Лабораторная работа 1.

Создание консольного приложения Изучение основ C++.

Перегрузка функций и шаблонные функции.

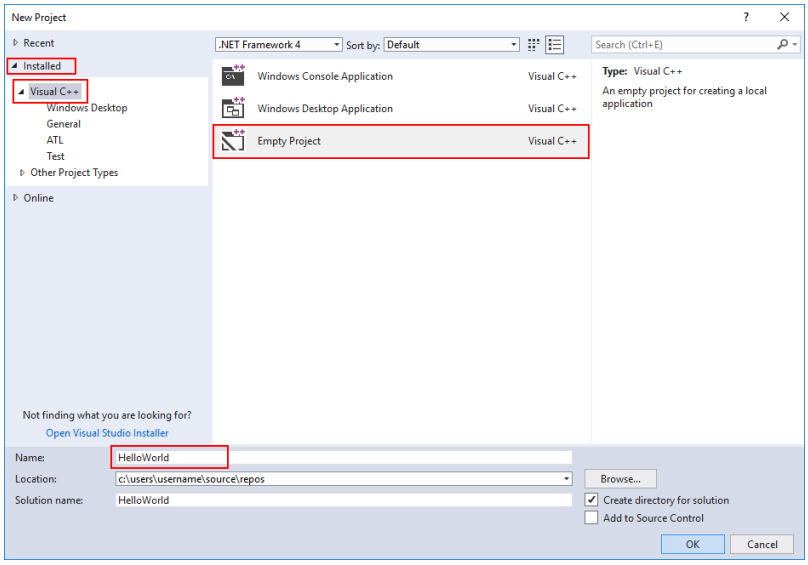
1. Создание консольного приложения

Visual Studio использует *проекты*, чтобы упорядочить код для приложения, и *решения*, чтобы упорядочить проекты. Проект содержит все параметры, конфигурации и правила, используемые для сборки приложения. Кроме того, он управляет связью между всеми файлами проекта и любыми внешними файлами. Чтобы создать приложение, сначала создайте проект и решение.

Создать консольное приложение можно двумя способами:

- Создание пустого проекта и написание кода с нуля.
- Создание консольное приложение с помощью мастера. В этом случае мастер создаст все необходимое файлы и заполнит их шаблонами кода.

1. в Visual Studio откройте меню **файл** и выберите пункт **создать Project** , чтобы открыть диалоговое окно **создание Project** .
2. в диалоговом окне **создание Project** выберите **установленный Visual C++** , если он еще не выбран, а затем выберите **пустой шаблон Project** . В поле **Имя** введите *HelloWorld*. Нажмите кнопку **OK**, чтобы создать проект.



Visual Studio создаст пустой проект. Вы можете приступать к его настройке в соответствии с типом создаваемого приложения и добавлению файлов исходного кода.

Visual Studio позволяет создавать приложения и компоненты самых разных типов как для

Windows, так и для других платформ. Шаблон **Пустой проект** не определяет тип создаваемого приложения. *Консольное приложение* — это приложение, которое выполняется в консоли или в окне командной строки. Чтобы создать его, необходимо сообщить Visual Studio, что приложение будет использовать подсистему консоли.

3. В Visual Studio в меню **Проект** выберите пункт **Свойства**, чтобы открыть диалоговое окно **Страницы свойств HelloWorld**.
4. В диалоговом окне **страницы свойств** выберите пункт **Свойства конфигурации система компоновщика**, а затем выберите поле редактирования рядом со свойством **подсистемы** . В появившемся раскрывающемся меню выберите пункт **Консоль (/SUBSYSTEM:CONSOLE)** . Выберите **OK** для сохранения внесенных изменений.
5. В обозревателе решений выберите проект HelloWorld. В меню Проект выберите команду Добавить новый элемент, чтобы открыть диалоговое окно Добавление нового элемента.
6. В диалоговом окне Добавление нового элемента выберите вариант Visual C++ в поле Установленные, если он еще не выбран. В центральной области выберите Файл C++ (.cpp) . Измените имя на HelloWorld.cpp. Нажмите кнопку Добавить, чтобы закрыть диалоговое окно и создать файл.

Visual Studio создаст пустой файл исходного кода и откроет его в окне редактора, где в него можно ввести код

Для сборки проекта выберите в меню **Сборка** пункт **Собрать решение**. Окно **Вывод** отображает результаты процесса сборки.

Чтобы создать консольное приложение с помощью мастера в меню **Файл** выбираем пункт **Создать | Проект**. В открывшемся окне выделяем пункт **Консольное приложение Windows**. Вводим название проекта (например, *welcome*) в поле **Имя**. Введенное название автоматически копируется в поле **Имя решения**. В поле **Расположение** указываем путь к каталогу, в котором будет сохранен проект, и устанавливаем флажок **Создать каталог для решения**. Нажимаем кнопку **OK**. В результате откроется окно **Мастер приложений**. Нажимаем кнопку **Далее**. В группе переключателей **Тип приложения** выбираем **Консольное приложение**. Устанавливаем флажок **Предварительно скомпилированный заголовок**. Флажок **Пустой проект** должен быть снят. Нажимаем кнопку **Готово**.

В результате будут созданы следующие файлы:

welcome.cpp — основной файл проекта;

stdafx.h — файл для подключения заголовочных файлов, используемых в проекте. Этот файл необходимо подключать во всех файлах проекта;

stdafx.cpp — файлы stdafx.h и stdafx.cpp используются для построения файла предкомпилированного заголовка (welcome.pch) и файла предкомпилированных типов (stdafx.obj);

targetver.h — в этом файле подкючается файл SDKDDKVer.h, который обеспечивает определение последней доступной платформы Windows.

Файлы stdafx.cpp и targetver.h оставляем без изменений. В конец файла stdafx.h вставляем строку, позволяющую использовать в программе объект **cout**, предназначенный для вывода данных в окно консоли:

#include <iostream> Содержимое файла stdafx.h :

#pragma once #include "targetver.h" #include <stdio.h> #include <tchar.h> #include <iostream>

Директива **pragma** со значением **once** указывает, что содержимое файла может быть включено только один раз. Директивы **include** подключают файлы, в которых содержатся объявления идентификаторов. Такие файлы называются *заголовочными файлами*. Они содержат только объявление идентификаторов без описания их реализации. Обратите внимание на то, что в директиве **include** заголовочные файлы подключаются разными способами. Поиск файлов, указанных внутри угловых скобок, производится в системных каталогах, а поиск файлов, указанных внутри кавычек, производится в каталоге проекта. Таким образом, названия файлов стандартной библиотеки необходимо указывать внутри угловых скобок, а пользовательских файлов внутри кавычек.

Наличие расширения файла принято в стандартной библиотеке языка C. В стандартной библиотеке языка C++ расширение файла принято не указывать. Так как язык C++ наследует все библиотеки языка C, то файлы можно подключать как в стиле языка C, так и в стиле языка C++. Например, файл string.h из стандартной библиотеки языка C доступен в языке C++ под названием cstring, а файл math.h под названием cmath. Отличие между этими способами подключения заключается в импортировании идентификаторов. В языке C при подключении файла (например, math.h) все идентификаторы импортируются в глобальное пространство имен, а в языке C++ при подключении файла (например, cmath) идентификаторы определяются в пространстве

имен под названием **std**. Поэтому перед идентификатором необходимо указать название пространства имен (например, **std::cout**). Использование пространств имен позволяет избежать конфликта имен в программе.

Чтобы посмотреть содержимое заголовочного файла (например, stdio.h) щелкните правой кнопкой мыши на строке "#include <stdio.h>" и из констекстного меню выберите пункт **Открыть документ <stdio.h>**. В результате заголовочный файл stdio.h будет открыт на отдельной вкладке.

Скомпилировать и запустить программу на исполнение можно также как и в предыдущем разделе.

Для ввода данных в языке C++ предназначен объект **cin** (console input — ввод с консоли), объявленный в файле iostream. Объект **cin** позволяет ввести данные любого встроенного типа, например, число, символ или строку.

```
#include <iostream> int main() {
char name[20];
std::cout << "vvedite name\n"; std::cin>>name;
std::cout << "Hello, world!                " << name<<  std::endl; return 0;
}
```

2. **Основы языка Константы**.

Возможно использование трех типов цифровых констант: десятичные : 80, 9, 1000000; восьмеричные: 060, 07, 02345; шестнадцатиричные: 0x80, 0xAFC **Типы данных**

Любая переменная или константа перед использованием должна быть объявлена с помощью оператора вида

[<спецификатор класса памяти>] [const] <спецификатор типа> <идентификатор>[=<начальное значение>]

3. **Управляющие структуры языка**

- if (*условие)*

оператор;

else

оператор2;

Проверка числа на четность

```
#include <iostream> #include <clocale> int main() {
int x = 0; char c; c ='n';
```

```
 std::setlocale(LC_ALL, "Russian_Russia.1251"); do{
std::cout << "Введите число: "; std::cin>> x;
if (!std::cin.good()) {
std::cout << std::endl << "Вы ввели не число" << std::endl; std::cin.clear();
// Сбрасываем флаг ошибки std::cin.ignore(255, '\n').get();
}
else {
if (x % 2 == 0)
  std::cout << x << " - четное число" << std::endl; else
std::cout << x << " - нечетное число" << std::endl;
}
std::cout << "завершить?"; std::cin>>c;
}
while((c!='y')&&(c!='Y')); return 0;
}
```

- switch (*выражение*)

```
{         case константа_1 : jператоры; break;
        case константа_2 :  операторы; break;
          case константа_m : операторы
}
char Answer = '';
cout « "Продолжить работу? ";
cin » Answer;
switch(Answer)
{
case 'Y':
case 'y':
case 'д':
case 'Д':
          cout « "Продолжим...\n"; break;
default:
```

cout « "Завершение...\n";

}

- while (*выражение*)// выход если выражение ложно (равно 0)

Оператор;

while (1) //бесконечный цикл

- do

оператор

while (*условие*);

- Цикл for более гибкий чем в других языках программирования for

([*выражение1*];[*выражение2*];[*выражение3*])

оператор;

Выражение 1 чаще всего служит в качестве инициализации какой-нибудь переменной, выполняющей роль счетчика итераций.

Выражение 2 используется как проверочное условие и на практике часто содержит выражения с операторами сравнения. По умолчанию принимает истинное значение.

Выражение 3 служит чаще всего для приращения значения счетчика циклов либо содержит выражение, влияющее, каким бы то ни было образом, на проверочное условие.

for(; ;) //Бесконечный цикл. for (i=1,r=1;i<=10;i++,r*=y)//$r=y^i$

Суммирование неопределенного количества чисел

```
#include <iostream> #include <clocale> int main() {
int x = 0, summa = 0;
std::setlocale(LC_ALL, "Russian_Russia.1251");
std::cout << "Введите число 0 для получения результата"
        << std::endl; for ( ; ; ) {
std::cout << "Введите число: "; std::cin >> x;
if (!std::cin.good()) {
std::cout << "Вы ввели не число!" << std::endl; std::cin.clear(); // Сбрасываем флаг ошибки
std::cin.ignore(255, '\n');
continue;
}
if (!x) break; summa += x;
}
```

```
std::cout << "Сумма чисел равна: " << summa << std::endl; std::cin.ignore(255, '\n').get();
return 0;
}
```

Функция — это фрагмент кода, который можно неоднократно вызвать из любого места программы, они позволяют уменьшить избыточность программного кода и повысить его структурированность.

Описание функции состоит из двух частей: ***объявления и определения***. ***Объявление функции*** (называемое также *прототипом функции*) содержит информацию о типе. Используя эту информацию компилятор может найти несоответствие типа и количества параметров. Формат прототипа функции:

<Тип рез> <Назв функции>([<Тип> [<Назв парам1>] [, ..., <Тип> [<Назв парамN>]]]);

Параметр **<Тип результата>** задает тип значения, которое возвращает функция с помощью оператора **return**. Если функция не возвращает никакого значения, то вместо типа указывается ключевое слово **void**. Название функции должно быть допустимым идентификатором. После названия функции, внутри круглых скобок, указывается тип и название параметров через запятую. Названия параметров в прототипе функции можно не задавать вообще, так как компилятор интересует только тип данных и количество параметров. Если функция не принимает параметров, то указываются только круглые скобки Пример объявления функций:

Определение функции содержит описание типа и названия параметров, а также реализацию. Определение функции имеет следующий формат:

Тип рез> <Назв функции>([<Тип> [<Назв парам1>] [, ..., <Тип> [<Назв парамN>]]])
{ <Тело функции>
[return[<Возвращаемое значение>];
}

В отличие от прототипа в определении функции после типа обязательно должно быть указано название параметра, которое является локальной переменной. Эта переменная создается при вызове функции, а после выхода из функции она удаляется.

В качестве примера создадим три разные функции и вызовем их .

```
#include <iostream>
```

```
// Объявления функций int sum(int , int );
void print(const char *);
void print_ok();              // или void print_ok(void); int main() {
// Вызов функций
print("Message");                    // Message print_ok();        // OK
std::cout<<"x= "; std::cin>>x; std::cout<<" y= "; std::cin>>y;
      std::cout << sum(10, 20) << std::endl; // 30 std::cin.get();
return 0;
}
// Определения функций
int sum(int x, int y) {              return x + y;}
void print(const char *str) {    std::cout << str << std::endl;} void print_ok() {    std::cout << "OK"
<< std::endl;}
```

При увеличении размера программы ее можно разделить на несколько отдельных файлов. Объявления функций выносят в заголовочный файл с расширением h, а определения функций размещают в одноименном файле с расширением cpp. Все файлы располагают в одной папке с основным файлом, содержащим функцию **main()**. В дальнейшем с помощью директивы **#include** подключают заголовочный файл во всех остальных файлах. *Если в директиве **#include** название заголовочного файла указывается внутри угловых скобок, то поиск файла осуществляется в системных папках. Если название указано внутри кавычек, то поиск вначале производится в папке с основным файлом, а затем в системных папках.*

Способы передачи параметров в функцию

- Передача параметров по значению

По умолчанию в функцию передается копия значения переменной, а не ссылка на переменную. Таким образом, изменение значения внутри функции не затронет значение исходной переменной.

```
#include <iostream> void func(int x);
int main() { int x = 10; func(x);
std::cout << x << std::endl; // 10, а не 30 std::cin.get();
return 0;
}
```

```
void func(int x) {
x = x + 20; // Значение нигде не сохраняется!
}
```

- Передача параметров через указатель

При использовании массивов и объектов, а также при необходимости изменить значение исходной переменной, лучше применять передачу указателя. Для этого при вызове функции перед названием переменной указывается оператор **&** (взятие адреса), а в прототипе функции объявляется указатель. В этом случае в функцию передается не значение переменной, а ее адрес. Внутри функции вместо переменной используется указатель.

```
#include <iostream> void func(int *y);
int main() { int x = 10;
func(&x);                    // Передаем адрес std::cout << x << std::endl; // 30, а не 10
std::cin.get();
return 0;
}
void func(int *y) {
*y = *y + 20; // Изменяется значение переменной x
}
```

- Передача параметров по ссылке

В языке C++ существует еще один способ передачи параметров — механизм ссылок. Внутри функции переменная является псевдонимом исходной переменной. Любые изменения псевдонима отражаются на исходной переменной. При использовании механизма ссылок перед названием переменной в объявлении и определении функции указывается оператор **&**, а при вызове — только название переменной.

```
#include <iostream> void func(int &y); int main() {
int x = 10; func(x);
std::cout << x << std::endl; // 30, а не 10 std::cin.get();
return 0;
}
void func(int &y) {
```

// Переменная y является псевдонимом переменной x y = y + 20; // Изменяется значение переменной x

}

- Передача массивов в функцию

Передача одномерных массивов осуществляется с помощью указателей. При вызове функции перед названием переменной не нужно добавлять оператор **&**, так как название переменной содержит адрес первого элемента массива.

```
#include <iostream> void func1(char *s); void func2(char s[]); int main() {
char str[] = "String";
std::cout << sizeof(str) << std::endl; // 7
func1(str); // Оператор & перед str не указывается!!! func2(str);
std::cout << str << std::endl; // stringG return 0;
}
void func1(char *s) {
s[0] = 's'; // Изменяется значение элемента массива str std::cout << sizeof(s) << std::endl;
// 4, а не 7!!!
}
void func2(char s[]) {
s[5] = 'G'; // Изменяется значение элемента массива str
}
```

Объявление **char *s** эквивалентно объявлению **char s[]**. И в том и в другом случае объявляется указатель на тип **char**. Обратите внимание на значения, возвращаемые оператором **sizeof** вне и внутри функции. Вне функции оператор возвращает размер всего символьного массива, в то время как внутри функции оператор **sizeof** возвращает только размер указателя. Это происходит потому что внутри функции переменная **s** является указателем, а не массивом. Поэтому если внутри функции необходимо знать размер массива, то количество элементов (или размер в байтах) следует передавать в дополнительном параметре.

При передаче многомерного массива необходимо явным образом указывать все размеры (например, **int a[4][4][4]**). Самый первый размер допускается не указывать (например, **int a[][4][4]**).

```
#include <iostream> void func(int a[][4]); int main() {
```

```
const short ROWS = 2, COLS = 4; int i, j;
int arr[ROWS][COLS] = {                    {1, 2, 3, 4},      {5, 6, 7, 8}   };
func(arr);
// Выводим значения for (i=0; i<ROWS; ++i) { for (j=0; j<COLS; ++j) {
std::cout.width(4);                        // Ширина поля std::cout << arr[i][j];
}
std::cout << std::endl;
}
std::cin.get(); return 0;
}
void func(int a[][4]) { // или void func(int a[2][4])
a[0][0] = 80;
}
```

Такой способ передачи многомерного массива нельзя назвать универсальным, так как существует жесткая привязка к размеру. Одним из способов решения проблемы является создание дополнительного массива указателей. В этом случае в функцию передается адрес первого элемента массива указателей, а объявление параметра в функции выглядит так:

```
int *a[] или так:int **a #include <iostream>
void func(int *a[], short rows, short cols); int main() {
const short ROWS = 2, COLS = 4;
int arr[ROWS][COLS] = {                    {1, 2, 3, 4},     {5, 6, 7, 8}   };
// Создаем массив указателей int *parr[] = {arr[0], arr[1]};
// Передаем адрес массива указателей func(parr, ROWS, COLS);
return 0;
}
void func(int *a[], short rows, short cols) {
// или void func(int **a, short rows, short cols) int i, j;
                                        for (i=0; i<rows; ++i) { for (j=0; j<cols; ++j) { std::cout.width(4);
                                                                                  std::cout << a[i][j];
}
std::cout << std::endl;
```

```
}
}
```

Однако и этот способ имеет недостаток, так как нужно создавать дополнительный массив указателей. Наиболее приемлемым способом является передача многомерного массива как одномерного. В этом случае в функцию передается адрес первого элемента массива, а в параметре объявляется указатель. Так как все элементы многомерного массива располагаются в памяти последовательно, зная количество элементов или размеры можно вычислить положение текущего элемента, используя адресную арифметику.

```
#include <iostream>
void func(int *a, short rows, short cols); int main() {
const short ROWS = 2, COLS = 4;
int arr[ROWS][COLS] = {                {1, 2, 3, 4},      {5, 6, 7, 8}   };
// Передаем в функцию адрес первого элемента массива func(arr[0], ROWS, COLS);
return 0;
}
void func(int *a, short rows, short cols)
{ int i, j;
for (i=0; i<rows; ++i) { for (j=0; j<cols; ++j)
  {       std::cout.width(2);
          std::cout << a[i * cols + j]; // Вычисляем положение элемента
}
std::cout << std::endl;
}
}
```

4. Перегрузка функций

Перегрузка функции — это возможность использования одного названия для нескольких функций, различающихся типом параметров или их количеством. В качестве примера перегрузим функцию **sum()** таким образом, чтобы ее название можно было использовать для суммирования как целых чисел, так и вещественных.

```
#include <iostream> int sum(int x, int y);
double sum(double x, double y); int main() {
```

```
// Суммирование целых чисел
std::cout << sum(10, 20) << std::endl;                    // 30
// Суммирование вещественных чисел std::cout << sum(10.5, 20.7) << std::endl; // 31.2
return 0;
}
int sum(int x, int y) { return x + y;
}
double sum(double x, double y) { return x + y;
}
```

5. Шаблонные (обобщенные) функции

Если присмотреться к реализации функции **sum()**, то можно заметить, что вне зависимости от типа параметров внутри функции будет одна и та же инструкция. Для таких случаев в языке C++ вместо перегрузки функции следует применять *шаблонные функции*. Компилятор на основе шаблонной функции автоматически создаст перегруженные версии функции в зависимости от имеющихся способов ее вызова в программе. Описывается шаблонная функция по следующей схеме:

template<class Tun1[, ..., class TunN]>
Тип Название_функции(Тип Назв_перем1 [, ..., Тип Назв_перемN])
{
Иструкции;
}

После ключевого слова **template** внутри угловых скобок через запятую указываются обобщенные названия типов. Эти названия используются для описания типов параметров и могут использоваться внутри функции. При компиляции обобщенные типы будут заменены реальными типами данных. Перед названием обобщенного типа могут быть указаны ключевые слова **class** или **typename**, которые обозначают одно и то же. Остальное описание шаблонной функции совпадает с описанием обычной функции, только вместо реальных типов данных указываются названия обобщенных типов, перечисленных после ключевого слова **template**.

```
#include <iostream> template<class Type> Type sum(Type x, Type y); int main() {
std::cout << sum(10, 20) << std::endl;                    // 30 std::cout << sum(10.5, 20.4) <<
std::endl;  // 30.9 std::cout << sum(10.5f, 20.7f) << std::endl; // 31.2 std::cin.get();
```

```
return 0;
}
template<class Type>
Type sum(Type x, Type y) { return x + y;
}
```

Компилятор на основе шаблонной функции и способах ее вызова автоматически создаст следующие перегруженные версии функции:

```
int sum(int x, int y);
float sum(float x, float y);  double sum(double x, double y);
```