

Константин А. Паньков

**Язык программирования С
для C++ разработчиков**

Версия 0.2
2025 г.

Содержание

Введение	1
1. Общие сведения о языке С	4
2. Первая программа на С	9
3. Процесс трансляции	12
4. Лексические элементы	16
5. Структура программы	23
6. Комментарии	26
7. Функции	28
8. Функция main	41
9. Типы данных	45
10. Переменные и константы	72
11. Чтение сложных объявлений	78
12. Операторы, выражения и инструкции	82
13. Препроцессор и макросы	103
14. Соглашения о вызовах	115
15. Опережающее объявление	118
16. Общие сведения о модулях	121
17. Класс хранения	123
18. Внутренняя и внешняя компоновка	126
19. Заголовочные файлы	134
20. Предкомпилированные заголовки	142
21. Управление памятью	146
22. Работа со строками	164
Инструменты разработки	185
Литература	193

Введение

Эта книга появилась в результате компиляции многочисленных заметок, которые автор накопил за более чем 20 лет работы разработчиком ПО.

Для кого эта книга?

Книга рассчитана в первую очередь на C++ разработчиков, которые по тем или иным причинам не знакомы на должном уровне с языком С.

Автор, помимо программирования на C++, активно вовлечен в процесс собеседований C++ разработчиков и часто сталкивается с тем, что в резюме указывают знания и опыт разработки на C/C++, хотя по факту C++ действительно могут хорошо знать и даже иметь многолетний опыт написания кода на нем, а вот с языком С дела обстоят очень плохо. К сожалению, среди таких разработчиков сильно развито заблуждение, что при хороших знаниях C++ разработчик знает, в том числе и С. Речь не только про джунов, но даже среди мидлов встречается такое.

Например, очень простой вопрос "на отсев самозванцев", который вызывает трудности у многих кандидатов: почему этот код собирается в g++, но падает с ошибками в gcc и как его исправить, чтобы собирался везде?

```
#include <stdio.h>

struct Point {
    int x;
    int y;
};

int main() {
    Point p1 = {10, 20};
    printf("Point: (%d, %d)\n", p1.x, p1.y);
    return 0;
}
```

Совет: указывайте в резюме знание С и C++ (если вы действительно знаете оба языка) через запятую, а не в виде C/C++. Это своего рода маркер и опытный ревьюер

обязательно поинтересуется знаете ли вы действительно язык С.

Очень важно понимать, что хоть С++ и возник на основе языка С, но это все же разные языки программирования. В приложении С стандарта языка С++ (ISO/IEC 14882:2024) есть даже целый раздел [diff.iso], посвященный совместимости С и С++. И на протяжении всей этой книги постоянно будет отмечаться совместимость этих языков, а также поддержка тех или иных возможностей.

Также эта книга рассчитана на начинающих свой путь в С++ (школьников старших классов, студентов колледжей и ВУЗов). Материал в книге подразумевает, что у читателя есть определенные знания основ информатики (компьютерных наук), в частности, архитектуры компьютера, операционных систем, азов программирования и т.д. Изучающим программирование с нуля эта книга вряд ли подойдет.

Нужно ли изучать С++ разработчикам языка С?

Путь освоения С++, вне всяких сомнений, лежит через изучение языка С. Именно С создает прочный фундамент, на котором сверху построены возможности С++, реализующие ООП и шаблонное метaprogramмирование.

С практической точки зрения знание С позволяет легко разбираться, внедрять и сопровождать огромное количество библиотек, написанных на С. В частности, критичные уязвимости в open-source коде, которые порой игнорируются авторами, вынуждают других разработчиков создавать свои исправления.

Почему эту книгу стоит прочитать?

Книга написана С++ программистом для программистов. В книге дан ряд ценных рекомендаций и советов, которые стоит взять на вооружение. Подобную книгу автор сам хотел бы прочитать в начале своей карьеры.

Разумеется, существует большое множество других хороших книг по языку С, список которых приведен в конце книги. И неплохо бы их все прочитать, так как в каждой найдется что-нибудь полезное.

И если вы все еще затрудняетесь с ответом на вопрос в вышеприведенном коде, то вот один из возможных вариантов его решения:

```
#include <stdio.h>

// В С++ можно опускать 'struct' при объявлении переменных, но в С – нет.
```

```
// typedef избавляет от этой необходимости
typedef struct {
    int x;
    int y;
} Point;

int main() {
    Point p1 = {10, 20};
    printf("Point: (%d, %d)\n", p1.x, p1.y);
    return 0;
}
```

Буду признателен за любую найденную ошибку в тексте или коде. Сообщения можно присыпать на мою почту: konstantin.pankov@fsight.ru

1. Общие сведения о языке С

Язык С (си) является компилируемым статически типизированным языком общего назначения. Это определение из Wikipedia и требует разъяснения терминологии.

Языки программирования классифицируют по способам выполнения кода:

- Компилируемые. Код преобразуется (компилируется) в машинный до выполнения (C, C++, Rust, Go и т.д.);
- Интерпретируемые. Код выполняется построчно интерпретатором без предварительного преобразования (Python, JavaScript, Ruby, PHP и т.д.);
- Гибридные. Код компилируется во время выполнения (так называемая JustInTime или JIT-компиляция: используется в Java(JVM), C#(.NET), JavaScript (V8, Node.js), LuaJIT и т.д.);
- Компилируемые в байт-код. Код предварительно преобразуется в промежуточный байт-код, который потом выполняется виртуальной машиной (Java(JVM));
- и другие, менее распространенные.

Язык С использует компилятор для преобразования кода в машинный (двоичный код, который может исполнять процессор, поэтому его еще иногда называют исполняемым кодом). Компиляция выполняется раздельно под разные архитектуры и разные операционные системы. Скомпилировать один раз программу под все виды устройств и систем просто невозможно, да и необходимости в этом нет.

Существует два вида типизации: статическая и динамическая. Разница заключается во времени проверки типов данных:

- Статическая типизация проверяет типы во время компиляции;
- Динамическая типизация проверяет типы во время исполнения.

Благодаря статической типизации достигается высокое быстродействие (во время исполнения нет затрат на проверку типов) и безопасность (нет риска ошибок во время исполнения). Типизация в С считается слабой (в отличие от строгой в C++), так как позволяет некоторые неявные преобразования и отсутствие строгой проверки типов.

Языки общего назначения (**General-Purpose Languages, GPL**) занимаются решением широкого круга задач в самых разных областях.

Специализированные языки (**Domain-Specific Languages**, **DSL**) решают узкоспециализированные задачи (SQL, HTML, PHP и т.п.)

Иногда язык С называют низкоуровневым языком, хотя формально таковым он не является. Это происходит под влиянием других более высокоуровневых языков, по сравнению с которыми язык С выглядит действительно простым и элементарным. Под низкоуровневым программированием подразумевается ассемблер и машинный код. Низкоуровневое программирование позволяет создавать более эффективный код за счет оптимизаций, использующих аппаратные особенности. Однако, существующие механизмы оптимизации в современных компиляторах способны создавать довольно оптимальный код, порой даже более производительный, чем тот же код, написанный программистом на ассемблере.

Историческая справка

В 1969 году в подразделении Bell Labs компании AT&T (США) небольшая команда программистов, среди которых были Кен Томпсон (Ken Thompson), Деннис Ритчи (Dennis Ritchie) и Дуг Макилрой (Douglas McIlroy), начали создавать операционную систему UNIX на базе мини-компьютера PDP-7 от компании DEC. Сперва они разрабатывали ее на ассемблере, но в какой-то момент возникла потребность в использовании переносимого высокоуровневого языка.

Кен Томпсон (Ken Thompson [\[1\]](#)) при участии Дена Ритчи разработал в 1969 язык В (упрощенный потомок языка BCPL, 1967). Дэн Ритчи, в свою очередь, при поддержке Кена Томпсона и на базе его языка В стал создавать новый язык, который получил название С.

Родословную языка С можно описать следующей последовательностью:

ALGOL - CPL - BCPL - B - C

12 июня 1972 года в Bell Labs завершили работы над созданием Unix Second Edition с первым компилятором языка С (cc). В 1973 году ядро и большинство утилит были переписаны на языке С.

В 1973 году Алан Снайдер (Alan Snyder), работая в Bell Labs, написал компилятор portable С, который потом описал в своей магистерской диссертации. Суть этого компилятора, и это отражено в его названии, заключалась в его способности адаптировать код к различным архитектурам. И это послужило в дальнейшем одной из причин популярности языка.

После начала официального распространения **UNIX V5** вне стен Bell Labs (1975 год), язык получил огромную популярность, сперва в высших учебных заведениях США, а затем в коммерческих организациях и по всему миру.

В 1978 году выходит в свет книга Брайна Кернигана и Денниса Ритчи "Язык программирования Си" (синтаксис языка из этой книги впоследствии назовут K&R C), которая фактически стала стандартом языка до выхода C89.

К 1979 году Стивеном Джонсоном (Stephen C. Johnson) из Bell Labs был написан компилятор pcc (portable c compiler) на основе разработок Алана Снайдера. Этот компилятор был распространен в составе 7 редакции UNIX (Version 7 Unix). При этом компилятор в отличии от классического cc был разделен на front-end (построение промежуточного представления) и back-end (кодогенерация).

С распространением UNIX, стали появляться сторонние коммерческие компиляторы (за основу брался компилятор pcc), в том числе для других платформ и операционных систем, среди которых BSD C (pcc был основным компилятором в BSD UNIX, а команда из Беркли активно дорабатывала и улучшала его), IBM C, HP C, DEC C и другие.

В 1982 году Компания Lifeboat Associates выпускает первую версию **Lattice C** - первый компилятор C для IBM PC, которую позже в 1983 году у Lifeboat Associates лицензирует компания Microsoft и будет распространять под своим брендом.

Создатели компиляторов добавляли свои языковые расширения, в результате чего появилось множество диалектов языка C и это становилось большой проблемой - страдала переносимость исходного кода. Поэтому в 1983 под эгидой Американского национального института стандартов (ANSI) был создан комитет стандартизации языка C.

В 1985 году Microsoft с нуля написали свой компилятор C и выпустили его под версией 3.0 (отказавшись от компилятора Lattice). Именно на этом компиляторе будут написаны первые версии Windows.

В 1986 году компания THINK Technologies выпустила компилятор и среду разработки LightSpeed C (автор Michael Kahl), позднее переименованную в THINK C. Фактически THINK C стал стандартной средой разработки на C на Mac. Даже когда в MPW появился язык C, многие разработчики все равно предпочитали использовать THINK C (в основном из-за более дешевой цены).

В 1987 году компания Borland выпускает компилятор Turbo C 1.0, который ранее выкупила у его разработчика - Bob Jervis (компилятор назывался Wizard C).

23 мая 1987 года появился первый компилятор Си проекта GNU, gcc 1.0 (разработчики Leonard H. Tower Jr. и # Richard Stallman).

В этом же 1987 году вышел компилятор Microsoft QuickC 1.0 (облегченная версия MS C без поддержки некоторых оптимизаций и позиционировалась как конкурент Borland Turbo C)

В 1989 году, после долгих 6 лет работы комитета стандартизации, появился стандарт для языка Си, который через год уже утвердили в ISO. Соответственно С89 называют еще ANSI C. А С90 - это просто международное название. По сути это один и тот же стандарт.

Были несколько исправлений в стандарт, вносились даже новые фичи, такое тоже бывает, но название при этом не менялось. Пока в 1999 году не приняли новый стандарт С99, в который вошло множество изменений. Многие проекты в настоящее время придерживаются именно этого стандарта.

Следующий стандарт С11 вышел в 2011 году, практически вместе с C++11, но в отличие от C++ не получил такого же большого обновления. Надо сказать, что комитеты по стандартизации этих языков как-то плохо между собой взаимодействуют, а разработчики компиляторов Си достаточно консервативны и не очень то приветствуют масштабные изменения в языке.

В 2018 году выходит следующий стандарт С17 (иногда обозначается как С18), который не внес никаких новых возможностей. Были только уточнения и исправления.

И, наконец, в 2024 году вышел стандарт С23.

Актуален ли язык С в современной разработке ПО?

Достаточно взглянуть на индекс TIOBE: <https://www.tiobe.com/tiobe-index/>

Язык стабильно занимает верхние строчки рейтинга. Несмотря на различные нападки со стороны американских и европейских правительственные структур на языки С и C++ в плане их безопасности, пока массового перехода на другие языки ожидать не стоит.

На языке С написаны операционные системы, сотни библиотек, фреймворков, и прочего кода, используемого везде и всюду:

- Linux, FreeBSD, MacOS, Windows написаны на С;
- OpenGL, OpenSSL, GTK, curl, zlib, libxml2, SQLite написаны на С;
- Python (имеется ввиду эталонный CPython), Ruby, PHP, Lua, JavaScript написаны на С;
- Git, Nginx, Redis, PostgreSQL, MySQL написаны на С;

- И многое другое.

На чем был написан Doom? С первого раза угадаете?

1. Кен Томпсон - американский программист, один из ключевых разработчиков UNIX, а также языка программирования В (предшественника языка С). Автор редактора ed (основной текстовый редактор в UNIX). Один из разработчиков языка Go. ↪

2. Первая программа на С

В 1978 году вышла книга "Язык программирования С" ("The C Programming Language"), написанная **Брайаном Керниганом** (Brian Kernighan^[1]) и **Деннисом Ритчи** (Dennis Ritchie^[2]), которую часто кратко обозначают K&R. Это была первая книга по языку С и до появления стандарта ANSI C (C89) являлась де-факто стандартом языка.

Именно в этой книге был приведен пример программы, выводящей на экран сообщение "Hello, world". Эта знаменитая программа стала традиционным первым примером многих других книг, в том числе и по другим языкам программирования.

Вот она:

```
main()
{
    printf("hello, world\n");
}
```



Из чего состоит и что делает программа Hello world?

- Программа состоит из одной функции `main`. Любая программа на си начинается с этой функции. Это главная ее точка входа;
- Круглые скобки содержат параметры функции и в данном случае параметров нет;
- Фигурные скобки обозначают начало и конец тела функции;
- `printf` - это функция стандартной библиотеки си, которая выводит строку на стандартный вывод, обычно это экран. На самом деле это очень сложная функция, которая выполняет так называемый форматированный вывод;
- `"hello, world\n"` - выводимая строка, записывается в кавычках. `\n` является символом перевода строки;
- Точка с запятой завершает инструкцию (вызов функции `printf`).

Читатели, знакомые уже с языками С или С++ могут возмутиться, что в этом коде нет:

- возвращаемого типа `int` в функции `main`
- `return 0;`
- `#include <stdio.h>`

Тем не менее, этот код компилировался и компилируется до сих пор. Давайте разберемся почему:

- Если не было предварительного объявления функции, компилятор считает по умолчанию, что возвращаемый тип является `int` (хотя и выводит то же предупреждение об этом);
- `return 0;` не является обязательным для функции `main` по стандарту C99 и по умолчанию принимается, что если в функции `main` отсутствует `return`, то программа возвращает в систему код 0;
- Компилятор для ряда функций (например, для часто используемой `printf`) автоматически подставляют требуемые заголовочные файлы, а на этапе линковки автоматически подключается стандартная библиотека, в которой есть функция `printf`. Начиная со стандарта C89, теперь явное включение заголовочных файлов является обязательным (хотя тот же gcc продолжает компилировать код, но выдает при этом предупреждение, см. `-Wimplicit-function-declaration`). И во втором издании книги K&R в программе появилась директива препроцессора (`#include <stdio.h>`).

```
#include <stdio.h>

main()
{
    printf("hello, world\n");
}
```

На современном языке си эту программу рекомендуется записывать в следующем виде:

```
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
}
```

`int` явно сообщает тип возвращаемого значения. А `void` указывает, что функция не принимает никаких аргументов.

Чтобы запустить эту программу, необходимо ее сохранить в файле, например в `hello.c` и скомпилировать соответствующей командой.

для unix:

```
cc hello.c
```

для linux:

```
gcc hello.c
```

для windows:

```
cl hello.c
```

В результате под UNIX и Linux будет создан исполняемый файл с именем по умолчанию a.out, который следует запустить командой:

```
./a.out
```

Запуск этой программы приведет к выводу на экран надписи:

```
hello, world
```

Под Windows будет создан файл `hello.exe`, запуск которого приведет к тому же результату.

1. Брайан Керниган - канадский программист, ученый. Работая в Bell Labs, участвовал в разработке операционной системы UNIX (он же придумал название UNIX). В соавторстве с Деном Ритчи написал книгу **Язык программирования С** ↪
2. Деннис Ритчи - американский программист. Автор языка программирования С. Работая в Bell Labs совместно с Кеном Томпсоном разработал операционную систему UNIX. ↪

3. Процесс трансляции

В классической литературе процесс трансляции программы на С в машинный код состоит из четырех основных шагов:

1. Препроцессинг (Preprocessing)
2. Компиляция (Compilation)
3. Ассемблирование (Assembly)
4. Компоновка (Linking)

При этом каждый транслируемый файл с расширением .c называется модулем или единицей трансляции (translation unit).

На выходе получают или исполняемый файл (.exe в Windows, в Linux расширения не принято назначать, но по умолчанию gcc генерирует a.out), или динамическую библиотеку (.dll в Windows, .so в Linux), или объектные файлы (.obj в Windows, .o в Linux). Объектные файлы, в свою очередь собираются в статические библиотеки (.lib в Windows, lib*.a в Linux).

Интересный факт: название выходного файла a.out не меняется уже на протяжении полувека, но при этом утратило свой первоначальный смысл - assembler output. Особенности первых версий компилятора cc:

- внешний препроцессор cpp (и это вовсе не c++ компилятор);
- cc не занимался кодогенерацией, а создавал файлы с ассемблерным кодом;
- ассемблерный код подавался на вход обычному ассемблеру as, который и генерировал .o файлы.

В современных компиляторах ассемблирование не выполняется, а кодогенерация выполняется с использованием промежуточного представления (intermediate representation), однако для совместимости в компиляторах сохранена возможность генерации ассемблерного кода.

Препроцессинг

Препроцессинг - это обработка исходного текста до ее синтаксического анализа. На этом этапе выполняется:

- файл считывается в память и разбивается на строки

- объединение строк разбитых символом `\` (пробел при этом не добавляется);
- замена комментариев пробельным символом;
- токенизация (файл разбивается на препроцессинговые токены);
- исполняются директивы препроцессора (include, pragma, макросы, условная компиляция).

В этом месте можно сбросить промежуточный результат в файл:

```
gcc -E main.c -o main.i # только препроцессинг
```

- заменяются escape-последовательности;
- соединяются строковые литералы;
- удаляются лишние пробелы;
- препроцессинговые токены становятся токенами.

Тут важно понимать, что препроцессор в основном выполняет текстовые подстановки и не занимается проверкой кода на валидность. Препроцессинговые токены могут быть чем угодно, любым текстом. После препроцессинга полученные токены должны принадлежать к одному из пяти типов:

- ключевые слова;
- идентификаторы;
- константы;
- операторы;
- разделители.

Все остальное будет считаться ошибкой на этапе компиляции.

Компиляция

Компиляция - это преобразование "чистого" С-кода в ассемблерный (платформозависимый) код. При этом выполняется:

- синтаксический анализ;
- семантический анализ.

Получение ассемблерного кода:

```
gcc -S main.i -o main.s # генерация ассемблера
```

Ассемблирование

Ассемблирование - это преобразование ассемблерного кода в объектный код (машинные инструкции). В результате получается объектный файл (`.o` или `.obj`).

```
gcc -c main.s -o main.o # создание объектного файла
```

Компоновка

Компоновка - это объединение всех объектных файлов (`*.o`) или в исполняемый файл (`a.out` , `main.exe` и т. д.) или библиотеку (`*.a` , `*.so` / `*.dll`). При этом выполняется связывание имен (разрешение ссылок на функции и переменные).

```
gcc main.o -o program # линковка в исполняемый файл
```

Как выше было сказано, все современные компиляторы (gcc, clang, MSVC) пропускают этап ассемблирования и процесс (с некоторыми различиями) выглядит следующим образом:

- Трансляция;
 - лексический анализ (лексер), включающий препроцессинг;
 - синтаксический анализ (парсер);
- Семантический анализ (построение AST - abstract syntax tree);
- Построение IR (intermediate representation);
- Оптимизации;
- Кодогенерация;
- Компоновка;

В результате трансляции, как уже было выше сказано, получаются либо исполняемые файлы, либо статические библиотеки, либо динамические библиотеки. Драйверы и прочие файлы мы рассматривать не будем.

Статическая библиотека - это архив, содержащий объектные файлы. Статические библиотеки используются при компоновке.

Чтобы создать статическую библиотеку, нужно подготовить объектные файлы:

```
g++ -O2 -c code.cpp # с - создать объектный файл и не линковать
```

Чтобы собрать библиотеку, нужно воспользоваться специальным архиватором, входящим в состав binutils:

```
ar rcs libcode.a code.o
```

где,

r - вставка в архив с заменой,

c - создать новый архив,

s - запись индекса

Сборка со статической библиотекой выполняется следующим образом:

```
g++ -O2 main.cpp -L. -lcode
```

Важно! -lcode подразумевает, что будет найден файл библиотеки с именем libcode.a.

Префикс lib и расширение .a подставится автоматически и указывать их не нужно.

Динамическая библиотека является готовым самостоятельным файлом, загружаемым исполняемым файлом в рантайме.

Нужно также сперва подготовить объектные файлы:

```
g++ -c -fPIC code.cpp # PIC = position independent code
```

Сборка выполняется с параметром -shared:

```
g++ -shared code.o -o libcode.so
```

Сборка с динамической библиотекой:

```
g++ -O2 main.cpp -L. -lcode -Wl,-rpath,.
```

Важно передать линкеру параметр rpath, иначе при загрузке программы получим ошибку:

```
./main: error while loading shared libraries: libcode.so: cannot open shared
object file: No such file or directory
```

4. Лексические элементы

Исходный код после препроцессора представляет собой очищенный текст, который разбивается лексером на следующие токены:

1. идентификаторы
2. ключевые слова
3. литералы
4. операторы
5. разделители

Идентификаторы

Идентификатор - это последовательность символов, используемая для именования переменных, функций, новых типов данных и макросов препроцессора.

Идентификатор может содержать буквы, цифры и символ подчеркивания, при этом первый символ не может быть цифрой. Идентификаторы регистрозависимы, то есть foo и Foo - это разные идентификаторы.

Обратите внимание, что **идентификаторы, начинающиеся с `_` и заглавной буквы или содержащие два подчеркивания подряд, зарезервированы для стандартной библиотеки и компилятора.**

Например, `_MAX`, `--func--` — такие имена могут конфликтовать с системными определениями.

Идентификаторы, начинающиеся с `_` и следующей строчной буквы, обычно безопасны для пользовательского кода.

```
int _count; // normally
```

Глобальные идентификаторы, начинающиеся с `_`, могут конфликтовать с системными именами. Поэтому лучше избегать `_` в начале имен в глобальной области видимости.

Запрещается использовать в качестве идентификаторов ключевые слова.

```
int i = 0; // Ok
double long = 3.14; // Ошибка, long это тип, его нельзя использовать в
качестве идентификатора.
```

Идентификаторы не сохраняются в итоговой скомпилированной программе, поэтому не нужно пытаться их "шифровать", чтобы запутать хакеров.

Ключевые слова

Ключевое слово - это специальный идентификатор, зарезервированный для нужд языка. Ключевые слова запрещается использовать в других целях (как имена переменных, функций и т.д.)

Список ключевых слов C99:

- `auto` - Локальная переменная (устарело, по умолчанию).
- `break` - Выход из цикла или `switch`.
- `case` - Вариант в `switch`.
- `char` - Символьный тип (1 байт).
- `const` - Константное значение (неизменяемое).
- `continue` - Переход к следующей итерации цикла.
- `default` - Вариант по умолчанию в `switch`.
- `do` - Начало цикла `do-while`.
- `double` - Число с плавающей точкой двойной точности.
- `else` - Альтернативная ветка `if`.
- `enum` - Перечисляемый тип.
- `extern` - Объявление внешней переменной/функции.
- `float` - Число с плавающей точкой (одинарная точность).
- `for` - Цикл с инициализацией, условием и инкрементом.
- `goto` - Безусловный переход к метке.
- `if` - Условное выполнение кода.
- `inline` - Подсказка компилятору встроить функцию.
- `int` - Целочисленный тип.
- `long` - Длинный целочисленный тип.
- `register` - Подсказка хранить переменную в регистре (устарело).
- `restrict` - Указатель без пересечений (оптимизация).
- `return` - Возврат значения из функции.
- `short` - Короткое целое число.
- `signed` - Знаковый целочисленный тип.

`sizeof` - Размер типа/объекта в байтах.

`static` - Статическая переменная/функция (видимость/память).

`struct` - Составной тип данных (структурата).

`switch` - Множественное ветвление.

`typedef` - Создание псевдонима типа.

`union` - Структура с перекрывающимися полями.

`unsigned` - Беззнаковый целочисленный тип.

`void` - Отсутствие типа (пустота).

`volatile` - Переменная может измениться извне.

`while` - Цикл с условием.

`_Bool` - Логический тип (true/false).

`_Complex` - Комплексное число.

`_Imaginary` - Мнимая часть числа (редко используется).

Литералы

Литералом (literal) называют неизменяемое значение определенного типа данных, записанное непосредственно в коде программы. Другими словами это неименованная константа (nameless constant). Литерал не является переменной, так как у переменных есть имя.

Основные типы литералов в С:

1. Целочисленные литералы

Могут быть представлены в десятичном, восьмеричном (с префиксом 0), шестнадцатиричном (с префиксом 0x) или в двоичном представлении (с префиксом 0b, начиная с C23):

```
int a = 42;      // десятичное представление
int b = 052;     // восьмиричное (42 в десятичном)
int c = 0x2A;    // шестнадцатиричное (42 в десятичном)
int d = 0b1010;  // двоичное (C23)
```

Могут иметь суффиксы:

- u или U означает беззнаковое целое
- l или L означает тип long
- ll или LL означает тип long long (C99)
- z означает тип size_t (C23)

Суффиксы можно комбинировать и использовать в любых представлениях целых чисел.
Например:

```
unsigned int a = 42U;
unsigned long long b = 042ull;
unsigned long c = 0x2A1U;           // Допустимо, но 0x2A1l будет более привычным
unsigned int d = 0b1010U;
long long e = 0b1111LL;
```

Рекомендуется использовать суффиксы в нижнем регистре, однако l (long) может выглядеть как 1 в некоторых моноширинных шрифтах.

2. Вещественные (с плавающей точкой) литералы

Записываются с десятичной точкой (неважно какая используется локаль, в языке С используется всегда точка) или в экспоненциальной форме.

Экспоненциальная (или научная) форма записи вещественных чисел позволяет компактно представлять очень большие или очень маленькие числа, используя степень числа 10.

Синтаксис:

```
[цифры] [.цифры] [e | E] [±]показатель_степени
```

Где

- e или E — обозначает экспоненту ($\times 10^n$).
- ± — необязательный знак степени (+ можно опустить).

Примеры:

```
double a = 6.022e23; // 6.022 × 1023 (число Авогадро)
double b = 1.602e-19; // 1.602 × 10-19 (заряд электрона)
double c = -3.4E+5; // -3.4 × 105
```

По умолчанию вещественный литерал имеет тип double. Суффикс f указывает на тип float. Суффикс l указывает на тип long double.

3. Символьные литералы

- Заключаются в одинарные кавычки (' ').
- Могут быть обычными символами, escape-последовательностями или ASCII-кодами

```
char a = 'A';
char nl = '\n'; // символ новой строки
char ax = '\x41'; // 'A' в шестнадцатеричном формате
```

4. Строковые литералы

- Заключаются в двойные кавычки (" ").
- В конце строки автоматически добавляется нулевой символ (\0).

```
char message[] = "hello, world"; // компилятор создаст строку "hello, world\0"
```

5. Нулевой указатель

В С существует специальный указательный литерал, указывающий на нуль - NULL

```
int* p = NULL;
```

6. Логические литералы (C99)

Введены с типом _Bool и макросами true / false из <stdbool.h>

```
#include <stdbool.h>

bool is_valid = false;
```

Тип bool стал встроенным (нет необходимости включать заголовочный файл stdbool.h) начиная с C23.

7. Составные литералы (C99, не поддерживаются в C++)

Представляют собой способ создания **анонимных (безымянных) объектов** (массивов, структур или объединений) прямо в коде, без явного объявления переменной. Они были добавлены в стандарте **C99** и позволяют инициализировать сложные типы данных "на лету".

Синтаксис:

(тип) { инициализатор }

Где

- **тип** — это тип создаваемого объекта (например, `int[3]`, `struct Point`).
- **инициализатор** — список значений в фигурных скобках `{ }`, как при обычной инициализации.

Примеры:

```
struct Point {  
    int x;  
    int y;  
};  
  
// Создание и передача структуры в функцию  
print_point((struct Point){10, 20});  
  
int *arr = (int[]){1, 2, 3}; // анонимный массив из 3 элементов
```

Плохим тоном считается использование в коде магических чисел.

Магические числа (magic numbers) - это литералы, которые встречаются в коде без пояснения их смысла. Они делают программу менее понятной, потому что их значение неочевидно без дополнительных комментариев или документации.

```
if (status == 42) { // что означает 42?  
    printf("Success!\n");  
}
```

Вместо подобных значений в коде следует использовать самодокументируемые средства:

- Макросы

```
#define STATUS_SUCCESS 42
```

- перечисления

```
enum StatusCodes {
    STATUS_SUCCESS = 0,
    STATUS_ERROR = -1,
    STATUS_TIMEOUT = -2
};
```

- константные переменные

```
const int kSuccessStatus = 42;
```

Операторы

Оператор - это специальный токен, который выполняет операцию, такую как сложение или вычитание, с одним, двумя или тремя operandами.

Разделители

Разделители разделяют токены. При этом пробел тоже является разделителем, но не является токеном. Остальные разделители - это односимвольные токены:

```
( ) [ ] { } ; , . :
```

Под пробелом понимается не только сам пробел (Space, ASCII код 32), но и табуляция (Horizontal Tab, код 9), символ новой строки (Line Feed, 10), вертикальная табуляция (Vertical Tabulation, 11) и символ ввода формы (Form Feed, 12).

Пробелы игнорируются, за исключением использования в строках и символьных константах. Они служат для разделения токенов, а в ряде случаев они необязательны, например, для разделения операторов и operandов.

5. Структура программы

Программа на С состоит из **функций и переменных**. Также могут использоваться директивы препроцессора и комментарии.

Любая программа на С обязана иметь функцию `main`. С этой функции начинается и заканчивается выполнение программы.

```
/* минимальная программа, которая ничего не делает,
но ее можно скомпилировать */
int main(void) {}
```

Функция `main` вызывает другие функции, если они есть, а те, в свою очередь, тоже могут вызывать функции.

Чтобы функция могла вызвать другую функцию, необходимо до точки вызова либо объявить прототип (создать объявление) вызываемой функции, либо иметь определение этой функции, иначе компилятор выдаст ошибку. То же самое касается переменных - до первого их использования, они должны быть объявлены.

Объявление (declaration) - это инструкция, которая сообщает компилятору о существовании сущности (функции, переменной или константы).

Определение (definition) - это инстанцирование или реализация сущности.

```
// Определение глобальной переменной без инициализации
тип_переменной имя_переменной;
// Объявление функции (прототип)
тип_возвращаемого_значения имя_функции(параметры);

// Определение функции
тип_возвращаемого_значения имя_функции(параметры) {
    // Тело функции
    тип_переменной имя_переменной; // Определение локальной переменной без
инициализации
    return значение; // (если тип не void)
}
```

Определение может быть только одно в пределах одной программы. Это требование называется **One Definition Rule** (ODR) и относится любым именам (функции, глобальные переменные, структуры, объединения, перечисления).

ODR существует для предотвращения конфликтов при компоновке программы. Если одна и та же сущность (например, функция или глобальная переменная) определена несколько раз, компоновщик (linker) не сможет однозначно выбрать правильную версию, что приведёт к ошибке.

Таким образом, программа на С - это дерево вызовов функций.

Функции позволяют декомпозировать программу на небольшие функциональные части.

Каждая функция имеет блок, заключенный в фигурные скобки (тело функции), содержащий объявления переменных и инструкции (statements). Практически каждая инструкция заканчивается символом ;

Переменные при этом могут быть как локальные (видимые только внутри функций), так и глобальные.

Комментарии помогают задокументировать неочевидные нюансы работы, оставить подсказки другим программистам, которые будут сопровождать код.

Директивы препроцессора в основном занимаются подстановкой текста до компиляции кода.

```
/*
 Пример, поясняющий структуру программы на языке С
*/
#include <stdio.h> // директива препроцессора, вставляющая текст из файла
stdio.h

// Объявление функции
unsigned long long fibonacci(int n);

// Главная функция программы – main
int main() {
    int n;
    printf("Введите номер числа Фибоначчи: ");
    scanf("%d", &n);
```

```
if (n < 0) {
    printf("Номер должен быть неотрицательным!\n");
} else {
    printf("F(%d) = %llu\n", n, fibonacci(n));
}

return 0;
}

// Определение функции, вычисляющей число Фибоначчи
unsigned long long fibonacci(int n) {
    if (n <= 1) return n;

    unsigned long long a = 0, b = 1, c;
    for (int i = 2; i <= n; i++) {
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}
```

6. Комментарии

Изначально в языке С поддерживались только многострочные комментарии (начинаются с `/*` и заканчиваются `*/`):

```
int a = 42; /* Это комментарий */  
/*  
И это тоже комментарий  
*/
```

Компилятор удаляет (или заменяет на пробелы) комментарии на этапе трансляции исходного кода. Таким образом, в итоговую программу комментарии никогда не попадают, поэтому можно писать в них любой текст.

С появлением в начале 1990-х компиляторов C++ (в частности, GCC, Borland C/C++, Microsoft C/C++), которые, в том числе компилировали код на языке С, появилась поддержка односторонних комментариев (для этого требовались специальные параметры компилятора, например `-std=gnu89`). Под влиянием этого в C99 была включена поддержка односторонних комментариев.

```
int b = 42; // Это односторонний комментарий
```

Односторонний комментарий игнорируется компилятором от `//` и до окончания строки (до символа `\n`)

Для чего нужны комментарии?

Широко известно утверждение "хороший код должен быть самодокументирующимся и не требовать комментариев". Самодокументирующийся означает, что за счет использования понятных имен переменных и функций, код становится легко читаемым и понятным.

Эдсгер Дейкстра (Edsger Dijkstra^[1]) писал:

"Комментарии — это 'шум', мешающий понимать код."

Однако, для написания комментариев все же существует ряд причин:

- Пояснение сложной логики. С другой стороны, если код слишком сложный, его необходимо попытаться упростить, или еще говорят, провести рефакторинг.
- Иногда нет времени (или желания) на написание хорошего кода, поэтому оставляют комментарии типа **TODO** и **FIXME** для исправления в будущем, но это плохой подход.
- Документирование кода. При использовании специальных комментариев, например, в стиле Doxygen, возможна автоматическая генерация документации из исходных кодов.
- Для объяснения общих идей реализации, что облегчает задачу поддержки кода в будущем (и, возможно, силами других разработчиков).
- Для указания информации о правах, авторстве, лицензиях и т.п. (обычно в начале файла).

При написании комментариев избегайте избыточных комментариев и старайтесь отвечать не на вопрос "Что/как здесь написано", а на вопрос "Зачем/Почему это написано", то есть почему решение именно такое.

1. Эдсгер Дейкстра - нидерландский ученый, программист, оказавший сильное влияние на развитие информационных технологий и программирование в частности, написал компилятор языка ALGOL 60. Известен своей критикой оператора GOTO и продвижения идеи структурного программирования. Его именем назван алгоритм нахождения кратчайшего пути в графах. ↵

7. ФУНКЦИИ

Функция (function) - это подпрограмма (самостоятельный блок кода, который выполняет определенную задачу), которая может быть вызвана из другой части программы (вызывающей функции) по ее имени.

Функции позволяют решить ряд задач:

- упрощение кода - разбиение сложных задач на подзадачи;
- переиспользование кода - сокращение дублирующего кода;
- улучшение читаемости - хорошо написанные функции, благодаря своим именам и параметрам, могут "сообщать", что они делают, без необходимости вчитываться и вникать в суть своего кода. В результате программу легче сопровождать (исследовать, дорабатывать и исправлять ошибки).

Синтаксис функции

```
тип_возвращаемого_значения имя_функции(параметры) {  
    // Тело функции  
    return значение; // (если тип не void)  
}
```

Функция связывает тело функции, заключенное в фигурных скобках с идентификатором (именем функции).

Функция состоит из:

1. типа возвращаемого значения. Если функция не возвращает значения (аналог процедуры в Pascal), используется тип `void` ;
2. имени функции. Обязано быть уникальным внутри программы;
3. параметров функции. Указывают какие переменные принимает функция. Параметры могут отсутствовать;
4. тела функции. Это блок инструкций, заключенных в фигурные скобки. Если функция возвращает значение, в теле функции обязан быть оператор `return`.

В программе `Hello world` ранее использовались две функции - `main` и `printf`.

Вот еще примеры:

```
int          // возвращается результат типа int
add(int a, int b) { // имя функции add, используются два параметра типа int
    return a + b; // return возвращает результат выражения a + b
}
```

```
void log(char* str) { // функция ничего не возвращает
    printf(str);
}
```

Обратите внимание, что необязательно записывать сигнатуру функции в одну строку, но нужно придерживаться единого стиля, определенного для проекта, в команде разработки или в целом в компании.

Сигнтура функции содержит имя, параметры и возвращаемое значение. Однако в С компилятор различает функции только по имени, поэтому, например, перегрузки функций в Си нет (в отличие от C++).

Параметры функции

Чтобы вызвать функцию с параметрами, необходимо передать в нее аргументы:

```
int a = 2;
int b = 4;
int c = add(a, b);
```

Аргументы функции и параметры функции — это связанные, но разные понятия в программировании.

Параметры - это **переменные**, указанные в **определении функции**. Они представляют собой просто имена, которые будут заменены конкретными значениями (аргументами) при вызове функции.

Аргументы - это **конкретные значения**, которые передаются в функцию **при её вызове**.

Возврат из функции

Результат из функции может возвращаться двумя способами: через оператор `return` (по значению или по указателю) или через параметры функции (по указателю)

Возврат через оператор return (по значению или по указателю)

Возврат по значению (by value) означает, что функция создает копию результата (фактически помещает ее в специальный регистр или на стек) и передает ее вызывающему коду (вызывающая функция копирует ее в свою локальную переменную). Это стандартный способ для простых типов (int, char, float и т.д.).

```
int sum(int a, int b) {
    return a + b; // Возвращает сумму a и b
}
```

Но что возвращать в случае обнаружения какой-либо ошибки?

Стандартным способом обработки ошибок в UNIX-подобных системах является использование глобальной переменной `errno`, которая определена в заголовочном файле `<errno.h>` и хранит значение последней ошибки. Многие функции С (особенно работающие с файлами, памятью, процессами) не возвращают явный код ошибки, а вместо этого устанавливают `errno` при неудаче.

К сожалению, при таком подходе программисты часто забывают проверять значение в `errno` и это чревато серьезными проблемами. Ярким примером может служить функция `strtol`, которая конвертирует строку в числовое значение.

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

int main() {
    const char *str = "99999999999999999999"; // Слишком большое для long!
    char *endptr;
    long num = strtol(str, &endptr, 10);

    // ОШИБКА: забыли проверить errno и переполнение!
    printf("Число: %ld\n", num); // Может вывести мусор или LONG_MAX

    return 0;
}
```

- Если число не влезает в `long`, `strtol` возвращает `LONG_MAX` или `LONG_MIN`, но без проверки `errno` вы не узнаете, была ли это ошибка или корректное значение.

2. Если строка содержит мусор (например, "123abc"), `endptr` не проверяется, и программа может работать с некорректными данными.

Правильный вариант:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <limits.h>

int main() {
    const char *str = "99999999999999999999"; // Проверим переполнение
    char *endptr;
    errno = 0; // Сбрасываем errno перед вызовом
    long num = strtol(str, &endptr, 10);

    // 1. Проверка переполнения
    if (errno == ERANGE) {
        printf("Ошибка: число вне диапазона long!\n");
        return 1;
    }

    // 2. Проверка, что строка вообще содержит число
    if (endptr == str) {
        printf("Ошибка: в строке нет числа!\n");
        return 1;
    }

    // 3. Проверка, что вся строка корректна (нет мусора в конце)
    if (*endptr != '\0') {
        printf("Ошибка: в строке есть мусор: '%s'\n", endptr);
        return 1;
    }

    // Если всё OK – выводим результат
    printf("Успешно: %ld\n", num);
    return 0;
}
```

По значению также можно передавать структуры и перечисления, однако для больших данных возникают накладные расходы на копирование и это может приводить к заметному замедлению программы.

```
struct Point {  
    int x;  
    int y;  
};  
  
struct Point create_point(int x, int y) {  
    struct Point p = {x, y};  
    return p;  
}
```

Возврат по указателю (by pointer) возвращает адрес на данные, а не их копию.

```
int* create_array(int size) {  
    int *arr = (int*)malloc(size * sizeof(int));  
    return arr; // Возвращает указатель на выделенную память  
}
```

Возврат через параметры функции (по указателю) возвращает адрес на данные через параметры. При этом через оператор `return` можно сообщать код ошибки вместо использования глобальной переменной `errno`.

```
int divide(int a, int b, double *result) {  
    if (b == 0) {  
        return -1; // Ошибка: деление на ноль  
    }  
    *result = (double)a / b;  
    return 0; // Успешное выполнение  
}
```

При написании функций, возвращающих результат в виде указателя через оператор `return`, нужно убедиться, что объект, на который указывает указатель, останется валидным после выхода из функции.

Пример:

```
int* sum(int a, int b) {
    int result = a + b;
    return &result; // Ошибка! Нельзя возвращать адрес локальной переменной
}
```

Здесь локальная переменная `result` перестанет существовать при выходе из функции, поэтому указатель будет невалидным, то есть содержать некорректный адрес.

Объявление и определение функций

Функция должна иметь объявление и определение. Определение при этом может служить одновременно и объявлением, если это определение размещено до точки своего вызова.

Объявление функции (function declaration) - это указание компилятору о том, что функция существует, какие у неё имя, тип возвращаемого значения, количество аргументов, и их типы. Оно не содержит тела функции. Объявление еще называют прототипом функции. Объявление должно предшествовать вызову функции.

Зачем это нужно?

- Позволяет вызывать функцию **до её определения**.
- Помогает компилятору проверить корректность вызова функции по количеству и типам аргументов. В противном случае компилятор выдаст ошибку, если функция не найдена или ее параметры не соответствуют передаваемым аргументам.

Определение функции (function definition) — это реализация функции. Оно содержит **тело функции**, заключенное в фигурные скобки, то есть код, который будет выполнен при вызове этой функции.

Без определения функции компоновщик выдаст ошибку.

```
// Объявление функции (прототип)
типа_возвращаемого_значения имя_функции(параметры);

// Определение функции
типа_возвращаемого_значения имя_функции(параметры) {
    // Тело функции
    return значение; // (если тип не void)
}
```

Определение может быть только одно в пределах одной программы. Это требование называется **One Definition Rule** (ODR) и относится не только к функциям, но и к другим сущностям (глобальные переменные, структуры, объединения, перечисления).

ODR существует для предотвращения конфликтов при компоновке программы. Если одна и та же сущность (например, функция или глобальная переменная) определена несколько раз, компоновщик (linker) не сможет однозначно выбрать правильную версию, что приведёт к ошибке.

Вызов функции и выход из нее

При вызове функции происходит следующее:

1. **Создаётся новый стековый кадр** (stack frame) - выделяется область в стеке, где хранятся:
 - Адрес возврата (точка, куда нужно вернуться после выполнения функции).
 - Аргументы, переданные в функцию.
 - Локальные переменные функции.
2. **Управление передаётся вызываемой функции** - процессор начинает выполнять её код.
3. **После завершения работы функции:**
 - Стековый кадр освобождается.
 - Поток выполнения возвращается в вызывающую функцию (по сохранённому адресу возврата).
 - Если функция возвращает значение, оно передаётся через регистр или стек.

Рекурсивные функции

Рекурсивная функция - это функция, которая вызывает саму себя в процессе выполнения.

Рекурсия позволяет решать задачи, состоящих из вложенных подзадач. Например, такие функции удобны в задачах обхода иерархических структур данных (таких, как деревья), графов. В частности, для обхода файловых систем, форматов xml, json и других, для синтаксического анализа кода и т.п.

Нужно понимать, что число рекурсивных вызовов (глубина рекурсии) ограничено размером стека, и его переполнение (stack overflow) вызывает аварийное завершение программы.

Таким образом для рекурсии должна соблюдать два принципа:

- условие завершения, называемое еще базовым случаем - это условие при котором, функция прекращает вызывать саму себя.
- рекурсивный вызов - функция вызывает саму себя с измененными аргументами, приближаясь к базовому случаю.

Пример вычисления факториала числа:

```
#include <stdio.h>

int factorial(int n) {
    if (n == 0) { // базовый случай
        return 1;
    } else {
        return n * factorial(n - 1); // рекурсивный вызов
    }
}

int main() {
    int num = 5;
    printf("Факториал %d = %d\n", num, factorial(num));
    return 0;
}
```

Часто в образовательных целях алгоритмы сортировки, такие как `QuickSort` и `MergeSort`, работающих по принципу "Разделяй и властвуй" (Divide and Conquer), реализуют через рекурсию:

```
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pivot = partition(arr, low, high);
        quickSort(arr, low, pivot - 1); // левая часть
        quickSort(arr, pivot + 1, high); // правая часть
    }
}
```

Однако реальный код той же функции `qsort` из стандартной библиотеки использует итерационный подход, т.к. он более эффективен и безопасен на больших объемах данных.

Совет: анализируйте объемы и глубину рекурсии прежде чем реализовывать подобные функции. Рассмотрите вариант с циклами (итерационный подход).

Функции без параметров

Функции без параметров должны использовать `void` внутри круглых скобок.

```
// Объявление функции foo без аргументов, возвращающая int
int foo(void);
```

Синтаксис без `void` тоже работает, но с ним связаны определенные проблемы в C (но не в C++).

В старом K&R стиле, широко использовавшимся до стандарта ANSI C, определение функций выглядело несколько иначе. Вот функция возведения в степень из первого издания книги K&R:

```
// Код в старом K&R стиле
power(x, n)
int x, n;
{
    int p;

    for (p = 1; n > 0; --n)
        p = p * x;
    return(p);
}
```

При этом объявление этой функции выглядело так:

```
power();
```

Как же тогда компилятор проверял корректность переданных параметров (количество и типы)? Короткий ответ - никак! То есть не проверял вообще. Пустые скобки означали сколько угодно параметров. Поэтому можно написать вот такой код:

```
// Код в старом K&R стиле
power();
```

```

main()
{
    printf("power = %d\n", power(2, 4, 3)); // Три параметра?
}

power(x, n)
int x, n;
{
    int p;

    for (p = 1; n > 0; --n)
        p = p * x;
    return(p);
}

```

И компилятор скомпилирует это. К сожалению, запретить такой синтаксис не смогли, слишком много кода на С было написано уже к выходу ANSI C в 1989 году.

Когда компиляторы научились проверять параметры, возник вопрос: а как отличить функцию без параметров от функций в старом стиле? Проблему частично решили вводом ключевого слова `void` внутри круглых скобок, что означало отсутствие аргументов. А использование пустых круглых скобок после выхода ANSI C стало дурным тоном, затем в C99 было объявлено устаревшим (`deprecated`), а в C23 и вовсе запретили старый стиль.

Вариативные функции

Вариативная функция (variadic function) - это функция, которая принимает произвольное количество аргументов любых типов. Наиболее ярким примером может служить функция форматированного вывода `printf` из стандартной библиотеки.

Вариативная функция обязана иметь не менее одного аргумента, а последним аргументом принимать многоточие (...).

```
int printf(const char* format, ...); // прототип функции printf
```

Чтобы работать с переменным числом аргументов, нужно знать их количество и типы. Эта информация в том или ином виде должна передаваться через один из именованных параметров. Это может быть форматная строка, а может параметр любого другого типа.

Для работы с переменными аргументами используются макросы из `<stdarg.h>` (до ANSI С использовался `varargs.h`, но с принятием стандарта был признан устаревшим).

В частности используется тип `va_list` и четыре макроса:

- `va_start` - инициализирует переменную `va_list`
- `va_arg` - получает значение аргумента и смещает переменную `va_list` на следующий аргумент
- `va_copy` - копирует `va_list`
- `va_end` - очищает переменную `va_list`

Реализация `va_list` зависит от архитектуры и компилятора и как именно она устроена волновать разработчиков не должно. Можно воспринимать этот тип как указатель на очередной аргумент функции, то есть адрес на стеке.

Для инициализации переменной типа `va_list` используется макрос `va_start`, которому нужно передать последний именованный аргумент, и, зная адрес этого аргумента и его размер, по сути просто расчитывается адрес следующего аргумента на стеке.

```
void va_start(va_list ap, paramN);
```

Начиная с C23 можно не указывать последний именованный аргумент. Однако, этом может вызвать проблемы с совместимостью со старым кодом.

`va_start` нужно всегда использовать в паре с `va_end`, который всего лишь обнуляет переменную `va_list`.

```
void va_end(va_list ap);
```

`va_arg` при каждом вызове возвращает значение аргумента, на который указывает переменная `va_list` и смещает её на следующий аргумент. При этом важно правильно указывать тип аргумента.

```
T va_arg(va_list ap, T);
```

Пример:

```
include <stdio.h>
include <stdarg.h>
```

```

double average(int i, ...);

int main(void) {
    double x = 12.4;
    double y = 4.5;
    double z = 7.7

    printf( "%s%.3f\n",
        "The average of x, y, and z is ",
        average( 3, x, y, z ) );
}

double average(int i, ...) {
    double total = 0;

    va_list val;
    va_start(val, i); // указываем последний именованный аргумент - i

    for (int j = 1; j <= i; ++j) {
        total += va_arg(val, double);
    }

    va_end(val); // очистка списка
    return total / i;
}

```

На самом деле, для подобных функций (с одинаковым типом аргументов) лучше (безопаснее и производительнее) использовать передачу массива с указанием количества элементов этого массива:

```

#include <stdio.h>

double average(double arr[], int count);

int main(void) {
    double x = 12.4;
    double y = 4.5;
    double z = 7.7;

```

```

// Создаем массив и передаем его в функцию
double numbers[] = {x, y, z};
int count = sizeof(numbers) / sizeof(numbers[0]);

printf("%s%.3f\n",
       "The average of x, y, and z is ",
       average(numbers, count));

return 0;
}

double average(double arr[], int count) {
    double total = 0;

    for (int i = 0; i < count; ++i) {
        total += arr[i];
    }

    return total / count;
}

```

В C99 добавили макрос `va_copy`, который создает копию переменной `va_list`. Эту копию удобно использовать при многократном проходе по аргументам или для передачи этой копии в другие функции.

```
void va_copy(va_list dest, va_list src);
```

8. Функция main

Любая программа на языке С имеет главную функцию, которая должна называться `main`. Она является отправной точкой для каждой программы. В ней начинается выполнение, из нее вызываются другие функции и в ней же выполнение программы завершается, возвращая результат в систему.

Она имеет две сигнатуры:

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

Первый вариант используется в тех случаях, когда нет необходимости передавать аргументы в программу через командную строку. Программа "Hello world", рассмотренная ранее тому пример.

В случае, когда в программе нужно обрабатывать параметры командной строки, используется второй вариант:

```
int main(int argc, char *argv[])
{
    if (argc > 1) // если передан параметр
        printf(argv[1]);
    return 0;
}
```

Первый параметр `argc` сообщает количество параметров, включая имя самой программы.

Сами же параметры командной строки доступны через параметр `argv`, который представлен массивом строк или точнее указателей на тип `char`.

В вышеприведенном примере `argv[1]` - это второй элемент массива `argv`. Проверка `if` необходима, чтобы случайно не вызвать `printf` в случае, когда в программу аргументы не были переданы.

Скомпилируйте этот код:

```
gcc args.c
```

и запустите командой:

```
./a.out "hello, world"
```

В результате на экран будет выведено сообщение:

```
hello, world
```

Можно обойтись и без параметра `argc`, т.к. массив `argv` завершает элементом, равным `NULL`, а значит массив легко перебрать в цикле до тех пор, пока не встретится указатель `NULL`:

```
int main(int argc, char *argv[])
{
    char** arg = argv;
    while (arg++)
        printf("%s\n", arg);
    return 0;
}
```

Более того, под Windows и Linux доступна еще одна сигнатура функции `main` (отсутствует в стандартах языка C), в которой есть массив переменных среды:

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

И эти переменные также легко перебрать в цикле:

```
int main(int argc, char *argv[], char *envp[])
{
    char** p = envp;
    while (p++)
        printf("%s\n", p);
    return 0;
}
```

Под Windows также используются сигнатуры `main` для `wchar_t` (для UNICODE-приложений):

```
int wmain(int argc, wchar_t *argv[]);
int wmain(int argc, wchar_t *argv[], wchar_t *envp[]);
```

В современных консольных приложениях для Windows должны использоваться именно эти функции.

Для классических GUI-приложений предусмотрены другие функции в качестве `main`:

```
int WINAPI WinMain(
    HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPSTR     lpCmdLine,
    int       nShowCmd);

int WINAPI wWinMain(
    HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPWSTR    pCmdLine,
    int       nShowCmd);
```

Во времена, когда писали приложения совместимые с ANSI-версиями и UNICODE-версиями Windows использовали специальный заголовочный файл `<tchar.h>` и макросы `_T`, `_TCHAR` и т.д., а функция `main` для консольных приложений выглядела так:

```
#include <tchar.h>

int _tmain(int argc, _TCHAR *argv[]);
```

а для GUI-приложений так:

```
#include <tchar.h>

int APIENTRY _tWinMain(
    HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPTSTR    lpCmdLine,
    int       nCmdShow)
```

Начиная с Visual Studio 2005 это считается legacy-практикой и современные приложения должны использовать функции `wmain` или `wWinMain`.

И теперь, когда мы почти все знаем про функции `main`, можно выложить уже все карты на стол. На самом деле, функции `main` - это вовсе не точки входа в программу.

В консольных Windows приложениях точкой входа является функция `mainCRTStartup`, которая и вызывает `main`. В Linux-системах точкой входа является функция `_start`.

После запуска приложения, производится довольно много работы под капотом и когда двигатель прогреется, тогда вызывается функция `main`.

Один из излюбленных среди хакеров способов уменьшить исполняемый exe-файл (вирусы намного приятнее распространять когда они весят всего скажем 6-20 килобайт) - использование `mainCRTStartup` или вообще своей функции и отказ от C Runtime библиотеки:

```
#include <Windows.h>

void __stdcall EntryPoint() {
    MessageBoxW(NULL, L"No CRT, no standard entry point!", L"WinAPI", MB_OK);
    ExitProcess(0); // Важно завершить процесс явно!
}
```

Для сборки этого примера в Visual Studio понадобятся параметры:

```
/ENTRY:EntryPoint
/NODEFAULTLIB
```

9. Типы данных

Базовые типы

В языке С имеются базовые (встроенные, примитивные) типы данных, которые делятся на категории:

1. Целочисленные типы (Integer types):

- `char` – символьный тип (не менее 8 бит, обычно 1 байт).
- `short (short int)` – короткое целое (не менее 16 бит, обычно 2 байта).
- `int` – целое число (не менее 16 бит, обычно 4 байта).
- `long (long int)` – длинное целое (не менее 32 бит, всегда 4 байта в Windows и 4 байта в 32-битной Linux и 8 байт в 64-битной Linux).
- `long long (long long int)` – очень длинное целое (не менее 64 бит, обычно 8 байт).

По-умолчанию, они являются знаковыми (`signed`, можно не указывать явно). Для получения беззнакового типа нужно применить модификатор `unsigned`.

Примеры:

```
int n; // знаковый (signed) int
signed int m; // знаковый int
unsigned short int x; // беззнаковый short
```

С самого первого стандарта ANSI C доступен тип `size_t`, в котором удобно и более безопасно (по сравнению с обычным `int`) хранить размеры и индексы.

Кроме того, в C99 были приняты типы с фиксированной шириной, такие как

```
int8_t , int16_t , int32_t и int64_t ,
```

а также их беззнаковые варианты:

```
uint8_t , uint16_t , uint32_t и uint64_t
```

Для их использования необходимо подключить заголовочный файл `<stdint.h>`

Для хранения адресов указателей удобно использовать типы `intptr_t`, `uintptr_t`.

2. Типы с плавающей запятой (Floating-point types)

- `float` – число с плавающей точкой (обычно 4 байта).
- `double` – двойной точности (обычно 8 байт).
- `long double` – расширенной точности (размер зависит от системы, обычно 10 или 16 байт).

3. Пустой тип (`void`)

Используется для указания отсутствия типа. Например, в функциях, которые ничего не возвращают, или в указателях на неизвестный тип.

4. Логический тип (в C99 и новее)

`_Bool` (или `bool`, если подключён `<stdbool.h>`) – хранит `true` (1) или `false` (0).

В C23 включен как базовый и не требует подключения заголовочного файла `<stdbool.h>`

5. Производные типы (Derived types)

На базе встроенных типов язык С позволяет создавать любые производные, к которым относят:

- Указатели (`int*`, `char*` и т. д.)
- Массивы (`int[10]`, `char[]`)
- Структуры (`struct`)
- Объединения (`union`)
- Перечисления (`enum`)

Перечисления

Перечисление (enumeration) - это производный (пользовательский) тип данных, используемый для хранения именованных константных целочисленных значений. По умолчанию эти значения имеют тип `signed int`.

Определение перечисления производится с помощью ключевого слова `enum`.

Синтаксис:

```
enum Name {  
    CONSTANT1,  
    CONSTANT2,
```

```
// ...
};
```

Каждая константа автоматически получает целочисленное значение, начиная с 0 (если явно не указано иное). Можно явно задавать значения:

```
enum Colors { RED = 1, GREEN = 2, BLUE = 4 };
```

В памяти enum хранится как целое число (int), соответственно размер его эквивалентен sizeof(int).

enum, как и целое число можно использовать в арифметических операциях и сравнениях. enum можно передавать в функции, ожидающие int, что небезопасно из-за неявного преобразования.

Пример использования:

```
enum TrafficLight { RED, YELLOW, GREEN };

void HandleLight(enum TrafficLight light) {
    switch (light) {
        case RED:   printf("Stop!\n"); break;
        case YELLOW: printf("Wait...\n"); break;
        case GREEN:  printf("Go!\n"); break;
    }
}
```

enum - это удобный способ работать с именованными константами, улучшающий читаемость и поддержку кода.

Объединения

Объединение (union) — это специальный тип данных, который позволяет хранить разные типы данных в одной и той же области памяти. Все члены (поля) union используют один и тот же блок памяти, но интерпретируют его по-разному в зависимости от типа:

```
union MyUnion {
    int x;
    float y;
```

```
    char z;  
};
```

Определение объединения состоит из ключевого слова `union`, идентификатора, членов, заключенных в фигурные скобки, в завершении ставится точка с запятой. Члены объявляются как обычные переменные.

Размер `union` равен размеру наибольшего члена.

Нужно быть крайне осторожным при работе с `union` - изменение любого члена ведет к изменению всех остальных.

Доступ к членам `union` происходит через оператор `.` (точка). Этот оператор так и называется - оператор доступа к членам.

```
union Numbers {  
    int i;  
    float f;  
};  
  
union Numbers first_number;  
first_number.i = 5;  
first_number.f = 3.9;
```

Инициализация происходит через присвоение значения первому члену `union`:

```
union Numbers first_number = { 5 };
```

Либо можно непосредственно указать имя члена, которому нужно присвоить значение (C99):

```
// Примечание: такой синтаксис не поддерживается в C++.  
union Numbers first_number = { f: 3.14159 };  
// or  
union Numbers first_number = { .f = 3.14159 };
```

Использование `union`

`union` часто используют в embedded разработке, где экономия памяти до сих пор актуальна. Если в функции есть переменные, которые никогда не используются одновременно, `union` позволяет сэкономить память.

Или, например, можно разбирать байты числа по отдельности:

```
union IntBytes {
    int value;
    unsigned char bytes[sizeof(int)];
};
```

Особенно популярным является тип `Variant`, который позволяет через специальное поле структуры контролировать тип. Вот одна из его упрощенных версий:

```
typedef enum { INT, FLOAT, CHAR } Type;

struct Variant {
    Type type;
    union {
        int i;
        float f;
        char c;
    } value;
};
```

Структуры

Структура (`structure`) - это тип данных, определенный программистом, состоящий из переменных других типов данных, включая другие структуры.

Определение структуры состоит из ключевого слова `struct`, идентификатора, членов заключенных в фигурные скобки, в завершении ставится точка с запятой. Члены объявляются как обычные переменные.

```
struct Point {
    int x;
    int y;
};
```

Структуры позволяют объединять связанные переменные разных типов под одним именем.

Нельзя создать рекурсивную структуру, в которой один из членов имеет тот же тип.

```
struct Node {  
    int data;  
    struct Node next; // ОШИБКА: поле не может быть структурой того же типа  
};
```

Но можно это сделать через использование указателей (т.к. размер указателей известен и компилятор может легко вычислить размер структуры для выделения нужного места в памяти):

```
struct Node {  
    int data;  
    struct Node* next; // OK: указатель имеет фиксированный размер  
};
```

Доступ к членам `struct` происходит через оператор `.` (точка).

Для создания переменной типа структуры, используется ключевое слово `struct`, идентификатор структуры и идентификатор переменной:

```
struct Point pt;
```

Инициализация выполняется через оператор присваивания и фигурные скобки:

```
struct Point pt = { 5, 7 };
```

Либо через указание имен полей (C99, не поддерживается в C++):

```
struct Point ptA = { .y = 7, .x = 42 };
```

```
struct Point ptB = { y: 6, x: -24 };
```

Разница между объединением и структурой заключается в том, что члены объединения располагаются в памяти по одному и тому же адресу. В структурах члены располагаются друг за другом с учетом выравнивания полей.

Выравнивание полей в структурах.

Выравнивание (alignment) полей в структурах С определяет, как компилятор размещает данные внутри структуры в памяти. Это делается для оптимизации производительности и обеспечения совместимости с аппаратным обеспечением. Обычно компилятор автоматически выравнивает поля, исходя из их типа данных, чтобы обеспечить доступ к ним за одну операцию процессора.

Компилятор автоматически выравнивает поля структуры по границам, кратным их собственному размеру. Например, `char` (1 байт) выравнивается по границе, кратной 1 байту, `short` (2 байта) по границе, кратной 2 байтам, `int` (4 байта) по границе, кратной 4 байтам и т.д.

Если поля не выровнены должным образом, это может привести к проблемам, особенно при доступе к данным на разных платформах или при работе с аппаратными интерфейсами. Например, чтение 4-байтного целого числа из адреса, не кратного 4, может потребовать нескольких операций процессора, что замедляет работу.

Рассмотрим следующую структуру:

```
struct BadExample {
    char a;          // 1 байт
    int b;           // 4 байта
    short c;         // 2 байта
};
```

Ее размер в памяти будет равен 12 байтам. Компилятор "организует" эту структуру по своему усмотрению:

```
struct BadExample {
    char a;          // 1 байт
    char _pad1[3];   // заполнение 3-мя байтами
    int b;           // теперь int выровнен по 4 байтам
    short c;         // 2 байта
    char _pad2[2];   // чтобы размер структуры был кратен максимальному
```

```
выравниванию
```

```
};
```

Можно воспользоваться оператором `sizeof`, чтобы узнать размер структуры или другого типа данных в байтах:

```
printf("Размер структуры BadExample: %zu байт\n", sizeof(struct BadExample));
```

Обратите внимание, `sizeof` - это не функция, это встроенный в язык унарный оператор, который можно использовать без скобок:

```
size_t n = sizeof int; // скобки необязательны
```

Чтобы не терять память на заполнениях, необходимо поля в структурах размещать с учетом выравнивания.

```
struct GoodExample {
    int b;          // теперь int выровнен по 4 байтам
    short c;        // 2 байта
    char a;         // 1 байт
    char _pad1;     // заполнение в 1 байт
};
```

Теперь эта структура занимает 8 байт, т.е. на 25% меньше. Представьте, что у вас таких структур несколько миллионов. Теперь посчитайте, сколько памяти вы сэкономили на простой перестановке полей.

Упаковка структур

Директива препроцессора `#pragma pack` упаковывает (packing) структуры с учетом заданного выравнивания, вплоть до 1 байта:

```
#pragma pack(push, 1) // упаковывать без выравнивания
struct NetworkPacket {
    char header;
    int data;
    short checksum;
```

```
};  
#pragma pack(pop) // вернуть обычное выравнивание
```

Использование `#pragma pack` или других механизмов позволяет "упаковывать" структуру, уменьшая размер, но потенциально увеличивая время доступа к данным. Например, если установить уровень упаковки в 1, то все поля будут выровнены по 1-байтной границе, что может привести к более компактной структуре, но увеличению времени доступа.

Упаковка может быть оправдана в сетевых протоколах и различных обменах данными.

В C11 разрешили **пустые структуры**, размер которых может равняться 0:

```
struct empty {  
};
```

Это довольно тонкий момент, поскольку стандарт не гарантирует размер структуры, т.е. это implementation-defined (зависит от компилятора: GCC/clang считает его как 0 байт, MSVC - 1 байт, как в C++).

Таким образом, в переносимом коде использовать пустые структуры не рекомендуется.

Анонимные структуры

Анонимная структура (C11) - это структура без имени, объявленная внутри другой структуры или union'a. Она позволяет обращаться к вложенным полям напрямую, без указания имени промежуточной структуры.

```
struct Outer {  
    struct { // ← Анонимная структура (без имени!)  
        int x;  
        int y;  
    }; // Поля x и y доступны напрямую  
    int z;  
};
```

Анонимные структуры часто используются для эмуляции наследования (из ООП) в си:

```
struct Base {  
    int id;
```

```
};

struct Derived {
    struct Base; // Анонимная вложенная структура
    int data;
};
```

Битовые поля

Битовые поля (bit fields) - это специальная возможность структур, которая позволяет определять элементы структуры с точным указанием количества битов, которые они должны занимать. Это полезно для экономии памяти, особенно когда в памяти выделяется огромное количество таких структур.

Синтаксис:

```
struct structure_name {
    field_type field_name : width_in_bits;
    // ...
};
```

Все поля должны быть одного базового целочисленного типа (`int`, `long int`, `short` или `char`) и обычно используют беззнаковые (`unsigned`) типы. То есть `float`, `double` или смешивать типы в битовых полях нельзя.

Например:

```
struct flags {
    unsigned int is_readable : 1;
    unsigned int is_writable : 1;
    unsigned int is_executable : 1;
    unsigned int padding : 5; // неиспользуемые биты
};
```

Такая структура `flags` будет занимать в памяти 4 байта, так как `unsigned int` занимает 4 байта. Это избыточно. Поэтому вместо `unsigned int` можно использовать `unsigned char`:

```
struct flags {
    unsigned char is_readable : 1;
    unsigned char is_writable : 1;
    unsigned char is_executable : 1;
    unsigned char padding : 5; // неиспользуемые биты
};
```

Нельзя у битового поля взять адрес оператором `&`.

Битовые поля удобно использовать совместно с `union` для удобного и быстрого получения значений конкретных битов числа. Например:

```
#include <stdio.h>

// Объединение для доступа к битам float
union FloatBits {
    float value; // Число с плавающей точкой (32 бита)
    struct {
        unsigned int mantissa : 23; // Мантисса (23 бита)
        unsigned int exponent : 8; // Экспонента (8 бит)
        unsigned int sign : 1; // Знак (1 бит)
    } bits;
};

int main() {
    union FloatBits fb;
    fb.value = -12.375f; // Пример числа

    // Вывод битового представления
    printf("Число: %f\n", fb.value);
    printf("Знак: %u (1 = отрицательное)\n", fb.bits.sign);
    printf("Экспонента: %u (смещённая на 127)\n", fb.bits.exponent);
    printf("Мантисса: %u (дробная часть)\n", fb.bits.mantissa);

    // Вывод в шестнадцатеричном формате
    printf("Шестнадцатеричное представление: 0x%08X\n", *(unsigned
int*)&fb.value);
```

```
    return 0;  
}
```

В этом примере демонстрируется вывод отдельных битов типа float (согласно стандарту IEEE 754). `union FloatBits` позволяет интерпретировать одно и то же место в памяти либо как `float`, либо как структуру с битовыми полями.

Структура `bits` разбивает 32-битное число на:

- `sign` (1 бит) — знак числа (`0 = '+', 1 = '-'`).
- `exponent` (8 бит) — экспонента (хранится со смещением +127).
- `mantissa` (23 бита) — мантисса (дробная часть).

Внимание! Этот способ **не переносим** на системы с другим представлением `float`.

Массивы

Массив (`array`) - это структура данных, которая хранит последовательный набор элементов (один или несколько) одного типа в непрерывной области памяти.

Свойства массивов:

- все элементы массива имеют одинаковый тип (`int`, `float`, `char` и т.д.)
- размер массива задается при объявлении и не может быть изменен
- доступ к элементам осуществляется по индексу (начинается с 0)
- элементы расположены в памяти последовательно и непрерывно

Синтаксис:

```
type array_name[length];
```

Объявление массива сопровождается указанием его длины (количество элементов) - целое положительное число.

```
int numbers[10];           // Массив из 10 целых чисел  
float temperatures[31];   // Массив из 31 числа с плавающей точкой  
char name[20];            // Массив из 20 символов
```

До C99 появились гибкие массивы-члены (Flexible Array Members), которые ранее существовали как расширение GCC, допускающее массив нулевой длины. Такие массивы

используются для создания гибких структур данных переменного размера:

```
struct string {
    int length;
    char data[0];
};

// Создание строки нужной длины
struct string* create_string(int length) {
    struct string* s = malloc(sizeof(struct string) + length + 1);
    s->length = length;
    return s;
}

// Использование
struct string* my_str = create_string(10);
strcpy(my_str->data, "Hello");
```

Важно! Гибкие массивы-члены должны быть последним полем структуры.

В C99 и рекомендуется использовать синтаксис `char data[];` вместо `char data[0];`

В C++ гибкие массивы-члены не предусмотрены стандартом, однако все распространенные компиляторы (GCC, Clang, MSVC) их поддерживают.

Инициализация массивов

При объявлении массива можно сразу инициализировать его элементы, указав в фигурных скобках значения, разделенные запятыми:

```
// Инициализация при объявлении
int primes[5] = {2, 3, 5, 7, 11};

// Можно опустить размер, если инициализировать сразу все элементы
int primes[] = {2, 3, 5, 7, 11}; // Компилятор сам определит размер

// Частичная инициализация
int arr[5] = {1, 2}; // Остальные элементы будут 0
```

Обратите внимание на "автоматическое" обнуление неинициализированных элементов. Согласно стандарта, если количество инициализаторов меньше размера массива, остальные элементы инициализируются **как если бы они были статическими** (т.е. нулями).

Внимание! Для того чтобы в C инициализировать все элементы нулями, до C23 нельзя было использовать способ из C++:

```
int arr[5] = {};
```

Правильно:

```
int arr[5] = {0}; // Первый элемент 0, остальные неявно обнуляются
```

В C99 добавили еще один способ инициализации (не поддерживается в C++) через указание конкретного элемента по индексу в квадратных скобках и опционально оператор присваивания:

```
int arr[5] = {[1] 42, [3] 99};  
// эквивалентно  
int arr[5] = {[1] = 42, [3] = 99};  
// эквивалентно  
int arr[5] = {0, 42, 0, 99, 0};
```

Неуказанные элементы будут обнулены.

Доступ к элементам массива происходит через указание индекса (нумерация начинается с нуля) в квадратных скобках:

```
arr[2] = 99; // Присвоить третьему элементу массива значение 99  
int n = arr[0]; // Записать в переменную n значение первого элемента массива
```

Размер массива

Размер можно получить через оператор `sizeof`:

```
int arr[5] = {1, 2, 3, 4, 5};  
size_t n = sizeof(arr); // вернет 5 * sizeof(int)
```

Обратите внимание, что возвращает не количество элементов, а размер в байтах! Чтобы получить количество элементов, нужно разделить результат на размер одного элемента:

```
int arr[5] = {1, 2, 3, 4, 5};  
size_t n = sizeof(arr) / sizeof(int); // вернет 5
```

Но такой код чреват ошибками, если в будущем, кто-нибудь решит изменить тип элементов массива arr. Поэтому более безопасным будет следующий код:

```
int arr[5] = {1, 2, 3, 4, 5};  
size_t n = sizeof(arr) / sizeof(arr[0]); // вернет 5
```

Многомерные массивы

Многомерные массивы - это массив массивов. Для дополнительного измерения нужно добавить еще пару квадратных скобок с указанием длины измерения:

```
int arr[2][3] = {{0, 1, 2}, {3, 4, 5}};
```

В С исторически принято хранение по строкам (row-major order), что, скорей всего, было заимствовано из FORTRAN и это удобно при работе с матрицами или табличными данными.

При обходе такого массива циклы работают так как мы обычно читаем текст в книге: по строкам сверху вниз и по буквам слева направо, то есть интуитивно понятно:

```
for (int i = 0; i < 2; i++) {      // По строкам  
    for (int j = 0; j < 3; j++) {  // По столбцам  
        printf("%d ", arr[i][j]);  
    }  
}
```

То есть для удобства можно запомнить такую схему:

```
int arr[ROWS][COLS]; // [Количество строк][Количество столбцов]
```

Многомерные массивы продолжают располагаться в памяти последовательно.

Разумеется, что на двумерных массивах возможности С не заканчиваются. Измерений может быть сколько угодно, но учтите, что чрезмерное употребление количеством измерений вредно для стека или чревато другими проблемами:

```
int arr[100][1000][10000][100000][1000000];
```

В массивах часто хранят строки. Причем инициализировать строку в виде массива можно разными способами

```
char red[4] = {'r', 'e', 'd', '\0'};  
char green[6] = "green";  
char blue[] = {'b', 'l', 'u', 'e', '\0'};  
char yellow[] = "yellow";
```

Последний способ, пожалуй, самый дружелюбный по отношению к программисту.

Указатели

Указатель (pointer) хранит адрес размещенного в памяти объекта (переменной, константы, функции) определенного типа. Указатели позволяют работать с данными косвенно, через их расположение в памяти, а не напрямую через их значения. Особенно удобны указатели при работе с массивами.

Объявление указателя похоже на объявление обычной переменной. Отличие заключается в добавлении перед идентификатором **оператора разыменования** `*` (indirection operator), который сообщает, что переменная является указателем.

Синтаксис:

```
int * a;
```

Пробелы вокруг оператора разыменования не являются значимыми, поэтому следующие объявления являются эквивалентными:

```
int *a; // чаще используется в С  
int* b; // чаще используется в C++  
int * c; // редко используемый и не рекомендуемый стиль
```

Важно! Придерживайтесь стиля, прописанного в соглашении о кодировании (style guide), принятом в компании или в проекте. Если такого документа нет, что само по себе плохо, придерживайтесь общепринятого подхода, указанного в комментариях выше в примере.

При этом есть важное преимущество у `int *a`. Такая запись позволяет избегать ошибок при объявлении нескольких указателей в одной строке:

```
int *a, *b; // a и b – указатели  
int* c, d; // c – указатель, d – переменная типа int
```

Инициализация указателей

Указатель можно инициализировать при его объявлении, указав адрес переменной, на которую указывает этот указатель. Адрес можно получить при помощи **оператора взятия адреса** `&` (address operator). Например, в следующем примере объявляется переменная `int i` и указатель, который инициализируется адресом `i`:

```
int i = 42;  
int *p_i = &i;
```

Указателю можно присвоить адрес другой переменной:

```
int x = 1;  
p_i = &x;
```

Указатель можно обнулить:

```
p_i = NULL;  
// или  
p_i = 0;
```

`NULL` – это макрос из `<stddef.h>`. В C23 появился `nullptr` (займствован из C++), как более безопасная альтернатива `NULL`.

Разыменование указателей

Для чтения или изменения данных, лежащих по указателю, используется оператор разыменования `*`:

```
int i = 42;
int *p_i = &i;
*p_i = 7; // теперь переменная i равна 7
int x = *p_i; // переменная x инициализируется равной 7
```

Важно! Остерегайтесь чтения или записи памяти по невалидным адресам, в частности `NULL`. Такие обращения к памяти ведут к ошибке `Segmentation Fault` и аварийному завершению программы.

Арифметика указателей

С указателями можно выполнять следующие операции:

- Увеличение (`+`, `++`) и уменьшение (`-`, `--`) с учетом размера типа данных указателя.

```
int arr[] = {1, 2, 3, 4, 5, 6, 7};
int *ptr = &arr[0]; // указатель на первый элемент массива

++ptr; // ptr указывает на второй элемент массива (arr[1])
ptr = ptr + 2; // ptr указывает на четвертый элемент массива (arr[3])
ptr -= 3; // ptr указывает на первый элемент массива (arr[0])
ptr += 6; // ptr указывает на последний элемент массива (arr[6])
```

Очень важно учитывать границы массива. Попытка чтения или записи за пределами массива может привести к аварийному завершению программы. Подобные ошибки являются очень распространенными и активно эксплуатируются хакерами для взлома.

- Разность указателей одного типа позволяет получить количество элементов между ними.

```
int arr[] = {1, 2, 3, 4, 5, 6, 7};
int *begin = &arr[0];
int *end = &arr[7]; // адрес за пределами массивами (разрешено)
ptrdiff_t diff = end - begin; // diff = 7 элементов в массиве
```

Тип `ptrdiff_t` из `<stddef.h>` используется как специальный тип разницы указателей. При использовании базовых типов `int` возможно переполнение типа, что тоже может использоваться как уязвимость.

Обратите внимание, что указатель `end` получен через взятие адреса на элемент, следующий сразу за последним элементом массива. Это разрешено, но читать по адресу этого указателя или выполнять запись запрещено.

Если бы мы в качестве `end` использовали адрес последнего элемента, тогда для корректного расчета элементов необходимо было бы прибавить к результату единицу, что добавляет работы и снижает производительность.

- Сравнение указателей (`==`, `!=`, `<`, `>`, `<=`, `>=`)

Указатели можно сравнивать и это может быть очень эффективно использовано. Вот пример функции, которая заменяет все символы `old_char` в строке на символ `new_char`, начиная с позиции `first` и заканчивая позицией `last` (без проверки валидности аргументов):

```
int replace(char *str, char old_char, char new_char, int first, int last) {

    int replaced = 0;
    for (int i = first; i <= last; i++) {
        if (str[i] == old_char) {
            str[i] = new_char;
            replaced++;
        }
    }

    return replaced;
}
```

Функция вполне рабочая, но ее можно написать эффективнее через сравнение указателей (обратите внимание на то, что сигнатура функции тоже изменилась):

```
int replace(char* from, char* to, char old_char, char new_char) {

    int replaced = 0;
    while (from <= to) {
        if (*from == old_char) {
            *from = new_char;
            replaced++;
        }
    }
}
```

```

    ++from;
}

return replaced;
}

```

Сравните до:

```

; if (str[i] == old_char) {
mov      eax, DWORD PTR [rbp-8]
movsx   rdx, eax
mov     rax, QWORD PTR [rbp-24]
add     rax, rdx
movzx  eax, BYTE PTR [rax]
cmp    BYTE PTR [rbp-28], al
jne     .L3
; str[i] = new_char;
mov      eax, DWORD PTR [rbp-8]
movsx   rdx, eax
mov     rax, QWORD PTR [rbp-24]
add     rdx, rax
movzx  eax, BYTE PTR [rbp-32]
mov    BYTE PTR [rdx], al
; replaced++;
add     DWORD PTR [rbp-4], 1

```

и после:

```

; if (*from == old_char) {
mov      rax, QWORD PTR [rbp-16]
movzx  eax, BYTE PTR [rax]
cmp    BYTE PTR [rbp-44], al
jne     .L3
; *from = new_char;
mov      rax, QWORD PTR [rbp-16]
movzx  edx, BYTE PTR [rbp-48]
mov    BYTE PTR [rax], dl

```

```
; replaced++;
add     DWORD PTR [rbp-4], 1
```

На самом деле, код с большой вероятностью будет оптимизирован и оба варианта будут одинаково производительны.

Совет: старайтесь избегать использования операторов доступа по индексу, если есть возможность использовать указатели.

Часто указатели сравнивают с `NULL`. Например, при обходе списка или массива указателей, где элемент, равный `NULL` означает конец.

- Присваивание указателей

Можно присваивать указатели **одного типа** (или `void*` с неявным приведением):

```
int i = 42;
void* p = &i; // p хранит адрес i, но не хранит информацию о типе
int* p_i = p; // p неявно приводится к int*, p_i теперь указывает на i
```

Неявное приведение `void*` к указателю на тип разрешено в C, но запрещено в C++, в котором требуется явное приведение к типу:

```
int i = 42;
void* p = &i; // C++ разрешает неявное приведение к void*
int* p_i = (int*)p; // C++ требует приведения void* к int*
```

- Разыменование (`*` и `->`)

Оператор разыменования `*` (indirection operator) используется для получения значения по адресу. В разговорной речи и различных источниках чаще употребляется как dereference operator.

```
int x = 42;
int *ptr = &x; // ptr хранит адрес x

int value = *ptr; // разыменовываем ptr и получаем 42
*ptr = 100;       // изменяем x через ptr (теперь x = 100)
```

Оператор стрелка `->` или **оператор доступа к члену через указатель** (member access via pointer operator) позволяет обращаться к полям структуры. Можно обойтись и без этого оператора, используя оператор разыменования и обычный оператор доступа к членам структуры (`.` - точка), но это слишком громоздко и неудобно:

```
(*struct_pointer).field // то же самое, что struct_pointer->field
```

Пример использования:

```
#include <stdio.h>

typedef struct {
    int age;
    char name[32];
} Person;

int main() {
    Person person = {30, "Alice"};
    Person *ptr = &person; // указатель на структуру

    printf("Age: %d\n", ptr->age); // то же, что (*ptr).age
    printf("Name: %s\n", ptr->name); // то же, что (*ptr).name

    return 0;
}
```

- Взятие адреса (`&`)

Оператор взятия адреса `&` (address operator) возвращает адрес переменной (константы, функции) или другими словами указатель.

Важно! Оператор `&` применим только к элементам, имеющим имя. Поэтому, например, нельзя получить адрес литерала:

```
int *p = &42; // Ошибка!
```

Важно! Оператор `&` возвращает временное значение (rvalue), которое нельзя изменить:

```
&x = NULL; // Ошибка!
```

Можно получить адрес указателя (указатель на указатель):

```
int *p = &i;  
int **pp = &p;
```

- Умножение и деление указателей бессмыслены и запрещены (компилятор выдаст ошибку).

Указатели и массивы

Указатели и массивы тесно связаны. Через указатели можно осуществлять доступ к элементам массива, передавать массивы в функции, а также динамически выделять память под массивы.

Имя массива в сущности это указатель на его первый элемент.

```
int arr[5] = {1, 2, 3, 4, 5};  
int *ptr = arr; // Эквивалентно int *ptr = &arr[0]
```

С указателями можно использовать оператор доступа к элементам массива `[]` (Array subscripting)

```
printf("%d\n", arr[2]); // 3 – доступ через имя массива  
printf("%d\n", ptr[2]); // 3 – доступ через указатель
```

Доступ по индексу в сущности выполняет операцию добавления индекса к указателю на первый элемент и разыменование результата:

```
printf("%d\n", *(arr + 2)); // 3 – доступ через имя массива  
printf("%d\n", *(ptr + 2)); // 3 – доступ через указатель
```

Вот почему в циклах лучше использовать указатели, а не оператор `[]`

Приведение типов

Приведение типов (type casting) — это преобразование значения одного типа данных в значение другого типа данных.

Синтаксис (явное приведение):

```
(тип) выражение
```

В языке С автоматически (неявно) происходит приведение типов в следующих случаях:

1. Присваивание

```
short a = 42;
int b = a; // short неявно приводится к int

double d = 3.14;
int i = d; // double неявно приводится к int (теряется дробная часть)
```

2. Передача параметров в функции

```
void foo(double d) { /* ... */ }

int main() {
    int value = 42;
    foo(value); // int неявно приводится к double
}
```

3. Возврат значений из функции

```
double foo() {
    int result = 100;
    return result; // int неявно приводится к double
}
```

4. Массивы и указатели

```
int arr[10];
int* ptr = arr; // массив неявно приводится к указателю
```

Указатель на `void` неявно приводится к любым другим указателям и также обратно (в C++ такое преобразование запрещено и требует явного приведения).

5. Арифметические и побитовые операции

```
int a = 10;
double b = 3.5;
double result = a + b; // переменная a неявно приводится к double

char c = 'A';
int result = c << 2; // переменная c неявно приводится к int
```

Здесь действует следующее правило неявного приведения (оставим за рамками этого правила типы `_Decimal?`, `complex` и `imaginary`):

- если один из операндов `long double`, то другой operand приводится к `long double`
- иначе если один из operandов `double`, то другой operand приводится к `double`
- иначе если один из operandов `float`, то другой operand приводится к `float`
- иначе оба операнда приводятся к `int`. Это называется целочисленным продвижением (integer promotions).

Поэтому следующий пример не должен никого смущать в плане возможного переполнения целых чисел:

```
short a = 32767;
short b = 42;
printf("a + b = %d\n", a + b);
```

Прежде, чем будет выполнена операция сложения `a` и `b`, эти operandы будут неявно приведены к типу `int`, согласно выше описанному правилу, т.е. код будет преобразован в следующий вид:

```
short a = 32767;
short b = 42;
printf("a + b = %d\n", (int)a + (int)b);
```

Соответственно, результат сложения тоже будет `int`.

В языке С нет возможности выполнить операции над типами, меньше `int`. Эта особенность языка помогает в большинстве случаев обходить проблему переполнения целых чисел, которая активно эксплуатируется хакерами.

6. Сравнение

В операциях сравнения применяется то же самое правило неявного приведения, рассмотренное выше.

Рассмотрим следующий код:

```
int IsValidAddition(unsigned short x, unsigned short y) {
    if (x + y < x)
        return 0;
    return 1;
}

int main() {
    unsigned short x = 0xF000;
    unsigned short y = 0x1000;

    printf("IsValidAddition(x, y) = %d\n", IsValidAddition(x, y));

    return 0;
}
```

Предполагается, что при переполнении типа `unsigned short` функция `IsValidAddition` вернет `0`, но этого не происходит, так сравнение всегда будет ложно:

```
((int)x + (int)y) < (int)y
```

Правильный код функции должен выглядеть так:

```
int IsValidAddition(unsigned short x, unsigned short y) {
    if ((unsigned short)(x + y) < x) // явное приведение
        return 0;
    return 1;
}
```

Таким образом, в большинстве случаев нет необходимости явно приводить типы.

Рассмотрим некоторые примеры, где без явного приведения не обойтись:

1. Предотвращение потери данных

```
int a = 10;  
int b = 3;  
float result = (float)a / b; // 3.333 вместо 3
```

2. Указатели разных типов

```
int* int_ptr = malloc(10 * sizeof(int));  
char* char_ptr = (char*)int_ptr; // преобразование указателя
```

3. Манипуляции с битовыми операциями

```
unsigned int flags = 0xABCD1234;  
unsigned char byte = (unsigned char)(flags >> 16); // получить третий байт  
(0xCD)
```

Важно! Приведение типов небезопасно и следует применять его крайне осторожно. По этой причине в C++ приведение разделили на громоздкие `static_cast`, `const_cast` и т.д., чтобы применять их было больно, поскольку в чистом коде приведений типов быть не должно (в идеале).

10. Переменные и константы

Переменная (variable) - это именованная область памяти определенного типа, хранящая некоторое значение, которое может изменяться.

Константы (constant) - это именованная область памяти определенного типа, хранящая некоторое значение, которое не должно изменяться (на самом деле есть способы изменить значение константы, но делать это настоятельно не рекомендуется).

Другими словами, переменные и константы - это объекты, на которые можно ссылаться, используя их имена (идентификаторы). Идентификаторы должны быть уникальными в рамках одной области видимости.

Затенение переменных (shadowing)

Идентификатор должен быть уникальным в рамках одной области видимости (в противном случае компилятор выдаст ошибку переопределения переменной), но при этом допускается создание переменных с тем же именем во вложенных блоках `{}`. Это называется затенением (shadowing) переменной:

```
#include <stdio.h>

int main() {
    int x = 10; // Внешняя переменная
    printf("Внешний x до ввода: %d\n", x);

    // Вложенный блок с затенением переменной x
    {
        // Здесь начинается новая область видимости
        int x; // Локальная переменная (затеняет внешнюю x)
        printf("Введите целое число: ");
        scanf("%d", &x); // Записываем в локальную x

        printf("Локальный x после ввода: %d\n", x);
    }

    // Здесь снова видна внешняя x
}
```

```
    printf("Внешний x после блока: %d\n", x);

    return 0;
}
```

Несмотря на то, что язык разрешает затенение, в реальном коде делать это крайне не рекомендуется, так как это является частой причиной различных ошибок, которые потом сложно отлаживать.

Объявление и определение переменных и констант

Объявление (declaration) - это инструкция, которая сообщает компилятору о существовании переменной или константы.

Определение (definition) - это инструкция, которая выделяет место в памяти (в основном на стеке) и определяет область видимости переменной. При этом может выполняться инициализация или переменная остается в неинициализированном виде.

Область видимости переменной сообщает место, где ее ввели, и до места, где она будет уничтожена.

До C99 переменные нужно было объявлять в начале файла или блока (scope) из-за того, что компиляторы были однопроходными и приходилось сперва размещать их на стеке и только потом с ними работать. Начиная с C99 переменные можно смешивать с остальным кодом. При этом даже рекомендуется объявлять переменные непосредственно перед их использованием.

Пример:

```
unsigned long long factorial(int n)
{
    unsigned long long result = 0; // Плохо, можно объявить ниже

    if (n < 0)
        return result;
    ++result;
    for (int i = 2; i <= n; i++)
        result *= i;
    return result;
}
```

Исправленная версия:

```
unsigned long long factorial(int n)
{
    if (n < 0)
        return 0;
    unsigned long long result = 1; // Ok
    for (int i = 2; i <= n; i++)
        result *= i;
    return result;
}
```

Когда говорят об объявлении (declaration) переменных, то фактически речь идет об объявлении и определении (definition). Объявление без определения возможно только для `extern` переменных и констант. В остальных случаях происходит и объявление и определение.

Инициализация задает начальное значение переменной при ее определении. Для констант инициализация обязательна, т.е. невозможно объявить константу без инициализации, т.к. это приведет к ошибке компиляции.

Важно! Настоятельно не рекомендуется вводить переменные без инициализации, т.к. это зачастую приводит к ошибкам и уязвимостям. Всегда инициализируйте переменные.

Присваивание - это отдельная инструкция и не стоит путать ее с инициализацией, т.к. инициализация происходит при создании переменной, а присваивание выполняется уже когда переменная создана.

```
const float pi = 3.14f; // объявление и инициализация глобальной константы pi
типа float значением 3.14

void foo()
{
    int i; // объявление неинициализированной переменной типа int
    int n = 42; // объявление переменной типа int и инициализация ее значением
42

    n = 0; // присваивание переменной n значения 0
} // здесь переменные i и n перестанут существовать
```

Предварительное определение

Как вам следующий код?

```
#include <stdio.h>

int g_var;

int main() {
    printf("g_var = %d\n", g_var);
}

int g_var = 42;
```

Разве можно дважды определить переменную?

Да, можно! Здесь мы имеем дело с предварительным определением (tentative definition) - это объявление переменной без спецификатора класса хранения (extern или static) и инициализатора. Предварительное определение становится полным определением, если достигнут конец модуля (единицы трансляции) и не появилось определение с инициализатором для идентификатора.

Примечание: в C++ это считается переопределением (redefinition), и этот пример выдаст ошибку.

Рекомендация: избегайте использования предварительных определений переменных.

Локальные переменные

Локальными или автоматическими называют переменные, объявленные внутри тела функции. Такие переменные не доступны за пределами функции и время жизни таких переменных ограничено временем выполнения функции.

```
void foo()
{
    auto int x = 42; // auto необязательно указывать. Эквивалентно int x = 42;
    // ...
}
```

Глобальные переменные

Глобальные переменные объявляются вне функций и доступны во всех функциях определенных ниже своего объявления, а также в других модулях при наличии в них объявления со спецификатором `extern`:

```
int g_var = 42;

int get_var() {
    return g_ar;
}
```

В случае необходимости сделать эту глобальную переменную недоступной для изменения в других модулях, ей нужно назначить спецификатор `static`:

```
static int g_var = 42;

int get_var() {
    return g_ar;
}
```

Статические переменные

Статические локальные переменные сохраняют свое состояние при выходе из функции и время жизни их продолжается до завершения программы:

```
#include <stdio.h>

// Функция со статической переменной
void counter() {
    static int count = 0; // Переменная сохраняет значение между вызовами
    count++;
    printf("Функция вызвана %d раз(а)\n", count);
}

int main() {
    // Вызываем функцию несколько раз
    counter(); // 1-й вызов
}
```

```
    counter(); // 2-й вызов
    counter(); // 3-й вызов

    return 0;
}
```

Статические переменные часто добавляют в целях отладки.

Множественное объявление переменных

Можно объявить сразу в одной строке несколько переменных одного типа и проинициализировать их:

```
// Объявление нескольких переменных одного типа в одну строку
int a = 5, b = 10, c = 15;
```

Обратите внимание, что следующее объявление НЕ означает инициализацию всех перечисленных переменных значением 42:

```
int a, b, c = 42;
```

а и b если на трубе будут содержать случайные данные, а с будет равна 42.

Константы и макросы

Константы, вводимые через макрос `#define` - это, вообще говоря, не константы.

```
#define PI 3.1415926535
```

Это некоторое значение, которое будет вставлено в код через макроподстановку. В современном коде следует избегать подобных макросов и использовать настоящие константы:

```
const double const_pi = 3.1415926535;
```

11. Чтение сложных объявлений

Сложные объявления (например, указатели на функции, массивы указателей и т. д.) могут выглядеть слишком запутанными, но существуют два распространенных метода их чтения: **правило cdecl** (cdecl rule) и "спиральное правило" (Clockwise/Spiral Rule).

Мы рассмотрим **правило cdecl** (оно же **right-left rule**), а спиральное правило оставим для самостоятельного изучения желающими.

Правило cdecl предлагает схему чтения изнутри, начиная с идентификатора, наружу, меняя направление вправо-влево:

1. Найдите имя переменной;
2. Двигайтесь **вправо**, пока не закончатся символы () , [] или ;);
3. Двигайтесь **влево**, пока не закончатся символы ((или [[);
4. Если встречаете () — это функция, возвращающая...;
5. Если встречаете [] — это массив из...;
6. Если встречаете * — указатель на...

Примеры:

```
int arr[10];
```

1. arr - это
2. справа встречаем [10] - массив из десяти элементов (каких?)
3. встретили ; , двигаемся влево, встречаем int - типа int

Ответ: arr - это массив из десяти элементов типа int

```
const int* arr[10];
```

1. arr - это
2. справа встречаем [10] - массив из десяти элементов (каких?)
3. встретили ; , двигаемся влево, встречаем * - указателей
4. двигаемся далее влево, встречаем int - типа int
5. двигаемся далее влево, встречаем const - тип константный

Ответ: arr - это массив из десяти указателей на константный тип int

`const int` и `int const` взаимозаменяемы (одно и то же), поэтому эти записи эквивалентны:

```
const int* arr[10];
int const* arr[10];
```

Рассмотрим более сложные примеры.

```
const int (*arr)[10];
```

1. `arr` - это
2. справа встречаем `)`, меняем направление влево
3. встречаем `*` - значит arr - это указатель
4. встречаем `(`, меняем направление вправо
5. встречаем `[10]` - массив из десяти элементов (каких?)
6. встретили `;`, двигаемся влево, встречаем (прочитаем сразу) `const int` - константных типов `int`

Ответ: arr - это указатель на массив из десяти константных элементов типа int

```
int (*funcs[5])(char);
```

1. `funcs` - это
2. встречаем `[5]` - массив из пяти...
3. встречаем `)`, идем влево
4. встречаем `*` - указателей
5. встречаем `(`, идем вправо, встречаем `(char)` - на функцию, принимающую char
6. встречаем `;`, идем влево, встречаем `int` - возвращающую `int`

Ответ: funcs - это массив из пяти указателей на функции, принимающие char и возвращающие int

Сложный случай:

```
char (*(*x())[5]))();
```

1. `x` - это
2. `()` - функция
3. `)`, движемся влево
4. `*` - возвращающая указатель на
5. `(`, движемся вправо
6. `[5]` - массив из 5 элементов
7. `)`, движемся влево
8. `*` - из пяти указателей
9. `(`, движемся вправо
10. `()` - на функции без аргументов
11. `;` - движемся влево
12. `char` - возвращающие тип `char`

Ответ: `x` - это функция, возвращающая указатель на массив из пяти указателей на функции без аргументов, возвращающие `char`.

Согласитесь, читать такое неприятно. Поэтому для сложных случаев рекомендуется использовать `typedef` для декомпозиции. Например, для последнего примера, можно было бы написать так:

```
typedef char (*char_func_ptr)(); // Указатель на функцию, возвращающую char
typedef char_func_ptr (*func_array_ptr)[5]; // Указатель на массив из 5 таких
                                             // указателей

func_array_ptr x();
```

Можно возразить, что такая запись слишком многословна. Но ее проще читать, а это немаловажно при сопровождении кода.

У начинающих разработчиков возникают некоторые затруднения в использовании спецификатора `const`.

```
const int n;
int const m;
```

Эти записи эквивалентны. Компилятор одинаково их обрабатывает.

Почему появились и существуют обе эти формы?

Спецификатор `const` возник в эпоху стандартизации языка С под влиянием С++, где `const` уже существовал и чаще использовалась первая форма.

Вторая форма возникла из основного принципа чтения переменных в С "читай изнутри наружу" и часто используется в объявлениях указателей:

```
char const *message; // указатель на константный тип char  
char const *const note; // константный указатель на константный тип char
```

Для тех, кто хочет себя проверить на правильность прочтения или понимания сложных объявлений создан сайт, позволяющий "расшифровать" тарабарщину С (C gibberish) на понятный английский - <https://cdecl.org/>

12. Операторы, выражения и инструкции

"Операции, выражения и операторы" - именно так называется глава 5 в книге Стивена Прата (Stephen Prata) "Язык программирования С" в русском переводе. К сожалению, в русском языке существует определенная путаница с терминологией. В оригинале глава называется "Operators, Expressions, and Statements" и почему-то в русскоязычных изданиях слово "statement" принято переводить как "оператор", а "operator" - как "операция". В результате, зачастую, только из контекста можно понять о каком операторе идет речь, об "operator" или "statement".

Мы будем использовать "инструкцию" в качестве перевода для statement (дословно "утверждение"), чтобы избегать этой путаницы. Нужно заметить, что instruction не то же самое, что statement, и имеет несколько иное значение. Можно также встретить использование англицизма - стейтмент, но большого распространения он не получил.

При этом operator будем переводить как "оператор", а expression - как "выражение".

Операторы

Оператор (operator) - это специальный токен (символ или ключевое слово), который выполняет операцию над одним или несколькими operandами (значениями или переменными), является частью выражения (expression) и участвует в вычислении значения.

Унарные операторы выполняют операции над одним операндом, **бинарные** над двумя, а **тернарные** используют три операнда.

По функциональности выделяют следующие виды операторов:

- операторы присваивания;
- операторы инкремента/декремента;
- арифметические;
- операторы сравнения/отношения;
- логические;
- операторы доступа к членам;
- прочие (запятая, тернарные, sizeof, приведения типа).

Распространенное заблуждение называть if, for, while и т.д. операторами возникает из-за неоднозначного перевода statement (а это именно statements, а не operators).

Арифметические операторы

Арифметические операторы (Arithmetic operators) выполняют стандартные математические, а также побитовые операции над операндами:

Оператор	Название	Пример	Результат
<code>+</code>	Унарный плюс	<code>+a</code>	Значение операнда
<code>-</code>	Унарный минус	<code>-a</code>	Отрицательное значение операнда
<code>+</code>	Сложение	<code>a + b</code>	Сумма a и b
<code>-</code>	Вычитание	<code>a - b</code>	Разность a и b
<code>*</code>	Умножение	<code>a * b</code>	Произведение a и b
<code>/</code>	Деление	<code>a / b</code>	Частное от деления a на b
<code>%</code>	Остаток от деления	<code>a % b</code>	Остаток от деления a на b
<code>~</code>	Побитовое НЕ (NOT)	<code>~a</code>	Инверсия битов a
<code>&</code>	Побитовое И (AND)	<code>a & b</code>	Побитовое И между a и b
<code> </code>	Побитовое ИЛИ (OR)	<code>a b</code>	Побитовое ИЛИ между a и b
<code>^</code>	Побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR)	<code>a ^ b</code>	Побитовое XOR между a и b
<code><<</code>	Побитовый сдвиг влево	<code>a << b</code>	Сдвиг битов a влево на b позиций
<code>>></code>	Побитовый сдвиг вправо	<code>a >> b</code>	Сдвиг битов a вправо на b позиций

Унарный плюс не рекомендуется использовать в современном коде, т.к. он практически ничего не делает. Унарный плюс и унарный минус выполняют целочисленное продвижение. Это свойство используют иногда для явного приведения к int, но такой код вызывает только лишнюю когнитивную нагрузку при чтении (не все знают тонкости языка, особенно джуны), поэтому делать так не стоит:

```
short s = 100;
int expanded = +s; // Явное расширение до int

int value = s; // Лучше использовать неявное приведение
```

Унарный минус может вызывать переполнение для INT_MIN:

```
int min_int = -2147483648;
int problematic = -min_int; // Переполнение!
```

Стандартные математические (аддитивные и мультипликативные) операции обычно не вызывают сложностей. Стоит обратить внимание на то, что деление целых чисел возвращает целочисленный результат, а не число с плавающей точкой. Например:

```
float f = 10 / 4; // Результат равен 2.0f, а не 2.5f
```

Если требуется результат с плавающей точкой, нужно один из операндов явно привести к типу float:

```
float f = (float)10 / 4; // Теперь
```

Побитовые операторы

Рассмотрим применение побитовых (bitwise) операций на примере с enum :

```
enum FilePermissions {
    NONE      = 0,      // 0000 0000
    READ      = 1,      // 0000 0001
    WRITE     = 2,      // 0000 0010
    EXECUTE   = 4,      // 0000 0100
    DELETE    = 8,      // 0000 1000
    ALL       = READ | WRITE | EXECUTE | DELETE // 0000 1111
};

// Функция для проверки наличия флага
int has_permission(int permissions, enum FilePermissions flag) {
    return (permissions & flag) == flag;
```

```
}

// Функция для добавления флага
int add_permission(int permissions, enum FilePermissions flag) {
    return permissions | flag;
}
```

Для того чтобы "обнулить" флаг удобно использовать NOT:

```
int remove_permission(int permissions, enum FilePermissions flag) {
    return permissions & ~flag;
}
```

Чтобы переключить флаг удобно использовать XOR:

```
int toggle_permission(int permissions, enum FilePermissions flag) {
    return permissions ^ flag;
}
```

Операторы сдвига

Используя побитовый сдвиг влево, можно этот функционал переписать на макросах:

```
// Макрос для создания битовых масок
#define BIT(n) (1 << (n))

// Макрос для создания маски из нескольких битов
#define BITS_MASK(high, low) (((1u << ((high) - (low) + 1)) - 1) << (low))

// Макрос для установки бита
#define SET_BIT_FLAG(reg, bit) ((reg) |= BIT(bit))

// Макрос для сброса бита
#define CLEAR_BIT_FLAG(reg, bit) ((reg) &= ~BIT(bit))

// Макрос для проверки бита
#define CHECK_BIT_FLAG(reg, bit) ((reg) & BIT(bit))
```

```
// Макрос для переключения бита
#define TOGGLE_BIT_FLAG(reg, bit) ((reg) ^= BIT(bit))
```

Следующий пример использует побитовый сдвиг вправо для подсчета количества битов, установленных в 1:

```
int bitcount(unsigned int value)
{
    int result = 0;
    for (result = 0; value != 0; value >>= 1)
        if (value & 1)
            ++result;
    return result;
}
```

Можно встретить код, в котором разработчики пытаются заниматься оптимизацией умножения или деления на степени двойки. Например, умножение на 2 можно выразить через операцию сдвига влево на 1 бит:

```
#include <stdio.h>

// Обычное умножение на 2
int multiply_by_2(int x) {
    return x * 2;
}

// Оптимизированная версия с битовым сдвигом
int shift_multiply_2(int x) {
    return x << 1; // Сдвиг влево на 1 бит = умножение на 2
}

int main() {
    int number = 5;

    printf("Обычное умножение: %d * 2 = %d\n", number, multiply_by_2(number));
    printf("Сдвиговая версия: %d << 1 = %d\n", number,
shift_multiply_2(number));
```

```
    return 0;  
}
```

Для академических целей полезно такой код изучить и понять как он работает. Но в реальном коде такое делать не нужно, так как компилятор подобные оптимизации делает сам и намного эффективнее.

В этом коде сдвиг на 1 эквивалентен умножению на 2. Соответственно, сдвиг на 2 даст умножение на 4, сдвиг на 3 = умножению на 8 и т.д.

Операторы присваивания

Операторы присваивания (Assignment operators) являются бинарными операторами, которые изменяют переменную слева, используя значение выражения справа.

Простой оператор присваивания имеет синтаксис:

```
lhs = rhs
```

Составные операторы присваивания имеют синтаксис:

```
lhs op rhs
```

где `op` один из:

Оператор	Название	Пример	Описание	Эквивалент
<code>+=</code>	Присваивание со сложением	<code>a += b</code>	Прибавляет правый операнд к левому	<code>a = a + b</code>
<code>-=</code>	Присваивание с вычитанием	<code>a -= b</code>	Вычитает правый операнд из левого	<code>a = a - b</code>
<code>*=</code>	Присваивание с умножением	<code>a *= b</code>	Умножает левый операнд на правый	<code>a = a * b</code>
<code>/=</code>	Присваивание с делением	<code>a /= b</code>	Делит левый операнд на правый	<code>a = a / b</code>
<code>%=</code>	Присваивание с остатком от деления	<code>a %= b</code>	Присваивает остаток от деления левого операнда на правый	<code>a = a % b</code>

Оператор	Название	Пример	Описание	Эквивалент
<code><<=</code>	Присваивание сдвигом влево	<code>a <<= b</code>	Сдвигает биты левого операнда влево на значение правого	<code>a = a << b</code>
<code>>>=</code>	Присваивание сдвигом вправо	<code>a >>= b</code>	Сдвигает биты левого операнда вправо на значение правого	<code>a = a >> b</code>
<code>&=</code>	Присваивание побитовым И	<code>a &= b</code>	Выполняет побитовое И между operandами и результат присваивает левому	<code>a = a & b</code>
<code>^=</code>	Присваивание побитовым исключающим ИЛИ	<code>a ^= b</code>	Выполняет побитовое XOR между operandами и результат присваивает левому	<code>a = a ^ b</code>
<code> =</code>	Присваивание побитовым ИЛИ	<code>a = b</code>	Выполняет побитовое ИЛИ между operandами и результат присваивает левому	<code>a = a b</code>

Операторы инкремента/декремента

Унарные операторы инкремента/декремента увеличивают или уменьшают значение на единицу. Имеют две формы:

- Префиксная (`++expr` , `--expr`) - сначала изменяет значение переменной, затем возвращает новое значение.
- Постфиксную (`expr++` , `expr--`) - сначала возвращает текущее значение переменной, затем изменяет её.

```
int a = 1;
int b;

// Префиксный инкремент
b = ++a; // a становится 2, b получает 2
```

```
// Сбросим значения  
a = 1;  
  
// Постфиксный инкремент  
b = a++; // b получает 1, затем a становится 2
```

Важно! Префиксная форма возвращает не копию операнда, а что-то вроде ссылки на этот operand, или еще говорят возвращает `lvalue` (т.е. то, что находится слева от оператора присваивания), а постфиксная форма возвращает `rvalue` (то, что находится справа от присваивания). Из-за этого нельзя написать что-то типа такого:

```
int x = 1;  
int n = ++x++; // Ошибка
```

Казалось бы, сперва должен выполниться постфиксный инкремент (см. далее приоритеты операций), а затем к нему примениться префиксный. Но так как постфиксный инкремент возвращает `rvalue`, то применить к нему префиксный инкремент компилятор не может. И скобки тут не помогут, так как проблема не в порядке вычисления или группировке операторов.

С операторами инкремента и декремента нужно проявлять особую внимательность и осторожность. Допустим, нам необходимо вывести на экран 3 раза фразу `Hello world`:

```
int n = 3;  
while(--n) // Неправильно!  
    printf("Hello world\n");
```

Данный код выведет фразу только 2 раза. А вот с постфиксной формой код отработает правильно.

```
int n = 3;  
while(n--) // Ok  
    printf("Hello world\n");
```

Бытует мнение, что префиксная форма быстрее постфиксной за счет того, что не делает копии значения, необходимого для возврата. То есть, например, вычисление факториала следующим образом будет немного быстрее:

```

unsigned long long factorial(unsigned int n) {
    unsigned long long result = 1;
    for (unsigned int i = 2; i <= n; ++i) { // Быстрее чем i++???
        result *= i;
    }
    return result;
}

```

Для простых типов никакой разницы не будет, причем даже в debug-сборке без оптимизаций:

```
add     DWORD PTR [rbp-12], 1
```

А вот в C++ особенно с использованием перегрузки операторов `++` и `--` разница может быть заметной.

Операторы сравнения

Операторы сравнения (Comparison operators) - бинарные операторы, которые проверяют условие и возвращают **1**, если это условие логически **истинно** (true), и **0**, если это условие **ложно** (false).

Обратите внимание, в языке С возвращается результат типа `int`, а не `bool` или `_Bool`, тогда как в C++ для встроенных типов возвращается результат типа `bool`.

Разделяются на операторы проверки равенства (equality operators: `==`, `!=`) и отношения (relational operators: `<`, `>`, `<=`, `>=`)

Оператор	Название	Пример	Описание
<code>==</code>	Равно	<code>a == b</code>	Проверяет, равны ли значения двух operandов
<code>!=</code>	Не равно	<code>a != b</code>	Проверяет, не равны ли значения двух operandов
<code><</code>	Меньше	<code>a < b</code>	Проверяет, меньше ли значение левого operandса, чем правого
<code>></code>	Больше	<code>a > b</code>	Проверяет, больше ли значение левого operandса, чем правого

Оператор	Название	Пример	Описание
<code><=</code>	Меньше или равно	<code>a <= b</code>	Проверяет, меньше или равно значение левого операнда правому
<code>>=</code>	Больше или равно	<code>a >= b</code>	Проверяет, больше или равно значение левого операнда правому

Логические операторы

Логические операторы выполняют стандартные операции булевой алгебры над операндами. Результатом является тип `int`.

Оператор	Название	Пример	Результат
<code>!</code>	логическое НЕ (NOT)	<code>!a</code>	логическое НЕ для <code>a</code>
<code>&&</code>	логическое И (AND)	<code>a && b</code>	логическое И для <code>a</code> и <code>b</code>
<code> </code>	логическое ИЛИ (OR)	<code>a b</code>	логическое ИЛИ для <code>a</code> и <code>b</code>

Обратите внимание, что следующие конструкции не являются эквивалентными: `!(x && y)` и `!x && !y`. Аналогично не являются эквивалентными: `!(x || y)` и `!x || !y`.

Для таких случаев следует применять Правила де Моргана (De Morgan^[1]):

- Отрицание конъюнкции есть дизъюнкция отрицаний.

$$!(x \&\& y) == !x \mid\mid !y$$

- Отрицание дизъюнкции есть конъюнкция отрицаний.

$$!(x \mid\mid y) == !x \&\& !y$$

С точки зрения производительности нет разницы (оптимизатор любезно сделает все за вас) какую форму выбирать, в скобках или без. С точки зрения читаемости лучше использовать бесскобочный вариант: `!x || !y` или `!x && !y`.

Операторы доступа к членам

Операторы доступа к члену (Member access operators) позволяют обращаться к членам своих operandов:

Оператор	Название	Пример	Описание
[]	индекс массива	a[b]	доступ к b -му элементу массива a
*	разыменование указателя	*a	разыменовать указатель на a для доступа к объекту или функции, на которые он ссылается
&	взятие адреса	&a	создать указатель, который ссылается на объект или функцию
.	доступ к члену	a.b	доступ к члену b структуры или объединения a
->	доступ к члену через указатель	a->b	доступ к члену b структуры или объединения на которое указывает a

Прочие операторы

Ниже перечислены операторы не подпадающие под вышеперечисленные категории.

Оператор	Название	Пример	Описание
(...)	В вызов функции	f(...)	вызвать функцию f() с аргументами или без
,	Оператор запятая	a, b	вычислить выражение a , проигнорировать его возвращаемое значение, но выполнить все побочные эффекты, затем вычислить выражение b , возвращая тип и результат этого вычисления
(type)	Приведение типа	(type)a	привести тип a к type
? :	Условный оператор	a ? b : c	если a логически истинно (не равно нулю), то вычислить выражение b , в противном случае вычислить выражение c

Оператор	Название	Пример	Описание
<code>sizeof</code>	Оператор sizeof	<code>sizeof a</code>	размер a в байтах
<code>_Alignof</code>	(C11) Оператор выравнивания	<code>_Alignof(type)</code>	выравнивание, требуемое для type
<code>typeof</code>	Оператор типа	<code>typeof(a)</code>	тип a

Приоритет операций

Ниже перечислены приоритеты (в порядке убывания) и ассоциативность операторов языка С.

Приоритет	Оператор	Описание	Ассоциативность
1	<code>++</code> <code>--</code>	Постфиксный инкремент и декремент	Слева направо
	<code>()</code>	Вызов функции	
	<code>[]</code>	Доступ к массиву по индексу	
	<code>.</code>	Доступ к членам структуры/объединения	
	<code>-></code>	Доступ к членам структуры/объединения через указатель	
	<code>(type)</code> <code>{list}</code>	Составной литерал (C99)	
2	<code>++</code> <code>--</code>	Префиксный инкремент и декремент	Справа налево
	<code>+</code> <code>-</code>	Унарный плюс и минус	
	<code>!</code> <code>~</code>	Логическое НЕ и побитовое НЕ	
	<code>(type)</code>	Приведение к типу	
	<code>*</code>	Разыменование	
	<code>&</code>	Взятие адреса	
	<code>sizeof</code>	Получение размера	
	<code>_Alignof</code>	Требования выравнивания (C11)	

Приоритет	Оператор	Описание	Ассоциативность
3	* / %	Умножение, деление и получение остатка	Слева направо
4	+ -	Сложение и вычитание	
5	<< >>	Побитовый сдвиг влево и вправо	
6	< <=	Операторы отношения меньше, меньше или равно	
	> >=	Операторы отношения больше, большие или равно	
7	== !=	Отношение равно и не равно	
8	&	Побитовое И	
9	^	Побитовое исключающее ИЛИ	
10		Побитовое ИЛИ	
11	&&	Логическое И	
12		Логическое ИЛИ	
13	? :	Тернарный оператор условия	Справа налево
14	=	Простое присваивание	
	+= -=	Присваивание со сложением и разницей	
	*= /= %=	Присваивание с умножением, делением и вычислением остатка	
	<<= >>=	Присваивание со сдвигом влево и вправо	
	&= ^= =	Присваивание с побитовым И, Исключающим ИЛИ и ИЛИ	
15	,	Запятая	Слева направо

Запомнить эту таблицу довольно сложно, поэтому обычно в любой непонятной ситуации используют круглые скобки.

```
int a = 5, b = 10;
int max = a < b ? a : b + 1; // Ошибка
```

В примере, допустим, ожидается сложение 1 с результатом тернарного оператора. Для решения проблемы можно использовать скобки:

```
int a = 5, b = 10;
int max = (a < b ? a : b) + 1; // Ok
```

Порядок вычислений

Часто на собеседованиях просят решить что-то типа такого:

```
int x = 1;
int n = x++ + ++x; // Чему равно n?
```

Проблема в том, что стандарт не регламентирует порядок вычисления. Что первым выполнится, `x++` или `++x` нельзя заранее предугадать. Поэтому данный пример является типичным **неопределенным поведением** (undefined behavior, UB).

Для того, чтобы результат стал надежным, необходимо вынести операции инкремента за пределы всей инструкции:

```
int x = 1;
int n = x++; // x = 2, n = 1
n += ++x;    // x = 3, n = 4
```

Выражения

Это последовательность операторов и их operandов, которая при вычислении возвращает значение определенного типа.

Инструкция

Инструкция (statement) определяет действие, которое должно быть выполнено (A statement specifies an action to be performed). Большинство инструкций в С заканчиваются символом `'.'`. Инструкции выполняются последовательно (это требование

распространяется на все императивные языки программирования). Таким образом, любая программа представляет собой последовательность выполняемых инструкций.

В языке С определены следующие инструкции:

- метка (labeled-statement);
- составная инструкция (compound-statement);
- инструкция-выражение (expression-statement);
- инструкция выбора (selection-statement);
- циклы (iteration-statement);
- инструкция перехода (jump-statement).

Метка

Синтаксис:

```
identifier : statement          (1)
case constant-expression : statement (2)
default : statement           (3)
```

(1) идентификатор (должен быть уникальным внутри функции) метки используется в инструкции `goto`.

(2-3) используются только в инструкции выбора `switch`

Пример:

```
#include <stdio.h>

int main(int argc, char** argv)
{
    if (argc < 2)
        goto error;
    printf("%s\n", argv[1]);
    goto exit;
error:
    printf("error\n");
exit:
    return 0;
}
```

В этом примере проверяется количество аргументов программы и если аргументов нет, то выводится сообщение "error", иначе выводится на экран первый аргумент.

Можно запустить программу при помощи команды:

```
./goto "hello, world"
```

На экран будет выведено:

```
hello, world
```

Составная инструкция

Блок кода, заключенный в фигурные скобки {}, который внутри себя имеет другие инструкции, называется составной (compound) инструкцией.

Синтаксис:

```
{  
    declaration  
    statement  
}
```

statement в данном случае может быть также составным и сколько угодно вложенными.

В противном случае инструкция является простой (simple).

Инструкция-выражение и нулевая инструкция

Инструкция-выражение (expression-statement) - это выражение, за которым следует точка с запятой ; . При этом значение выражения вычисляется, но игнорируется.

Синтаксис:

```
expression;
```

Выражение - это последовательность операторов (operators) и operandов (operands), которая:

- вычисляет значение;
- ссылается на объект или функцию;

- генерирует побочные эффекты (side effects), например, функция printf выводит текст на экран;
- или выполняет комбинацию выше перечисленных действий.

Оператор (operator) - это специальный символ или ключевое слово, которые выполняют действия над переменными или значениями (операндами).

Выделяют следующие операторы:

- арифметические;
- сравнения (реляционные);
- логические;
- битовые;
- присваивания;
- тернарный (условный);
- размера и выравнивания типа;
- для работы с адресами, указателями и полями структур;
- оператор запятая;
- приведения типа.

Нулевая инструкция ничего не делает, обозначается точкой с запятой ; .

Примеры:

```
// простые выражения (бессмысленные, но допустимые)
5;      // Число 5 (не имеет эффекта)
x;      // Чтение переменной x (без побочных эффектов ничего не делает)
x + y; // Сумма x и y (результат отбрасывается)

// вызов функции
printf("hello, world\n"); // Выражение с побочным эффектом (вывод в консоль)

// присваивание
x = 10;      // Присваивание (меняет значение x)
y = x * 2 + 1; // Вычисление и присваивание

// Инкремент/декремент
x++;      // Увеличивает x на 1 (возвращает старое значение)
--y;      // Уменьшает y на 1 (возвращает новое значение)
```

```
// Составные инструкции присваивания  
x += 5; // Эквивалентно x = x + 5  
y *= 2; // Эквивалентно y = y * 2  
  
// Пустая инструкция  
; // Пустое выражение (иногда используется в циклах)
```

Инструкция выбора

Инструкцию выбора (selection-statement) еще называют инструкцией ветвления.

Синтаксис:

```
if (expression) statement          (1)  
if (expression) statement else statement (2)  
switch (expression) statement      (3)
```

(1-2) Условные инструкции

(3) Инструкция выбора

Примеры:

```
int number = 10;  
  
if (number > 0) {  
    printf("Число положительное\n");  
} else {  
    printf("Число отрицательное или ноль\n");  
}
```

```
switch (day) {  
    case 1:  
        printf("Понедельник\n");  
        break;  
    case 2:  
        printf("Вторник\n");  
        break;
```

```
case 3:  
    printf("Среда\n");  
    break;  
default:  
    printf("Неизвестный день\n");  
}
```

Последний пример с инструкцией выбора switch можно переписать с использованием инструкции if

```
if (day == 1) {  
    printf("Понедельник\n");  
} else if (day == 2) {  
    printf("Вторник\n");  
} else if (day == 3) {  
    printf("Среда\n");  
} else {  
    printf("Неизвестный день\n");  
}
```

Возникает вопрос, что лучше использовать и в каких случаях?

Инструкция if использует последовательный перебор всех условий, в то время как switch предназначен для быстрого прыжка на нужную ветку кода в случае, когда используется проверка значения одной конкретной переменной (причем можно использовать только перечислимые типы - enum, int, char и т.п., но не строки или другие составные типы). То есть switch - это оптимизированный вариант if, проверяющего равенство значений.

Здесь нужно задуматься, а как в switch происходит этот магический прыжок (в сущности goto) в нужное место?

Компилятор создает что-то вроде таблицы (можно представлять ее как массив пар) значение - адрес для прыжка. Когда значения лежат плотно друг к другу без разрывов, то такой код выполняется крайне эффективно (за $O(1)$), а если не плотно, тогда используется бинарный поиск (за $O(\log n)$). По этой причине лучше использовать неразрывные диапазоны значений в enum.

Если в switch используется не более 4 констант, то компилятор его перепишет на обычные if-else, так как это эффективнее, чем создавать таблицу прыжков для такого случая.

Поэтому рекомендация может выглядеть так: если вы используете не более 4 проверок - используйте if, иначе рассмотрите вариант со switch.

Циклы

Циклические инструкции (iteration-statement) выполняют повторно инструкцию, называемую телом цикла, до тех пор пока управляющее выражение не равно 0.

Синтаксис:

```
while (expression) statement          (1)
do statement while (expression);      (2)
for (expression{opt}; expression{opt}; expression{opt}) statement (3)
for (declaration expression{opt}; expression{opt}) statement      (4)
```

- (1) Вычисление управляющего выражения выполняется перед каждым выполнением тела цикла.
- (2) Вычисление управляющего выражения происходит после каждого выполнения тела цикла.

Примеры:

```
// Цикл с предусловием
// Вывести числа от 1 до 5
int i = 1;

while (i <= 5) {
    printf("%d ", i);
    i++;
}

// Цикл с постусловием
// Ввод пароля
int password = 1234;
int input;

printf("Введите пароль: ");
scanf("%d", &input);
```

```
while (input != password) {  
    printf("Неверный пароль! Попробуйте снова: ");  
    scanf("%d", &input);  
}  
  
printf("Доступ разрешен!\n");  
  
// Цикл for  
// Вывод чисел от 0 до 4  
for (int i = 0; i < 5; i++) {  
    printf("%d ", i);  
}
```

Инструкция перехода

Синтаксис:

```
goto identifier;          (1)  
continue;                (2)  
break;                  (3)  
return expression{opt}; (4)
```

Инструкция `goto` выполняет безусловный переход на указанную метку.

-
1. Огастес де Морган (1806 - 1871) - шотландский ученый-математик, профессор математики в Университетском колледже Лондона. Известен своими трудами по математической логике и теории рядов. В его честь названы законы (законы де Моргана) теоретико-множественных соотношений. ↩

13. Препроцессор и макросы

Препроцессор подготавливает код перед его компиляцией. Его поведением (что именно нужно сделать) управляют директивы препроцессора, начинающиеся с символа `#`, при этом `;` в конце, как у инструкций (statements), не требуется.

Синтаксис:

```
#инструкция
```

Директивы можно разделить на следующие группы:

- подстановка
- включение
- условная компиляция
- вызов ошибки или предупреждения
- нестандартные директивы

Подстановка

Выполняется через **определение** макросов - правил подстановки текста.

Синтаксис:

```
#define identifier replacement-list{opt}          (1)
#define identifier(parameters) replacement-list{opt}   (2)
#define identifier(parameters, ...) replacement-list{opt} (3) (since C99)
#define identifier(...) replacement-list{opt}           (4) (since C99)
#undef identifier
```

1. простая замена текста, либо введение пустого идентификатора

```
#define PI 3.14159
#define MAX_SIZE 100

int main() {
    double area = PI * 5 * 5; // Препроцессор заменит PI на 3.14159
```

```
int array[MAX_SIZE]; // Препроцессор заменит MAX_SIZE на 100
}
```

Таким образом можно вводить константы, хотя в современном коде лучше использовать настоящие константы (глобальные или локальные).

2. макрос-функция использует параметры

```
#define SQUARE(x) ((x) * (x))
#define MAX(a, b) ((a) > (b) ? (a) : (b))

int main() {
    int result = SQUARE(5); // = ((5) * (5));
    int max_val = MAX(10, 20); // = ((10) > (20) ? (10) : (20));
}
```

Важно: чтобы избежать ошибок при подстановке, следует использовать скобки вокруг аргументов и всего выражения, иначе могут возникнуть проблемы с приоритетом операторов.

3, 4) Вариативные макросы (C99) работают подобно вариативным функциям

```
#include <stdio.h>

#define DEBUG_PRINT(fmt, ...) printf("[DEBUG] " fmt, __VA_ARGS__)

int main() {
    DEBUG_PRINT("Values: %d, %d, %d\n", 1, 2, 3);
    return 0;
}
```

`__VA_ARGS__` позволяет выполнять подстановку переменных аргументов, соответствующих многоточию в макросе.

Пример с пустыми аргументами:

```
#define DEBUG(...) fprintf(stderr, __VA_ARGS__)
```

```
DEBUG("Error occurred\n");
DEBUG("Value: %d\n", value);
```

Стандарт предписывает передачу не менее одного параметра в многоточие. Иначе возникает проблема:

```
#define LOG(format, ...) printf(format, __VA_ARGS__)

LOG("Hello"); // Ошибка! Раскрывается в: printf("Hello", )
```

Для решения этой проблемы в C23 ввели `__VA_OPT__`. Он раскрывается в свое содержимое только если `__VA_ARGS__` не пуст.

Синтаксис:

```
#define MACRO(...) __VA_OPT__(content)
```

Пример:

```
#define LOG(format, ...) printf(format __VA_OPT__(,) __VA_ARGS__)

LOG("Hello");           // Раскрывается в: printf("Hello")
LOG("Value: %d", 42); // Раскрывается в: printf("Value: %d", 42)
```

5. `#undef` отменяет определенный ранее макрос

```
#define DEBUG_MODE 1
// ... какой-то код ...
#undef DEBUG_MODE
// Теперь использовать DEBUG_MODE здесь нельзя
```

Многострочные макросы используют символ `\` в конце строки

```
// Многострочный макрос для безопасного выделения памяти
#define SAFE_MALLOC(ptr, type, size) \
    do { \
        ptr = (type*)malloc((size) * sizeof(type)); \

```

```

    if (ptr == NULL) { \
        fprintf(stderr, "ОШИБКА выделения памяти в файле %s, строка %d\n",
    \
            __FILE__, __LINE__); \
        exit(EXIT_FAILURE); \
    } \
    printf("Память выделена: %zu байт [Файл: %s, Стока: %d]\n", \
        (size) * sizeof(type), __FILE__, __LINE__); \
} while(0)

// Многострочный макрос для безопасного освобождения памяти
#define SAFE_FREE(ptr) \
do { \
    if (ptr != NULL) { \
        free(ptr); \
        ptr = NULL; \
        printf("Память освобождена [Файл: %s, Стока: %d]\n", \
            __FILE__, __LINE__); \
    } \
} while(0)

```

Важно! В макросах следует каждый параметр, а также все выражение обрамлять в скобки для предотвращения ошибок при подстановке и контроля приоритета операций.

Например:

```

#define SQUARE(x) x * x

int result = SQUARE(2 + 3); // 2 + 3 * 2 + 3 = 11 (а не 25!)

```

Правильно:

```

#define SQUARE(x) ((x) * (x))

int result = SQUARE(2 + 3); // → ((2 + 3) * (2 + 3)) = 25

```

Или может быть следующий эффект:

```
#define MAX(a, b) a > b ? a : b

int x = 1, y = 2;
int z = MAX(x++, y++); // → x++ > y++ ? x++ : y++
// x и y увеличиваются дважды!
```

Правильно:

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

Также будьте аккуратны с многострочными макросами:

```
#define PRINT_SUM(a, b) \
    printf("Sum: "); \
    printf("%d\n", a + b)

if (condition)
    PRINT_SUM(x, y); // → только первый printf в if!
```

Операторы # и

Оператор # (stringizing operator) преобразует параметр макроса в строковый литерал (препроцессор обрамляет параметр в двойные кавычки).

```
#define DEBUG_PRINT(expr) printf(#expr " = %d\n", expr)

int main() {
    int x = 5;
    int y = 10;

    DEBUG_PRINT(x);          // printf("x" " = %d\n", x)
    DEBUG_PRINT(x + y);      // printf("x + y" " = %d\n", x + y)
    DEBUG_PRINT(x * y);      // printf("x * y" " = %d\n", x * y)

    return 0;
}
```

Оператор `##` (token pasting operator) объединяет (склеивает) два токена в один.

```
// Создание набора переменных с автоматической нумерацией
#define DECLARE_VAR(n) int var##n = n

// Создание функций с похожими именами
#define CREATE_GETTER(type, name) \
    type get_##name() { return name; }

typedef struct {
    int width;
    int height;
} Rectangle;

CREATE_GETTER(int, width)
CREATE_GETTER(int, height)

int main() {
    DECLARE_VAR(1); // int var1 = 1;
    DECLARE_VAR(2); // int var2 = 2;
    DECLARE_VAR(3); // int var3 = 3;

    printf("%d %d %d\n", var1, var2, var3);

    Rectangle rect = {10, 20};
    printf("Width: %d, Height: %d\n", get_width(), get_height());

    return 0;
}
```

В языке имеется ряд предопределенных макро имен:

- `__DATE__` - дата трансляции модуля (строка в формате "MMM DD YYYY")
- `__TIME__` - время трансляции модуля (строка в формате "HH:MM:SS")
- `__FILE__` - имя исходного файла (строка)
- `__LINE__` - текущий номер строки в файле (целое число)
- `__func__` - имя текущей функции (C99)

```
// Макрос для отладочного вывода
#define DEBUG_PRINT(msg) \
    printf("DEBUG: %s [Файл: %s, Стока: %d]\n", msg, __FILE__, __LINE__)
```

`__STDC__` - равен 1, если компилятор соответствует стандарту языка
`__STDC_VERSION__` - версия стандарта языка C:

- 199409L - C94
- 199901L - C99
- 201112L - C11
- 201710L - C17
- 202311L - C23

```
#if defined(__STDC_VERSION__)
    #if __STDC_VERSION__ >= 201112L
        printf("C11 или новее\n");
    #elif __STDC_VERSION__ >= 199901L
        printf("C99\n");
    #elif __STDC_VERSION__ >= 199409L
        printf("C94\n");
    #else
        printf("C89/C90\n");
    #endif
#else
    printf("До C89\n");
#endif
```

Для проверки операционной системы и архитектуры можно воспользоваться следующими макроименами:

```
#include <stdio.h>

// Проверка операционной системы
#ifndef _WIN32
    #define OS "Windows"
#elif __linux__
    #define OS "Linux"
#elif __APPLE__
    #define OS "Mac OS X"
```

```

#define OS "macOS"
#else
#define OS "Unknown"
#endif

// Проверка архитектуры процессора
#ifdef __x86_64__
#define ARCH "x86-64"
#elif __i386__
#define ARCH "x86"
#elif __arm__
#define ARCH "ARM"
#else
#define ARCH "Unknown"
#endif

int main() {
    printf("Операционная система: %s\n", OS);
    printf("Архитектура: %s\n", ARCH);

    // Проверка отладочного режима
    #ifdef DEBUG
    printf("Отладочный режим включен\n");
    #else
    printf("Режим релиза\n");
    #endif

    return 0;
}

```

Компиляторы могут использовать свои макроопределения.

```

__GNUC__           // Версия GCC
__GNUC_MINOR__    // Минорная версия GCC
__GNUC_PATCHLEVEL__ // Патч-версия GCC

_MSC_VER          // Версия Microsoft C/C++ компилятора

```

Включение

Директива `#include` выполняет вставку (включает) текстового содержимого указанного файла и имеет две формы:

1. Для системных заголовочных файлов

```
#include <path-spec>
```

1. Для несистемных заголовочных файлов

```
#include "path-spec"
```

Условная компиляция

Условная компиляция позволяет включать или исключать части кода на этапе препроцессинга.

```
#if / #elif / #else / #endif
```

```
#if условие1
    // код, если условие1 истинно
#elif условие2
    // код, если условие2 истинно
#elif условие3
    // код, если условие3 истинно
#else
    // код, если все условия ложны
#endif
```

Для проверки условия часто используется оператор `defined`

```
#if defined MACRO      // без скобок
#if defined(MACRO)      // со скобками
#if defined(MACRO1) && defined(MACRO2) // комбинации
```

Например:

```
#if defined(WINDOWS) && defined(X64)
    printf("64-bit Windows\n");
#endif
```

Также можно использовать отрицание

```
#if !defined(RELEASE)
    printf("Not release version\n");
#endif
```

Для простых проверок следует использовать более короткие альтернативы

`#ifdef` - более короткая запись `#if defined`
`#ifndef` - более короткая запись `#if !defined`

Например, ранее широко распространенный способ защититься от повторного включения заголовочного файла - `#include guards`:

```
#ifndef MYHEADERFILE_H
#define MYHEADERFILE_H

// содержимое заголовочного файла

#endif
```

В C23 добавили новые директивы - `#elifdef`, `#elifndef`:

```
#ifdef WINDOWS
    // Windows код
#elifdef LINUX      // C23
    // Linux код
#elifndef MACOS     // C23
    // Не macOS код
#endif
```

Генерация ошибки и предупреждения

Директива `#error` позволяет выводить ошибку во время компиляции. Например:

```
#if __STDC_VERSION__ < 201112L
#error "C11 or newer required"
#endif
```

C23 добавил директиву `#warning`, которая раньше поддерживалась в gcc и clang.

```
#warning "Сообщение с предупреждением"
```

До C23 использовались различные кроссплатформенные конструкции:

```
#if __STDC_VERSION__ >= 202311L
    #warning "C23 standard warning"
#elif defined(__GNUC__) || defined(__clang__)
    #warning "Compiler extension warning"
#elif defined(_MSC_VER)
    #pragma message("MSVC warning")
#endif
```

Директива `#pragma`

`#pragma` используется для передачи компилятору специальных инструкций, зависящих от реализации.

```
#pragma directive_name [параметры]
```

Если компилятор не знает инструкцию, он ее игнорирует (обычно без выдачи ошибки).

`#pragma` - довольно мощный инструмент для тонкой настройки компилятора, но требует осторожного использования из-за отличий поддержки в разных компиляторах.

Распространенные инструкции:

```
#pragma once // защита от множественного включения

#pragma message("Компилируется этот файл") // сообщение в MSVC

// управление выравниванием
#pragma pack(push, 1)      // сохраняем текущее выравнивание, устанавливаем 1
```

```

struct TightStruct {
    char a;
    int b; // будет занимать 5 байт вместо 8
};

#pragma pack(pop) // восстанавливаем предыдущее выравнивание

// управление предупреждениями (MSVC)
#pragma warning(disable: 4996) // отключаем предупреждение 4996
#pragma warning(enable: 4996) // включаем предупреждение 4996

// управление предупреждениями (GCC, Clang)
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wunused-variable"
int unused_var; // не будет предупреждения
#pragma GCC diagnostic pop

// инструкции линковщика (MSVC)
#pragma comment(lib, "user32.lib") // линкуем библиотеку
#pragma comment(linker, "/OPT:REF") // опции линковщика

```

В C23 появился `_Pragma` оператор, который можно использовать в макросах:

```

#define DISABLE_WARNING _Pragma("GCC diagnostic ignored \\"-Wunused\\\"")

DISABLE_WARNING
int unused_var; // без предупреждения

```

14. Соглашения о вызовах

При работе с различными API можно столкнуться с так называемыми соглашениями о вызовах (Calling Conventions), которые определяют:

- Как передаются аргументы в функцию (через стек или регистры).
- Кто очищает стек после вызова (вызывающая или вызываемая функция).
- Как возвращаются значения.
- Какие регистры сохраняются, а какие могут изменяться.

Среди наиболее часто используемых можно выделить:

- cdecl (используется по-умолчанию в языке C)
- stdcall (используется в WinAPI)
- fastcall (передача аргументов через регистры)
- thiscall (используется для методов классов в C++)
- vectorcall (оптимизация для SIMD)

Для языка C принято соглашение `cdecl`, которое необязательно явно указывать:

```
int __cdecl add(int a, int b); // cdecl явно указан
int add(int a, int b);        // по-умолчанию в C используется cdecl
```

`cdecl` :

- Аргументы передаются **через стек** (справа налево, т.е. последний аргумент кладётся первым).
- **Вызывающая функция** очищает стек (поэтому поддерживаются функции с переменным числом аргументов, например, `printf`).
- Имя функции в ассемблерном коде не изменяется (не декорируется).

Когда разработчики Windows столкнулись с выбором соглашения о вызовах для WinAPI, а в то время (середина 1980-х) самым популярным языком разработки был Pascal, в котором соглашения были прямо противоположны `cdecl`, они выбрали гибридный вариант и назвали его `__stdcall` :

- Аргументы передаются **через стек** (справа налево, как в `cdecl`).

- **Вызываемая функция** сама очищает стек (поэтому нельзя использовать `va_args`).
- Имя функции в ассемблере декорируется (например, `_func@8`, где `8` — размер аргументов в байтах).

Плюсы: быстрее, чем `cdecl`, т.к. нет лишних инструкций очистки стека.

```
__declspec(dllexport) // указывает, что функция импортируется из другой dll
int
__stdcall
MessageBoxW(
    HWND hWnd,
    LPCWSTR lpText,
    LPCWSTR lpCaption,
    UINT uType);
```

`__fastcall`:

- Первые 2–3 аргумента передаются **через регистры** (обычно `ECX`, `EDX` в x86, `RCX`, `RDX`, `R8`, `R9` в x64).
- Остальные аргументы передаются через стек как в `__stdcall`.
- **Вызываемая функция** очищает стек.

Плюсы: быстрее, чем `cdecl` и `__stdcall`, т.к. минимизирует работу со стеком. В GCC/Clang используется аналог `__attribute__((fastcall))`.

`thiscall` используется в C++:

- Используется для **методов классов** в C++.
- В **x86**:
 - Указатель `this` передаётся в регистре `ECX`.
 - Остальные аргументы — через стек.
- В **x64**:
 - `this` передаётся в `RCX`, остальные аргументы — в `RDX`, `R8`, `R9`, затем стек.

```
class MyClass {
public:
    int __thiscall add(int a, int b); // Обычно не указывается явно
};
```

`__vectorcall`:

- Аргументы **векторных типов** (`float`, `double`, SIMD) передаются через регистры `XMM0 – XMM5`.
- Остальные аргументы — как в `__fastcall`.

Поддерживается в MSVC и некоторых других компиляторах. Используется в высокопроизводительных вычислениях.

`__clrcall` - это соглашение вызова, специфичное для **управляемого кода** в C++/CLI. Оно используется для вызова управляемых функций (функций, работающих в среде CLR - Common Language Runtime) из других управляемых функций. Его главная цель - обеспечить эффективный вызов в контексте .NET. Важно отметить, что функции компилируются не в машинный код, а в IL (Intermediate Language - промежуточное представление, понятое виртуальной машиной .NET). Кроме того, `__clrcall` использует не нативный стек, а стек виртуальной машины.

15. Опережающее объявление

Опережающее объявление (forward declarations) - это особый вид объявлений структуры (чаще всего в заголовочном файле) без ее полного определения, которое позволяет использовать указатели на типы до их полного описания, что позволяет разорвать циклические ссылки и сократить количество включений заголовочных файлов.

Пример:

```
// user.h

#include "profile.h" // Содержит определение Profile

struct User {
    char name[50];
    struct Profile* profile; // Нужен только указатель
};

void initializeUser(struct User* user, struct Profile* prof); // И здесь
```

```
// profile.h

#include "user.h" // Содержит определение User

struct Profile {
    int age;
    struct User* user; // Нужен только указатель
};

void updateProfile(struct Profile* prof, struct User* user);
```

В итоге мы имеем циклическое включение. Опережающее объявление позволяет разорвать эту зависимость:

```
// user.h
```

```

// НЕ включаем profile.h
struct Profile; // Forward declaration вместо включения

struct User {
    char name[50];
    struct Profile* profile; // Указатель – достаточно forward declaration
};

void initializeUser(struct User* user, struct Profile* prof);

```

```

// profile.h

// НЕ включаем user.h
struct User; // Forward declaration вместо включения

struct Profile {
    int age;
    struct User* user; // Указатель – достаточно forward declaration
};

void updateProfile(struct Profile* prof, struct User* user);

```

Включения необходимо перенести в .c-файлы:

```

// user.c
#include "user.h"
#include "profile.h" // Включаем ТОЛЬКО где нужно полное определение

void initializeUser(struct User* user, struct Profile* prof) {
    user->profile = prof;
    // Теперь можно работать с полями Profile
}

```

```

// profile.c

#include "profile.h"
#include "user.h" // Включаем ТОЛЬКО где нужно полное определение

```

```
void updateProfile(struct Profile* prof, struct User* user) {  
    prof->user = user;  
    // Теперь можно работать с полями User  
}
```

Преимущества такого подхода:

- устранение циклических зависимостей
- сокращение времени компиляции (меньшее количество включений заголовочных файлов)
- чистая архитектура (меньше связности между модулями)

16. Общие сведения о модулях

Разбиение кода на части существовало еще до появления языка С, например, в FORTRAN II, ALGOL68 и других, в которых для экономии памяти (которая в 60-х годах была крайне ценным и ограниченным ресурсом) применялась раздельная компиляция.

Со временем разработчики осознали важность разбиения кода с точки зрения архитектурных решений. Вместо огромного монолитного кода намного удобнее работать с набором файлов, особенно при командной разработке, когда разные разработчики могут работать со своими отдельными файлами не мешая друг другу (на самом деле, так редко получается в современном мире, но тем не менее это сильно упрощает выполнять слияние изменений).

Разработанные модули можно многократно переиспользовать в разных проектах (по аналогии с использованием одной и той же функции в разных местах программы). При этом важно логически связанные функции и данные группировать в одном файле, избегая зависимостей от других файлов.

В 1972 году Дэвид Парнас (David Parnas^[1]) описал принцип скрытия информации, что стало прорывом в распространении модульной разработки:

Модуль должен быть спроектирован так, чтобы скрывать от пользователя некоторое решение (детали реализации). Пользователю предоставляется только интерфейс, необходимый для использования модуля.

С появлением заголовочных файлов в языке С появилась возможность предоставлять "публичный" интерфейс и скрывать детали реализации.

За счет раздельной компиляции необязательно перекомпилировать каждый раз всю программу. Достаточно скомпилировать только измененный код. Это называется инкрементальной компиляцией.

Таким образом, разделение программ на .c файлы (модули трансляции) преследует следующие цели:

- Управление сложностью;
- Скрытие реализации;
- Повторное использование;
- Ускорение компиляции;

- Упрощение командной работы.

При разделении кода следует руководствоваться следующими принципами:

- Разделение интересов (Separation of Concerns) - каждый файл должен содержать код, относящийся к одной области ответственности, чтобы изменения в одной части кода не влияли на другие.
- Принцип единственной ответственности (Single Responsibility Principle, SRP) - модуль должен иметь только одну причину для изменения.
- Уменьшение размера сущностей (модулей) через стратегию "Разделяй и властвуй" - модули должны быть компактными и иметь четкое назначение.

Эти и другие принципы детально описал Роберт Мартин (Robert C. Martin^[2]) в своих книгах "Чистая архитектура" и "Чистый код".

1. Дэвид Парнас - канадский ученый-программист, удостоенный множества наград, опубликовавший в декабре 1972 года статью "On the Criteria To Be Used in Decomposing Systems into Modules", в которой описал свой знаменитый принцип сокрытия информации. ↩
2. Роберт Мартин (также известный как Дядя Боб) - американский программист, консультант и автор книг в области разработки ПО. Один из авторов Agile-манифеста. ↩

17. Класс хранения

Класс хранения (storage class) в языке С определяет три ключевых аспекта переменной или функции:

1. **Где хранится** (память: стек/статическая/регистры)
2. **Как долго существует** (время жизни)
3. **Где видна** (область видимости)

В стандарте С определены следующие классы хранения:

Спецификатор	Описание
auto	Локальные переменные (используется по умолчанию, обычно опускается)
register	Рекомендует компилятору разместить переменную в регистре процессора
static	Сохраняет переменную/функцию в статической памяти, ограничивает область видимости
extern	Указывает, что переменная/функция определена в другом модуле трансляции
typedef	Создает псевдоним типа (формально является storage class specifier)

Рассмотрим подробно каждый.

1. `auto` (автоматическое хранение)

Локальные (также называемые автоматическими) переменные внутри функций, которые не участвуют в компоновке, по-умолчанию имеют спецификатор `auto`, который явно указывать необязательно. Но важно понимать, что полное определение переменной выглядит так:

```
void func() {
    auto int x = 42; // Эквивалентно int x = 42;
```

```
}
```

Переменная видна внутри блока {} и его вложенных блоках, создается на стеке при входе в блок и уничтожается при выходе из него.

2. `register` (регистровое хранение)

Используется с локальными переменными, не участвует в компоновке. `register` сообщает компилятору, что переменную стоит разместить в регистрах процессора, но компилятор имеет полное право проигнорировать эту просьбу (современные компиляторы так и поступают). Нельзя получить адрес такой переменной (оператор &)

```
void func() {  
    register int counter; // Может быть размещено в регистре  
}
```

Избегайте использования устаревшего спецификатора `register` в современном коде.

3. `static` (статическое хранение)

- для глобальных переменных, констант и функций указывает на внутреннюю компоновку, т.е. ограничивает область видимости имени текущим файлом (модулем трансляции)
- для локальных переменных сохраняет значение между вызовами функции. При этом инициализируется один раз при первом входе в функцию.

```
void counter() {  
    static int count = 0; // Сохранит значение между вызовами  
    ++count;  
    printf("%d", count);  
}  
// counter() → 1, counter() → 2, counter() → 3
```

4. `extern` (внешнее связывание)

Для глобальных переменных, констант и функций указывает на внешнюю компоновку, т.е. имена доступны из других модулей.

5. `typedef` (псевдоним типа)

Создает новые имена для типов:

```
typedef unsigned long ulong;  
ulong x = 1000;
```

Формально является спецификатором класса хранения, хотя на хранение не оказывает никакого влияния.

18. Внутренняя и внешняя компоновка

Компоновка (linkage), определяет, как имена (идентификаторы) видны и связываются между разными модулями программы. Она бывает двух видов: внутренняя и внешняя.

Примечание: говоря об external/internal linkage, многие часто употребляют вместо "компоновки" слова "линковка" или "связывание", что не является ошибкой. Поскольку "компоновка" является более формальным термином из стандартов C/C++, здесь и далее будет использоваться именно это слово.

Внешняя компоновка (external linkage)

Внешняя компоновка позволяет связывать идентификаторы между разными модулями (единицами трансляции) программы.

В языке Си глобальные переменные, константы и функции по-умолчанию доступны во всех модулях программы.

Примечание: в языке С++ для глобальных констант по-умолчанию устанавливается видимость только внутри модуля.

```
// file1.c
int g_var = 42; // Внешняя компоновка (можно использовать в других файлах)
const float SPEED_OF_LIGHT = 2.99792458e8f; // Внешняя компоновка
void foo(int bar); // Внешняя компоновка
```

Чтобы использовать имена из другого модуля, применяют extern-объявления. Ключевое слово extern указывает на внешнюю компоновку. Причем для функций указывать extern необязательно.

```
// file2.c
extern int g_var; // Сообщаем компилятору, что переменная определена в другом
файле
extern const float SPEED_OF_LIGHT; // Константа определена в другом файле
void foo(int bar); // Можно не указывать extern
```

Внутренняя компоновка (internal linkage)

Внутренняя компоновка явно запрещает связывание идентификаторов между разными модулями (единицами трансляции).

Чтобы глобальные переменные, константы и функции были доступны только внутри модуля, необходимо использовать ключевое слово static:

```
// file1.c
static int hidden_var = 10; // Переменная не видна в других модулях
static const float SPEED_OF_LIGHT = 2.99792458e8f; // Константа не видна в
других модулях
static void secret_func() { // Функция не видна в других модулях
    printf("Я в домике!");
}
```

Уменьшение количества внешних имен положительно влияет на время линковки, а также уменьшает вероятность конфликтов имен.

Если объявление функции написано с ключевым словом static, то и определение также должно быть со static. Можно опускать static в объявлении функции, если определение написано со static и находится в этом же модуле, но лучше так не делать и писать согласованные объявления и определения, т.к. это уже тонкости о которых неопытные разработчики могут и не знать.

Использование extern в заголовочных файлах

Если функция определена в одном файле, а используется в другом, то вместо того, чтобы многократно копировать их объявления в файлах, достаточно вынести объявление в .h, а определение оставить в .c. При этом не нужно в заголовочном файле писать extern, так как функции по-умолчанию имеют внешнюю компоновку.

Чтобы сделать доступными глобальные переменные или константы, необходимо в заголовочном файле объявить их через спецификатор класса хранения extern. При этом в модуле (файле .c) определение не должно содержать `extern`.

Вот пример, как в Си с использованием заголовочных файлов можно обращаться к переменным, функциям и константам из другого модуля.

```
// Заголовочный файл utils.h
#ifndef UTILS_H
#define UTILS_H
```

```
// Объявление глобальной переменной
extern int g_var;

// Объявление const-константы
extern const float SPEED_OF_LIGHT;

// Прототип функции
void foo(int bar);

#endif
```

```
// Реализация utils.c
#include "utils.h"
#include <stdio.h>

// Определение глобальной переменной
int g_var = 42;

// Определение const-константы
const float SPEED_OF_LIGHT = 2.99792458e8f;

// Реализация функции
void foo(int bar) {
    printf("foo: bar=%d, g_var=%d, speed=%.2e\n", bar, g_var, SPEED_OF_LIGHT);
}
```

```
// Главный файл main.c
#include <stdio.h>
#include "utils.h"

int main(void) {
    // Использование глобальной переменной
    printf("g_var = %d\n", g_var);
    g_var = 100;

    // Использование const-константы
    printf("Speed of light = %.2e m/s\n", SPEED_OF_LIGHT);
```

```
// Вызов функции
foo(123);

return 0;
}
```

Обратите внимание, что `extrn` из В заменили на `extern` для лучшей читаемости. `extrn` имел сокращенный вид записи из-за ограниченных ресурсов (18-битный PDP-7 имел всего 4 K words, т.е. около 9 КБ памяти).

Почему в заголовочных файлах для функций `extern` необязателен?

Как мы уже знаем, глобальные переменные, константы и функции по-умолчанию имеют внешнюю компоновку, но в заголовочных файлах почему-то переменные и константы обязательно требуют `extern`, а функции нет? Давайте, разберемся почему.

После работы препроцессора и подстановки текста из .h файла в .c файл в нашем примере мы получим следующее содержимое:

```
// utils.c
// текст файла stdio.h мы здесь не будем приводить )

extern int g_var;
extern const float SPEED_OF_LIGHT;
void foo(int bar);

int g_var = 42;
const float SPEED_OF_LIGHT = 2.99792458e8f;
void foo(int bar) {
    printf("foo: bar=%d, g_var=%d, speed=%e\n", bar, g_var, SPEED_OF_LIGHT);
}
```

```
// main.c
// текст файла stdio.h мы здесь не будем приводить )

extern int g_var;
extern const float SPEED_OF_LIGHT;
void foo(int bar);
```

```
int main(void) {  
// ...
```

В модуле utils.c мы получили три объявления и затем три определения, ранее объявленных идентификаторов.

В модуле main.c мы получили три объявления, которые сообщают, что где-то в другом модуле или ниже в том же модуле есть определения сущностей.

Если мы уберем из заголовочного файла extern, мы получим следующее:

```
// utils.c  
// текст файла stdio.h мы здесь не будем приводить ()  
  
int g_var;  
const float SPEED_OF_LIGHT;  
void foo(int bar);  
  
int g_var = 42;  
const float SPEED_OF_LIGHT = 2.99792458e8f;  
void foo(int bar) {  
    printf("foo: bar=%d, g_var=%d, speed=%.2e\n", bar, g_var, SPEED_OF_LIGHT);  
}
```

```
// main.c  
// текст файла stdio.h мы здесь не будем приводить ()  
  
int g_var;  
const float SPEED_OF_LIGHT;  
void foo(int bar);  
  
int main(void) {  
// ...
```

В модуле utils.c мы получили три предварительных определения (tentative definition) и затем три определения. Это легально и проблем здесь никаких нет.

В модуле main.c мы получили два определения (глобальная переменная g_var и константа SPEED_OF_LIGHT), которые по-умолчанию инициализированы нулем (т.к. в этом модуле не встретилась их инициализация) и объявление foo.

Переменные и константы без extern и инициализации являются либо предварительным определением, если позже в этом же файле встречается их инициализация, либо определением с инициализацией нулем. При этом происходит выделение памяти под эти объекты и запись в эту память нулей.

При наличии extern, переменные и константы являются объявлением. При этом сами объекты не создаются, т.е. память не выделяется, а компилятор понимает, что где-то в другом месте есть их определение.

А вот с функциями все просто. Как говорят в полиции: нет тела - нет дела! Без тела функция однозначно является объявлением, иначе - определением.

В итоге, файл main.c будет скомпилирован, но на этапе линковки мы получим ошибку: **множественное определение** (multiple definition)

Таким образом, мы обязаны указывать extern для переменных и констант в заголовочных файлах, чтобы не создавать дублирующие объекты в модулях, использующих эти заголовочные файлы. А для функций необязательно.

А что будет, если указать спецификатор extern у функции?

```
// utils.h

extern void foo(int bar);

// utils.c

extern void foo(int bar) {
    // ...
}
```

Ответ: ничего не изменится. Но повторим еще раз: **указывать extern для функций избыточно** и обычно никто так не пишет.

Необязательность указания extern для функций существовала с самых ранних версий си. Но плохое понимание или незнание этой особенности привело к тому, что весьма

распространенной практикой стало всегда указывать `extern`.

В K&R C, т.е. до первой стандартизации языка (C89) существовала еще одна проблема - объявления функций указывали без типов аргументов:

```
/* header.h */
extern int foo(); // На самом деле ожидает double!

/* file.c */
foo("hello"); // Передали char* вместо double. Ошибка только в рантайме.
```

Это приводило к трудноуловимым багам и в среде получило название "проклятие `extern`" (`extern declaration hell` или `extern linkage problem`).

Давайте разберем еще пару вопросов.

Что будет, если переменную и константу определить в заголовочном файле?

```
// utils.h

int g_var = 42;
const float SPEED_OF_LIGHT = 2.99792458e8f;
```

Ответ: будут созданы `extern` копии в каждом модуле, а значит на этапе линковки получим ошибку, рассмотренную выше - множественное определение.

Что будет, если определить глобальную переменную в заголовочном файле через `static`?

```
// utils.h

static int local_global = 10;
```

Да, можно определить `static` глобальную переменную в `.h`, но тогда у каждого `.c`-файла будет своя копия переменной, и обычно это не то, что хотел сделать программист, то есть это признак возможной ошибки, хотя программа при этом скомпилируется. То есть это не настоящая глобальная переменная, а скорее "модульно-локальная".

А что будет, если определить константу в заголовочном файле через `static`?

```
// utils.h

static const float SPEED_OF_LIGHT = 2.99792458e8f;
```

То же самое, в каждом .c файле будет своя локальная копия этой константы. Но в отличие от глобальных переменных это не создает никаких проблем, так как это неизменяемые данные. По этой причине, в C++ константы по-умолчанию имеют внутреннюю компоновку и им нужно принудительно задавать extern, чтобы сделать доступными в других модулях.

19. Заголовочные файлы

Язык В (предшественник языка С), разработанный Кеном Томпсоном, не имел заголовочных файлов. И при разделении кода на несколько модулей (файлов) возникала проблема объявления общих функций и переменных. Из-за того, что компилятор был однопроходной, функцию необходимо было определить до ее первого вызова в том же файле, либо через использование ключевого слова `extrn` которое использовалось для предварительного объявления и сообщало, что указанное имя где-то существует (или в этом же файле ниже или в другом модуле).

```
extrn bar; // Объявление функции bar

main() {
    bar();
}

bar() {    // Определение
    print("Hello!");
}
```



Обратите внимание, что параметры и возвращаемый тип не указывались.

При использовании двух и более файлов необходимо было объявлять также через `extrn` функции из другого модуля:

```
/* file1.b */
extrn foo; // Объявление: "foo определена где-то ещё"

main() {
    print(foo());
}

/* file2.b */
foo() {
    return 42;
}
```

Когда файлов становилось много, приходилось вручную копировать десятки имен функций и переменных в каждом файле через `extrn`, что часто приводило к ошибкам и усложняло поддержку кода. А с добавлением стандартной и других библиотек стало очевидно, что с этим нужно что-то делать.

Деннис Ритчи, разрабатывая язык Си, нашел решение этой проблемы в виде заголовочных файлов с расширением `.h`. Идея была заимствована из BCPL, где использовался `GET`, и PL/I, где использовался `%INCLUDE`, а также из ассемблера, где использовались "include-файлы" с макросами и константами.

Директива `#include` выполняет вставку текстового содержимого указанного файла.

Таким образом, заголовочные файлы помогают решить целый ряд важных задач:

1. Объявление функций, глобальных переменных, констант, структур, макросов и т.д., используемых в нескольких .с-файлах, но без дублирования кода в модулях. Это позволяет компилятору узнать о существовании этих сущностей до их фактического определения, а также проверить корректность их использования (количество и типы аргументов).
2. Разделение интерфейса и реализации. Заголовочный файл является интерфейсом, который сообщает какой функционал доступен для использования. При этом исходный файл содержит определения этого функционала, то есть его реализацию.
3. Разделение программы на модули. Заголовочный файл выполняет связующую роль (контракт) между модулями.
4. Подключение функционала стандартных и других библиотек. Достаточно просто включить нужный заголовочный файл вместо добавления целой пачки прототипов функций.

```
#include <stdio.h> // Функции ввода-вывода (printf, scanf)  
#include <stdlib.h> // malloc, free и т.д.
```

Недостатки заголовочных файлов

1. **Циклические зависимости:** если `a.h` включает `b.h`, а `b.h` включает `a.h`, компилятор может "зациклиться". Решение — `#pragma once` или `#include guards`. Пример заголовочного файла `utils.h` можно переписать через `#pragma once`:

```
// Заголовочный файл utils.h  
#pragma once
```

```
// Объявление глобальной переменной  
extern int g_var;  
  
// Объявление const-константы  
extern const float SPEED_OF_LIGHT;  
  
// Прототип функции  
void foo(int bar);
```

2. **Дублирование кода:** Каждый `.c`, включающий `.h`, обрабатывает его содержимое, что замедляет компиляцию. В качестве решения проблемы можно использовать **предкомпилированные заголовки** (PCH).

Директива `#include` имеет две формы:

1. Для системных заголовочных файлов

```
#include <path-spec>
```

1. Для несистемных заголовочных файлов

```
#include "path-spec"
```

Разница между этими формами заключается в порядке путей поиска.

Чем отличаются системные заголовочные файлы от несистемных?

Системные - это файлы, размещаемые в системных каталогах:

- стандартные заголовочные файлы (ISO C / POSIX)
- файлы операционной системы
- другие библиотеки (устанавливаемые в системные каталоги, в Linux обычно пакетным менеджером apt, yum, rpm и т.п.)

Несистемные (пользовательские):

- это все остальные файлы, не относящиеся к системным, и размещаемые непосредственно в проекте или в связанных несистемных каталогах

В директиве `#include` всегда указываются относительные пути. Конечно, можно указать абсолютный полный путь к требуемому файлу, но это делает код непереносимым.

Порядок поиска заголовочного файла зависит от опций компилятора и операционной системы.

Порядок в MSVC:

1. Для системных файлов:

- в каталогах, указанных в параметре `/I`
- если не найден, то в каталогах, указанных в переменной среды `INCLUDE`

2. Для несистемных файлов:

- в том же каталоге, что и файл, содержащий инструкцию `#include`.
- если не найден, то в каталогах файлов, включающих текущий файл (поиск происходит в обратном порядке включения, от самого глубоко вложенного файла к верхнему)
- если не найден, то в каталогах для системных заголовочных файлов (см. выше)

Порядок в GCC:

1. Для системных файлов:

- в каталогах, указанных в параметре `-I`
- если не нашел, то в каталогах, указанных в параметре `-isystem`
- если не нашел, то в стандартных системных каталогах GCC:

Примерный список:

```
/usr/lib/gcc/x86_64-linux-gnu/15/include  
/usr/local/include  
/usr/include/x86_64-linux-gnu  
/usr/include
```

2. Для несистемных файлов:

- в том же каталоге, что и файл, содержащий инструкцию `#include`.
- если не нашел, в каталогах, указанных в параметре `-iquote`
- если не нашел, то продолжает поиск в каталогах для системных заголовочных файлов.

Посмотреть список можно в выводе команды:

```
gcc -v -xc -E /dev/null
```

Нужно обратить внимание, что аналогов `-isystem` и `-iquote` в MSVC нет.

Рекомендации

Настоятельно не рекомендуется использовать в относительных путях использовать `../` (родительская директория), `.` (текущая директория), а также вложенные каталоги. При реорганизации структуры проекта, эти пути легко могут стать невалидными и придется во множестве файлов корректировать их. Пути необходимо "привязывать" не к текущему файлу, а к путям, указанным в параметре `-I` (или `/I` в MSVC).

Плохой пример:

```
#include "../include/myheader.h"
```

Правильно (указать в параметрах gcc `-linclude`):

```
#include "myheader.h"
```

Под Windows используйте `/` вместо `\` для переносимости

Включайте в файл только необходимые заголовочные файлы. Что это значит?

Уже знакомый нам пример мы могли бы написать так:

```
// Заголовочный файл utils.h
#ifndef UTILS_H
#define UTILS_H

#include <stdio.h>

// Объявление глобальной переменной
extern int g_var;

// Объявление const-константы
extern const float SPEED_OF_LIGHT;
```

```
// Прототип функции
void foo(int bar);

#endif
```

```
// Реализация utils.c
#include "utils.h"

// Определение глобальной переменной
int g_var = 42;

// Определение const-константы
const float SPEED_OF_LIGHT = 2.99792458e8f;

// Реализация функции
void foo(int bar) {
    printf("foo: bar=%d, g_var=%d, speed=%.2e\n", bar, g_var, SPEED_OF_LIGHT);
}
```

```
// Главный файл main.c
#include "utils.h"

int main(void) {
    // Использование глобальной переменной
    printf("g_var = %d\n", g_var);
    g_var = 100;

    // Использование const-константы
    printf("Speed of light = %.2e m/s\n", SPEED_OF_LIGHT);

    // Вызов функции
    foo(123);

    return 0;
}
```

То есть вместо того, чтобы писать в каждом модуле `#include <stdio.h>`, мы взяли и поместили его в `utils.h`. Многим такая идея очень нравится и на практике такое встречается очень часто. Но если я пишу модуль, который включает `utils.h`, но не использует функцию `foo`, а также функционал из `<stdio.h>`, то мне "бесплатно" достается совершенно ненужный дополнительный заголовочный файл. При этом компилятор затратит время на него, а значит это небесплатно!

Так вот, в идеале заголовочные файлы не должны содержать другие включения. Да, зачастую этого трудно достичь, но к этому нужно стремиться. Это один из критериев "чистого кода".

И последний вопрос, который мы здесь рассмотрим, это в каком порядке включать заголовочные файлы.

Рекомендуемый порядок включения:

- Предкомпилируемый заголовок (`stdafx.h`, `pch.h`) и только в модулях (.c)
- Заголовок текущего модуля
- Заголовки стандартной системной библиотеки С
- Заголовки сторонних библиотек
- Остальные заголовки проекта

```
// utils.c
// 1. Предкомпилируемый заголовок (если используется)
#include "pch.h"

// 2. Заголовок текущего модуля (соответствующий .c файлу)
#include "utils.h"

// 3. Заголовки стандартной системной библиотеки С
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

// 4. Заголовки сторонних библиотек
#include <openssl/ssl.h>
#include <zlib.h>

// 5. Остальные заголовки проекта
```

```
#include "color.h"
#include "shapes.h"

// ...
```

20. Предкомпилированные заголовки

Компилятор тратит определенное время на то, чтобы включить заголовочные файлы (скопировать их содержимое) в модуль трансляции (файл .c) и проанализировать их. Когда одни и те же заголовки (`<stdlib.h>`, `<stdio.h>` и т.д.) включаются в разных модулях компилятор часть работы выполняет заново. Для больших проектов из сотен и даже тысяч модулей трансляции это время может быть существенным и это является большой проблемой языков С и С++, особенно когда используются большие заголовочные файлы, например, `<windows.h>`, `<Cocoa/Cocoa.h>`, различные header-only библиотеки.

Предкомпилируемые заголовки (Precompiled headers - PCH) являются одним из способов решить эту проблему, то есть предназначены для ускорения сборки проекта.

Предкомпилированный заголовок - это особый заголовочный файл, который компилируется в промежуточную форму (файлы .pch в MSVC, .gch в GCC), более быструю для обработки компилятором.

Предкомпилированные заголовки создаются перед компиляцией остальных модулей трансляции проекта. Процесс незначительно различается в разных компиляторах, но общий принцип таков:

- компилятору указывается .h-файл, в который включают наиболее часто включаемые заголовочные файлы. Компилятор парсит его и создает из него бинарный файл - предкомпилированный заголовок.
- компилятор компилирует .cpp-файлы, в которых в самом первом `#include` указан .h-файл из первого пункта и указаны определенные опции компиляции. При этом компилятор использует предкомпилированный заголовок, а не его .h-файл.

Предкомпилированные заголовки в GCC/Clang

Шаги для подключения PCH в GCC:

1. Скомпилировать обычным образом .h-файл:

```
# Создаем common.h.gch
gcc -O2 common.h
```

2. В .cpp-файлах в самом начале подключам .h-файл:

```
// main.c  
#include "common.h"  
// далее остальные includes
```

3. Компилируем .cpp-файлы:

```
gcc -std=c11 -O2 -c main.c
```

GCC автоматически находит .gch-файлы и подключает их при компиляции.

Важно! При изменении .h-файла следует пересоздать предкомпилируемый заголовок. Отсюда важный вывод можно сделать: .h-файл должен меняться как можно реже.

Предкомпилированные заголовки в MSVC

Поскольку MSVC не имеет встроенных мастеров для проектов на языке C, сперва рассмотрим как происходит поддержка предкомпилируемых заголовков в C++.

В MSVC обязательно создается пара файлов .cpp и .h, так как нет возможности скомпилировать предкомпилируемый заголовок без .cpp или .c-файла. До Visual Studio 2017 по-умолчанию это были файлы `stdafx.cpp` и `stdafx.h`.

Интересный факт: имя `stdafx` расшифровывает как "Standard AFX". AFX - это библиотека Application Framework eXtensions от Microsoft, предшественник MFC (Microsoft Foundation Classes - фреймворк для разработки на C++ под Windows)

Начиная с Visual Studio 2017 файлы переименовали в `pch.cpp` и `pch.h`.

При этом для `pch.cpp` в свойствах проекта (Properties -> C/C++ -> Precompiled Headers) устанавливается свойство **Precompiled Header** в **Create (/Yc)**.

Для всех остальных `.cpp` файлов устанавливается свойство **Precompiled Header** в **Use (/Yu)** и указываем имя заголовка (`pch.h`)

Важно! В каждом файле первым включением в .cpp-файлах было `#include "pch.h"`.

Все тоже самое работает и с файлами .c, только делать все придется вручную через свойства проекта.

При сборке компилятор сперва генерирует файл `.pch`. Затем запустит многопоточную компиляцию остальных модулей.

Можно использовать командную строку следующим образом:

```
# Создание PCH
cl /TC /Yc pch.h pch.c

# Компиляция с использованием PCH
cl /TC /Yu pch.h main.c utils.c
```

Рекомендации

Рекомендуется включать в pch.h:

- системные и несистемные заголовки, включаемые во множестве модулей (т.е. это не два-три модуля, речь про десятки и более файлов)
- редко меняющиеся заголовки, чтобы не тратить время на перекомпиляцию предкомпилируемых заголовков при инкрементальной сборке

Обратите внимание на заголовки, включаемые во включаемых файлах и даже через несколько вложенных включений. Среди них могут оказаться часто изменяемые файлы. Чтобы получить список включаемых файлов:

```
# GCC/Clang
gcc -H -c myfile.c

# MSVC
cl /showIncludes myfile.c
```

В случае очень больших проектов имеет смысл держать несколько .pch файлов для разных наборов включаемых файлов.

Для оценки времени компиляции файлов компиляторы имеют встроенные возможности:

```
# Детальная статистика со временем компиляции
gcc -ftime-report -c myfile.c

# Только время компиляции (ничего лишнего)
gcc -ftime-report -c myfile.c 2>&1 | grep -E "(TOTAL|cc1)" | head -5

# Аналогично в Clang
```

```
clang -ftime-report -c myfile.c

# Показать время компиляции в MSVC
cl /Bt /c myfile.c

# Более детальная статистика
cl /d2cgsummary /c myfile.c
```

Также для анализа скорости сборки в MS Visual Studio 2019 и позже также доступно средство от MS - C++ Build Insights, главным компонентом которого является утилита vcperf, а также расширение для Windows Performance Analyzer.

21. Управление памятью

Виртуальное адресное пространство

Каждый процесс (запущенная программа) на большинстве операционных систем работает не напрямую с физической памятью, а с виртуальной памятью (virtual memory, далее VM) и у каждого процесса есть свое виртуальное адресное пространство. Тем самым повышается общая безопасность за счет изоляции данных и кода программ, а также производительность системы. Кроме того это упрощает разработку программного обеспечения, так как программистам (и компиляторам) не нужно знать и учитывать нюансы работы физической памяти.

Автоматической трансляцией адресов VM в физические занимается процессор через MMU (memory management unit).

Виртуальные адреса начинаются с нуля и растут вверх. Условно всё пространство делится на две части:

- **Пространство пользователя (User space):** Где живет и работает код самого процесса.
- **Пространство ядра (Kernel space):** Где живет код операционной системы (ядро). Это пространство **одинаково для всех процессов** и защищено от прямого доступа из пользовательского режима.

Пространство пользователя

1. Самые младшие адреса (0 - 0x00007fffffffffffff) являются **зарезервированными** и доступ к ним запрещен, в т.ч. для чтения.
2. **Сегмент кода** (Code segment, .text) отвечает за машинные инструкции программы (скомпилированный бинарный код). Сегмент имеет атрибуты read-only + execute. Несколько запущенных экземпляров программы используют один и тот же сегмент (нет необходимости обновления)
3. **Сегмент данных** (Data segment) отвечает за данные и делится на две секции:
 - Инициализированные данные (.data) - отвечает за глобальные и статические переменные, инициализированные ненулевыми значениями (rwdata), а также константы, известные на этапе компиляции (rodata)
 - Неинициализированные данные (.bss - block started by symbol) - отвечает за глобальные и статические неинициализированные либо инициализированные

нулем переменные. Операционная система обнуляет эту область при запуске программы.

4. Динамическая память, куча (Heap) - динамическая память, выделяемая при использовании функций `malloc/calloc/realloc`. Куча растет снизу-вверх (от младших адресов).
5. Память, отображающая файлы (Memory-Mapped Files) - отвечает за динамические библиотеки (.dll в Windows, .so в Linux), а также за "отображеные" файлы в памяти.
6. Автоматическая память, стек (Stack) отвечает за локальные переменные, аргументы функций, адреса возврата. Стека растет сверху-вниз (в сторону младших адресов). У каждого потока имеется свой собственный стек.

Пространство ядра

Пространство ядра содержит код и данные операционной системы. Этот регион защищен от прямого доступа пользовательского кода. Переход в него происходит только через строго определенные интерфейсы — **системные вызовы** (syscalls). При таком вызове процессор переключается в привилегированный режим.

Следующий пример демонстрирует размещение различных данных в памяти:

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

// Глобальные переменные:
int global_uninit;           // .bss (неинициализированная)
int global_init = 10;         // .data (инициализированная)

// Функция находится в сегменте .text
void demonstrate_memory_layout(int stack_param) { // stack_param - в стеке
    // Статические переменные:
    static int static_uninit;    // .bss
    static int static_init = 5;  // .data

    // Локальные переменные - в стеке:
    int local_var = 100;
    int local_array[3] = {1, 2, 3};

    // Динамическая память - в куче (heap):
    int *heap_var = malloc(sizeof(int) * 4);
```

```

// Строковый литерал – обычно в .rodata (read-only data, часть .text)
char *text_literal = "Hello, Memory Layout!";

// Заполняем кучу данными
for(int i = 0; i < 4; i++) {
    heap_var[i] = i * 10;
}

// Выводим адреса для наглядности
printf("==== Memory Layout Analysis ===\n\n");

printf("Code Segment (.text):\n");
printf("  Function address: %p\n\n", demonstrate_memory_layout);

printf("Data Segment (.data):\n");
printf("  global_init:      %p\n", &global_init);
printf("  static_init:      %p\n\n", &static_init);

printf("BSS Segment (.bss):\n");
printf("  global_uninit:    %p\n", &global_uninit);
printf("  static_uninit:    %p\n\n", &static_uninit);

printf("Heap Segment:\n");
printf("  heap_var:         %p\n", heap_var);
printf("  (points to):     %p\n\n", &heap_var[0]);

printf("Stack Segment:\n");
printf("  stack_param:      %p\n", &stack_param);
printf("  local_var:         %p\n", &local_var);
printf("  local_array:       %p\n", local_array);
printf("  local_array[0]:    %p\n\n", &local_array[0]);

printf("Read-Only Data (.rodata):\n");
printf("  text_literal:      %p\n", text_literal);
printf("  (points to):      %p\n", "Hello, Memory Layout!");

// Не забываем освободить память!
free(heap_var);

```

```
}

int main() {
    demonstrate_memory_layout(42);
    return 0;
}
```

Выделение динамической памяти

Стандартный способ выделить динамическую память в Linux заключается в вызове функций стандартной библиотеки C - `malloc/calloc/realloc`. Выделение динамической памяти из кучи процесса через эти функции скрывает отличия в механизме распределения памяти через нативные API операционных систем и избавляет от необходимости учитывать нюансы и особенности этого процесса.

Рассмотрим в общих чертах эти отличия.

Основная функция выделяющая динамическую память в MS Windows - `HeapAlloc` из `Kernel32.dll`. Это прокси-функция, которая вызывает другую функцию `RtlAllocateHeap` из менеджера кучи (`Windows Heap Manager`), в котором и происходит вся магия, в частности из него вызывается функция `VirtualAlloc` из менеджера памяти (`NT Memory Manager`), выделяющая страницы виртуальной памяти.

Примечание: страница памяти - это наименьшая единица, с которой работает виртуальная память. Сопоставление виртуальных адресов с физическими происходит именно страницами, а не байтами, словами и т.д.

Эти менеджеры постоянно совершенствуются и, благодаря этому, одни и те же программы работают на разных версиях Windows с разной производительностью. Также есть устаревшие функции `LocalAlloc` и `GlobalAlloc`, которые оставлены в Windows для поддержки совместимости, но фактически дублируют `HeapAlloc`, так как внутри теперь также вызывают `RtlAllocateHeap`.

Таким образом, функция `malloc` из стандартной библиотеке C для Windows, если отбросить несущественные детали, просто вызывает `HeapAlloc`. Это потокобезопасно и переносимо между различными версиями Windows.

В Linux выделение динамической памяти реализовано в самой стандартной библиотеке. Функция `malloc` в ранних версиях `libc` использовала собственный механизм распределения памяти через системные функции `brk/sbrk`, увеличивающие размер кучи процесса (двигают вверх границу кучи).

То есть через `brk/sbrk` запрашивался определенный кусок (chunk) из кучи, внутри которого уже происходило распределение памяти. Как только память в запрошенном куске заканчивалась, запрашивался новый.

Потом добавилась функция `mmap` (условно аналог `VirtualAlloc` из Windows) для выделения больших объемов памяти (>128 Кб). В отличие от `brk/sbrk` позволяет занимать практически любую область памяти.

Функции `brk/sbrk` работают только с одной непрерывной областью памяти, что создает определенные проблемы с фрагментацией памяти, а также возникают сложности при создании многопоточных приложений.

Современная реализация `malloc` в GNU libc использует аллокатор общего назначения `ptmalloc2`, который для каждого потока (включая главный) создает арену - по сути отдельная куча с пулами памяти. В FreeBSD используется аллокатор `jemalloc`.

Таким образом, `malloc` в Linux - это не просто обертка над другой функцией из Linux API, а самостоятельный и достаточно сложный аллокатор памяти.

Существуют альтернативные аллокаторы памяти, которые можно попробовать использовать для решения проблем производительности, но для большинства задач достаточно использовать стандартный `malloc`.

Функция `malloc`

Функция `malloc` (`<stdlib.h>`) выделяет не менее `size` байт в куче и возвращает указатель на начало блока.

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

В случае неудачи возвращает `NULL`, поэтому всегда следует проверять результат этой функции.

Функция `malloc` обеспечивает выравнивание в памяти, необходимое для стандартных базовых типов.

Обратите внимание, что `malloc` не обязана выделять ровно столько, сколько запросили у нее через `size`. Это значение будет округлено до некоторого кратного числа, зависящего от реализации. Кроме того, перед выделяемым блоком будет размещен контрольный блок, необходимый аллокатору для управления выделенной памятью.

В Linux можно проверить сколько именно было выделено памяти через функцию `malloc_usable_size`. В Windows имеется аналог - `_msize`.

Функция `calloc`

Для выделения динамической памяти под массив используется функция `calloc` (`<stdlib.h>`).

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

Функция `calloc` кроме того, что выделяет память, еще и обнуляет ее. Но это вовсе не значит, что под капотом она просто вызывает пару функций `malloc+memset`. Для логического объяснения, конечно, подойдет такой код:

```
void *calloc(size_t n, size_t size) {
    size_t total_size = n * size;
    void *ptr = malloc(total_size);
    if (ptr != NULL) {
        memset(ptr, 0, total_size);
    }
    return ptr;
}
```

Но настоящая реализация немного хитрее и намного эффективнее. В частности, вызов `memset` для больших объемов является очень дорогим. И вместо этого, например в Linux, используется трюк с использованием так называемой нулевой страницы (доступна только для чтения), заполненной нулями. Когда происходит первая запись в эту страницу, она копируется в новую физическую страницу и таким образом происходит "зануление". Эта оптимизация называется **lazy initialization** или **zero-on-demand**.

В Windows функция `HeapAlloc` имеет специальный флаг `HEAP_ZERO_MEMORY` и зануление эффективно происходит в недрах менеджера памяти Windows.

Из этого можно сделать простой вывод: в случае, если нужно выделить зануленную память всегда вместо пары функций `malloc+memset` используйте `calloc`.

Однако, если вам не нужна зануленная память, используйте `malloc`. То есть нужно понимать, что `calloc` обладает определенными накладными расходами, а значит производительность ниже, чем у `malloc`. Функция `malloc` может переиспользовать

ранее освобожденные страницы памяти, но все еще находящиеся в распоряжении приложения. Это еще одна оптимизация, нацеленная на высокую производительность.

Функция realloc

Функция `realloc` (`<stdlib.h>`) используется для изменения (увеличения или уменьшения) размера ранее выделенного блока памяти.

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

Внимание! Не нужно и даже опасно освобождать старый указатель! При необходимости, функция `realloc` сама освободит старый блок памяти.

```
size_t len = 128;
void* ptr = malloc(len);
// ...
void* new_ptr = realloc(ptr, len + 512);
free(ptr); // Ошибка!!! Нельзя освобождать ptr
// ...
free(new_ptr); // Правильно
```

В Linux при увеличении размера, в случае если сразу за ранее выделенным блоком памяти имеется достаточно свободного места или при уменьшении размера, происходит соответственно расширение или урезание за счет изменения информации в контрольных блоках **без копирования старых данных**. Но если свободного места нет, то происходит выделение новой памяти и копирование данных в новое место.

В Windows функция `realloc` вызывает `HeapReAlloc`, которая почти всегда перевыделяет память и копирует данные. По этой причине, `realloc` в Linux работает эффективнее.

Проверить (в исследовательских целях), произошло ли перевыделение с копированием или без, можно сравнив указатели:

```
void* old_ptr = ptr;
ptr = realloc(ptr, new_size);
if (ptr != old_ptr) {
    // Копирование, т.к. выделен новый блок
}
```

Функция free

Освобождением памяти занимается функция `free` (`<stdlib.h>`).

```
void free( void *ptr );
```

Классический вопрос на собеседованиях: Нужно ли проверять указатель на `NULL` перед вызовом функции `free`?

```
if (ptr != NULL)  
    free(ptr);
```

Ответ: нет, не нужно. Проверка не будет ошибкой, но она избыточна, поскольку такая проверка есть в самой функции `free`. Поэтому грамотный код должен выглядеть так (без лишних проверок):

```
free(ptr);
```

Функция aligned_alloc

Функция `aligned_alloc` (`<stdlib.h>`) появилась в C11 и используется для выделения динамической памяти с указанным выравниванием. То есть выделенный блок будет размещен по адресу, кратному выравниванию. При этом значение выравнивания должно быть значением степени двойки, но не менее размера указателя - `sizeof(void*)`, а размер должен быть кратен выравниванию.

```
void* aligned_alloc(size_t alignment, size_t size ); // since C11
```

Эта функция полезна для некоторых оптимизаций. Однако, она не поддерживается в CRT под Windows (так как не имеет соответствующей поддержки в ядре) и Microsoft даже предлагает свою функцию с другой сигнатурой:

```
void* _aligned_malloc(size_t size, size_t alignment);
```

Обратите внимание на порядок аргументов `size` и `alignment`. Под капотом она использует `malloc` и дополнительную память при каждом выделении.

Таким образом, используя `aligned_alloc` в своем коде, вы делаете его непереносимым.

Важно! Для освобождения памяти, выделенной с помощью функции `aligned_alloc` должна использоваться специальная функция `aligned_free`, а не `free`. Под Windows соответственно нужно использовать функцию `_aligned_free`.

Копирование памяти

Следующие функции находятся в заголовочном файле `<string.h>` и из названия становится понятно, что предназначены они для работы со строками в том числе, но фактически работают они с любыми массивами памяти и в отличии от остальных функций из `<string.h>`, начинаются с префикса `mem`, а не `str`:

Непосредственно для копирования используются две функции:

```
void* memcpuy(void *dest, const void *src, size_t count);
void* memmovey(void* dest, const void* src, size_t count);
```

Функция `memcpuy` используется для копирования `count` байт из источника (`src`) в область назначения (`dest`) при условии, что эти области памяти не перекрываются (не пересекаются). Если области памяти могут перекрываться, то использование `memcpuy` приводит к неопределённому поведению.

Функция `memmovey` используется, когда области памяти `src` и `dest` могут пересекаться. `memmovey` копирует данные с гарантией корректности в случае перекрытия, обеспечивая правильное копирование путем временного буфера или другими способами (из-за этого функция может работать медленнее `memcpuy`).

Отсюда рекомендация: используйте `memcpuy`, если области не пересекаются, иначе используйте `memmovey`.

Заполнение памяти

Функция `memset` заполняет начиная с адреса `dest` первые `count` элементов значением `ch`.

```
void* memsety(void *dest, int ch, size_t count);
```

Важно! Распространенная ошибка думать, что `dest` заполняется числами типа `int`:

```
int arr[10];
```

```
memset(arr, 1, sizeof(arr)); // Ошибка!
```

В этом примере заполнится только первые 10 байт, а не первые 10 элементов!

Функция `memset` имеет побайтовую природу, а данные заполняются только младшим байтом аргумента `ch`, то есть `(unsigned char)ch`.

Сравнение памяти

Функция `memcmp` используется для сравнения первых `count` байт двух областей памяти, на которые указывают `lhs` и `rhs`:

```
int memcmp(const void* lhs, const void* rhs, size_t count);
```

Поиск в памяти

Функция `memchr` производит поиск `ch` среди первых `count` байт начиная с адреса `ptr`:

```
void* memchr(const void* ptr, int ch, size_t count);
```

Имеет тоже побайтовую природу как и `memset` и фактически ищет младший байт `ch`, то есть `(unsigned char)ch`.

Не изобретайте велосипеды!

Не пишите свои циклы (или функции с циклами) для побайтного копирования, заполнения, сравнения и поиска. Используйте стандартные `mem`-функции `memcpuy/memmove/memset/memcmp/memchr`, так как в этих функциях применяются оптимизации, позволяющие эффективно выполнять требуемые операции.

Ошибки при работе с памятью

Отказ в выделении памяти

Когда система исчерпала доступную память функции `malloc/calloc/realloc` возвращают `NULL`. Результат в обязательном порядке нужно проверять:

```
int *ptr = malloc(very_large_size);
if (ptr == NULL) {
```

```
// Обработка ошибки  
}
```

Согласно стандарту POSIX, при невозможности выделить память `malloc` устанавливает `errno = ENOMEM`. Значение `ENOMEM` буквально означает "Out of memory" (недостаточно памяти). Функция `realloc` может установить `errno = EINVAL`, в случае неверного параметра.

Стандарт С этого не требует, однако большинство компиляторов придерживаются требований POSIX. Поэтому проверки могут выглядеть следующим образом:

```
#include <stdlib.h>  
#include <errno.h>  
#include <stdio.h>  
  
int *allocate_memory(size_t size) {  
    // Сбрасываем errno перед вызовом  
    errno = 0;  
  
    int *ptr = malloc(size);  
    if (ptr == NULL) {  
        if (errno == ENOMEM) {  
            fprintf(stderr, "Недостаточно памяти для выделения %zu байт\n",  
size);  
        } else {  
            fprintf(stderr, "Неизвестная ошибка при выделении памяти: %d\n",  
errno);  
        }  
    }  
    return ptr;  
}  
  
int main() {  
    int *data = allocate_memory(1ULL << 40); // Пытаемся выделить 1 ТБ  
  
    if (data == NULL) {  
        // Дополнительная обработка ошибки  
        return EXIT_FAILURE;  
    }
```

```
// Работа с памятью...
free(data);
return EXIT_SUCCESS;
}
```

В Windows дополнительно можно вызвать WinAPI функцию `GetLastError`, которая вернет `ERROR_NOT_ENOUGH_MEMORY`.

Возникает вопрос: что же делать в случае когда память исчерпана и выделить новую не получается. Ответ сводится к двум решениям:

- завершить программу с выводом сообщения;
- попытаться освободить ранее выделенную память и попытаться снова выделить память.

Однако, обратите внимание, что в вышеприведенном примере вызывается `fprintf` или какая-то другая функция, которая также может пытаться выделять память и она также потерпит неудачу. Поэтому предпринимаемые меры должны учитывать этот фактор.

Неинициализированные указатели

Указатели, объявленные без инициализации содержат мусор из стека и могут использоваться для проведения хакерских атак. Это серьезная уязвимость класса **Use of Uninitialized Variable**.

```
void leak_data() {
    void* ptr; // Содержит фрагменты стека
    printf("Утечка данных: %p\n", ptr); // Атакующий видит значения стека
}
```

Всегда инициализируйте подобные указатели нулем:

```
void* ptr = NULL;
```

Также используйте в обязательном порядке предупреждения компилятора:

`-Wuninitialized` для GCC

`/w4` для MSVC включает проверку неинициализированных переменных

Висячие указатели (Dangling pointers)

Указатель, который продолжает указывать на память, после ее освобождения, называется висячим и создает те же проблемы, что и неинициализированные указатели.

Как возникают висячие указатели:

1. Освобождение памяти без обнуления указателя:

```
int* ptr = malloc(sizeof(int));
*ptr = 42;
free(ptr); // Память освобождена, но ptr все еще указывает туда же
// ptr теперь - висячий указатель!
```

2. Возврат из функций указателей на локальные переменные

```
int* create_dangling() {
    int local = 100;
    return &local; // Возвращаем указатель на локальную переменную
} // local уничтожается здесь!

int main() {
    int *dangling = create_dangling(); // Висячий указатель!
    return 0;
}
```

3. Выход за пределы области видимости

```
void scope_issue() {
    int *ptr;
    {
        int x = 10;
        ptr = &x; // Указываем на локальную переменную
    } // x уничтожается здесь
    // ptr теперь висячий!
}
```

Висячие указатели могут приводить к утечкам данных, повреждению данных, и другим серьезным уязвимостям: Use After Free, Double Delete.

Самая превентивная мера против висячих указателей - это их зануление после освобождения:

```
free(ptr);
ptr = NULL; // Теперь безопасно!
```

Использование после освобождения (Use After Free)

Это одна из самых серьезных и часто эксплуатируемых уязвимостей. Суть заключается в том, что после освобождения памяти, ее продолжают использовать:

```
char *ptr = (char*)malloc(100);
strcpy(ptr, "Важные данные");
free(ptr); // Память освобождена

// ... какой-то код ...

// ОПАСНОСТЬ: использование после освобождения!
strcpy(ptr, "Новые данные"); // Use After Free
```

Такая уязвимость в лучшем случае может привести просто к неопределенному поведению, повреждению кучи или ошибке типа Segmentation Fault, а в худшем к выполнению произвольного кода злоумышленником, если атакующий сможет подменить освобожденные данные своим malicious-кодом и заставит программу выполнить его.

Успешная эксплуатация UAF может привести:

- Выполнение произвольного кода (Remote Code Execution, RCE)
- Эскалация привилегий (Elevation of Privilege, EoP)
- Обход защит (ASLR, DEP)

Двойное удаление (Double Free)

Уязвимость, которая проявляется когда программа пытается дважды освободить один и тот же блок памяти.

```
char *ptr = malloc(32);
// Работа с памятью...
free(ptr);      // Первое освобождение – ОК
// ... какой-то код ...
free(ptr);      // Второе освобождение – DOUBLE FREE!
```

Такие действия способны повредить кучу и даже выполнить произвольный код.

Поэтому после освобождения памяти важно обнулять указатели.

Переполнение стека (Stack Overflow)

Одна из самых эксплуатируемых уязвимостей. Когда программа превышает использование выделенного пространства стека, срабатывает защита системы, происходит ошибка Stack Overflow, приводящая к аварийному завершению.

Возникает из-за:

- бесконечного вызова рекурсивных функций (без базового случая)

```
void infinite_recursion() {
    int local_var[100]; // Каждый вызов выделяет место в стеке
    infinite_recursion(); // Бесконечная рекурсия
}
```

- использование больших локальных объектов, в частности массивов на стеке

```
void huge_stack_allocation() {
    int massive_array[1000000]; // 4МБ на стеке!
    // Обычный размер стека: 1–8МБ
}
```

- переполнение буфера на стеке (самая опасная разновидность SO)

```
void vulnerable_function(char *input) {
    char buffer[64]; // Маленький буфер
    strcpy(buffer, input); // Переполнение при длинном input!
}
```

Уязвимость приводит к:

- Выполнение произвольного кода
- Отказ в обслуживании (Denial of Service, DoS)
- Эскалация привилегий (EoP)
- Обход защит (ASLR, DEP, Stack Canaries)

Выход за границы диапазона (Out of Bounds Access)

Чтение и запись за границами выделенной области памяти также является уязвимостью, приводящей к тем же последствиям, что и переполнение буфера и стека: RCE, DoS, EoP, обход защит.

Самая частая ошибка - некорректные индексы массивов:

```
int buffer[10];
for (int i = 0; i <= 10; i++) { // Ошибка: должно быть i < 10
    buffer[i] = 0; // Запись за пределами массива при i = 10
}
```

На втором месте - ошибки в арифметике указателей:

```
size_t n = 100
int* ptr = malloc(n);
for (int i = 0; i < n; ++i, ++ptr) {
    *ptr = i; // на 25 итерации произойдет запись за пределами выделенной
    памяти
}
```

И другие логические ошибки:

```
int copy_data(char *input, size_t len) {
    char local_buf[64];
    // ОШИБКА: нет проверки длины!
    memcpuy(local_buf, input, len); // Переполнение при len > 64
}
```

Утечки памяти

Утечки возникают в ситуациях, когда после выделения динамической памяти функциями `malloc/calloc/realloc` не происходит их освобождение функцией `free`.

Рассмотрим распространенные случаи подобных ошибок.

1. Забыли вызвать `free`.

```
void foo()
{
    int* ptr = malloc(24);
    // утечка памяти, память не освобождена
}
```

2. Ранний выход из функции (ошибки в условиях)

```
void foo(int value)
{
    int* ptr = malloc(24);
    // ...
    if (value <= 0)
        return; // утечка памяти, память не освобождена
    // ...
    free(ptr);
}
```

3. Перезапись указателя

```
void foo(int value)
{
    int* ptr = malloc(24);
    // ...
    ptr = malloc(128); // утечка памяти, память не освобождена
    // ...
    free(ptr);
}
```

Утечки зачастую сложно обнаружить, так как они могут себя проявлять под серьезными нагрузками или в результате длительной работы. Поэтому необходимо на постоянной основе проверять программу статическими анализаторами, санитайзерами и другими инструментами. В Linux, например, стандартным инструментом для поиска утечек является Valgrind.

22. Работа со строками

В данной главе пойдет речь про классические однобайтовые ANSI-строки. Также в языке предусмотрена работа с мультибайтными (multibyte) и широкими (wide) строками, о которых мы поговорим в следующих главах.

Однобайтные ANSI-строки (null-terminated byte strings) представляются в виде массивов символов типа `char`, заканчивающиеся нулевым терминатором - `\0`:

```
char str[] = "Hello"; // Автоматически добавляется '\0'
```

Здесь для массива `str` выделяется на стеке память размером 6 байт (в том числе для `\0`) и выполняется инициализация массива при помощи строкового литерала "Hello". Можно записать эту инструкцию и в другом более громоздком виде:

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

Но чаще вместо массива используют указатель для такой записи:

```
const char* str = "Hello"; // Только для чтения
```

Очень важно понять разницу между использованием массива и указателя при инициализации строковыми литералами.

```
// Строковые литералы нельзя изменять
char *str1 = "Hello"; // массив не выделяется на стеке
// str1[0] = 'h'; // ОШИБКА! Неопределенное поведение

// Для изменяемых строк используйте массивы
char str2[] = "Hello"; // выделяется память на стеке под массив
str2[0] = 'h'; // Правильно
```

Учтите, что код `str1[0] = 'h';` может быть успешно скомпилирован компилятором и даже правильно отработать. Но нельзя рассчитывать на это при написании переносимого кода, так как это поведение не гарантируется стандартом.

Большие строки обычно размещают в куче, выделяя для этого память через `malloc` или `calloc`:

```
size_t len = strlen(src);
char* dest = malloc(len + 1); // +1 для '\0'
strcpy(dest, src); // Копирует ВСЕ символы включая '\0'
// dest[len] = '\0'; << это избыточно!
// какой-то код
free(dest);
```

Обратите внимание, что явно устанавливать нулевой терминатор в конец строки `dest` нет необходимости, так как эту работу за нас сделает `strcpy`.

Нулевой терминатор и функция `strlen`

Нулевой терминатор является самым простым способом определить размер строки и многие функции используют это свойство в своих алгоритмах. Вот как может быть реализована функция получения длины строки:

```
size_t strlen(const char* str)
{
    const char* s = str;
    while(*s)
        s++;
    return s - str;
}
```

На самом деле реализация этой функции в стандартной библиотеке намного сложнее (благодаря оптимизациям) и настоятельно не рекомендуется использовать свои реализации в надежде, что они будут быстрее или надежнее.

Обратите внимание, что в этой функции, как и в настоящей реализации `strlen` (`<string.h>`) НЕТ проверки аргумента на `NULL`:

```
size_t strlen(const char* str) {
    // Проверка на NULL указатель отсутствует
    //if (str == NULL)
    //    return 0;
```

```
// Остальной код...
}

int main() {
    // Это вызовет неопределенное поведение (Undefined Behavior)
    size_t len = strlen(NULL);
}
```

В C11 ввели более безопасный вариант этой функции:

```
size_t strnlen_s(const char* str, size_t strsz); // since C11
```

Эта функция вернет 0 если str равно NULL или если в первых strsz символах не встретится \0 (строка слишком длинная или она содержит "мусор").

Неправильно использовать оператор sizeof для получения размера строки:

```
char str[] = "Hello";
printf("sizeof(str) = %zu\n", sizeof(str) - 1);
```

Этот код выведет правильный размер строки (- 1 используется для вычитания нулевого терминатора). Но следующие примеры выведут некорректную длину:

```
char str1[10] = "Hello";
printf("sizeof(str1) = %zu\n", sizeof(str1) - 1); // Выведет 9

char *str2 = "Hello";
printf("sizeof(str2) = %zu\n", sizeof(str2) - 1); // Выведет 7 в 64-разрядных
системах
```

ФУНКЦИИ КЛАССИФИКАЦИИ СИМВОЛОВ

Заголовочный файл <ctype.h> содержит следующие функции для определения к какой категории принадлежит символ:

```
int isalnum(int ch); // цифробуквенный символ
int isalpha(int ch); // буквенный символ (A-Z, a-z)
int islower(int ch); // символ нижнего регистра
```

```
int isupper(int ch); // символ верхнего регистра
int isdigit(int ch); // цифра (0-9)
int iscntrl(int ch); // управляющий символ (0x00-0x1F, 0x7F)
int isgraph(int ch); // графический символ (исключая пробел)
int isspace(int ch); // пробельный символ (пробел, \t, \n, \r, \f, \v)
int isblank(int ch); // пробел или \t (C99)
int isprint(int ch); // печатный символ (включая пробел)
int ispunct(int ch); // знак пунктуации
```

В качестве результата возвращается ненулевое значение в случае успеха и 0 в противном случае.

К знакам пунктуации относят: !"#\$%&'()/*,-./:;=>?@[\]^_`{|}~

Функции изменения регистра

Следующие функции также находятся в заголовочном файле `<ctype.h>`:

```
int tolower( int ch ); // получить символ в нижнем регистре
int toupper( int ch ); // получить символ в верхнем регистре
```

Пример:

```
char* string_to_upper(char *str)
{
    char *p = str;
    if (str == NULL)
        return NULL;
    while (*p)
    {
        *p = (char)toupper((unsigned char)*p);
        ++p;
    }
    return str;
}
```

Приведения типов очень важны в этом коде. Например, русские символы в СР-1251 кодировке являются отрицательными и при неявном приведении к `int` поведение

функции `toupper` станет неопределенным/ошибочным.

ФУНКЦИИ СРАВНЕНИЯ

```
int strcmp(const char* lhs, const char* rhs);           (1)
int strncmp(const char* lhs, const char* rhs, size_t count); (2)
int strcoll(const char* lhs, const char* rhs);          (3)
```

1. функция `strcmp` сравнивает две строки, оканчивающиеся нуль-терминатором. Если любая из строк равна `NULL`, то поведение неопределено.
2. функция `strncmp` сравнивает первые `count` символов двух строк. Если при этом нуль-терминатор в любой из этих строк встречается раньше или любая из строк равна `NULL`, то поведение неопределено.
3. функция `strcoll` сравнивает две строки с учетом локали.

Функции `strcmp` и `strncmp` выполняют лексикографическое сравнение (посимвольно слева направо) до первой отличающейся пары. По сути выполняется бинарное сравнение. Часто используются для проверки на равенство (неравенство) двух строк когда локаль не важна, но важно быстродействие.

Функция `strcoll` используется когда важна локаль. Например, для сортировки строк и отображения этого списка пользователю.

Следующий код демонстрирует сортировку массива строк без учета локали:

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

int compare_strings(const void* a, const void* b) {
    return strcmp(*((const char**)a), *((const char**)b));
}

int main() {
    char* words[] = {
        "яблоко", "абрикос", "ёжик", "еж",
        "Юля", "юла", "манго", "Ёж",
        "зебра", "Зоя", "йод", "эхо"
```

```

};

int n = sizeof(words) / sizeof(words[0]);

qsort(words, n, sizeof(char*), compare_strings);

printf("Отсортированный список:\n");
for (int i = 0; i < n; i++) {
    printf("%s ", words[i]);
}
printf("\n");

return 0;
}

```

Вывод:

```

Отсортированный список:
Ёж Зоя Юля абрикос еж зебра йод манго эхо юла яблоко ёжик

```

Обратите внимание на положение слов Ёж, еж и ёжик. Функция `strcmp` не годится для алфавитной сортировки строк.

Если заменить `strcmp` на `strcoll` и установить нужную локаль:

```

// ...

int compare_strings(const void* a, const void* b) {
    return strcoll(*(const char**)a, *(const char**)b);
}

int main() {
//...
    int n = sizeof(words) / sizeof(words[0]);

    if (setlocale(LC_COLLATE, "ru_RU.UTF-8") == NULL) {
        printf("Не удалось установить русскую локаль\n");
        return -1;
    }
}

```

```
qsort(words, n, sizeof(char*), compare_strings);

// ...

}
```

То получим список, отсортированный по алфавиту:

Отсортированный список:

абрикос еж Ёж ёжик зебра Зоя йод манго эхо Юла яблоко

Учтите, что функция `strcoll` медленнее, чем `strcmp`.

ФУНКЦИИ ПОИСКА

```
char* strchr(const char* str, int ch);                                (1)
char* strrchr(const char* str, int ch);                                 (2)
size_t strspn(const char* dest, const char* src);                      (3)
size_t strcspn(const char *dest, const char *src);                     (4)
char* strpbrk(const char *dest, const char *breakset);                 (5)
char* strstr(const char* str, const char* substr);                      (6)
char* strtok(char* str, const char* delim);                            (7)
char* strtok_s(char* restrict str, rsize_t* restrict strmax,           (8)
               const char* restrict delim, char** restrict ptr);
```

- ищет первое вхождение символа (char)ch в строке str. Возвращает указатель на найденный символ, либо NULL.
- ищет последнее вхождение символа (char)ch в строке str. Возвращает указатель на найденный символ, либо NULL.
- возвращает длину начального сегмента строки dest, в котором присутствуют только символы из src :

```
char str[] = "12345abc";
char digits[] = "0123456789";

size_t length = strspn(str, digits);
printf("Длина начального сегмента цифр: %zu\n", length);
// Вывод: Длина начального сегмента цифр: 5
```

4. возвращает длину начального сегмента строки `dest`, который не содержит символы из `src`:

```
char email[] = "user@example.com";
char at_symbol[] = "@";

size_t username_length = strcspn(email, at_symbol);
```

5. находит **первое вхождение любого символа** из заданного набора `breakset` в строке `dest`:

```
const char* str = "Пришел, увидел, победил!";
const char* sep = " ,!";

int cnt = 0;
do {
    str = strpbrk(str, sep); // найти разделитель
    if(str) str += strspn(str, sep); // пропустить разделитель
    ++cnt; // увеличим счетчик
}
while(str && *str);

printf("Найдено %u слова\n", cnt);
// Вывод: Найдено 3 слова
```

6. находит первое вхождение подстроки `substr`

7. находит следующий токен в строке `str`, используя разделители из `delim`:

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

int main() {
    char text[] = "apple,banana,orange,grape";
    char delimiter[] = ",";

    printf("Разбиваем строку: '%s'\n", text);
    printf("Токены:\n");
```

```

// Первый вызов
char *token = strtok(text, delimiters);

// Последующие вызовы
while (token != NULL) {
    printf("- '%s'\n", token);
    token = strtok(NULL, delimiters);
}

return 0;
}

```

Внимание! Функция `strtok` изменяет исходную строку, заменяя разделители на `\0`!

- В первом вызове передается строка для разбиения
- В последующих вызовах передается `NULL` для продолжения разбиения той же строки
- Каждый раз сохраняется указатель на следующую часть строки

Обратите внимание на разницу в функциях `strchr` и `memchr`:

```

// Ищет символ в строке (до нулевого терминатора)
char* strchr(const char* str, int ch);

// Ищет символ в блоке памяти размером count байт
void* memchr(const void* ptr, int ch, size_t count);

```

Манипуляции со строками

```

char* strcpy(char* dest, const char* src); (1)
char* strncpy(char* dest, const char* src, size_t count); (2)
char* strcat(char* dest, const char* src); (3)
char* strncat(char* dest, const char* src, size_t count); (4)
size_t strxfrm(char* dest, const char* src, size_t count); (5)

```

1. копирует строку `src` в строку `dest`. Количество копируемых символов определяется по нулевому терминатору (включительно). Поведение неопределено, если строки перекрываются или в `dest` недостаточно выделенной памяти.

2. копирует `count` символов строки `src` в строку `dest`. Если при копировании был достигнут нулевой терминатор в `src`, то в `dest` оставшиеся символы заполняются нулями.

3. добавляет строку `src` в конец `dest`.

Важно! Функция `strcat` не выделяет память и не возвращает новую строку `dest + src`! Возвращается указатель `dest`, который не нужно пытаться освобождать:

```
// Так делать нельзя!
char* str1 = malloc(100);
char* str2 = malloc(32);
// какой-то код
char* result = strcat(str1, str2);
free(result); // Ошибка!
free(str2);
free(str1); // Здесь произойдет Double Free
```

4. добавляет `count` символов строки `src` в конец строки `dest`. Возвращается указатель `dest`.

5. выполняет **трансформацию строки** для локале-зависимого сравнения. Она преобразует строку в такую форму, что после трансформации сравнение с помощью `strcmp` даст тот же результат, что и локале-зависимое сравнение исходных строк с помощью `strcoll`. Если нужно много раз сравнивать одни и те же строки, выгоднее один раз преобразовать их через `strxfrm`, а потом использовать быстрый `strcmp`

Многих удивляет, что в стандартной библиотеке нет многих полезных функций. Например, функций перевода строк в верхний или нижний регистр. Давайте напишем их!

```
#include <ctype.h>

// Приведение строки к нижнему регистру
char* strlower(char* str) {
    for (char* ptr = str; *ptr; ptr++)
        *ptr = tolower((unsigned char)*ptr);
    return str;
}

// Приведение строки к верхнему регистру
```

```

char* strupper(char* str) {
    for (char* ptr = str; *ptr; ptr++)
        *ptr = toupper((unsigned char)*ptr);
    return str;
}

```

Важно! Так как `ptr` является `signed char` и русские символы имеют отрицательные значения в ASCII необходимо их предварительно приводить к типу `unsigned char`, который потом неявно приводится к типу аргумента функций `tolower / toupper - int`.

Замена символов в строке тоже не должна вызывать сложностей в написании:

```

// Замена символов в строке
size_t strreplace(char* str, int old, int new) {
    size_t count = 0;
    if (!str) return 0;

    char old_char = (char)old;
    char new_char = (char)new;

    for (char* p = str; *p; p++) {
        if (*p == old_char) {
            *p = new_char;
            count++;
        }
    }

    return count;
}

```

Функция замены подстрок более сложная и требует выделения памяти для результирующей строки:

```

#include <stdlib.h>
#include <string.h>
#include <stddef.h>

// Замена подстрок
char* strrepl(char* str, const char* old, const char* new, size_t* count) {

```

```

if (count) *count = 0;

if (!str || !old || !*old || !new) {
    return str;
}

size_t old_len = strlen(old);
size_t new_len = strlen(new);
size_t replacements = 0;

// Подсчет замен и вычисление новой длины
char* pos = str;
while ((pos = strstr(pos, old)) != NULL) {
    replacements++;
    pos += old_len;
}

if (replacements == 0) {
    return str;
}

// Выделение новой памяти
size_t new_size = strlen(str) + replacements * (new_len - old_len) + 1;
char* result = malloc(new_size);
if (!result) {
    return NULL;
}

// Выполнение замен
char* dest = result;
char* src = str;
size_t actual_count = 0;

while (*src) {
    if (strstr(src, old) == src) {
        strcpy(dest, new);
        dest += new_len;
        src += old_len;
        actual_count++;
    }
}

```

```

    } else {
        *dest++ = *src++;
    }
*dest = '\0';

// Обновляем count только при успешной замене
if (count) *count = actual_count;
return result;
}

```

Как видим, код достаточно сложный и может быть реализован через разные стратегии (например, выделять память только если итоговая строка больше исходной, а в остальных случаях использовать уже имеющуюся память). Кроме того, нужно предусмотреть возможность использования пользовательского аллокатора. По этим причинам эта функция не имеет стандартной реализации. Функции стандартной библиотеки представляют собой простые примитивы из которых программист может реализовать любой функционал.

Преобразования чисел в строки и обратно

Для преобразования строк в числа в стандарте сохранились устаревшие и небезопасные функции из `<stdlib.h>` которые не рекомендуется использовать в современных программах:

```

int atoi(const char* str);          (1)
long atol(const char* str);        (2)
long long atoll(const char* str);  (3)
double atof(const char* str);      (4)

```

а в именах функций указывает на то, что `str` - это строка в кодировке ASCII.

1-3) преобразуют строку в целочисленное значение

4) преобразует строку в тип `double`. Функции преобразования в `float` не существует (нет необходимости)

Данные функции имеют безопасные альтернативы:

```

long strtol(const char* str, char** str_end, int base); (5)
long long strtoll( const char* str, char** str_end, int base ); (6)
unsigned long strtoul(const char* str, char** str_end, int base); (7)
unsigned long long strtoull(const char* str, char** str_end, int base); (8)
float strtof(const char* str, char** str_end ); (9)
double strtod(const char* str, char** str_end); (10)
long double strtold(const char* str, char** str_end); (11)

```

5-8) преобразует строку в целочисленное значение. `base` используется для указания системы счисления (наиболее используемые 10, 8, 16 или 0 для автоопределения). `str_end` получает указатель на первый нецифровой символ (можно передавать NULL).
 9-11) преобразует строку в типы с плавающей запятой.

Так как преобразование строкового представления числа с плавающей точкой зависит от локали ("3,141593" в русской локали, "3.141593" - в английской), логично предположить, что функции `strtod`, `strtold`, `strtod` тратят определенные ресурсы на обращение к ним, а значит могут быть неэффективными с точки зрения производительности.

Для того, чтобы корректно конвертировать числа с русским разделителем целой и дробной части необходимо установить русскую локаль:

```

#include <stdio.h>
#include <locale.h>
#include <stdlib.h>
#include <string.h>

int main() {
    const char* russian_number = "3,141593";

    // Сохраняем текущую локаль
    char* old_locale = setlocale(LC_NUMERIC, NULL);
    char* saved_locale = NULL;

    if (old_locale)
        saved_locale = strdup(old_locale);

    setlocale(LC_NUMERIC, "ru_RU.UTF-8");

    // Парсим в русской локали

```

```

char* end = NULL;
double result1 = strtod(russian_number, &end1);

// Восстанавливаем исходную локаль
if (saved_locale) {
    setlocale(LC_NUMERIC, saved_locale);
    free(saved_locale);
}

return 0;
}

```

Этот пример можно оформить в виде функции-обертки над `strtod`. Обратите внимание, что функция `strdup` появилась в стандарте C23, однако всегда поддерживалась в GCC и CLang (как POSIX функция), а вот в MS поддержка появилась в Visual Studio 2015. В переносимом коде ее можно реализовать следующим образом:

```

#include <stdlib.h>
#include <string.h>

// Проверка поддержки C23 strdup
#if defined(__STDC_VERSION__) && __STDC_VERSION__ >= 202311L
    // C23 - strdup стандартизирована
    #define HAS_STRDUP 1
#elif defined(_POSIX_C_SOURCE) && _POSIX_C_SOURCE >= 200809L
    // POSIX 2008 - strdup доступна
    #define HAS_STRDUP 1
#elif defined(_MSC_VER) && _MSC_VER >= 1900
    // MSVC 2015+
    #define HAS_STRDUP 1
#else
    #define HAS_STRDUP 0
#endif

#if !HAS_STRDUP
    // Реализация для старых компиляторов
    char* strdup(const char* str) {
        if (!str) return NULL;

```

```

size_t len = strlen(str) + 1;
char* copy = malloc(len);
if (copy) {
    memcpy(copy, str, len);
}
return copy;
}

#endif

```

Вернемся к примеру с локалями. Очевидно, что такой подход будет плохо работать в многопоточных программах, где возможна частая смена локали, из-за race conditions. То есть требуются более безопасные функции для преобразования строк в числа.

Существуют нестандартные функции `strtol_l`, `strtod_l`, в том числе в компиляторах Microsoft, которые можно обернуть в подобный код (для случая когда конвертируем в "C" локали):

```

#include <stdlib.h>
#include <locale.h>
#include <string.h>

// Проверка доступности POSIX локалей
#if defined(__unix__) || defined(__unix) || defined(__APPLE__) ||
defined(__linux__)
#define HAS_POSIX_LOCALE 1
#include <xlocale.h> // или locale.h может быть достаточно
#else
#define HAS_POSIX_LOCALE 0
#endif

// Проверка доступности Microsoft _l функций
#ifndef _MSC_VER
#define HAS_MS_LOCALE_FUNCTIONS 1
#else
#define HAS_MS_LOCALE_FUNCTIONS 0
#endif

// Универсальная функция парсинга double
double parse_double(const char* str) {

```

```

if (str == NULL) return 0.0;

#if HAS_MS_LOCALE_FUNCTIONS
    // Microsoft версия
    _locale_t c_locale = _create_locale(LC_ALL, "C");
    double result = _strtod_l(str, NULL, c_locale);
    _free_locale(c_locale);
    return result;

#elif HAS_POSIX_LOCALE
    // POSIX версия (Linux, macOS, BSD)
    locale_t c_locale = newlocale(LC_ALL_MASK, "C", NULL);
    if (c_locale == (locale_t)0) {
        // Fallback на стандартный strtod
        return strtod(str, NULL);
    }
    double result = strtod_l(str, NULL, c_locale);
    freelocale(c_locale);
    return result;

#else
    // Универсальная версия с временным переключением локали
    char* old_locale = setlocale(LC_NUMERIC, NULL);
    char* saved_locale = NULL;

    if (old_locale) {
        saved_locale = strdup(old_locale);
    }

    // Временно устанавливаем C локаль
    setlocale(LC_NUMERIC, "C");
    double result = strtod(str, NULL);

    // Восстанавливаем локаль
    if (saved_locale) {
        setlocale(LC_NUMERIC, saved_locale);
        free(saved_locale);
    }

```

```
    return result;
#endif
}
```

Существует также вариант преобразования через функцию `sscanf` (`<stdio.h>`):

```
double parse_double_safe(const char* str) {
    double result;
    sscanf(str, "%lf", &result);
    return result;
}
```

Учтите, что функция `sscanf` использует "C" локаль для чисел, независимо от текущей локали процесса, поэтому русскую запятую в "3,141593" придется заменить вручную или искать другие способы конвертации.

Рекомендации:

- Всегда используйте `strto*` семейство функций вместо `ato*`
- Проверяйте возвращаемые значения и `errno`
- Анализируйте `str_end` для определения успешности преобразования
- Учитывайте текущую локаль
- Пишите безопасный с точки зрения многопоточности код

Для преобразования чисел в строку используются функции из семейства `printf` (`<stdio.h>`):

```
int sprintf(char* buffer, const char* format, ...);
int snprintf(char* restrict buffer, size_t bufsz, const char* format, ...);
```

При этом `format` должен использовать следующие спецификаторы:

Спецификатор	Конвертирует
i или d	знаковые типы <code>char</code> (C99), <code>short</code> , <code>int</code> , <code>long</code> , <code>long long</code> (C99)
u	беззнаковые типы <code>unsigned char</code> (C99), <code>unsigned short...</code>
f	тип <code>double</code> (с плавающей точкой)

Для конвертации чисел с плавающей точкой по-умолчанию принимается точность 6 знаков после запятой. Если нужно изменить точность применяют спецификатор точности (precision). Например, для вывода не более двух знаков используют "%.2f":

```
printf("%.2f", 3.14159); // выведет 3.14
```

`%f` всегда ожидает `double`, а не `float`. Когда вы передаете `float` в variadic functions (как `sprintf`), он автоматически преобразуется в `double` согласно стандарту C (это называется "default argument promotions").

`sprintf` также зависит от локали, а значит имеет проблемы с многопоточностью.

В книге K&R приводится реализация функции `itoa` с намеренно допущенной ошибкой, которую читателю предлагается исправить в качестве упражнения:

```
void itoa(int n, char s[])
{
    int i, sign;
    if ((sign = n) < 0)
        n = -n; // проблема для INT_MIN
    i = 0;
    do {
        s[i++] = n % 10 + '0';
    } while ((n /= 10) > 0);
    if (sign < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverse(s);
}
```

Суть ошибки: наибольшее отрицательное число в дополнительном коде, в частности, `INT_MIN` для 32-битного `int`: -2 147 483 648, не может быть корректно преобразовано в положительное число через `n = -n;` из-за переполнения.

Решений, как это обычно и бывает, может быть несколько. Вот одно из них:

```
void itoa(int n, char s[]) {
    int i, sign;
    unsigned int n2;
```

```

i = 0;

if ((sign = n) < 0) {
    n2 = -n;
}
else {
    n2 = n;
}

do {
    s[i++] = (n2 % 10) + '0';
}
while ((n2 /= 10) > 0);
if (sign < 0) {
    s[i++] = '-';
}
s[i] = '\0';

reverse(s);
}

```

Функция `itoa` не входит ни в один стандарт С или С++ и даже ее нет в POSIX. Тем не менее, она включена в `<stdlib.h>` в качестве расширения и доступна под Linux. Однако в MSVC доступна другая её версия с другой сигнатурой:

```
char* _itoa(int value, char *buffer, int radix);
```

Таким образом, `itoa` из коробки непереносима. Проблема решается выше исправленным вариантом из K&R. Этот код можно немного улучшить, убрав лишний вызов `strlen`:

```

void itoa_kr(int n, char s[]) {
    int len = 0;
    unsigned int un = n < 0 ? (unsigned int)~n + 1u : n;

    do {
        s[len++] = (un % 10) + '0';
    }

```

```
while ((un /= 10) > 0);

if (n < 0)
    s[len++] = '-';

s[len] = '\0';

// reverse
char c;
for (int i = 0, j = len - 1; i < j; i++, j--) {
    c = s[i];
    s[i] = s[j];
    s[j] = c;
}

}
```

Инструменты разработки

Для разработки на С/С++ необходимы редактор кода, компилятор, отладчик и система сборки. В случае интегрированных сред разработки (IDE) это все становится доступно сразу из коробки, но именно знание и понимание работы отдельных базовых инструментов высоко ценится среди профессионалов.

Компиляторы

В зависимости от используемой вами операционной среды доступны следующие компиляторы:

1. **GCC** (GNU Compiler Collection) - это набор компиляторов (C, C++, Fortran, Objective-C, Ada и т.д.) проекта GNU. Используется в большинстве UNIX-подобных систем, включая Linux и MacOS. Распространяется свободно.
2. **Clang** - это фронтенд языков C, C++, Objective-C и других для LLVM. Доступен как в UNIX-подобных системах, так и в Windows. Ввиду серьезного инвестирования со стороны Apple развивается и поддерживается гораздо быстрее, чем GCC. Является компилятором по-умолчанию в среде разработки XCode. Распространяется свободно.
3. **Microsoft Visual C++** - продукт и одновременно компилятор языков С/С++, используемый в основном в Windows и ориентированный в первую очередь на разработку под Windows. Проприетарный, но есть бесплатная версия Visual Studio Community.
4. **MinGW** - это порт GCC для разработки приложений под Windows. Ранее назывался mingw32, но с появлением поддержки 64-битных приложений "32" убрали из названия. Существует также отдельный порт MinGW-w64 с более полной поддержкой WinAPI. Распространяется свободно.
5. **ICC** (Intel C++ Compiler, теперь Intel oneAPI DPC++/C++) - компилятор от Intel с оптимизациями под процессоры Intel. Проприетарный, но есть бесплатная версия.

Отладчики

1. **GDB** (GNU Debugger) используется с GCC (MacOS, Linux) или MinGW (Windows). Распространяется свободно.
2. **LLDB** (отладчик LLVM/Clang) - используется с Clang (MacOS, Linux, Windows). Распространяется свободно.

3. **MSVC Debugger** - входит в состав Visual Studio. Также можно его использовать через Console Debugger (входит в состав Windows SDK и Debugging Tools for Windows). Используется под Windows.
4. **WinDbg** - мощный отладчик от Microsoft, входит в состав Windows SDK и WDK. Удобен для анализа дампов памяти, отладки режима ядра и системного ПО.

Также можно перечислить специализированные отладчики, используемые в основном в хакерской среде для реверс-инжиниринга, анализа вредоносного ПО и взлома программ:

1. **x64dbg** (Open-Source, x86/x64) - по сути альтернатива и замена устаревшего отладчика OllyDBG.
2. **IDA Pro** - своего рода золотой стандарт дизассемблеров (платный, есть бесплатная версия).
3. **Ghidra** - от NSA, бесплатный)
4. **Radare2 + Cutter** (Open-Source, бесплатный)

Системы сборки

Сборщики (Build Systems):

1. **Make** (GNU Make, MS NMake) - одна из первых утилит (1976) для автоматизации сборки программ. До нее обычно собирали программы шеловскими скриптами.
2. **Ninja** - улучшенная версия Make, современная (появилась в 2012 году) и кроссплатформенная. Создана Эваном Мартином, разработчиком из Google.
3. **MSBuild** - система сборки от Microsoft, современная замена NMake. Интегрирована в Visual Studio 2013 и более поздние версии.
4. **Bazel** (Google) - кроссплатформенная система сборки от Google (2015)

Генераторы систем сборки (Build System Generators):

1. **CMake** - на текущий момент (2025 год) это самый популярный генератор проектов для систем сборки. Называть CMake системой сборки неверно, т.к. она фактически только конфигурирует и создает необходимые файлы, передаваемые непосредственно на вход системам сборки. Преимущественно используется в C и C++ проектах. Использует файлы CMakeLists.txt для описания проекта и сборки.
2. **Meson** - альтернатива CMake, более современная, более быстрая, но менее гибкая, если сравнивать с CMake. Использует файлы meson.build для описания проекта и сборки.
3. **Autotools** (GNU Build System) - традиционная система для UNIX (`./configure && make`), но актуальная только для UNIX-подобных систем. Фактически используется

только в легаси-проектах.

4. **QMake** - система сборки Qt-проектов. Тоже почти утратила актуально. Даже в Qt6 уже используется CMake.
5. **PreMake** - достаточно простой генератор в виде утилиты командной строки, использующей скрипты Lua. Подходит для небольших проектов C/C++/C#

Интегрированные среды разработки (IDE)

1. **Visual Studio** - мощная среда разработки от Microsoft (не путать с Visual Studio Code). Со времени своего первого выхода в 1987 эта среда развилась в целую экосистему разработки на разных языках программирования и для разных систем, включая разработку под Linux. Поддерживает помимо родного компилятора MSVC еще и CLang.
2. **CLion** - среда разработки на C/C++ от JetBrains (2015), на базе IntelliJ IDEA. Соответственно написана на Java. Тяжеловесная и перегруженная, требуется хорошее железо для работы с ней.
3. **Xcode** - среда разработки от Apple. В качестве основного компилятора использует CLang, но можно подключить и GCC.
4. **Code::Blocks** - свободная кроссплатформенная IDE, написанная на C++ с wxWidgets. Поддерживает компиляторы GCC, Clang, MSVC. Более простая по своим возможностям, если сравнивать с MSVS, CLion, XCode, и менее требовательная по ресурсам. Имеет устаревший интерфейс в духе начала 2000-х.
5. **RAD Studio** — среда быстрой разработки приложений (RAD) от фирмы Embarcadero Technologies. Ориентирована на Windows, объединяет Delphi и C++ Builder в единую интегрированную среду разработки.
6. **Qt Creator** - кроссплатформенная среда разработки, входит в состав фреймворка Qt, используемого для разработки кроссплатформенных десктоп и других приложений. Написана на C++ и Qt. Годится, в том числе для написания программ, не использующих Qt.
7. **Eclipse CDT** - бесплатная кроссплатформенная среда разработки на базе платформы Eclipse. CDT (# C/C++ Development Tooling) вышел как плагин Eclipse в 2002 году. Изначально использовал GCC, затем подключили другие компиляторы и отладчики. Написана на Java, поэтому очень медленная. Устаревший интерфейс.
8. **NetBeans** - бесплатная open-source среда разработки. Написана на Java и изначально для разработки на Java и даже была официальной Java-IDE, когда Java принадлежала Sun Microsystems. В настоящее время развивается под крылом Apache. Работает немного медленнее CLion, но заметно быстрее Eclipse CDT.

9. **CodeLite** - альтернатива Code::Blocks. Аскетична по возможностям, написана на C++ с wxWidgets. Подойдет на embedded разработки, начинающим разработчикам, а также для несложных проектов.
10. **IntelliJ IDEA** - бесплатная кроссплатформенная среда разработки от JetBrains (2001), изначально заточенная для разработки на Java. На базе этой среды теперь выпускается сразу несколько инструментов, в том числе CLion и другие IDE от JetBrains, Android Studio от Google и даже десктопная Sber GigalDE.

Редакторы

Существует достаточно большой список терминальных текстовых редакторов, т.е. работающих в терминале без графической оболочки. Такие редакторы очень популярны в среде UNIX-подобных систем и среди опытных олд-скульных разработчиков. Среди этих редакторов можно выделить модальные текстовые редакторы, в которых переключаются режимы (mode) работы, например: выделение, редактирование, навигация и т.д., и простые редакторы для работы в одном единственном универсальном режиме.

Модальные текстовые редакторы:

1. **Vim/NeoVim** - Vim стал развитием редактора vi. NeoVim - это более современный форк Vim. Редакторы позволяют очень быстро писать код при условии хорошего навыка слепой печати и знания наизусть определенного набора комбинаций клавиш и команд, без использования мыши. Вот серьезно, мышь просто не нужна).
2. **Emacs** - появился примерно в то же время, что и редактор vi, в 1976 году, задолго до появления Vim (1991). Написан Ричардом Столлманом и Гаем Стилом. Был переписан на С в середине 80-х и стал основным редактором в проекте GNU. Достаточно сложный для освоения новичками, но преданно любим программистами-мастадонтами.
3. **Helix** - относительно свежий проект. Редактор похож на Vim/NeoVim. Написан на Rust. Кодовая база намного меньше и хорошо спроектирована. Пока набирает популярность и имеет все шансы потеснить Vim/NeoVim.

Немодальные текстовые редакторы:

1. **Nano** - простой редактор от проекта GNU. Имеет подсветку синтаксиса (если включить в настройках)
2. **Micro** - более современный редактор с плагинами и подсветкой синтаксиса, а также поддержкой мыши.

Графические текстовые редакторы:

1. **Visual Studio Code** - бесплатный редактор, написанный на TypeScript и использующий фреймворк Electron (на базе Chromium и Node.js). Работает в Linux, MacOS, Windows. Благодаря огромному количеству плагинов получил невероятную популярность среди разработчиков.
2. **Sublime** - легкий, быстрый и в то же время мощный текстовый редактор (написан на C++ и Python). Работает также в Linux, MacOS, Windows. Бесплатный для ознакомления, потом начинает напоминать, что неплохо бы купить лицензию, но при этом функциональность не ограничивает.
3. **Notepad++** - бесплатный редактор под Windows, с подсветкой синтаксиса (используется Scintilla), с возможностью подключать компиляторы, отладчики, запускать плагины (набор невелик, но покрывает большинство задач).

Тестирование

В разработке программного обеспечения применяются различные виды тестирования: модульное, интеграционное, системное, регрессионное, приемочное, нагрузочное и т.д. Модульное тестирование (unit testing) занимает особое и базовое место среди прочих.

Юнит-тесты проверяют корректную работу отдельных функций и модулей. Их написанием обычно занимаются разработчики. Для этого используются различные фреймворки. Вот некоторые наиболее популярные из них:

- **Unity (+ CMock)** - идеален для embedded разработки, header-only
- **Check** - современный (замена устаревшего и заброшенного CUnit, но написан с нуля), поддерживает фикстуры, запускает тесты в отдельных процессах
- **Criterion** - современный, поддерживает фикстуры и параметризацию отчетов, создает красивые отчеты
- **CMocka** - поддерживает фикстуры, есть мокинг

Для C++ используют чаще:

- **gtest (+ gmock)** - популярный фреймворк от Google.
- **Boost.Test** - фреймворк проекта Boost

Для проверки покрытия кода тестами используют специальные утилиты.

gcov - утилита из состава GCC, собирающая данные о покрытии кода во время выполнения программы. Программа компилируется с определенными флагами (-fprofile-arcs -fprofile-coverage) и при ее выполнении формируются файлы со статистикой (.gcda + .gcdno). Далее запускается gcov, который создает текстовый файл с отчетом.

lcov - это утилита (устанавливается отдельно от GCC), которая позволяет генерировать html-отчет для удобства анализа в браузере (при помощи утилиты genhtml, входящей в состав lcov).

В Clang используется **Clang Source-based Code Coverage** (аналог gcov). Для сбора данных на при компиляции нужны флаги: -fprofile-instr-generate -fcoverage-mapping. Далее используют утилиту llvmt-profdta для объединения профайлов, а llvmt-cov — генерирует отчёты (текст/HTML)

Профилирование

Для динамического анализа, в частности анализа производительности, а также потребления памяти используют специальные программы-профилировщики (profilers).

gprof - классический профайлер производительности пользовательского кода, использует метод инструментирования, поэтому ресурсоемкий. Входит в состав binutils.

gprofng - современная альтернатива gprof. Более мощный, поддерживает многопоточность, системные вызовы. Входит в состав binutils.

valgrind - это фреймворк для создания инструментов динамического анализа. Часто valgrind называют отладчиком, однако, он не позволяет управлять потоком выполнения программы, устанавливать точки останова, просматривать стеки вызова, регистры и переменные во время выполнения программы. Valgrind скорее является профилировщиком, точнее ядром для создания профилировщиков, т.к. предоставляет средства, с помощью которых можно создавать свои инструменты для профилирования. Из коробки в valgrind доступны ряд готовых инструментов, а именно:

- memcheck (используется по-умолчанию) - профилировщик памяти: утечки, неинициализированная память, доступ вне выделенной памяти, за пределами границы стека, двойное освобождение, использование после освобождения и т.д.
- massif - профилировщик кучи (динамической памяти)
- cachegrind - профилировщик кэша
- callgrind - профилировщик производительности (медленный, т.к. использует инструментирование, а не сэмплирование, но зато точный)
- helgrind - профилировщик состояния гонки (в многопоточном выполнении кода)
- и другие

perf - профилировщик из Linux Kernel Tools, позволяющий анализировать как пользовательский код, так и код ядра. Использует метод сэмплирования (использует аппаратные счетчики процессора). Может показывать горячие участки кода (где больше

всего тратится процессорного времени). Если сравнивать с gprof, то perf предпочтительнее использовать, он более современный и позволяет проводить более детальный анализ производительности.

Intel VTune - мощный профилировщик от Intel с графической средой под Windows и Linux. Позволяет находить узкие места в производительности, особенно незаменим для анализа многопоточных программ (взаимоблокировки, ожидания, анализ параллельности). Использует сэмплирование, а также аппаратные счетчики.

Прочие инструменты и утилиты:

В Linux при компиляции программ используется как минимум компоновщик (линкер) ld из состава **binutils**, который является набором различных полезных программ. Вот некоторые из них:

- addr2line - позволяет преобразовать адреса в имена файлов и номера строк при наличии отладочных символов
- ar - архиватор, используемый для создания статических библиотек
- gprof - профайлер.
- gprofng - более современный профайлер.
- objdump - для анализа исполняемых и объектных файлов (ELF, PE и других форматов)
- и другие

При установке gcc автоматически установится и пакет binutils, а также стандартная библиотека си (libc-dev) и препроцессор (cpp):

```
sudo apt update  
sudo apt install gcc
```

Обычно к gcc добавляют еще make, рассмотренный выше:

```
sudo apt install gcc make
```

В Linux-дистрибутивах на основе Debian часто рекомендуют устанавливать метапакет build-essential для установки gcc и сопутствующих пакетов:

```
sudo apt install build-essential
```

build-essential устанавливает:

- GCC - коллекция компиляторов, который через зависимости ставит binutils
- g++ - компилятор C++
- libc-dev - стандартная библиотека си
- make - система сборки
- dpkg-dev - инструмент сборки deb-пакетов

Во-первых, не на всех дистрибутивах есть этот метапакет. Во-вторых, в нем многое не хватает. Поэтому, имеет смысл, как минимум, расширить установку следующим образом:

```
sudo apt install build-essential gdb valgrind cmake
```

Либо установить необходимые пакеты вручную (если вам пока не нужен сборщик deb-пакетов)

```
sudo apt install gcc g++ make gdb valgrind cmake
```

Литература

1. Керниган Б., Ритчи Д. Язык программирования С, 2-е издание, 1989 : Пер. с англ. - М. : Издательский дом "Вильямс", 2009. - 304 с. : ISBN 978-5-8459-0891-9
2. Прата С. Язык программирования С. Лекции и упражнения, 6-е изд. : Пер. с англ. - М. : ООО "И.Д. Вильямс", 2015. - 928 с. : ISBN 978-5-8459-1950-2
3. Дейтел П., Дейтел Х. С для программистов с введением в С11 : Пер. с англ. - М. : ДМК Пресс, 2014. - 544 с. : ISBN 978-5-97060-073-3
4. Сикорд Р. Эффективный С. Профессиональное программирование. : Пер. с англ. - СПб. : Питер, 2022. - 304 с. : ISBN 978-5-4461-1851-9
5. Сикорд Р. Безопасное программирование на С и С++, 2-е изд. : Пер. с англ. - М. : ООО "И.Д. Вильямс", 2015. - 496 с. : ISBN 978-5-8459-1908-3