Основы C

Ник Парланте перевод by bread

В этом документе Stanford CS Education сделана попытка кратко описать все основные возможности языка C language. Документ написан довольно быстро, поэтому он подходит для ознакомления или для тех, кто уже имеет некоторый опыт программирования на другом языке. Темы включают переменные, типы int, типы с плавающей точкой, продвижение, усечение, операторы, управляющие структуры (if, while, for), функции, параметры значения, параметры ссылки, структуры, указатели, массивы, препроцессор и стандартные функции библиотеки C.

Самая последняя версия всегда находится на сайте Stanford CS Education Library URL http://cslibrary.stanford.edu/101/. Пожалуйста, присылайте свои комментарии tonick.parlante@cs.stanford.edu.

Я надеюсь, что вы сможете поделиться этим документом и насладиться им в духе доброй воли, в которой он раздается - Ник Парланте, 4/2003, Стэнфорд, Калифорния.

Таблица содержимого

Ресурсы

- Язык программирования C, 2-е издание, авторы Керниган и Ритчи. Тонкая книга, которая долгие годы была библией для всех программистов на языке Си. Написана оригинальными разработчиками языка. Объяснения довольно краткие, поэтому эта книга больше подходит в качестве справочника, чем для начинающих.

- http://cslibrary.stanford.edu/102/ Указатели и память. О локальной памяти, указателях, параметрах ссылок и куче памяти рассказано гораздо подробнее, чем в этой статье, а память - это действительно самая сложная часть языка C и C++.

- http://cslibrary.stanford.edu//103/ Основы работы со связанными списками. После того как вы поняли основы работы с указателями и языком C, эти задачи помогут вам попрактиковаться.

Язык C

Си - это профессиональный язык программиста. Он был создан для того, чтобы как можно меньше мешать человеку. Керниган и Ритчи написали оригинальное определение языка в своей книге *The C Programming Language* (см. ниже) в рамках своих исследований в AT&T. Unix и C++ появились в тех же лабораториях. В течение нескольких лет я пользовался услугами AT&T как оператора междугородней связи в знак признательности за все эти исследования в области CS, но, услышав в миллионный раз "спасибо за то, что пользуетесь AT&T", я исчерпал запас доброй воли.

Некоторые языки просты. Программисту требуется лишь базовое представление о том, как все работает. Ошибки в коде отмечаются системой компиляции или времени выполнения, и программист может продираться сквозь них и в конце концов исправить все так, чтобы все работало правильно. С языком С все не так.

Модель программирования на языке С заключается в том, что программист точно знает, что он хочет сделать и как использовать конструкции языка для достижения этой цели. Язык позволяет эксперту-программисту выразить то, что он хочет, за минимальное время, не мешая ему.

Язык С "прост" тем, что количество компонентов в нем невелико. Если две языковые функции выполняют примерно одно и то же действие, то в С будет включена только одна из них. Синтаксис языка С является более строгим, и язык не ограничивает то, что "разрешено" - программист может делать практически все, что захочет.

Система типов и проверки ошибок в языке С существуют только во время компиляции. Скомпилированный код запускается в урезанной модели времени выполнения без проверок безопасности для плохих приведений типов, плохих индексов массивов или плохих указателей. Нет сборщика мусора для управления памятью. Вместо этого программист управляет кучей памяти вручную. Все это делает язык Си быстрым, но хрупким

Анализ - Где подходит C

Из-за вышеперечисленных особенностей язык С сложен для начинающих. Функция может отлично работать в одном контексте, но дать сбой в другом. Программисту необходимо понять, как работают функции, и правильно их использовать. С другой стороны, количество функций довольно невелико.

Как и большинство программистов, я испытывал настоящую ненависть к языку С. Он может быть раздражающе послушным - вы вводите что-то неправильно, а он имеет свойство компилироваться нормально и просто делать то, чего вы не ожидаете во время выполнения. Однако, став более опытным программистом, я полюбил прямой стиль языка Си. Я научился не попадать в его маленькие ловушки и ценю его простоту.

Возможно, лучший совет - просто быть осторожным. Не вводите то, чего не понимаете. Отладка занимает слишком много времени. Мысленно представляйте (или реально рисуйте), как ваш код использует память. Это хороший совет для любого языка, но для С он крайне важен.

Perl и Java более "переносимы", чем С (их можно запускать на разных компьютерах без перекомпиляции). Java и C++ более структурированы, чем С. Структура полезна для больших проектов. С лучше всего подходит для небольших проектов, где важна производительность, а у программистов есть время и навыки, чтобы сделать все на С. В любом случае, С - очень популярный и влиятельный язык. В основном это объясняется чистым (пусть и минимальным) стилем Си, отсутствием раздражающих или вызывающих сожаление конструкций, а также относительной простотой написания компилятора Си.

Раздел 1 Основные типы и операторы

Язык С предоставляет стандартный, минимальный набор базовых типов данных. Иногда их называют "примитивными" типами. На основе этих базовых типов можно создавать более сложные структуры данных.

Целочисленные типы

Интегральные типы в С образуют семейство целочисленных типов. Все они ведут себя как целые числа, их можно смешивать и использовать аналогичными способами. Различия связаны с разным количеством битов ("шириной"), используемых для реализации каждого типа - более широкие типы могут хранить больший диапазон значений.

char	ASCII-символ - не менее 8 бит. На практике char всегда является байтом, который состоит из 8 бит, что достаточно для хранения одного символа ASCII. 8 бит обеспечивают знаковый диапазон -128...127 или беззнаковый диапазон -0...255. char также должен быть "наименьшей адресуемой единицей" для машины - каждый байт в памяти имеет свой адрес.
short	Маленькое целое число - не менее 16 бит, обеспечивающее знаковый диапазон -32768...32767. Типичный размер - 16 бит. Используется не так часто
int	Целое число по умолчанию - не менее 16 бит, обычно 32 бита. Определяется как "наиболее удобный" размер для компьютера. Если вам не очень важен диапазон для целочисленной переменной, объявите ее int, так как это, скорее всего, будет подходящий размер (16 или 32 бита), который хорошо работает на данной машине.
long	Большое целое число - не менее 32 бит. Типичный размер - 32 бита, что дает подписанный диапазон примерно -2 миллиарда ...+2 миллиарда. Некоторые компиляторы поддерживают "long long" для 64-битных чисел.

Перед целыми типами может стоять квалификатор unsigned, который запрещает представление отрицательных чисел, но удваивает наибольшее положительное число, которое можно представить. Например, 16-битная реализация short может хранить числа в диапазоне 32768...32767, а unsigned short может хранить 0...65535. Вы можете считать указатели разновидностью unsigned long на машине с 4-байтовыми указателями. На мой взгляд, лучше избегать использования unsigned, если в этом нет особой необходимости. Он вызывает больше непонимания и проблем, чем стоит того.

Дополнительно: Проблемы портативности

Вместо того чтобы определять точные размеры целых типов, Си определяет нижние границы. Это облегчает реализацию компиляторов Си на широком спектре аппаратных средств. К сожалению, это иногда приводит к ошибкам, когда программа на 16-битной машине работает иначе, чем на 32-битной. В частности, если вы разрабатываете функцию, которая будет реализовываться на нескольких разных машинах, хорошей идеей будет использование типизаторов для задания таких типов, как Int32 для 32-битных int и Int16 для 16-битных int. Таким образом, вы можете создать прототип функции Foo(Int32) и быть уверенным, что для каждой машины будут заданы свои типы, и функция действительно примет именно 32-битный int. Таким образом, код будет вести себя одинаково на всех разных машинах.

char Константы

Константа char записывается в одинарных кавычках ('), например 'A' или 'z'. Константа char 'A' на самом деле является синонимом обычного целочисленного значения 65, которое является ASCII-значением для uppercase 'A'. Существуют специальные константы char, такие как '\t' для табуляции, для символов, которые неудобно набирать на клавиатуре.

'A'	прописной символ 'A'
'\n'	символ новой строки

'\t'	символ табуляции
'\0'	"нулевой" символ -- целочисленное значение 0 (отличается от цифры '0')
'\012'	символ со значением 12 в восьмеричной системе, которая является десятичной 10

int Константы

Числа в исходном коде, такие как 234, по умолчанию относятся к типу int. За ними может следовать 'L' (в верхнем или нижнем регистре), чтобы обозначить, что константа должна быть длинной, например 42L.Целочисленная константа может быть записана с ведущим 0x, чтобы указать, что она выражается в шестнадцатеричной системе счисления - 0x10 является способом выражения числа 16. Аналогично, константа может быть записана в восьмеричной системе с помощью предшествующего ей "0" - 012 является способом выражения числа 10.

Сочетание типов и продвижение

Интегральные типы можно смешивать в арифметических выражениях, поскольку все они, по сути, являются целыми числами с разной шириной. Например, char и int можно объединять в арифметических выражениях типа ('b' + 5). Как компилятор справится с разной шириной, присутствующей в таком выражении? В этом случае перед объединением значений компилятор "продвигает" меньший тип (char) до того же размера, что и больший тип (int). Продвижение определяется во время компиляции, основываясь исключительно на типах значений в выражениях. Продвижение не приводит к потере информации - оно всегда преобразует тип к совместимому большему типу, чтобы избежать потери информации.

Подводный камень - переполнение int

Однажды у меня была часть кода, которая пыталась вычислить количество байт в буфере с помощью выражения (k * 1024), где k - это int, представляющий нужное мне количество килобайт. К сожалению, это происходило на машине, где int был равен 16 битам. Поскольку и k, и 1024 были int, повышения не произошло. Для значений k >= 32 произведение было слишком большим, чтобы поместиться в 16-битный int, что приводило к переполнению. В ситуациях переполнения компилятор может делать все, что ему заблагорассудится - обычно старшие биты просто исчезают. Одним из способов исправить код было переписать его как (k * 1024L) - длинная константа заставляла переходить в int. Отслеживать этот баг было не очень весело - в исходном коде выражение выглядело вполне разумно. Только переход в отладчике за ключевую строку показал проблему переполнения. "Язык профессионального программиста". Этот пример также демонстрирует, как C продвигает только на основе типов в выражении. Компилятор не учитывает значения 32 или 1024, чтобы понять, что операция будет переполнена (в общем случае эти значения все равно не существуют до времени выполнения). Компилятор просто смотрит на типы времени компиляции, в данном случае int и int, и считает, что все в порядке.

Типы с плавающей точкой

float	Число с плавающей точкой одинарной точности, типичный размер: 32 бита
double	Число с плавающей точкой двойной точности, типичный размер: 64 бита
long double	Возможно, еще большее число с плавающей точкой (несколько неясно)

Константы в исходном коде, например, в 3.14, по умолчанию имеют тип double, если они не имеют суффикса 'f' (float) или 'l' (long double). Одинарная точность равна примерно 6 цифрам 5-й степени точности, а двойная - примерно 15 цифрам точности. Большинство программ на языке Си используют double для своих вычислений. Основная причина использования float - экономия памяти, если нужно хранить много чисел. Главное, что нужно помнить о числах с плавающей точкой, - это то, что они неточны. Например, каково значение следующего выражения double?

```
(1.0 / 3.0 + 1.0 / 3.0 + 1.0 / 3.0)  // точно ли это равно 1.0?
```

Сумма может быть равна 1.0, а может и не быть, и она может отличаться в зависимости от типа машины. По этой причине никогда не сравнивайте плавающие числа друг с другом для сравнения качества (==) - вместо этого используйте сравнение неравенства (<). Поймите, что

корректная программа на языке Си, запущенная на разных компьютерах, может выдавать немного разные результаты в крайних правых цифрах вычислений с плавающей точкой.

Комментарии

Комментарии в C заключаются в пары косая черта/звезда: /* ... комментарии ... */, которые могут пересекать несколько строк. В C++ появилась форма комментария, начинающаяся с двух косых черт и продолжающаяся до конца строки: // *комментарий до конца строки*
Форма комментария // настолько удобна, что многие компиляторы языка Си теперь также поддерживают ее, хотя технически она не является частью языка Си.

Наряду с хорошо подобранными именами функций, комментарии являются важной частью хорошо написанного кода. Комментарии не должны просто повторять то, что написано в коде. Комментарии должны описывать, чего добивается код, что гораздо интереснее, чем перевод того, что делает каждый оператор. Комментарии также должны рассказывать о том, что является сложным или неочевидным в том или ином участке кода.

Переменные

Как и в большинстве языков, объявление переменной резервирует и называет область в памяти во время выполнения программы для хранения значения определенного типа. Синтаксически в языке C сначала указывается тип, а затем имя переменной. В следующей строке объявляется переменная int с именем "num", а во второй строке в num записывается значение 42.

```
int num;
num = 42;
```

Переменная соответствует области памяти, которая может хранить значение заданного типа. Создание рисунка - отличный способ вспомнить о переменных в программе. Нарисуйте переменную в виде квадрата с текущим значением внутри квадрата. Может показаться, что это техника для начинающих, но когда я погряз в ужасно сложных проблемах программирования, я неизменно прибегаю к созданию рисунка, чтобы помочь обдумать проблему.

При выделении памяти во время выполнения переменные, такие как num, не очищаются и не устанавливаются. Переменные начинаются со случайных значений, и от программы зависит, установит ли она для них какое-нибудь разумное значение, прежде чем начать использовать их.

Имена в языке C чувствительны к регистру, поэтому "x" и "X" относятся к разным переменным. Имена могут содержать цифры и символы подчеркивания (_), но не могут начинаться с цифры. Несколько переменных можно объявить после типа, разделив их запятыми. Язык C является классическим языком "compile time" - имена переменных, их типы и реализации вычисляются компилятором во время компиляции (в отличие от интерпретатора, который выясняет такие детали во время выполнения программы).

```
float x, y, z, X;
```

Оператор присваивания =

Оператор присваивания - это одиночный знак равенства (=).

```
i = 6;
i = i + 1;
```

Оператор присваивания копирует значение из правой части переменной в левую часть. Присваивание также действует как выражение, которое возвращает вновь присвоенное значение. Некоторые программисты используют эту возможность для написания таких фраз, как.

```
y = (x = 2 * x); // double x, and also put x's new value in y
```

Усечение

Согласно противоположности повышению, усечение перемещает значение из типа в меньший тип. В этом случае компилятор просто отбрасывает лишние биты. Он может генерировать или не генерировать предупреждение о потере информации. При присваивании от целого числа к меньшему целому числу (например, от long к int или от int к char) отбрасываются наиболее значимые биты. При присваивании от типа с плавающей точкой к целому числу отбрасывается дробная часть числа.

```
char ch;
int i;
i = 321;
ch = i; // truncation of an int value to fit in a char
// ch is now 65
```

При выполнении задания старшие биты числа int 321 будут отброшены. Младшие 8 бит числа 321 представляют собой число 65 (321 - 256). Таким образом, значение ch будет (char)65, что соответствует 'A'.

Присвоение типа с плавающей точкой целому типу отбрасывает дробную часть числа. Следующий код установит i в значение 3. Это происходит при присваивании числа с плавающей точкой целому числу или при передаче числа с плавающей точкой в функцию, принимающую целое число.

```
double pi;
int i;
pi = 3.14159;
i = pi; // truncation of a double to fit in an int
// i is now 3
```

Падение в яму - int vs. Арифметика float

Здесь приведен пример кода, в котором арифметика int vs. float может вызвать проблемы. Предположим, что следующий код должен масштабировать оценку домашнего задания в диапазоне 0...20 в диапазон 0...100.

```
{
        int score;
        ...// предположим, что счет будет задан в диапазоне 0...20.
        score = (score / 20) * 100; // НЕТ - score/20 усекается до 0
...
```

К сожалению, в этом коде оценка почти всегда будет равна 0, потому что целочисленное деление в выражении (оценка/20) будет равно 0 для каждого значения оценки меньше 20.

```
score = ((double)score / 20) * 100; // OK -- floating point division from cast
score = (score / 20.0) * 100; // OK -- floating point division from 20.0
score = (int)(score / 20.0) * 100; // NO -- the (int) truncates the floating
// quotient back to 0
```

Нет Boolean - Используй int

C не имеет отдельного булевого типа - вместо него используется int. Язык рассматривает integer0 как false, а все ненулевые значения как true. Поэтому утверждение...

```
i = 0;
while (i - 10) {
...
```

Математические операторы

К ним относятся обычные двоичные и унарные арифметические операторы. Таблица старшинства приведена в приложении. Лично я просто широко использую круглые скобки, чтобы избежать ошибок из-за непонимания старшинства. Операторы чувствительны к типу операндов. Так, деление (/) с двумя целочисленными аргументами будет выполнять целочисленное деление. Если один из аргументов является плавающей запятой, то выполняется деление с плавающей запятой. Так, (6/4) оценивается в 1, а (6/4.0)- в 1.5 - 6 перед делением преобразуется в 6.0.

```
+ Addition
- Subtraction
/ Division
* Multiplication
% Remainder (mod)
```

Унарные операторы инкремента: ++ -

Унарные операторы ++ и -- увеличивают или уменьшают значение переменной. Для обоих операторов существуют варианты "pre" и "post", которые выполняют несколько разные действия (пояснения приведены ниже)

```
var++     increment     "post" variant
++var     increment     "pre"  variant
var--     decrement     "post" variant
--var     decrement     "pre"  variant
```

```
int i = 42;
i++; // increment on i
// i is now 43
i--; // decrement on i
// i is now 42
```

Вариации Pre и Post

Вариация Pre/Post связана с вложением переменной с оператором инкремента или декремента внутри выражения - должно ли все выражение представлять значение переменной переменной до или после изменения? Я никогда не использую операторы таким образом (см. ниже), но пример выглядит так...

```
int i = 42;
int j;
j = (i++ + 10);
// i is now 43
// j is now 52 (NOT 53)
j = (++i + 10)
// i is now 44
// j is now 54
```

Умение программировать на Си и проблемы эго

Опора на разницу между пред- и пост-вариантами этих операторов - это классическая область проявления эгоизма программиста на языке Си. Синтаксис немного запутанный. Он делает код немного короче. Эти качества побуждают некоторых программистов на Си демонстрировать, насколько они умны. какие они умные. Си приглашает к подобным действиям, поскольку в языке есть много областей (это лишь один из пример), где программист может получить сложный эффект, используя короткий и плотный.

Если я хочу, чтобы j зависело от значения i до инкремента, я пишу...

```
j = (i + 10);
i++;
```

Или если я хочу, чтобы j использовал значение после инкремента, я пишу...

```
i++;
j = (i + 10);
```

Разве это не лучше? (редакционная статья) Создавайте программы, которые делают что-то крутое, а не а не программы, которые подгоняют синтаксис языка. Синтаксис - кого это волнует?

Реляционные операторы

Они оперируют с целыми числами или значениями с плавающей точкой и возвращают булево значение 0 или 1.

==	Равные
!=	Не равно
>	Больше, чем
<	Меньше, чем
>=	Больше или равно
<=	Меньше или равно

Чтобы узнать, равен ли x трем, напишите что-то вроде:

```
if (x == 3) ...
```

Подводный камень = не ==

Абсолютно классический подводный камень - написать присваивание (=), когда подразумевается сравнение (==). В этом не было бы ничего страшного, за исключением того, что неправильная версия присваивания компилируется нормально, поскольку компилятор предполагает, что вы хотите использовать значение, возвращаемое присваиванием. Это редко бывает тем, что вам нужно.

```
if (x = 3) ...
```

Это не проверяет, равен ли x 3. Это устанавливает x в значение 3, а затем возвращает 3 в if для проверки. 3 - это не 0, поэтому оно каждый раз считается "истинным". Это, пожалуй, самая распространенная ошибка, которую допускают начинающие программисты на Си. Проблема в том, что компилятор ничем не может помочь - он считает, что обе формы подходят, поэтому единственная защита - предельная бдительность при кодировании. Или напишите "= ≠ ==" большими буквами на тыльной стороне ладони перед кодированием. Эта ошибка - абсолютная классика, и ее очень трудно отлаживать. Осторожно! И нужно ли говорить: "Язык профессионального программиста".

Логические операторы

Значение 0 является ложным, все остальное - истинным. Операторы оценивают слева направо и останавливаются сразу после того, как можно вывести истинность или ложность выражения. (Такие операторы называются "замыканием") В ANSI C они, кроме того, гарантированно используют 1 для обозначения истины, а не просто какой-то случайный ненулевой бит. Однако существует множество программ на языке C, в которых для обозначения true используются значения, отличные от 1 (например, ненулевые указатели), поэтому при программировании не следует полагать, что истинная булева величина обязательно равна 1.

!	Булево НЕ (унарное)
&&	Булево И
||	Булево ИЛИ

Побитовые операторы

C включает операторы для манипулирования памятью на битовом уровне. Это полезно для написания низкоуровневого кода аппаратного обеспечения или операционной системы, когда обычных абстракций чисел, символов, указателей и т. д... недостаточно - все более редкая необходимость. Код с битовыми манипуляциями имеет тенденцию быть менее "переносимым". Код является "переносимым", если без вмешательства программиста он корректно компилируется и работает на разных типах компьютеров. Побитовые операции типично используются с беззнаковыми типами. В частности, операции сдвига гарантированно переставляют 0 бит на освободившиеся позиции при использовании беззнаковых значений.

~	Побитовое отрицание (унарное) - перевернуть 0 в 1 и 1 в 0.
&	Побитовое И
|	Побитовое или
^	Побитовое исключающее или
>>	Сдвиг вправо на правую сторону (RHS) (деление на степень 2)
<<	Сдвиг влево на RHS (умножение на степень 2)

Не путайте побитовые операторы с логическими операторами. Побитовые операторы имеют ширину в один символ (&, |), в то время как логические операторы имеют ширину в два символа (&&,||). Побитовые операторы имеют более высокий приоритет, чем логические операторы. Компилятор никогда не выдаст вам ошибку типа, если вы используете &, когда подразумевали &&. С точки зрения программы проверки типов, они идентичны - оба принимают и производят целые числа, поскольку не существует отдельного булевого типа.

Другие операторы присваивания

Помимо простого оператора =, в языке C существует множество сокращенных операторов, которые представляют собой. Например, "+=" добавляет правую часть к левой. x = x + 10; может быть сокращено до x += 10;. Это наиболее полезно, если x - длинное число и в некоторых случаях может выполняться немного быстрее.

```
person->relatives.mom.numChildren += 2;          // increase children by 2
```

Вот список сокращенных операторов присваивания...

+=, -=	Увеличение или уменьшение на RHS
*=, /=	Умножение или деление на RHS
%=	Mod на RHS
>>=	Побитовый сдвиг вправо на RHS (деление на степень 2)
<<=	Побитовый сдвиг влево RHS (умножение на степень 2)
&=, |=, ^=	Побитовый и, или, xor на RHS

Раздел 2 Структуры управления

Фигурные скобки {}

В языке C фигурные скобки ({}) используются для объединения нескольких операторов. Операторы выполняются в порядке очереди. Некоторые языки позволяют объявлять переменные в любой строке (C++). Другие языки настаивают на том, чтобы переменные объявлялись только в начале функций (Pascal). C придерживается средней позиции - переменные можно объявлять в теле функции, но они должны следовать за символом '{'. Более современные языки, такие как Java и C++, позволяют объявлять переменные в любой строке, что очень удобно.

Утверждение If

В C доступны как if, так и if-else. Выражение <*expression*> может быть любым валидным выражением. Круглые скобки вокруг выражения обязательны, даже если это всего лишь одна переменная.

```
if (<выражение>) <условие>     // simple form with no {}'s or else clause

if (<выражение>) {       // simple form with {}'s to group statements
        <условие>
        <условие>
}

if (<выражение>) {       // full then/else form
        <условие>
}

else {
        <условие>
}
```

Условное выражение -or- Тернарный оператор

Условное выражение можно использовать как сокращение для некоторых операторов if-else. Общий синтаксис условного оператора таков:

```
<выражение1> ? <выражение2> : <выражение3>
```

Это выражение, а не оператор, поэтому оно представляет собой значение. Оператор работает, оценивая *выражение1*. Если оно истинно (ненулевое), оно оценивает и возвращает *выражение2*. В противном случае оно оценивает и возвращает *выражение3*.

Классический пример использования тернарного оператора - возврат меньшей из двух переменных. Время от времени следующая форма - это то, что вам нужно. Вместо…

```
if (x < y) {
        min = x;
}
else {
        min = y;
}
```

Вы просто говорите…

```
min = (x < y) ? x : y;
```

Оператор switch

Оператор switch - это своего рода специализированная форма if, используемая для эффективного разделения различных блоков кода на основе значения целого числа. Выражение switch оценивается, а затем поток управления переходит к соответствующему const-выражению case. Выражения case обычно представляют собой константы int или char. Оператор switch - это, пожалуй, самая синтаксически неудобная и чреватая ошибками функция языка C.

```
switch (<выражение>) {
        case <константное-выражение-1>:
                <условие>
                break;
        case <константное-выражение-2>:
                <условие>
                break;
        case <константное-выражение-3>:     // здесь мы объединяем случаи 3 и 4
        case <константное-выражение-4>:
                <условие>
                break;
        default:            // опционально
                <условие>
}
```

Для каждой константы требуется свое ключевое слово case и двоеточие (:). После перехода к определенному случаю программа будет продолжать выполнять все случаи, начиная с этого - эта так называемая операция "fall through" используется в приведенном выше примере, чтобы выражение-3 и выражение-4 выполняли одни и те же операторы. Явные операторы break необходимы для выхода из переключателя. Пропуск операторов break является распространенной ошибкой - он компилируется, но приводит к непреднамеренному выпадению.

Почему поведение оператора switch fall-through работает именно так? Лучшее объяснение, которое я могу придумать, заключается в том, что изначально язык Си разрабатывался для аудитории, состоящей из программистов на ассемблере программистов на языке ассемблера. Программисты на языке ассемблера привыкли к идее таблицы переходов с поведением падения, поэтому в языке C это реализовано именно так (также относительно легко реализовать его таким образом). К сожалению, аудитория языка C теперь совсем другая, и поведение fall-through широко рассматривается как ужасная часть языка.

Цикл while

В цикле while тестовое выражение оценивается перед каждым циклом, поэтому он может выполняться нуль раз, если условие изначально ложно. Он требует скобки, как и цикл if.

```
while (<выражение>) {
        <условие>
}
```

Цикл Do-While

Как while, но с условием проверки в нижней части цикла. Тело цикла будет всегда будет выполняться хотя бы один раз. Do-while - непопулярная об ласть языка, большинство все стараются использовать прямой while, если это вообще возможно.

```
do {
        <условие>
} while (<выражение>)
```

Цикл For

Цикл For в языке C является наиболее общей конструкцией циклов. Заголовок цикла содержит три части: инициализацию, условие продолжения и действие.

```
for (<инициализация>; <продолжение>; <действие>) {
        <условие>
}
```

Инициализация выполняется один раз перед входом в тело цикла. Цикл продолжает выполняться до тех пор, пока условие продолжения остается истинным (как в while). После каждого выполнения цикла выполняется действие. В следующем примере цикл выполняется 10 раз, отсчитывая 0...9. Многие циклы выглядят очень похоже на следующий…

```
for (i = 0; i < 10; i++) {
        <условие>
}
```

В программах на языке C часто встречаются серии вида 0...(какое-то_число-1). В языке Си идиоматично для цикла типа above начинать с 0 и использовать < в тесте, чтобы серия выполнялась до верхней границы, но не равнялась ей. В других языках вы можете начать с 1 и использовать <= в тесте.

Каждая из трех частей цикла for может состоять из нескольких выражений, разделенных запятыми. Выражения, разделенные запятыми, выполняются по порядку, слева направо, и представляют собой значение последнего выражения. (Для демонстрации сложного цикла for см. пример с обратным переводом строки ниже).

Break

Оператор break переносит управление за пределы цикла или оператора switch. Стилистически говоря, break может показаться немного вульгарным. Предпочтительнее использовать прямой цикл с единственным тестом в верхней части, если это возможно. Иногда вы вынуждены использовать break, потому что тест может возникнуть только где-то посреди утверждений в теле цикла. Чтобы код оставался читабельным, убедитесь, что прерывание очевидно - забывание учесть действие прерывания является традиционным источником ошибок в поведении цикла.

```
while (<выражение>) {
        <условие>
        <условие>
        if (<условие, которое может быть оценено только здесь>)
                break;
        <условие>
        <условие>
} // управление переходит вниз, на перерыв
```

break не работает с if. Он работает только в циклах и переключателях. Если думать, что break относится к if, то на самом деле он относится к вложенному while, это приводит к появлению некоторых высококачественных ошибок. Если вы используете break, лучше написать цикл, который будет выполнять итерацию наиболее простым, очевидным, нормальным способом, а затем использовать break для явного отлова исключительных, странных случаев.

Continue

Оператор continue заставляет управление перейти в нижнюю часть цикла, эффективно пропуская любой код, расположенный ниже continue. Как и break, этот оператор имеет репутацию вульгарного, поэтому используйте его осторожно. Почти всегда можно добиться более явного эффекта, используя if внутри цикла.

```
while (<условие>) {
        if (<состояние>)
                continue; // управление переходит на начало цикла
}
```

Раздел 3 Сложные типы данных

В языке С есть обычные средства для объединения объектов в составные типы - массивы и записи (которые называются "структурами"). Следующее определение объявляет тип "struct fraction", который имеет два целочисленных подполя с именами "numerator" и "denominator". Если вы забудете точку с запятой, это приведет к синтаксической ошибке во всем, что следует за объявлением struct.

```
struct fraction {
        int numerator;
        int denominator;
};      // Не забудьте про точку с запятой!
```

Это объявление вводит тип struct fraction (оба слова обязательны) как новый тип. В языке С для доступа к полям записи используется точка (.). Вы можете скопировать две записи одного типа с помощью одного оператора присваивания, однако == не работает со структурами.

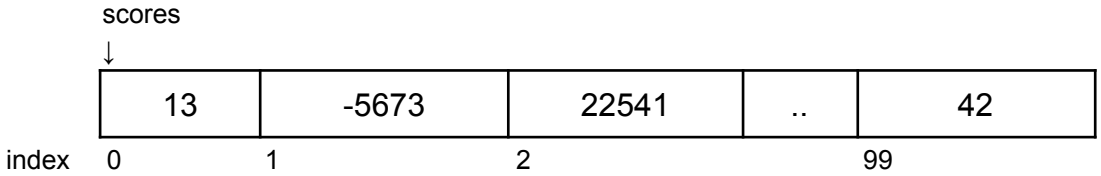
```
struct fraction f1, f2;       // объявить две дроби
f1.numerator = 22;
f1.denominator = 7;
f2 = f1;              // копирует всю структуру
```

Массивы

Простейший тип массива в языке С - это массив, который объявляется и используется в одном месте. Существуют более сложные варианты использования массивов, которые я рассмотрю позже, наряду с указателями. В следующем примере объявляется массив scores для хранения 100 целых чисел и задаются первый и последний элементы. Массивы в языке С всегда индексируются от 0. Поэтому первый int в массиве scores - scores[0], а последний - scores[99].

```
int scores[100];
scores[0] = 13;             // установить первый элемент
scores[99] = 42;            // установить последний элемент
```

Имя массива относится ко всему массиву. (реализация) он работает, представляя собой указатель на начало массива



Для каждого элемента int в массиве scores отведено место - этот элемент обозначается как scores[0].	Элементы по индексу 1 и 2 имеют случайные значения, потому что код еще не инициализировал их для чего-либо.	Чужая память с любого конца массив - не читайте и не не читайте и не записывайте эту память.

Это очень распространенная ошибка, когда вы пытаетесь обратиться к несуществующему элементу scores[100]. С не делает никакой проверки границ массивов ни во время выполнения, ни во время компиляции. Во время выполнения код просто обращается к памяти, которая ему попадется, и ведет себя непредсказуемо. "Язык профессионального программиста". В языке присутствует конвенция нумерации вещей 0...(количество вещей - 1). Чтобы лучше интегрироваться с Си и другими программистами на Си, вы должны использовать такую нумерацию и в своих собственных структурах данных

Многомерные массивы

Следующее объявляет двумерный массив целых чисел 10 на 10 и устанавливает первый и последний элементы равными 13.

```
int board [10][10];
board[0][0] = 13;
board[9][9] = 13;
```

Реализация массива хранит все элементы в одном непрерывном блоке памяти. Другой возможной реализацией была бы комбинация нескольких отдельных одномерных массивов - в языке C это не так. В памяти массив располагается так, что элементы крайнего правого индекса находятся рядом друг с другом. Другими словами, board[1][8] находится прямо перед board[1][9] в памяти.

Как правило, эффективно обращаться к памяти, которая находится рядом с другой недавно использованной памятью. Это означает, что наиболее эффективным способом чтения по частям массива является наиболее частое изменение крайнего правого индекса, так как в этом случае доступ к элементам будет осуществляться в памяти, расположенной рядом друг с другом.

Массив структур

Следующее объявляет массив с именем "numbers", который содержит 1000 структурных дробей.

```
struct fraction numbers[1000];
numbers[0].numerator = 22;      /* set the 0th struct fraction */
numbers[0].denominator = 7;
```

Вот общий прием для разгадывания объявлений переменных в языке C: посмотрите на правую часть и представьте, что это выражение. Тип этого выражения - левая часть. В приведенном выше объявлении выражение, которое выглядит как правая часть (numbers[1000], или вообще что-либо в форме numbers[...]), будет иметь тип в левой части (struct fraction).

Указатели

Указатель - это значение, представляющее собой ссылку на другое значение, иногда называемое "указателем". Надеюсь, вы уже где-то узнали об указателях, поскольку предыдущее предложение, вероятно, не является достаточным объяснением. В этом обсуждении мы сконцентрировано на синтаксисе указателей в языке C - для более полного обсуждения указателей и их использования см. http://cslibrary.stanford.edu/102/, Pointers and Memory.

Синтаксис

Синтаксически C использует звездочку или "звездочку" (*) для обозначения указателя. C определяет типы указателей на основе типа pointee. char* - это тип указателя, который ссылается на один символ. a struct fraction* - тип указателя, который ссылается на struct fraction.

```
int* intPtr;        // объявить целочисленную переменную-указатель intPtr
char* charPtr;      // объявляет символьный указатель -
                    // очень распространенный тип указателя

// Объявите два указателя на дробь struct
// (при объявлении нескольких переменных в одной строке используется символ *
// должен идти справа с переменной)
struct fraction *f1, *f2;
```

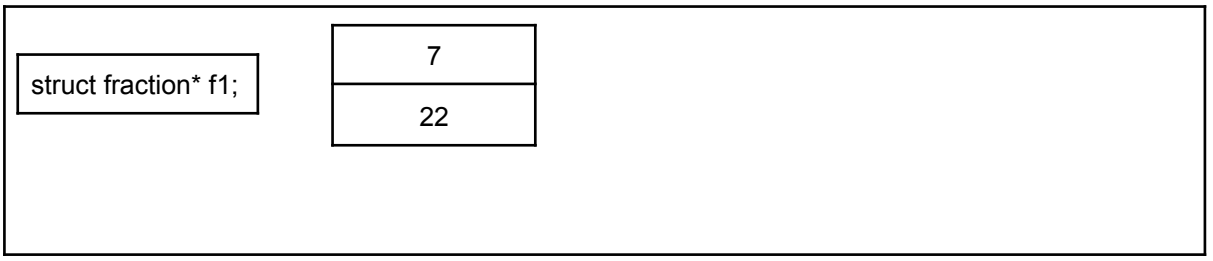
Плавающий ‘*’

В синтаксисе звездочка может находиться в любом месте между базовым типом и именем переменной. У программистов есть свои собственные соглашения. Я обычно помещаю * слева от типа. Таким образом, приведенное выше объявление intPtr можно записать эквивалентно…

```
int *intPtr;        // все они одинаковые
int * intPtr;
int* intPtr;
```

Разыменование указателя

Вскоре мы увидим, как указатель устанавливается, чтобы указать на что-то, а пока просто предположим, что указатель указывает на память соответствующего типа. В выражении унарная * слева от указателя разыменовывает его, чтобы получить значение, на которое он указывает. На следующем рисунке показаны типы, связанные с одним указателем, указывающим на дробь struct.



Существует альтернативный, более удобный синтаксис для разыменования указателя на структуру. Через "->" справа от указателя можно получить доступ к любому из полей структуры. Так, ссылка на поле числителя может быть записана как f1->numerator.

Вот несколько более сложных деклараций…

```
struct fraction** fp;                   // a pointer to a pointer to a struct fraction
struct fraction fract_array[20];        // an array of 20 struct fractions
struct fraction* fract_ptr_array[20];   // an array of 20 pointers to
// struct fractions
```

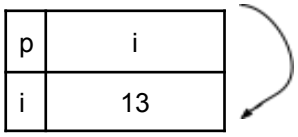
Одна из приятных особенностей синтаксиса типов языка C заключается в том, что он позволяет избежать проблем с круговыми определениями. которые возникают, когда структура указателя должна ссылаться сама на себя. Следующее определение определяет узел в связанном списке. Обратите внимание, что никакого подготовительного объявления указателя узла не требуется.

```
struct node {
        int data;
        struct node* next;
};
```

Оператор &

Оператор & - это один из способов установки указателей на объекты. Оператор & вычисляет указатель на аргумент справа от него. Аргументом может быть любая переменная, занимающая место в стеке или куче (технически она называется "LValue"). Так что &i и &(f1->numerator) подходят, а &6 - нет. Используйте &, когда у вас есть память, и вам нужен указатель на эту память.

```
void foo() {
        int* p;     // p is a pointer to an integer
        int i;      // i is an integer
        p = &i;   // Set p to point to i
        *p = 13;// Change what p points to -- in this case i -- to 13
        // At this point i is 13. So is *p. In fact *p is i.
}
```



При использовании указателя на объект, созданный с помощью &, важно использовать его только до тех пор, пока объект существует. Локальная переменная существует только до тех пор, пока выполняется функция, в которой она объявлена (мы рассмотрим функции в ближайшее время). В приведенном выше примере i существует только до тех пор, пока выполняется foo(). Поэтому все указатели, которые были инициализированы с помощью &i, действительны только до тех пор, пока выполняется функция foo(). Это ограничение "времени жизни" локальной памяти является стандартным во многих языках, и его необходимо учитывать при использовании оператора &.

NULL

Указателю можно присвоить значение 0, чтобы явно показать, что в данный момент у него нет указателя. Наличие стандартного представления "нет текущего указателя" оказывается очень удобным при использовании указателей. Константа NULL определена как 0 и обычно используется при установке указателя на NULL. Поскольку это просто 0, указатель NULL будет вести себя как булева false при использовании в булевом контексте. Разыменование указателя NULL является ошибкой, которую, если вам повезет, компьютер обнаружит во время выполнения программы - обнаружит ли он это, зависит от операционной системы.

Подводные камни - неинициализированные указатели

При использовании указателей необходимо следить за двумя сущностями. Указатель и память, на которую он указывает, иногда называемая "указателем". Для того чтобы связь указателя и памяти работала, необходимо соблюсти три условия…

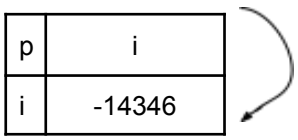
(1) Указатель должен быть объявлен и выделен

(2) Указатель должен быть объявлен и выделен

(3) Указатель (1) должен быть инициализирован так, чтобы он указывал на указатель (2)

Наиболее распространенной ошибкой, связанной с указателем, является следующая: Объявление и выделение указателя (шаг 1). Забудьте о шаге 2 и/или 3. Начните использовать указатель так, как будто он был настроен на что-то. Код с такой ошибкой часто компилируется нормально, но результаты выполнения оказываются плачевными. К сожалению, указатель не указывает ни на что хорошее, если не выполнены пункты (2) и (3), поэтому операции разыменования указателя с помощью * во время выполнения будут неправильно использовать и тратить память, что в какой-то момент приведет к случайному сбою.

```
int* p;
*p = 13; // НЕТ НЕТ НЕТ p еще не указывает на int
// Это просто перезаписывает случайную область в памяти
```



Конечно, ваш код не будет таким тривиальным, но ошибка имеет ту же основную форму: объявите указатель, но забываем настроить его так, чтобы он указывал на конкретный указатель.

Использование указателей

Объявление указателя выделяет место для самого указателя, но не выделяет место **для самого указателя**. Указатель должен на что-то указывать, прежде чем вы сможете его разыменовать.

Вот код, который не делает ничего полезного, но демонстрирует (1) (2) (3) для правильного использования указателя…

```
int* p;                    // (1) выделение указателя
int i;                     // (2) выделение указателя
struct fraction f1;        // (2) выделение указателя
p = &i;                    // (3) установите p для указания на i
*p = 42;                   // Можно использовать p, так как он настроен
p = &(f1.numerator);       // (3) настройка p для указания на другой int
*p = 22;
p = &(f1.denominator);     // (3)
*p = 7;
```

До сих пор мы использовали оператор & для создания указателей на простые переменные, такие как i. Позже мы рассмотрим другие способы получения указателей с помощью массивов и других техник.

C Strings

C имеет минимальную поддержку символьных строк. По большей части строки работают как обычные массивы символов. Их обслуживание возлагается на программиста, использующего стандартные возможности, доступные для массивов и указателей. В состав языка C входит стандартная библиотека функций, выполняющих общие операции со строками, но за управление строковой памятью и вызов нужных функций отвечает программист. К сожалению, вычисления, связанные со строками, очень распространены, поэтому, чтобы стать хорошим программистом на C, часто требуется научиться писать код, который управляет строками, что означает управление указателями и массивами.

Строка в языке Си - это просто массив символов с одним дополнительным условием: после последнего реального символа в массиве хранится нулевой символ ('\0'), чтобы отметить конец строки. Компилятор представляет строковые константы в исходном коде, такие как "binky", как массивы, которые следуют этому соглашению. Функции библиотеки строк (их подробный список приведен в приложении) работают со строками, хранящимися таким образом. Наиболее полезной библиотечной функцией является strcpy(char dest[], const char source[]), которая копирует байты одной строки в другую. Порядок аргументов в функции strcpy() аналогичен порядку аргументов в функции '=' - правый аргумент присваивается левому. Еще одна полезная строковая функция -strlen(const char string[]); которая возвращает количество символов в строке C, не считая идущего следом '\0'.

Обратите внимание, что обычный оператор присваивания (=) не выполняет копирование строк, поэтому необходима функция strcpy(). Подробнее о том, как работают массивы и указатели, читайте в разделе 6 "Расширенные указатели и массивы".

Следующий код выделяет массив из 10 символов и использует функцию strcpy() для копирования байтов строковой константы "binky" в этот локальный массив.

```
char localString[10];
strcpy(localString, "binky");
```

localString
↓

b	i	n	k	y	\0	x	x	x	x
0	1	2	...						

На рисунке в памяти показана локальная переменная localString, в которую скопирована строка "binky". Буквы занимают первые 5 символов, а символ '\0' обозначает конец строки после буквы 'y'. Символы "x" обозначают символы, которым не было присвоено какое-либо конкретное значение.

Если бы вместо этого код попытался сохранить в localString строку "I enjoy languages that have good string support", он бы просто упал во время выполнения, поскольку массив из 10 символов может содержать не более 9 символов. Большая строка будет записана в правую часть localString, перезаписывая все, что там хранилось.

Пример строкового кода

Здесь представлен умеренно сложный цикл for, который переворачивает строку, хранящуюся в локальном массиве. Он демонстрирует вызов функций стандартной библиотеки strcpy() и strlen() и показывает, что строка - это просто массив символов с символом '\0' для обозначения эффективного конца строки. Проверьте свои знания языка C о массивах и циклах for, сделав рисунок памяти для этого кода и проследив за его выполнением, чтобы увидеть, как он работает.

```
char string[1000]; // string - локальный массив из 1000 символов
int len;
strcpy(string, "binky");
len = strlen(string);
/*
        Переверните символы в строке:
        Я начинаю с самого начала и поднимаюсь вверх
        j начинается с конца и идет вниз
        i/j обмениваются своими персонажами по ходу дела, пока не встретятся
*/
int i, j;
char temp;
for (i = 0, j = len - 1; i < j; i++, j--) {
        temp = string[i];
        string[i] = string[j];
        string[j] = temp;
}
// в этот момент локальная строка должна быть "yknib"
```

"Достаточно большие" строки

В строках языка C принято, что владелец строки отвечает за выделение пространство массива, которое "достаточно велико" для хранения всего, что потребуется хранить строке. Большинство процедур не проверяют размер памяти строки, с которой они работают, они просто предполагают, что она достаточно большой, и работают дальше. Многие, очень многие программы содержат объявления, подобные следующие…

```
char localString[1000];
...
```

Программа работает нормально до тех пор, пока хранимые строки имеют длину 999 символов или меньше. Однажды, когда программе потребуется хранить строку длиной 1000 символов или больше, произойдет сбой. Такие проблемы с недостаточно большим массивом являются распространенным источником ошибок, а также источником так называемых проблем безопасности, связанных с переполнением буфера. Дополнительным недостатком этой схемы является то, что большую часть времени, когда в массиве хранятся короткие строки, 95% зарезервированной памяти фактически расходуется впустую. Лучшее решение - динамически выделять строку в куче, чтобы она имела нужный размер.

Чтобы избежать атак переполнения буфера, производственный код должен сначала проверить размер данных, чтобы убедиться, что они помещаются в целевую строку. См. функцию strlcpy() в Приложении A.

char*

Поскольку язык C работает с типами массивов, тип приведенной выше переменной localString по сути является char*. Программы на Си очень часто манипулируют строками, используя переменные типа char*, которые указывают на массивы символов. Для манипулирования фактическими символами в строке требуется код, который манипулирует лежащим в основе массивом, или использование библиотечных функций, таких как strcpy(), которые манипулируют массивом за вас. Более подробно об указателях и массивах см. в разделе 6.

TypeDef

В операторе typedef вводится сокращенное имя типа. Синтаксис...

```
typedef <type> <name>;
```

Ниже определен тип Fraction как тип (struct fraction). C чувствителен к регистру, поэтому fraction отличается от Fraction. Удобно использовать typedef для создания типов с именами в верхнем регистре и использовать версию того же слова в нижнем регистре в качестве переменной.

```
typedef struct fraction Fraction;
Fraction fraction;          // Объявите переменную "fraction" типа "Fraction".
                            // что на самом деле является просто синонимом "struct fraction".
```

Следующее типовое определение определяет имя Tree как стандартный указатель на узел двоичного дерева, где каждый узел содержит некоторые данные и указатели на "меньшее" и "большее" поддерево.

```
typedef struct treenode* Tree;
        struct treenode {
        int data;
        Tree smaller, larger;       // Эквивалентно, в этой строке можно сказать
};                                  // "struct treenode *smaller, *larger"
```

Раздел 4 Функции

Во всех языках есть конструкция для разделения и упаковки блоков кода. В языке Си для упаковки блоков кода используется "функция". Эта статья посвящена синтаксису и особенностям функций языка Си. Мотивация и дизайн для разделения вычислений на отдельные блоки - это целая дисциплина.

У функции есть имя, список аргументов, которые она принимает при вызове, и блок кода, который она выполняет при вызове. Функции языка Си определяются в текстовом файле, а имена всех функций в программе на языке Си объединяются в одно плоское пространство имен. Специальная функция, называемая "main", является местом, с которого начинается выполнение программы. Некоторые программисты предпочитают начинать имена своих функций с верхнего регистра, используя нижний регистр для переменных и параметров, Вот простое объявление функции на языке Си. Здесь объявляется функция Twice, которая принимает один аргумент int с именем num. Тело функции вычисляет значение, которое в два раза больше аргумента num, и возвращает это значение вызывающей стороне.

```
/*
        Вычисляет двойное значение числа.
        Работает, если утроить число, а затем вычесть, чтобы вернуть к удвоенному.
*/
static int Twice(int num) {
        int result = num * 3;
        result = result - num;
        return(result);
}
```

Синтаксис

Ключевое слово "static" определяет, что функция будет доступна вызывающим ее пользователям только в том файле, в котором она объявлена. Если функцию нужно вызвать из другого файла, она не может быть статической и требует прототипа - см. прототипы ниже. Статическая форма удобна для служебных функций, которые будут использоваться только в том файле, где они объявлены. Далее, "int" в приведенной выше функции - это тип ее возвращаемого значения. Далее следует имя функции и список ее параметров. При ссылке на функцию по имени в документации или другой прозе принято сохранять суффикс скобки (), поэтому в данном случае я называю функцию "Twice()". Параметры перечисляются с указанием их типов и имен, как и переменные.

Внутри функции параметр num и локальная переменная result являются "локальными" для функции - они получают свою собственную память и существуют только до тех пор, пока функция выполняется. Такая независимость "локальной" памяти является стандартной особенностью большинства языков (подробное обсуждение локальной памяти см. в разделе CSLibrary/102).

Код "вызывающего", который вызывает Twice(), выглядит как…

```
int num = 13;
int a = 1;
int b = 2;
a = Twice(a);               // call Twice() passing the value of a
b = Twice(b + num);         // call Twice() passing the value b+num
// a == 2
// b == 30
// num == 13                // this num is totally independent of the "num" local to Twice()
```

На что стоит обратить внимание…

(словарь) Выражение, передаваемое функции вызывающим ее пользователем, называется "фактическим параметром" - например, "a" и "b + num" выше. Хранилище параметров, локальное для функции, называется "формальным параметром", например "num" в "static int Twice(intnum)".

Параметры передаются "по значению", что означает единственную операцию копирования присваивания (=) из каждого фактического параметра для установки каждого формального параметра. Фактический параметр оценивается в контексте вызывающей стороны, а затем его значение копируется в формальный параметр функции непосредственно перед началом ее выполнения. Альтернативный механизм задания параметров - "по ссылке", который в C не реализован напрямую, но при необходимости программист может реализовать его вручную (см. ниже). Когда параметр является структурой, он копируется.

Переменные, локальные для Twice(), num и result, существуют только временно, пока выполняется Twice(). Это стандартное определение "локального" хранения для функций.

Возврат в конце Twice() вычисляет возвращаемое значение и завершает работу функции. Выполнение возобновляется вызывающей стороной. Внутри функции может быть несколько операторов возврата, но хорошим стилем является хотя бы один в конце, если необходимо указать возвращаемое значение. Забыть о возврате в середине функции - традиционный источник ошибок.

C-ing и Небытие - void

void - это формализованный в ANSI C тип, который означает "ничто". Чтобы указать, что функция ничего не возвращает, используйте void в качестве возвращаемого типа. Также, по соглашению, указатель, который не указывает на какой-либо конкретный тип, объявляется как void*. Иногда void* используется для того, чтобы заставить два тела кода не зависеть друг от друга, в то время как void* переводится как "это указывает на что-то, но я не говорю вам (клиенту) тип указателя, потому что вам это не нужно знать". Если функция не принимает никаких параметров, ее список параметров пуст, или она может содержать ключевое слово void, но этот стиль сейчас не в моде.

```
void TakesAnIntAndReturnsNothing(int anInt);
int TakesNothingAndReturnsAnInt();
int TakesNothingAndReturnsAnInt(void); // equivalent syntax for above
```

Вызов по значению против вызова по ссылке

C передает параметры "по значению", что означает, что фактические значения параметров копируются в локальную память. Вызывающая и вызываемая функции не делят память - у каждой из них есть своя копия. Эта схема подходит для многих целей, но у нее есть два недостатка.

1) Поскольку у вызывающей стороны есть своя собственная копия, изменения в этой памяти не передаются обратно вызывающей стороне. Поэтому параметры значения не позволяют вызывающему сообщать об этом. Возвращаемое значение функции может передать некоторую информацию вызывающей стороне, но не все проблемы можно решить с помощью единственного возвращаемого значения.

2) Иногда нежелательно копировать значение от вызывающей стороны к вызываемой, потому что значение большое и копировать его дорого, или потому что на концептуальном уровне копирование значения нежелательно.

Альтернативой является передача аргументов "по ссылке". Вместо того чтобы передавать копию значения avalue от вызывающей стороны к вызываемой, передайте указатель на значение. Таким образом, в любой момент времени существует только одна копия значения, и вызывающий и вызываемый получают доступ к этому единственному значению через указатели.

Некоторые языки поддерживают параметры ссылок автоматически. В языке C этого нет - программист должен реализовать ссылочные параметры вручную, используя существующие в языке конструкции указателей.

Пример подкачки

Классическим примером желания изменить память вызывающей стороны является функция swap(), которая обменивается двумя значениями. Поскольку в языке C используется вызов по значению, следующая версия Swap не будет работать…

```
void Swap(int x, int y) {  // NO does not work
        int temp;
        temp = x;
        x = y;          // эти операции просто изменяют локальные значения x, y, temp
        y = temp;       // - Ничто не соединяет их с вызывающими a,b.
}
// Some caller code which calls Swap()...
int a = 1;
int b = 2;
Swap(a, b);
```

Swap() не влияет на аргументы a и b в вызывающей функции. Приведенная выше функция работает только с копиями a и b, локальными для самой Swap(). Это хороший пример того, как ведет себя "локальная" память типа (x, y, temp) - она существует независимо от всего остального только пока работает ее собственная функция. Когда функция-владелец завершает работу, ее локальная память исчезает.

Техника ссылочных параметров

Чтобы передать объект X в качестве ссылочного параметра, программист должен передать указатель на X вместо этого самого X. Формальный параметр будет указателем на интересующее значение. Вызывающая сторона должна будет использовать & или другие операторы для вычисления правильного фактического параметра указателя. Вызывающий должен будет разыменовать указатель с помощью *, где это необходимо, чтобы получить доступ к интересующему значению. Вот пример корректной функции Swap().

```
static void Swap(int* x, int* y) {   // params are int* instead of int
        int temp;
        temp = *x;                   // use * to follow the pointer back to the caller's memory
        *x = *y;
        *y = temp;
}
// Some caller code which calls Swap()...
int a = 1;
int b = 2;
Swap(&a, &b);
```

На что стоит обратить внимание…

- Формальные параметры - int* вместо int.
- Вызывающая сторона использует & для вычисления указателей на свою локальную память (a,b).
- Вызывающая сторона использует * для разыменования указателей на формальные параметры, чтобы получить память вызывающей стороны.

Поскольку оператор & выдает адрес переменной - &a является указателем на a. В самой функции Swap() формальные параметры объявлены как указатели, и доступ к интересующим значениям (a,b) осуществляется через них. Между именами, используемыми для фактических и формальных параметров, нет никаких особых отношений. Вызов функции сопоставляет фактические и формальные параметры по их порядку - первый фактический параметр присваивается первому формальному параметру, и так далее. Я намеренно использовал разные имена (a,b против x,y), чтобы подчеркнуть, что имена не имеют значения.

const

Квалификатор const может быть добавлен слева от переменной или типа параметра, чтобы объявить, что код, использующий переменную, не будет ее изменять. На практике в сообществе программистов на языке Си const используется очень редко. Однако у него есть одно очень удобное применение, которое заключается в уточнении роли параметра в прототипе функции...

```
void foo(const struct fraction* fract);
```

В прототипе foo() const заявляет, что foo() не собирается изменять переданную ему struct fraction pointee. Поскольку дробь передается по указателю, мы не могли знать, собирается ли foo() изменять нашу память или нет. Используя const, foo() проясняет свои намерения. Объявление этого дополнительного бита информации помогает прояснить роль функции для ее исполнителя и вызывающего пользователя.

Пример с большим указателем

Следующий код представляет собой большой пример использования ссылочных параметров. В этом примере есть несколько общих черт программ на Си... Ссылочные параметры используются для того, чтобы функции Swap() и IncrementAndSwap() могли влиять на память вызывающих их лиц. Внутри IncrementAndSwap() есть хитрый случай, когда она вызывает Swap() - в этом случае не требуется дополнительного использования &, поскольку параметры x, y внутри InrementAndSwap() уже являются указателями на интересующие значения. Имена переменных в программе (a, b, x, y, alice, bob) не обязательно должны совпадать каким-то определенным образом, чтобы параметры работали. Механизм параметров зависит только от типов параметров и их порядка в списке параметров, но не от их имен. Наконец, вот пример того, как выглядят несколько функций в файле и как они вызываются из функции main().

```
static void Swap(int* a, int* b) {
	int temp;
	temp = *a;
	*a = *b;
	*b = temp;
}
static void IncrementAndSwap(int* x, int* y) {
	(*x)++;
	(*y)++;
	Swap(x, y);	// don't need & here since a and b are already
			// int*'s.
}
int main() {
	int alice = 10;
	int bob = 20;
	Swap(&alice, &bob);
	// at this point alice==20 and bob==10
	IncrementAndSwap(&alice, &bob);
	// at this point alice==11 and bob==21
	return 0;
}
```

Раздел 5 Разные мелочи

main()

Выполнение программы на языке Си начинается с функции main(). Все файлы и библиотеки программы на языке Си компилируются вместе для создания одного файла программы. Этот файл должен содержать ровно одну функцию main(), которую операционная система использует в качестве начальной точки программы. Функция main() возвращает значение int, которое, по условию, равно 0, если программа завершилась успешно, и ненулевое значение, если программа завершилась из-за ошибки.

Несколько файлов

Для программы любого размера удобно разделить функции на несколько отдельных файлов. Чтобы обеспечить взаимодействие функций в отдельных файлах и при этом позволить компилятору работать с файлами независимо, программы на языке Си обычно зависят от двух особенностей…

Прототипы

Прототип" функции содержит ее имя и аргументы, но не ее тело. Чтобы вызывающая сторона в любом файле могла использовать функцию, она должна видеть прототип этой функции. Например, вот как будут выглядеть прототипы для Twice() и Swap(). Тело функции отсутствует, а прототип завершается точкой с запятой (;)...

```
int Twice(int num);
void Swap(int* a, int* b);
```

В до-ANSI C правила работы с прототипами были очень небрежными - вызывающий не обязан был видеть прототипы перед вызовом функций, и в результате можно было попасть в ситуацию, когда компилятор генерировал код, который ужасно падал.

В ANSI C я немного упрощу и скажу, что…

1) функция может быть объявлена статической, в этом случае она может использоваться только в том же файле, где она используется ниже места ее объявления. Статические функции не требуют отдельного прототипа, если они определены до или выше места их вызова, что экономит часть работы.

2) Нестатическая функция нуждается в прототипе. Когда компилятор компилирует определение функции, он должен предварительно увидеть прототип, чтобы убедиться, что они совпадают (правило "прототип перед определением"). Прототип также должен быть виден любому клиентскому коду, который хочет вызвать функцию (правило "клиенты должны видеть прототипы"). (Поведение require-prototypes на самом деле является в некоторой степени опцией компилятора, но разумно оставить его включенным).

Препроцессор

Предварительная обработка происходит с исходным текстом на языке C до того, как он попадает в компилятор. Две наиболее распространенные директивы препроцессора - #define и #include…

#define

Директива #define может быть использована для установки символьных замен в исходном тексте. Как и все операции препроцессора, #define крайне не интеллектуальна - она просто выполняет текстовую замену без понимания. Операторы #define используются как грубый способ создания символических констант.

```
#define MAX 100
#define SEVEN_WORDS that_symbol_expands_to_all_these_words
```

В последующем коде можно использовать символы MAX или SEVEN_WORDS, которые будут заменены текстом справа от каждого символа в его #define.

#include

Директива "#include" вставляет текст из разных файлов во время компиляции. Директива #include очень не интеллектуальна и не структурирована - она просто вставляет текст из заданного файла и продолжает компиляцию. Директива #include используется в приведенном ниже файловом соглашении .h/.c, которое используется для удовлетворения различных ограничений, необходимых для корректной работы прототипов.

```
#include "foo.h"        // ссылается на "пользовательский" файл foo.h -
                        // в исходном каталоге для компиляции
#include <foo.h>        // ссылается на "системный" файл foo.h -
                        // в каталоге компилятора.
```

foo.h vs foo.c

Всеобщее соглашение для языка C заключается в том, что для файла с именем "foo.c", содержащего кучу функций…

- Отдельный файл с именем foo.h будет содержать прототипы функций из foo.c, которые клиенты могут захотеть вызвать. Функции в foo.c, которые предназначены только для "внутреннего использования" и никогда не должны вызываться клиентами, должны быть объявлены статическими.
- Рядом с вершиной foo.c будет следующая строка, которая гарантирует, что определения функций в foo.c видят прототипы в foo.h. определения в foo.c видят прототипы в foo.h, что обеспечивает соблюдение правила "прототип перед определением". #include "foo.h"
  // показать содержимое "foo.h"
  // компилятору в этот момент
- Любой файл xxx.c, который хочет вызвать функцию, определенную в foo.c, должен содержать следующую строку, чтобы увидеть прототипы, обеспечивая правило "клиенты должны видеть прототипы". выше. #include "foo.h"

#if

Во время компиляции существует некоторое пространство имен, определяемых #defines. Тест #if можно использовать во время компиляции, чтобы посмотреть на эти символы и включить или выключить, какие строки использует компилятор. Следующий пример зависит от значения символа FOO #define. Если он истинен, то строки "aaa" (какими бы они ни были) компилируются, а строки "bbb" игнорируются. Если бы FOO был равен 0, то все было бы наоборот.

```
#define FOO 1
...
#if FOO
        aaa
        aaa
#else
        bbb
        bbb
#endif
```

Вы можете использовать #if 0 ... #endif, чтобы эффективно закомментировать участки кода, которые вы не хотите компилировать, но хотите сохранить в исходном файле.

Множественные #includes -- #pragma once

Иногда возникает проблема, когда файл .h #включается в файл более одного раза, что приводит к ошибкам компиляции. Это может быть серьезной проблемой. Поэтому, если это возможно, лучше избегать #включения .h-файлов в другие .h-файлы. С другой стороны, #включение.h-файлов в .c-файлы - это нормально. Если вам повезет, ваш компилятор будет поддерживать функцию #pragma once, которая автоматически предотвращает включение одного файла более одного раза в любой другой файл. Это в значительной степени решает проблему множественного #include.

```
// foo.h
// The following line prevents problems in files which #include "foo.h"
#pragma once
<rest of foo.h ...>
```

Assert

Выходящие за границы массива ссылки являются чрезвычайно распространенной формой ошибки времени выполнения на языке C. Вы можете использовать функцию assert(), чтобы посыпать свой код собственными проверками границ. Несколько секунд, потраченных на вставку утверждений assert, могут сэкономить вам часы отладки.Устранение всех ошибок - самая трудная и страшная часть написания большого программного обеспечения. Операторы утверждения - один из самых простых и эффективных помощников на этом сложном этапе.

```
#include <assert.h>
#define MAX_INTS 100
{
        int ints[MAX_INTS];
        i = foo(<something complicated>);       // Я должен быть в пределах,
                                                // Но так ли это на самом деле?
        assert(i>=0);           // утверждения о безопасности
        assert(i<MAX_INTS);
        ints[i] = 0;
}
```

В зависимости от опций, заданных во время компиляции, выражения assert() будут оставлены в коде для тестирования или могут быть проигнорированы. По этой причине в тесты assert() следует помещать только те выражения, которые не нужно оценивать для правильного функционирования программы…

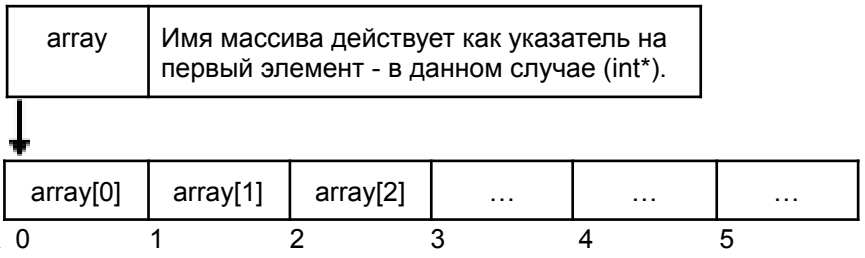
```
int errCode = foo();        // да
assert(errCode == 0);
assert(foo() == 0);         // НЕТ, foo() не будет вызвана, если
                            // компилятор удаляет assert()
```

Раздел 6 Продвинутые массивы и указатели

Массивы в языке C

В языке C массив формируется путем размещения всех элементов в памяти. Для обращения к элементам массива можно использовать синтаксис квадратных скобок. Массив в целом обозначается адресом первого элемента, который также известен как "базовый адрес" всего массива.

```
int array[6];
int sum = 0;
sum += array[0] + array[1];   // ссылайтесь на элементы с помощью []
```



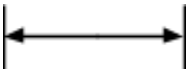
Программист может ссылаться на элементы массива с помощью простого синтаксиса [ ], например array[1]. Эта схема работает путем комбинирования базового адреса всего массива с индексом для вычисления базового адреса нужного элемента в массиве. Для этого требуется лишь немного арифметики. Каждый элемент занимает фиксированное количество байт, которое известно во время компиляции. Таким образом, адрес элемента n в массиве при использовании индексации по 0 будет находиться на расстоянии (n * element_size) байт от базового адреса всего массива.

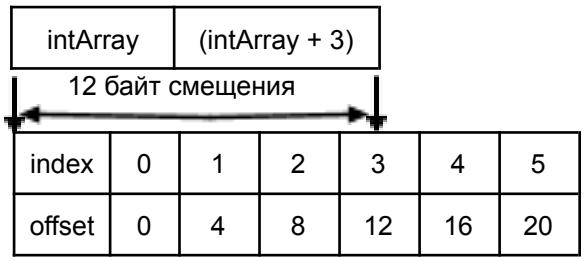
```
address of nth element = address_of_0th_element + (n * element_size_in_bytes)
```

Синтаксис квадратных скобок [ ] выполняет эту адресную арифметику за вас, но полезно знать, что он делает. знать, что он делает. Синтаксис [ ] берет целочисленный индекс, умножается на размер элемента, добавляет полученное смещение к базовому адресу массива и, наконец, разыменовывает полученный указатель чтобы перейти к нужному элементу.

```
int intArray[6];
intArray[3] = 13;
```

Предположим, что sizeof(int) = 4,
то есть каждый элемент массива
занимает 4 байта.





index	0	1	2	3	4	5
offset	0	4	8	12	16	20

```
int bytes = n * elem_size
```

Синтаксис '+'

Синтаксис + между указателем и целым числом выполняет то же вычисление со смещением, но оставляет результат в виде указателя. Синтаксис квадратных скобок дает следующий элемент, а синтаксис + - указатель на n-й элемент.

Таким образом, выражение (intArray + 3) является указателем на целое число intArray[3].(intArray + 3) имеет тип (int*), а intArray[3] - тип int. Эти два выражения отличаются только тем, разыменовывается ли указатель или нет. Таким образом, выражение(intArray + 3) в точности эквивалентно выражению (&(intArray[3])). Более того, эти два выражения, вероятно, компилируются в один и тот же код. Они оба представляют указатель на элемент с индексом 3.

Любое выражение [] можно записать с помощью синтаксиса +. Нужно только добавить разыменование указателя. Так, intArray[3] эквивалентен *(intArray + 3). Для большинства целей проще и удобнее использовать синтаксис []. Время от времени + будет удобен, если вам нужен указатель на элемент, а не сам элемент.

Pointer++ Style - strcpy()

Если p - указатель на элемент массива, то (p+1) указывает на следующий элемент в массиве массиве. Код может воспользоваться этим, используя конструкцию p++ для перебора элементов массива по указателю массива. Это не улучшает читабельность, поэтому я не могу рекомендовать эту технику, но вы можете но вы можете увидеть ее в коде, написанном другими людьми.

(Этот пример был первоначально вдохновлен Майком Клероном) Существует библиотечная функция strcpy(char* destination, char* source), которая копирует байты Cstring из одного места в другое. Ниже приведены четыре различных реализации strcpy()в порядке: от самой многословной до самой загадочной. В первой из них обычно прямой цикл while на самом деле довольно хитер, чтобы обеспечить копирование завершающего нулевого символа. Во втором эта хитрость устраняется путем переноса присваивания в тест. Последние два варианта симпатичны (и демонстрируют использование ++ для указателей), но это не совсем тот код, который вы хотите поддерживать. Из всех четырех вариантов, на мой взгляд, strcpy2() - самый удачный с точки зрения стилистики. При умном компиляторе все четыре будут компилироваться в практически одинаковый код с одинаковой эффективностью.

```
// К сожалению, прямой цикл while или for не подойдет.
// Лучшее, что мы можем сделать, это использовать while (1) с тестом в середине цикла.
void strcpy1(char dest[], const char source[]) {
	int i = 0;
	while (1) {
		dest[i] = source[i];
		if (dest[i] == '\0') break; // we're done
			i++;
	}
}
// Переместите задание в тест
void strcpy2(char dest[], const char source[]) {
	int i = 0;
	while ((dest[i] = source[i]) != '\0') {
		i++;
	}
}
// Избавьтесь от i и просто перемещайте указатели.
// Зависит от старшинства * и ++.
void strcpy3(char dest[], const char source[]) {
	while ((*dest++ = *source++) != '\0') ;
}
// Полагайтесь на то, что '\0' эквивалентно FALSE
void strcpy4(char dest[], const char source[]) {
	while (*dest++ = *source++) ;
}
```

Влияние типа указателя

Как [ ], так и + неявно используют тип указателя во время компиляции для вычисления размера элемента (theelement_size), что влияет на арифметику смещения. Глядя на код, легко предположить, что все в единицах байтов.

```
int *p;
p = p + 12; // во время выполнения, что это добавляет к p? 12?
```

Приведенный выше код не добавляет число 12 к адресу в p - это увеличило бы p на 12 байт. Приведенный выше код увеличивает p на 12 интов. Каждый int, вероятно, занимает 4 байта, поэтому во время выполнения код будет эффективно увеличивать адрес в p на 48. Компилятор настраивает все это в зависимости от типа указателя.

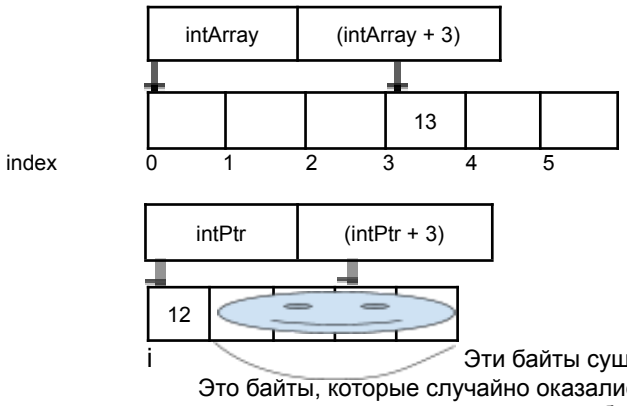
Используя приведение, следующий код на самом деле просто добавляет 12 к адресу в указателе p. Это работает, сообщая компилятору, что указатель указывает на char, а не на int. Размер char определен как ровно 1 байт (или как наименьшая адресуемая единица на компьютере). Другими словами, sizeof(char) всегда равен 1. Затем мы приводим полученное значение 36(char*) обратно к значению (int*). Программист может приводить любой тип указателя к любому другому типу указателя, чтобы изменить код, который генерирует компилятор.

```
p = (int*) ( ((char*)p) + 12);
```

Массивы и указатели

Одним из следствий схемы массивов в языке Си является то, что компилятор не делает существенного различия между массивами и указателями - они оба выглядят как указатели. В следующем примере значение intArray - это указатель на первый элемент массива, так что это (int*). Значение переменной intPtr также (int*), и она установлена так, чтобы указывать на один integer. Так в чем же разница между intArray и intPtr? С точки зрения компилятора разница невелика. Они оба являются просто указателями (int*), и компилятор вполне может применить синтаксис [] или + к любому из них. Ответственность за то, чтобы элементы, на которые ссылается операция [] или +, действительно существовали, лежит на программисте. На самом деле это просто старое правило, что С не делает никакой проверки границ. С воспринимает единственное целое число i как некий вырожденный массив размера 1.

```
int intArray[6];
int *intPtr;
int i;
intPtr = &i;
intArray[3] = 13;  // хорошо
intPtr[0] = 12;        // Странно, но хорошо. Изменения i.
intPtr[3] = 13;        // ПЛОХО! Здесь нет ни одного целого числа!
```



Эти байты существуют, но они не были явно зарезервированы. Это байты, которые случайно оказались рядом с памятью для i. Вероятно, они уже используются для хранения чего-либо, например, разбитого смайлика. 13 просто записывается вслепую поверх смайлика. Эта ошибка станет очевидной только позже, когда программа пытается прочитать данные о смайлике.

Имена массивов являются постоянными

Одним из тонких различий между массивом и указателем является то, что указатель, представляющий базовый адрес массива, не может быть изменен в коде. Базовый адрес массива ведет себя как указатель const. Это ограничение распространяется на имя массива в том месте, где он объявлен в коде - переменная ints в примере ниже.

```
int ints[100]
int *p;
int i;
ints = NULL;      // НЕТ, невозможно изменить базовое значение addr ptr
ints = &i;        // НЕТ
ints = ints + 1;  // НЕТ
ints++; // NO
p = ints;         // {ХОРОШО, p - это обычный указатель, который может быть изменен
                  // здесь он получает копию указателя ints
p++;              // ХОРОШО, p все еще можно изменить (a ints - нет).
p = NULL;         // ХОРОШО
p = &i;           // ХОРОШО
foo(ints);        // ХОРОШО (возможные определения foo приведены ниже)
```

Параметры массива передаются как указатели. Следующие два определения foo выглядят по-разному, но для компилятора они означают одно и то же. Для удобства чтения предпочтительнее использовать тот синтаксис, который точнее. Если передаваемый указатель действительно является базовым адресом целого массива, то используйте [ ].

```
void foo(int arrayParam[]) {
        arrayParam = NULL;     // Глупо, но обоснованно. Просто меняет локальный указатель
}
void foo(int *arrayParam) {
        arrayParam = NULL;      // тоже
}
```

Память кучи

C предоставляет программистам стандартный набор средств для выделения и удаления динамической памяти heap. Предупреждение: написание программ, которые управляют памятью кучи, довольно сложно. Этим отчасти объясняется большая популярность таких языков, как Java и Perl, которые управляют кучей автоматически. Эти языки берут на себя задачу, которая оказалась чрезвычайно сложной для программиста. В результате программы на Perl и Java программы работают немного медленнее, но в них гораздо меньше ошибок. (Подробное обсуждение кучи памяти см. на сайте http://cslibrary.stanford.edu/102/, Указатели и память).

C предоставляет доступ к функциям кучи через библиотечные функции, которые может вызывать любой код на C. Прототипы этих функций находятся в файле <stdlib.h>, поэтому любой код, который хочет их вызвать, должен #include этот заголовочный файл. Интерес представляют три функции…

void malloc(size_t size)* Запрос непрерывного блока памяти заданного размера в куче. malloc() возвращает указатель на блок в куче или NULL, если запрос не может быть удовлетворен. Тип size_t - это, по сути, unsignedlong, который указывает, какой размер блока хочет получить вызывающая сторона, измеряемый в байтах. Поскольку указатель блока, возвращаемый malloc(), является void* (т.е. он не заявляет о типе своего указателя), при сохранении указателя void* в указатель с обычным типом, вероятно, потребуется приведение.

void free(void block)* Зеркальное отражение malloc() - free принимает указатель на блок кучи, ранее выделенный malloc(), и возвращает этот блок в кучу для повторного использования. После выполнения free() клиент не должен обращаться к какой-либо части блока или предполагать, что блок является действительной памятью. Блок не должен освобождаться второй раз.

void realloc(void* block, size_t size)* Берет существующий блок кучи и пытается переместить его в блок кучи заданного размера, который может быть больше или меньше исходного размера блока. Возвращает указатель на новый блок или NULL, если перемещение было неудачным. Не забывайте отлавливать и проверять возвращаемое значение realloc() - это распространенная ошибка продолжать использовать старый указатель блока. Realloc() позаботится о перемещении байтов из старого блока в новый. Realloc() существует потому, что она может быть реализована с использованием низкоуровневых функций, которые делают ее более эффективной, чем код на языке C, который мог бы написать клиент.

Управление памятью

Вся память программы деаллоцируется автоматически при выходе из нее, поэтому использовать free() во время выполнения нужно только в том случае, если программе важно перерабатывать свою память во время работы - обычно потому, что она использует много памяти или работает в течение длительного времени. Указатель, передаваемый в free(), должен быть именно тем указателем, который был первоначально возвращен malloc() или realloc(), а не просто указателем куда-то в блок кучи.

Динамические массивы

Поскольку массивы - это просто смежные области байтов, вы можете выделять свои собственные массивы в куче с помощью malloc(). Следующий код выделяет два массива по 1000 int - один в стеке обычным "локальным" способом, а другой в куче с помощью malloc(). За исключением разных выделений, эти два способа синтаксически схожи.

```
int a[1000];
int *b;
b = (int*) malloc( sizeof(int) * 1000);
assert(b != NULL);          // проверить, что распределение прошло успешно
a[123] = 13;                // Просто используйте старую добрую [] для доступа к элементам
b[123] = 13;                // в обоих массивах.
free(b);
```

Хотя к обоим массивам можно обращаться с помощью [ ], правила их обслуживания совершенно разные.....

Преимущества нахождения в куче

- Размер (в данном случае 1000) может быть определен во время выполнения. Для массива типа "a" это не так.
- Массив будет существовать до тех пор, пока он не будет явно деаллоцирован вызовом free().
- Вы можете изменить размер массива по своему усмотрению во время выполнения с помощью функции realloc(). Следующее

изменяет размер массива на 2000. Realloc() позаботится о копировании старого массива элементы.

```
b = realloc(b, sizeof(int) * 2000);
assert(b != NULL);
```

Недостатки нахождения в куче

- Вы должны помнить о выделении массива, и вы должны сделать это правильно.
- Вы должны помнить о деаллокации массива ровно один раз, когда вы закончите работу с ним, и вы должны сделать это правильно.
- Два вышеупомянутых недостатка имеют один и тот же основной профиль:
  если вы ошибетесь с ними, ваш код все равно будет выглядеть правильно.
  Он хорошо компилируется. Он даже работает для небольших случаев,
  но для некоторых входных случаев он просто неожиданно падает, потому что
  случайная память перезаписывается в каком-то месте, как смайлик.
  Выявление такого рода ошибки "Разрушитель случайной памяти" может стать
  серьезным испытанием.

Динамические строки

Динамическое распределение массивов отлично работает при распределении строк в куче. Преимущество выделения строки в куче в том, что блок кучи может быть достаточно большим, чтобы хранить фактического количества символов в строке. Обычная техника локальных переменных, например char string[1000]; в большинстве случаев выделяет слишком много места, тратят его впустую. неиспользуемые байты, и при этом терпит неудачу, если строка становится больше фиксированного размера переменной.

```
#include <string.h>
/*
        Принимает на вход строку с и создает копию этой строки.
        в куче. Вызывающая сторона получает право собственности на новую строку
        и отвечает за его освобождение.
*/
char* MakeStringInHeap(const char* source) {
        char* newString;
        newString = (char*) malloc(strlen(source) + 1); // +1 для '\0'
        assert(newString != NULL);
        strcpy(newString, source);
        return(newString);
}
```

Раздел 7
Справочник по стандартным библиотекам

Приоритет и ассоциативность

function-call() [] -> .	Слева направо
! ~ ++ -- + - *(ptr deref) sizeof &(addr of) (all unary ops are the same)	Справа налево
* / % (the top tier arithmetic binary ops)	Слева направо
+ - (second tier arithmetic binary ops)	Слева направо
< <= > >=	Слева направо
== !=	Слева направо
in order: & ^ | && || (note that bitwise comes before boolean)	Слева направо
= and all its variants	Справа налево
, (comma) .	Слева направо

A combinations which never works right without parens: *structptr.field
You have to write it as (*structptr).field or structptr->field

Стандартные функции библиотеки

Многие базовые функции для ведения домашнего хозяйства доступны программе на языке Си в виде стандартных библиотечных функций. Чтобы вызвать их, программа должна #include соответствующий файл .h. Большинство компиляторов по умолчанию подключают код стандартной библиотеки. Функции, перечисленные в следующем разделе, являются наиболее часто используемыми, но существует множество других, которые не перечислены. здесь.

stdio.h	ввод и вывод файлов
ctype.h	тесты char
string.h	строковые операции
math.h	математические функции, такие как sin() и cos()
stdlib.h	полезные функции, такие как malloc() и rand()
assert.h	отладочный макрос assert()
stdarg.h	поддержка функций с переменным числом аргументов
setjmp.h	поддержка нелокальных скачков управления потоком
signal.h	поддержка сигналов об исключительных условиях
time.h	дата и время
limits.h, float.h	константы, определяющие значения диапазона типов, такие как INT_MAX

stdio.h

Stdio.h - это очень распространенный файл для #include - он включает функции для печати и чтения строк из файлов, а также для открытия и закрытия файлов в файловой системе.

FILE* fopen(const char* fname, const char* mode)

Открывает файл с именем в файловой системе и возвращает для него FILE*.
Mode = "r" чтение, "w" запись, "a" - добавление, при ошибке возвращается NULL. Стандартные файлы stdout, stdin, stderr автоматически открываются и закрываются системой.

int fclose(FILE* file)

Закрытие ранее открытого файла. При ошибке возвращает EOF. Операционная система закрывает все файлы программы при выходе из нее, но лучше сделать это заранее. Кроме того, обычно существует ограничение на количество файлов, которые программа может открывать одновременно.

int fgetc(FILE* in)

Считывает и возвращает следующий беззнаковый символ из файла или EOF, если файл был исчерпан. (подробно) Эта и другие файловые функции возвращают инты вместо символов, потому что константа EOF, которую они потенциально могут использовать, не является символом, а является интом. getc() - это альтернативная, более быстрая версия, реализованная в виде макроса, который может оценивать выражение FILE* более одного раза.

char* fgets(char* dest, int n, FILE* in)

Считывает следующую строку текста в строку, предоставленную вызывающей стороной. Считывает не более n-1 символов из файла, останавливаясь на первом символе '\n'. В любом случае строка будет '\0' завершается. Символ '\n' включается в строку. Возвращает NULL при EOF или ошибке.

int fputc(int ch, FILE* out)

Запись символа в файл в виде беззнакового символа. Возвращает ch или EOF при ошибке. putc() - альтернативная, более быстрая версия, реализованная в виде макроса, который может оценивать выражение FILE* более одного раза.

int ungetc(int ch, FILE* in)

Верните последний символ fgetc() в файл. Возвращает ch или EOF при ошибке.

int printf(const char* format_string, ...)

Выводит на стандартный вывод строку со значениями, которые могут быть в нее вставлены. Принимает переменное количество аргументов - сначала строку формата, а затем несколько подходящих аргументов. Строка формата содержит текст, смешанный с директивами %, которые отмечают значения, которые будут вставлены в вывод. %d = int, %Ld=long int, %s=string, %f=double, %c=char. Каждая директива % должна иметь соответствующий аргумент нужного типа после строки формата. Возвращает количество записанных символов или отрицательное значение при ошибке. Если директивы percent не соответствуют количеству и типу аргументов, printf() имеет тенденцию к аварийному завершению работы или иному неправильному действию во время выполнения. fprintf() - это вариант, который принимает дополнительный аргумент FILE*, указывающий файл для печати. Примеры…

```
printf("hello\n");                          печатает: hello
printf("hello %d там %d\n", 13, 1+1);       печатает: hello 13 там 2
printf("hello %c там %d %s\n", 'A', 42, "ok");  печатает: hello A там 42 ok
```

int scanf(const char* format, ...)

Противоположность printf() - считывает символы из стандартного ввода, пытаясь найти соответствие элементам строки формата. Каждая директива percent в строке формата должна иметь соответствующий указатель в списке аргументов, который scanf() использует для хранения найденных значений. scanf()пропускает пробельные символы при попытке прочитать каждую директиву percent. Возвращает количество успешно обработанных директив percent или EOF при ошибке. scanf(), как известно, чувствительна к ошибкам программистов. Если вызвать scanf() с любыми, кроме правильных, указателями после строки форматирования, то во время выполнения она, как правило, аварийно завершает работу или делает что-то не то. sscanf() - это вариант, который принимает дополнительную начальную строку, из которой происходит чтение. fscanf() - это вариант, который принимает дополнительный начальный FILE*, из которого происходит чтение. Пример…

```
int num;
char s1[1000];
char s2[1000];
scanf("hello %d %s %s", &num, s1, s2);
```

Ищет слово "hello", за которым следует число и два слова (все разделены пробелами). scanf() использует указатели &num, s1 и s2 для сохранения найденного в локальных переменных.

ctype.h

В файле ctype.h содержатся макросы для выполнения простых тестов и операций над символами

isalpha(ch)	// ch - это строчная или заглавная буква
islower(ch), isupper(ch)	// то же, что и выше, но для верхнего/нижнего регистра
isspace(ch)	// ch - символ пробела, такой как табуляция, пробел, новая строка и тд
isdigit(ch)	// цифра, например '0'...'9'
toupper(ch), tolower(ch)	// Возвращаем строчный или прописной вариант строки алфавитного // символа, в противном случае передайте его без изменений.

string.h

Ни одна из этих строковых подпрограмм не выделяет память и не проверяет, что переданная память имеет правильный размер. Вызывающая сторона несет ответственность за то, чтобы убедиться, что памяти "достаточно" для выполнения операции. Тип size_t - это беззнаковое целое число, достаточно широкое для размещения в памяти компьютера - скорее всего, беззнаковое long.

size_t strlen(const char* string)
Возвращает количество символов в строке языка C. EG strlen("abc")==3

char* strcpy(char* dest, const char* source)
Копирование символов из исходной строки в конечную.

size_t strlcpy(char* dest, const char* source,size_t dest_size)
Как strcpy(), но знает размер dest. При необходимости усекает. Используйте ее, чтобы избежать ошибок памяти и проблем с безопасностью переполнения буфера. Эта функция не такая стандартная, как strcpy(), но в большинстве систем она есть. Не используйте старую функцию strncpy() - ее трудно использовать правильно.

char *strcat(char* dest, const char* source)
Добавьте символы из исходной строки в конец строки назначения. (Существует нестандартный вариант strlcat(), который принимает размер dest в качестве третьего аргумента).

int strcmp(const char* a, const char* b)
Сравнивает две строки и возвращает int, который кодирует их порядок. нулевой:a==b, отрицательный:a<b, положительный:a>b. Распространенной ошибкой является представление результата strcmp() как как булеву истину, если строки равны, что, к сожалению, совершенно неверно.

char* strchr(const char* searchIn, char ch)
Поиск в заданной строке первого появления заданного символа. Возвращает указатель на символ или NULL, если он не найден.

char* strstr(const char* searchIn, const char* searchFor)
Аналогично функции strchr(), но поиск ведется не по одному символу, а по всей строке. Адрес поиск чувствителен к регистру.

void* memcpy(void* dest, const void* source, size_t n)
Копирует заданное количество байт из источника в место назначения. Источник и назначения не должны пересекаться. Это может быть реализовано в специализированном, но очень оптимизированным способом для конкретного компьютера.

void* memmove(void* dest, const void* source, size_t n)
Аналогично memcpy(), но позволяет перекрывать области. Вероятно, это работает немного медленнее, чем memcpy().

stdlib.h

int rand()

Возвращает псевдослучайное целое число в диапазоне 0...RAND_MAX (limits.h), которое находится на уровне не менее 32767.

void srand(unsigned int seed)

Последовательность случайных чисел, возвращаемых функцией rand(), изначально контролируется глобальной переменной "srand() устанавливает эту переменную, которая по умолчанию начинается со значения 1. Передайте выражение time(NULL) (time.h), чтобы установить значение seed, основанное на текущем времени. времени, чтобы гарантировать, что случайная последовательность будет отличаться от одного запуска к другому.

void* malloc(size_t size)

Выделяет блок кучи заданного размера в байтах. Возвращает указатель на блок или NULL в случае неудачи. Может потребоваться приведение для сохранения указателя void* в обычный типизированный указатель. [ред: см. раздел Выделение кучи выше для более подробного обсуждения malloc(), free() и realloc()].

void free(void* block)

Противоположность функции malloc(). Возвращает предыдущий блок malloc в систему для повторного использования

void* realloc(void* block, size_t size)

Изменение размера существующего блока кучи до нового размера. Занимается копированием байтов из старого блока в новый. Возвращает новый базовый адрес блока кучи. Это распространенная ошибка - забыть перехватить возвращаемое значение из realloc(). Возвращает NULL, если изменение размера не удалось выполнить.

void exit(int status)

Остановите и завершите работу программы и передайте операционной системе состояние int. Передайте 0, чтобы сигнализировать о нормальном завершении программы, ненулевое значение в противном случае.

void* bsearch(const void* key, const void*base, size_t len, size_t elem_size,<compare_function>)

Выполняет бинарный поиск в массиве элементов. Последним аргументом является функция, которая принимает указатели на два элемента для сравнения. Ее прототип должен иметь вид: int compare(const void* a, const void* b);, и она должна возвращать 0, -1 или 1, как это делает strcmp() делает. Возвращает указатель на найденный элемент, или NULL в противном случае. Обратите внимание, что strcmp() нельзя использовать непосредственно в качестве функции сравнения для bsearch() для массива char* поскольку strcmp() принимает аргументы char*, а bsearch() потребуется компаратор. который принимает указатели на элементы массива - char**.

void qsort(void* base, size_t len, size_t elem_size,<compare_function>)

Сортировка массива элементов. Принимает указатель на функцию, как и besearch().