**Билет 1**

1. Работа с файлами: методы, режимы, примеры.

Работа с файлами в C++ осуществляется через использование класса fstream из стандартной библиотеки <fstream>. fstream позволяет открывать файлы для чтения и записи, а также производить другие операции с файлами, такие как создание, удаление и перемещение.

Для открытия файла в C++ можно использовать различные **режимы**:

ios::in - открытие файла для чтения

ios::out - открытие файла для записи

ios::trunc - удаление содержимого файла при открытии для записи

ios::app - открытие файла для записи в конец

ios::binary - открытие файла в двоичном режиме

**Пример:**

#include <iostream>

#include <fstream>

int main() {

// Создание объекта класса fstream для работы с файловым потоком

std::fstream file;

// Открытие файла для записи в конец

file.open("example.txt", std::ios::out | std::ios::app);

// Проверка успешного открытия файла

if (!file.is\_open()) {

std::cout << "Ошибка открытия файла" << std::endl;

return 1;

}

// Запись данных в файл

file << "Hello, world!" << std::endl;

// Закрытие файла

file.close();

return 0;

}

**Основные методы** для работы с файлами в C++:

* ifstream: класс для чтения данных из файла.
* ofstream: класс для записи данных в файл.
* fstream: класс для работы с файлами в режиме чтения и записи.

**Билет 2**

Вопрос: Подсистемы управления ресурсами: управление процессами, памятью, файлами, внешними устройствами.

1. **Управление процессами**: данная подсистема отвечает за управление выполнением процессов в операционной системе. Она позволяет создавать, запускать, приостанавливать, возобновлять и завершать процессы. Также она обеспечивает планирование процессов для эффективного использования ресурсов ЦПУ.
2. **Управление памятью**: эта подсистема отвечает за управление выделением и освобождением оперативной памяти для процессов. Она позволяет выделять память для хранения данных и кода процессов, а также следить за их использованием и освобождением.
3. **Управление файлами:** данная подсистема отвечает за управление файловой системой операционной системы. Она позволяет создавать, открывать, закрывать, перемещать, копировать и удалять файлы. Также она предоставляет интерфейс для чтения и записи данных в файлы.
4. **Управление внешними устройствами:** эта подсистема отвечает за управление взаимодействием операционной системы с внешними устройствами, такими как принтеры, сканеры, клавиатуры, мыши и другие периферийные устройства. Она обеспечивает обмен данными между операционной системой и устройствами, а также контролирует их работу и состояние.

**Общие для всех ресурсов – это подсистемы:**

* прикладного программного и пользовательского интерфейсов;
* защиты данных и администрирования.

**Билет 3**

Управление процессами: понятие, архитектура, состояния, завершение процесса.

Управление процессами в операционной системе относится к координации выполнения программных процессов.

**Понятие** процесса включает в себя исполняемую программу и связанные с ней ресурсы, такие как файлы, память и открытые устройства. Процесс может быть запущен как фоновая задача или взаимодействовать с пользователем через интерфейс.

**Архитектура** процесса включает в себя данные процесса (например, значения переменных), стек вызовов, регистры ЦПУ и дескрипторы открытых файлов. Они образуют контекст процесса, который сохраняется при переключении процессора между процессами.

Процессы имеют различные **состояния** во время выполнения, например:

1. Создание (Creation): процесс создан, но еще не выполнен.
2. Готовность (Ready): процесс готов к выполнению, но еще не получил процессорное время.
3. Выполнение (Running): процесс исполняется на процессоре.
4. Ожидание (Waiting): процесс ожидает какого-либо события, например, ввода-вывода или сигнала от другого процесса.
5. Завершение (Termination): выполнение процесса завершено, и он освобождает свои ресурсы.

**Завершение процесса** может происходить по разным причинам, например:

1. Нормальное завершение: процесс завершает свою работу по своей логике.
2. Завершение по сигналу: процесс получает сигнал от другого процесса или операционной системы для завершения.
3. Завершение из-за ошибки: процесс обнаруживает ошибку в своей работе и завершается.

В языке программирования C++ управление процессами осуществляется с помощью системных вызовов и функций из стандартной библиотеки, таких как fork(), exec(), wait(), exit() и kill().

**Билет4**

Управление потоками: создание, приостановка и возобновление выполнения, завершение потока.

В C++ управление потоками может быть осуществлено с помощью множества инструментов из стандартной библиотеки и операционной системы.

**Создание потока**:

В C++11 и выше создание потоков осуществляется с помощью класса std::thread. Для создания потока нужно передать функцию или лямбда-выражение в качестве аргумента конструктора std::thread.

**Приостановка выполнения потока**:

Приостановка выполнения потока может быть осуществлена с помощью функции std::this\_thread::sleep\_for. Эта функция позволяет потоку приостановить свое выполнение на определенный промежуток времени.

**Возобновление выполнения потока:**

Возобновление выполнения приостановленного потока может быть осуществлено с помощью функции std::condition\_variable::notify\_one или std::condition\_variable::notify\_all.

**Завершение потока:**

Поток можно завершить с помощью функции std::thread::join или std::thread::detach.

1. std::thread::join приводит к ожиданию завершения потока в текущем потоке исполнения.
2. std::thread::detach отсоединяет поток, позволяя ему работать самостоятельно до завершения. При этом необходимо следить за тем, чтобы объект std::thread не был уничтожен до завершения потока.

Важно отметить, что при завершении главного потока выполнение всех остальных потоков автоматически прекращается.

**Билет 5**

Параллельная обработка потоков: реализация многопоточности, синхронизация, достоинства и недостатки, примеры.

**Параллельная обработка потоков** - это подход к программированию, при котором задачи выполняются одновременно в нескольких потоках. В C++ многопоточность реализуется с помощью стандартной библиотеки.

**Реализация многопоточности** в C++ может быть достигнута созданием нескольких объектов std::thread.

**Синхронизация** между потоками часто осуществляется с помощью мьютексов (std::mutex). Мьютексы позволяют одновременному доступу к общим данным только одному потоку в определенный момент времени.

**Достоинства** параллельной обработки потоков включают:

* Ускорение выполнения задачи путем распараллеливания ее выполнения на несколько потоков.
* Возможность одновременного выполнения нескольких задач, что позволяет получать результаты быстрее.

**Недостатки** параллельной обработки потоков включают:

* Сложность реализации из-за необходимости синхронизации доступа к общим данным.
* Возможность возникновения состояний гонки и других проблем синхронизации при неправильной реализации.

Примеры задач, которые могут быть эффективно решены с использованием параллельной обработки потоков, включают:

* Обработка больших объемов данных, например, сортировка массива или поиск в большой базе данных.
* Выполнение независимых вычислений, например, поиск простых чисел или решение задачи о ранце.
* Параллельная обработка ввода-вывода, чтобы ускорить выполнение операций с диском или сетью.

**Пример:**

#include <iostream>

#include <thread>

void printHello()

{

std::cout << "Hello from thread!" << std::endl;

}

int main()

{

std::thread threadObj(printHello);

threadObj.join();

std::cout << "Hello from main thread!" << std::endl;

return 0;

}

В этом примере создается объект потока threadObj, который запускает функцию printHello. После выполнения функции, вызывается метод join(), чтобы дождаться окончания работы потока.

**Билет 6**

Создание процессов: понятие, назначение, функция CreateProcess.

**Создание процессов** - это процесс создания новой независимой программы с ее собственным адресным пространством и контекстом выполнения.

**Назначение** создания процессов включает организацию параллельного или конкурентного выполнения программ, управление ресурсами и изоляцию приложений друг от друга.

В языке программирования C++ **функция CreateProcess** из библиотеки <windows.h> позволяет создать новый процесс. Ее сигнатура выглядит следующим образом:

* lpApplicationName: имя исполняемого файла процесса или NULL, если исполняемый файл определяется lpCommandLine.
* lpCommandLine: командная строка, которая передается процессу. Этот параметр может быть NULL.
* lpProcessAttributes и lpThreadAttributes: атрибуты безопасности, которые применяются к новому процессу и потоку. Эти параметры могут быть NULL, что означает, что процесс или поток наследуют атрибуты безопасности вызывающего процесса.
* bInheritHandles: флаг, указывающий, будут ли дескрипторы открытых объектов наследоваться процессом.
* dwCreationFlags: флаги, определяющие создание процесса. Например, можно указать флаг CREATE\_NEW\_CONSOLE, чтобы создать новую консоль для процесса.
* lpEnvironment: указатель на блок памяти, содержащий переменные среды для нового процесса.
* lpCurrentDirectory: текущий рабочий каталог для нового процесса. Если эта строка является пустой, процесс использует текущий каталог вызывающего процесса.
* lpStartupInfo: указатель на структуру STARTUPINFO, которая указывает атрибуты для нового процесса, такие как главное окно, размеры и положение окна и т. д.
* lpProcessInformation: указатель на структуру PROCESS\_INFORMATION, которая будет содержать информацию о созданном процессе, такую как его идентификатор (PID) и дескриптор процесса.

Таким образом, функция CreateProcess позволяет создавать новые процессы с заданными параметрами и получать информацию о созданных процессах.

**Билет 7**

Создание потоков: понятие, назначение, функция CreateThread.

**Создание потоков** в программировании позволяет выполнять несколько задач параллельно в одновременном режиме. В языке C++ для создания потоков используется функция CreateThread().

**Назначение** создания потоков в C++ заключается в параллельном выполнении нескольких задач или процессов одновременно. Создание потоков позволяет распределить нагрузку на процессор между несколькими ядрами и повысить общую производительность программы.

**Функция CreateThread()** позволяет создать новый поток в программе.

Параметры функции:

lpThreadAttributes - атрибуты безопасности потока. В большинстве случаев этот параметр можно просто указать как NULL.

dwStackSize - размер стека для потока. Если указать 0, то стек будет установлен по умолчанию.

lpStartAddress - указатель на функцию, которая будет выполняться в новом потоке.

lpParameter - указатель на данные, которые будут переданы в функцию lpStartAddress.

dwCreationFlags - флаги создания потока. Например, CREATE\_SUSPENDED приостанавливает выполнение потока после его создания.

lpThreadId - указатель на переменную, в которую будет записан идентификатор (ID) созданного потока.

Функция CreateThread() создает новый поток, который начинает выполнение с указанной функции lpStartAddress. Тело этой функции должно быть определено пользователем и должно соответствовать сигнатуре LPTHREAD\_START\_ROUTINE.

**Билет 8**

Обмен данными между процессами: методы межпроцессорного взаимодействия, передача сообщений.

**Взаимодействие между процессами** в операционной системе может происходить с помощью различных методов. Один из таких методов - метод передачи сообщений.

В языке C++ для **передачи сообщений** между процессами можно использовать различные механизмы, такие как:

1. Сокеты (sockets): Сокеты позволяют процессам обмениваться данными по сети или на одной машине. Реализация сокетов в C++ осуществляется с помощью сетевых библиотек (например, библиотеки Winsock для Windows или библиотеки Berkeley Sockets для Unix-подобных систем).
2. Именованные каналы (named pipes): Именованные каналы позволяют процессам обмениваться данными через именованный канал, созданный операционной системой. В C++ для работы с именованными каналами можно использовать функции из библиотек, таких как WinAPI или Posix.
3. Конвейеры (pipelines): Конвейеры представляют собой упорядоченную последовательность процессов, связанных между собой таким образом, что вывод одного процесса становится вводом для следующего. Для работы с конвейерами в C++ можно использовать создание процессов с помощью функций fork и exec.
4. Системные вызовы (system calls): Системные вызовы позволяют процессам выполнять определенные действия, включая отправку и получение сообщений. В C++ для работы с системными вызовами можно использовать функции из стандартной библиотеки C или специфичные для операционной системы.

Кроме того, существуют и другие методы **межпроцессорного взаимодействия,** такие как разделяемая память, семафоры, мьютексы и т.д., которые также могут быть использованы для обмена данными между процессами в C++. Выбор конкретного метода зависит от требований приложения и особенностей операционной системы, на которой оно будет работать.

**Билет 9**

Обмен данными между процессами: анонимные и именованные каналы.

В C++ существуют два основных способа обмена данными между процессами с использованием каналов: анонимные каналы и именованные каналы.

**Анонимные каналы:**

* Анонимные каналы позволяют обмениваться данными между процессами, которые находятся в отношении родительского-потомок. Они являются однонаправленными, то есть данные передаются только в одном направлении.
* Для создания анонимного канала в C++ необходимо использовать функцию **pipe()**. Эта функция возвращает два дескриптора файлов: один для чтения данных из канала (read end), и второй для записи данных в канал (write end).
* Данные записываются в канал с помощью функции **write()**, а читаются из канала с помощью функции **read()**.

**Именованные каналы:**

* Именованные каналы позволяют обмениваться данными между процессами, которые не находятся в отношении родительского-потомок. То есть эти процессы могут быть запущены независимо друг от друга.
* Для создания именованного канала в C++ необходимо использовать функцию **mkfifo()**. Эта функция создает файл с указанным именем, который будет представлять собой именованный канал.
* Данные записываются в именованный канал с помощью функции **open()** с флагом **O\_WRONLY**, а читаются из него с помощью функции **open()** с флагом **O\_RDONLY**.

Как анонимные, так и именованные каналы являются синхронными, то есть процесс, записывающий данные в канал, блокируется, пока процесс, считывающий данные из канала, не прочитает их.

**Билет 10**

Обмен данными между процессами: почтовые ящики.

Взаимодействие процессов с помощью **почтовых ящиков** - это один из способов обмена данными. В языке C++ для этого можно использовать различные механизмы, такие как сокеты, FIFO (First-In-First-Out) или разделяемую память.

Реализация обмена данными между процессами с помощью почтовых ящиков зависит от конкретной ситуации и требований к проекту. Необходимо выбрать подходящий механизм в соответствии с требованиями и возможностями проекта.

**Билет 11**

Сетевое программирование сокетов: понятие, серверные функции, клиентские функции сокета.

**Сетевое программирование сокетов** — это процесс разработки программ, использующих сокеты для обмена данными между компьютерами через сеть. Сам сокет представляет собой интерфейс между сетевым приложением и транспортным протоколом.

В C++, для работы с сокетами, используется стандартная библиотека **<sys/socket.h>**. Для создания сервера и клиента, используются различные функции сокета.

**Серверные функции сокета:**

1. socket() - создает новый сокет и возвращает его дескриптор. Принимает параметры, такие как домен (AF\_INET для IPv4) и тип (SOCK\_STREAM для TCP, SOCK\_DGRAM для UDP) сокета.
2. bind() - связывает сокет с адресом и портом, на которых он будет прослушивать входящие соединения. Принимает дескриптор сокета, структуру, содержащую информацию об адресе сервера.
3. **listen()** - указывает сокету прослушивать входящие соединения. Принимает дескриптор сокета и максимальное количество ожидающих соединений.
4. accept() - принимает входящее соединение и создает новый сокет для обмена данными с клиентом. Принимает дескриптор сокета и указатель на структуру, в которую будет записан адрес клиента.

Клиентские функции сокета:

1. socket() - создает новый сокет и возвращает его дескриптор. Принимает параметры, такие же, как и для сервера.
2. connect() - устанавливает соединение с сервером на указанном адресе и порту. Принимает дескриптор сокета, структуру с информацией об адресе сервера и ее размер.
3. send() - отправляет данные на удаленную сторону. Принимает дескриптор сокета, указатель на буфер с данными и их размер.
4. recv() - принимает данные от удаленной стороны. Принимает дескриптор сокета, указатель на буфер, в который будут записаны принятые данные, и максимальный размер буфера.

Это основные функции для сетевого программирования сокетов в C++. С их помощью можно создавать клиент-серверные приложения и обмениваться данными по сети.

**Билет 12**

Сетевое программирование сокетов: понятие, работа с протоколом HTTP.

**Сетевое программирование сокетов** — это процесс разработки программ, использующих сокеты для обмена данными между компьютерами через сеть. Сам сокет представляет собой интерфейс между сетевым приложением и транспортным протоколом.

**Протокол HTTP** (Hypertext Transfer Protocol) — это протокол прикладного уровня, который используется для передачи данных между клиентом и сервером в сети Интернет. Программа, работающая с протоколом HTTP, может отправлять HTTP-запросы на сервер и получать HTTP-ответы от сервера.

Для работы с протоколом HTTP в языке программирования C++ можно использовать различные библиотеки, такие как libcurl или boost::beast.

**Билет 13**

Сетевое программирование сокетов: понятие, работа с протоколами SMTP и FTP.

**Сетевое программирование сокетов** — это процесс разработки программ, использующих сокеты для обмена данными между компьютерами через сеть. Сам сокет представляет собой интерфейс между сетевым приложением и транспортным протоколом.

**SMTP** (Simple Mail Transfer Protocol) - это протокол, используемый для отправки электронной почты. Для работы с протоколом SMTP в C++ существуют различные библиотеки, такие как Poco, Boost.Asio и libcurl. С помощью этих библиотек можно установить соединение с сервером SMTP, отправить электронное письмо с указанными заголовками и содержимым, а также обработать ответы сервера.

**FTP** (File Transfer Protocol) - это протокол, используемый для передачи файлов по сети. В C++ для работы с протоколом FTP можно использовать библиотеку libcurl. С ее помощью можно устанавливать соединение с FTP-сервером, загружать и скачивать файлы, а также выполнять другие операции, поддерживаемые протоколом FTP.

**Билет 14**

Статические библиотеки: достоинства и недостатки, создание и установка.

**Статические библиотеки** являются одним из способов организации исходного кода в программировании. Они представляют собой набор объектных файлов, объединенных в единый файл с расширением .lib (в Windows) или .a (в UNIX-подобных системах).

**Достоинства статических библиотек:**

1. Простота использования: после компиляции библиотеки, она становится доступной для использования в других проектах без необходимости компиляции исходного кода снова.
2. Более низкая вероятность конфликта зависимостей: статические библиотеки включают в себя все необходимые зависимости, что уменьшает возможность конфликтов с другими версиями зависимых библиотек.
3. Ускорение времени выполнения: так как код из статической библиотеки уже скомпилирован и объединен, он может выполняться быстрее по сравнению с динамическими библиотеками, которые требуют загрузки и связывания при запуске программы.

**Недостатки статических библиотек:**

1. Размер: статическая библиотека может занимать много места на жестком диске, особенно если она содержит много функциональности.
2. Необновляемость: после создания статической библиотеки ее код не может быть обновлен без повторной компиляции всего проекта, который использует эту библиотеку.
3. Дублирование кода: если несколько приложений используют одну и ту же статическую библиотеку, код из нее дублируется для каждого приложения, что может занимать дополнительное место на диске.

**Создание и установка** статической библиотеки в C++:

1. Написание исходного кода: создайте и реализуйте необходимые функции, классы и прочие компоненты, которые вы хотите включить в библиотеку.
2. Компиляция: скомпилируйте исходный код с использованием компилятора C++. Для создания объектных файлов включите флаг -c, чтобы компилятор не выполнил линковку.
3. Создание статической библиотеки: используйте утилиту ar для создания статической библиотеки из объектных файлов.
4. Установка: после создания статической библиотеки ее можно скопировать в нужное место в файловой системе и указать пути к ней при компиляции проекта, который будет использовать эту библиотеку.

**Билет 15**

Динамически подключаемые библиотеки DLL: понятие, преимущества, механизм работы, явное и неявное связывание.

**Динамически подключаемая библиотека (DLL)** - это файл, который содержит код и данные, которые могут быть использованы несколькими приложениями одновременно. Такие библиотеки позволяют приложениям использовать общие функции и ресурсы, что упрощает разработку и обновление программного обеспечения.

**Преимущества использования DLL:**

1. Экономия памяти: DLL разделяется несколькими приложениями, поэтому каждое приложение может использовать общие функции и ресурсы без необходимости повторного их создания.
2. Легкость обновления: при изменении кода или данных в DLL, достаточно заменить только этот файл, не затрагивая само приложение.
3. Упрощение разработки: можно создавать подключаемые модули с общими функциями и использовать их в разных приложениях.

**Механизм работы с DLL:**

1. Компиляция: код и данные функций, которые вы хотите сделать доступными в DLL, компилируются в отдельные объектные файлы.
2. Связывание: компоновщик собирает объектные файлы в DLL, создавая таким образом исполняемый код и данные, доступные приложениям.
3. Подключение DLL: приложение должно явно или неявно подключить DLL, чтобы получить доступ к ее функциям и данным.
4. Динамическое связывание: при явном связывании, функции и данные из DLL вызываются или используются с помощью указателей на функции или переменные. При неявном связывании, функции и данные из DLL вызываются или используются непосредственно, как будто они находятся в самом приложении.

Явное связывание и неявное связывание - два способа использования функций и данных из DLL.

**Явное связывание** - это когда приложение явно вызывает функции из DLL с помощью указателей на функции или переменные, которые предварительно загружаются из DLL.

**Неявное связывание** - это когда приложение использует функции и данные из DLL так, будто они находятся в самом приложении. В этом случае компилятор автоматически создает вызовы функций или доступ к данным из DLL без явного указания указателей.

**Билет 16**

Сервисы: понятие и примеры служб, управление службами, алгоритм создания службы.

**Сервисы** - это фрагменты кода, которые предоставляют определенные функциональные возможности или услуги для приложения. Они могут быть реализованы в виде отдельных классов или функций, которые выступают в роли сервисных провайдеров.

Ниже приведены **примеры служб в C++:**

1. Служба логирования: предоставляет функции записи логов приложения в файл или на консоль.
2. Служба работы с базами данных: обеспечивает возможности работы с различными типами баз данных, включая подключение, выполнение запросов и получение результатов.
3. Служба сетевого взаимодействия: обеспечивает функции работы с сетевым протоколом, таким как TCP или UDP, для обмена данными между клиентом и сервером.
4. Служба сжатия данных: предоставляет возможности сжатия и декомпрессии данных для уменьшения их размера и улучшения производительности.
5. Служба графического интерфейса: предоставляет набор функций и классов для создания графического пользовательского интерфейса (GUI).

Это только некоторые примеры служб в C++. Фактически, сервисы могут включать в себя широкий спектр функциональных возможностей и различных областей применения.

**Управление службами в С++** осуществляется с использованием функций из стандартной библиотеки Windows.h.

**Алгоритм создания службы в C++:**

1. Определить функциональность службы - что она будет делать и какие ресурсы будет предоставлять;
2. Разработать класс или модуль, реализующий эту функциональность;
3. Создать экземпляр этого класса или вызвать этот модуль при запуске службы;
4. Настроить обработку событий - например, чтобы служба могла отвечать на запросы или изменять свою работу в ответ на определенные события;
5. Настроить автоматическое запуск и остановку службы при старте и выключении системы;
6. Скомпилировать и запустить службу, чтобы убедиться, что она работает правильно;
7. Настроить логирование или вывод отладочной информации для службы, чтобы мы могли отслеживать ее работу и исправлять ошибки.

**Билет 17**

Виртуальная память: понятие, реализация в операционной системе структура.

**Виртуальная память** - это механизм, который позволяет операционной системе использовать дополнительную область памяти на жестком диске в качестве расширения оперативной памяти компьютера. Операционная система разделяет физическую память на равные блоки, называемые страницами, и поддерживает отображение этих страниц на виртуальную память.

Реализация виртуальной памяти в операционной системе обычно включает следующие компоненты и структуры:

1. Таблица страниц (Page Table): Это основная структура данных, используемая операционной системой для отображения виртуальных страниц на физические. Каждая страница виртуальной памяти имеет соответствующую запись в таблице страниц, содержащую адрес физической страницы.
2. Трансляционный буфер (Translation Lookaside Buffer, TLB): Это кэш-подобная структура данных, которая хранит последние использованные соответствия между виртуальными и физическими адресами. TLB ускоряет процесс поиска соответствующих физических страниц, избегая обращения к таблице страниц.
3. Страница отображения (Page Mapping): Операционная система использует страницу отображения для отображения виртуальной памяти на физическую память. При доступе к виртуальной памяти операционная система проверяет таблицу страниц и использует страницу отображения для определения соответствующей физической страницы.

В языке программирования C++ взаимодействие с виртуальной памятью обычно осуществляется через использование указателей и функций работы с памятью, таких как new и delete. C++ предоставляет возможность динамического выделения памяти с использованием оператора new и освобождение памяти с помощью оператора delete. Операционная система занимается управлением реальной выделенной памятью и ее отображением в виртуальную память при необходимости.

**Билет 18**

Выделение памяти процессам: функция VirtualAlloc, куча процесса, трансляция адресов.

Выделение памяти для процессов в операционной системе Windows можно осуществить с использованием функции VirtualAlloc, которая доступна в языке программирования C++.

**Функция VirtualAlloc** позволяет выделить блок памяти виртуального адресного пространства процесса. Она имеет следующую сигнатуру:

* **lpAddress** - указатель на начальный адрес выделенного блока памяти. Если этот параметр равен NULL, то операционная система сама выберет доступный адрес.
* **dwSize** - размер выделяемой памяти в байтах.
* **flAllocationType** - тип выделения памяти. Может быть одним из следующих значений:
  + MEM\_COMMIT: выделить память и заполнить ее нулями.
  + MEM\_RESERVE: зарезервировать память без физического выделения. Фактическое выделение может быть произведено позже с помощью функции VirtualAlloc с флагом MEM\_COMMIT.
  + MEM\_RESET: сбросить содержимое выделенной памяти до нулей.
* **flProtect** - тип защиты выделенной памяти. Может быть одним из следующих значений:
  + PAGE\_EXECUTE\_READ: память доступна для выполнения и чтения.
  + PAGE\_READWRITE: память доступна для чтения и записи.

Кроме функции VirtualAlloc, в C++ также используется понятие "**кучи процесса". Куча** - это область памяти, используемая для динамического выделения и освобождения памяти во время выполнения программы. Выделение памяти с использованием функции VirtualAlloc можно рассматривать как выделение памяти из кучи процесса.

**Трансляция адресов в C++** обычно происходит автоматически компилятором и операционной системой. Программа работает с виртуальными адресами, которые впоследствии транслируются в реальные физические адреса памяти при выполнении программы. Этот процесс называется виртуальной памятью и обеспечивается операционной системой.

**Билет 19**

Управление памятью: сборка мусора, освобождение ресурсов, создание деструктора, указателей.

Управление памятью является важным аспектом программирования на C++. Оно включает в себя сборку мусора, освобождение ресурсов, создание деструктора и управление указателями.

**Сборка мусора в C++** отсутствует в стандартной библиотеке языка. Вместо этого программист самостоятельно отвечает за освобождение динамически выделенной памяти после ее использования. Это обычно делается с помощью оператора delete для указателей на объекты или оператора delete[] для указателей на массивы.

**Освобождение ресурсов** также требуется при работе с неуправляемыми ресурсами, такими как файлы, сетевые соединения и т. д. Для этого используются функции и/или классы, предоставленные операционной системой или сторонними библиотеками. Например, в случае файлов можно использовать функцию fclose для закрытия файла после его использования.

Для обеспечения освобождения ресурсов в C++ можно использовать концепцию RAII (Resource Acquisition Is Initialization), которая заключается в создании класса, ответственного за управление ресурсами, и освобождении их в его **деструкторе.** Объект такого класса создается при получении ресурса и уничтожается при его освобождении. Это позволяет гарантировать, что ресурсы будут освобождены даже в случае исключения или неожиданного выхода из функции.

**Управление указателями** также является важной частью управления памятью в C++. В C++ есть несколько видов указателей, таких как указатели на объекты, указатели на массивы и указатели на функции. Правильное использование и освобождение памяти, на которую они указывают, является ответственностью программиста. Неправильное использование указателей может привести к утечкам памяти или нежелательному поведению программы.

В целом, управление памятью в C++ требует аккуратности и осознанного подхода к работе с динамически выделенной памятью, ресурсами и указателями. Соблюдение правил и использование соответствующих техник и средств языка поможет избежать утечек памяти и обеспечить корректную работу программы.

**Билет 20**

Работа с буфером обмена: назначение буфера обмена, форматы данных, функции для работы с буфером обмена.

**Буфер обмена** представляет собой временное хранилище, используемое для копирования и перемещения данных между различными приложениями или компонентами в операционной системе.

**Назначение буфера обмена** состоит в том, чтобы позволить пользователям копировать текст, изображения, файлы и другие данные из одного приложения в другое, а также перемещать эти данные внутри приложений.

**Форматы данных,** поддерживаемые в буфере обмена, зависят от операционной системы, на которой работает приложение. В Windows, например, буфер обмена может содержать текст, изображения, файлы, HTML-код и другие форматы данных. В Linux и macOS тоже поддерживаются различные форматы данных.

В C++ буфер обмена можно использовать с помощью **функций** из стандартной библиотеки. Некоторые из них:

1. Функция **OpenClipboard** используется для открытия буфера обмена.
2. Функция **CloseClipboard** закрывает буфер обмена.
3. Функция **EmptyClipboard** очищает содержимое буфера обмена.
4. Функция **IsClipboardFormatAvailable** позволяет проверить наличие определенного формата данных в буфере обмена.
5. Функции **SetClipboardData** и **GetClipboardData** используются для записи и чтения данных из буфера обмена соответственно.

Кроме стандартных функций, существуют также сторонние библиотеки, которые предоставляют более удобные и функциональные возможности для работы с буфером обмена в C++, например, Qt или Boost.

Практика

// 1. Ввод и вывод данных

#include <iostream>

int main() {

int a; std::cin >> a; std::cout << "Вы ввели: " << a << std::endl;

return 0;

}

// 2. Работа с условным оператором

#include <iostream>

int main() {

int a; std::cin >> a;

if (a > 0) std::cout << "Положительное";

else if (a < 0) std::cout << "Отрицательное";

else std::cout << "Ноль";

return 0;

}

// 3. Использование циклов

#include <iostream>

int main() {

for (int i = 1; i <= 10; ++i) std::cout << i << " ";

return 0;

}

// 4. Обработка действительных чисел

#include <iostream>

#include <cmath>

int main() {

double a; std::cin >> a;

std::cout << "Квадратный корень: " << sqrt(a) << std::endl;

return 0;

}

// 5. Работа с массивами

#include <iostream>

int main() {

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

for (int i = 0; i < 5; ++i) std::cout << arr[i] << " ";

return 0;

}

// 6. Создание функций

#include <iostream>

int sum(int a, int b) { return a + b; }

int main() {

std::cout << sum(2, 3);

return 0;

}

// 7. Обработка строк и символов

#include <iostream>

#include <string>

int main() {

std::string s; std::cin >> s;

std::cout << "Длина строки: " << s.size() << std::endl;

return 0;

}

// 8. Словари и множества

#include <iostream>

#include <set>

int main() {

std::set<int> s = {1, 2, 3};

s.insert(4);

for (int x : s) std::cout << x << " ";

return 0;

}

// 9. Обработка файлов

#include <fstream>

int main() {

std::ofstream out("file.txt");

out << "Hello, file!";

out.close();

return 0;

}

// 10. Использование алгоритмов стандартной библиотеки

#include <iostream>

#include <vector>

#include <algorithm>

int main() {

std::vector<int> v = {3, 1, 4, 1, 5};

std::sort(v.begin(), v.end());

for (int x : v) std::cout << x << " ";

return 0;

}

// 11. Создание статической библиотеки

// Создайте файл mylib.h и mylib.cpp с функцией, например:

// mylib.h

#pragma once

int square(int x);

// mylib.cpp

#include "mylib.h"

int square(int x) { return x \* x; }

// Основной файл

#include <iostream>

#include "mylib.h"

int main() {

std::cout << square(5);

return 0;

}

// 12. Создание динамической библиотеки

// Файл mylib.h

int multiply(int a, int b);

// Файл mylib.cpp

int multiply(int a, int b) { return a \* b; }

// Основной файл

#include <iostream>

extern "C" int multiply(int, int);

int main() {

std::cout << multiply(3, 4);

return 0;

}

// 13. Создание и использование процессов

#include <windows.h>

int main() {

STARTUPINFO si = { sizeof(si) };

PROCESS\_INFORMATION pi;

wchar\_t command[] = L"notepad";

CreateProcess(NULL, command, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi);

CloseHandle(pi.hProcess);

CloseHandle(pi.hThread);

return 0;

}

// 14. Создание и использование потоков

#include <iostream>

#include <thread>

void func() { std::cout << "Поток!"; }

int main() {

std::thread t(func);

t.join();

return 0;

}

// 15. Обмен данными между процессами

#include <windows.h>

#include <iostream>

int main() {

HANDLE h = CreateFileMapping(INVALID\_HANDLE\_VALUE, NULL, PAGE\_READWRITE, 0, 256, L"MyMap");

char\* data = (char\*)MapViewOfFile(h, FILE\_MAP\_ALL\_ACCESS, 0, 0, 256);

strcpy\_s(data, 256, "Hello!");

UnmapViewOfFile(data);

CloseHandle(h);

return 0;

}

// 16. Использование асинхронных методов

#include <future>

#include <iostream>

int compute() { return 42; }

int main() {

auto fut = std::async(compute);

std::cout << fut.get();

return 0;

}

// 18. Сетевое программирование сокетов

#include <winsock2.h>

#include <iostream>

#pragma comment(lib, "ws2\_32.lib")

int main() {

WSADATA wsa; WSAStartup(MAKEWORD(2, 2), &wsa);

SOCKET s = socket(AF\_INET, SOCK\_STREAM, 0);

closesocket(s);

WSACleanup();

return 0;

}

// 19. Создание программы для архивации файлов

#include <iostream>

#include <fstream>

#include <sstream>

int main() {

std::ifstream in("input.txt", std::ios::binary);

std::ofstream out("output.zip", std::ios::binary);

std::ostringstream oss;

oss << in.rdbuf();

out << oss.str();

return 0;

}

// 20. Работа с буфером обмена

#include <windows.h>

#include <iostream>

int main() {

OpenClipboard(0);

EmptyClipboard();

HGLOBAL h = GlobalAlloc(GMEM\_MOVEABLE, 256);

memcpy(GlobalLock(h), "Hello, Clipboard!", 17);

GlobalUnlock(h);

SetClipboardData(CF\_TEXT, h);

CloseClipboard();

return 0;

}