Оглавление

[1. Windows HOOK. Назначение и особенности использования. 1](#_Toc186077351)

[2. Актуальность синхронизации потоков при параллельной обработке данных 4](#_Toc186077352)

[3. Статические и динамические библиотеки. Область применения и особенности использования 5](#_Toc186077353)

[4. Многопоточный режим работы компьютера. преимущества использования многопоточности 7](#_Toc186077354)

[5. Особенности использования строкового типа данных. Массивы строк 8](#_Toc186077355)

[6. Особенности разработки программ с использованием API информационной системы (основные отличия от программ, не использующих API) 9](#_Toc186077356)

[7. Особенности разработки программ для работы с системными событиями использования мыши и клавиатуры. 10](#_Toc186077357)

[8. Подходы к обработке исключений в программах, написанных на си 11](#_Toc186077358)

[9. Понятие «системное программирование». предназначение и специфика системного программирования 13](#_Toc186077359)

[10. Предназначение и особенности использования директив препроцессора и заголовочных файлов. 14](#_Toc186077360)

[11. Принципы взаимодействия операционной системы с прикладными программами (в том числе раскрыть понятия “процесс”, “поток”, “дескриптор”) 18](#_Toc186077361)

[12. Принципы функционирования системного программного обеспечения (на основе анализа отличий от других типов ПО) 19](#_Toc186077362)

[13. Назначение реестра Windows. Особенности использования системного реестра программами. Структура системного реестра. Типы данных в системном реестре. 20](#_Toc186077363)

[14. Системный буфер обмена. Особенности взаимодействия программного обеспечения с системным буфером обмена 22](#_Toc186077364)

[15. Структуры данных. Актуальность использования структур. Передача структуры как параметра. 23](#_Toc186077365)

[16. Типы данных в ОС Windows. Назначение и использование системных диалоговых окон. 27](#_Toc186077366)

[17. Указатель как тип данных. Особенности работы с указателями 29](#_Toc186077367)

[18. Массив как тип данных. Особенности использования массивов при разработке программ на си. 31](#_Toc186077368)

[19. Линейный односвязный список. Особенности создания и примеры использования. 33](#_Toc186077369)

[20. Указатель на функцию. Назначение и примеры использования. 36](#_Toc186077370)

[21. Файл как тип данных. Использование файлов в работе программного обеспечения. 38](#_Toc186077371)

[22. Функции для работы с файлами WinAPI. Синхронный и асинхронный доступ к файлу. 39](#_Toc186077372)

[23. Функции для работы с файлами в UNIX-подобных операционных системах. 43](#_Toc186077373)

[24. Анонимные каналы (в Windows и Linux). Назначение и использование. 46](#_Toc186077374)

[25. Именованные каналы (в Windows и Linux). Назначение и использование. 48](#_Toc186077375)

[26. Создание процессов в UNIX-подобных операционных системах. 50](#_Toc186077376)

[27. Создание процессов в ОС семейства Windows 53](#_Toc186077377)

[28. Функция запуска приложений. Разновидности функции запуска. Аргументы командной строки 55](#_Toc186077378)

[29. Сигналы. Особенности создания и использования сигналов. 57](#_Toc186077379)

[30. Фоновые процессы (в Windows и Linux). Актуальность их создания и использования 59](#_Toc186077380)

**Что необходимо запомнить**

Тип HANDLE обозначает 32-разрядное целое, используемое в качестве дескриптора.

Дескриптор выступает в качестве идентификатора определенного ресурса.

Процесс – это выполняемая программа, имеющая собственное виртуальное адресное пространство, код, данные, а также потребляющие ресурсы ОС (файлы, объекты синхронизации).

Поток — это, по сути, последовательность инструкций, которые выполняются параллельно с другими потоками. Каждая программа создает по меньшей мере один поток: основной, который запускает функцию main. Программа, использующая только главный поток, является однопоточной; если добавить один или более потоков, она станет многопоточной.

# **Windows HOOK. Назначение и особенности использования.**

В операционной системе Microsoft® Windows™ хуком называется механизм перехвата особой функцией событий (таких как сообщения, ввод с мыши или клавиатуры) до того, как они дойдут до приложения.

Эта функция может затем реагировать на события и, в некоторых случаях, изменять или отменять их.

Функции, получающие уведомления о событиях, называются **фильтрующими функциями** и различаются по типам перехватываемых ими событий. Пример - фильтрующая функция для перехвата всех событий мыши или клавиатуры.

Чтобы Windows смогла вызывать функцию-фильтр, эта функция должна быть установлена - то есть, прикреплена - к хуку (например, к клавиатурному хуку). Прикрепление одной или нескольких фильтрующих функций к какому-нибудь хуку называется **установкой хука**.

Если к одному хуку прикреплено несколько фильтрующих функций, Windows реализует стек функций (функция, прикрепленная последней, оказывается в начале очереди, а самая первая функция - в ее конце)

Когда к хуку прикреплена одна или более функций-фильтров и происходит событие, приводящее к срабатыванию хука, Windows вызывает первую функцию из очереди функций-фильтров. Это действие называется вызовом хука.

Для установки и доступа к фильтрующим функциям приложения используют функции **SetWindowsHookEx** и **UnhookWindowsHookEx**.

Хуки предоставляют мощные возможности для приложений Windows. Приложения могут использовать хуки в следующих целях:

* Обрабатывать или изменять все сообщения в системе и программах, использующих её API (все виды сообщений), получаемые (обрабатывать ввод с клавиатуры, мыши, управление окнами и их частями);
* Вызывать окна сообщений;
* Обрабатывать, изменять или удалять, записывать или проигрывать клавиатурные и мышиные события (WH\_JOURNALRECORD, WH\_JOURNALPLAYBACK).
* Реагировать на определенные действия системы, делая возможным разработку приложений компьютерного обучения - computer-based training (WH\_CBT).
* Предотвратить вызов другой функции-фильтра (WH\_DEBUG).

Приложения уже используют хуки для следующих целей:

* Обеспечить поддержку клавиатурных макросов;
* Управлять сообщениями системы пользователю.
* Симулировать мышиный и клавиатурный ввод. Хуки - единственный надежный способ симуляции этих действий. Если попытаться имитировать их через посылку сообщений, не будет происходить обновление состояния клавиатуры или мыши во внутренних структурах Windows, что может привести к непредсказуемому поведению. Если для воспроизведения клавиатурных или мышиных событий используются хуки, эти события обрабатываются в точности так, как и настоящий ввод с клавиатуры или мыши. Microsoft Excel использует хуки для реализации макрофункции SEND.KEYS.

**Как пользоваться хуками**

Чтобы пользоваться хуками, вам необходимо знать следующее:

* Как использовать функции Windows для добавления и удаления фильтрующих функций из очереди функций хука.
* Какие действия должна будет выполнить фильтрующая функция, которую вы устанавливаете.
* Какие существуют виды хуков, что они могут делать, и какую информацию (параметры) они передают вашей функции.

**Функции Windows для работы с хуками**

Приложения Windows используют функции **SetWindowsHookEx**, **UnhookWindowsHookEx**, и **CallNextHookEx** для управления очередью функций-фильтров хука. До версии 3.1 Windows предоставляла для управления хуками функции **SetWindowsHook**, **UnhookWindowsHook**, и **DefHookProc**. Хотя эти функции до сих пор реализованы в Win32, у них меньше возможностей, чем у их новых (**Ex**) версий. Всегда старайтесь использовать только эти новые функции в своих проектах.

**SetWindowsHookEx** и **UnhookWindowsHookEx** описаны ниже. Обратитесь к разделу "Вызов следующей функции в очереди фильтрующих функций" за информацией по **CallNextHookEx**.

**SetWindowsHookEx**

Функция **SetWindowsHookEx** добавляет функцию-фильтр к хуку.

Для удаления функции-фильтра из очереди хука вызовите функцию **UnhookWindowsHookEx****.** Эта функция принимает хэндл хука, полученный от SetWindowsHookEx и возвращает логическое значение, показывающее успех операции. На данный момент UnhookWindowsHookEx всегда возвращает TRUE.

**Фильтрующие функции**

Фильтрующие (*хуковые*) функции — это функции, прикрепленные к хуку. Из-за того, что эти функции вызываются Windows, а не приложением, их часто называют **функциями обратного вызова** (callback functions)

# **Актуальность синхронизации потоков при параллельной обработке данных**

Поток — это, по сути, последовательность инструкций, которые выполняются параллельно с другими потоками.

Синхронизация потоков при параллельной обработке данных остается актуальной из-за увеличения количества многоядерных процессоров в современных компьютерах.

Параллельная обработка данных позволяет улучшить производительность приложений, однако она также может привести к возникновению проблем с согласованностью данных и конфликтами доступа к общим ресурсам.

Синхронизация потоков позволяет контролировать доступ к общим данным и избежать гонок данных, когда несколько потоков пытаются одновременно изменить один и тот же ресурс. Различные механизмы синхронизации, такие как мьютексы, семафоры, условные переменные и атомарные операции, используются для обеспечения правильного доступа к данным и предотвращения возможных проблем.

Таким образом, синхронизация потоков остается важной темой в разработке программного обеспечения, особенно в контексте параллельных вычислений и многопоточных приложений. Разработчики должны учитывать потенциальные проблемы с согласованностью данных и использовать соответствующие методы синхронизации для обеспечения корректной и эффективной работы своих приложений.

# **Статические и динамические библиотеки. Область применения и особенности использования**

Определения

**Библиотека** — это кодовая база, строительный элемент программы, скомпилированный под одну или несколько архитектур.

В случае **статических библиотек** код, используемый приложением, копируется в исполняемый файл во время компиляции. То есть после компиляции исполняемый файл **не нуждается** в поддержке внешним кодом или бинарниками, т. к. все используемые функции уже находятся в нем. Конечно, это будет и минусом статической библиотеки, ведь если кодовая база изменится, то для работоспособности исполняемого файла его придется заново перекомпилировать.

**Динамические библиотеки** же отличаются от статических тем, что они "линкуются" с исполняемым бинарником, т.е. код нашей библиотеки не попадает в исполняемый файл, а подгружается во время выполнения. В результате размер исполняемого бинарника меньше, а так как код загружается только в момент обращения к нему, то время запуска обычно уменьшается; но если все динамические библиотеки загружаются при запуске приложения, то это может сильно повлиять на время запуска приложения.

**Статические библиотеки (.lib в Windows, .a в Unix-подобных системах):**

Область применения: подходят для **небольших проектов**, где размер исполняемого файла не является критичным, или когда требуется полная независимость от внешних библиотек. Идеально подходят для ситуаций, когда вы хотите убедиться, что у вас есть определенная версия библиотеки и не будет конфликтов с другими версиями, используемыми другими частями системы.

В случае статических будет:

* Код статической библиотеки скомпилируется и скопируется непосредственно в исполняемый файл во время процесса компоновки (линковки).
* Исполняемый файл **становится самодостаточным** — ему не нужны дополнительные файлы для работы.
* Размер исполняемого файла **увеличивается на размер** статической библиотеки.
* Обновление библиотеки требует перекомпиляции и перелинковки всего проекта.
* Меньше проблем с зависимостями и версиями, так как код библиотеки напрямую встраивается в приложение.

**Динамические библиотеки (.dll в Windows, .so в Linux, .dylib в macOS):**

Область применения: предпочтительны для **больших проектов**, где важен размер исполняемого файла, или когда библиотека может использоваться несколькими программами одновременно. Позволяют обновлять библиотеки без перекомпиляции всего проекта.

В случае динамических будет:

* Код динамической библиотеки остается **отдельным файлом**.
* Исполняемый файл **содержит только ссылки** на функции из динамической библиотеки.
* Исполняемый файл **меньше по размеру**, чем в случае со статическими библиотеками.
* Обновление библиотеки не требует перекомпиляции программ, использующих её. Достаточно заменить файл динамической библиотеки.
* Требуется, чтобы динамическая библиотека была доступна во время выполнения программы (обычно находится в стандартных местах, указанных в переменных окружения, таких как PATH или LD\_LIBRARY\_PATH).
* Могут возникнуть проблемы с зависимостями и версиями, если разные программы используют разные версии одной и той же библиотеки (конфликты версий).

# **Многопоточный режим работы компьютера. преимущества использования многопоточности**

**Многопоточность** — это способ организации работы программы, при котором несколько потоков выполнения (потоков) работают одновременно внутри одного процесса. Это не то же самое, что несколько процессов, работающих параллельно. Потоки внутри одного процесса делят общие ресурсы (память, файлы и т.д.).

Преимущества использования многопоточности:

* **Улучшенная отзывчивость приложения**: даже если один поток заблокирован (например, ожидает ввода-вывода), другие потоки могут продолжать свою работу. Это делает приложение более отзывчивым на действия пользователя, так как пользовательский интерфейс не замирает. Например, веб-браузер может загружать страницы в фоновом режиме, не останавливая работу с другими страницами или интерфейсом.
* **Ускорение выполнения**: если задачи можно разделить на независимые части, многопоточность может существенно ускорить выполнение программы, особенно на многоядерных процессорах. Потоки могут одновременно использовать разные ядра процессора, выполняя вычисления параллельно. Это эффективно, когда задача может быть разбита на независимые подзадачи, например, обработка изображений или выполнение нескольких сложных вычислений.
* **Эффективное использование ресурсов**: Многопоточность позволяет более эффективно использовать ресурсы системы. Например, потоки могут использовать ресурсы системы (например, CPU, память и сетевые соединения) параллельно, избегая ненужной задержки. За счет этого увеличивается производительность.
* **Простая модель параллелизма**: в некоторых случаях организация параллельной работы в виде нескольких процессов может быть сложнее, чем в виде нескольких потоков. Например, создание и уничтожение процессов, а также передача данных между процессами требуют больше ресурсов и времени, чем обмен данными между потоками.
* **Упрощение разработки сложных приложений**: Разделение задач на независимые потоки упрощает разработку сложных приложений, которые требуют одновременной обработки нескольких процессов, например, приложения с графическим интерфейсом пользователя (GUI), обработки сетевых запросов или управления устройствами.

# **Особенности использования строкового типа данных. Массивы строк**

В языке C строковый тип данных не является встроенным типом, как, например, int или float. Строки в C представляются как **массивы символов**, заканчивающиеся нулевым символом **(\0)**. Это накладывает определенные особенности на их использование.

Особенности использования строкового типа данных:

* **Нуль-терминация**: Каждый массив символов, представляющий строку, должен заканчиваться нулевым символом (\0). Этот символ указывает на конец строки. Без него функции работы со строками могут читать память за пределами выделенного массива, что приведет к неопределенному поведению программы (сегментация памяти).
* **Неявное преобразование**: В C нет автоматического преобразования между строками и другими типами данных. Необходимо использовать специальные функции для работы со строками, например, функции из заголовочного файла string.h.

Массивы строк в C:

**Массив строк в C** — это двумерный массив символов, где каждая строка заканчивается нулевым символом.

Объявление:

|  |
| --- |
| char strings[5][20]; *// Массив из 5 строк, каждая длиной не более 19 символов (+ нулевой символ)*  char strings[5][20] = {  "Hello",  "World",  "This",  "is",  "an array of strings"  }; |

# **Особенности разработки программ с использованием API информационной системы (основные отличия от программ, не использующих API)**

**API (англ. Application Programming Interface** — программный интерфейс приложения) — это набор способов и правил, по которым различные программы общаются между собой и обмениваются данными.

**Интерфейс** — это граница между двумя функциональными системами, на которой происходит их взаимодействие и обмен информацией.

Практически все операционные системы (Unix, Windows, Mac OS, и т. д.) имеют API, с помощью которого программисты могут создавать приложения для этой операционной системы. **Главный API операционных систем — это множество системных вызовов.**

В индустрии программного обеспечения общие стандартные API для стандартной функциональности имеют важную роль, так как они гарантируют, что все программы, использующие общий API, будут работать одинаково хорошо или, по крайней мере, типичным привычным образом. В случае API графических интерфейсов это означает, что программы будут иметь похожий пользовательский интерфейс, что облегчает процесс освоения новых программных продуктов.

При использовании API информационной системы программы могут обмениваться данными с другими приложениями или внешними сервисами, что позволяет расширить функциональность программы и улучшить взаимодействие с внешними системами, например:

* **работать с файловой системой;**
* **отрисовывать графику;**
* **хранить данные;**
* **использовать сетевые возможности;**
* **воспроизводить аудио и так далее.**

Использование системного API в разработке прикладных программ – это наиболее близкий способ взаимодействия с системой.

Соответственно, главным отличием программ, использующих API, от не использующих его, более свободный доступ к системным ресурсам и возможность обработки системных событий на более низком уровне, потому что больше интерфейсов взаимодействия с системой.

# **Особенности разработки программ для работы с системными событиями использования мыши и клавиатуры.**

Использование WinAPI: для перехват а и обработки системных событий мыши и клавиатуры в Windows используется WinAPI. Это означает, что разработчику необходимо быть знакомым с функциями WinAPI, такими как **SetWindowsHookEx**, **GetMessage**, **TranslateMessage**, **DispatchMessage** и другими.

* **Права администратора**: Для установки глобальных хуков (например, для перехвата клавиш мыши или клавиатуры) требуются права администратора. Пользователь должен быть предупрежден о том, что программа будет иметь доступ к системным событиям использования мыши и клавиатуры.
* **Безопасность**: Использование глобальных хуков для перехвата клавиш мыши и клавиатуры может представлять потенциальную угрозу безопасности. Поэтому разработчику необходимо быть особенно осторожным при разработке подобных программ, чтобы избежать возможного злоупотребления или нарушения приватности пользователей.
* **Обработка событий в фоновом режиме**: При разработке программ для работы с системными событиями использования мыши и клавиатуры важно учитывать, что программа будет работать в фоновом режиме и должна быть способна обрабатывать события даже когда пользователь работает с другими приложениями.
* **Тестирование и отладка**: поскольку такие программы могут взаимодействовать с системой на низком уровне, тестирование и отладка играют особенно важную роль. Разработчику необходимо убедиться, что программа корректно обрабатывает все возможные сценарии использования мыши и клавиатуры.

# **Подходы к обработке исключений в программах, написанных на си**

**Windows**:

* Проверка возвращаемых значений функций API: большинство функций Windows API возвращают логическое значение (TRUE / FALSE) или значение типа HRESULT (в COM). Значение FALSE или код ошибки в HRESULT указывают на сбой. Необходимо проверять эти значения после каждого вызова функции.

|  |
| --- |
| BOOL result = CreateFile(filename, GENERIC\_READ, 0, NULL, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);  if (result == FALSE) {  DWORD error = GetLastError();  // Обработка ошибки с помощью error  printf("Ошибка создания файла: %lu\n", error);  } |

* **GetLastError():** После вызова функции, завершившегося ошибкой, GetLastError() возвращает код последней ошибки.
* **Проверка кодов возврата функций**: некоторые функции могут возвращать значение int больше 0, где каждое число – это своя ошибка.

**Linux:**

* **Проверка возвращаемых значений функций**: многие функции в POSIX-совместимых системах (включая Linux) возвращают -1 в случае ошибки, устанавливая глобальную переменную errno.

|  |
| --- |
| int fd = open("myfile.txt", O\_RDONLY);  if (fd == -1) {  perror("Ошибка открытия файла"); // perror выводит описание ошибки из errno  exit(1);  } |

* **errno**: Глобальная переменная errno (объявленная в <errno.h>) содержит код последней ошибки. Функция strerror() преобразует код ошибки в строку с описанием.

|  |
| --- |
| if (fd == -1) {  fprintf(stderr, "Ошибка открытия файла: %s\n", strerror(errno));  } |

* **Сигналы**: Linux использует сигналы для обработки асинхронных событий, в том числе ошибок. Программа может устанавливать обработчики сигналов, чтобы реагировать на различные типы ошибок (например, SIGSEGV — нарушение сегментации).

|  |
| --- |
| struct sigaction sa;  sigemptyset(&sa.sa\_mask);  sa.sa\_flags = 0;  sa.sa\_handler = my\_signal\_handler;  sigaction(SIGSEGV, &sa, NULL); |

* **Проверка кодов возврата функций в библиотеках**: как и в Windows, библиотеки могут возвращать собственные коды ошибок.

# **Понятие «системное программирование». предназначение и специфика системного программирования**

Системное программирование — это направление программирования, связанное с созданием программного обеспечения, которое работает на низком уровне и взаимодействует с аппаратным обеспечением компьютера или операционной системой.

Основное предназначение системного программирования — это создание системного программного обеспечения, такого как операционные системы, драйверы устройств, компиляторы, ассемблеры и другие инструменты, необходимые для работы компьютерной системы.

Специфика системного программирования заключается в работе на низком уровне аппаратного обеспечения, использовании специфических языков программирования (таких как ассемблер), оптимизации производительности и ресурсов, обеспечении надежности и безопасности программного обеспечения.

Системные программисты также должны иметь глубокие знания об аппаратных особенностях компьютера, операционных системах, сетевых протоколах и других технических аспектах.

Основные задачи системного программирования включают в себя создание программ, которые управляют ресурсами компьютера (памятью, процессором, устройствами ввода-вывода), обеспечивают взаимодействие между различными компонентами компьютерной системы, обеспечивают безопасность и защиту от ошибок, оптимизируют производительность и ресурсы.

Системные программы представляют собой **комплекс управляющих и обрабатывающих программ. описаний и инструкций, обеспечивающих функционирование вычислительной системы**, а также предоставляющих пользователю ПЭВМ определенные услуги. Программы системного программного обеспечения различаются по функциональному назначению и характеру исполнения.

Можно выделить следующие виды системных программ:

* **операционные системы;**
* **драйверы**, предназначенные для расширения возможностей операционных систем по управлению аппаратными средствами ПЭВМ (клавиатура, монитор, мышь и т. д.);
* **программы-оболочки, которые обеспечивают более удобный и наглядный, по сравнению с реализуемым в операционной системе пользовательский интерфейс;**

# **Предназначение и особенности использования директив препроцессора и заголовочных файлов.**

Директивы препроцессора и заголовочные файлы используются в системном программировании для обеспечения модульности, повторного использования кода и управления зависимостями между различными компонентами программы.

**Препроцессорные директивы**

Директива #include вставляет код из указанного файла в текущий файл, то есть, просто подключив другой файл, мы можем пользоваться его функциями, классами, переменными. Заголовочные файлы обычно находятся либо в текущей директории, либо в стандартном системном каталоге.

Подключение заголовочных файлов выполняется **во время компиляции**, либо как файл, который является частью вашего проекта. Эта функция зависит от конкретной реализации вашего компилятора, поэтому для получения более подробной информации, покопайтесь в настройках своего компилятора.

Если подключаемый файл не найден, процесс компиляции завершается с ошибкой.

Директива #define принимает две формы:

1. определение констант;
2. определение макросов.

Определение констант

|  |
| --- |
| #define nameToken value |

При использовании имени константы — nameToken, оно будет заменено значением value, то есть, грубо говоря — это та же самая переменная, значение которой изменить нельзя. Смотрим пример использования константы:

|  |
| --- |
| #include <iostream>  #define TEXT "Марс" // определение константы    int main()  {  std::cout << TEXT;  return 0;  } |

|  |
| --- |
| #define nameMacros(arg1, arg2, ... ) expression |

К примеру, определим макрос, который будет возвращать максимальное из двух значений.

|  |
| --- |
| #define MAX(num1, num2) ((num1) > (num2) ? (num1) : (num2)) |

Внимание, для определения многострочного макроса, в каждой строке, в конце, должен ставиться символ , что сообщает препроцессору, что макрос еще не завершен.

* **Директива #undef**

Директива #undef переопределяет константу или препроцессорный макрос, ранее определенный с помощью директивы #define.

|  |
| --- |
| #undef nameToken |

Давайте посмотрим пример использования директивы #undef:

|  |
| --- |
| #define E 2.71828 // раннее определенный макрос  int sumE = E + E; // обращение к макросу  #undef E // теперь E - не макрос |

Как правило, директива #undef используются для **снятия, ранее определенной константы или макроса**, в небольшой области программы. Это делается для того, чтобы для всей программы, макроc или константа оставались, а для некоторой области, эти же макрос или константа могут быть переопределены.

Небезопасно было бы во всей программе переопределять константу, но в короткий области, это сравнительно безопасно. Директива #undef является единственным способом создания этой области, так как область действия макросов или констант действует от директивы #define до #undef.

* **Директива #if**

|  |
| --- |
| #if value  // код, который выполнится, в случае, если value - истина  #elsif value1  // этот код выполнится, в случае, если value1 - истина  #else  // код, который выполнится в противном случае  #endif |

Директива #if проверяет, является ли значение value истиной и, если это так, то выполняется код, который стоит до закрывающей директивы #endif. В противном случае, код внутри #if не будет компилироваться, он будет удален компилятором, но это не влияет на исходный код в исходнике.

Обратите внимание, что в #if могут быть вложенные директивы #elsif и #else. Ниже показан пример кода для комментирования блоков кода, используя следующую конструкцию:

|  |
| --- |
| #if 0  // код, который необходимо закомментировать  #endif |

Если у вас в программе есть блоки кода, которые содержат многострочные комментарии и вам требуется обернуть полностью этот блок кода в комментарий — ничего не получится, если вы воспользуетесь /\*многострочный комментарий\*/. Другое дело — конструкция директив #if #endif.

* **Директива #ifdef**

|  |
| --- |
| #ifdef nameToken  // код, который выполнится, если nameToken определен  #else  // код, который выполнится, если nameToken не определен  #endif |

Директива #ifdef проверяет, был ли ранее определен макрос или символическая константа как #define. Если — да, компилятор включает в программу код, который находится между директивами #ifdef и #else, если nameToken ранее определен не был, то выполняется код между #else и #endif, или, если нет директивы #else, компилятор сразу переходит к #endif.

Например, макрос \_\_cpp определен в C++, но не в Си. Вы можете использовать этот факт для смешивания C и C++ кода, используя директиву #ifdef:

|  |
| --- |
| #ifdef \_\_cpp  // C++ код  #else  // Си код  #endif |

* **Директива #ifndef**

|  |
| --- |
| #ifndef nameToken  // код, который выполнится, если nameToken не определен  #else  // код, который выполнится, если nameToken определен  #endif |

Директива #ifndef проверяет, был ли ранее определен макрос или символическая константа как #define. Если — да, компилятор включает в программу код, который находится между директивами #else и #endif, если nameToken ранее определен не был, то выполняется код между #ifndef и #else, или, если нет директивы #else, компилятор сразу переходит к #endif.

Директива #ifndef может быть использована для подключения заголовочных файлов. если они не подключены, для этого использовать символическую константу, как индикатор подключенного к проекту функционала.

Например, в заголовочном файле есть интерфейс класса, который необходимо подключить к проекту, если ранее этот класс не был подключен.

|  |
| --- |
| #ifndef PRODUCT\_H  #define PRODUCT\_H    class Product  {  // код класса...  };    #endif PRODUCT\_H |

В этом случае используется пустая символьная константа PRODUCT\_H, которая может быть определена в программе только вместе с классом Product. Поэтому, если мы обнаружим, что константа PRODUCT\_H уже определена, значит класс тоже и тогда мы исключим повторное определение класса, которое может привести к ошибке переопределения.

* **Директива #error**

|  |
| --- |
| #error "Этот код не должен компилироваться" |

Директива #error позволяет отображать в списке ошибок компиляции сообщение, в случае возникновения соответствующей ошибки. Эту директиву наиболее полезно использовать в сочетании с директивами #if, #elsif, #else для проверки компиляции, если некоторое условие не верно. Например:

|  |
| --- |
| #ifndef \_\_unix\_\_ // \_\_unix\_\_ обычно поддерживается в юникс-системах  #error "Поддерживается только в Unix"  #endif |

* **Директива #pragma**

Директива pragma указывает компилятору выполнить определенное действие во время компиляции. Директивы pragma зависят от компилятора. Например, чтобы задать оптимизации для выполнения в программе, можно использовать директиву pragma optimize.

**Заголовочный файл (header file)** - это файл с расширением .h или .hpp, который содержит объявления функций, классов, переменных и других элементов, которые могут быть использованы в нескольких файлах исходного кода.

Заголовочные файлы позволяют:

* Разделять код на логические части
* Повторно использовать код в разных файлах
* Скрывать детали реализации
* Улучшать читаемость кода

*//#include "header.h"*

# **Принципы взаимодействия операционной системы с прикладными программами (в том числе раскрыть понятия “процесс”, “поток”, “дескриптор”)**

Операционная система взаимодействует с прикладными программами посредством управления процессами, потоками и дескрипторами.

**Процесс** - это экземпляр выполняющейся программы. Каждый процесс имеет свое собственное адресное пространство, в котором хранится его код, данные, стек вызовов и другие ресурсы. Операционная система управляет процессами, планируя их выполнение на процессоре, выделяя им память, обеспечивая взаимодействие между ними и контролируя их выполнение.

**Поток** - это наименьшая единица выполнения внутри процесса. Один процесс может содержать несколько потоков, которые могут выполняться параллельно или конкурентно. Операционная система управляет потоками, планируя их выполнение на ядрах процессора и обеспечивая синхронизацию и взаимодействие между ними.

**Дескриптор** - это абстрактный указатель на ресурс, такой как файл, сетевое соединение, устройство ввода-вывода и т. д. Операционная система использует дескрипторы для представления и управления доступом к ресурсам.

Прикладные программы могут работать с ресурсами через их дескрипторы, выполняя операции чтения, записи, открытия, закрытия и другие.

Таким образом, операционная система взаимодействует с прикладными программами, обеспечивая им доступ к ресурсам, управление выполнением и обработку ввода-вывода через принципы работы с процессами, потоками и дескрипторами.

# **Принципы функционирования системного программного обеспечения (на основе анализа отличий от других типов ПО)**

Системное программное обеспечение (СПО) отличается от других типов программного обеспечения, таких как прикладное ПО или утилиты, в первую очередь **своими функциями и целями**. Вот несколько основных принципов функционирования системного ПО:

* **Управление ресурсами**: Системное ПО отвечает за управление аппаратными и программными ресурсами компьютерной системы. Это включает в себя управление памятью, процессором, устройствами ввода-вывода, сетевыми соединениями и другими ресурсами.
* **Взаимодействие с аппаратурой**: Системное ПО обеспечивает интерфейс между аппаратурой компьютера и прикладными программами. Оно управляет драйверами устройств, обеспечивает доступ к файловой системе, управляет сетевыми соединениями и т. д.
* **Обеспечение безопасности**: Системное ПО отвечает за обеспечение безопасности компьютерной системы. Это включает в себя контроль доступа к ресурсам, защиту от вредоносных программ, шифрование данных и другие меры безопасности.
* **Управление процессами и потоками**: Как упоминалось ранее, системное ПО управляет процессами и потоками выполнения программ. Оно планирует выполнение процессов на процессоре, выделяет им память, обеспечивает синхронизацию и взаимодействие между потоками.
* **Обеспечение надежности и стабильности:** Системное ПО должно обеспечивать надежную работу компьютерной системы. Это включает в себя обработку ошибок, восстановление после сбоев, мониторинг состояния системы и другие меры для обеспечения стабильной работы.

Таким образом, системное ПО функционирует по принципам управления ресурсами, взаимодействия с аппаратурой, обеспечения безопасности, управления процессами и потоками, а также обеспечения надежности и стабильности компьютерной системы.

# **Назначение реестра Windows. Особенности использования системного реестра программами. Структура системного реестра. Типы данных в системном реестре.**

Формально реестр представляет из **себя базу данных.** Элементами этой базы данных являются **параметры**. Организация параметров представляет из себя иерархию. Чаще всего реестр используется для хранения параметров операционной системы или прикладных программ. Реестр задумывался как некое **общее хранилище настроек**. До этого программы хранили свои настройки в конфигурационных файлах (или файлах инициализации). Использование конфигурационных файлов имела ряд недостатков:

* Не было централизованного хранилища этих файлов.
* Проблема защиты файлов: Текстовые файлы можно было удалитью. Можно было изменить содержимое текстовых файлов по-своему ведому

Для работы с реестром в системе Windws предусмотрена специальная утилита RegEdit.exe Сам реестр имеет 5 глобальных веток:

* **HKEY CLASSES ROOT** — хранится информация о зарегистрированных классах, расширениях документов;
* **HKEY CLASSES USER** — хранится информация о текущей пользовательской конфигурации, внешнем виде рабочего стола, сетевых настройках;
* **HKEY LOCAL MACHINE** — хранится информация о системной и аппаратной конфигурации;
* **HKEY USERS** — хранится информация обо всех зарегистрированных пользователях;
* **HKEY\_CURRENT\_CONFIG** — текущая аппаратная конфигурация.

Остальные параметры имеют более длинные пути, которые начинаются в одной из глобальных веток. Крупные ветки называются ульями. Более мелкие ветки называются ключами реестра. Каждый ключ может содержать внутри себя другие ключи или параметры. Параметры также имеют свои определенные типы:

* **Строковый параметр**. Содержит последовательность символов в определенной кодировке. Его идентификатор в системе REG\_SZ
* **Двоичный параметр**. По сути содержит массив байт. Его идентификатор в системе REG\_DWORD
* **Машинное слово для 32-битных систем**. Его идентификатор в системе RED\_DWORD
* **Машинное слово для 64 разрядной системы**. Его идентификатор в системе RED\_QWORD
* **Мультистроковый параметр**. другими словами массив строк. Его идентификатор в системе REG\_MULTI\_SZ
* **Расширяемый строковый параметр**. По сути представляет из себя строку переменной длины. REG\_EXPAND\_SZ

# **Системный буфер обмена. Особенности взаимодействия программного обеспечения с системным буфером обмена**

Системный буфер обмена — это временное хранилище данных, которое позволяет программам обмениваться информацией между собой. Он используется для копирования, вырезания и вставки текста, изображений и других объектов.

Программы могут взаимодействовать с системным буфером обмена, помещая данные в буфер при копировании или вырезании и извлекая их оттуда при вставке. Это позволяет пользователям легко перемещать информацию между различными приложениями, необходимо лишь скопировать данные в буфер обмена в одной программе и вставить их в другой.

Однако системный буфер обмена имеет свои ограничения. Например, он может содержать только один объект одновременно, поэтому копирование новых данных заменяет старые. Кроме того, формат данных в буфере обмена должен быть совместим с программой, в которую они вставляются.

По сути буфер обмена можно рассматривать **как канал или глобальный указатель**, к которому имеют доступ все прикладные программы. Однако при работе с буфером следует учитывать и его недостатки:

* Буфер обмена **только один в системе** и любое приложение может его перезаписать. Поэтому нет гарантии целостности получения данных
* Информация **не может быть приватной** (Во сколько буфер обмена доступен всем), Буфер обмена на позволяет хранить данные различных типов. Однако при работе с ним программы учитывают этот контекст. В зависимости от формата данных в буфере с ним могут взаимодействовать те или иные приложения.

Причины использования создания локального буфера обмена, а не использования каналов:

* Канал **не может функционировать без сервера** (Если программа сервер завершится, то канал закроется);
* Доступ к каналу нужно **специально организовывать из программного кода**.

Одним из решений данной проблемы является использование системного **буфер обмена**.

Работа с буфером обмена.

Буфер обмена нужно скорее воспринимать как указатель, а не как переменную. У буфера **нет своей заранее выделенной области**. Поэтому память выделяется в самой программе

# **Структуры данных. Актуальность использования структур. Передача структуры как параметра.**

Есть несколько определений структур данных:

Структуры данных — это способы организации и хранения данных в компьютере. Они играют важную роль в программировании, поскольку позволяют эффективно хранить и обрабатывать информацию.

Структура данных в программировании — **это объект-контейнер**, в котором хранится информация. Данные внутри контейнера структурированы по особой системе — она различается в зависимости от вида структуры. Так разработчики **оптимизируют доступ к информации.**

Через структуру можно по определенному алгоритму получить доступ к каждому из значений — или почти к каждому. Какие типы могут быть у этих переменных, как именно они будут структурированы — зависит от конкретной сущности.

Актуальность использования структур данных заключается в их способности удобно представлять сложные данные и связи между ними.

Из-за того, что саму по себе структуру можно определить как переменную (или указатель на неё), её можно передать в качестве параметра функции и использовать в ней, что, соответственно, позволит и вернуть её же.

**Характеристики структуры (если хочется почитать)**

Структуры данных в общем можно разделить на линейные и нелинейные. Разница в том, как они хранят свои элементы — данные, которые в них расположены.

* В линейных структурах элементы расположены **как бы «по цепочке», друг за другом.** Они выстраиваются в последовательность. Например, слово можно представить как линейную структуру данных, где данные — буквы.
* В нелинейных элементы могут **ветвиться, образовывать таблицы или схемы**. Пример нелинейной структуры данных из жизни — схема дорог.

**Массивы**

Это наиболее простой пример структуры данных. Массив — это **линейная последовательность значений**, у каждого из которых есть свой номер. Номера называются индексами — по ним можно получить доступ к любому элементу массива.

Индексы идут последовательно. Это целые числа, последовательность которых начинается с 0.

Можно выделить одномерные и многомерные массивы. Одномерные — это просто ряд пронумерованных значений. Многомерные — матрица: каждое из значений массива само является массивом. Соответственно, доступ к элементу возможен по двум индексам — строки и столбца.

Классические массивы — статические, то есть у них конечная и неизменная длина. Если такой массив создать с длиной 5 — там всегда будет место ровно для 5 элементов, не больше и не меньше. Еще бывают динамические массивы: их длина изменяется в зависимости от того, сколько значений в них «положили».

**Очереди**

Очередь похожа на массив. Это тоже линейная структура, состоящая из последовательности элементов. Разница в том, что доступ к этим элементам **возможен только по принципу FIFO: First In, First Out**. Это значит, что из очереди можно быстро и легко извлечь элемент, который расположен **в самом ее начале и находится в ней дольше всего.** А вот операций для доступа с конца или из середины может вообще не быть. Добавляются же элементы, наоборот, только в конец — как с реальной очередью в магазине.

**Стеки**

Стек — структура, обратная очереди. Это последовательность, в которой доступ **работает по принципу LIFO: Last In, First Out**. Элементы добавляются в конец, а быстро получить и извлечь их можно опять же с конца. То есть **чем позже элемент добавили в стек, тем легче до него добраться.**

Английское слово stack означает «стопка», например, стопка бумаг. Так можно представить и стек: как стопку бумаг, где самый верхний элемент берется первым.

**Деки**

Деками называют двусторонние очереди: они **объединяют возможности и очереди, и стека**. Такие структуры данных могут работать и по принципу FIFO, и по принципу LIFO. Доступ к элементам возможен с любого конца.

**Связанные списки**

Еще один распространенный пример линейной структуры данных. Это последовательность элементов, каждый из которых **хранит данные и ссылку на следующий или предыдущий элемент**. В результате от одного элемента можно быстро получить доступ к его «соседу».

Сами элементы, в отличие от массива, хранятся отдельно друг от друга. У них нет номеров. Последовательность достигается исключительно за счет указателей «следующий» и «предыдущий». Последний элемент будет указывать на «пустое» значение.

У односвязных списков есть ссылки только на следующий элемент, **у двусвязных — и на следующий, и на предыдущий.** Их еще называют однонаправленными или двунаправленными. Есть и круговые списки — они закольцовываются, и последний их элемент указывает на первый.

Первый элемент называется **головой списка**, последний — **хвостом**.

**Множества**

Множество — еще одна линейная структура данных. Ее особенность в том, что элементы не имеют четкого порядка, зато **уникальны по значению**. Ее можно представить не как последовательность или список, а как «облако тегов»: неупорядоченный набор уникальных значений. Это определение близко к математическому понятию множества. И работают с ним так же: не сортируют и не структурируют, но хранят, получают и анализируют данные.

Множество еще называют сетом. По-английски название звучит как Set.

**Карты**

По-английски эта структура данных называется Map. Ее еще называют ассоциативным массивом или словарем. Она действительно похожа на массив, но вместо упорядоченных числовых индексов в ней используются «ключи» — заданные пользователем числа или строки. **Весь массив — это набор пар из ключей и значений**.

Можно представить эту структуру данных как телефонный справочник. Ключ — **это имя**, по которому человек ищет нужный номер. А само значение — собственно, **номер телефона.**

Разработчик сам задает ключи или генерирует их автоматически. Часто ключи строковые: это облегчает понимание и поиск информации.

Обычно в ассоциативных массивах можно хранить данные разных типов. Ключи тоже могут различаться.

**Хэш-таблицы**

Эта структура данных похожа на ассоциативный массив и иногда реализуется через него. Разница в том, что ключ для каждого значения задается автоматически с помощью **специальной формулы — хэш-формулы**. Эта формула применяется к самому значению — в результате генерируется ключ, основанный на нем.

С помощью хэш-таблицы можно генерировать ключи автоматически, например, в ситуациях, когда их название не должно нести полезной нагрузки. Это удобнее и быстрее, чем словарь, если речь идет о больших объемах данных. Кроме того, использование хэшей позволяет шифровать информацию — правда, одной таблицы для этого недостаточно.

При работе с хэш-таблицами важно избегать коллизий — ситуаций, когда применение формулы к разным значениям дает одинаковые ключи. Чтобы такого не было, нужно грамотно подбирать формулу для каждого случая. Также существуют специальные стратегии для предотвращения коллизий.

**Графы**

Граф — **нелинейная структура организации данных**, которая состоит из «вершин» и «ребер» между ними. Каждая вершина — это значение, а ребра — пути, которые соединяют между собой вершины. Получается своеобразная «сетка», похожая на карту дорог или созвездие.

Графы бывают неориентированными, когда ребра не имеют конкретного направления, и ориентированными. Во втором случае по ребрам можно пройти только в одну сторону — как по дороге с односторонним движением. Есть смешанные графы, в которых есть и ориентированные, и неориентированные ребра.

Также существуют взвешенные графы, у ребер которых есть «вес» — то или иное значение. Например, в карте дорог весом ребра-дороги можно назвать его длину.

Графы обычно реализуют с помощью связанных списков или матриц — двумерных массивов.

**Деревья**

Деревья можно назвать **частным случаем графов**. Это тоже структуры из вершин и ребер, но имеющие древовидный формат. Вершины деревьев называются узлами. От одного узла может отходить несколько вершин-потомков, но предок у каждого узла может быть только один. Так и получается древовидная иерархическая структура.

В программировании применяют несколько видов деревьев. Большинство из них реализуют с помощью графов.

1. **Бинарные деревья поиска** — самый распространенный формат деревьев. У каждого узла может быть не более двух потомков. Если значение узла-потомка меньше предка, он располагается слева. Если равно или больше — справа. С помощью таких деревьев удобно сортировать данные и искать в них нужное значение с помощью бинарного поиска.
2. **Префиксные деревья**, они же нагруженные деревья или боры — это деревья, которые хранят данные «по цепочке». Узлы-предки — префиксы, которые нужны, чтобы перейти к потомкам. При прохождении дерева «собираются» полные данные. Например, в каждом узле — буква. При прохождении от начала до конца получаются слова. Обычно такие деревья используют для хранения повторяющихся данных или, например, для реализации автодополнения при вводе.
3. **Двоичные кучи** — двоичные деревья, заполненные целиком. У каждого узла два потомка. Потомки в зависимости от типа дерева должны быть строго больше предков или меньше либо равны им.

Также существуют N-арные деревья, красно-черные деревья, AVL-деревья, сбалансированные, 2-3-4-деревья и так далее.

# **Типы данных в ОС Windows. Назначение и использование системных диалоговых окон.**

Я НЕ ПОНИМАЮ КАК ЭТО СОКРАТИТЬ!

В операционной системе Windows используются различные типы данных, многие из которых определены в заголовочных файлах Windows API. Они обеспечивают взаимодействие между приложениями и самой системой. Некоторые ключевые типы данных:

**Основные типы**: int, long, short, char, float, double, bool — стандартные типы C/C++, не так часто используемые в Windows API.

**Указатели**:

* HANDLE, LPVOID, HWND, HDC, HINSTANCE и другие — указатели на различные ресурсы системы (файлы, окна, устройства вывода и т.д.).
* HANDLE — это общий тип указателя на ресурс, конкретный тип зависит от контекста.
* LPVOID — указатель на нетипизированную область памяти.
* HWND — указатель на окно.
* HDC — указатель на контекст устройства (для отрисовки). HINSTANCE — дескриптор экземпляра приложения.

**Строки**:

TCHAR, LPTSTR, LPWSTR — типы для работы со строками.

TCHAR — зависит от UNICODE (широкие символы) или \_MBCS (узкие символы).

LPTSTR — указатель на строку TCHAR.

LPWSTR — указатель на строку широких символов (Unicode).

**Структуры**: RECT, POINT, SIZE, MSG, WNDCLASS, STARTUPINFO и множество других — структуры данных, определённые Windows API для представления различной информации: геометрия окон, координаты, системные сообщения, классы окон и т. д.

**Дескрипторы**: HANDLE — уникальные идентификаторы системных ресурсов (файлов, процессов, событий, взаимодействий и т. д.). Эти дескрипторы используются для управления этими ресурсами.

**Специальные типы**: BOOL, DWORD, UINT, LPARAM, WPARAM — типы данных, часто используемые в функциях Windows API, имеют особое назначение в контексте системы.

**Назначение и использование системных диалоговых окон:**

Системные диалоговые окна — это **предопределённые окна**, предоставляемые операционной системой Windows, которые упрощают взаимодействие пользователя с приложением. Они предлагают стандартные элементы управления (кнопки, поля ввода, списки и т. д.) для выполнения общих задач. Использование системных диалоговых окон вместо создания собственных имеет ряд преимуществ:

**Стандартный интерфейс**: пользователи знакомы с внешним видом и поведением стандартных диалоговых окон, что повышает удобство использования приложения.

**Простота использования**: API функций для отображения системных диалогов просты и понятны. Это сокращает время разработки.

**Локализация**: системные диалоговые окна автоматически адаптируются к региональным настройкам операционной системы, обеспечивая поддержку различных языков.

**Доступность**: системные диалоговые окна часто соответствуют стандартам доступности, что делает приложение более удобным для пользователей с ограниченными возможностями.

Примеры системных диалоговых окон:

MessageBox: Для отображения простых сообщений.

OpenFileDialog: Для выбора файлов.

SaveFileDialog: Для сохранения файлов.

ColorDialog: Для выбора цвета.

FontDialog: Для выбора шрифта.

FindReplaceDialog: Для поиска и замены текста.

# **Указатель как тип данных. Особенности работы с указателями**

Указатель — это переменная, хранящая адрес в памяти, **по которому находится другое значение**. Он является фундаментальным понятием в программировании, особенно в языках C и C++. Понимание указателей необходимо для работы с динамической памятью, структурами данных и низкоуровневым программированием.

**Особенности работы с указателями:**

* Объявление: указатели объявляются с **помощью символа \*** перед именем переменной. Тип указателя должен соответствовать типу данных, на которые он указывает.

|  |
| --- |
| int\* ptrToInt; // Указатель на целое число  char\* ptrToChar; // Указатель на символ  double\* ptrToDouble; // Указатель на число с плавающей точкой  Point\* ptrToPoint; // Указатель на структуру Point (предполагается, что структура Point определена) |

* **Инициализация**: указатели должны быть инициализированы перед использованием. Это может быть нулевое значение (nullptr в C++, NULL в C), адрес переменной или адрес памяти, выделенной динамически.

|  |
| --- |
| int x = 10;  int\* ptr = &x; // ptr теперь указывает на x  int\* ptr2 = nullptr; // ptr2 не указывает ни на что  int\* ptr3 = (int\*)malloc(sizeof(int)); // Выделение памяти, ptr3 указывает на выделенную область  \*ptr3 = 20; // Запись значения в выделенную область  free(ptr3); // Освобождение выделенной памяти |

* **Операция разыменования: оператор \*** (звездочка) используется для доступа к значению, на которое указывает указатель. Это называется разыменованием указателя.

|  |
| --- |
| int value = \*ptr; // value будет равно 10  \*ptr = 20; // значение по адресу ptr (x) изменится на 20 Оператор адреса: оператор & (амперсанд) возвращает адрес переменной.  int x = 10;  int\* ptr = &x; // ptr хранит адрес x |

* Арифметика указателей: можно выполнять арифметические операции над указателями (сложение, вычитание). Важно помнить, что приращение указателя на 1 **перемещает его на размер типа данных**, на который он указывает.

|  |
| --- |
| int arr[5] = {1, 2, 3, 4, 5};  int\* ptr = arr; // ptr указывает на начало массива  int\* next = ptr + 1; // next указывает на второй элемент массива |

* Указатели на функции: **указатели могут хранить адреса функций**. Это позволяет передавать функции в качестве аргументов другим функциям или вызывать функции динамически.

|  |
| --- |
| int add(int a, int b) { return a + b; }  int (\*funcPtr)(int, int) = add; // Указатель на функцию add  int result = funcPtr(5, 3); // Вызов функции через указатель |

* Висячие указатели: это указатели, которые указывают на память, которая уже была освобождена. Доступ к памяти через висячий указатель приводит к неопределённому поведению программы (сегментация или другие ошибки).

# **Массив как тип данных. Особенности использования массивов при разработке программ на си.**

Массив в языке C — это непрерывный блок памяти, предназначенный для хранения элементов одного и того же типа данных. Элементы массива доступны по индексу (начиная с 0).

**Особенности использования массивов в C:**

* **Объявление**: Массив объявляется указанием типа данных элементов, имени массива и размера (количества элементов) в квадратных скобках.

|  |
| --- |
| int numbers[10]; // Массив из 10 целых чисел  float prices[5]; // Массив из 5 чисел с плавающей точкой  char name[20]; // Массив из 20 символов (строка) |

* **Инициализация**: массив может быть инициализирован при объявлении:

|  |
| --- |
| int numbers[5] = {1, 2, 3, 4, 5};  char name[] = "Hello"; // Размер определяется автоматически |

Если при инициализации указано меньше элементов, чем размер массива, оставшиеся элементы инициализируются нулями.

* **Доступ к элементам**: элементы массива доступны по индексу, указанному в квадратных скобках:

|  |
| --- |
| numbers[0] = 10; // Изменение первого элемента  int x = numbers[2]; // x будет равно 3 |

Индексы начинаются с 0 и заканчиваются размер - 1. Обращение к элементу за пределами массива (индекс вне допустимого диапазона) приводит к неопределённому поведению и обычно к сбою программы, а если быть точнее либо к ошибке, либо к ошибке сегментации (зависит от типа обращения).

* **Размер массива**: размер массива фиксируется при компиляции. Нельзя изменить размер массива во время выполнения программы.
* **Передача массивов в функции**: при передаче массива в функцию фактически передаётся указатель на его первый элемент.

|  |
| --- |
| void printArray(int arr[], int size) { // arr - это указатель на int  for (int i = 0; i < size; i++) {  printf("%d ", arr[i]);  }  printf("\n");  }  int main() {  int numbers[5] = {1, 2, 3, 4, 5};  printArray(numbers, 5); // Передача массива в функцию  return 0;  } |

Внутри функции printArray arr — это указатель, поэтому можно изменять элементы исходного массива внутри функции. Необходимо передавать размер массива в функцию, так как внутри функции размер массива неизвестен.

* **Многомерные массивы**: в C можно создавать многомерные массивы (массивы массивов):

|  |
| --- |
| int matrix[3][4]; // Матрица 3x4  matrix[1][2] = 10; // Присваивание значения элементу |

* **Строки как массивы**: строки в C представляются как массивы символов, заканчивающиеся нулевым символом (\0).

|  |
| --- |
| char message[] = "Hello, world!"; |

* **Динамическое выделение памяти**: для создания массивов переменного размера можно использовать функции malloc и calloc:

|  |
| --- |
| int size;  scanf("%d", &size);  int\* dynamicArray = (int\*)malloc(size \* sizeof(int)); // Выделение памяти для массива  // ... использование dynamicArray ...  free(dynamicArray); // Освобождение памяти |

Важно освобождать память, выделенную с помощью malloc, после использования, чтобы избежать утечек памяти. calloc инициализирует выделенную память нулями.

# **Линейный односвязный список. Особенности создания и примеры использования.**

Линейный односвязный список — это **динамическая структура данных**, представляющая собой последовательность элементов (узлов), где каждый узел содержит **данные и указатель на следующий узел в списке**. Последний узел указывает на NULL (или nullptr в C++). В отличие от массивов, размер списка не фиксирован и может изменяться во время выполнения программы.

**Особенности создания:**

Структура узла: обычно узел представляется структурой, содержащей данные и указатель на следующий узел:

|  |
| --- |
| struct Node {  int data; // Данные  Node\* next; // Указатель на следующий узел  }; |

**Создание узла: для создания списка необходимо определить структуру узла, содержащую данные и указатель на следующий узел. Затем можно создать функции для добавления, удаления и обращения к элементам списка.**

|  |
| --- |
| Node\* createNode(int data) {  Node\* newNode = new Node;  newNode->data = data;  newNode->next = nullptr;  return newNode;  } |

**Добавление узла: новый узел может быть добавлен в список, путем создания нового узла и изменения указателей на предыдущий и следующий узлы.**

|  |
| --- |
| void addNodeToBeginning(Node\*& head, int data) { // head передается по ссылке  Node\* newNode = createNode(data);  newNode->next = head;  head = newNode;  }  Добавление в конец требует поиска последнего узла:  void addNodeToEnd(Node\*& head, int data) {  Node\* newNode = createNode(data);  if (head == nullptr) {  head = newNode;  return;  }  Node\* current = head;  while (current->next != nullptr) {  current = current->next;  }  current->next = newNode;  } |

**Удаление узла: узел можно удалить из списка, путем изменения указателей на предыдущий и следующий узлы, чтобы исключить удаляемый узел из списка.**

|  |
| --- |
| void deleteNode(Node\*& head, int data) {  if (head == nullptr) return;  if (head->data == data) {  Node\* temp = head;  head = head->next;  delete temp;  return;  }  Node\* current = head;  while (current->next != nullptr && current->next->data != data) {  current = current->next;  }  if (current->next != nullptr) {  Node\* temp = current->next;  current->next = current->next->next;  delete temp;  }  } |

**Поиск узла:  элементы списка могут быть доступны по их индексу, начиная с начала списка и переходя по указателям на следующие узлы.**

|  |
| --- |
| bool searchNode(Node\* head, int data) {  Node\* current = head;  while (current != nullptr) {  if (current->data == data) return true;  current = current->next;  }  return false;  } |

**Примеры использования:**

Хранение данных в порядке вставки: список можно использовать для хранения данных в том порядке, в котором они были добавлены.

Реализация стека или очереди: односвязный список может быть использован для реализации стека (LIFO) или очереди (FIFO).

Реализация других структур данных: односвязные списки могут быть использованы как основа для более сложных структур данных, таких как деревья.

Ситуации с частым добавлением/удалением элементов в начало списка: в таких случаях односвязный список более эффективен, чем массив.

# **Указатель на функцию. Назначение и примеры использования.**

Указатель на функцию — это переменная, которая **хранит адрес функции**. Это позволяет:

* **Передача функций в качестве аргументов другим функциям**: это позволяет создавать более гибкие и модульные функции, которые могут работать с разными функциями в зависимости от ситуации.
* **Динамический вызов функций**: позволяет выбирать, какую функцию вызывать во время выполнения программы в зависимости от каких-либо условий.
* **Создавать таблицы функций**: указатели на функции могут использоваться для создания массивов или других структур данных, содержащих адреса функций. Это полезно для создания систем обработки событий или для реализации полиморфизма.

**Объявление:**

Указатель на функцию объявляется так же, как и обычный указатель, но с **указанием типа возвращаемого значения и типов параметров функции**.

|  |
| --- |
| // Объявление указателя на функцию, которая принимает два целых числа и возвращает целое число  int (\*funcPtr)(int, int);  // Объявление указателя на функцию, которая не принимает аргументы и не возвращает значения  void (\*voidFuncPtr)(); |

**Инициализация:**

Указатель на функцию инициализируется адресом функции:

|  |
| --- |
| int add(int a, int b) { return a + b; }  int (\*funcPtr)(int, int) = add; // funcPtr теперь указывает на функцию add  void printHello() { std::cout << "Hello!" << std::endl; }  void (\*voidFuncPtr)() = printHello; // voidFuncPtr указывает на printHello |

**Вызов функции через указатель:**

Функция, на которую указывает указатель, вызывается так же, как и обычная функция, но через указатель:

|  |
| --- |
| int result = funcPtr(5, 3); // Вызов функции add через указатель funcPtr  voidFuncPtr(); // Вызов функции printHello через указатель voidFuncPtr |

**Пример использования:**

|  |
| --- |
| #include <iostream>  #include <vector>  #include <functional> // для std::function  // Функция, которая принимает указатель на функцию и два аргумента  int applyFunction(int a, int b, std::function<int(int,int)> func) {  return func(a, b);  }  int add(int a, int b) { return a + b; }  int subtract(int a, int b) { return a - b; }  int main() {  std::vector<std::function<int(int,int)>> functions = {add, subtract};  for (const auto& func : functions) {  std::cout << "Результат: " << applyFunction(10, 5, func) << std::endl;  }  //Использование указателей на функции без std::function (менее предпочтительно в C++)  int (\*funcPtr)(int, int) = add;  int result = applyFunction(10, 5, funcPtr); // Неявное преобразование к std::function  std::cout << "Результат с указателем: " << result << std::endl;  return 0;  } |

В этом примере функция applyFunction принимает указатель на функцию в качестве аргумента и вызывает эту функцию. std::function используется для более безопасной и гибкой работы с указателями на функции.

# **Файл как тип данных. Использование файлов в работе программного обеспечения.**

Файл - именованная область данных (байт), располагающаяся в каком-либо носителе информации. В большинстве своём, мы работаем с указателем, файлом можно называть указатель на адрес, где хранится все содержимое файла

Указатель на структуру файлового типа - FILE, HANDLE - указатель на источник дальнейшего извлечения данных.

Каталог - массив указателей на файлы, который в ней содержится.

Файлы бываются:

* **Текстовые. Это набор символов в некоторой кодировке.**
* **Бинарные. Последовательность двоичных данных, имеющие определенную структуру, которую нужно интерпретировать.**

Как используют файлы:

* **Хранение данных (конфигурация программы, хранение кода или текста).**
* **Обмен данными.**
* **Управление данными. (отслеживание изменений в коде и т.д.)**
* **Представление данных (html, md, docx)**
* **Настройка параметров**: конфигурационные файлы хранят настройки программ. Это позволяет пользователям настраивать поведение программы без изменения исходного кода.
* **Архивирование**: файлы архивируются для экономии места и защиты данных от повреждения.
* **Кэширование**: программы могут использовать файлы для кэширования данных, чтобы ускорить доступ к часто используемой информации (файлы подкачки).

Программное обеспечение может работать с файлами, открывая их для чтения или записи, изменяя их содержимое, перемещая, копируя или удаляя их. Для работы с файлами в различных языках программирования существуют специальные функции и библиотеки, которые облегчают взаимодействие с файловой системой операционной системы.

# **Функции для работы с файлами WinAPI. Синхронный и асинхронный доступ к файлу.**

В для работы с файлами используются различные функции, позволяющие открывать, читать, записывать, перемещать и удалять файлы. Для синхронного доступа к файлам используются функции, которые блокируют выполнение программы до завершения операции ввода-вывода, в то время как для асинхронного доступа используются функции, которые позволяют продолжать выполнение программы без блокировки.

Для работы с файлами, необходимо работать с их дескриптором (как в Windows, так и в Linux), а все функции работы делятся на 2 категории:

* **Работа с ASCII – A**
* **Работа с Unicode, UTF-16 – L**

От данных категорий зависит написание пути к файлу и тип вызываемой функции.

**Основные функции:**

CreateFile(): Создает или открывает файл, возвращая дескриптор файла (HANDLE). Эта функция очень гибкая и позволяет настраивать различные параметры доступа к файлу.

ReadFile(): Читает данные из файла, используя дескриптор файла.

WriteFile(): Записывает данные в файл, используя дескриптор файла.

CloseHandle(): Закрывает дескриптор файла. Важно: всегда закрывайте дескрипторы файлов после завершения работы с ними, чтобы освободить ресурсы.

GetFileSize(): Получает размер файла.

SetFilePointer(): Устанавливает указатель текущей позиции в файле.

SetEndOfFile(): Устанавливает конец файла на текущую позицию.

DeleteFile(): Удаляет файл.

Мы можем делать следующие вещи **синхронно/асинхронно:**

* **Чтение;**
* **Запись;**
* **Создание;**

Для синхронности/асинхронности, указывается специальный параметр. В WriteFile и ReadFile и подобным это lpOverlapped, в CreateFile это dwFlagsAndAttributes. Синхронно - NULL, асинхронно - FILE\_FLAG\_OVERLAPPED.

Когда мы указываем делать это **синхронно**, она не возвращает контекст выполнение программы до своего завершения. То есть во время вызова функции синхронно, программа останавливается до тех пор\, пока не выполнится операция.

Когда мы вызываем **асинхронно** программа продолжает свою работу, выполняя операцию в фоновом режиме, и, соответственно, чтобы работать с ней дальше можно подождать или вызвать специальную функцию, чтобы её дождаться.

**Асинхронный** режим позволяет, экономить время, если процесс будет весьма трудоемкий, то есть пока операция выполняется мы выполняем другие действия.

**Синхронный доступ к файлу (пример)**

Синхронный доступ — это то, что обычно используется в простых программах. Функции ReadFile() и WriteFile() **блокируют выполнение программы** до завершения операции чтения или записи.

|  |
| --- |
| Пример синхронного доступа (C++):  #include <windows.h>  #include <iostream>  int main() {  HANDLE hFile = CreateFile(L"myFile.txt", GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS, FILE\_ATTRIBUTE\_NORMAL, NULL);  if (hFile == INVALID\_HANDLE\_VALUE) {  std::cerr << "Ошибка создания файла: " << GetLastError() << std::endl;  return 1;  }  DWORD bytesWritten;  const char\* data = "Hello, world!\n";  BOOL success = WriteFile(hFile, data, strlen(data), &bytesWritten, NULL);  if (!success) {  std::cerr << "Ошибка записи в файл: " << GetLastError() << std::endl;  CloseHandle(hFile);  return 1;  }  CloseHandle(hFile);  std::cout << "Данные успешно записаны в файл." << std::endl;  return 0;  } |

**Асинхронный доступ к файлу (пример)**

Асинхронный доступ позволяет выполнять другие операции, **пока происходит чтение или запись в файл.** Для асинхронного доступа используются функции ReadFileEx() и WriteFileEx(), а также механизмы событий или завершения (порты завершения). Асинхронный доступ сложнее в реализации, но позволяет создавать более отзывчивые программы

|  |
| --- |
| #include <windows.h>  #include <iostream>  // ... (другие функции) ...  DWORD WINAPI AsyncReadFileThread(LPVOID lpParameter) {  OVERLAPPED overlapped = {0};  HANDLE hFile = (HANDLE)lpParameter; //Дескриптор файла передается в поток  char buffer[1024];  DWORD bytesRead;  overlapped.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL); //Создаем событие  BOOL success = ReadFileEx(hFile, buffer, sizeof(buffer), &overlapped, AsyncReadCompletionRoutine);  if (!success && GetLastError() != ERROR\_IO\_PENDING) {  CloseHandle(overlapped.hEvent);  std::cerr << "Ошибка: " << GetLastError() << std::endl;  return 1;  }  WaitForSingleObject(overlapped.hEvent, INFINITE); // Ждем завершения асинхронной операции  CloseHandle(overlapped.hEvent);  return 0;  }  // Функция обратного вызова для завершения асинхронной операции чтения  VOID CALLBACK AsyncReadCompletionRoutine(DWORD dwErrorCode, DWORD dwNumberOfBytesTransfered, LPOVERLAPPED lpOverlapped) {  // Обработка результата асинхронного чтения  }  int main(){  // ... (открытие файла) ...  HANDLE hFile = ...;  CreateThread(NULL, 0, AsyncReadFileThread, hFile, 0, NULL); //Запускаем поток  // ... (другие операции, выполняемые параллельно с чтением) ...  return 0;  } |

В этом фрагменте используется ReadFileEx() со OVERLAPPED структурой и функцией обратного вызова (AsyncReadCompletionRoutine) для обработки результатов асинхронного чтения после завершения операции. Обратите внимание, что асинхронный подход требует более сложного кода для управления потоками и обработки событий.

# **Функции для работы с файлами в UNIX-подобных операционных системах.**

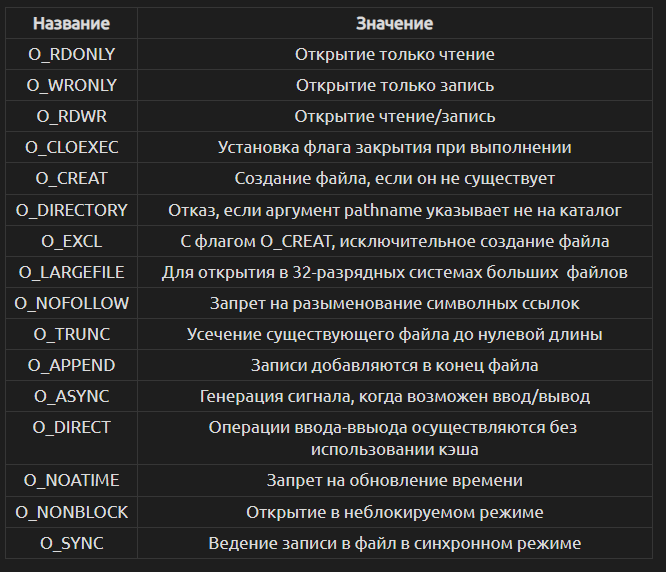
В UNIX-подобных операционных системах (Linux, macOS, BSD и др.) для работы с файлами используются функции из библиотеки unistd.h и семейства функций open(), read(), write(), close(), а также другие функции из sys/stat.h, fcntl.h, dirent.h и других заголовочных файлов. Этот подход более низкоуровневый, чем stdio.h в C, поэтому требует более внимательного отношения к деталям.

**Основные функции:**

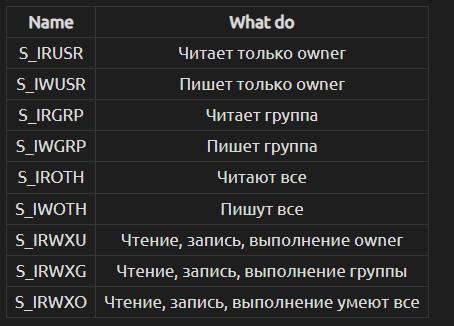
* open(): Открывает файл. Возвращает файловый дескриптор (целое число), который используется для последующих операций с файлом. Эта функция позволяет указать режим доступа (чтение, запись, создание) и другие флаги.

|  |
| --- |
| int fd = open("myfile.txt", O\_RDONLY); // Открытие для чтения  int fd2 = open("myfile2.txt", O\_WRONLY | O\_CREAT, 0644); // Открытие для записи, создание если не существует, права доступа 0644 |

Для взаимодействия с файлом после его открытия, необходимо указать в функции open соответствующие флаги (константы типа int (O\_” - флаг.)).



Так же предоставляются права доступа – моды (ип данных “mode\_t”, предназначен для создания и открытия файлов, практически все что начинается с “S\_” – mod).

****

Флаги и моды можно комбинировать с помощью оператора «|».

* read(): Читает данные из файла.

|  |
| --- |
| ssize\_t bytesRead = read(fd, buffer, 1024); // Чтение 1024 байтов в буфер |

* write(): Записывает данные в файл.

|  |
| --- |
| ssize\_t bytesWritten = write(fd2, data, strlen(data)); // Запись данных в файл |

* close(): Закрывает файл, освобождая связанные с ним ресурсы. Важно: всегда закрывайте файловые дескрипторы после завершения работы с ними.

|  |
| --- |
| close(fd);  close(fd2); |

* lseek(): Изменяет текущую позицию в файле.
* stat(): Получает информацию о файле (размер, время последнего доступа и т. д.).
* unlink() (или remove()): Удаляет файл.
* fstat(): Получает информацию о файле, используя файловый дескриптор.
* dup() и dup2(): Дублируют файловые дескрипторы.

**Работа с каталогами:**

opendir(): Открывает каталог.

readdir(): Читает запись из каталога.

closedir(): Закрывает каталог.

mkdir(): Создает каталог.

rmdir(): Удаляет каталог.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <fcntl.h>  #include <string.h>  #include <errno.h> // Для работы с кодами ошибок  int main() {  int fd = open("myfile.txt", O\_WRONLY | O\_CREAT | O\_TRUNC, 0644); // Открытие для записи, создание, обрезание, права 0644  if (fd == -1) {  perror("Ошибка открытия файла"); // Вывод сообщения об ошибке  return 1;  }  const char\* message = "Hello, world!\n";  ssize\_t bytesWritten = write(fd, message, strlen(message));  if (bytesWritten == -1) {  perror("Ошибка записи в файл");  close(fd);  return 1;  }  if (close(fd) == -1) {  perror("Ошибка закрытия файла");  return 1;  }  printf("Данные успешно записаны в файл.\n");  return 0;  } |

Кроме того, в UNIX-подобных операционных системах используются специальные системные вызовы для работы с процессами, такие как fork и exec, которые могут использоваться для создания новых процессов и выполнения других программ, включая работу с файлами.

При работе с файлами в UNIX-подобных операционных системах также важно учитывать особенности работы с правами доступа к файлам, обработку ошибок и управление многозадачностью.

# **Анонимные каналы (в Windows и Linux). Назначение и использование.**

Канал - Область виртуального пространства, которое может быть использовано для **совместного доступа различными процессами**.

Канал — это всего лишь буфер, находящийся в памяти ядра. И его максимальная вместимость ограничена. Если канал заполнен, то все дальнейшие операции записи в него блокируются, пока считывающий процесс не удалит из него часть данных.

Однако в силу этого канал не может храниться отдельно от какого-то процесса. Поэтому информация о нём будет очищена если завершить процесс, который его создал. Будем называть процесс, который создаёт канал с сервером, а процессы, которые подключаются к каналу - клиентами.

Каналы имеют несколько разновидностей:

* Симплексные или дуплексные. Симплексный это **однонаправленные**. Например, сервер только записывает данные, а Клиенты только читают их. Или клиенты только пишут, а сервер только читает. Дуплексные это когда и клиент и сервер может и читать, и писать
* Бинарные или текстовые. По аналогии с бинарными или текстовыми файлами
* С общим или разделяемым доступом к содержимому
* Анонимные или именованные (также называются очередями FIF) Анонимные каналы. В качестве их идентификатора используется дескрипторы на чтение или на запись.

В Windows Анонимные каналы — это механизм меж процессного взаимодействия (MPI), **позволяющий двум или более процессам обмениваться данными без необходимости устанавливать прямое соединение между ними**. Вместо этого они используют промежуточное хранилище (канал), в которое один процесс записывает данные, а другой их считывает.

Ключевое отличие от именованных каналов заключается в том, что анонимные каналы создаются динамически и не имеют постоянных имён, а общаются с помощью дискрипторов. Они существуют только во время работы процессов, которые их используют.

В Windows нет прямого функционала для анонимных каналов, просто в другой процесс передаются дескрипторы записи и чтения и через методы Writefile, ReadFile происходит обмен.

В Linux Анонимный канал — это неименованный односторонний канал, который обычно передает данные между родительским процессом и дочерним процессом. Анонимные каналы всегда являются локальными; они не могут использоваться для обмена данными по сети.

В UNIX-подобных системах анонимные каналы (pipes) создаются с помощью pipe(). Это создаёт два файловых дескриптора: один для записи, а другой для чтения. Они связаны между собой, и данные, записанные в один дескриптор, доступны для чтения в другом.

Особенности работы с каналами:

* Канал представляет собой поток байтов, при его использовании не существует понятия сообщений или их границ.
* Данные проходят через канал последовательно — байты считываются из него в том же порядке, в котором они были записаны.
* Произвольный доступ к содержимому канала с помощью вызова lseek() невозможен.
* Реализацию принципа отдельных сообщений в канале следует делать на уровне приложения.
* Любая попытка прочитать данные из пустого канала блокируется до тех пор, пока кто-нибудь не запишет в этот канал хотя бы один байт.
* Если закрыть другой конец канала, процесс, выполняющий чтение, получит конец файла (то есть операция read() вернет 0), как только дочитает оставшиеся данные.
* Анонимный канал может быть только симплексным (только либо на чтение, либо на запись).

# **Именованные каналы (в Windows и Linux). Назначение и использование.**

Канал - Область виртуального пространства, которое может быть использовано для **совместного доступа различными процессами**.

Канал — это всего лишь буфер, находящийся в памяти ядра. И его максимальная вместимость ограничена. Если канал заполнен, то все дальнейшие операции записи в него блокируются, пока считывающий процесс не удалит из него часть данных.

Однако в силу этого канал не может храниться отдельно от какого-то процесса. Поэтому информация о нём будет очищена если завершить процесс, который его создал. Будем называть процесс, который создаёт канал с сервером, а процессы, которые подключаются к каналу - клиентами.

Каналы имеют несколько разновидностей:

* Симплексные или дуплексные. Симплексный это **однонаправленные**. Например, сервер только записывает данные, а Клиенты только читают их. Или клиенты только пишут, а сервер только читает. Дуплексные это когда и клиент и сервер может и читать, и писать
* Бинарные или текстовые. По аналогии с бинарными или текстовыми файлами
* С общим или разделяемым доступом к содержимому
* Анонимные или именованные (также называются очередями FIF) Анонимные каналы. В качестве их идентификатора используется дескрипторы на чтение или на запись.

Именованные каналы (named pipes) — это **механизм межпроцессного взаимодействия (MPI),** который позволяет процессам обмениваться данными, используя имя канала в качестве механизма адресации (Либо так же можно использовать дескриптор, тк он всё равно будет существовать). В отличие от анонимных каналов, именованные каналы имеют постоянное имя, что позволяет процессам подключаться к ним независимо от времени их создания.

**Windows**:

В Windows именованные каналы реализованы с помощью функций WinAPI. Они используются для синхронного или асинхронного обмена данными между процессами.

Основные функции:

CreateNamedPipe(): Создает именованный канал. Указывает имя канала, режимы доступа, тип канала и другие параметры.

ConnectNamedPipe(): Серверный процесс использует эту функцию для ожидания подключения клиентского процесса.

WaitNamedPipe(): Клиентский процесс может использовать эту функцию для ожидания создания именованного канала сервером.

CreateFile(): Клиентский процесс использует эту функцию для подключения к уже существующему именованному каналу.

ReadFile() и WriteFile(): Чтение и запись данных через канал.

CloseHandle(): Закрывает дескриптор канала.

**Linux**:

В Linux именованные каналы реализованы с помощью файловой системы. Они представляются как специальные файлы в /tmp или других каталогах.

**Основные функции:**

mkfifo(): Создает именованный канал.

open(): Открывает именованный канал для чтения или записи.

read() и write(): Чтение и запись данных через канал.

close(): Закрывает файловый дескриптор канала.

Особенности работы с каналами:

* Канал представляет собой поток байтов, при его использовании не существует понятия сообщений или их границ.
* Данные проходят через канал последовательно — байты считываются из него в том же порядке, в котором они были записаны.
* Произвольный доступ к содержимому канала с помощью вызова lseek() невозможен.
* Реализацию принципа отдельных сообщений в канале следует делать на уровне приложения.
* Любая попытка прочитать данные из пустого канала блокируется до тех пор, пока кто-нибудь не запишет в этот канал хотя бы один байт.
* Если закрыть другой конец канала, процесс, выполняющий чтение, получит конец файла (то есть операция read() вернет 0), как только дочитает оставшиеся данные.

# **Создание процессов в UNIX-подобных операционных системах.**

Создание процессов в UNIX-подобных операционных системах (Linux, macOS, BSD и др.) осуществляется в основном с **помощью системного вызова fork().** Этот вызов создаёт **копию текущего процесса**, называемую дочерним процессом, котороый содержит в себе копию того, что до создания принадлежало родительскому процессу (стек, куча, текст). После вызова fork() родительский и дочерний процессы продолжают **выполнение с того места, где был вызван fork(),** но с разными значениями возвращаемого значения.

**fork():**

Функция fork() возвращает:

0 в дочернем процессе.

ID дочернего процесса (положительное число) в родительском процессе.

-1 в случае ошибки.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <sys/wait.h>  int main() {  pid\_t pid = fork();  if (pid < 0) {  perror("fork failed");  return 1;  } else if (pid == 0) {  // Код дочернего процесса  printf("Я дочерний процесс, мой PID: %d\n", getpid());  } else {  // Код родительского процесса  printf("Я родительский процесс, мой PID: %d, PID дочернего процесса: %d\n", getpid(), pid);  int status;  wait(&status); // Ожидание завершения дочернего процесса  printf("Дочерний процесс завершен.\n");  }  return 0;  } |

Чтобы запустить **совершенно новый процесс** (с другой программой), используется семейство функций exec():

execl(), execlp(), execle(), execv(), execvp(), execve() — эти функции заменяют текущий процесс новым процессом. Они принимают имя исполняемого файла и аргументы командной строки.

Текст существующей программы сбрасывается, а для новой программы заново создаются сегменты со стеком, данными и кучей. Эту операцию часто называют выполнением новой программы.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <sys/wait.h>  int main() {  pid\_t pid = fork();  if (pid < 0) {  perror("fork failed");  return 1;  } else if (pid == 0) {  // Запуск нового процесса с помощью execl  execl("/bin/ls", "ls", "-l", NULL);  perror("execl failed"); // Эта строка выполнится только в случае ошибки execl  return 1;  } else {  // Код родительского процесса  int status;  wait(&status); // Ожидание завершения дочернего процесса  printf("Дочерний процесс завершен.\n");  }  return 0;  } |

В этом примере дочерний процесс заменяется процессом /bin/ls (команда ls -l). Если execl() выполняется успешно, код дочернего процесса не продолжается.

**wait() и waitpid():**

Функции wait() и waitpid() используются родительским процессом для ожидания завершения дочернего процесса и получения информации о его завершении (код возврата).

**Важные замечания:**

* **Копирование памяти**: fork() создает копию памяти родительского процесса. Изменения, внесенные в одну из копий, не затрагивают другую (за исключением некоторых областей памяти, которые являются общими).
* **Обработка ошибок**: всегда проверяйте возвращаемые значения fork() и exec() на наличие ошибок.
* **Управление процессами**: используйте wait() или waitpid() для управления завершением дочерних процессов. Если не дождаться завершения дочернего процесса, это может привести к появлению «зомби-процессов».
* **Сигналы**: процессы могут взаимодействовать друг с другом с помощью сигналов.

fork() и exec() — фундаментальные системные вызовы в UNIX-подобных системах для управления процессами. Понимание их работы необходимо для создания многопроцессных приложений.

# **Создание процессов в ОС семейства Windows**

В операционных системах семейства Windows создание процессов **осуществляется с помощью функции** CreateProcess(). Эта функция более сложная, чем fork() в UNIX-подобных системах, поскольку она позволяет более тонко настраивать параметры создаваемого процесса.

**CreateProcess():**

Функция CreateProcess() принимает множество параметров, включая:

* lpApplicationName: Путь к исполняемому файлу. Может быть NULL, если имя указано в lpCommandLine.
* lpCommandLine: Командная строка, которая будет передана новому процессу. Может включать имя исполняемого файла и его аргументы.
* lpProcessAttributes и lpThreadAttributes: Дескрипторы безопасности для процесса и потока соответственно. Могут быть NULL.
* bInheritHandles: указывает, наследуются ли дескрипторы родительского процесса дочерним процессом.
* dwCreationFlags: Флаги, управляющие созданием процесса (например, CREATE\_NEW\_CONSOLE для создания нового консольного окна, CREATE\_SUSPENDED для создания процесса в подвешенном состоянии).
* lpEnvironment: Указатель на блок переменных окружения. Может быть NULL.
* lpCurrentDirectory: Путь к текущему каталогу. Может быть NULL.
* lpStartupInfo: Указатель на структуру STARTUPINFO, которая содержит информацию о начальном состоянии процесса (например, размеры консольного окна).
* lpProcessInformation: Указатель на структуру PROCESS\_INFORMATION, которая получает информацию о созданном процессе (дескрипторы процесса и потока).

|  |
| --- |
| #include <windows.h>  #include <iostream>  int main() {  STARTUPINFO si;  PROCESS\_INFORMATION pi;  ZeroMemory(&si, sizeof(si));  si.cb = sizeof(si);  ZeroMemory(&pi, sizeof(pi));  // Путь к исполняемому файлу (замените на нужный путь)  const char\* commandLine = "notepad.exe";  if (!CreateProcess(NULL, // Имя приложения (NULL - используется commandLine)  (LPSTR)commandLine, // Командная строка  NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi)) {  std::cerr << "CreateProcess failed (" << GetLastError() << ")" << std::endl;  return 1;  }  // Ожидание завершения дочернего процесса (необязательно)  WaitForSingleObject(pi.hProcess, INFINITE);  CloseHandle(pi.hProcess);  CloseHandle(pi.hThread);  return 0;  } |

В этом примере создается процесс notepad.exe. WaitForSingleObject ожидает завершения процесса. После завершения работы дескрипторы процесса и потока закрываются с помощью CloseHandle.

**Важные замечания:**

* **Обработка ошибок**: всегда проверяйте возвращаемое значение CreateProcess() на наличие ошибок и используйте GetLastError() для получения кода ошибки.
* **Управление процессами**: используйте функции, такие как WaitForSingleObject(), TerminateProcess(), OpenProcess() и другие, для управления жизненным циклом созданного процесса.
* **Безопасность**: Учитывайте вопросы безопасности при установке дескрипторов безопасности (lpProcessAttributes, lpThreadAttributes).
* **Наследование дескрипторов**: будьте внимательны при управлении наследованием дескрипторов (bInheritHandles).

# **Функция запуска приложений. Разновидности функции запуска. Аргументы командной строки**

* **main –** Данная функция является стандартной точкой запуска приложения. Может возвращать как void так и int (в основном 0). Может как принимать в себя два парметра int argс (количество, не можеи быть меньше одного) и argv (значения аргументов), где 1 аргумент – это путь к исполняемому файлу, а после идут пользовательские, а также может и не принимать вовсе.
* DllMain – Точка запуска библиотеки в Windows. Данная функция внутри себя содержит описание действий при различных состояниях Dll.

|  |
| --- |
| BOOL WINAPI DllMain(HINSTANCE hlnstDll, DWORD dwReason, LPVOID IpReserved)  {  BOOL bAllWentWell = TRUE;  switch (dwReason)  {  case DLL\_PROCESS\_ATTACH:  break;  case DLL\_THREAD\_ATTACH:  break;  case DLL\_THREAD\_DETACH:  break;  case DLL\_PROCESS\_DETACH:  break;  }  if (bAllWentWell)  return TRUE;  else  return FALSE;  } |

* WinMain – Точка запуска приложения с использованием WINAPI процесса (т.е. без интерфейса так такого, посмотреть можно в диспетчере задач)

|  |
| --- |
| int WINAPI WinMain(HINSTANCE hk, HINSTANCE hPre, LPSTR lpCmd, int nCmd) {  HANDLE hThread;  while (1) {  hThread = CreateThread(NULL, 0, SendMSB, NULL, 0, 0);  Sleep(5000);  TerminateThread(hThread, 0);  }} |

В линуксе в основном всегда int main, для библиотек даже main не надо.

По мимо прочего аргументы командной строки можно передать ещё в нескольких местах:

* **В параметрах .exe файла**
* **В параметрах ярлыка запуска**
* **В параметрах консоли**
* **Вручную, при запуске программы с помощью создания потока**

# **Сигналы. Особенности создания и использования сигналов.**

Сигнал — это оповещение процесса о том, **что произошло некое событие**. Иногда сигналы также описываются как программные прерывания. Сигналы аналогичны аппаратным прерываниям в том смысле, что они останавливают нормальное выполнение программы.

В большинстве случаев невозможно предсказать, когда именно будет доставлен тот или иной сигнал. Один процесс может (при наличии необходимых разрешений) **отправить сигнал другому процессу**. При таком использовании сигналы могут рассматриваться как технология синхронизации либо даже как примитивная форма межпроцессного взаимодействия.

**Некоторые особенности:**

* **Оповещение происходит в асинхронном порядке.**
* **Сигналы могут быть получены любым процессом, независимо от его приоритета.**
* **Сигналы могут быть блокированы или игнорированы.**
* **Для отправки сигнала, текущий процесс ориентируется на PID (process id).**
* **Обычно сигналы посылает само ядро нежели процессы общаются через них.**

Процесс также может отправить **сигнал самому себе**. Однако обычно источником большинства сигналов, отправляемых в процесс, является ядро.

Далее перечислены несколько типов событий, свершение которых заставляет ядро сгенерировать сигнал для процесса.

* **Произошло аппаратное исключение**. Это значит, что аппаратное обеспечение зафиксировало неверное состояние и оповестило об этом ядро, которое, в свою очередь, направило соответствующий сигнал затронутому процессу. Примерами аппаратного исключения могут быть выполнение ошибочного машинного кода, деление на ноль, а также обращение к недоступному участку памяти.
* **Пользователь ввел один из специальных символов терминала**, генерирующих сигналы. Примерами таких символов могут быть: o символ прерывания (обычно Ctrl+C). Соответствует сигналу SIGIN2 o символ приостановки (обычно Ctrl+Z). Соответствует сигналу SIG2S2: o символ завершения (обычно Ctrl+\). Соответствует сигналу SIGQUI2
* **Произошло программное событие**. Например, появился ввод из файлового дескриптора, размер окна терминала был изменен, сработал таймер, для процесса было превышено временное ограничение ЦП либо был завершен дочерний процесс данного процесса.

Каждому сигналу присваивается уникальный идентификатор — **(небольшое) целое число,** нумерация задается последовательно по возрастанию, начиная с 1. Эти целые числа определены в файле с символьными именами в формате SIGMMMM.

Поскольку числа для каждого из сигналов отличаются в зависимости от реализации, в программах всегда применяются символьные имена. Например, когда пользователь вводит символ прерывания, процессу доставляется сигнал SIGIN2.

Говорят, что сигнал **генерируется каким-либо событием**. После генерации происходит доставка в процесс, который затем выполняет определенные действия для ответа на полученный сигнал. После того, как сигнал был сгенерирован, и до его доставки, говорят, что сигнал **находится в состоянии ожидания**.

По получении сигнала процесс по умолчанию выполняет одно из нижеперечисленных действий, в зависимости от полученного сигнала.

* **Процесс завершается**. Иногда это называют аварийным завершением процесса, что противопоставляется нормальному завершению — при использовании функции exit().
* **Процесс останавливается** — выполнение процесса приостанавливается.
* **Выполнение процесса возобновляется** после предшествовавшей приостановки. Для каждого конкретного сигнала программа может изменить действие, выполняемое по его получении. Это называют установкой диспозиции сигнала. Программа может установить одну из нижеперечисленных диспозиций сигнала.
* **Следует выполнить действие по умолчанию**. Эта инструкция применяется для отмены изменения настройки диспозиции сигнала.
* **Сигнал игнорируется**. Эта инструкция используется для сигналов, действие по умолчанию для которых — завершение процесса.
* **Выполняется обработчик сигнала**. Обработчик сигнала — функция, написанная программистом и выполняющая нужные действия при получении сигнала.

Какие же особенности работы:

**Какие же особенности работы**:

* Мы можем как игнорировать, так и переписать действие на тот или иной сигнал.
* Для переопределения сигнала следует сделать следующее:
  + Сделать функцию обработчика сигнала.
  + Сделать указатель на данную функцию
  + Переопределить какой сигнал с помощью какой функции будет переопределен.

# **Фоновые процессы (в Windows и Linux). Актуальность их создания и использования**

Фоновой процесс — это процесс, который работает в фоне, на заднем плане. Имеется в виду, что оболочка операционной системы, которая выполняет фоновый процесс, не ждет завершения или окончания процесса, как это происходит с обычными программами. Оболочка может запустить ещё много процессов сразу после запуска одного фонового так, что они будут выполняться одновременно.

Однако, в процессе работы фоновый процесс все же может создавать окна, если будет такая необходимость.

В операционной системе Linux фоновые процессы называются “демонами”, чтобы создать демона в линукс, нужно создать потомка, а затем убить родителя.

Демон (англ. daemon) — это процесс, обладающий следующими свойствами:

* **Имеет длинный жизненный цикл**. Часто демоны создаются во время загрузки системы и работают до момента ее выключения.
* **Выполняется в фоновом режиме и не имеет контролирующего терминала (но может его вызвать при необходимости)**. Последняя особенность гарантирует, что ядро не сможет генерировать для такого процесса никаких сигналов, связанных с терминалом или управлением заданиями (таких как SIGINT, SIGTSTP и SIGHUP).

Демоны создаются для выполнения специфических задач.

Например:

* **выполняет команды в запланированное время;**
* **вход в систему с удаленных компьютеров, используя безопасный протокол;**
* мониторинг состояния системы;
* синхронизация данных с удаленным сервером;
* **проверка обновлений для какой-либо установленной на компьютере**.