Ульяновский государственный университет

Факультет математики, информационных и авиационных технологий

Кафедра Информационных технологий

Лабораторная работа по теме:

**«Сведения о системе»**.

Выполнил: Студент гр. ПРИ-0-22/1

Аксенов И.С.

Проверил: Волков М.А.

.

Ульяновск – 2022 г.

Целое число –12, вещественное число 12.25.

1.В соответствии с вариантом задания записать представление целого числа в типе *char* и вещественного числа в типе *float*.

*#include* <bitset>

*#include* <climits>

*#include* <iomanip>

*#include* <iostream>

*#include* <sstream>

*#include* <string>

*#define* BINARY\_SIZE 8

using namespace std;

char \*convert\_to\_binary(int *number*) {

  char \*char\_buffer = new char[BINARY\_SIZE]{

      '0', '0', '0', '0',

      '0', '0', '0', '0'}; *// Создаем в куче массив из 8 символов*

  int i = 0;

*while* (*number* >= 1) {

*if* (*number* % 2 == 1) {

      char\_buffer[i] = '1';

    } *else* {

      char\_buffer[i] = '0';

    }

*number* /= 2;

    i++;

  }

  char \*new\_char\_buffer =

      new char[BINARY\_SIZE]{'0', '0', '0', '0', '0', '0', '0', '0'};

*for* (int i = 0; i < BINARY\_SIZE; i++) {

    new\_char\_buffer[BINARY\_SIZE - 1 - i] = char\_buffer[i];

  }

  delete[] char\_buffer;

*return* new\_char\_buffer;

}

char \*convert\_to\_signed\_binary(int *number*) {

  char \*char\_buffer = convert\_to\_binary(abs(*number*));

*if* (*number* < 0) {

    char\_buffer[0] = '1';

  }

*return* char\_buffer;

}

bitset<sizeof(float) \* CHAR\_BIT> bits\_float\_binary(float *number*) {

*return* bitset<sizeof(float) \* CHAR\_BIT>(*number*);

}

int main() {

  int number = -12;

  float f\_number = 12.5;

  char \*p\_number\_bin = convert\_to\_binary(abs(number));

  cout << "Number converted to binary: ";

*for* (int i = 0; i < BINARY\_SIZE; i++) {

    cout << \*(p\_number\_bin + i);

  }

  cout << endl;

  char \*p\_number\_bin\_signed = convert\_to\_signed\_binary(number);

  cout << "Number converted to signed binary: ";

*for* (int i = 0; i < BINARY\_SIZE; i++) {

    cout << \*(p\_number\_bin\_signed + i);

  }

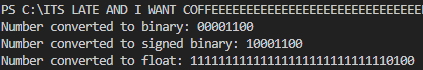
  cout << endl;

  cout << "Number converted to float: " << bits\_float\_binary(number);

  delete[] p\_number\_bin\_signed;

  delete[] p\_number\_bin;

}

**

2.С помощью функций WinAPI определить информацию об оперативной памяти.

С помощью функций WinAPI определить информацию о памяти на одном из жёстких дисков.

*#include* <vcruntime.h>

*#include* <windows.h>

*#include* <iostream>

double getDiskFreeSpacePercentage()

{

    DWORD lpSectorsPerCluster,

        lpBytesPerSector,

        lpNumberOfFreeClusters,

        lpTotalNumberOfClusters;

*if* (GetDiskFreeSpace(NULL,

        &lpSectorsPerCluster,

        &lpBytesPerSector,

        &lpNumberOfFreeClusters,

        &lpTotalNumberOfClusters))

    {

*return* double(lpNumberOfFreeClusters) \* double(lpSectorsPerCluster) \* double(lpBytesPerSector);

    }

*else*

    {

*return* 0;

    }

}

int main(int *argc*, char \**argv*[])

{

    using namespace std;

    setlocale(LC\_ALL, ".1251");

    MEMORYSTATUSEX statex;

    statex.dwLength = sizeof (statex);

    GlobalMemoryStatusEx (&statex);

    cout << "RAM " << statex.ullTotalPhys / (1024 \* 1024) << " MB's\n";

    cout << "HDD " << getDiskFreeSpacePercentage()  / (1024 \* 1024)<< " MB's\n";

*return* 0;

}



С помощью инструкции cpuid определить название процессора.

*// InstructionSet.cpp*

*// Compile by using: cl /EHsc /W4 InstructionSet.cpp*

*// processor: x86, x64*

*// Uses the \_\_cpuid intrinsic to get information about*

*// CPU extended instruction set support.*

*#include* <iostream>

*#include* <vector>

*#include* <bitset>

*#include* <array>

*#include* <string>

*#include* <intrin.h>

class InstructionSet

{

*// forward declarations*

    class InstructionSet\_Internal;

public:

*// getters*

    static std::string Vendor(void) { *return* CPU\_Rep.vendor\_; }

    static std::string Brand(void) { *return* CPU\_Rep.brand\_; }

private:

    static const InstructionSet\_Internal CPU\_Rep;

    class InstructionSet\_Internal

    {

    public:

        InstructionSet\_Internal()

            : nIds\_{ 0 },

            nExIds\_{ 0 },

            isIntel\_{ false },

            isAMD\_{ false },

            f\_1\_ECX\_{ 0 },

            f\_1\_EDX\_{ 0 },

            f\_7\_EBX\_{ 0 },

            f\_7\_ECX\_{ 0 },

            f\_81\_ECX\_{ 0 },

            f\_81\_EDX\_{ 0 },

            data\_{},

            extdata\_{}

        {

*//int cpuInfo[4] = {-1};*

            std::array<int, 4> cpui;

*// Calling \_\_cpuid with 0x0 as the function\_id argument*

*// gets the number of the highest valid function ID.*

            \_\_cpuid(cpui.data(), 0);

            nIds\_ = cpui[0];

*for* (int i = 0; i <= nIds\_; ++i)

            {

                \_\_cpuidex(cpui.data(), i, 0);

                data\_.push\_back(cpui);

            }

*// Capture vendor string*

            char vendor[0x20];

            memset(vendor, 0, sizeof(vendor));

            \*reinterpret\_cast<int\*>(vendor) = data\_[0][1];

            \*reinterpret\_cast<int\*>(vendor + 4) = data\_[0][3];

            \*reinterpret\_cast<int\*>(vendor + 8) = data\_[0][2];

            vendor\_ = vendor;

*if* (vendor\_ == "GenuineIntel")

            {

                isIntel\_ = true;

            }

*else* *if* (vendor\_ == "AuthenticAMD")

            {

                isAMD\_ = true;

            }

*// load bitset with flags for function 0x00000001*

*if* (nIds\_ >= 1)

            {

                f\_1\_ECX\_ = data\_[1][2];

                f\_1\_EDX\_ = data\_[1][3];

            }

*// load bitset with flags for function 0x00000007*

*if* (nIds\_ >= 7)

            {

                f\_7\_EBX\_ = data\_[7][1];

                f\_7\_ECX\_ = data\_[7][2];

            }

*// Calling \_\_cpuid with 0x80000000 as the function\_id argument*

*// gets the number of the highest valid extended ID.*

            \_\_cpuid(cpui.data(), 0x80000000);

            nExIds\_ = cpui[0];

            char brand[0x40];

            memset(brand, 0, sizeof(brand));

*for* (int i = 0x80000000; i <= nExIds\_; ++i)

            {

                \_\_cpuidex(cpui.data(), i, 0);

                extdata\_.push\_back(cpui);

            }

*// load bitset with flags for function 0x80000001*

*if* (nExIds\_ >= 0x80000001)

            {

                f\_81\_ECX\_ = extdata\_[1][2];

                f\_81\_EDX\_ = extdata\_[1][3];

            }

*// Interpret CPU brand string if reported*

*if* (nExIds\_ >= 0x80000004)

            {

                memcpy(brand, extdata\_[2].data(), sizeof(cpui));

                memcpy(brand + 16, extdata\_[3].data(), sizeof(cpui));

                memcpy(brand + 32, extdata\_[4].data(), sizeof(cpui));

                brand\_ = brand;

            }

        };

        int nIds\_;

        int nExIds\_;

        std::string vendor\_;

        std::string brand\_;

        bool isIntel\_;

        bool isAMD\_;

        std::bitset<32> f\_1\_ECX\_;

        std::bitset<32> f\_1\_EDX\_;

        std::bitset<32> f\_7\_EBX\_;

        std::bitset<32> f\_7\_ECX\_;

        std::bitset<32> f\_81\_ECX\_;

        std::bitset<32> f\_81\_EDX\_;

        std::vector<std::array<int, 4>> data\_;

        std::vector<std::array<int, 4>> extdata\_;

    };

};

*// Initialize static member data*

const InstructionSet::InstructionSet\_Internal InstructionSet::CPU\_Rep;

*// Print out supported instruction set extensions*

int main()

{

    std::cout << InstructionSet::Vendor() << std::endl;

    std::cout << InstructionSet::Brand() << std::endl;

}

