

int main(void)

{

char str[80];

printf("Введите строку: ");

scanf("%s", str);

printf("Вот Ваша строка: %s", str);

return 0;

}

Функция scanf() поддерживает спецификатор формата общего назначения, называемый набором сканируемых символов (scanset). *Набор сканируемых символов* представляет собой множество символов. Когда scanf() обрабатывает такое множество, то вводит только те символы, которые входят в набор сканируемых символов. Читаемые символы будут помещаться в массив символов, который указан аргументом, соответствующим набору сканируемых символов. Этот набор определяется следующим образом: все те символы, которые предстоит сканировать, помещают в квадратные скобки. Непосредственно перед открывающей квадратной скобкой должен находиться знак %. Например, следующий набор сканируемых символов дает указание scanf() сканировать только символы X, Y и Z:

%[XYZ]

При использовании набора сканируемых символов функция scanf() продолжает читать символы, помещая их в соответствующий массив символов, пока не встретится символ, не входящий в этот набор. При возвращении из scanf() в массиве символов будет находиться строка, состоящая из считанных символов, причем эта строка будет заканчиваться символом конца строки. Чтобы увидеть, как это все работает, запустите следующую программу:

#include <stdio.h>

int main(void)

{

int i;

char str[80], str2[80];

scanf("%d%[abcdefg]%s", &i, str, str2);

printf("%d %s %s", i, str, str2);

return 0;

}

Введите 123abcdtye, а затем нажмите клавишу <ENTER>. После этого программа выведет 123 abсd tye. Так как в данном случае 't' не входит в набор сканируемых символов, то scanf() прекратила чтение символов в переменную str сразу после того, как встретился символ 't'. Оставшиеся символы были помещены в переменную str2.

Кроме того, можно указать набор сканируемых символов, работающий с точностью до наоборот; тогда первым символом в таком наборе должен быть ^. Этот символ дает указание scanf() принимать любой символ, который не входит в набор сканируемых символов.

В большинстве реализаций для указания диапазона можно использовать дефис. Например, указанный ниже набор сканируемых символов дает функции scanf() указание принимать символы от А до Z:

%[A-Z]

Следует обратить внимание на такой важный момент: набор сканируемых символов чувствителен к регистру букв. Если нужно сканировать буквы и на верхнем, и на нижнем регистре, то их надо указывать отдельно для каждого регистра.

Разделитель в управляющей строке дает scanf() указание пропустить в потоке ввода один или несколько начальных разделителей. Разделителями являются пробелы, табуляции, вертикальные табуляции, подачи страниц и разделители строк. В сущности, один разделитель в управляющей строке заставляет scanf() читать, но не сохранять любое количество (в том числе и нулевое) разделителей, которые находятся перед первым символом, не являющимся разделителем.

Если в управляющей строке находится символ, не являющийся разделителем, то функция scanf() прочитает символ из входного потока, проверит, совпадает ли прочитанный символ с указанным в управляющей строке, и в случае совпадения пропустит прочитанный символ. Например, "%d,%d" заставляет scanf() прочитать целое значение, прочитать запятую и пропустить ее (если это была запятая!), а затем прочитать следующее целое значение. Если же указанный символ во входном потоке не будет найден, то scanf() завершится.

Как и printf(), функция scanf() дает возможность модифицировать некоторое число своих спецификаторов формата. В спецификаторах формата моно указать модификатор максимальной длины поля. Чтобы считывать в переменную str не более 20 символов, пишите:

scanf("%20s", str);

Если поток ввода содержит больше 20 символов, то при следующем вызове функций ввода считывание начнется после того места, где оно закончилось при предыдущем вызове.

scanf() может прочитать поле, но не присваивать прочитанное значение никакой переменной; для этого надо перед литерой-спецификатором формата поля поставить звездочку, \*. Например, когда выполняется оператор

scanf("%d%\*c%d", &x, &y);

можно ввести пару координат 10,10. Запятая будет прочитана правильно, но ничему не будет присвоена. Подавление присвоения особенно полезно тогда, когда нужно обработать только часть того, что вводится.

-- продолжение на следующем занятии --

## Файловый ввод/вывод

Далее рассмотрим работу с файловой системой в языке C. Как мы и говорили, в языке С система ввода/вывода реализуется с помощью библиотечных функций, а не ключевых слов. Благодаря этому система ввода/вывода является очень мощной и гибкой. Например, во время работы с файлами данные могут передаваться или в своем внутреннем двоичном представлении или в текстовом формате, то есть в более удобочитаемом виде. Это облегчает задачу создания файлов в нужном формате.

Так как С является фундаментом C++, то иногда возникает путаница в отношениях его файловой системы с аналогичной системой C++. Во-первых, C++ поддерживает всю файловую систему С. Таким образом, при перемещении более старого С-кода в C++ нет необходимости менять все процедуры ввода/вывода. Во-вторых, следует иметь в виду, что в C++ определена своя собственная, объектно-ориентированная система ввода/вывода, в которую входят как функции, так и операторы ввода/вывода. В системе ввода/вывода C++ полностью поддерживаются все возможности аналогичной системы С и это делает излишней файловую систему языка С. Вообще говоря, при написании программ на языке C++ обычно более удобно использовать именно его систему ввода/вывода, но, если необходимо воспользоваться файловой системой языка С, то это также вполне возможно.

## Потоки и файлы

Перед тем как начать изучение файловой системы языка С, необходимо уяснить, в чем разница между *потоками* и *файлами*. В системе ввода/вывода С для программ поддерживается единый интерфейс, не зависящий от того, к какому конкретному устройству осуществляется доступ. То есть в этой системе между программой и устройством находится нечто более общее, чем само устройство. Такое обобщенное устройство ввода или вывода (устройство более высокого уровня абстракции) называется *потоком*.

Файловая система языка С предназначена для работы с самыми разными устройствами, в том числе терминалами, дисководами и накопителями на магнитной ленте. Даже если какое-то устройство сильно отличается от других, буферизованная файловая система все равно представит его в виде логического устройства, которое называется потоком. Все потоки ведут себя похожим образом. И так как они в основном не зависят от физических устройств, то та же функция, которая выполняет запись в дисковый файл, может ту же операцию выполнять и на другом устройстве, например, на консоли. Потоки бывают двух видов: *текстовые* и *двоичные*.

*Текстовый поток* — это последовательность символов. В стандарте С считается, что текстовый поток организован в виде строк, каждая из которых заканчивается символом новой строки. Однако в конце последней строки этот символ не является обязательным. В текстовом потоке по требованию базовой среды могут происходить определенные преобразования символов. Например, символ новой строки может быть заменен парой символов — возврата каретки и перевода строки. Поэтому может и не быть однозначного соответствия между символами, которые пишутся (читаются), и теми, которые хранятся во внешнем устройстве.

*Двоичный поток* — это последовательность байтов, которая взаимно однозначно соответствует байтам на внешнем устройстве, причем никакого преобразования символов не происходит.

Кроме того, количество тех байтов, которые пишутся (читаются), и тех, которые хранятся на внешнем устройстве, одинаково.

В языке С файлом может быть все что угодно, начиная с дискового файла и заканчивая терминалом или принтером. Поток связывают с определенным файлом, выполняя операцию *открытия*. Как только файл открыт, можно проводить обмен информацией между ним и программой.

Но не у всех файлов одинаковые возможности. Например, к дисковому файлу прямой доступ возможен, в то время как к некоторым принтерам – нет. Таким образом, мы пришли к одному важному принципу, относящемуся к системе ввода/вывода языка С: все потоки одинаковы, а файлы – нет.

Файл отсоединяется от определенного потока (т.е. разрывается связь между файлом и потоком) с помощью операции *закрытия*. Если программа завершает работу нормально, т.е. либо main() возвращает управление операционной системе, либо вызывается exit(), то все файлы закрываются автоматически. В случае аварийного завершения работы программы, например, в случае краха, файлы не закрываются.

Если вы новичок в программировании, то разграничение потоков и файлов может показаться излишним или даже "заумным". Однако надо помнить, что основная цель такого разграничения — это обеспечить единый интерфейс. Для выполнения всех операций ввода/вывода следует использовать только понятия потоков и применять всего лишь одну файловую систему. Ввод или вывод от каждого устройства автоматически преобразуется системой ввода/вывода в легко управляемый поток.

## Основные средства работы с файлами

Файловая система языка С состоит из нескольких взаимосвязанных функций. Самые распространенные из них показаны в табл. 8.3. Для их работы требуется заголовок <stdio.h>.

Таблица 8.3 – Основные функции для работы с файлами

| ***Имя*** | ***Назначение*** |
| --- | --- |
| fopen() | Открывает файл |
| fclose() | Закрывает файл |
| putc() | Записывает символ в файл |
| fputc() | To же, что и putc() |
| getc() | Читает символ из файла |
| fgetc() | To же, что и getc() |
| fgets() | Читает строку из файла |
| fputs() | Записывает строку в файл |
| fseek() | Устанавливает указатель текущей позиции на определенный байт файла |
| ftell() | Возвращает текущее значение указателя текущей позиции в файле |
| fprintf() | Для файла то же, что printf() для консоли |
| fscanf() | Для файла то же, что scanf() для консоли |
| feof() | Возвращает значение true (истина), если достигнут конец файла |
| ferror() | Возвращает значение true, если произошла ошибка |
| rewind() | Устанавливает указатель текущей позиции в начало файла |
| remove() | Стирает файл |
| fflush() | Дозапись потока в файл |

Заголовок <stdio.h> предоставляет прототипы функций ввода/вывода и определяет следующие три типа: size\_t, fpos\_t и FILE. size\_t и fpos\_t представляют собой определенные разновидности такого типа, как целое без знака. А о третьем типе, FILE, рассказывается в следующем подразделе.

Кроме того, в <stdio.h> определяется несколько макросов. Из них к материалу этой главы относятся NULL, EOF, FOPEN\_MAX, SEEK\_SET, SEEK\_CUR и SEEK\_END. Макрос NULL определяет пустой (null) указатель. FOPEN\_MAX определяет целое значение, равное максимальному числу одновременно открытых файлов. Другие макросы используются вместе с fseek() — функцией, выполняющей операции прямого доступа к файлу.

## Указатель файла

У каждого потока, связанного с файлом, имеется управляющая структура, содержащая информацию о файле; она имеет тип FILE.

*Указатель файла* – это указатель на структуру типа FILE. Он указывает на структуру, содержащую различные сведения о файле, например, его имя, статус и указатель текущей позиции в начало файла. В сущности, указатель файла определяет конкретный файл и используется соответствующим потоком при выполнении функций ввода/вывода. Чтобы выполнять в файлах операции чтения и записи, программы должны использовать указатели соответствующих файлов. Чтобы объявить переменную-указатель файла, используйте такого рода инструкцию:

FILE \* fp;

## Открытие, закрытие и стирание файлов

Функция fopen() открывает поток и связывает с этим потоком определенный файл. Затем она возвращает указатель этого файла. Чаще всего (а также в оставшейся части этой главы) под файлом подразумевается дисковый файл. Прототип функции fopen() такой:

FILE \*fopen(const char \**имя\_файла*, const char \**режим*);

где *имя\_файла* – это массив символов, представляющую собой допустимое имя файла, в которое также может входить спецификация пути к этому файлу. Строка, на которую указывает *режим*, определяет, каким образом файл будет открыт. В табл. 8.4 показано, какие значения строки *режим* являются допустимыми. Строки, подобные "r+b" могут быть представлены и в виде "rb+".

Таблица 8.4 – Режимы открытия файлов

|  |  |
| --- | --- |
| ***Режим*** | ***Что означает*** |
| r | Открыть текстовый файл для чтения |
| w | Создать текстовый файл для записи |
| a | Добавить в конец текстового файла |
| rb | Открыть двоичный файл для чтения |
| wb | Создать двоичный файл для записи |
| ab | Добавить в конец двоичного файла |
| r+ | Открыть текстовый файл для чтения/записи |
| w+ | Создать текстовый файл для чтения/записи |
| a+ | Добавить в конец текстового файла или создать текстовый файл для чтения/записи |
| r+b | Открыть двоичный файл для чтения/записи |
| w+b | Создать двоичный файл для чтения/записи |
| a+b | Добавить в конец двоичного файла или создать двоичный файл для чтения/записи |

Как уже упоминалось, функция fopen() возвращает указатель файла. Никогда не следует изменять значение этого указателя в программе. Если при открытии файла происходит ошибка, то fopen() возвращает пустой (null) указатель.

В следующем коде функция fopen() используется для открытия файла по имени TEST для записи.

FILE \*fp;

if ((fp = fopen("test","w"))==NULL) {

printf("Ошибка при открытии файла.\n");

exit(1);

}

Этот метод помогает при открытии файла обнаружить любую ошибку, например, защиту от записи или полный диск, причем обнаружить еще до того, как программа попытается в этот файл что-либо записать. Вообще говоря, всегда нужно вначале получить подтверждение, что функция fopen() выполнилась успешно, и лишь затем выполнять с файлом другие операции.

Функция fclose() закрывает поток, который был открыт с помощью вызова fopen().Функция fclose() записывает в файл все данные, которые еще оставались в дисковом буфере.

Прототип функции fclose() такой:

int fclose(FILE \**уф*);

где *уф* – указатель файла, возвращенный в результате вызова fopen(). Возвращение нуля означает успешную операцию закрытия. В случае же ошибки возвращается EOF. Обычно отказ при выполнении fclose() происходит только тогда, когда диск был преждевременно удален с дисковода или на диске не осталось свободного места.

Функция remove() стирает указанный файл. Вот ее прототип:

int remove(const char \**имя\_файла*);

В случае успешного выполнения эта функция возвращает нуль, а в противном случае — ненулевое значение.

## Чтение и запись символов и строк

В системе ввода/вывода языка С определяются две эквивалентные функции, предназначенные для вывода символов: putc() и fputc(). Будем дальше рассматривать функцию putc().

Функция putc() записывает символы в файл, который с помощью fopen() уже открыт в режиме записи. Прототип этой функции следующий:

int putc(int *ch*, FILE \**уф*);

Если функция putc() выполнилась успешно, то возвращается записанный символ. В противном же случае возвращается EOF.

Для ввода символа также имеются две эквивалентные функции: getc() и fgetc(). Обе определяются для сохранения совместимости со старыми версиями С. Будем использовать getc().

Функция getc() записывает символы в файл, который с помощью fopen() уже открыт в режиме для чтения. Прототип этой функции следующий:

int getc(FILE \**уф*);

Если достигнут конец файла, то функция getc() возвращает EOF. Поэтому, чтобы прочитать символы до конца текстового файла, можно использовать следующий код;

do {

ch = getc(fp);

} while(ch!=EOF);

Однако проверка значения, возвращенного getc(), возможно, не является наилучшим способом узнать, достигнут ли конец файла. Во-первых, файловая система языка С может работать как с текстовыми, так и с двоичными файлами. Когда файл открывается для двоичного ввода, то может быть прочитано целое значение, которое, как выяснится при проверке, равняется EOF. В таком случае программа ввода сообщит о том, что достигнут конец файла, чего на самом деле может и не быть. Во-вторых, функция getc() возвращает EOF и в случае отказа, а не только тогда, когда достигнут конец файла. Если использовать только возвращаемое значение getc(), то невозможно определить, что же на самом деле произошло. Для решения этой проблемы в С имеется функция feof(), которая определяет, достигнут ли конец файла. Прототип функции feof() такой:

int feof(FILE \**уф*);

Если достигнут конец файла, то feof() возвращает ненулевое значение в противном же случае эта функция возвращает нуль.

Предыдущий пример с использованием функции feof():

while(!feof(fp)) ch = getc(fp);

Кроме getc() и putc(), в языке С также поддерживаются родственные им функции fgets() и fputs(). Первая из них читает строки символов из файла, а вторая записывает строки такого же типа в файл. Прототипы функций fgets() и fputs() следующие:

int fputs(const char \**cmp*, FILE \**уф*);

char \*fgets(char \**cmp*, int *длина*, FILE \**уф*);

Функция fputs() пишет в определенный поток строку, на которую указывает *cmp*. В случае ошибки эта функция возвращает EOF.

Функция fgets() читает из определенного потока строку, и делает это до тех пор, пока не будет прочитан символ новой строки или количество прочитанных символов не станет равным *длина*-1. Если был прочитан разделитель строк, он записывается в строку (этим функция fgets() отличается от функции gets()). Полученная в результате строка будет оканчиваться символом конца строки ('0'). При успешном завершении работы функция возвращает *cmp*, а в случае ошибки – пустой указатель (null).

Функция rewind() устанавливает указатель текущей позиции в файле на начало файла, указанного в качестве аргумента этой функции. Иными словами, функция rewind() выполняет "перемотку" (rewind) файла. Вот ее прототип:

void rewind(FILE \**уф*);

где *уф* — это допустимый указатель файла.

Функция ferror() определяет, произошла ли ошибка во время выполнения последней операции с файлом. Прототип этой функции следующий:

int ferror(FILE \**уф*);

где *уф* — допустимый указатель файла. Она возвращает значение *true* (истина), если при последней операции с файлом произошла ошибка; в противном же случае она возвращает *false* (ложь).

Для дозаписи содержимого выводного потока в файл применяется функция fflush(). Вот ее прототип:

int fflush(FILE \**уф*);

Эта функция записывает все данные, находящиеся в буфере в файл, который указан с помощью *уф*. При вызове функции fflush() с пустым (null) указателем файла *уф* будет выполнена дозапись во все файлы, открытые для вывода.

## Чтение и запись двоичных данных (fread() и fwrite())

Выше был рассмотрены функции ввода-вывода, предназначенные для работы в текстовом режиме. Для работы с файлами в бинарном режиме в С имеется две функции: fread() и fwrite(). Их прототипы следующие:

size\_t fread(void \**буфер*, size\_t *колич\_байт*, size\_t *счетчик*, FILE \**уф*);

size\_t fwrite(const void \**буфер*, size\_t *колич\_байт*, size\_t *счетчик*, FILE \**уф*);

Для fread() *буфер* – это указатель на область памяти, в которую будут прочитаны данные из файла. Функция fread() возвращает количество прочитанных элементов. Если достигнут конец файла или произошла ошибка, то возвращаемое значение может быть меньше, чем счетчик.

Для fwrite() *буфер* – это указатель на данные, которые будут записаны в файл. Значение *счетчик* определяет, сколько считывается или записывается элементов данных, причем длина каждого элемента в байтах равна *колич\_байт*. Функция fwrite() возвращает количество записанных элементов. Если ошибка не произошла, то возвращаемый результат будет равен значению счетчик.

Как только файл открыт для работы с двоичными данными, fread() и fwrite() соответственно могут читать и записывать информацию любого типа. Например:

/\* Запись несимвольных данных в дисковый файл

и последующее их чтение. \*/

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

FILE \*fp;

double d = 12.23;

int i = 101;

long l = 123023L;

if((fp=fopen("test", "wb+"))==NULL) {

printf("Ошибка при открытии файла.\n");

exit(1);

}

fwrite(&d, sizeof(double), 1, fp);

fwrite(&i, sizeof(int), 1, fp);

fwrite(&l, sizeof(long), 1, fp);

rewind(fp);

fread(&d, sizeof(double), 1, fp);

fread(&i, sizeof(int), 1, fp);

fread(&l, sizeof(long), 1, fp);

printf("%f %d %ld", d, i, l);

fclose(fp);

return 0;

}

Обратите внимание, как в этой программе при определении длины каждого типа данных используется функция sizeof().

Одним из самых полезных применений функций fread() и fwrite() является чтение и запись данных пользовательских типов, особенно структур. Например, если определена структура

struct struct\_type {

float balance;

char name[80];

} cust;

то следующая инструкция записывает содержимое cust в файл, на который указывает fp:

fwrite(&cust, sizeof(struct struct\_type), 1, fp);

## Ввод/вывод при прямом доступе (fseek())

При прямом доступе можно выполнять операции ввода/вывода, используя систему ввода/вывода языка С и функцию fseek(), которая устанавливает указатель текущей позиции в файле. Вот прототип этой функции:

int fseek(FILE \**уф*, long int *колич\_байт*, int *начало\_отсчета*);

Здесь *колич\_байт* – количество байтов, считая от *начало\_отсчета*, оно определяет новое значение указателя текущей позиции, а *начало отсчёта* – это один из следующих макросов:

Начало отсчета Макрос

Начало файла SEEK\_SET

Текущая позиция SEEK\_CUR

Конец файла SEEK\_END

Поэтому, чтобы получить в файле доступ на расстоянии *колич\_байт* байтов от начала файла, *начало\_отсчета* должно равняться SEEK\_SET. Чтобы при доступе расстояние отсчитывалось от текущей позиции, используйте макрос SEEK\_CUR, а чтобы при доступе расстояние отсчитывалось от конца файла, нужно указывать макрос SEEK\_END. При успешном завершении своей работы функция fseek() возвращает нуль, а в случае ошибки — ненулевое значение.

Функцию fseek() можно использовать для доступа внутри многих значений одного типа, просто умножая размер данных на номер элемента, который вам нужен. Например, предположим, имеется список рассылки, который состоит из структур типа addr (определенных ранее). Чтобы получить доступ к десятому адресу в файле, в котором хранятся адреса, используйте следующий оператор:

fseek(fp, 9\*sizeof(struct addr), SEEK\_SET);

Текущее значение указателя текущей позиции в файле можно определить с помощью функции ftell(). Вот ее прототип:

long int ftell(FILE \**уф*);

Функция возвращает текущее значение указателя текущей позиции в файле. При неудачном исходе она возвращает -1.

Обычно прямой доступ может потребоваться лишь для двоичных файлов. Причина тут простая — так как в текстовых файлах могут выполняться преобразования символов, то может и не быть прямого соответствия между тем, что находится в файле и тем байтом, к которому нужен доступ. Единственный случай, когда надо использовать fseek() для текстового файла — это доступ к той позиции, которая была уже найдена с помощью ftell(); такой доступ выполняется с помощью макроса SEEK\_SET, используемого в качестве начала отсчета.

## Форматированный ввод-вывод при работе с файлами (fprintf() и fscanf())

Кроме основных функций ввода/вывода, о которых шла речь, в системе ввода/вывода языка С также имеются функции fprintf() и fscanf(). Эти две функции, за исключением того, что предназначены для работы с файлами, ведут себя точно так же, как и printf() и scanf(). Прототипы функций fprintf() и fscanf() следующие:

int fprintf(FILE \**уф*, const char \**управляющая\_строка*, ...);

int fscanf(FILE \**уф*, const char \**управляющая\_строка*, ...);

где *уф* – указатель файла, возвращаемый в результате вызова fopen(). Операции ввода/вывода функции fprintf() и fscanf() выполняют с тем файлом, на который указывает *уф*.

Пример:

/\* пример использования fscanf() и fprintf() \*/

#include <stdio.h>

#include <io.h>

#include <stdlib.h>

int main(void)

{

FILE \*fp;

char s[80];

int t;

if((fp=fopen("test", "w")) == NULL) {

printf("Ошибка открытия файла.\n");

exit(1);

}

printf("Введите строку и число: ");

fscanf(stdin, "%s %d", s, &t); /\* читать с клавиатуры \*/

fprintf(fp, "%s %d", s, t); /\* писать в файл \*/

fclose(fp);

if((fp=fopen("test","r")) == NULL) {

printf("Ошибка при открытии файла.\n");

exit(1);

}

fscanf(fp, "%s%d", s, &t); /\* чтение из файла \*/

fprintf(stdout, "%s %d", s, t); /\* вывод на экран \*/

return 0;

}

*Совет*. Хотя читать разносортные данные из файлов на дисках и писать их в файлы, расположенные также на дисках, часто легче всего именно с помошью функций fprintf() и fscanf(), но это не всегда самый эффективный способ выполнения операций чтения и записи. Так как данные в формате ASCII записываются так, как они должны появиться на экране (а не в двоичном виде), то каждый вызов этих функций сопряжен с определенными накладными расходами. Поэтому, если надо заботиться о размере файла или скорости, то, скорее всего, придется использовать fread() и fwrite().

## Стандартные потоки

В начале выполнения программы автоматически открываются три потока. Это stdin (стандартный поток ввода), stdout (стандартный поток вывода) и stderr (стандартный поток ошибок). Обычно эти потоки направляются к консоли, но в средах, которые поддерживают перенаправление ввода/вывода, они могут быть перенаправлены операционной системой на другое устройство. (Перенаправление ввода/вывода поддерживается, например, такими операционными системами, как UNIX, DOS, Windows и т.д.

Так как стандартные потоки являются указателями файлов, то они могут использоваться системой ввода/вывода языка С также для выполнения операций ввода/вывода на консоль. Например, putchar() может быть определена таким образом:

int putchar(char c)

{

return putc(c, stdout);

}

Вообще говоря, stdin обычно используется для считывания с консоли, a stdout и stderr – для записи на консоль.

В роли указателей файлов потоки stdin, stdout и stderr можно применять в любой функции, где используется переменная типа FILE \*. Например, для ввода строки с консоли можно написать примерно такой вызов fgets():

char str[255];

fgets(str, 80, stdin);

И действительно, такое применение fgets() может оказаться достаточно полезным. Как уже говорилось ранее, при использовании gets() не исключена возможность, что массив, который используется для приема вводимых пользователем символов, будет переполнен. Это возможно потому, что gets() не проводит проверку на отсутствие нарушения границ. Использование fgets (с ее возможностью ограничения длины считываемой информации) решает данную проблему. Однако, символ перевода строки придется удалять вручную:

#include <stdio.h>

#include <string.h>

int main(void)

{

const int MAX\_STR\_LEN = 10;

char str[MAX\_STR\_LEN];

int i;

printf("Введите строку: ");

fgets(str, MAX\_STR\_LEN, stdin);

/\* удалить символ новой строки, если он есть \*/

i = strlen(str)-1;

if(str[i]=='\n') str[i] = '\0';

printf("Это Ваша строка: %s", str);

return 0;

}

## Перенаправление стандартных потоков

Как говорилось в предыдущем разделе, ввод/вывод на консоль можно выполнять с помощью любой файловой функции языка С. Однако для вас может быть сюрпризом, что, оказывается, операции ввода/вывода на дисковых файлах можно выполнять с помощью функции консольного ввода/вывода, например, printf()! Дело в том, что все функции консольного ввода/вывода, о которых говорилось выше, выполняют свои операции с потоками stdin и stdout. В средах, поддерживающих перенаправление ввода/вывода, это равносильно тому, что stdin или stdout могут быть перенаправлены на устройство, отличное от клавиатуры или экрана. Например:

#include <stdio.h>

int main(void)

{

char str[80];

printf("Введите строку: ");

gets(str);

printf(str);

return 0;

}

Предположим, что эта программа называется TEST. При ее нормальном выполнении на экран выводится подсказка, затем читается строка, введенная с клавиатуры, и, наконец, эта строка выводится на экран. Однако в средах, в которых поддерживается перенаправление ввода/вывода, один из потоков stdin или stdout (или оба одновременно) можно перенаправить в файл. Например, в среде DOS или Windows следующий запуск TEST

TEST > OUTPUT

приводит к тому, что вывод этой программы будет записан в файл по имени OUTPUT. А следующий запуск TEST

TEST < INPUT > OUTPUT

направляет поток stdin в файл по имени INPUT, а поток стандартного вывода – в файл по имени OUTPUT.

# Обзор стандартной библиотеки языка C

В настоящем разделе приводится обзор заголовков стандартной библиотеки языка C. Целью раздела является ознакомительный обзор основных групп функций стандартной библиотеки C. Подробную информацию по стандартной библиотеке следует искать в справочниках.

Содержимое и форма стандартной библиотеки С задается Стандартом ANSI/ISO. Т.е. Стандарт С определяет тот набор функций, который должен поддерживать любой стандартный компилятор. Однако при этом большинство компиляторов предоставляют дополнительные функции, которые не специфицированы в Стандарте (например, функции работы с графикой, подпрограммы, управляемые с помощью мышки, и т.д.).

Список заголовков, определенных в стандарте C89 приведен в таблице 10.1.

Таблица 10.1 – Заголовки, определенные в стандарте C89

|  |  |
| --- | --- |
| ***Заголовок*** | ***Назначение*** |
| <assert.h> | Определяет макрос assert() |
| <ctype.h> | Обработка символов |
| <errno.h> | Выдача сообщения об ошибках |
| <float.h> | Задает пределы значений с плавающей точкой, зависящие от реализации |
| <limits.h> | Задает различные ограничения, зависящие от реализации |
| <locate.h> | Поддерживает локализацию |
| <math.h> | Различные определения, используемые математической библиотекой |
| <setjmp.h> | Поддерживает нелокальные переходы |
| <signal.h> | Поддерживает обработку сигналов |
| <stdarg.h> | Поддерживает списки входных параметров функции с переменным числом аргументов |
| <stddef.h> | Определяет некоторые наиболее часто используемые константы |
| <stdio.h> | Поддерживает систему ввода/вывода |
| <stdlib.h> | Смешанные объявления |
| <string.h> | Поддерживает функции обработки строк |
| <time.h> | Поддерживает функции, обращающиеся к системному времени |

## Функции ввода/вывода

С функциями ввода/вывода ассоциирован заголовок <stdio.h>. Этот заголовок определяет некоторые макросы и типы, которые используются файловой системой. Наиболее важным из них является тип FILE, который используется для объявления указателя на файл. Два других часто используемых типа – size\_t и fpos\_t. Тип size\_t, представляющий собой некоторую разновидность целых без знака, – это тип результата, возвращаемого функцией sizeof. Тип fpos\_t определяет объект, который однозначно задает каждую позицию в файле. Самым популярным макросом, определенным в этом заголовке, является макрос EOF, значение которого указывает на конец файла. Другие типы данных и макросы, определенные в заголовке <stdio.h>, описаны вместе с функциями, с которыми они связаны.

Обзор системы ввода/вывода приведено в разделе 8.

## Строковые и символьные функции

Стандартная библиотека языка С обладает богатым и разнообразным набором функций для обработки строк и символов. Строковые функции работают с массивами символов (строками), заканчивающимися символом конца строки. В языке С для работы со строковыми функциями используется заголовок <string.h>, для символьных функций – заголовочный файл <ctype.h>.

Поскольку в С не предусмотрен автоматический контроль нарушения границ массивов, вся ответственность за их переполнение ложится на программиста. Не следует этим пренебрегать, так как при переполнении массива может произойти аварийное завершение программы.

В этой главе описаны только те функции, которые работают с символами типа char. Эти функции были определены стандартом С с самого начала, и, безусловно, они являются наиболее популярными и поддерживаются большинством компиляторов. Двухбайтовые функции, работающие с символами типа wchar\_t рассматриваются в (Шилдт, 2008).

В таблице 10.2 приведены основные группы функций для работы со строками и символами.

Таблица 10.2 – Строковые и символьные функции

|  | **Группы функций** | **Назначение** |
| --- | --- | --- |
|  | isalnum, isalpha, isdigit, isgraph, islower, isprint, isspace, isupper, isxdigit | Определение принадлежности символов |
|  | memchr, memcmp, memcpy,  memmove, memset | Работа с буферами памяти |
|  | strcat, strchr, strcmp, strcoll, strcpy, strcspn, strerror, strlen, strncat, strncmp, strncpy, strpbrk, strrchr, strstr, strtok, strxfrm | Строковые операции |
|  | tolower, toupper | Перевод регистра символов |

В таблице 10.3 приведено назначение некоторых функций, предназначенных для определения принадлежности символов к определенным группам.

Таблица 10.3 – Функции определения принадлежности символов к определенным группам

|  | **Функция** | **Определяет** | **Заголовок** |
| --- | --- | --- | --- |
|  | int isalnum(int ch) | букву или цифру | <ctype.h> |
|  | int isalpha(int *ch*) | букву | <ctype.h> |
|  | int isdigit(int *ch*) | цифру | <ctype.h> |
|  | int isprint(int *ch*) | печатный символ | <ctype.h> |
|  | int isgraph(int *ch*) | печатный символ (но не пробел) | <ctype.h> |
|  | int isspace(int *ch*) | пробел | <ctype.h> |
|  | int islower(int *ch*) | строчную букву | <ctype.h> |
|  | int isupper(int *ch*) | прописную букву | <ctype.h> |
|  | int isxdigit(int *ch*) | шестнадцатиричную цифру (A-F, a-f или 0-9) | <ctype.h> |

В таблице 10.4 приведено назначение функций, предназначенных работы с буферами памяти и байтами.

Таблица 10.4 – Функции для работы с буферами

|  | **Функция** | **Назначение** | **Заголовок** |
| --- | --- | --- | --- |
|  | void \*memchr(const void \**buffer*, int *ch*, size\_t *count*) | Поиск первого вхождения байта в первых count символах | <string.h> |
|  | int memcmp(const void \*buf1, const void \*buf2, size\_t count) | Сравнивает первые count символов массивов (<0 – buf1 меньше buf2, ==0 – buf1 равен buf2, >0 – buf1 больше buf2) | <string.h> |
|  | void \*memcpy(void \*to, const void \*from, size\_t count) | Копирует count символов из массива, адресуемого параметром from, в массив, адресуемый параметром to. Если заданные массивы перекрываются, поведение функции не определено | <string.h> |
|  | void \*memmove(void \*to, const void \*from, size\_t count) | Копирует count символов из массива, адресуемого параметром from, в массив, адресуемый параметром to. Если заданные массивы перекрываются, процесс копирования проходит корректно, но содержимое массива from при этом изменится | <string.h> |
|  | void \*memset(void \*buf, int ch, size\_t count) | копирует младший байт параметра ch в первые count символов массива, адресуемого параметром buf | <string.h> |

В таблице 10.5 приведено назначение функций, предназначенных для определения принадлежности символов к определенным группам.

Таблица 10.5 – Строковые функции

|  | **Функция** | **Назначение** | **Заголовок** |
| --- | --- | --- | --- |
|  | char \*strchr(const char \*str, int ch) | Поиск первого вхождения символа | <string.h> |
|  | char \*strcat(char \*str1, const char \*str2) | Присоединяет к строке str1 копию строки str2 | <string.h> |
|  | int strcmp(const char \*str1, const char \*str2) | Сравнивает две строки в лексикографическом порядке | <string.h> |
|  | char \*strcpy(char \*str1, const char \*str2) | Копирует содержимое строки str2 в строку str1.  Если символьные массивы перекрываются, поведение не определено | <string.h> |
|  | char \*strpbrk(const char \*str1, const char \*str2) | Возвращает указатель на первый символ в строке, адресуемой параметром str1, который совпадает с любым символом в строке, адресуемой параметром str2 | <string.h> |
|  | char \*strrchr(const char \*str, int ch) | Возвращает указатель на последнее вхождение младшего байта параметра ch в строку | <string.h> |
|  | char \*strtok(char \**str1*, const char \**str2*); | Разбивает строку на части по указанному разделителю | <string.h> |
|  | int tolower(int ch) | Переводит символ в нижний регистр | <ctype.h> |
|  | int toupper(int *ch*) | Переводит символ в верхний регистр | <ctype.h> |

## Математические функции

## Функции времени, даты и локализации

## Функции динамического распределения памяти

## Служебные функции

# Структуры данных

В настоящем разделе рассматривается:

* понятие структуры данных;
* статические структуры данных;
* простейшие динамические структуры данных (стеки, очереди, списки, бинарные деревья).

## Статические структуры данных

Статические структуры данных относятся к разряду непримитивных структур, которые, фактически, представляют собой структурированное множество примитивных базовых структур. Статические структуры по определению отличаются неизменностью в процессе выполнения (имеется ввиду неизменность самой структуры, а не значений ее объектов). Память для них выделяется один раз и ее объем остается неизменным до уничтожения экземпляра структуры.

К статическим структурам данных относят структуры, объединения, массивы и т.д. Рассмотрим далее массивы.

Операции с массивами:

* получить элемент с индексом n;
* записать элемент с индексом n.

Особенности массивов:

* доступ к элементам за константное время;
* память тратится только на данные.

## Динамические структуры данных (динамические массивы, стеки, списки, деревья, таблицы)

Динамические структуры характеризуются изменчивостью своего состава и числа элементов в процессе выполнения программы. Для хранения данных в динамических структурах используется динамическая память, поэтому размер динамических структур ограничен доступным уровнем динамически распределяемой памяти.

Основные виды динамических структур данных:

* дин. массивы,
* стеки (stack),
* очереди (queue),
* деки (deque – double ended queue),
* списки (list),
* деревья,
* хеш-таблицы,
* разряженные массивы и т.д.

Далее рассмотрим примеры реализации простейших динамических структур данных, а именно: очереди, стеки, связные списки и бинарные деревья.

## Очереди

**Очередь** (queue) – это линейный список информации, работа с которой происходит по принципу "первым пришел – первым вышел" (FIFO, first-in, first-out). Это значит, что первый помещенный в очередь элемент будет получен из нее первым, второй помещенный элемент будет извлечен вторым и т.д. Это единственный способ работы с очередью; произвольный доступ к отдельным элементам не разрешается. Сравнение работы стека и очереди изображены на рисунке 9.1.

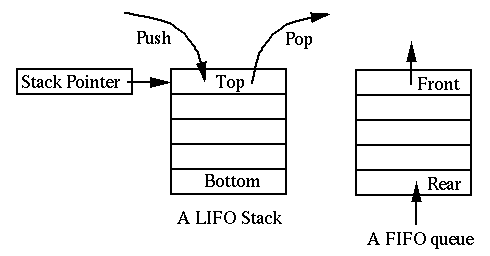


Рисунок 9. – Стек и очередь

Очереди очень часто встречаются в реальной жизни, например, около банков или ресторанов быстрого обслуживания. В программировании очереди применяются при решении многих задач. Один из наиболее популярных видов таких задач – симуляция. Очереди также применяются в планировщиках задач операционных систем и при буферизации ввода/вывода.

Основные операции с очередями:

* сохранить (поместить) значение ("store" – "сохранять");
* получить (извлечь) значение ("retrieve" – "получать").

Операция извлечения удаляет элемент из очереди. Поэтому после извлечения всех элементов очередь будет пуста. Обе операции обычно выполняются за константное время.

Введем две функции: store() и retrieve(). Функция store() помещает элемент в конец очереди, а функция retrieve() удаляет элемент из начала очереди и возвращает его значение.

Простейшая реализация очереди:

#define MAX 100

char \*p[MAX];

int spos = 0;

int rpos = 0;

/\* Сохранение элемента \*/

void store(char \*q)

{

if(spos==MAX) {

printf("Список переполнен\n");

return;

}

p[spos] = q;

spos++;

}

/\* Извлечение получение \*/

char \* retrieve()

{

if(rpos==spos) {

printf("Очередь пуста.\n");

return '\0';

}

rpos++;

return p[rpos-1];

}

Использование:

/\*Вставка\*/

char s[256], \*elem;

printf("Введите элемент %d: ", spos+1);

gets(s);

if(\*s!=0)

{

elem = (char \*) malloc(strlen(s)+1);

if(!elem)

{

printf("Не хватает памяти.\n");

exit(1);

}

strcpy(elem, s);

store(elem);

}

...

/\*Извлечение\*/

char \* p = retrieve());

if(!p) return; /\* Очередь пуста\*/

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

free(p);

**Циклическая очередь**

При достижении конца массива, в котором хранится очередь, можно не останавливать программу, а устанавливать индексы вставки (spos) и извлечения (rpos) так, чтобы они указывали на начало массива. Это позволит помещать в очередь любое количество элементов при условии их своевременного извлечения. Такая реализация очереди называется *циклической очередью*, поскольку массив используется так, будто он представляет собой не линейный список, а кольцо.

Чтобы организовать в программе циклическую очередь, функции store() и retrieve() необходимо переписать следующим образом:

#define MAX 100

char \*p[MAX];

int spos = 0;

int rpos = 0;

void store(char \* q)

{

/\* Очередь переполняется, когда spos на единицу

меньше rpos, либо когда spos указывает

на конец массива, а rpos - на начало.

\*/

if(spos+1==rpos || (spos+1==MAX && !rpos))

{

printf("Список полон\n");

return;

}

p[spos] = q;

spos++;

if(spos==MAX) spos = 0; /\* установить на начало \*/

}

char \* retrieve(void)

{

if(rpos==MAX) rpos = 0; /\* установить на начало \*/

if(rpos==spos)

{

printf("Встреч больше нет.\n");

return NULL;

}

rpos++;

return p[rpos-1];

}

В данной версии очередь переполняется, когда индекс записи находится непосредственно перед индексом извлечения; в противном случае еще есть место для вставки события. Очередь пуста, когда rpos равняется spos.

## Стеки

*Стек* (stack) является как бы противоположностью очереди, поскольку он работает по принципу "последним пришел – первым вышел" (last-in, first-out, LIFO). Иными словами, *в магазинном порядке*.

Чтобы наглядно представить себе стек, вспомните стопку тарелок. Первая тарелка, стоящая на столе, будет использована последней, а последняя тарелка, положенная наверх – первой. Стеки часто применяются в системном программном обеспечении, включая компиляторы и интерпретаторы.

При работе со стеками операции занесения и извлечения элемента являются основными. Данные операции традиционно называются "затолкать в стек" (push)и "вытолкнуть из стека" (pop). Поэтому для реализации стека необходимо написать две функции: push(), которая "заталкивает" значение в стек, и pop(), которая "выталкивает" значение из стека. Также необходимо выделить область памяти, которая будет использоваться в качестве стека. Для этой цели можно отвести массив или динамически выделить фрагмент памяти с помощью функций языка С, предусмотренных для динамического распределения памяти. Как и в случае очереди, функция извлечения получает из списка элемент и удаляет его, если он не хранится где-либо еше. Ниже приведена общая форма функций push() и pop(), работающих с целочисленным массивом.

int stack[MAX];

int tos=0; /\* вершина стека \*/

/\* Затолкать элемент в стек. \*/

void push(int i)

{

if(tos >= MAX) {

printf("Стак полон\n");

return;

}

stack[tos] = i;

tos++;

}

/\* Получить верхний элемент стека. \*/

int pop(void)

{

tos--;

if(tos < 0) {

printf("Стек пуст\n");

return 0;

}

return stack[tos];

}

## Связные списки

Связный список – базовая динамическая структура данных в информатике, состоящая из узлов, каждый из которых содержит как собственно данные, так и одну или две ссылки («связки») на следующий и/или предыдущий узел списка.

В отличие от стека или очереди, *связный список* допускает гибкие способы доступа, поскольку каждый фрагмент информации имеет ссылку на следующий элемент данных в цепочке. Кроме того, операция извлечения не приводит к удалению из списка и уничтожению элемента данных. В принципе, для этой цели необходимо ввести дополнительную специальную операцию удаления.

Связанные списки могут быть *односвязными* и *двусвязными*. Односвязный список содержит ссылку на следующий элемент данных (рисунке 9.2).

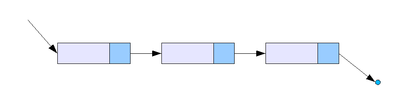


Рисунок 9. – Односвязный список

Двусвязный список содержит ссылки как на последующий, так и на предыдущий элементы списка (рисунке 9.3).

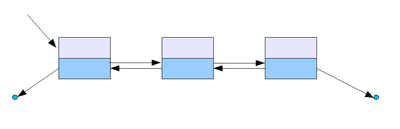


Рисунок 9. – Двусвязный список

В односвязном списке можно передвигаться только в сторону конца списка. Узнать адрес предыдущего элемента, опираясь на содержимое текущего узла, невозможно.

По двусвязному списку можно передвигаться в любом направлении – как к началу, так и к концу. В этом списке проще производить удаление и перестановку элементов, так как всегда известны адреса тех элементов списка, указатели которых направлены на изменяемый элемент.

Выбор типа применяемого списка зависит от конкретной задачи.

Разновидностью связных списков является *кольцевой (циклический, замкнутый) список*. Он тоже может быть односвязным или двусвязным. Последний элемент кольцевого списка содержит указатель на первый, а первый (в случае двусвязного списка) – на последний (рисунке 9.4).

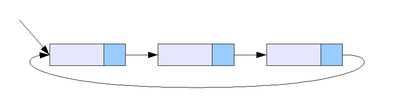


Рисунок 9. – Кольцевой односвязный список

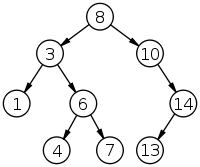
Связные списки имеют следующие преимущества в сравнении с динамическими массивами:

* лёгкость добавления и удаления элементов;
* размер ограничен только объёмом памяти компьютера и разрядностью указателей.

Недостатки:

* сложность определения адреса элемента по его индексу (номеру) в списке;
* на поля-указатели (указатели на следующий и предыдущий элемент) расходуется дополнительная память (в массивах, например, указатели не нужны);
* работа со списком медленнее, чем с массивами, так как к любому элементу списка можно обратиться, только пройдя все предшествующие ему элементы;
* элементы списка могут быть расположены в памяти разреженно, что окажет негативный эффект на кэширование процессора;
* над связными списками гораздо труднее (хотя и в принципе возможно) производить параллельные векторные операции, такие как вычисление суммы;
* кэш-промахи при обходе списка.

## Бинарные деревья



Для ознакомления с другими структурами данных рекомендуется посетить страницу «[Список структур данных](https://ru.wikipedia.org/wiki/Список_структур_данных)» на сайте Wikipedia.org:

# Функции стандарта POSIX

# Критерии качества программ на языке C

# Список литературы

# Список литературы

**Ликбез по типизации в языках программрования** [В Интернете] / авт. Груздев Денис // habrahabr.ru. - 3 Декабря 2012 г.. - http://habrahabr.ru/post/161205/.

**Полный справочник по C, 4-е издание** [Книга] / авт. Шилдт Герберт.. - Москва : Издательский дом "Вильямс", 2008. - стр. 704.

**Список структур данных** [В Интернете] // Wikipedia.org. - https://ru.wikipedia.org/wiki/Список\_структур\_данных.