

КУРС ЛЕКЦИЙ ПО ДИСЦИПЛИНЕ «ПРОЦЕДУРНОЕ ПРОГРАММИРОВАНИЕ»

Рекомендуемая литература

1. Романов Е.Л. Си++. От дилетанта до профессионала. М., 2014. 600 с.
2. Учебник по С++ для начинающих[Электронный ресурс]<http://www.programmersclub.ru/main>
3. Литвиненко Н. А. Технология программирования на С++[Электронный ресурс]http://www.proklondike.com/books/cpp/technology_of_programming_on_cplusplus.html
4. Стефан Р. Дэвис. С++ Для чайников[Электронный ресурс]http://www.proklondike.com/books/cpp/cplusplus_dlya_chaynikov.html
Примечание. В НТБ МИРЭА нет книг по С++, изданных в последние 5 лет.

Тема 1. Простая программа на С++

В качестве среды разработки под Windows будем использовать Visual Studio 2013 (6.5 Гб) для Windows 7 и выше.

После того, как вы установите все необходимое, приступим к написанию первой программы.

Создание проекта

Откройте меню «Файл → Создать → Проект». Перейдите на вкладку «Общие» и выберите «Пустой проект». Придумайте проекту любое название, например, «program1» и нажмите «ОК».

В окне обозревателя решений (обычно он находится в левом верхнем углу) щелкните правой кнопкой на папке «файлы исходного кода». В диалоговом окне выберите пункт меню «Добавить → Создать элемент». Введите название для нового файла — main.cpp и нажмите кнопку «Добавить».

Код первой программы

Наберите следующий код:

```
#include<iostream>
#include<cstdlib> // для system
using namespace std;

int main()
{
    cout<<"Hello, world!"<<endl;
    system("pause"); // Только для тех, у кого MS Visual Studio
    return 0;
```

}

Описание синтаксиса

Директива `#include` используется для подключения других файлов в код. Строка `#include <iostream>`, будет заменена содержимым файла «`iostream.h`», который находится в стандартной библиотеке языка и отвечает за ввод и вывод данных на экран.

`#include <cstdlib>` подключает стандартную библиотеку языка C. Это подключение необходимо для работы функции `system`.

Содержимое третьей строки — `using namespace std;` указывает на то, что мы используем по умолчанию пространство имен с названием «`std`». Все то, что находится внутри фигурных скобок функции `int main() { }` будет автоматически выполняться после запуска программы.

Строка `cout << "Hello, world!" << endl;` говорит программе выводить сообщение с текстом «**Hello, world**» на экран.

Оператор `cout` предназначен для вывода текста на экран командной строки. После него ставятся две угловые кавычки (`<<`). Далее идет текст, который должен выводиться. Он помещается в двойные кавычки. Оператор `endl` переводит строку на уровень ниже.

Если в процессе выполнения произойдет какой-либо сбой, то будет сгенерирован код ошибки, отличный от нуля. Если же работа программы завершилась без сбоев, то код ошибки будет равен нулю. Команда `return 0` необходима для того, чтобы передать операционной системе сообщение об удачном завершении программы.

— В конце каждой команды ставится **точка с запятой**.

Компиляция и запуск

Теперь скомпилируйте и запустите программу: нужно нажать сочетание клавиш «`Ctrl+F5`».

Если программа собралась с первого раза, то хорошо. Если компилятор говорит о наличии ошибок, значит вы что-то сделали неправильно.

Прочитайте текст ошибки и попробуйте ее исправить своими силами. Если не получится, обратитесь к преподавателю.

В качестве домашнего задания, переделайте эту программу так, чтобы вместо, сообщения «`Hello, World`» выводилось сообщение «`Hello, User`».

Тема 2. Переменные и типы данных в C++

Из школьного курса математики мы все знаем, что такое переменные. В программировании принципы довольно схожи. **Переменная** — это «ячейка» оперативной памяти компьютера, в которой может храниться какая-либо информация.

В программировании переменная, как и в математике, может иметь название, состоящее из одной латинской буквы, но также может состоять из нескольких символов, целого слова или нескольких слов.

Типы данных

В языке C++ *все переменные* имеют определенный тип данных. Например, переменная, имеющая целочисленный тип, не может содержать ничего кроме целых чисел, а переменная с плавающей точкой — только дробные числа.

Тип данных присваивается переменной при ее объявлении или инициализации. Ниже приведены основные типы данных языка C++, которые нам понадобятся.

Основные типы данных в C++

- **int** — целочисленный тип данных.
- **float** — тип данных с плавающей запятой.
- **double** — тип данных с плавающей запятой двойной точности.
- **char** — символьный тип данных.
- **bool** — логический тип данных.

Объявление переменной

Объявление переменной в C++ происходит таким образом: сначала указывается тип данных для этой переменной а затем название этой переменной.

Пример объявления переменных

```
int a; // объявление переменной a целого типа.  
float b; // объявление переменной b типа данных с плавающей запятой.  
double c = 14.2; // инициализация переменной типа double.  
char d = 's'; // инициализация переменной типа char.  
bool k = true; // инициализация логической переменной k.
```

- Заметьте, что в C++ **оператор присваивания** (=) — не является знаком равенства и не может использоваться для сравнения значений. Оператор равенства записывается как «двойное равно» — ==.
- Присваивание используется для сохранения определенного значения в переменной. Например, запись вида `a = 10` задает переменной `a` значение числа 10.

Простой калькулятор на C++

Сейчас мы напишем простую программу-калькулятор, которая будет принимать от пользователя два целых числа, а затем определять их сумму:

```
#include<iostream>  
using namespace std;  
  
int main()  
{  
    setlocale(0, "");  
    /*7*/ int a, b; // объявление двух переменных a и b целого типа данных.  
    cout<<"Введите первое число: ";  
    cin>> a; // пользователь присваивает переменной a какое-либо значение.  
    cout<<"Введите второе число: ";  
    cin>> b;  
    /*12*/ int c = a + b; // новой переменной c присваиваем значение суммы введенных пользователем  
    данных.  
    cout<<"Сумма чисел = "<< c << endl; // вывод ответа.
```

```
return 0;
}
```

Разбор кода

В 7-й строке кода программы мы объявляем переменные «a» и «b» целого типа `int`. В следующей строке кода выводится сообщение пользователю, чтобы он ввел с клавиатуры первое число.

В 9-й строке стоит еще незнакомая вам конструкция — `cin >>`. С помощью нее у пользователя запрашивается ввод значения переменной «a» с клавиатуры. Аналогичным образом задается значение переменной «b».

В 12-й строке мы производим инициализацию переменной «c» суммой переменных «a» и «b». Далее находится уже знакомый вам оператор `cout`, который выводит на экран строку и значение переменной «c».

- При выводе переменных, они *не заключаются в кавычки*, в отличие от строк.

Домашнее задание

Попробуйте провести несколько экспериментов с программой — сделайте аналогичный пример с умножением или вычитанием переменных. Не бойтесь издеваться над программным кодом, потому что ошибки — неотъемлемая часть обучения любому делу. И не забывайте про точки с запятой.

Тема 3. Конструкция ветвления в C++

Встречаются ситуации, когда программе нужно выбрать, какую операцию ей выполнить, в зависимости от определенного условия.

К примеру, мы вводим с клавиатуры целое число. Если это число больше десяти, то программа должна выполнить одно действие, иначе — другое. Реализуем этот алгоритм на C++ с помощью **конструкции ветвления**.

Оператор `if`

Оператор `if` служит для того, чтобы выполнить какую-либо операцию в том случае, когда условие является верным. *Условная конструкция в C++* всегда записывается в круглых скобках после оператора `if`.

Внутри фигурных скобок указывается тело условия. Если условие выполнится, то начнется выполнение всех команд, которые находятся между фигурными скобками.

Пример конструкции ветвления

```
#include<iostream>
using namespace std;
```

```
int main()
{
    setlocale(0, "");
    double num;
```

```
cout<<"Введите произвольное число: ";
cin>> num;

if (num <10) { // Если введенное число меньше 10.
cout<<"Это число меньше 10."<< endl;
} else{ // иначе
cout<<"Это число больше либо равно 10."<< endl;
}
return 0;
}
```

Если вы запустите эту программу, то при вводе числа, меньшего десяти, будет выводиться соответствующее сообщение.

Если введенное число окажется большим, либо равным десяти — отобразится другое сообщение.

```
if (num <10) { // Если введенное число меньше 10.
cout<<"Это число меньше 10."<< endl;
} else{ // иначе
cout<<"Это число больше либо равно 10."<< endl;
}
```

Здесь говорится: «**Если** переменная num меньше 10 — вывести соответствующее сообщение. **Иначе**, вывести другое сообщение».

Усовершенствуем программу так, чтобы она выводила сообщение, о том, что переменная num равна десяти:

```
if (num <10) { // Если введенное число меньше 10.
cout<<"Это число меньше 10."<< endl;
} elseif (num == 10) {
cout<<"Это число равно 10."<< endl;
} else{ // иначе
cout<<"Это число больше 10."<< endl;
}
```

Здесь мы проверяем три условия:

- Первое — когда введенное число меньше 10-ти
- Второе — когда число равно 10-ти
- И третье — когда число больше десяти

Заметьте, что во втором условии, при проверке равенства, мы используем оператор равенства — `==`, а не оператор присваивания, потому что мы не изменяем значение переменной при проверке, а сравниваем ее текущее значение с числом 10.

- Если поставить оператор присваивания в условии, то при проверке условия, значение переменной изменится, после чего это условие выполнится.

Каждому **оператору if** соответствует только один *оператор else*. Совокупность этих операторов — **else if** означает, что если не выполнилось предыдущее условие, то проверить данное. Если ни одно из условий не верно, то выполняется тело *оператора else*.

Если после оператора **if**, **else** или их связки **else if** должна выполняться только одна команда, то фигурные скобки можно не ставить. Предыдущую программу можно записать следующим образом:

```
#include<iostream>
usingnamespacestd;

intmain()
{
    setlocale(0, "");
    double num;

    cout<<"Введите произвольное число: ";
    cin>> num;

    if (num <10) // Если введенное число меньше 10.
        cout<<"Это число меньше 10."<< endl;
    elseif (num == 10)
        cout<<"Это число равно 10."<< endl;
    else// иначе
        cout<<"Это число больше 10."<< endl;

    return0;
}
```

Такой метод записи выглядит более компактно. Если при выполнении условия нам требуется выполнить более одной команды, то фигурные скобки необходимы. Например:

```
#include<iostream>
usingnamespacestd;

intmain()
{
    setlocale(0, "");
    double num;
    int k;

    cout<<"Введите произвольное число: ";
    cin>> num;

    if (num <10) { // Если введенное число меньше 10.
        cout<<"Это число меньше 10."<< endl;
        k = 1;
    } elseif (num == 10) {
        cout<<"Это число равно 10."<< endl;
        k = 2;
    } else{ // иначе
        cout<<"Это число больше 10."<< endl;
        k = 3;
    }

    cout<<"k = "<< k << endl;
    return0;
}
```

Данная программа проверяет значение переменной `num`. Если она меньше 10, то присваивает переменной `k` значение единицы. Если переменная `num` равна десяти, то

присваивает переменной k значение двойки. В противном случае — значение тройки. После выполнения ветвления, значение переменной k выводится на экран.

Хорошенько потренируйтесь, попробуйте придумать свой пример с ветвлением.

Тема 4. Циклы в C++

Иногда необходимо повторять одно и то же действие несколько раз подряд. Для этого используют циклы. В этом уроке мы научимся программировать **циклы на C++**, после чего посчитаем сумму всех чисел от 1 до 1000.

Цикл for

Если мы знаем точное количество действий (итераций) цикла, то можем использовать **цикл for**. Синтаксис его выглядит примерно так:

```
for (действие до начала цикла;  
    условие продолжения цикла;  
    действия в конце каждой итерации цикла) {  
    инструкция цикла;  
    инструкция цикла 2;  
    инструкция цикла N;  
}
```

Итерацией цикла называется один проход этого цикла

Существует частный случай этой записи, который мы сегодня и разберем:

```
for (счетчик = значение; счетчик < значение; шаг цикла) {  
    тело цикла;  
}
```

Счетчик цикла — это переменная, в которой хранится количество проходов данного цикла.

Описание синтаксиса

1. Сначала присваивается первоначальное значение счетчику, после чего ставится точка с запятой.
2. Затем задается конечное значение счетчика цикла. После того, как значение счетчика достигнет указанного предела, цикл завершится. Снова ставим точку с запятой.
3. Задаем шаг цикла. **Шаг цикла** — это значение, на которое будет увеличиваться или уменьшаться счетчик цикла при каждом проходе.

Пример кода

Напишем программу, которая будет считать сумму всех чисел от 1 до 1000.

```
#include<iostream>  
using namespace std;  
  
int main()  
{  
    int i; // счетчик цикла
```

```

int sum = 0; // сумма чисел от 1 до 1000.
setlocale(0, "");
for (i = 1; i <= 1000; i++) // задаем начальное значение 1, конечное 1000 и задаем шаг цикла - 1.
{
    sum = sum + i;
}
cout<<"Сумма чисел от 1 до 1000 = "<< sum << endl;
return0;
}

```

Если мы скомпилируем этот код и запустим программу, то она покажет нам ответ: 500500. Это и есть сумма всех целых чисел от 1 до 1000. Если считать это вручную, понадобится очень много времени и сил. Цикл выполнил всю рутинную работу за нас.

Заметьте, что конечное значение счетчика я задал нестрогим неравенством (\leq — меньше либо равно), поскольку, если бы я поставил знак меньше, то цикл произвел бы 999 итераций, т.е. на одну меньше, чем требуется. Это довольно важный момент, т.к. здесь новички часто допускают ошибки, особенно при работе с массивами (о них будет рассказано в следующем уроке). Значение шага цикла я задал равное единице. $i++$ — это тоже самое, что и $i = i + 1$.

В теле цикла, при каждом проходе программа увеличивает значение переменной `sum` на `i`. Еще один очень важный момент — в начале программы я присвоил переменной `sum` значение нуля. Если бы я этого не сделал, программа вылетела бы в сегфолт. При объявлении переменной без ее инициализации что эта переменная будет хранить «мусор».

Естественно к мусору мы ничего прибавить не можем. Некоторые компиляторы, такие как `gcc`, инициализирует переменную нулем при ее объявлении.

Цикл **while**

Когда мы не знаем, сколько итераций должен произвести цикл, нам понадобится цикл **while** или **do...while**. Синтаксис цикла **while** в C++ выглядит следующим образом.

```

while (Условие) {
    Тело цикла;
}

```

Данный цикл будет выполняться, пока условие, указанное в круглых скобках является истиной. Решим ту же задачу с помощью цикла **while**. Хотя здесь мы точно знаем, сколько итераций должен выполнить цикл, очень часто бывают ситуации, когда это значение неизвестно.

Ниже приведен исходный код программы, считающей сумму всех целых чисел от 1 до 1000.

```

#include<iostream>
using namespace std;

int main()
{
    setlocale(0, "");
    int i = 0; // инициализируем счетчик цикла.
    int sum = 0; // инициализируем счетчик суммы.
    while (i < 1000)

```



```

    {
    i++;
    sum += i;
    }
    cout<<"Сумма чисел от 1 до 1000 = "<< sum << endl;
    return 0;
}

```

После компиляции программа выдаст результат, аналогичный результату работы предыдущей программы. Но поясним несколько важных моментов. Я задал строгое неравенство в условии цикла и инициализировал счетчик `i` нулем, так как в цикле **while** происходит на одну итерацию больше, потому он будет выполняться, до тех пор, пока значение счетчика перестает удовлетворять условию, но данная итерация все равно выполнится. Если бы мы поставили нестрогое неравенство, то цикл бы закончился, когда переменная `i` стала бы равна 1001 и выполнилось бы на одну итерацию больше.

Теперь давайте рассмотрим по порядку исходный код нашей программы. Сначала мы инициализируем счетчик цикла и переменную, хранящую сумму чисел.

В данном случае мы обязательно должны присвоить счетчику цикла какое-либо значение, т.к. в предыдущей программе мы это значение присваивали внутри цикла **for**, здесь же, если мы не инициализируем счетчик цикла, то в него попадет «мусор» и компилятор в лучшем случае выдаст нам ошибку, а в худшем, если программа соберется — дефолт практически неизбежен.

Затем мы описываем условие цикла — **«пока переменная `i` меньше 1000 — выполняй цикл»**. При каждой итерации цикла значение переменной-счетчика `i` увеличивается на единицу внутри цикла.

Когда выполнится 1000 итераций цикла, счетчик станет равным 999 и следующая итерация уже не выполнится, поскольку 1000 не меньше 1000. Выражение `sum += i` является укороченной записью `sum = sum + i`.

После окончания выполнения цикла, выводим сообщение с ответом.

Цикл `do while`

Цикл **do while** очень похож на цикл **while**. Единственное их различие в том, что при выполнении цикла **do while** один проход цикла будет выполнен независимо от условия. Решение задачи на поиск суммы чисел от 1 до 1000, с применением цикла *do while*.

```

#include<iostream>
using namespace std;

int main()
{
    setlocale(0, "");
    int i = 0; // инициализируем счетчик цикла.
    int sum = 0; // инициализируем счетчик суммы.
    do { // выполняем цикл.
        i++;
        sum += i;
    } while (i < 1000); // пока выполняется условие.
    cout<<"Сумма чисел от 1 до 1000 = "<< sum << endl;
    return 0;
}

```

Принципиального отличия нет, но если присвоить переменной *i* значение, большее, чем 1000, то цикл все равно выполнит хотя бы один проход.

Попрактикуйтесь, поэкспериментируйте над собственными примерами задач. Циклы — очень важная вещь, поэтому им стоит уделить побольше внимания. Когда поймете, как работают циклы — можете смело переходить к изучению следующего урока.

Тема 5. Массивы в C++

Сегодня мы с поговорим о массивах. Вы уже знаете, что переменная — это ячейка в памяти компьютера, где может храниться одно единственное значение. **Массив** — это область памяти, где могут последовательно храниться несколько значений.

Возьмем группу студентов из десяти человек. У каждого из них есть фамилия. Создавать отдельную переменную для каждого студента — не рационально. Создадим массив, в котором будут храниться фамилии всех студентов.

Пример инициализации массива

```
string students[10] = {  
    "Иванов", "Петров", "Сидоров",  
    "Ахмедов", "Ерошкин", "Выхин",  
    "Андреев", "Вин Дизель", "Картошкин", "Чубайс"  
};
```

Описание синтаксиса

Массив создается почти так же, как и обычная переменная. Для хранения десяти фамилий нам нужен массив, состоящий из 10 элементов. Количество элементов массива задается при его объявлении и заключается в квадратные скобки.

Чтобы описать элементы массива сразу при его создании, можно использовать фигурные скобки. В фигурных скобках значения элементов массива перечисляются через запятую. В конце закрывающей фигурной скобки ставится точка с запятой.

Попробуем вывести наш массив на экран с помощью оператора `cout`.

```
#include<iostream>  
#include<string>
```

```
intmain()  
{  
    std::string students[10] = {  
        "Иванов", "Петров", "Сидоров",  
        "Ахмедов", "Ерошкин", "Выхин",  
        "Андреев", "Вин Дизель", "Картошкин", "Чубайс"  
    };  
    std::cout<< students <<std::endl; // Пытаемся вывести весь массив непосредственно  
    return0;  
}
```

Скомпилируйте этот код и посмотрите, на результат работы программы. Готово? А теперь запустите программу еще раз и сравните с предыдущим результатом. В моей операционной системе вывод был следующим:

- Первый вывод: 0x7fff8b85820
- Второй вывод: 0x7fff7a335f90
- Третий вывод: 0x7fff847eb40

Мы видим, что выводится адрес этого массива в оперативной памяти, а никакие не «Иванов» и «Петров».

Дело в том, что при создании переменной, ей выделяется определенное место в памяти. Если мы объявляем переменную типа `int`, то на машинном уровне она описывается двумя параметрами — ее адресом и размером хранимых данных.

Массивы в памяти хранятся таким же образом. Массив типа `int` из 10 элементов описывается с помощью адреса его первого элемента и количества байт, которое может вместить этот массив. Если для хранения одного целого числа выделяется 4 байта, то для массива из десяти целых чисел будет выделено 40 байт.

Так почему же, при повторном запуске программы, адреса различаются? Это сделано для защиты от атак переполнения буфера. Такая технология называется рандомизацией адресного пространства и реализована в большинстве популярных ОС.

Попробуем вывести первый элемент массива — фамилию студента Иванова.

```
#include<iostream>
#include<string>

intmain()
{
    std::string students[10] = {
        "Иванов", "Петров", "Сидоров",
        "Ахмедов", "Ерошкин", "Выхин",
        "Андреев", "Вин Дизель", "Картошкин", "Чубайс"
    };
    std::cout<<students[0] <<std::endl;
    return0;
}
```

Смотрим, компилируем, запускаем. Убедились, что вывелся именно «Иванов». Заметьте, что нумерация элементов массива в C++ начинается с нуля. Следовательно, фамилия первого студента находится в `students[0]`, а фамилия последнего — в `students[9]`.

В большинстве языков программирования нумерация элементов массива также начинается с нуля.

Попробуем вывести список всех студентов. Но сначала подумаем, а что если бы вместо группы из десяти студентов, была бы кафедра их ста, факультет из тысячи, или даже весь университет? Ну не будем же мы писать десятки тысяч строк с `cout`?

Конечно же нет! Мы будем использовать циклы.

Вывод элементов массива через цикл

```
#include<iostream>
#include<string>

intmain()
{
```

```
std::string students[10] = {
    "Иванов", "Петров", "Сидоров",
    "Ахмедов", "Ерошкин", "Выхин",
    "Андреев", "Вин Дизель", "Картошкин", "Чубайс"
};
for (int i = 0; i < 10; i++) {
    std::cout << students[i] << std::endl;
}

return 0;
}
```

Если бы нам пришлось выводить массив из нескольких тысяч фамилий, то мы бы просто увеличили конечное значение счетчика цикла — строку `for (...; i < 10; ...)` заменили на `for (...; i < 10000; ...)`.

Заметьте что счетчик нашего цикла начинается с нуля, а заканчивается девяткой. Если вместо оператора строгого неравенства — `i < 10` использовать оператор «меньше, либо равно» — `i <= 10`, то на последней итерации программа обратится к несуществующему элементу массива — `students[10]`. Это может привести к ошибкам сегментации и аварийному завершению программы. Будьте внимательны — подобные ошибки бывает сложно отловить.

Массив, как и любую переменную можно не заполнять значениями при объявлении.

Объявление массива без инициализации

```
string students[10];
// или
string teachers[5];
```

Элементы такого массива обычно содержат в себе «мусор» из выделенной, но еще не инициализированной, памяти. Некоторые компиляторы, такие как GCC, заполняют все элементы массива нулями при его создании.

При создании статического массива, для указания его размера может использоваться только константа. Размер выделяемой памяти определяется на этапе компиляции и не может изменяться в процессе выполнения.

```
int n;
cin >> n;
string students[n]; /* Неверно */
```

Выделение памяти в процессе выполнения возможно при работе с динамическими массивами. Но о них немного позже.

Заполним с клавиатуры пустой массив из 10 элементов.

Заполнение массива с клавиатуры

```
#include <iostream>
#include <string>

using std::cout;
using std::cin;
using std::endl;
```

```

int main()
{
    int arr[10];

    // Заполняем массив с клавиатуры
    for (int i = 0; i < 10; i++) {
        cout << "[" << i + 1 << "]" << ": ";
        cin >> arr[i];
    }

    // И выводим заполненный массив.
    cout << "\nВаш массив: ";

    for (int i = 0; i < 10; ++i) {
        cout << arr[i] << " ";
    }

    cout << endl;

    return 0;
}

```

Скомпилируем эту программу и проверим ее работу.

Терминал — zsh — 80×24

```

ssh
zsh
selevit ~ % ./foo
[1]: 4
[2]: 5
[3]: 3
[4]: 3
[5]: 5
[6]: 2
[7]: 8
[8]: 4
[9]: 4
[10]: 9

Ваш массив: 4 5 3 3 5 2 8 4 4 9
selevit ~ %

```

Если у вас возникают проблемы при компиляции исходников из уроков — внимательно прочитайте ошибку компилятора, попробуйте проанализировать и исправить ее.

Массивы — очень важная вещь в программировании. СоветуюТема 6. вам хорошо попрактиковаться в работе с ними.

Тема 6. Функции в С

Очень часто в программировании необходимо выполнять одни и те же действия. Например, мы хотим выводить пользователю сообщения об ошибке в разных местах программы, если он ввел неверное значение. без функций это выглядело бы так:

```
#include<iostream>
#include<string>

usingnamespacestd;

intmain()
{
    string valid_pass = "qwerty123";
    string user_pass;
    cout<<"Введитепароль: ";
    getline(cin, user_pass);
    if (user_pass == valid_pass) {
        cout<<"Доступразрешен."<<endl;
    } else {
        cout<<"Неверный пароль!"<< endl;
    }
    return0;
}
```

А вот аналогичный пример с функцией:

```
#include<iostream>
#include<string>

usingnamespacestd;

voidcheck_pass(string password)
{
    string valid_pass = "qwerty123";
    if (password == valid_pass) {
        cout<<"Доступразрешен."<<endl;
    } else {
        cout<<"Неверныйпароль!"<<endl;
    }
}

intmain()
{
    string user_pass;
    cout<<"Введитепароль: ";
    getline (cin, user_pass);
    check_pass (user_pass);
    return0;
}
```

По сути, после компиляции не будет никакой разницы для процессора, как для первого кода, так и для второго. Но ведь такую проверку пароля мы можем делать в нашей программе довольно много раз. И тогда получается копи-паст и код становится нечитаемым. Функции — один из самых важных компонентов языка C++.

- Любая функция имеет тип, также, как и любая переменная.
- Функция может возвращать значение, тип которого в большинстве случаев аналогично типу самой функции.
- Если функция не возвращает никакого значения, то она должна иметь тип **void** (такие функции иногда называют процедурами)
- При объявлении функции, после ее типа должно находиться имя функции и две круглые скобки — открывающая и закрывающая, внутри которых могут находиться один или несколько аргументов функции, которых также может не быть вообще.
- после списка аргументов функции ставится открывающая фигурная скобка, после которой находится само тело функции.
- В конце тела функции обязательно ставится закрывающая фигурная скобка.

Пример построения функции

```
#include<iostream>
using namespace std;

void function_name()
{
    cout<<"Hello, world"<< endl;
}

int main()
{
    function_name(); // Вызов функции
    return 0;
}
```

Перед вами тривиальная программа, **Hello, world**, только реализованная с использованием функций.

Если мы хотим вывести «Hello, world» где-то еще, нам просто нужно вызвать соответствующую функцию. В данном случае это делается так: `function_name();`. Вызов функции имеет вид имени функции с последующими круглыми скобками. Эти скобки могут быть пустыми, если функция не имеет аргументов. Если же аргументы в самой функции есть, их необходимо указать в круглых скобках.

Также существует такое понятие, как параметры функции по умолчанию. Такие параметры можно не указывать при вызове функции, т.к. они примут значение по умолчанию, указанно после знака присваивания после данного параметра и списке всех параметров функции.

В предыдущих примерах мы использовали функции типа `void`, которые не возвращают никакого значения. Как многие уже догадались, оператор `return` используется для возвращения вычисляемого функцией значения.

Рассмотрим пример функции, возвращающей значение на примере проверки пароля.

```

#include<iostream>
#include<string>

using namespace std;

string check_pass(string password)
{
    string valid_pass = "qwerty123";
    string error_message;
    if (password == valid_pass) {
        error_message = "Доступ разрешен.";
    } else {
        error_message = "Неверный пароль!";
    }
    return error_message;
}

int main()
{
    string user_pass;
    cout<<"Введите пароль: ";
    getline (cin, user_pass);
    string error_msg = check_pass (user_pass);
    cout<< error_msg << endl;
    return 0;
}

```

В данном случае функция **check_pass** имеет тип **string**, следовательно она будет возвращать только значение типа string, иными словами говоря строку. Давайте рассмотрим алгоритм работы этой программы.

Самой первой выполняется функция **main()**, которая должна присутствовать в каждой программе. Теперь мы объявляем переменную **user_pass** типа string, затем выводим пользователю сообщение «Введите пароль», который после ввода попадает в строку **user_pass**. А вот дальше начинает работать наша собственная функция **check_pass()**.

В качестве аргумента этой функции передается строка, введенная пользователем.

Аргумент функции — это, если сказать простым языком переменные или константы вызывающей функции, которые будет использовать вызываемая функция.

При объявлении функций создается **формальный параметр**, имя которого может отличаться от параметра, передаваемого при вызове этой функции. Но типы формальных параметров и передаваемых функции аргументов в большинстве случаев должны быть аналогичны.

После того, как произошел вызов функции **check_pass()**, начинает работать данная функция. Если функцию нигде не вызвать, то этот код будет проигнорирован программой. Итак, мы передали в качестве аргумента строку, которую ввел пользователь.

Теперь эта строка в полном распоряжении функции (хочу обратить Ваше внимание на то, что переменные и константы, объявленные в разных функциях независимы друг от друга, они даже могут иметь одинаковые имена. В следующих уроках я расскажу о том, что такое область видимости, локальные и глобальные переменные).

Теперь мы проверяем, правильный ли пароль ввел пользователь или нет. если пользователь ввел правильный пароль, присваиваем переменной `error_message` соответствующее значение. если нет, то сообщение об ошибке.

После этой проверки мы **возвращаем** переменную `error_message`. На этом работа нашей функции закончена. А теперь, в функции `main()`, то значение, которое возвратила наша функция мы присваиваем переменной `error_msg` и выводим это значение (строку) на экран терминала.

Также, можно организовать повторный ввод пароля с помощью **рекурсии** (о ней мы еще поговорим). Если объяснить вкратце, рекурсия — это когда функция вызывает сама себя. Смотритеещеодинапример:

```
#include<iostream>
#include<string>

using namespace std;

bool password_is_valid(string password)
{
    string valid_pass = "qwerty123";
    if (valid_pass == password)
        return true;
    else
        return false;
}

void get_pass()
{
    string user_pass;
    cout<<"Введите пароль: ";
    getline(cin, user_pass);
    if (!password_is_valid(user_pass)) {
        cout<<"Неверный пароль!"<< endl;
        get_pass (); // Здесь делаем рекурсию
    } else {
        cout<<"Доступ разрешен."<< endl;
    }
}

int main()
{
    get_pass ();
    return 0;
}
```

Функции очень сильно облегчают работу программисту и намного повышают читаемость и понятность кода, в том числе и для самого разработчика (не удивляйтесь этому, т. к. если вы откроете код, написанный вами полгода назад, не сразу поймете соль, поверьте на слово).

Не расстраивайтесь, если не сразу поймете все аспекты функций в C++, т. к. это довольно сложная тема и мы еще будем разбирать примеры с функциями в следующих уроках.

Совет: не бойтесь экспериментировать, это очень хорошая практика, а после прочтения данной статьи порешайте элементарные задачи, но с использованием функций. Это будет очень полезно для вас.

Если Вы найдете какие-либо ошибки в моем коде, обязательно напишите об этом в комментариях. здесь же можно задавать все вопросы.

Следующий урок: **Указатели в C++** →.

Тема 7. Указатели в C++

При выполнении любой программы, все необходимые для ее работы данные должны быть загружены в оперативную память компьютера. Для обращения к переменным, находящимся в памяти, используются специальные адреса, которые записываются в шестнадцатеричном виде, например, 0x100 или 0x200.

Если переменных в памяти потребуется слишком большое количество, которое не сможет вместить в себя сама аппаратная часть, произойдет перегрузка системы или её зависание.

Если мы объявляем переменные статично, так как мы делали в предыдущих уроках, они остаются в памяти до того момента, как программа завершит свою работу, а после чего уничтожаются.

Такой подход может быть приемлем в простых примерах и несложных программах, которые не требуют большого количества ресурсов. Если же наш проект является огромным программным комплексом с высоким функционалом, объявлять таким образом переменные, естественно, было бы довольно не умно.

Можете себе представить, если бы небезызвестная **Battlefield 3** использовала такой метод работы с данными? В таком случае, самым заядлым геймерам пришлось бы перезагружать свои высоконагруженные системы кнопкой reset после нескольких секунд работы игры.

Дело в том, что играя в тот же Battlefield, геймер в каждый новый момент времени видит различные объекты на экране монитора, например, сейчас я стреляю во врага, а через долю секунды он уже падает убитым, создавая вокруг себя множество спецэффектов, таких как пыль, тени, и т.п.

Естественно, все это занимает какое-то место в оперативной памяти компьютера. Если не уничтожать неиспользуемые объекты, очень скоро они заполнят весь объем ресурсов ПК.

По этим причинам, в большинстве языков, в том числе и C/C++, имеется понятие указателя. Указатель — это переменная, хранящая в себе адрес ячейки оперативной памяти, например, **0x100**.

Мы можем обращаться, например к массиву данных через указатель, который будет содержать адрес начала диапазона ячеек памяти, хранящих этот массив.

После того, как этот массив станет не нужен для выполнения остальной части программы, мы просто освободим память по адресу этого указателя, и она вновь станет доступна для других переменных.

Ниже приведен конкретный пример обращения к переменным через указатель и напрямую.

Пример использования статических переменных

```
#include<iostream>
using namespace std;
```

```

int main()
{
    int a; // Объявление статической переменной
    int b = 5; // Инициализация статической переменной b

    a = 10;
    b = a + b;
    cout << "b is " << b << endl;
    return 0;
}

```

Пример использования динамических переменных

```

#include <iostream>
using namespace std;

int main()
{
    int *a = new int; // Объявление указателя для переменной типа int
    int *b = new int(5); // Инициализация указателя

    *a = 10;
    *b = *a + *b;

    cout << "b is " << *b << endl;

    delete b;
    delete a;

    return 0;
}

```

Синтаксис первого примера вам уже должен быть знаком. Мы объявляем/инициализируем статические переменные **a** и **b**, после чего выполняем различные операции напрямую с ними.

Во втором примере мы оперируем динамическими переменными посредством указателей. Рассмотрим общий синтаксис указателей в C++.

Выделение памяти осуществляется с помощью оператора **new** и имеет вид: **тип_данных *имя_указателя = new тип_данных;**, например **int *a = new int;**. После удачного выполнения такой операции, в оперативной памяти компьютера происходит выделение диапазона ячеек, необходимого для хранения переменной типа **int**.

Логично предположить, что для разных типов данных выделяется разное количество памяти. Следует быть особенно осторожным при работе с памятью, потому что именно ошибки программы, вызванные утечкой памяти, являются одними из самых трудно находимых. На отладку программы в поисках одной ничтожной ошибки, может уйти час, день, неделя, в зависимости от упорности разработчика и объема кода.

Инициализация значения, находящегося по адресу указателя выполняется схожим образом, только в конце ставятся круглые скобки с нужным значением: **тип_данных *имя_указателя = new тип_данных(значение)**. В нашем примере это **int *b = new int(5)**.

Для того, чтобы получить **адрес** в памяти, на который ссылается указатель, используется имя переменной-указателя с префиксом **&**. перед ним (*не путать со знаком ссылки в C++*).

Например, чтобы вывести на экран адрес ячейки памяти, на который ссылается указатель **b** во втором примере, мы пишем `cout << "Address of b is " << &b << endl;`. В моей системе, я получил значение **0x1aba030**. У вас оно может быть другим, потому что адреса в оперативной памяти распределяются таким образом, чтобы максимально уменьшить фрагментацию. Поскольку, в любой системе список запущенных процессов, а также объем и разрядность памяти могут отличаться, система сама распределяет данные для обеспечения минимальной фрагментации.

Для того, чтобы получить **значение**, которое находится **по адресу**, на который ссылается указатель, **используется префикс ***. Данная операция называется **разыменованием указателя**.

Во втором примере мы выводим на экран значение, *которое находится в ячейке памяти* (у меня это **0x1aba030**): `cout << "b is " << *b << endl;`. В этом случае необходимо использовать знак *****.

Чтобы изменить значение, находящееся по адресу, на который ссылается указатель, нужно также использовать звездочку, например, как во втором примере — `*b = *a + *b;`

- Когда мы оперируем **данными**, то используем знак *****
- Когда мы оперируем **адресами**, то используем знак **&**

В этих вещах очень часто возникают недопонимания, и кстати, не только у новичков. Многие из тех, кто начинал программировать с того же php, также часто испытывают подобную путаницу при работе с памятью.

Для того, чтобы освободить память, выделенную оператором **new**, используется оператор delete.

Пример освобождения памяти

```
#include<iostream>
using namespace std;

int main()
{
    // Выделение памяти
    int *a = new int;
    int *b = new int;
    float *c = new float;

    // ... Любые действия программы

    // Освобождение выделенной памяти
    delete c;
    delete b;
    delete a;

    return 0;
}
```

При использовании оператора delete для указателя, знак * **не используется**.

Тема 8. Динамические массивы в С

Когда мы разбирали понятие массива, то при объявлении мы задавали массиву определенный постоянный размер. Возможно, кто-то пробовал делать так:

```
int n = 10;
int arr[n];
```

Но, как уже было сказано — при объявлении статического массива, его размером должна являться числовая константа, а не переменная. В большинстве случаев, целесообразно выделять определенное количество памяти для массива, значение которого изначально неизвестно.

Например, необходимо создать динамический массив из **N** элементов, где значение **N** задается пользователем. Мы уже учились выделять память для переменных, используя указатели. Выделение памяти для динамического массива имеет аналогичный принцип.

Создание динамического массива

```
#include<iostream>
using namespace std;

int main()
{
    int num; // размер массива
    cout<<"Enter integer value: ";
    cin>> num; // получение от пользователя размера массива

    int *p_darr = new int[num]; // Выделение памяти для массива
    for (int i = 0; i < num; i++) {
        // Заполнение массива и вывод значений его элементов
        p_darr[i] = i;
        cout<<"Value of "<< i <<" element is "<< p_darr[i] << endl;
    }
    delete [] p_darr; // очистка памяти
    return 0;
}
```

Синтаксис выделения памяти для массива имеет вид указатель = new тип[размер]. В качестве размера массива может выступать любое целое положительное значение.

Следующий урок: **Параметры командной строки в С++** →.

Тема 9. Параметры командной строки в С++

При запуске программы из командной строки, ей можно передавать дополнительные параметры в текстовом виде. Например, следующая команда

```
ping -t 5 google.com
```

Будет отправлять пакеты на адрес google.com с интервалом в 5 секунд. Здесь мы передали программе ping три параметра: «-t», «5» и «google.com», которые программа интерпретирует как задержку между запросами и адрес хоста для обмена пакетами.

В программе эти параметры из командной строки можно получить через аргументы функции main при использовании функции main в следующей форме:

```
int main(int argc, char* argv[]) { /* ... */ }
```

Первый аргумент содержит количество параметров командной строки. Вторым аргументом — это массив строк, содержащий параметры командной строки. Т.е. первый аргумент указывает количество элементов массива во втором аргументе.

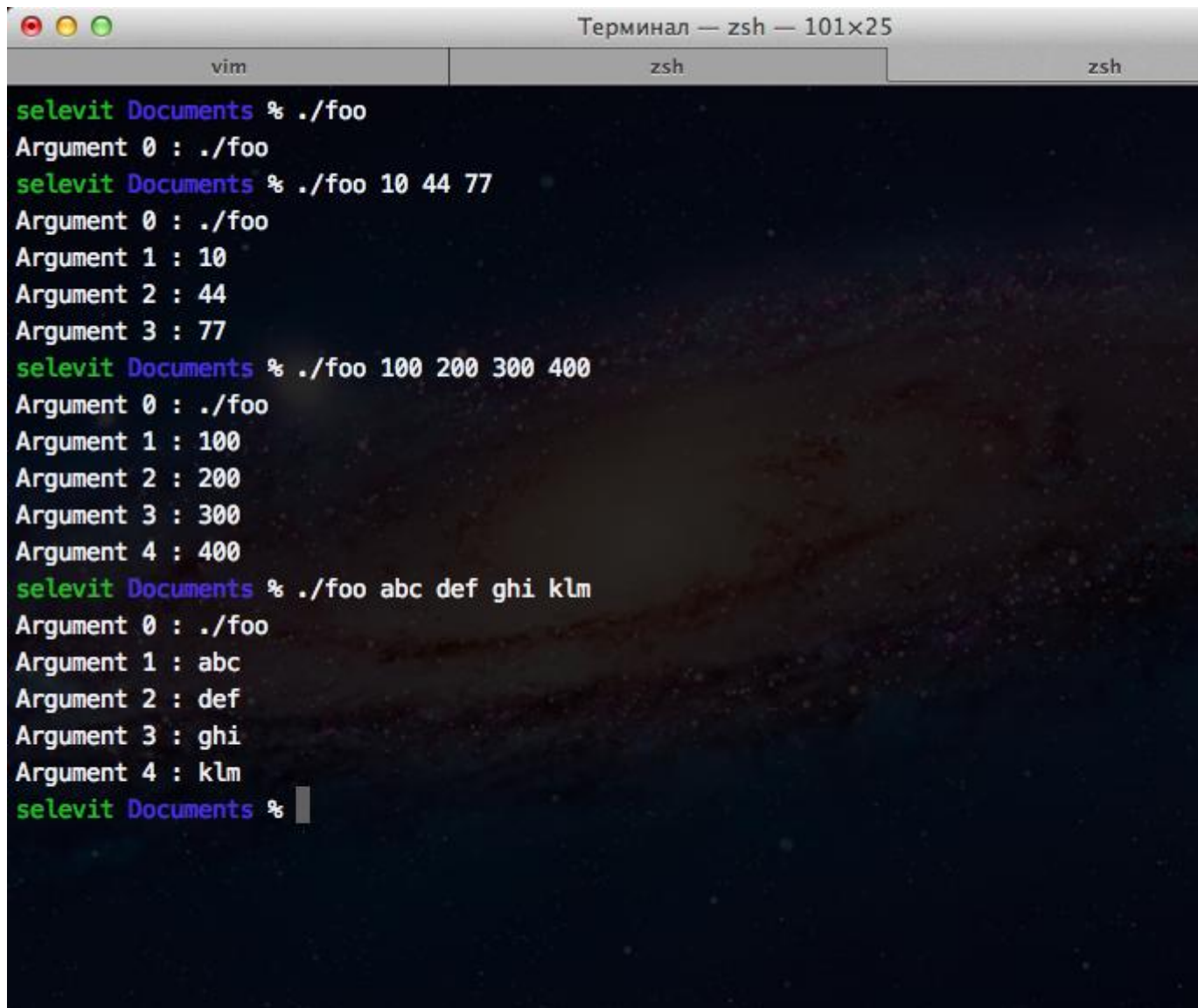
Первый элемент массива строк (argv[0]) всегда содержит строку, использованную для запуска программы (либо пустую строку). Следующие элементы (от 1 до argc - 1) содержат параметры командной строки, если они есть. Элемент массива строк argv[argc] всегда должен содержать 0.

Пример 1

```
#include<iostream>
using namespace std;
```

```
int main(int argc, char *argv[])
{
    for (int i = 0; i < argc; i++) {
        // Выводим список аргументов в цикле
        cout<<"Argument "<< i <<" : "<< argv[i] << endl;
    }
    return 0;
}
```

Откройте командную строку и запустите оттуда скомпилированную программу.

A terminal window titled "Терминал — zsh — 101x25" with tabs for "vim" and "zsh". The prompt is "selevit Documents %". The user enters several commands: 1. `./foo`, showing `Argument 0 : ./foo`. 2. `./foo 10 44 77`, showing `Argument 0 : ./foo`, `Argument 1 : 10`, `Argument 2 : 44`, and `Argument 3 : 77`. 3. `./foo 100 200 300 400`, showing `Argument 0 : ./foo`, `Argument 1 : 100`, `Argument 2 : 200`, `Argument 3 : 300`, and `Argument 4 : 400`. 4. `./foo abc def ghi klm`, showing `Argument 0 : ./foo`, `Argument 1 : abc`, `Argument 2 : def`, `Argument 3 : ghi`, and `Argument 4 : klm`. The prompt is then shown again with a cursor.

Для получения числовых данных из входных параметров, можно использовать функции `atoi` и `atof`.

Тема 10. Классы в C++

Весь реальный мир состоит из объектов. Города состоят из районов, в каждом районе есть свои названия улиц, на каждой улице находятся жилые дома, которые также состоят из объектов.

Практически любой материальный предмет можно представить в виде совокупности объектов, из которых он состоит. Допустим, что нам нужно написать программу для учета успеваемости студентов. Можно представить группу студентов, как класс языка C++. Назовем его `Students`.

Основные понятия

Классы в программировании состоят из свойств и методов. Свойства — это любые данные, которыми можно характеризовать объект класса. В нашем случае, объектом класса является студент, а его свойствами — имя, фамилия, оценки и средний балл.

У каждого студента есть имя — `name` и фамилия `last_name`. Также, у него есть промежуточные оценки за весь семестр. Эти оценки мы будем записывать в целочисленный массив из пяти элементов. После того, как все пять оценок будут проставлены, определим средний балл успеваемости студента за весь семестр — свойство `average_ball`.

Методы — это функции, которые могут выполнять какие-либо действия над данными (свойствами) класса. Добавим в наш класс функцию `calculate_average_ball()`, которая будет определять средний балл успеваемости ученика.

- **Методы** класса — это его функции.
- **Свойства** класса — его переменные.

```
class Students {  
public:  
    // Функция, считающая средний балл  
    void calculate_average_ball()  
    {  
        int sum = 0; // Сумма всех оценок  
        for (int i = 0; i < 5; ++i) {  
            sum += scores[i];  
        }  
        // считаем среднее арифметическое  
        average_ball = sum / 5.0;  
    }  
  
    // Имя студента  
    std::string name;  
    // Фамилия  
    std::string last_name;  
    // Пять промежуточных оценок студента  
    int scores[5];  
  
private:  
    // Итоговая оценка за семестр  
    float average_ball;  
};
```

Функция `calculate_average_ball()` просто делит сумму всех промежуточных оценок на их количество.

Модификаторы доступа `public` и `private`

Все свойства и методы классов имеют права доступа. По умолчанию, все содержимое класса является доступным для чтения и записи только для него самого. Для того, чтобы разрешить доступ к данным класса извне, используют модификатор доступа `public`. Все функции и переменные, которые находятся после модификатора `public`, становятся доступными из всех частей программы.

Закрытые данные класса размещаются после модификатора доступа `private`. Если отсутствует модификатор `public`, то все функции и переменные, по умолчанию являются закрытыми (как в первом примере).

Обычно, приватными делают все свойства класса, а публичными — его методы. Все действия с закрытыми свойствами класса реализуются через его методы. Рассмотрим следующий код.


```

class Students {
public:
    // Установка среднего балла
    void set_average_ball(float ball)
    {
        average_ball = ball;
    }
    // Получение среднего балла
    float get_average_ball()
    {
        return average_ball;
    }
    std::string name;
    std::string last_name;
    int scores[5];

private:
    float average_ball;
};

```

Мы не можем напрямую обращаться к закрытым данным класса. Работать с этими данными можно только посредством методов этого класса. В примере выше, мы используем функцию `get_average_ball()` для получения средней оценки студента, и `set_average_ball()` для выставления этой оценки.

Функция `set_average_ball()` принимает средний балл в качестве параметра и присваивает его значение закрытой переменной `average_ball`. Функция `get_average_ball()` просто возвращает значение этой переменной.

Программа учета успеваемости студентов

Создадим программу, которая будет заниматься учетом успеваемости студентов в группе. Создайте заголовочный файл **students.h**, в котором будет находиться класс `Students`.

```

/* students.h */
#include<string>

class Students {
public:
    // Установка имени студента
    void set_name(std::string student_name)
    {
        name = student_name;
    }
    // Получение имени студента
    std::string get_name()
    {
        return name;
    }
    // Установка фамилии студента
    void set_last_name(std::string student_last_name)
    {
        last_name = student_last_name;
    }
    // Получение фамилии студента
    std::string get_last_name()

```

```

{
    return last_name;
}
// Установка промежуточных оценок
void set_scores(int student_scores[])
{
    for (int i = 0; i < 5; ++i) {
        scores[i] = student_scores[i];
    }
}
// Установка среднего балла
void set_average_ball(float ball)
{
    average_ball = ball;
}
// Получение среднего балла
float get_average_ball()
{
    return average_ball;
}

private:
// Промежуточные оценки
int scores[5];
// Средний балл
float average_ball;
// Имя
std::string name;
// Фамилия
std::string last_name;
};

```

Мы добавили в наш класс новые методы, а также сделали приватными все его свойства. Функция `set_name()` сохраняет имя студента в переменной `name`, а `get_name()` возвращает значение этой переменной. Принцип работы функций `set_last_name()` и `get_last_name()` аналогичен.

Функция `set_scores()` принимает массив с промежуточными оценками и сохраняет их в приватную переменную `int scores[5]`.

Теперь создайте файл **main.cpp** со следующим содержанием.

```

/* main.cpp */
#include <iostream>
#include "students.h"

int main()
{
    // Создание объекта класса Student
    Student student;

    std::string name;
    std::string last_name;

    // Вводим имя с клавиатуры
    std::cout << "Name: ";
    getline(std::cin, name);
}

```

```

// Ввод фамилии
std::cout<<"Last name: ";
getline(std::cin, last_name);

// Сохранение имени и фамилии в объект класса Students
student.set_name(name);
    student.set_last_name(last_name);

// Оценки
int scores[5];
// Сумма всех оценок
int sum = 0;

// Ввод промежуточных оценок
for (int i = 0; i < 5; ++i) {
    std::cout<<"Score "<< i+1<<": ";
    std::cin>> scores[i];
    // суммирование
    sum += scores[i];
}

// Сохраняем промежуточные оценки в объект класса Student
student.set_scores(scores);
// Считаем средний балл
float average_ball = sum / 5.0;
// Сохраняем средний балл в объект класса Students
student.set_average_ball(average_ball);
// Выводим данные по студенту
std::cout<<"Average ball for "<< student.get_name() <<" "
<< student.get_last_name() <<" is "
<< student.get_average_ball() <<std::endl;

return 0;
}

```

В самом начале программы создается объект класса Students. Дело в том, что сам класс является только описанием его объекта. Класс Students является описанием любого из студентов, у которого есть имя, фамилия и возможность получения оценок.

Объект класса Students характеризует конкретного студента. Если мы захотим выставить оценки всем ученикам в группе, то будем создавать новый объект для каждого из них. Использование классов очень хорошо подходит для описания объектов реального мира.

После создания объекта student, мы вводим с клавиатуры фамилию, имя и промежуточные оценки для конкретного ученика. Пускай это будет Вася Пупкин, у которого есть пять оценок за семестр — две тройки, две четверки и одна пятерка.

Введенные данные мы передаем **set**-функциям, которые присваивают их закрытым переменным класса. После того, как были введены промежуточные оценки, мы высчитываем средний балл на основе этих оценок, а затем сохраняем это значение в закрытом свойстве average_ball, с помощью функции set_average_ball().

Скомпилируйте и запустите программу.

```
selevit@gentoo students-cpp % c++ main.cpp -o students
selevit@gentoo students-cpp % ./students
Name: Вася
Last name: Пупкин
Score 1: 3
Score 2: 3
Score 3: 4
Score 4: 4
Score 5: 5
Average ball for Вася Пупкин is 3.8
selevit@gentoo students-cpp %
```

Отделение данных от логики

Вынесем реализацию всех методов класса в отдельный файл **students.cpp**.

```
/* students.cpp */
#include<string>
#include"students.h"

// Установка имени студента
void Students::set_name(std::string student_name)
{
    Students::name = student_name;
}

// Получение имени студента
std::string Students::get_name()
{
    return Students::name;
}

// Установка фамилии студента
void Students::set_last_name(std::string student_last_name)
{
    Students::last_name = student_last_name;
}

// Получение фамилии студента
std::string Students::get_last_name()
{
    return Students::last_name;
}

// Установка промежуточных оценок
void Students::set_scores(int scores[])
{
    for (int i = 0; i < 5; ++i) {
        Students::scores[i] = scores[i];
    }
}

// Установка среднего балла
void Students::set_average_ball(float ball)
{
    Students::average_ball = ball;
}
```

```

}

// Получение среднего балла
float Students::get_average_ball()
{
    return Students::average_ball;
}

```

В заголовочном файле **students.h** оставим только прототипы этих методов.

```

/* students.h */
#pragma once /* Защита от двойного подключения заголовочного файла */
#include<string>

class Students {
public:
    // Установка имени студента
    void set_name(std::string);
    // Получение имени студента
    std::string get_name();
    // Установка фамилии студента
    void set_last_name(std::string);
    // Получение фамилии студента
    std::string get_last_name();
    // Установка промежуточных оценок
    void set_scores(int []);
    // Установка среднего балла
    void set_average_ball(float);
    // Получение среднего балла
    float get_average_ball();

private:
    // Промежуточные оценки
    int scores[5];
    // Средний балл
    float average_ball;
    // Имя
    std::string name;
    // Фамилия
    std::string last_name;
};

```

Такой подход называется абстракцией данных — одного из фундаментальных принципов объектно-ориентированного программирования. К примеру, если кто-то другой захочет использовать наш класс в своем коде, ему не обязательно знать, как именно высчитывается средний балл. Он просто будет использовать функцию `calculate_average_ball()` из второго примера, не вникая в алгоритм ее работы.

Над крупными проектами обычно работает несколько программистов. Каждый из них занимается написанием определенной части продукта. В таких масштабах кода, одному человеку практически нереально запомнить, как работает каждая из внутренних функций проекта. В нашей программе, мы используем оператор потокового вывода `cout`, не задумываясь о том, как он реализован на низком уровне. Кроме того, отделение данных от логики является хорошим тоном программирования.

В начале обучения мы говорили о пространствах имен (namespaces). Каждый класс в C++ использует свое пространство имен. Это сделано для того, чтобы избежать конфликтов

при именовании переменных и функций. В файле `students.cpp` мы используем оператор принадлежности `::` перед именем каждой функции. Это делается для того, чтобы указать компилятору, что эти функции принадлежат классу `Students`.

Создание объекта через указатель

При создании объекта, лучше не копировать память для него, а выделять ее в куче с помощью указателя. И освобождать ее после того, как мы закончили работу с объектом. Реализуем это в нашей программе, немного изменив содержимое файла `main.cpp`.

```
/* main.cpp */
#include<iostream>
#include"students.h"

intmain()
{
    // Выделение памяти для объекта Students
    Students *student = new Students;

    std::string name;
    std::string last_name;

    // Вводименисклавиатуры
    std::cout<<"Name: ";
    getline(std::cin, name);

    // Вводфамилии
    std::cout<<"Last name: ";
    getline(std::cin, last_name);

    // Сохранение имени и фамилии в объект класса Students
    student->set_name(name);
    student->set_last_name(last_name);

    // Оценки
    int scores[5];
    // Сумма всех оценок
    int sum = 0;

    // Вводпромежуточныхоценок
    for (int i = 0; i < 5; ++i) {
        std::cout<<"Score "<< i+1<<": ";
        std::cin>> scores[i];
        // суммирование
        sum += scores[i];
    }
    // Сохраняем промежуточные оценки в объект класса Student
    student->set_scores(scores);

    // Считаемсреднийбалл
    float average_ball = sum / 5.0;
    // Сохраняем средний балл в объект класса Students
    student->set_average_ball(average_ball);
    // Выводимданныепостуденту
    std::cout<<"Average ball for "<< student->get_name() <<" "
    <<student->get_last_name() <<" is "
```

```

<<student->get_average_ball() <<std::endl;
// Удаление объекта student из памяти
delete student;
return 0;
}

```

При создании статического объекта, для доступа к его методам и свойствам, используют операция прямого обращения — «.» (символ точки). Если же память для объекта выделяется посредством указателя, то для доступа к его методам и свойствам используется оператор косвенного обращения — «->».

Конструктор и деструктор класса

Конструктор класса — это специальная функция, которая автоматически вызывается сразу после создания объекта этого класса. Он не имеет типа возвращаемого значения и должен называться также, как класс, в котором он находится. По умолчанию, заполним двойками массив с промежуточными оценками студента.

```

class Students {
// Имя студента
std::string name;
// Фамилия
std::string last_name;
// Пять промежуточных оценок студента
int scores[5];
// Итоговая оценка за семестр
float average_ball;
};

```

```

class Students {
public:
// Конструктор класса Students
Students(int default_score)
{
for (int i = 0; i < 5; ++i) {
scores[i] = default_score;
}
}
}

```

```

private:
int scores[5];
};

```

```

int main()
{
// Передаем двойку в конструктор
Students *student = new Students(2);
return 0;
}

```

Мы можем исправить двойки, если ученик будет хорошо себя вести, и вовремя сдавать домашние задания. А на «нет» и суда нет :-)

Деструктор класса вызывается при уничтожении объекта. Имя деструктора аналогично имени конструктора, только в начале ставится знак тильды ~. Деструктор не имеет входных параметров.

```
#include<iostream>

class Students {
public:
    // Деструктор
    ~Students()
    {
        std::cout<<"Memory has been cleaned. Good bye."<<std::endl;
    }
};

intmain()
{
    Students *student = new Students;
    // Уничтожениеобъекта
    delete student;
    return0;
}
```

```
selevit@gentoo code % c++ destructor.cc -o destructor
selevit@gentoo code % ./destructor
Memory has been cleaned. Good bye.
selevit@gentoo code %
```

Тема 11. Классы — продолжение

Сегодня мы более детально познакомимся с **конструкторами** и **деструкторами** класса, а также научимся работать с файлами в потоковом режиме, с помощью библиотеки **fstream**. Продолжим написание программы учета оценок.

- [Конструктор Students](#)
- [Сохранение оценок в файл](#)
- [Деструктор Students](#)

Конструктор Students

Добавим в класс Students конструктор, который будет принимать имя и фамилию ученика, и сохранять эти значения в соответствующих переменных класса.

```
// Конструктор Students
Students::Students(std::string name, std::string last_name)
{
    Students::set_name(name);
    Students::set_last_name(last_name);
}
```


При создании нового объекта, мы должны передать конструктору имя и фамилию студента. Иначе компиляция программы завершится с ошибкой.

```
std::string name = "Василий";  
std::string last_name = "Пупкин";
```

```
Students *student = new Students(name, last_name);
```

Теперь добавим прототип конструктора в файл **students.h**.

```
/* students.h */  
#pragma once /* Защита от двойного подключения заголовочного файла */  
#include<string>  
  
class Students {  
public:  
    // Конструктор класса Students  
    Students(std::string, std::string);  
  
    // Установка имени студента  
    void set_name(std::string);  
    // Получение имени студента  
    std::string get_name();  
  
    // Установка фамилии студента  
    void set_last_name(std::string);  
    // Получение фамилии студента  
    std::string get_last_name();  
  
    // Установка промежуточных оценок  
    void set_scores(int []);  
  
    // Установка среднего балла  
    void set_average_ball(float);  
    // Получение среднего балла  
    float get_average_ball();  
private:  
    // Промежуточные оценки  
    int scores[5];  
    // Средний балл  
    float average_ball;  
    // Имя  
    std::string name;  
    // Фамилия  
    std::string last_name;  
};
```

В файле **students.cpp** определим сам конструктор.

```
/* students.cpp */  
#include<string>  
#include<fstream>  
#include "students.h"  
  
// Конструктор Students  
Students::Students(std::string name, std::string last_name)  
{  
    Students::set_name(name);
```

```

    Students::set_last_name(last_name);
}

// Установка имени студента
void Students::set_name(std::string student_name)
{
    Students::name = student_name;
}

// Получение имени студента
std::string Students::get_name()
{
    return Students::name;
}

// Установка фамилии студента
void Students::set_last_name(std::string student_last_name)
{
    Students::last_name = student_last_name;
}

// Получение фамилии студента
std::string Students::get_last_name()
{
    return Students::last_name;
}

// Установка промежуточных оценок
void Students::set_scores(int scores[])
{
    int sum = 0;
    for (int i = 0; i < 5; ++i) {
        Students::scores[i] = scores[i];
        sum += scores[i];
    }
}

// Установка среднего балла
void Students::set_average_ball(float ball)
{
    Students::average_ball = ball;
}

// Получение среднего балла
float Students::get_average_ball()
{
    return Students::average_ball;
}

```

В **main()** мы принимаем от пользователя имя и фамилию ученика, и сохраняем их во временных локальных переменных. После этого создаем новый объект класса **Students**, передавая его конструктору эти переменные.

```

/* main.cpp */
#include<iostream>
#include"students.h"

```

```

intmain(int argc, char *argv[])
{
    // Локальная переменная, хранящая имя ученика
    std::string name;
    // Иегофамилию
    std::string last_name;

    // Вводи имени
    std::cout<<"Name: ";
    getline(std::cin, name);
    // Ифамилии
    std::cout<<"Last name: ";
    getline(std::cin, last_name);

    // Передача параметров конструктору
    Students *student = newStudents(name, last_name);

    // Оценки
    int scores[5];
    // Сумма всех оценок
    int sum = 0;

    // Ввод промежуточных оценок
    for (int i = 0; i <5; ++i) {
        std::cout<<"Score "<< i+1<<": ";
        std::cin>> scores[i];
        // суммирование
        sum += scores[i];
    }
    // Сохраняем промежуточные оценки в объект класса Student
    student->set_scores(scores);

    // Считаем средний балл
    float average_ball = sum / 5.0;
    // Сохраняем средний балл
    student->set_average_ball(average_ball);

    // Выводим данные по студенту
    std::cout<<"Average ball for "<< student->get_name() <<" "
    <<student->get_last_name() <<" is "
    <<student->get_average_ball() <<std::endl;
    // Удаление объекта student из памяти
    delete student;
    return0;
}

```

Сохранение оценок в файл

Чтобы после завершения работы с программой, все данные сохранялись, мы будем записывать их в текстовый файл.

Оценки каждого студента будут находится в отдельной строке. Имя и фамилии будут разделяться пробелами. После имени и фамилии ученика ставится еще один пробел, а затем перечисляются все его оценки.

Пример файла с оценками:

Василий Пупкин 54533
Иван Сидоров 55345
Андрей Иванов 53333

Для работы с файлами мы воспользуемся библиотекой `fstream`, которая подключается в заголовочном файле с таким же именем.

```
#include<fstream>
```

```
// Запись данных о студенте в файл
void Students::save()
{
    std::ofstream fout("students.txt", std::ios::app);

    fout<< Students::get_name() <<" "
    << Students::get_last_name() <<" ";

    for (int i = 0; i <5; ++i) {
        fout<< Students::scores[i] <<" ";
    }

    fout<<std::endl;
    fout.close();
}
```

Переменная `fout` — это объект класса `ofstream`, который находится внутри библиотеки `fstream`. Класс `ofstream` используется для записи каких-либо данных во внешний файл. Кстати, у него тоже есть конструктор. Он принимает в качестве параметров имя выходного файла и режим записи.

В данном случае, мы используем режим добавления — `std::ios::app` (англ. `append`). После завершения работы с файлом, необходимо вызвать метод `close()` для того, чтобы закрыть файловый дескриптор.

Чтобы сохранить оценки студента, мы будем вызывать только что созданный метод `save()`.

```
Students student = newStudents("Василий", "Пупкин");
student->save();
```

Деструктор Students

Логично было бы сохранять все оценки после того, как работа со студентом закончена. Для этого создадим **деструктор** класса `Students`, который будет вызывать метод `save()` перед уничтожением объекта.

```
// Деструктор Students
Students::~Students()
{
    Students::save();
}
```

Добавим прототипы деструктора и метода `save()` в **students.h**.

```
/* students.h */
#pragma once /* Защита от двойного подключения заголовочного файла */
#include<string>
```

```

class Students {
public:
    // Запись данных о студенте в файл
    void save();
    // Деструктор класса Students
    ~Students();

    // Конструктор класса Students
    Students(std::string, std::string);
    // Установка имени студента
    void set_name(std::string);
    // Получение имени студента
    std::string get_name();

    // Установка фамилии студента
    void set_last_name(std::string);
    // Получение фамилии студента
    std::string get_last_name();

    // Установка промежуточных оценок
    void set_scores(int []);
    // Получение массива с промежуточными оценками
    int *get_scores();
    // Получение строки с промежуточными оценками
    std::string get_scores_str(char);

    // Установка среднего балла
    void set_average_ball(float);
    // Получение среднего балла
    float get_average_ball();
private:
    // Промежуточные оценки
    int scores[5];
    // Средний балл
    float average_ball;
    // Имя
    std::string name;
    // Фамилия
    std::string last_name;
};

```

И определим эти функции в **students.cpp**.

```

/* students.cpp */
#include <string>
#include <fstream>

#include "students.h"

// Деструктор Students
Students::~Students()
{
    Students::save();
}

// Запись данных о студенте в файл
void Students::save()

```

```

{
std::ofstream fout("students.txt", std::ios::app);

fout<< Students::get_name() <<" "
<< Students::get_last_name() <<" ";

for (int i = 0; i <5; ++i) {
fout<< Students::scores[i] <<" ";
}

fout<<std::endl;
fout.close();
}

// Конструктор Students
Students::Students(std::string name, std::string last_name)
{
    Students::set_name(name);
    Students::set_last_name(last_name);
}

// Установка имени студента
void Students::set_name(std::string student_name)
{
    Students::name = student_name;
}

// Получение имени студента
std::string Students::get_name()
{
    return Students::name;
}

// Установка фамилии студента
void Students::set_last_name(std::string student_last_name)
{
    Students::last_name = student_last_name;
}

// Получение фамилии студента
std::string Students::get_last_name()
{
    return Students::last_name;
}

// Установка промежуточных оценок
void Students::set_scores(int scores[])
{
    int sum = 0;
    for (int i = 0; i <5; ++i) {
        Students::scores[i] = scores[i];
        sum += scores[i];
    }
}

// Получение массива с промежуточными оценками
int *Students::get_scores()

```

```

{
return Students::scores;
}

// Установка среднего балла
void Students::set_average_ball(float ball)
{
    Students::average_ball = ball;
}

// Получение среднего балла
float Students::get_average_ball()
{
return Students::average_ball;
}

```

Содержимое **main.cpp** останется прежним. Скомпилируйте и запустите программу. Перед тем, как приложение завершит свою работу, в директории с исполняемым файлом будет создан новый текстовый файл с оценками — **students.txt**.

```

sele@webpp lesson-test % make
c++ -Wall -ggdb -o students students.cpp main.cpp
sele@webpp lesson-test % ./students
Name: Василий
Last name: Пупкин
Score 1: 5
Score 2: 5
Score 3: 3
Score 4: 4
Score 5: 3
Average ball for Василий Пупкин is 4
sele@webpp lesson-test % cat students.txt
Василий Пупкин 5 5 3 4 3
sele@webpp lesson-test % 

```

Тема 12. Векторы в C++

Вектор в C++ — это замена стандартному динамическому массиву, память для которого выделяется вручную, с помощью оператора `new`.

Разработчики языка рекомендуют использовать именно `vector` вместо ручного выделения памяти для массива. Это позволяет избежать утечек памяти и облегчает работу программисту.

Пример создания вектора

```

#include<iostream>
#include<vector>

int main()
{
// Вектор из 10 элементов типа int
std::vector<int> v1(10);

```

```

// Вектор из элементов типа float
// С неопределенным размером
std::vector<float> v2;

// Вектор, состоящий из 10 элементов типа int
// По умолчанию все элементы заполняются нулями
std::vector<int> v3(10, 0);

return 0;
}

```

Управление элементами вектора

Создадим вектор, в котором будет содержаться произвольное количество фамилий студентов.

```

#include<iostream>
#include<vector>
#include<string>

int main()
{
    // Поддержка кириллицы в консоли Windows
    setlocale(LC_ALL, "");

    // Создание вектора из строк
    std::vector<std::string> students;

    // Буфер для ввода фамилии студента
    std::string buffer = "";

    std::cout<<"Вводите фамилии студентов. "
    <<"По окончании ввода введите пустую строку"<<std::endl;

    do {
        std::getline(std::cin, buffer);
        if (buffer.size() > 0) {
            // Добавление элемента в конец вектора
            students.push_back(buffer);
        }
    } while (buffer != "");

    // Сохраняем количество элементов вектора
    unsigned int vector_size = students.size();

    // Вывод заполненного вектора на экран
    std::cout<<"Ваш вектор."<<std::endl;
    for (int i = 0; i < vector_size; i++) {
        std::cout<< students[i] <<std::endl;
    }

    return 0;
}

```


Результат работы программы:

```
selevit@gentoo code % ./a.out
Вводите фамилии студентов. По окончании ввода введите пустую строку
Иванов Иван Иванович
Петров Петр Петрович
Сидоров Александр Александрович
Попов Андрей Андреевич

Ваш вектор.
Иванов Иван Иванович
Петров Петр Петрович
Сидоров Александр Александрович
Попов Андрей Андреевич
selevit@gentoo code % █
```

Методы класса vector

Для добавления нового элемента в конец вектора используется метод `push_back()`. Количество элементов определяется методом `size()`. Для доступа к элементам вектора можно использовать квадратные скобки `[]`, также, как и для обычных массивов.

- `pop_back()` — удалить последний элемент
- `clear()` — удалить все элементы вектора
- `empty()` — проверить вектор на пустоту

Подробное описание всех методов `std::vector` (на английском) есть на [C++ Reference](#).

Следующий урок: **Наследование классов C++** →.

Тема 13. Наследование классов в C++

Наследование позволяет избежать дублирования лишнего кода при написании классов. Пусть в базе данных ВУЗа должна храниться информация о всех студентах и преподавателях. Представлять все данные в одном классе не получится, поскольку для преподавателей нам понадобится хранить данные, которые для студента не применимы, и наоборот.

- [Создание базового класса](#)
- [Наследование от базового класса](#)
- [Конструктор базового класса](#)
- [Создание объекта класса student](#)
- [Создание класса-наследника teacher](#)
- [Создание объекта класса teacher](#)
- [Когда нужно использовать конструктор](#)

Создание базового класса

Для решения этой задачи создадим базовый класс `human`, который будет описывать модель человека. В нем будут храниться имя, фамилия и отчество.

Создайте файл `human.h`:

```

// human.h
#ifndef HUMAN_H_INCLUDED
#define HUMAN_H_INCLUDED

#include<string>
#include<sstream>

class human {
public:
    // Конструктор класса human
    human(std::string last_name, std::string name, std::string second_name)
    {
        this->last_name = last_name;
        this->name = name;
        this->second_name = second_name;
    }

    // Получение ФИО человека
    std::string get_full_name()
    {
        std::ostringstream full_name;
        full_name << this->last_name << " "
        << this->name << " "
        << this->second_name;
        return full_name.str();
    }

private:
    std::string name; // имя
    std::string last_name; // фамилия
    std::string second_name; // отчество
};

#endif // HUMAN_H_INCLUDED

```

Наследование от базового класса

Теперь создайте новый класс student, который будет наследником класса human. Поместите его в файл student.h.

```

// student.h
#ifndef STUDENT_H_INCLUDED
#define STUDENT_H_INCLUDED

#include "human.h"
#include<string>
#include<vector>

class student : public human {
public:
    // Конструктор класса Student
    student(
        std::string last_name,
        std::string name,
        std::string second_name,
        std::vector<int> scores
    )
    {
        human(last_name, name, second_name);
        this->scores = scores;
    }
};

```

```

) : human(
    last_name,
    name,
    second_name
) {
this->scores = scores;
}

// Получение среднего балла студента
float get_average_score()
{
    // Общее количество оценок
    unsigned int count_scores = this->scores.size();
    // Сумма всех оценок студента
    unsigned int sum_scores = 0;
    // Средний балл
    float average_score;

    for (unsigned int i = 0; i < count_scores; ++i) {
        sum_scores += this->scores[i];
    }

    average_score = (float) sum_scores / (float) count_scores;
    return average_score;
}

private:
// Оценки студента
std::vector<int> scores;
};
#endif // STUDENT_H_INCLUDED

```

Функция `get_average_score` вычисляет среднее арифметическое всех оценок студента. Все публичные свойства и методы класса `human` будут доступны в классе `student`.

Конструктор базового класса

Для того, чтобы инициализировать конструктор родительского класса (в нашем случае — это сохранение имени, фамилии и отчества ученика), используется следующий синтаксис:

```

// Конструктор класса Student
student(
    // аргументы конструктора текущего класса
) : human(
    // инициализация конструктора родительского класса
) {
    // инициализация конструктора текущего класса
}

```

В конструктор класса `human` мы передаем инициалы человека, которые сохраняются в экземпляре класса. Для класса `students`, нам необходимо задать еще и список оценок студента. Поэтому конструктор `students` принимает все аргументы конструктора базового класса, а также дополнительные аргументы для расширения функционала:

```

// Конструктор класса Student
student(
    std::string last_name,

```

```

std::string name,
std::string second_name,
std::vector<int> scores
) : human(
    last_name,
    name,
    second_name
) {
this->scores = scores;
}

```

Список оценок студента хранится в векторе.

Создание объекта класса student

Реализуем пользовательский интерфейс для работы с классом student.

// main.cpp

```

#include<iostream>
#include<vector>

```

```

#include"human.h"
#include"student.h"

```

```

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

```

// Оценки студента
std::vector<int> scores;

// Добавление оценок студента в вектор
 scores.push_back(5);
 scores.push_back(3);
 scores.push_back(2);
 scores.push_back(2);
 scores.push_back(5);
 scores.push_back(3);
 scores.push_back(3);
 scores.push_back(3);
 scores.push_back(3);

// Создание объекта класса student
 student *stud = **new** student("Петров", "Иван", "Алексеевич", scores);

// Вывод полного имени студента (используется унаследованный метод класса human)
 std::cout<< stud->get_full_name() <<std::endl;

// Вывод среднего балла студента
 std::cout<<"Средний балл: "<< stud->get_average_score() <<std::endl;

```

return 0;
}

```

В этом примере мы написали программу, которая создает объект класса student, сохраняя в нем его имя, фамилию, отчество и список оценок.

После инициализации объекта, происходит вывод полного имени студента с помощью функции `get_full_name`. Эта функция была унаследована от базового класса `human`.

Затем программа вычисляет средний балл студента и выводит его на экран. Этим занимается функция `get_average_score`, которую мы описали внутри

```
selevit@gentoo codelive_class_inheritance % ./human
Петров Иван Алексеевич
Средний балл: 3.22222
selevit@gentoo codelive_class_inheritance %
```

класса `student`.

Мы реализовали часть функционала для нашей базы данных института (я конечно утрирую, когда оперирую столь серьезными высказываниями про настоящую базу данных :)

Создание класса-наследника `teacher`

Нужно создать еще один класс, в котором будут храниться данные преподавателей. Дадим ему название — `teacher`. Как вы уже поняли, мы не будем описывать все методы этого класса с нуля, а просто унаследуем его от класса `human`. Тогда, не нужно будет реализовывать хранение имени, фамилии и отчества препода. Это уже есть в базовом классе `human`.

Создайте файл `teacher.h`:

```
// teacher.h
#ifndef TEACHER_H_INCLUDED
#define TEACHER_H_INCLUDED

#include "human.h"
#include <string>

class teacher : public human {
// Конструктор класса teacher
public:
teacher(
std::string last_name,
std::string name,
std::string second_name,
// Количество учебных часов за семестр у преподавателя
unsigned int work_time
) : human(
last_name,
name,
second_name
) {
this->work_time = work_time;
}

// Получение количества учебных часов
unsigned int get_work_time()
{
return this->work_time;
}

private:
```

```
// Учебные часы
unsignedint work_time;
};
```

```
#endif// TEACHER_H_INCLUDED
```

У класса `teacher` появилось новое свойство — количество учебных часов, отведенное преподавателю на единицу времени (семестр). Весь остальной функционал наследуется от базового класса `human`. Если бы мы писали все с нуля, то одинакового кода бы получилось в разы больше, и его поддержка усложнилась бы на порядок.

Создание объекта класса `teacher`

Изменим содержимое файла `main.cpp`, чтобы проверить работу класса `teacher`.

```
#include<iostream>

#include"human.h"
#include"teacher.h"

intmain(int argc, char* argv[])
{

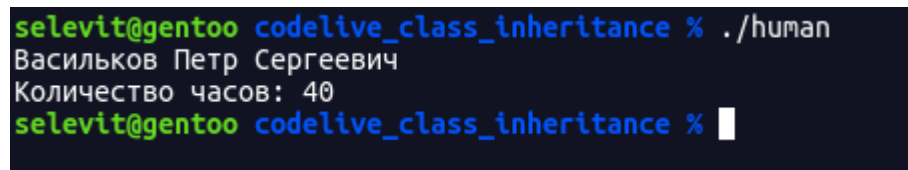
    // Количество учебных часов преподавателя
    unsignedint teacher_work_time = 40;

    teacher *tch = new teacher("Васильков", "Петр", "Сергеевич", teacher_work_time);

    std::cout<< tch->get_full_name() <<std::endl;
    std::cout<<"Количество часов: " << tch->get_work_time() <<std::endl;

    return0;
}
```

Если сборка программы прошла без ошибок, то результат работы программы будет таким:



```
selevit@gentoo codelive_class_inheritance % ./human
Васильков Петр Сергеевич
Количество часов: 40
selevit@gentoo codelive_class_inheritance %
```

Можно таким же образом создать класс, в котором будут храниться данные обслуживающего персонала или руководящего состава. Наследование используют, когда у каждой группы объектов есть общие параметры, но для каждой из этих групп нужно хранить более кастомные данные.

Также, мы можем создать класс, который будет описывать студента заочной формы обучения. Его мы унаследовали бы от класса `student`, добавив какие-либо дополнительные данные.

В класс `human` можно добавить еще больше свойств, которые будут описывать данные, имеющиеся у любого человека. Например, номер паспорта, дату рождения, прописку и место проживания.

Подобный подход позволяет в разы уменьшить дублирование кода в реальных проектах, и упростить его поддержку.

Когда нужно использовать конструктор

Если у класса много свойств — их совсем не обязательно задавать в конструкторе. Для сохранения отдельных свойств класса используют set-функции. Например, для сохранения номера паспорта, можно создать публичный метод `set_passport_number(std::string number)`, который будет принимать значение свойства и сохранять его в объекте, через переменную `this`.

Тема 14. Перегрузка функций в C++

Перегрузка функций в C++ используется, когда нужно сделать одно и то же действие с разными типами данных. Для примера, создадим простую функцию `max`, которая будет определять максимальное из двух целых чисел.

```
/* Функция max для целых чисел */  
intmax(int num1, int num2)  
{  
    if (num1 > num2)  
        return num1;  
    return num2;  
}
```

В эту функцию мы можем передавать только целочисленные параметры. Для того, чтобы сделать аналог этой функции для чисел с плавающей запятой, выполним перегрузку этой функции:

```
/* Функция max для чисел с плавающей запятой */  
doublemax(double num1, double num2)  
{  
    if (num1 > num2)  
        return num1;  
    return num2;  
}
```

Теперь, когда мы будем вызывать функцию `max` с целыми параметрами, то вызовется первая функция. А если с дробными — то вторая. Например:

```
// Здесь будет использоваться первый вариант функции max  
int imax = max(1, 10);  
// А здесь - второй  
double dmax = max(1.0, 20.0);
```

Задание: попробуйте написать функцию сортировки массива пузырьком для целочисленных массивов. А затем перегрузить эту же функцию для массивов типа `double`.

Исходный код примера из урока.

Следующий урок — перегрузка методов класса в C++.

Тема 15. Перегрузка методов класса в C++

Методы класса можно перегружать также, как и обычные функции. Особенно это удобно, когда нужно сделать несколько конструкторов, которые будут принимать разные параметры.

Например, попробуем создать основу класса `decimal`, который реализует длинную арифметику для чисел произвольной точности. В таких случаях, обычно хранят число внутри строки, а логика математических операций реализуется через написание соответствующих операторов класса.

Сделаем так, чтобы в конструктор этого класса можно было передавать и строку и число типа `double`.

```
// Передача в конструктор строки
decimal num1("10000000.999999");
// Передача числа
decimal num2(10000.0);
```

Для того, чтобы класс поддерживал такую универсальность, мы сделаем два разных конструктора для строки и числа:

```
/**
 * Представим, что этот класс реализует длинную арифметику для чисел любой
 * точности
 */
class decimal
{
public:
    /**
     * Конструктор, принимающий в качестве аргумента строку, содержащую число
     */
    decimal(string number)
    {
        clog<<"First constructor called\n";
    }

    /**
     * Конструктор принимает число типа double
     */
    decimal(double number)
    {
        clog<<"Second constructor called\n";
    }

private:
    string number;
};
```

При передаче строки будет вызван первый конструктор, а при передаче числа — второй.

Полный текст программы:

```
#include<iostream>
#include<string>

using namespace std;

/**
 * Представим, что этот класс реализует длинную арифметику для чисел любой точности
 */
class decimal
{
public:
```



```

/*
 * Конструктор, принимающий в качестве аргумента строку,
 * содержащее число
 */
decimal(string number)
{
    clog<<"First constructor called\n";
    this->number = number;
}

/**
 * Конструктор принимает число типа double
 */
decimal(double number)
{
/**
 * преобразуем double в строку с максимально возможной точностью
 * и записываем полученное значение в this->number
 */
    clog<<"Second constructor called\n";
}

private:
string number;
};

int main()
{
    // Будет вызван первый конструктор
    decimal num1("10000000.999999");
    // Будет вызван второй конструктор
    decimal num2(10000.0);
    cin.get();
    return 0;
}

```

Конечно, наш класс пока ничего не делает, потому что реализация длинной арифметики выходит за рамки данной статьи. Но на этом примере можно понять, когда может быть полезно использовать перегрузку методов класса.

Задание: попробуйте написать класс `student`, в конструктор которого можно будет передавать либо его имя и фамилию, либо имя и год рождения. При передаче года рождения, должен считаться примерный возраст студента. Пример использования класса:

```

student stud1("Иван", "Иванов");
student stud2("Иван", 1990);

```

Тема 16. Определение и перегрузка операторов класса в C++

В C++ можно определять пользовательские операторы для собственных типов данных. Оператор определяется, как обычная функция-член класса, только после определения возвращаемого типа ставится ключевое слово `operator`.

Пример определения оператора сложения:

```
intoperator+ (int value) { return number + value; }
```

Оператор может быть унарным или бинарным. Унарный оператор не принимает никаких аргументов. Например, оператор отрицания — «!». Бинарный оператор принимает дополнительный параметр. Например, в случае со сложением, принимается второе слагаемое.

Чтобы прояснить картину, попробуем написать класс `simple_fraction`, который будет описывать простую дробь с целыми числителем и знаменателем. И определим операторы сложения, вычитания, умножения и деления для этого класса.

```
/*
 * Класс, описывающий простую дробь
 */
class simple_fraction
{
public:
    simple_fraction(int numerator, int denominator)
    {
if (denominator == 0) // Ошибка деления на ноль
throwstd::runtime_error("zero division error");
this->numerator = numerator;
this->denominator = denominator;
    }

    // Определение основных математических операций для простой дроби
    doubleoperator+ (int val) { return number() + val; } // Сложение
    doubleoperator- (int val) { return number() - val; } // Вычитание
    doubleoperator* (int val) { return number() * val; } // Умножение
    doubleoperator/ (int val) // Деление
    {
if (val == 0) {
throwstd::runtime_error("zero division error");
    }
returnnumber() / val;
    }

    // Получение значения дроби в виде обычного double-числа
    doublenumber(){ return numerator / (double) denominator; }
private:
    int numerator; // Числитель
    int denominator; // Знаменатель
};
```

Для операции деления, мы также сделали проверку деления на ноль.

Пример использования класса `simple_fraction`:

```
// Простая дробь 2/3
simple_fraction fr(2, 3);

double sum = fr + 10; // сумма
double diff = fr - 10; // разность
double factor = fr * 10; // произведение
double div = fr / 10; // частное
```

Операторы можно перегружать так же, как и обычные функции-члены класса. Например, можно перегрузить оператор сложения для двух простых дробей, который будет

возвращать новую простую дробь. Тогда, нам придется привести дроби к общему знаменателю и вернуть другую простую дробь.

Задание: усовершенствуйте класс `simple_fraction`. Перегрузите операторы сложения, вычитания, умножения и деления так, чтобы можно было производить операции над двумя простыми дробями и получать новую простую дробь. Реализуйте приведение двух дробей к общему знаменателю.

Пример использования будущего класса:

```
simple_fraction fr1(2, 3);
simple_fraction fr2(3, 4);

// 2/3 + 3/4 — это 17/12
simple_fraction sum = fr1 + fr2;
```

Тема 15. Раздельная компиляция программ на C++

Когда мы пишем программу на C/C++ в одном файле, проблем обычно не возникает. Они ждут того момента, когда исходный текст необходимо разбить на несколько файлов. В этой статье я постараюсь рассказать, как это сделать правильно.

Термины

Пара слов о терминах. Ниже даны определения терминов так, как они используются в данной статье. В некоторых случаях эти определения имеют более узкий смысл, чем общепринятые. Это сделано намеренно, дабы не утонуть в деталях и лишних уточнениях.

Исходный код — программа, написанная на языке программирования, в текстовом формате. А также текстовый файл, содержащий исходный код.

Компилятор — программа, выполняющая компиляцию (неожиданно! не правда ли?). На данный момент среди начинающих наиболее популярными компиляторами C/C++ являются *GNU g++* (и его порты под различные ОС) и *MS Visual Studio C++* различных версий. Подробнее см. в Википедии статьи: [Компиляторы](#), [Компиляторы C++](#).

Компиляция — преобразование исходного кода в объектный модуль.

Объектный модуль — двоичный файл, который содержит в себе особым образом подготовленный исполняемый код, который может быть объединён с другими объектными файлами при помощи редактора связей (компоновщика) для получения готового исполняемого модуля, либо библиотеки. ([подробности](#))

Компоновщик (редактор связей, линкер, сборщик) — это программа, которая производит компоновку («линковку», «сборку»): принимает на вход один или несколько объектных модулей и собирает по ним исполнимый модуль. ([подробности](#))

Исполняемый модуль (исполняемый файл) — файл, который может быть запущен на исполнение процессором под управлением операционной системы. ([подробности](#))

Препроцессор — программа для обработки текста. Может существовать как отдельная программа, так и быть интегрированной в компилятор. В любом случае, входные и

выходные данные для препроцессора имеют **текстовый формат**. Препроцессор преобразует текст в соответствии с *директивами препроцессора*. Если текст не содержит директив препроцессора, то текст остаётся без изменений. Подробнее см. в Википедии: [Препроцессор](#) и [Препроцессор Си](#).

IDE (англ. Integrated Development Environment) — интегрированная среда разработки. Программа (или комплекс программ), предназначенных для упрощения написания исходного кода, отладки, управления проектом, установки параметров компилятора, линкера, отладчика. Важно не путать IDE и компилятор. Как правило, компилятор самодостаточен. В состав IDE компилятор может не входить. С другой стороны с некоторыми IDE могут быть использованы различные компиляторы. ([подробности](#))

Объявление — описание некой сущности: сигнатура функции, определение типа, описание внешней переменной, шаблон и т.п. Объявление уведомляет компилятор о её существовании и свойствах.

Определение — реализация некой сущности: переменная, функция, метод класса и т.п. При обработке определения компилятор генерирует информацию для объектного модуля: исполняемый код, резервирование памяти под переменную и т.д.

От исходного кода к исполняемому модулю

Создание исполняемого файла издавна производилось в три этапа: (1) обработка исходного кода препроцессором, (2) компиляция в объектный код и (3) компоновка объектных модулей, включая модули из объектных библиотек, в исполняемый файл. Это классическая схема для компилируемых языков. (Сейчас уже используются и другие схемы.)

Часто *компиляцией* программы называют весь процесс преобразования исходного кода в исполняемый модуль. Что неправильно. Обратите внимание, что в IDE этот процесс называется *построение (build)* проекта.

IDE обычно скрывают три отдельных этапа создания исполняемого модуля. Они проявляются только в тех случаях, когда на этапе препроцессинга или компоновки обнаруживаются ошибки.

Итак, допустим, у нас есть программа на C++ «*Hello, World!*»:

```
#include<iostream>
```

```
intmain(){  
    std::cout<<"Hello, World!\n";  
}
```

Сначала исходный код обрабатывается препроцессором. Препроцессор находит директиву `#include <iostream>`, ищет файл *iostream* и заменяет директиву текстом из этого файла, попутно обрабатывая все директивы препроцессора во включаемом тексте.

Файл, указанный в директиве `#include`, в данном случае является *заголовочным файлом* (или «хеадером», «хидером», «заголовком»). Это обычный текстовый файл, содержащий **объявления** (объявления типов, прототипы функций, шаблоны, директивы препроцессора и т.п.). После **текстуального** включения заголовочного файла в текст программы (или модуля) становится возможным использование в тексте программы всего того, что описано в этом заголовочном файле.

Затем результат работы препроцессора передаётся компилятору. Компилятор производит весь положенный комплекс работ: от синтаксического разбора и поиска ошибок до создания объектного файла (понятно, что если имеются синтаксические ошибки, то объектный файл не создаётся). В объектном файле обычно имеется *таблица внешних ссылок* — некая таблица, в которой, в частности, перечислены имена подпрограмм, которые используются в объектном модуле, но код которых отсутствует в данном объектном модуле. Эти подпрограммы *внешние* по отношению к модулю.

Исходный код, который может быть откомпилирован, называется **единицей компиляции**. Наша программа содержит одну единицу компиляции.

Что бы получить нормальный исполняемый модуль, необходимо «разрешить» внешние ссылки. Т.е. добавить в исполняемый модуль код отсутствующих подпрограмм и настроить соответствующим образом все ссылки на этот код. Этим занимается компоновщик. Он анализирует таблицу внешних ссылок объектного модуля, ищет в объектных библиотеках недостающие модули, копирует их в исполняемый модуль и настраивает ссылки. После этого исполняемый модуль готов.

Библиотека (объектная библиотека) — это набор откомпилированных подпрограмм, собранных в единый файл определённой структуры. Подключение библиотеки происходит на этапе компоновки исполняемого файла из объектных файлов (т.е. из тех файлов, которые получаются в результате компиляции исходного текста программы).

Необходимые объектные библиотеки входят в комплект поставки компилятора. В комплект поставки библиотек (любых) входит набор заголовочных файлов, которые содержат объявления, необходимые компилятору.

Если исходный код программы разделён на несколько файлов, то процесс компиляции и сборки происходит аналогично. Сначала все единицы компиляции по отдельности компилируются, а затем компоновщик собирает полученные объектные модули (с подключением библиотек) в исполняемый файл. Собственно, этот процесс и называется **раздельной компиляцией**.

Разделение текста программы на модули

Разделение исходного текста программы на несколько файлов становится необходимым по многим причинам:

1. С большим текстом просто неудобно работать.
2. Разделение программы на отдельные модули, которые решают конкретные подзадачи.
3. Разделение программы на отдельные модули, с целью повторного использования этих модулей в других программах.
4. Разделение интерфейса и реализации.

Я намеренно использовал слово «модуль», поскольку модулем может быть как класс, так и набор функций — вопрос используемой технологии программирования.

Как только мы решаем разделить исходный текст программы на несколько файлов, возникают две проблемы:

1. Необходимо от простой компиляции программы перейти к раздельной. Для этого надо внести соответствующие изменения либо в последовательность действий при построении приложения вручную, либо внести изменения в командные или make-файлы, автоматизирующие процесс построения, либо внести изменения в проект IDE.

2. Необходимо решить каким образом разбить текст программы на отдельные файлы.

Первая проблема — чисто техническая. Она решается чтением руководств по компилятору и/или линкеру, утилите *make* или IDE. В самом худшем случае просто придётся проштудировать все эти руководства. Поэтому на решении этой проблемы мы останавливаться не будем.

Вторая проблема — требует гораздо более творческого подхода. Хотя и здесь существуют определённые рекомендации, несоблюдение которых приводит либо к невозможности собрать проект, либо к трудностям в дальнейшем развитии проекта.

Во-первых, нужно определить какие части программы выделить в отдельные модули. Что бы это получилось просто и естественно, программа должна быть правильно спроектирована. Как правильно спроектировать программу? — на эту тему написано много больших и правильных книг. Обязательно поищите и почитайте книги по методологии программирования — это очень полезно. А в качестве краткой рекомендации можно сказать: вся программа должна состоять из слабо связанных фрагментов. Тогда каждый такой фрагмент может быть естественным образом преобразован в отдельный модуль (единицу компиляции). Обратите внимание, что под «фрагментом» подразумевается не просто произвольный кусок кода, а функция, или группа логически связанных функций, или класс, или несколько тесно взаимодействующих классов.

Во-вторых, нужно определить интерфейсы для модулей. Здесь есть вполне чёткие правила.

Интерфейс и реализация

Когда часть программы выделяется в модуль (единицу компиляции), остальной части программы (а если быть точным, то компилятору, который будет обрабатывать остальную часть программы) надо каким-то образом объяснить что имеется в этом модуле. Для этого служат *заголовочные файлы*.

Таким образом, модуль состоит из двух файлов: заголовочного (интерфейс) и файла реализации.

Заголовочный файл, как правило, имеет расширение *.h* или *.hpp*, а файл реализации — *.cpp* для программ на C++ и *.c*, для программ на языке C. (Хотя в STL включаемые файлы вообще без расширений, но, по сути, они являются заголовочными файлами.)

Заголовочный файл должен содержать все объявления, которые должны быть видны снаружи. Объявления, которые не должны быть видны снаружи, делаются в файле реализации.

Что может быть в заголовочном файле

Правило 1. Заголовочный файл может содержать только объявления. Заголовочный файл не должен содержать определения.

То есть, при обработке содержимого заголовочного файла компилятор не должен генерировать информацию для объектного модуля.

Единственным «исключением» из этого правила является определение метода в объявлении класса. Но по стандарту языка, если метод определён в объявлении класса, то для этого метода используется инлайновая подстановка. Поэтому, такое объявление не

порождает исполняемого кода — код будет генерироваться компилятором только при вызове этого метода.

Аналогичная ситуация и с объявлением переменных-членов класса: код будет порождаться при создании экземпляра этого класса.

Правило 2. Заголовочный файл должен иметь механизм защиты от повторного включения.

Защита от повторного включения реализуется директивами препроцессора:

```
#ifndef SYMBOL
#define SYMBOL
```

```
// набор объявлений
```

```
#endif
```

Для препроцессора при первом включении заголовочного файла это выглядит так: поскольку условие "символ SYMBOL не определён" (`#ifndef SYMBOL`) истинно, определить символ SYMBOL (`#define SYMBOL`) и обработать все строки до директивы `#endif`. При повторном включении — так: поскольку условие "символ SYMBOL не определён" (`#ifndef SYMBOL`) ложно (символ был определён при первом включении), то пропустить всё до директивы `#endif`.

В качестве SYMBOL обычно применяют имя самого заголовочного файла в верхнем регистре, обрамлённое одинарными или двоянными подчеркиками. Например, для файла *header.h* традиционно используется `#define __HEADER_H__`. Впрочем, символ может быть любым, но обязательно уникальным в рамках проекта.

В качестве альтернативного способа может применяться директива `#pragma once`. Однако преимущество первого способа в том, что он работает на любых компиляторах.

Заголовочный файл сам по себе не является единицей компиляции.

Что может быть в файле реализации

Файл реализации может содержать как определения, так и объявления. Объявления, сделанные в файле реализации, будут лексически локальны для этого файла. Т.е. будут действовать только для этой единицы компиляции.

Правило 3. В файле реализации должна быть директива включения соответствующего заголовочного файла.

Понятно, что объявления, которые видны снаружи модуля, должны быть также доступны и внутри.

Правило также гарантирует соответствие между описанием и реализацией. При несовпадении, допустим, сигнатуры функции в объявлении и определении компилятор выдаст ошибку.

Правило 4. В файле реализации не должно быть объявлений, дублирующих объявления в соответствующем заголовочном файле.

При выполнении **Правила 3**, нарушение **Правила 4** приведёт к ошибкам компиляции.

Практический пример

Допустим, у нас имеется следующая программа:

main.cpp

```
#include<iostream>
```

```
using namespace std;
```

```
const int cint = 10; // глобальная константа
```

```
int global_var = 0; // глобальная переменная
```

```
int module_var = 0; // глобальная переменная для func1 и func2
```

```
int func1(){  
    ++global_var;  
    return ++module_var;  
}
```

```
int func2(){  
    ++global_var;  
    return --module_var;  
}
```

```
class CClass {  
public:  
    CClass() : priv(cint) { ++counter; }  
    ~CClass() { --counter; }  
    void change(int arg);  
    int get_priv() const;  
    int get_counter() const;  
private:  
    int priv;  
    static int counter;  
};
```

```
int CClass::counter = 0;
```

```
void CClass::change(int arg) {  
    priv += arg;  
}
```

```
int CClass::get_priv() const {  
    return priv;  
}
```

```
int CClass::get_counter() const {  
    return counter;  
}
```

```
int main()  
{  
    int balance;  
    balance = func1();  
    balance = func2();
```



```
cout<<"balance: "<< balance <<" counter: "<< global_var << endl;
```

```
    CClass c1, c2;  
    if (c1.get_priv() == cint)  
        cout<<"Ok"<< endl;  
    cout<< c2.get_counter() << endl;  
    return 0;  
}
```

Эта программа не является образцом для подражания, поскольку некоторые моменты идеологически неправильны, но, во-первых, ситуации бывают разные, а во-вторых, для демонстрации эта программа подходит очень неплохо.

Итак, что у нас имеется?

- 1) глобальная константа `cint`, которая используется и в классе, и в `main`;
- 2) глобальная переменная `global_var`, которая используется в функциях `func1`, `func2` и `main`;
- 3) глобальная переменная `module_var`, которая используется только в функциях `func1` и `func2`;
- 4) функции `func1` и `func2`;
- 5) класс `CClass`;
- 6) функция `main`.

Вроде вырисовываются три единицы компиляции: (1) функция `main`, (2) класс `CClass` и (3) функции `func1` и `func2` с глобальной переменной `module_var`, которая используется только в них.

Не совсем понятно, что делать с глобальной константой `cint` и глобальной переменной `global_var`. Первая тяготеет к классу `CClass`, вторая — к функциям `func1` и `func2`. Однако предположим, что планируется и эту константу, и эту переменную использовать ещё в каких-то, пока не написанных, модулях программы. Поэтому прибавится ещё одна единица компиляции.

Теперь пробуем разделить программу на модули.

Сначала, как наиболее связанные сущности (используются во многих местах программы), выносим глобальную константу `cint` и глобальную переменную `global_var` в отдельную единицу компиляции.

globals.h

```
#ifndef __GLOBALS_H_  
#define __GLOBALS_H_  
  
const int cint = 10;          // глобальная константа  
extern int global_var;        // глобальная переменная  
  
#endif // __GLOBALS_H_
```

globals.cpp

```
#include "globals.h"  
  
int global_var = 0; // глобальная переменная
```

Обратите внимание, что глобальная переменная в заголовочном файле имеет спецификатор `extern`. При этом получается *объявление* переменной, а не её *определение*.

Такое описание означает, что где-то существует переменная с таким именем и указанным типом. А определение этой переменной (с инициализацией) помещено в файл реализации. Константа описана в заголовочном файле.

С объявлением констант в заголовочном файле существует одна тонкость. Если константа тривиального типа, то её можно объявить в заголовочном файле. В противном случае она должна быть определена в файле реализации, а в заголовочном файле должно быть её объявление (аналогично, как для переменной). «Тривиальность» типа зависит от стандарта (см. описание того стандарта, который используется для написания программы).

Также обратите внимание (1) на защиту от повторного включения заголовочного файла и (2) на включение заголовочного файла в файле реализации.

Затем выносим в отдельный модуль функции `func1` и `func2` с глобальной переменной `module_var`. Получаем ещё два файла:

funcs.h

```
#ifndef __FUNCS_H__
#define __FUNCS_H__

intfunc1();
intfunc2();

#endif// __FUNCS_H__
```

funcs.cpp

```
#include "funcs.h"
#include "globals.h"

int module_var = 0;    // глобальная переменная для func1 и func2

intfunc1(){
    ++global_var;
    return ++module_var;
}

intfunc2(){
    ++global_var;
    return --module_var;
}
```

Поскольку переменная `module_var` используется только этими двумя функциями, её объявление в заголовочном файле отсутствует. Из этого модуля «на экспорт» идут только две функции.

В функциях используется переменная из другого модуля, поэтому необходимо добавить `#include "globals.h"`.

Наконец выносим в отдельный модуль класс `CClass`:

CClass.h

```
#ifndef __CCLASS_H__
#define __CCLASS_H__
```

```

class CClass {
public:
    CClass();
    ~CClass();
    void change(int arg);
    int get_priv() const;
    int get_counter() const;
private:
    int priv;
    static int counter;
};

#endif // __CCLASS_H__

```

CClass.cpp

```

#include "CClass.h"
#include "globals.h"

int CClass::counter = 0;

CClass::CClass() : priv(cint) {
    ++counter;
}

CClass::~CClass() {
    --counter;
}

void CClass::change(int arg) {
    priv += arg;
}

int CClass::get_priv() const {
    return priv;
}

int CClass::get_counter() const {
    return counter;
}

```

Обратите внимание на следующие моменты.

- (1) Из объявления класса убрали определения тел функций (методов). Это сделано по идеологическим причинам: интерфейс и реализация должны быть разделены (для возможности изменения реализации без изменения интерфейса). Если впоследствии будет необходимость сделать какие-то методы инлайновыми, это всегда можно сделать с помощью спецификатора.
- (2) Класс имеет статический член класса. Т.е. для всех экземпляров класса эта переменная будет общей. Её инициализация выполняется не в конструкторе, а в глобальной области модуля.
- (3) В файл реализации добавлена директива `#include "globals.h"` для доступа к константе `cint`.

Классы практически всегда выделяются в отдельные единицы компиляции.

В файле *main.cpp* оставляем только функцию *main*. И добавляем необходимые директивы включения заголовочных файлов.

main.cpp

```
#include<iostream>
#include"funcs.h"
#include"CClass.h"
#include"globals.h"

usingnamespacestd;

intmain()
{
    int balance;
    balance = func1();
    balance = func2();
    cout<<"balance: "<< balance <<" counter: "<< global_var << endl;

    CClass c1, c2;
    if (c1.get_priv() == cint)
        cout<<"Ok"<< endl;
    cout<< c2.get_counter() << endl;
    return0;
}
```

Последний шаг: необходимо изменить «проект» построения программы так, что бы он отражал изменившуюся структуру файлов исходного кода. Детали этого шага зависят от используемой технологии построения программы и используемого ПО. Но в любом случае сначала должны быть откомпилированы четыре единицы компиляции (четыре *src*-файла), а затем полученные объектные файлы должны быть обработаны компоновщиком для получения исполняемого файла.

Типичные ошибки

Ошибка 1. Определение в заголовочном файле.

Эта ошибка в некоторых случаях может себя не проявлять. Например, когда заголовочный файл с этой ошибкой включается только один раз. Но как только этот заголовочный файл будет включён более одного раза, получим либо ошибку компиляции «*многократное определение символа ...*», либо ошибку компоновщика аналогичного содержания, если второе включение было сделано в другой единице компиляции.

Ошибка 2. Отсутствие защиты от повторного включения заголовочного файла.

Тоже проявляет себя при определённых обстоятельствах. Может вызывать ошибку компиляции «*многократное определение символа ...*».

Ошибка 3. Несовпадение объявления в заголовочном файле и определения в файле реализации.

Обычно возникает в процессе редактирования исходного кода, когда в файл реализации вносятся изменения, а про заголовочный файл забывают.

Ошибка 4. Отсутствие необходимой директивы *#include*.

Если необходимый заголовочный файл не включён, то все сущности, которые в нём объявлены, останутся неизвестными компилятору. Вызывает ошибку компиляции «*не определён символ ...*».

Ошибка 5. Отсутствие необходимого модуля в проекте построения программы.

Вызывает ошибку компоновки «*не определён символ ...*». Обратите внимание, что имя символа в сообщении компоновщика почти всегда отличается от того, которое определено в программе: оно дополнено другими буквами, цифрами или знаками.

Ошибка 6. Зависимость от порядка включения заголовочных файлов.

Не совсем ошибка, но таких ситуаций следует избегать. Обычно сигнализирует либо об ошибках в проектировании программы, либо об ошибках при разделении исходного кода на модули.

Заключение

Мы рассмотрели все случаи, возникающие при раздельной компиляции. Бывают ситуации, когда разделение программы или большого модуля на более мелкие кажется невозможным. Обычно это бывает, когда программа плохо спроектирована (в данном случае, части кода имеют сильные взаимные связи). Конечно, можно приложить дополнительные усилия и всё-таки разделить код на модули (или оставить как есть), но эту мозговую энергию лучше потратить более эффективно: на изменение структуры программы. Это принесёт в дальнейшем гораздо большие дивиденды, чем просто силовое решение.