Работа с адресами и памятью ПК;

Указатель и ссылка. Введение в адреса.

Адрес - это числовое значение, которое указывает на конкретное место в памяти. Конкретно в C++ адрес может храниться в указателе, т.к **Указатель** - это переменная, которая хранит адрес другой переменной.

Синтаксис указателя: `int* ptr;` (*ptr - указатель на int*).

Взятие адреса (это ссылка): оператор & используется для получения адреса переменной. *Например*,

```
int a = 10;
```

```
int* ptr = &a;
```

Процесс Разыменование: оператор * используется для доступа к значению по адресу. Например, int value = *ptr;.

Адрес как глобальное;

адреса памяти часто представляются в шестнадцатеричной системе счисления. Адреса, такие как **0xA1, 0xFFF1, 0xBB3, 0xGH1345**, являются шестнадцатеричными числами, где префикс **0x** указывает на то, что число записано в шестнадцатеричной системе (X16).

Шестнадцатеричная система (или **хексадесятичная система**) использует основание 16. Она включает цифры от 0 до 9 и буквы от A до F, где A соответствует 10, B — 11, и так далее, до F, которое соответствует 15.

Например, **0xA1** (X16-> X10) в десятичной системе равно 10×16^1+1×16^0=161.

Адреса памяти:

- Адреса памяти представляют собой числа, указывающие на определённое место в памяти компьютера.
- Они записываются в шестнадцатеричном формате для удобства, так как они короче и легче читаются по сравнению с десятичными числами, особенно для больших адресов.

Адресное пространство;

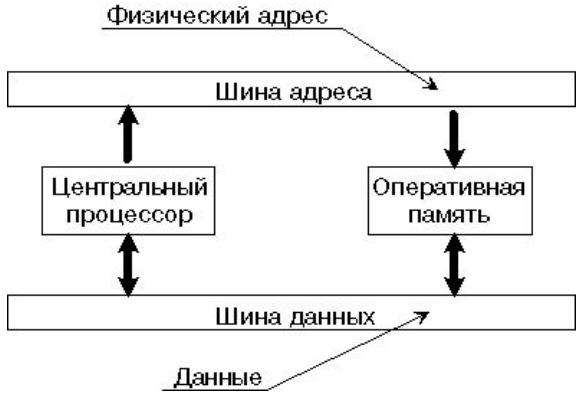
Адресное пространство — это абстракция, используемая операционными системами и процессорами для управления памятью. Оно представляет собой диапазон адресов, которые процесс может использовать для обращения к памяти. Адресное пространство может быть виртуальным или физическим.

Физическое адресное пространство:

- Это реальная память, имеющаяся в системе, и включающая оперативную память (RAM) и другие физические устройства памяти.
- Размер физического адресного пространства ограничен количеством установленных модулей памяти.

Виртуальное адресное пространство:

- Это абстракция, позволяющая каждому процессу иметь своё собственное адресное пространство, независимо от физической памяти.
- Операционная система и аппаратное обеспечение (например, MMU — Memory Management Unit) отображают виртуальные адреса в физические адреса.
- Виртуальное адресное пространство часто больше физической памяти, что позволяет использовать технику подкачки (paging) для управления памятью и выполнения более крупных программ.



Компоненты/сегменты адресного пространства;

Сегмент кода (text segment):

- Содержит исполняемый код программы.
- Обычно этот сегмент защищён от записи, чтобы предотвратить модификацию кода во время выполнения.

Сегмент данных (data segment):

- Включает в себя глобальные и статические переменные.
- Делится на две части:
  - **Инициализированные данные**: переменные, которые имеют начальное значение.
  - **Неинициализированные данные (BSS — Block Started by Symbol)**: переменные, которые не инициализированы.

Куча (heap):

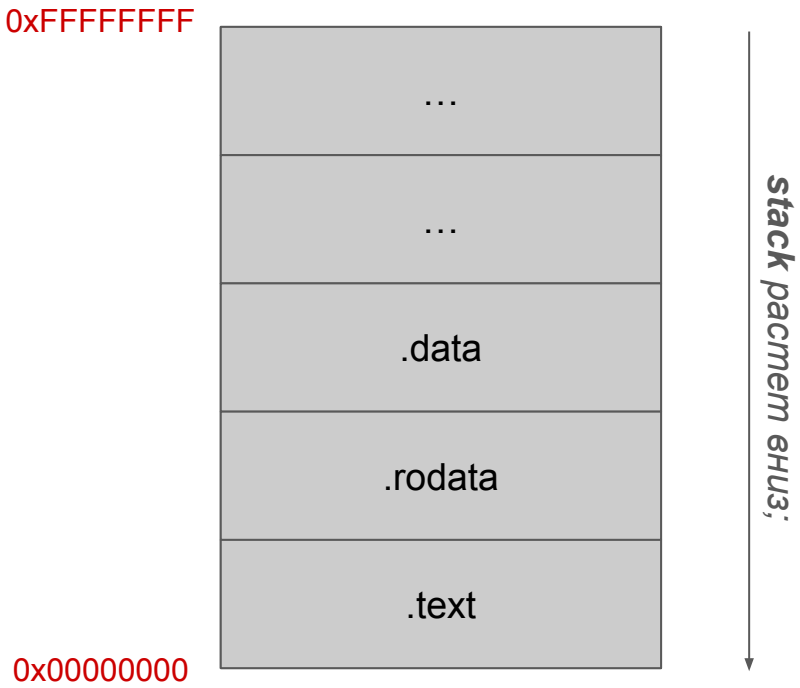
- Используется для динамического выделения памяти во время выполнения программы.
- Расширяется и сокращается по мере выделения и освобождения памяти.

Стек (stack):

- Используется для хранения локальных переменных, параметров функций и адресов возврата.
- Стек растёт и уменьшается по мере вызова и завершения функций.

32 битное адресное пространство;

В 32-битной системе виртуальное адресное пространство процесса может выглядеть примерно так:



0x00000000:

- Это наименьший адрес в 32-битной системе.
- В двоичном формате: 00000000000000000000000000000000 (32 нуля).
- В десятичной системе: 0.

0xFFFFFFFF:

- Это наибольший адрес в 32-битной системе.
- В двоичном формате: 11111111111111111111111111111111 (32 единицы).
- В десятичной системе: 2^32−1=4294967295.

тип uint

типы данных uint и их варианты используются для представления целых чисел без знака (unsigned integers). Эти типы данных важны для работы с числами, которые всегда неотрицательны, *например*, для представления размеров, индексов, или адресов в памяти.

самый привычным нам уже тип это:

unsigned int:

- **Размер**: Занимает 4 байта (32 бита) на большинстве современных систем.
- **Диапазон значений**: (от 0 до 4,294,967,295)

```
unsigned int u = 10;
```

тип uint. разновидности;

uint8_t:

- **Размер**: 1 байт (8 бит).
- **Диапазон значений**: От 0 до 255.
- `uint8_t u8 = 255;`

#include <cstdint>

uint16_t:

- **Размер**: 2 байта (16 бит).
- **Диапазон значений**: От 0 до 65,535

uint32_t:

- **Размер**: 4 байта (32 бита).
- **Диапазон значений**: От 0 до 4,294,967,295

uint64_t:

- **Размер**: 8 байт (64 бита).
- **Диапазон значений**: От 0 до 264−12^{64} - 1264−1 (от 0 до 18,446,744,073,709,551,615).

Для чего нам нужны безнаковый типы?

1. **Оптимизация использования памяти**: Использование типов данных без знака позволяет более эффективно использовать память и представлять большие числа, так как вся доступная область значений используется для положительных чисел.
2. **Логическая безопасность**: Для значений, которые не могут быть отрицательными (например, размеры массивов, индексы), использование беззнаковых типов данных предотвращает логические ошибки и добавляет ясность коду.
3. **Совместимость с низкоуровневыми операциями**: В системном программировании и работе с аппаратным обеспечением часто используются беззнаковые типы для работы с адресами памяти, регистрами процессора и другими подобными задачами.

Введение в WINAPI

Windows API (Application Programming Interface) — это набор функций, предоставляемых операционной системой Windows для разработки приложений. Он позволяет разработчикам взаимодействовать с различными компонентами операционной системы, такими как файловая система, процессы, потоки, сеть, графический интерфейс и другие ресурсы.

Windows API делится на несколько категорий:

1. **Base Services**: Работа с файлами, устройствами, памятью и процессами.
2. **Advanced Services**: Работа с объектами синхронизации, журналами событий и защитой данных.
3. **Graphics Device Interface (GDI)**: Рендеринг графики и управление устройствами вывода.
4. **User Interface**: Создание и управление окнами, диалоговыми окнами и элементами управления.
5. **Network Services**: Работа с сетевыми протоколами и ресурсами.
6. **Windows Shell**: Интеграция приложений с оболочкой Windows (например, Проводник).

тип HANDLE

HANDLE — это тип, используемый в Windows API для представления дескрипторов объектов. Дескриптор — это уникальный идентификатор, который операционная система присваивает различным ресурсам для управления ими.

- **Тип данных**: HANDLE обычно определяется как void* (указатель на void), что позволяет ему хранить указатель на любой тип данных.
- **Использование**: HANDLE возвращается функциями Windows API при создании или открытии объекта и используется для взаимодействия с этим объектом.
- **Примеры объектов**: файлы, процессы, потоки, события, мьютексы, семафоры, окна и т. д.

тип DWORD

DWORD — это тип данных, представляющий 32-битное беззнаковое целое число. Его часто используют для представления различных числовых значений в Windows API.

- **Тип данных**: typedef unsigned long DWORD; (обычно unsigned long — это 32-битное целое число).
- **Использование**: DWORD используется для представления длин чисел, состояний, кодов ошибок, размеров и других данных.

Основные функции WINAPI

Функции Windows API предоставляют множество возможностей для работы с файлами, процессами, памятью и другими ресурсами операционной системы.

- **CreateFile** — это функция Windows API, которая открывает существующий файл или создает новый файл, устройство, каталог или объект почтового ящика. Она возвращает дескриптор, который можно использовать для доступа к файлу или объекту.
- **WriteFile** — это функция Windows API, которая записывает данные в файл или устройство, на которое указывает дескриптор.
- **ReadFile** — это функция Windows API, которая считывает данные из файла или устройства, на которое указывает дескриптор.
- **CloseHandle** — это функция Windows API, которая закрывает дескриптор, открытый функциями CreateFile, CreateProcess и другими.
- и тп

CreateFile

```
HANDLE CreateFile(
    LPCSTR lpFileName,           // Имя файла или объекта
    DWORD dwDesiredAccess,       // Режим доступа (чтение, запись и т.д.)
    DWORD dwShareMode,           // Режим совместного доступа
    LPSECURITY_ATTRIBUTES lpSecurityAttributes, // Указатель на структуру
безопасности
    DWORD dwCreationDisposition, // Действие создания (создать, открыть и т.д.)
    DWORD dwFlagsAndAttributes, // Атрибуты и флаги файла
    HANDLE hTemplateFile         // Дескриптор шаблона файла (может быть NULL)
);
```

WriteFile

```
BOOL WriteFile(
   HANDLE hFile,                // Дескриптор файла
   LPCVOID lpBuffer,            // Указатель на буфер данных
   DWORD nNumberOfBytesToWrite, // Число байтов для записи
   LPDWORD lpNumberOfBytesWritten, // Указатель на переменную для записи числа
записанных байтов
   LPOVERLAPPED lpOverlapped    // Указатель на структуру OVERLAPPED (может быть
NULL)
);
```

ReadFile

```
BOOL ReadFile(
   HANDLE hFile,                // Дескриптор файла или устройства
   LPVOID lpBuffer,             // Указатель на буфер для чтения данных
   DWORD nNumberOfBytesToRead,  // Количество байтов для чтения
   LPDWORD lpNumberOfBytesRead, // Указатель на переменную для хранения
количества прочитанных байтов
   LPOVERLAPPED lpOverlapped    // Указатель на структуру OVERLAPPED для
асинхронного чтения (может быть NULL)
);
```

CloseHandle

```
BOOL CloseHandle(
   HANDLE hObject  // Дескриптор объекта
);
```

Параметры:

- **hObject**: Дескриптор объекта, который нужно закрыть.

Возвращаемое значение:

Возвращает TRUE в случае успеха и FALSE в случае ошибки.

Тип PROCESSENTRY32 и функции работы с ним

PROCESSENTRY32 — это структура в Windows API, используемая для хранения информации о процессе, который найден в момент создания снимка процессов системы. Эта структура используется совместно с функциями CreateToolhelp32Snapshot, Process32First и Process32Next, которые позволяют получать список текущих процессов в системе.

PROCESSENTRY32 определена в заголовочном файле tlhelp32.h и содержит следующую информацию;

PROCESSENTRY32

```
typedef struct tagPROCESSENTRY32 {
   DWORD   dwSize;             // Размер структуры в байтах
   DWORD   cntUsage;           // Количество дескрипторов, открытых для этого процесса
   DWORD   th32ProcessID;      // Идентификатор процесса
   ULONG_PTR th32DefaultHeapID;// Идентификатор дефолтного heap для этого процесса
   DWORD   th32ModuleID;       // Идентификатор модуля
   DWORD   cntThreads;         // Количество потоков в процессе
   DWORD   th32ParentProcessID;// Идентификатор родительского процесса
   LONG    pcPriClassBase;     // Базовый приоритет процесса
   DWORD   dwFlags;            // Различные флаги
   CHAR    szExeFile[MAX_PATH];// Имя исполняемого файла
} PROCESSENTRY32;
```

CreateToolhelp32Snapshot

```
HANDLE CreateToolhelp32Snapshot(
    DWORD dwFlags,     // Флаги для указывающие, что именно нужно включить в снимок
    DWORD th32ProcessID // Идентификатор процесса (если применимо)
);
```

создает снимок всех процессов, потоков, модулей и heap, используемых в системе.

Process32First

```
BOOL Process32First(
   HANDLE hSnapshot,        // Дескриптор снимка, полученный от
CreateToolhelp32Snapshot
   LPPROCESSENTRY32 lppe    // Указатель на структуру PROCESSENTRY32
);
```

извлекает информацию о первом процессе в указанном снимке.

Process32Next

```
BOOL Process32Next(
   HANDLE hSnapshot,        // Дескриптор снимка
   LPPROCESSENTRY32 lppe    // Указатель на структуру PROCESSENTRY32
);
```

извлекает информацию о следующем процессе в указанном снимке.

Общий пример кода;

```
void ListProcesses() {
    HANDLE hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (hSnapshot == INVALID_HANDLE_VALUE) {
        std::cerr << "Failed to create snapshot" << std::endl;
        return;
    }
    PROCESSENTRY32 pe32;
    pe32.dwSize = sizeof(PROCESSENTRY32);
    if (!Process32First(hSnapshot, &pe32)) {
        std::cerr << "Failed to retrieve information about the first process" << std::endl;
        CloseHandle(hSnapshot);
        return;
    }
    do {
        std::cout << "Process ID: " << pe32.th32ProcessID << std::endl;
        std::cout << "Executable name: " << pe32.szExeFile << std::endl;
        std::cout << "Parent process ID: " << pe32.th32ParentProcessID << std::endl;
        std::cout << "Number of threads: " << pe32.cntThreads << std::endl;
        std::cout << std::endl;

    } while (Process32Next(hSnapshot, &pe32));
    CloseHandle(hSnapshot);
```

Поиск процесса в системе;

```
PROCESSENTRY32 pe32{ sizeof pe32 };
// pe32.dwSize = sizeof(pe32);

Process32First(process_app, &pe32);

do {
    if (std::string(pe32.szExeFile) == NAME_PROCESS)
        return pe32.th32ProcessID;

} while (Process32Next(process_app, &pe32));
```

cheat engine;

Cheat Engine - это такое ПО, которое позволяет управлять адресным пространством компьютера через корректировки адресов.

https://www.cheatengine.org/downloads.php

программка CASHTESTER:
https://disk.yandex.ru/d/gtvVs_I1XKls6w