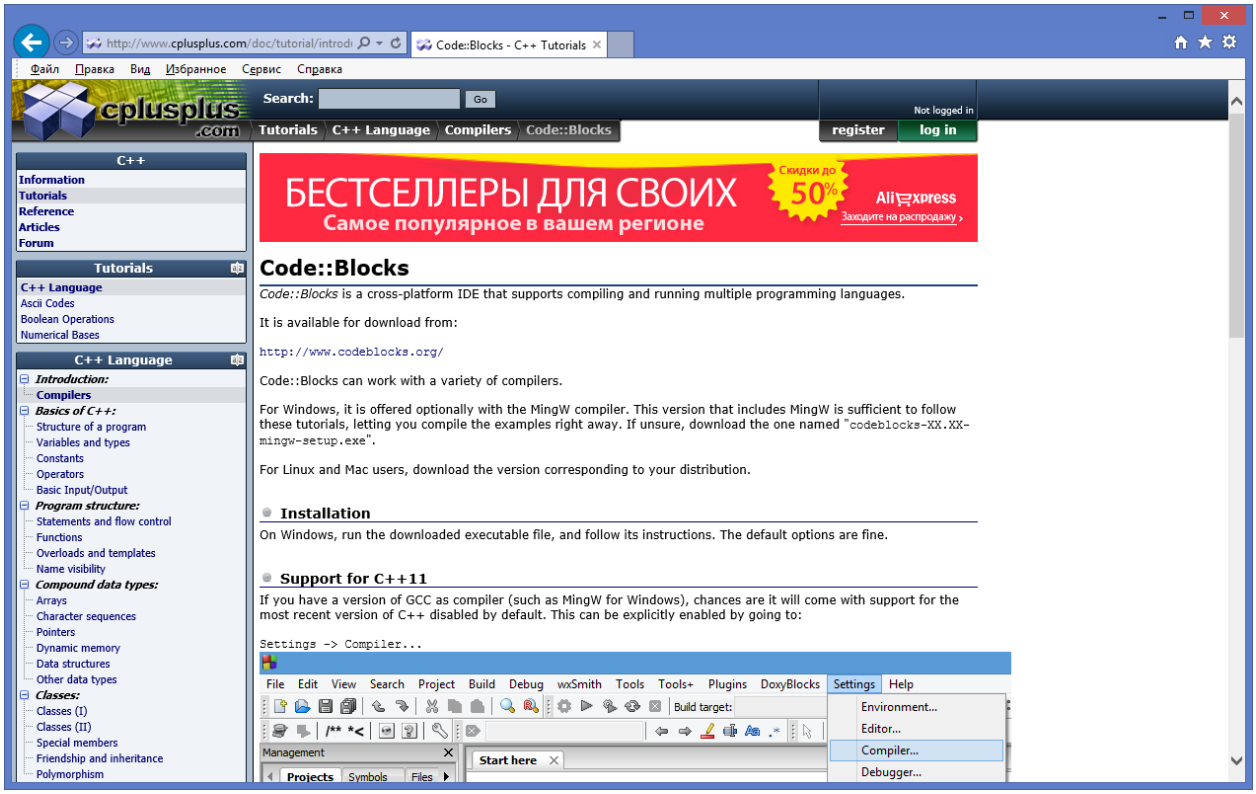
В.Н. Гололобов

О ПЕРЕВОДЕ ОДНОГО РУКОВОДСТВА



Москва -2015

Я не программист. И я не готов к рассказу о программировании ни как о философском учении, ни как об инструкции по применению. Это так. Но сегодня многие радиолюбители интересуются программированием микроконтроллеров, что с моей точки зрения подразумевает обязательное знакомство с языком программирования высокого уровня. Справедливо говорят, что нет плохих языков программирования, есть плохие программисты, поэтому выбор языка – дело каждого.

Один из популярных сегодня языков программирования – это Си. А если вспомнить, что часто устройства на базе микроконтроллеров, как свою часть используют компьютер, то С++.

Занимаясь сочинительством, я не занимаюсь программированием, поэтому забываю то немногое, что когда-то знал. Поэтому время от времени пытаюсь восстановить потери.

В поисках бесплатной среды разработки я наткнулся на упоминание программы Code::Blocks на сайте, который показан на обложке, а, просматривая написанное, на рассказ о языке С++. Мне показалось, что этот рассказ наиболее практическое из того, что я встречал раньше. Не знаю автора, но надеюсь, он не будет против перевода его рассказа. Но ещё больше надеюсь, что его рассказ будет полезен тем, кто начинает знакомство с языком программирования Си.

Оглавление

Вступление

Всё, что написано мной в этом вступлении, вы прочтёте в руководстве ниже, где оно описано и полнее, и лучше. Но десяток страниц, как мне кажется, способны одолеть и самые начинающие, и самые нетерпеливые, чтобы определиться, стоит ли читать остальное. Итак.

Любая программа – это последовательность действий, которые должен выполнить процессор, чтобы реализовать нашу задумку. Задумаем простую вещь – сложим два числа.

Я использую в качестве среды для работы Code::Blocks, поскольку с этой программы всё и начиналось. Позже в самом руководстве можно увидеть, что после запуска программы можно выбрать вид приложения. Я выбираю сейчас консольное приложение. После завершения предварительных настроек (имя проекта, имя файла и т.п.) в редакторе обнаруживается заготовка с программой, которую все называют «Привет, мир!». Я уберу эту программу и напишу свой код программы, назовём её «калькулятор»:

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

int main()
{
    int a;
    a = 2+3;
    cout<<a;
    return 0;
}
```

Мой текст программы выделен цветом. Проверим, работает ли программа, запустив её на компиляцию и работу. Для этого (после правки текста) служат пункты основного меню *Build->Build*, затем *Run*, или кнопки инструментальной панели.

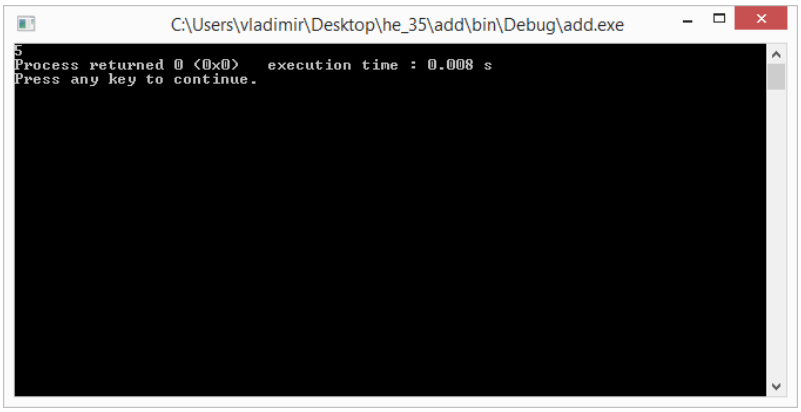


Рис. 1. Работа программы в консоли

Кстати, в записи a = 2+3 знак равенства не означает равенства, как это принято в математике. В этом можно убедиться, если изменить выражение на следующее: a = a + 1. С точки зрения арифметики это неправильная запись, а программа будет работать. Причина в том, что знак равенства в языке Си – это оператор присваивания. То есть, каждое из выражений можно прочитать так: «a есть…» или «a присваивается значение…».

В верхней части консоли можно увидеть число 5. Это и есть результат работы программы. В коде программы объявлена переменная «a», которая приобретает значение после сложения двух чисел, а затем выводится на экран. Вот такой у нас калькулятор.

Думаю, никто не будет спорить, что это калькулятор – он выполняет сложение. Но никому и не будет нужен такой калькулятор, выполняющий только одну единственную операцию. Первая причина убогости программы в том, как я использовал данные (они же операнды в нашем случае). Я уже говорил, что любая программа выполняет некоторые действия. Сейчас программа должна сложить два числа. Эти числа я записал как постоянные значения. В любой программе вы должны объявить многое, чтобы компилятор понимал вас. Для результата операции я объявил (или, если угодно, декларировал) переменную, с которой процессор будет выполнять процесс или оперировать, и я (и не только я) должен выбрать тип данных для этой переменной. Отчего я говорю о переменной? Пока операция не будет выполнена, этот объект не получит значения. Но в программе не может быть неясностей, поэтому любые объекты должны иметь какие-то значения. По умолчанию переменной будет присвоено значение (или мы при желании можем его присвоить самостоятельно). После операции это значение изменится. Вот отчего мы имеем дело с переменной, а не с постоянной. Есть фундаментальные типы данных, с которыми работают все языки программирования. Разные типы данных связаны с тем, что вся программа, все операнды в любом компьютере хранятся в памяти, где базовой ячейкой памяти с момента создания оперативной памяти компьютеров была восьми битовая конструкция. Восемь бит – это байт. Перебирая все возможные комбинации для восьми бит, вы обнаружите, что максимальное число, хранимое в такой ячейке памяти, будет 255. Согласитесь, что калькулятор, где вы можете сложить числа с суммой не более 255, это совсем плохой калькулятор. Поэтому для хранения данных и результатов операций потребовалось последовательно соединять ячейки памяти. Минимальная цепочка – это две ячейки памяти. Они предназначены для хранения целых чисел в диапазоне до 64 тысяч. Такой калькулятор уже позволяет использовать его для многих целей. Данные, которые мы называли целыми числами, обозначаются словом int (часть слова integer, целое). И, наконец, наше число, как результат операции, должно быть как-то обозначено или идентифицировано. То есть, наша переменная должна иметь имя (идентификатор). В выборе имени мы ограничены только условиями, заявляемыми компилятором. Компилятор - это транслятор (переводчик) с языка Си на исполняемый код программы, состоящий из нулей и единиц – единственный язык, который понимает компьютер. Во многих случаях есть ограничения на длину имени, во многих случаях есть и другие ограничения, о которых вы прочитаете позже. А сейчас я несколько усложню программу, записав такой код;

```
const int a = 2;
const int b = 3;
int c = 0;
c = a+b;
```

Усложнение коснулось только того, что я заменил два числа их представлением в виде такого объекта, как константа. Константами эти два числа и являются. Но для их объявления константами потребовалось использовать ключевое слово *const*. Ключевое, поскольку оно «открывает» их сущность. Кроме того, я явно задал первоначальное значение переменной «c». Вы можете убедиться, что результат работы программы не изменился. Спрашивается, зачем усложнять программу? Когда программа столь проста, смысла её усложнять, я согласен, никакого. Но в

большой программе, если вам потребуется изменить значения постоянных, проще это сделать в одном месте, а не бегать по всей программе в поисках мест, где вы используете ваши постоянные.

Пока мы не ушли от простейшей программы, давайте заменим тип наших чисел. Оба операнда поместятся в байт. Пусть это будет байт. Разные компиляторы поддерживают разное обозначение, применим тип *char*, предназначенный для хранения символов. Символы помещаются в один байт. У нас не символы, а числа, но и символы, в конечном счёте, имеют вид чисел, поскольку компьютер понимает только числа.

```
const char a = 200;
const char b = 100;
char c = 0;
c = a+b;
cout<<c;
```

Программа будет исправно оттранслирована, но вы-то знаете, что результат не поместится в байт. Поэтому следует осторожно и внимательно относится к типам данных.

Однако даже для простого калькулятора, выполняющего только сложение, наша программа слишком плоха. Давайте исправим её. Оба операнда мы будем вводить с клавиатуры. Для этого кроме команды вывода на экран (`cout<<c;`) добавим две команды ввода с клавиатуры:

```
int a = 0;
int b = 0;
int c = 0;
cin>>a;
cin>>b;
c = a+b;
cout<<c;
```

Теперь программа позволяет нам ввести два числа (каждое ожидает нажатия клавиши **Enter**), сложить их, и выводит результат. Вот так:

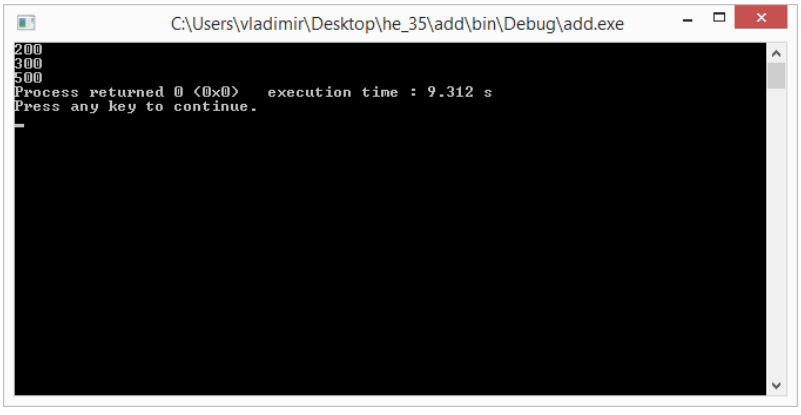


Рис. 2. Работа модифицированной программы

Наш калькулятор немного повзрослел, но остаётся очень неудобным даже для сложения двух целых чисел. Сложив два числа, мы вынуждены запускать программу вновь, если нам захочется сложить ещё два числа. Чтобы этого избежать, мы может использовать такой элемент программы, как цикл. Он есть в любом языке программирования. Более того, есть разные виды циклов. С

ними вы тоже познакомитесь в руководстве далее, а сейчас используем цикл *while*. Цикл означает, что блок программы, заключённый в теле цикла, будет повторяться до тех пор, пока выполняется условие работы цикла. В нашем случае мы позже озаботимся вопросом, как прекратить работу цикла, а сейчас запишем программу в таком виде:

```
int a = 0;
int b = 0;
int c = 0;
while (1)  {
cin>>a;
cin>>b;
c = a+b;
cout<<c;   }
```

Запустив программу после компиляции, вы увидите следующее (первое сложение - это 10+20, второе 100+200):

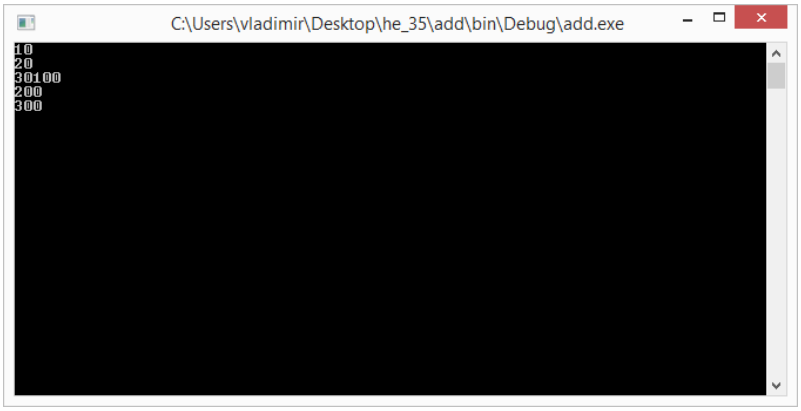


Рис. 3. Работа программы в бесконечном цикле

Есть небольшое неудобство – после сложения двух первых чисел и вывода результата следующее число записывается за результатом сложения. Это исправляется добавлением специального символа, который означает, что следует перейти на новую строку «\n». Выглядит это так (часть кода):

```
cin>>a;
cin>>b;
c = a+b;
cout<<c;
cout<<"\n";
```

Осталось ещё одно неудобство – бесконечный цикл. Конечно, есть правильные решения этой проблемы, но об этом вы узнаете из руководства, а сейчас можно либо оставить вопрос открытым, либо попробовать обойти проблему. В качестве обходного манёвра я предлагаю исправить проблему так:

```
int a = 1;
int b = 0;
int c = 0;
while (a != 0)
    {
cin>>a;
```

```
    cin>>b;
    c = a+b;
    cout<<c;
        }
```

Идея в том, что никому не нужно сложение с нулём, а два символа «!=» означают «не равно», то есть, пока а не равно нулю цикл выполняется.

Если какой-то блок кода часто выполняется в программе, его удобно объединить в некоторый объект, который в Си называется функцией. Блок кода, которому мы даём имя, будет выполняться, когда мы его вызываем в программе. Подобный блок кода мы должны показать в программе до её начала. Это становится актуально, когда мы задумываем кроме сложения на нашем калькуляторе выполнять и вычитание. В этом случае последовательность операций по этим операциям мы объединим в два блока, а в программе будем вызывать функции. Начнём, впрочем, с одной операции, отложив вторую операцию и цикл до следующих вариантов программы.

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

    int a = 0;
    int b = 0;
    int c = 0;

int add ()
{
    c = a+b;
    return c;
}

int main()
{
    cin>>a;
    cin>>b;
    add();
    cout<<c;
    cout<<"\n";
    return 0;
}
```

Итак, мы выделили часть инструкций в блок, который назвали *add*. По правилам написания кода перед именем блока обозначен тип возвращаемого функцией значения, а за именем пара скобок. Сами инструкции заключены в фигурные скобки. В основной программе мы вызываем функцию, записывая её имя.

Есть, правда, одно неудобство – функцию мы должны описать до основной программы; но мы используем в функции переменные, которые вынуждены вынести за пределы основной программы. Чтобы этого избежать, мы можем передать функции значения переменных при вызове функции. Это потребует небольшой переделки в описании функции и при её вызове.

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

```
int c = 0;
int add (int one, int two) {
    c = one+two;
    return c;               }

int main()
{
    int a = 0;
    int b = 0;
    cin>>a;
    cin>>b;
    add(a,b);
    cout<<c;
    cout<<"\n";
    return 0;
}
```

Выполнение программы не отличается от выполнения предыдущих вариантов:

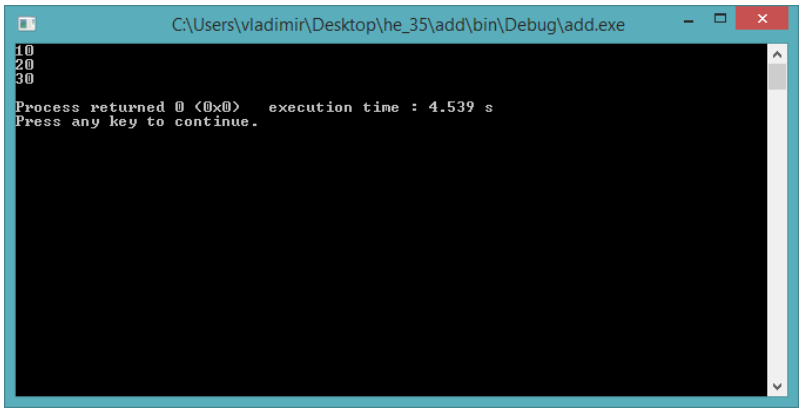


Рис. 4. Работа переделанной программы

То, как мы обошлись с функцией, называется передачей параметров функции по значению. То есть, при вызове функции ей передаются значения, необходимые ей для работы. Мы говорили, что при определении функции перед её именем стоит тип возвращаемого значения. Кроме того, блок операций в функции завершается ключевым словом *return*, сопровождаемым именем возвращаемой переменной. Мы можем убедиться, что это не пустые слова, если немного изменим предыдущий вариант основного блока программы.

```
int main()
{
    int a = 0;
    int b = 0;
    cin>>a;
    cin>>b;
    cout<< add(a,b);
    cout<<"\n";
    return 0;
}
```

Программа сложит два числа и выведет на экран результат. Имея готовый блок (функцию) инструкций для сложения двух чисел, мы можем переделать его под операцию вычитания двух чисел. А для выбора операции добавить знаки плюс и минус, чтобы объявить программе, что мы хотим сделать. Теперь «наш калькулятор» начинает походить на нечто более похожее на реальный калькулятор.

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

int add (int one, int two) //функция сложения двух чисел
{
    int rez = 0;
    rez = one+two;
    return rez;
}

int sub (int three, int four) //функция вычитания двух чисел
{
    int s = 0;
    s = three-four;
    return s;
}

int main()
{
    int a = 0;
    int b = 0;
    char c;

    cin>>a;
    cin>>b;
    cin>>c;
    if(c=='+')  cout<<add(a,b);
    if(c=='-')  cout<<sub(a,b);
    cout<<"\n";

    return 0;
}
```

Вспомнив про использование цикла, мы можем усовершенствовать программу.

```
int main()
{
    int a = 0;
    int b = 0;
    char c;
while (c != 'e') {
    cin>>a;
    cin>>b;
    cin>>c;
    if(c=='+')  cout<<add(a,b);
    if(c=='-')  cout<<sub(a,b);
    cout<<"\n";
```

```
}
  return 0;
}
```

Программа позволяет много раз производить операции сложения и вычитания. Вот работа программы:

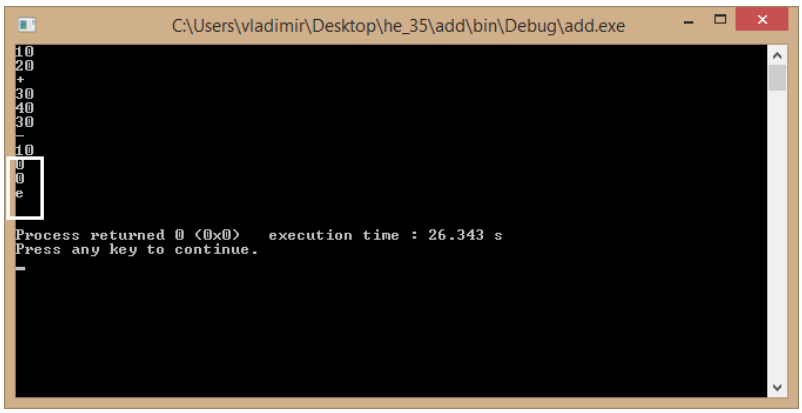


Рис. 5. Работа усовершенствованного калькулятора

В работе программы остаётся дефект, он отмечен на рисунке. Если мы хотим выйти из программы, мы должны ввести два любых числа перед вводом символа выхода из программы. Это создаёт некоторые неудобства, а при вводе символа выхода из программы вместо первого числа программа работает неправильно. И этот дефект можно устранить, если заменить порядок ввода и добавить небольшое дополнение:

```
int main()
{
    int a = 0;
    int b = 0;
    char c;
while (c != 'e') {
    cin>>c;
if (c == 'e') break;
    cin>>a;
    cin>>b;

    if(c=='+')  cout<<add(a,b);
    if(c=='-')  cout<<sub(a,b);
    cout<<"\n";
}

    return 0;
}
```

Теперь программа стала удобнее. Перед вводом данных мы вводим операцию, поскольку мы знаем, что мы хотим сделать с числами. А для выхода из программы вводим нужный символ.

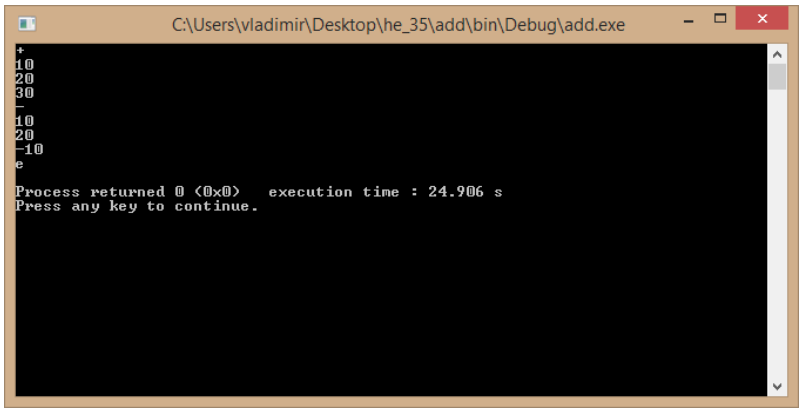


Рис. 6. Окончательный вариант работы программы

Я думаю, что вы легко можете продолжить усовершенствование программы калькулятора, но я хотел показать только одно: программа создаётся так же, как создаётся любая электрическая схема. Вам нужно знать значение и работу всех компонентов электрической схемы. Вам нужно познакомиться с наиболее употребительными блоками: усилитель, стабилизатор и т.п. Аналогично для программирования на языке Си вы должны знать простейшие составляющие языка: переменные, присваивание, переходы и условия и т.п. И должны знать, как создать блок операций, как правильно его использовать. Всему этому можно научиться, используя предлагаемое руководство. И пусть вас не смущает, что все программы работают в консоли, а мы давно привыкли, что любая программа создаёт картинку с меню и прочими атрибутами. Я бы уподобил последнее оформлению устройства: будет ли плата печатной или нет, какая коробка будет использована и с какими надписями и т.д. Но для этого вначале нужно научиться создавать устройства!

Code::Blocks

Code::Blocks – это кросс-платформенная среда разрабоки, которая поддерживает компиляцию и работу с многими языками программирования.

Загрузка возможна с:

`http://www.codeblocks.org/`

Code::Blocks может работать с разными компиляторами.

Для Windows предлагается не обязательный компилятор MingW. Версия, включающая MingW, подходит для изучения этого руководства, позволяя вам правильно компилировать примеры. Если вы не уверены, загрузите версию, названную "`codeblocks-XX.XX-mingw-setup.exe`".

Для пользователей Linux и Mac загрузите версию, соответствующую вашему дистрибутиву.

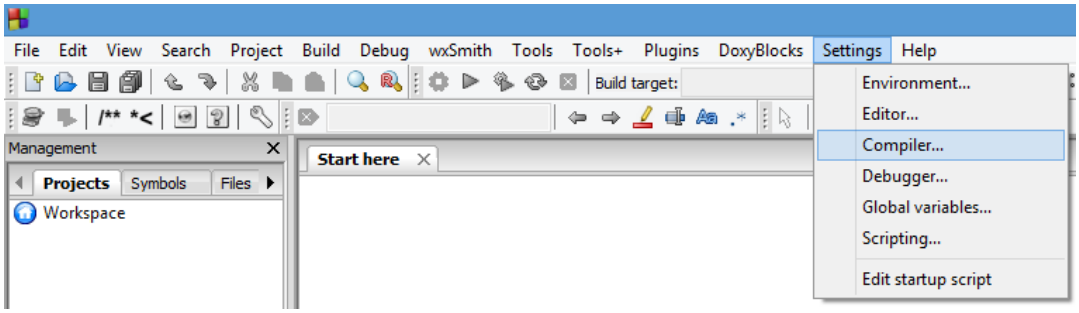
Установка

В Windows запустите загруженный исполняемый файл установки и следуйте его инструкциям. Опции по умолчанию вполне подойдут вам.

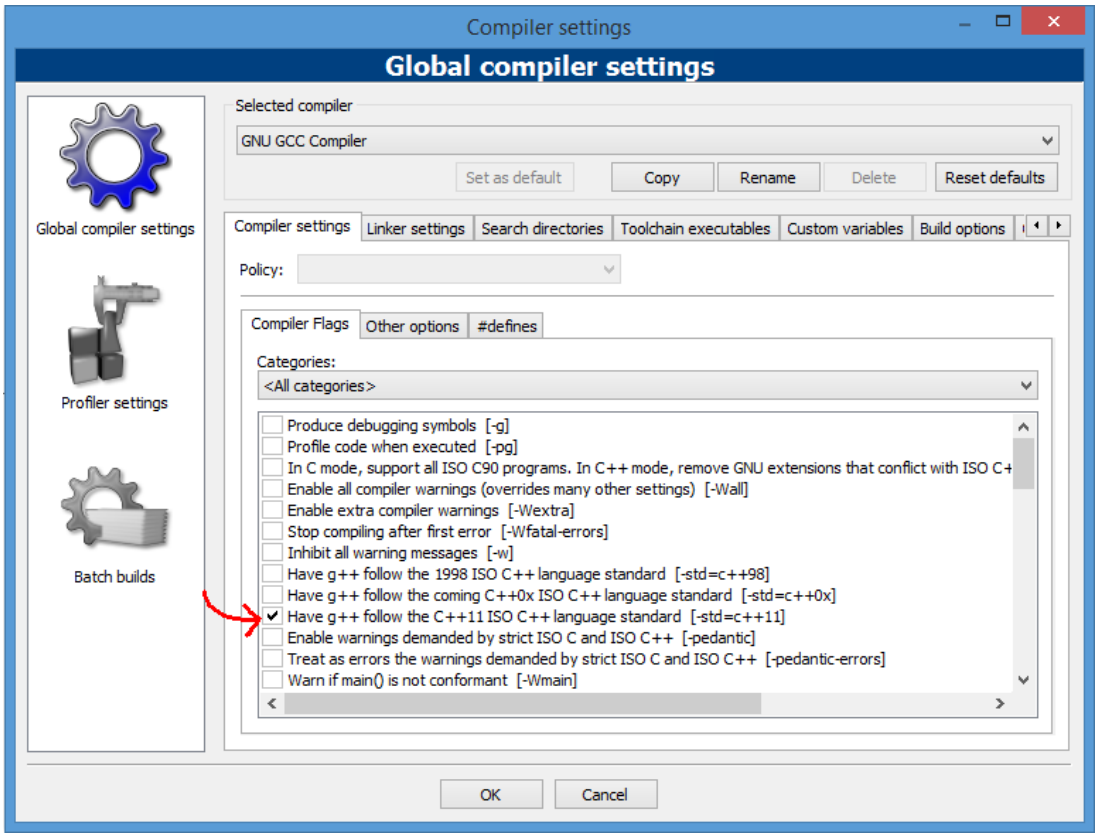
Поддержка C++11

Если у вас версия компилятора GCC (такого как MingW для Windows), есть шансы, что она пришла с поддержкой самой последней версии C++, которая недоступна по умолчанию. Это легко исправить так:

`Settings -> Compiler...`



Здесь в "Global compiler settings" на закладке "Compiler settings" отметьте окошко *"Have g++ follow the C++11 ISO C++ language standard [-std=c++11]"*:



Консольное приложение

Чтобы комплировать и запустить простые консольные приложения, как те, что используются в качестве примеров в этом руководстве, достаточно открыть файл программой Code::blocks и нажать `F9`.

В качестве примера попробуйте:

```
File -> New -> Empty File
```

Здесь напишите следующее:

```
1 #include <iostream>
2 int main()
3 {
4   auto x = R"(Hello world!)";
5   std::cout << x;
6 }
```

Затем:

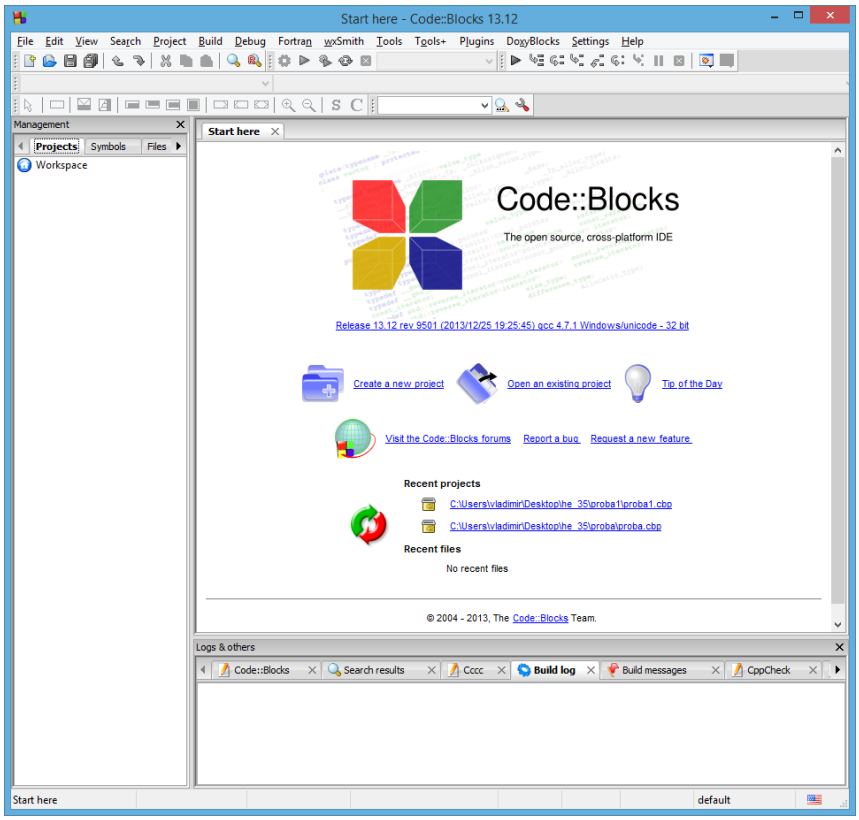
```
File -> Save file as...
```

И сохраните файл с именем и расширением `.cpp` , например, `example.cpp`.
Теперь, нажав `F9`, вы скомпилируете и запустите программу.

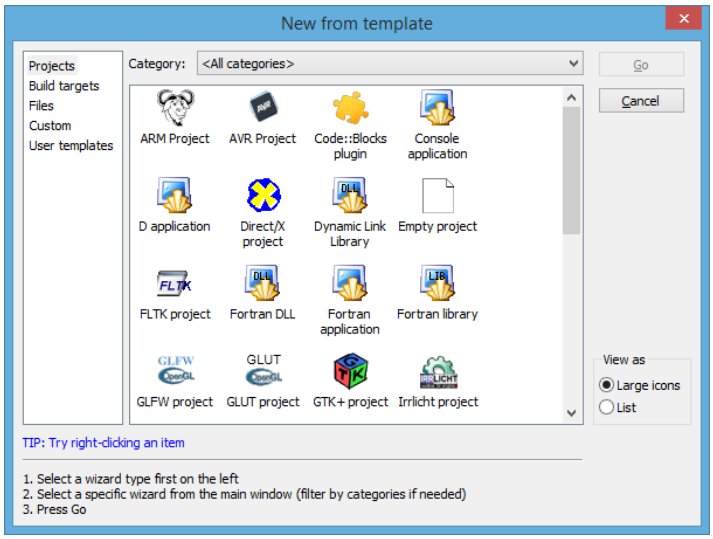
Если появится ошибка, связанная с типом переменной `x`, это означает, что компилятор не понимает значения нового типа переменной `auto` от C++11. Пожалуйста, убедитесь, что у вас подходящий компилятор и что вы задали опцию для компилятора C++11, как описано выше.

Примечание:

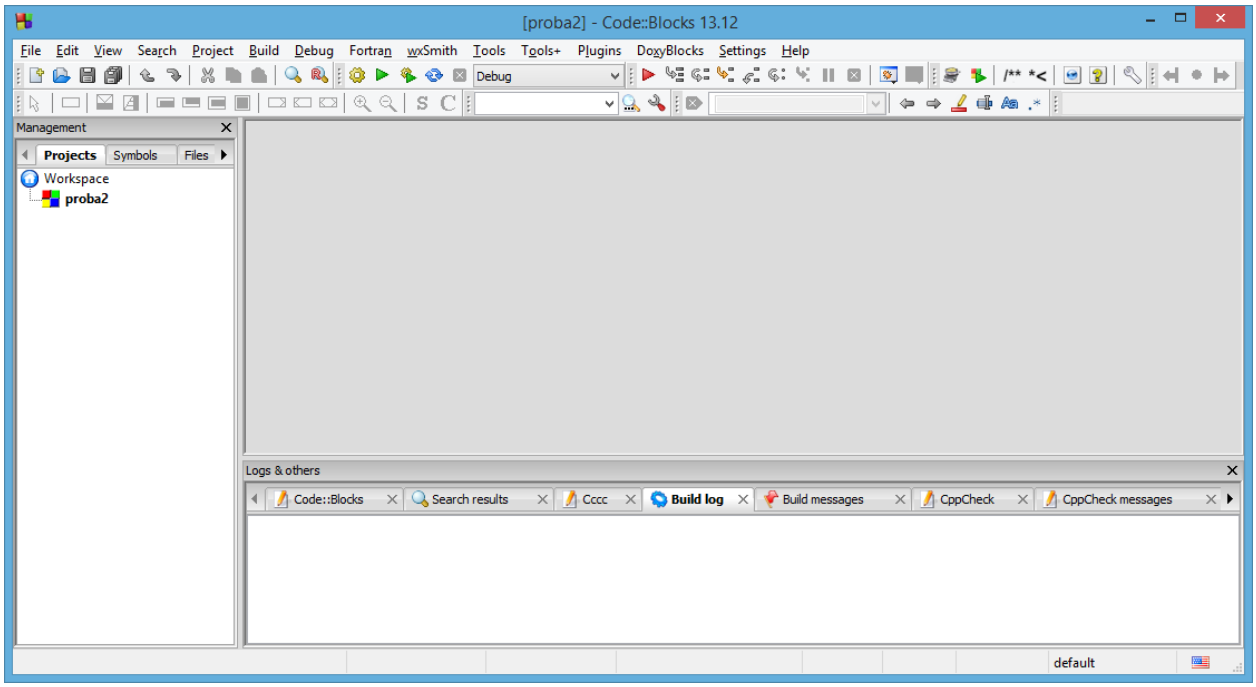
В данный момент есть версия программы 13.12. При запуске программы появляется первое меню выбора:



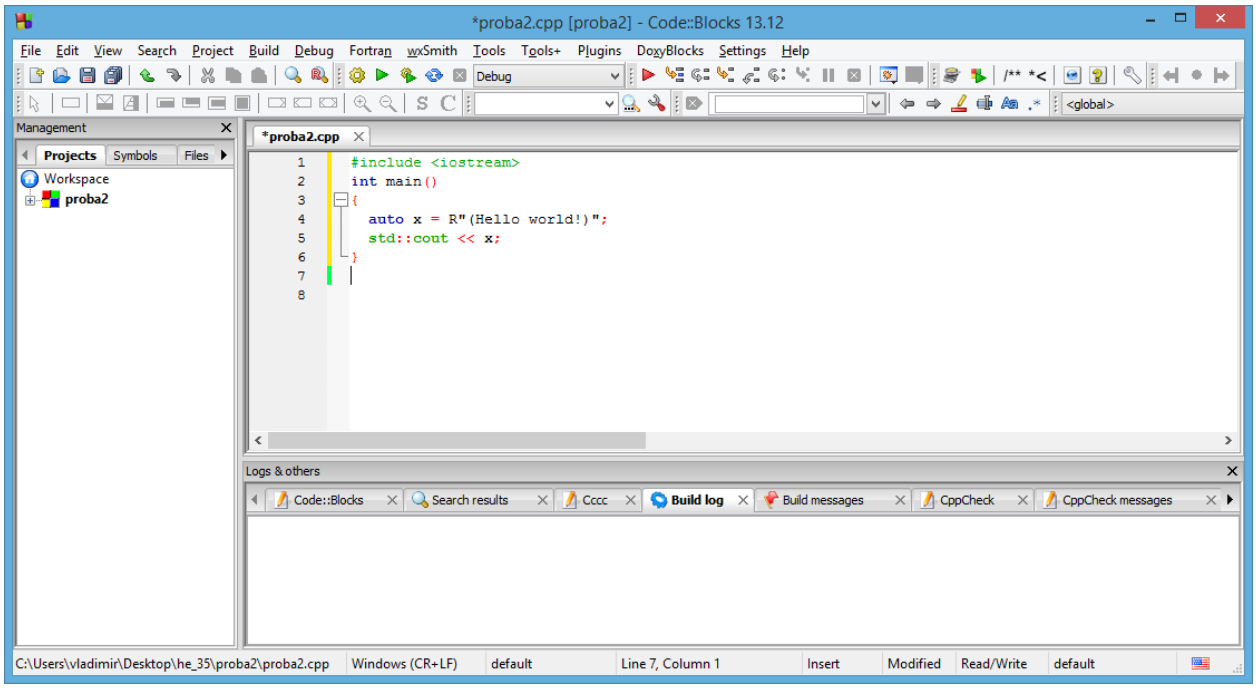
После первого запуска начало действий – это создание нового проекта (*Create a new project*). В диалоговых окнах помощника создания проекта задаётся имя и путь к проекту. Но предваряется это выбром типа проекта.



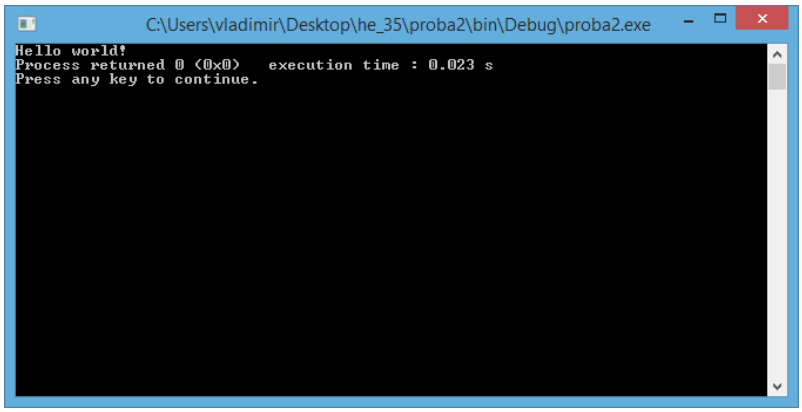
Можно выбрать пустой проект (Empty project). В этом случае мы получим следующее:



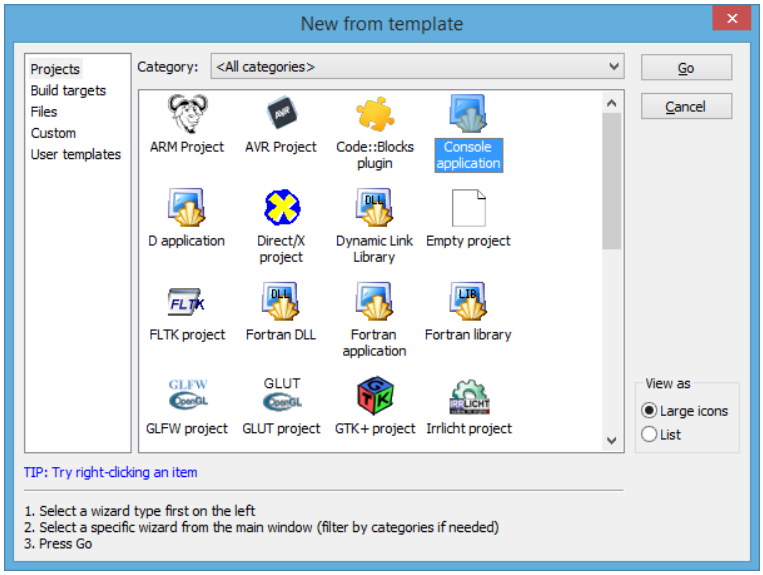
Теперь можно создать новый файл (File->New->Empty file). Будет предложено дать ему имя (как описано выше). В окно редактора можно скопировать текст простейшей программы, предложенной ранее в качестве проверочного проекта.



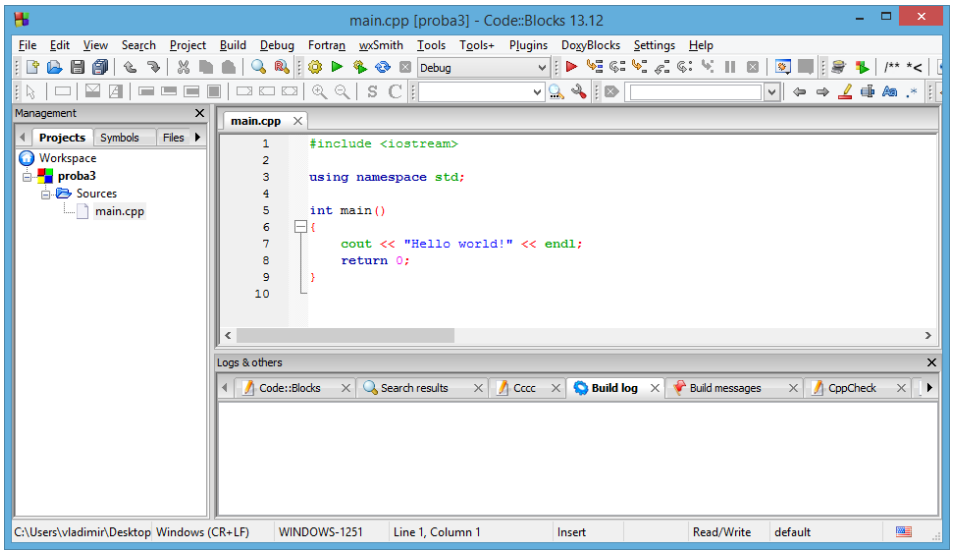
Нажав клавишу `F9`, мы получим:



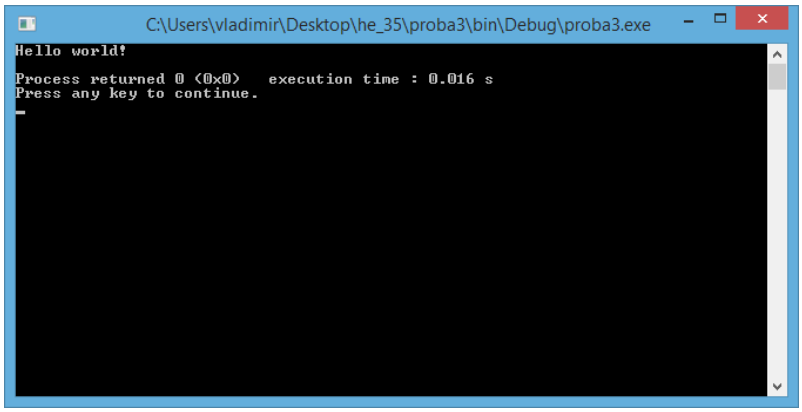
Но можно поступить и иначе, если после запуска программы и при выборе вида нового проекта выбрать не пустой проект, а консольное приложение.



Далее в диалоге нужно дать имя проекту и можно выбрать язык Си или C++. После завершения создания нового проекта вы получите и исходный файл.



Клавиша F9 вновь позволит компилировать и запустить программу (ничего не нужно добавлять или исправлять).



Результат работы программ одинаков, хотя текст различается. С подобными различиями вы можете встретиться и позже, пусть они вас не смущают, овладев языком, вы сами будете выбирать удобный вам стиль кодирования.

Язык C++

Это руководство рассказывает о языке C++ от основ до последних возможностей, включённых в версию C++11. Главы имеют практическую ориентацию, а примеры программ во всех разделах предназначены к применению на практике.

Введение

Компиляторы

Необходимыми средствами для следования этому руководству будут компьютер и компилятор, пригодный для компиляции C++ кода и создания работающей программы.

C++ – это язык, который эволюционировал на протяжении многих лет, а это руководство затрагивает многие возможности, добавленные в язык недавно. Тем не менее, чтобы правильно следовать руководству, вам понадобится современный компилятор, поддерживающий (хотя бы частично) стандарт 2011.

Многие производители компиляторов учитывают новые возможности в разных объёмах. Внизу этой страницы указаны некоторые компиляторы, которые имеют поддержку требуемых новшеств. Некоторые из компиляторов относятся к СПО!

Если по каким-то причинам вам нужен некий старый компилятор, вы можете получить доступ к старым версиям этого руководства (более не обновляемым).

Что такое компилятор?

Компьютеры понимают только один язык, и этот язык состоит из набора инструкций, соостоящих из единиц и нулей. Этот компьютерный язык справедливо называют *машинным языком*.

Отдельная инструкция компьютеру будет похожа на эту:

00000	10011110

Часть программы для компьютера на машинном языке, позволяющая пользователю ввести два числа, сложить их и отобразить результат, должна включать инструкции машинного кода:

00000	10011110
00001	11110100
00010	10011110
00011	11010100
00100	10111111
00101	00000000

Как вы можете видеть, программирование компьютера непосредственно в машинных кодах, использующих только единицы и нули, занятие весьма утомительное, увеличивающее шансы на ошибки. Чтобы облегчить процесс программирования, были разработаны языки высокого уровня. Программы на языке высокого уровня также облегчают программистам их проверку и понимание.

Вот фрагмент кода, написанного на C++, который служит тем же целям:

```
1 int a, b, sum;
2
3 cin >> a;
4 cin >> b;
```

```
5
6 sum = a + b;
7 cout << sum << endl;
```

Даже если вы пока ещё не можете понять код выше, вы согласитесь, что так программировать много легче, чем с использованием машинных кодов.

Поскольку компьютер понимает только машиный язык, а люди хотят писать на языке высокого уровня, код должен быть переписан (транслирован) в какой-то момент на машинный язык. Этим занимаются специальные программы, которые называются компиляторами, интерпретаторами или ассемблерами, и которые встраиваются в разные приложения для программирования.

Язык C++ разработан для компиляции, что означает, что трансляция программы на машинный язык должна быть понятна непосредственно системе, делая сгенерированную программу высокоэффективной. Для этого нужен набор инструментов, известных как «development toolchain, ряд инструментов для разработки». Основу этого составляют компилятор и компоновщик.

Консольные программы

Консольные программы предназначены для коммуникации с пользователями через вывод текста на экран и ввод данных и команд с клавиатуры.

С консольными программами легко взаимодействовать, и, как правило, у них предсказуемое поведение, которое одинаково для всех платформ. Они легко реализуются, а, следовательно, очень полезны при изучении основ языка программирования: примеры в этом руководстве представлены именно для консольных программ.

Способ компиляции консольных программ зависит от конкретного инструментария, который вы используете.

Самый лёгкий путь для начинающего компилировать C++ программы – это использовать Integrated Development Environment (IDE, интегрированная среда разработки). IDE обычно включает несколько инструментов для разработки: это текстовый редактор и средства компиляции программы непосредственно из него.

Вот инструкции, как компилировать и запускать консольные программы, используя разные свободные Integrated Development Interfaces (IDIs):

IDE	Platform	Console programs
Code::blocks	Windows/Linux/MacOS	Компиляция программ в Code::blocks
Visual Studio Express	Windows	Компиляция программ в VS Express 2013
Dev-C++	Windows	Компиляция программ в Dev-C++

Если вам случилось иметь средства разработки в Linux или Mac, вы можете компилировать примеры непосредственно из терминала, включив C++11 флажки в команды для компиляции:

Compiler	Platform	Command
GCC	Linux, among others...	`g++ -std=c++0x example.cpp -o example_program`
Clang	OS X, among others...	`clang++ -std=c++11 -stdlib=libc++ example.cpp -o example_program`

Основы C++

Структура программ

Наилучший способ изучить программирование – это писать программы. Типичный случай первой программы начинающего – это программа, названная "Hello World", которая просто выводит фразу"Hello World" на экран вашего компьютера. Хотя она крайне проста, она содержит все фундаментальные компоненты, которые имеют программы на языке C++:

```
1 // моя первая программа на C++
2 #include <iostream>
3
4 int main()
5 {
6   std::cout << "Hello World!";
7 }
```

```
Hello World!
```

Левая панель показывает C++ код этой программы. Правая панель показывает результат, когда программа выполняется компьютером. Серые номера слева от панелей – это номера строк, облегчающие обсуждение программы и отыскание ошибок. Они не являются частью программы.

Давайте обсудим эту программу строка за строкой:

Строка 1: `// моя первая программа на C++`
Двойная косая черта показывает, что вся строка за ней – это комментарий, вставленный программистом, но не сказывающийся на поведении программы. Программисты пользуют их для включения коротких пояснений или замечаний о коде программы. В этом случае комментарий представляет собой краткое введение в описание программы.

Строка 2: `#include <iostream>`
Строка начинается со значка решётки (`#`), который указывает, что это предназначено для чтения и интерпретации тем, что известно как *препроцессор*. Это специальные строки, интерпретируемые до начала собственно компиляции программы. В данном случае директива в строке `#include <iostream>` инструктирует препроцессор включить раздел стандартного кода C++, известный как *header iostream (заголовочный файл потокового ввода/вывода)*, позволяющий выполнять стандартные операции ввода и вывода, такие как вывод в программе (`Hello World`) на экран.

Строка 3: чистая строка.
Чистая строка не сказывается на работе программы. Она просто улучшает обозримость кода.

Строка 4: `int main ()`
Эта строка объявляет функцию. По существу функция – это группа инструкций кода, которой дано имя: в данном случае это имя «main» для группы кода далее. Функции будут детально обсуждаться в следующей главе, но, собственно, их определение вводится последовательно типом (`int`), именем (`main`) и парой скобок (`()`), включающих (не обязательно) параметры.

Функция, названная `main`, это специфическая функция во всех C++ программах; это функция, вызываемая при пуске программы. Выполнение всех C++ программ начинается с `main` функции, независимо от того, в каком месте эта функция расположена в коде программы.

Строки 5 и 7: `{` и `}`
Открывающая скобка (`{`) в строке 5 показывает начало определения функции `main`, а закрывающая скобка (`}`) в строке 7 показывает её окончание. Всё между этими скобками – это тело функции, определяющее, что случится, когда функция `main` будет вызвана. Все функции используют скобки для обозначения начала и окончания их определения.

Строка 6: `std::cout << "Hello World!";`

Это строка изложения инструкций на C++. Выражения в этом изложении кода – это то, что реально производит эффект при работе программы. Это существо программы, определяющее её актуальное поведение. Выражения выполняются в той последовательности, в какой они появляются в теле функции.

Это изложение имеет три части: первая, `std::cout`, которая идентифицирует стандартное устройство символьного вывода (обычно это экран компьютера); вторая, оператор ввода (`<<`), показывает, что следующее вводится в `std::cout`; наконец, предложение в двойных кавычках ("Hello world!") – это содержание, вводимое в стандартный вывод.

Заметьте, что выражения заканчиваются точкой с запятой (`;`). Этот символ отмечает конец выражения так же, как точка отмечает конец предложения в английском языке. Все выражения в C++ должны оканчиваться точкой с запятой. Одна из очень распространённых синтаксических ошибок в C++ – это забытое окончание выражения точкой с запятой.

Вы могли заметить, что не все строки этой программы выполняют некие действия при выполнении кода. Есть строка, содержащая комментарий (начинающийся с `//`). Есть строка с директивой для препроцессора (начиналющаяся с `#`). И есть строка, определяющая функцию (в нашем случае `main` функцию). Наконец, есть строка с выражением, заканчивающимся точкой с запятой (ввод в `cout`), который был в блоке, ограниченном скобками (`{ }`), функции `main`.

Программа была структурирована разными строками и выделена отступами, чтобы облегчить её понимание человеком, читающим её. Но C++ не имеет строгих правил, определяющих, как разделять инструкции по разным строкам. Например, вместо

```
int main ()
{
  std::cout << " Hello World!";
}
```

мы могли бы написать:

```
int main () { std::cout << "Hello World!"; }
```

всё в одной строке, и это имеет тот же смысл, что и ранее приведённый код.

В C++ разделение между выражениями задаётся завершающими точкой с запятой (`;`) с размещением на разных строках, что не существенно для этих целей. Многие выражения могут быть написаны одной строкой, или каждое выражение может иметь собственную строку. Распределение кода по разным строкам служит только для большей чёткости и схематичности для человека, читающего код, но не сказывается на реальном поведении программы.

Теперь давайте добавим выражение в нашу первую программу:

```
// my second program in C++
#include <iostream>

int main ()
{
  std::cout << "Hello World! ";
  std::cout << "I'm a C++ program";
}
```

```
Hello World! I'm a C++ program
```

В этом случае программа выполняет два ввода в `std::cout` в двух разных выражениях. И опять, разделение кода на строки просто даёт больше удобства при чтении программы, поскольку `main` будет правильно определена и так:

```
int main () { std::cout << " Hello World! "; std::cout << " I'm a C++
program "; }
```

Исходный код можно разделить на большее число строк:

```
1 int main ()
2 {
3   std::cout <<
4     "Hello World!";
5   std::cout
6     << "I'm a C++ program";
7 }
```

И результат будет таким же, как и в предыдущих примерах.

Директивы препроцессора (те, что начинаются с `#`) выпадают из этих общих правил, поскольку они не являются выражениями. Они читаются и выполняются препроцессором до начала компиляции. Директивы препроцессора должны задаваться в отдельных строках, и, поскольку они не являются выражениями, не завершаться точкой с запятой (`;`).

Комментарии

Как отмечено выше, комментарии не сказываются на работе программы; однако они представляют важный аспект непосредственного документирования в исходном коде, подсказывая, что программа делает и как она это делает.

C++ поддерживает два способа комменитрования кода:

```
1 // строка комментария
2 /* блок комментария */
```

Первый из них, известный как *line comment (строка комментария)*, исключает всё, что идёт за парой косых чёрточек (`//`) вполоть до конца строки. Второй, известный как *block comment (блок комментария)*, исключает всё, что находится между символами, `/*` от этих символов и до первого появления символов `*/`, предполагая, что можно включать множество строк между символами.

Давайте добавим комментарии к нашей второй программе:

```
1  /* моя вторая программа на C++
2     с длиным комментарием */
3
4  #include <iostream>
5
6  int main ()
7  {
8    std::cout << "Hello World! ";
9  // выводит Hello World!
10   std::cout << "I'm a C++ program";
   // выводит I'm a C++ program
   }
```

```
Hello World! I'm a C++ program
```

Если комментарии вставлены в исходный код программы без использования символов комментария `//` или `/*` or `*/`, компилятор будет считать их выражениями C++, что, вероятнее всего, приведёт к отказу при компиляции с одной или несколькими сообщениями об ошибках.

Использование пространства имён std

Если вы рассматривали код C++ в самом начале, вы могли заметить `cout` (читается – сиаут), использующееся вместо `std::cout`. Оба варианта называют тот же самый объект: первый вариант использует его *unqualified name* (`cout`), неквалифицированное имя, тогда как второй вариант квалифицирует его непосредственно внутри (пространства имён) *namespace* `std` (как `std::cout`).

`cout` – это часть стандартной библиотеки, и все его элементы в стандартной библиотеке C++ объявлены внутри того, что называется *namespace (пространство имён)*: namespace `std`.

В порядке обращения к элементам в пространстве имён `std` программа либо квалифицирует каждый и каждое использование элементов библиотеки (как мы делали с помощью предварения `cout` с помощью `std::`), либо вводит видимость его компонентов. Наиболее типичный способ ввести видимость этих компонентов – это *декларирование использования*:

```
using namespace std;
```

Объявление выше позволяет всем элементам в пространстве имён `std` появляться на *неквалифицированный* манер (без `std::` префикса).

Имея это в виду, последний пример можно переписать, сделав *неквалифицированное* использование `cout` как:

```
1 // my second program in C++
2 #include <iostream>
3 using namespace std;
4
5 int main ()
6 {
7   cout << "Hello World! ";
8   cout << "I'm a C++ program";
9 }
```

```
Hello World! I'm a C++ program
```

Оба способа доступа к элементам пространства имён `std` (применяя квалифицирование и объявление использования (*using)*) правильны в C++ и определяют одинаковое поведение. Для упрощения и улучшения читаемости примеры в этом руководстве будут часто использовать последний вариант с декларацией использования (*using)*, хотя отметьте, что *explicit qualification (явная квалификация)* – это единственный путь, гарантирующий, что никогда не возникнет коллизии имён.

Пространства имён подробнее рассматриваются в следующей главе.

Переменные и типы

Польза программы «Hello World», показанной в предыдущей главе, довольно сомнительна. Мы должны были написать несколько строк кода, компилировать их, и затем выполнить результирующую программу только для того, чтобы получить в результате простое предложение на экране. Быстрее было бы написать это предложение вручную.

Однако программирование не ограничено только печатью простых текстов на экране. В плане дальнейшего продвижения, и чтобы стать способными написать программы, выполняющие нечто полезное, реально поддерживающее нашу работу, нам нужно ввести концепцию *переменной* (*variable)*.

Давайте представим, что я прошу вас запомнить число 5, а затем одновременно сохранить в памяти число 2. Вы только сохранили в своей памяти два разных значения (5 и 2). Теперь, если я попрошу вас добавить 1 к первому названному мной числу, вы должны будете хранить число 6 (что есть 5+1) и 2 в вашей памяти. Затем мы можем, например, вычесть эти значения и получить в результате 4.

Весь процесс, описанный выше, похож на то, что компьютер может сделать с двумя переменными. Этот процесс может быть представлен в C++ следующим набором выражений:

```
1 a = 5;
2 b = 2;
3 a = a + 1;
4 result = a - b;
```

Понятно, это очень простой пример, поскольку мы использовали только два маленьких целых числа, но представьте, ваш компьютер может запоминать миллионы чисел подобных этим одновременно и проводить сложные математические операции с ними.

Теперь мы готовы определить *переменную* (*variable)* как часть памяти для хранения значения.

Каждая переменная нуждается в имени, которое идентифицирует её и отличает от других. Например, в предыдущем коде имена переменных были `a`, `b` и `result`, но мы могли назвать переменные любыми именами, которые придумали бы, с длиной, допустимой для C++ идентификаторов.

Идентификаторы

Правильный *идентификатор* (*valid identifier)* – это последовательность одной или более букв, цифр или символов подчёркивания (_). Пробелы, знаки препинания и служебные символы не могут быть частью идентификатора. Вдобавок, идентификатор должен всегда начинаться с буквы. Они могут начинаться также с символа подчёркивания (_), но подобные идентификаторы во многих случаях задумывались для резервирования специфических компьютерных ключевых слов или внешних идентификаторов, как и идентификаторы, содержащие где-либо два соседних символа подчёркивания. Ни в коем случае они не могут начинаться с цифры.

Язык C++ использует некоторое количество ключевых слов для идентификации операций и описания данных. По этой причине идентификаторы, создаваемые программистом, не должны совпадать с этими ключевыми словами. Стандартом зарезервированы ключевые слова, которые не должны использоваться программистом при создании идентификаторов:

```
alignas, alignof, and, and_eq, asm, auto, bitand, bitor, bool, break, case,
catch, char, char16_t, char32_t, class, compl, const, constexpr, const_cast,
continue, decltype, default, delete, do, double, dynamic_cast, else, enum,
explicit, export, extern, false, float, for, friend, goto, if, inline, int,
long, mutable, namespace, new, noexcept, not, not_eq, nullptr, operator, or,
or_eq, private, protected, public, register, reinterpret_cast, return, short,
signed, sizeof, static, static_assert, static_cast, struct, switch, template,
this, thread_local, throw, true, try, typedef, typeid, typename, union,
```

`unsigned`, `using`, `virtual`, `void`, `volatile`, `wchar_t`, `while`, `xor`, `xor_eq`

Некоторые компиляторы могут иметь дополнительные зарезервированные ключевые слова.

Очень важно: язык C++ относится к языкам «чувствительным к регистру». Это значит, что идентификатор, написанный большими буквами не эквивалентен другому, который имеет то же имя, но написанное маленькими буквами. То есть, например, переменная `RESULT` не та же, что переменная `result` или переменная `Result`. Это три разных идентификатора, определяющих разные переменные.

Фундаментальные типы данных

Значения переменных хранятся где-нибудь в неопределённом месте памяти компьютера как нули и единицы. Нашей программе нет необходимости знать конкретное место хранения переменной; она может просто обращаться к переменной по имени. Но в чём программа должна быть уверена, так это в том, какого рода данные хранит переменная. Совсем не одно и то же – хранить ли простое целое, как хранится буква, или хранить большое число с плавающей точкой; хотя все они представлены нулями и единицами, трактуются они по-разному, а во многих случаях они требуют разного количества памяти.

Фундаментальные типы данных – это основные типы, встроенные непосредственно в язык, который представляет основные единицы памяти, поддерживаемые исходно большинством систем. Они могут классифицироваться:

- **Символьные типы:** они могут представлять единственный символ, такой как `'A'` или `'$'`. Основной тип – это `char`, являющийся однобайтовым. Другие типы предусмотрены для расширенных символов.
- **Числовые целые типы:** они могут хранить целые значения, такие как `7` иди `1024`. Они могут быть разных размеров, и могут быть и *signed (со знаком)*, и *unsigned (беззнаковые)* в зависимости от того, поддерживают ли они отрицательные значения или нет.
- **Типы с плавающей точкой:** они могут поддерживать действительные значения, такие как `3.14` или `0.01`, с разной степенью точности, что зависит от того, какой из трёх типов с плавающей точкой используется.
- **Булев тип:** булев тип, известный в C++ как `bool`, может представлять только два состояния `true` или `false`.

Вот полный список фундаментальных типов в C++:

Группа	Имена типов*	Замечание по размеру / точности
Символьные типы	**`char`**	Точно один байт, 8 бит.
	`char16_t`	Не менее, чем `char`. По крайней мере, 16 бит.
	`char32_t`	Не менее, чем `char16_t`. Хотя бы 32 бита.
	`wchar_t`	Может представлять наибольший поддерживаемый набор символов.
Целые типы (signed)	**`signed char`**	То же размер, что и `char`. Хотя бы 8 бит.
	`signed` **`short`** *`int`*	Не менее, чем `char`. Хотя бы 16 бит.
	`signed` **`int`**	Не менее, чем `short`. Хотя бы 16 бит.
	`signed` **`long`** *`int`*	Не менее, чем `int`. Хотя бы 32 бита.
	`signed` **`long long`** *`int`*	Не менее, чем `long`. Хотя бы 64 бита.
Целые типы (unsigned)	**`unsigned char`**	(тот же размер, что их двойник со знаком)
	`unsigned short` *`int`*	
	`unsigned` *`int`*	
	`unsigned long` *`int`*	

	unsigned long long *int*	
С плавающей точкой	**float**	
	double	Точность не менее чем `float`
	long double	Точность не менее чем `double`
Булев тип	**bool**	
Void тип	**void**	Не отводится место для хранения
Нулевой указатель	**decltype(nullptr)**	

* Имена некоторых целых типов могут быть укорочены без их `signed` и `int` составляющих – только часть без выделения курсивом требуется для идентификации типа, часть, выделенная курсивом, необязательна. То есть, *`signed`* `short` *`int`* может записываться как `signed short`, `short int` или просто `short`; они все идентифицируются как тот же фундаментальный тип.

В каждой из групп выше различие между типами только в их размере (то есть, как много места они занимают в памяти): первый тип в каждой из групп самый маленький, а последний самый большой, где каждый тип не менее предыдущего в той же группе. За исключением того, что типы в группе имееют одинаковые свойства.

Заметьте, что в таблице выше иные типы, чем `char` (имеющий размер точно в один байт), ни один из фундаментальных типов не имеет заданного стандартного размера (только минимальный размер, самое большее). Следовательно, тип не требует (и во многих случаях это так) в точности этого минимального размера. Это не означает, что эти типы имеют неопределённый размер, но только то, что нет стандартного размера для всех компиляторов и компьютеров. В каждой реализации компилятора могут задаваться размеры для этих типов, наилучшим образом подходящие к архитектуре компьютера, где программа будет работать. Эта, скорее универсальная, спецификация для типов придаёт языку C++ большую гибкость в адаптации к оптимальной работе на всех платформах, как существующих, так и будущих.

Размеры типов выше выражены в битах; чем больше битов имеет тип данных, тем больше различных значений он может представлять, но в то же время и отбирает больше памяти:

Размер	**Уникальные представляемые значения**	**Примечания**
8-bit	256	= 2^8
16-bit	65 536	= 2^{16}
32-bit	4 294 967 296	= 2^{32} (~4 миллиарда)
64-bit	18 446 744 073 709 551 616	= 2^{64} (~18 миллиардов миллиардов)

Для целых типов наиболее представимые величины означают, что диапазон величин, которые они могут представлять, больше. Например, 16-битовое беззнаковое целое может представить 65536 разных значений в диапазоне от 0 до 65535, тогда как его двойник со знаком сможет представить, во многих случаях, значения между -32768 и +32767. Заметьте, что диапазон положительных чисел приблизительно вдвое меньше при сравнении данных со знаком с беззнаковыми, поскольку один бит из 16 используется для знака. Это относительно скромная разница, и она редко оправдывает использование беззнаковых типов, основанных чисто на диапазоне положительных чисел, которые они могут представлять

Для типов с плавающей точкой размер зависит от их точности, имея больше или меньше битов для их значащей части и экспоненты.

Если размер или точность типа не важны, тогда `char`, `int` и `double` обычно выбираются для использования с символами, целыми и числами с плавающей точкой соответственно. Другие типы в их соответствующих группах используются только в особенных случаях.

Свойства фундаментальных типов в обычных системах и реализациях компиляторов могут быть получены при использовании классов `numeric_limits` (см. стандартный заголовочных файл `<limits>`). Если есть какие-то соображения, когда нужны типы особых размеров, библиотека определяет некоторые альтернативные названия типов фиксированного размера в заголовочном

файле <cstdint>.

Типы, описанные выше (символьные, целые, с плавающей точкой и булевы), совместно известны как арифметические типы. Но есть два дополнительных фундаментальных типа: void, идентифицирующий отсутствие типа, и тип nullptr, который означает специальный тип указателя. Оба типа будут обсуждаться позже в главе, посвящённой указателям.

C++ поддерживает большое разнообразие типов, основанных на фундаментальных типах данных, описанных выше. Эти другие типы известны как *compound data types (сложные типы данных)*, и они представляют одну из основных сильных сторон языка C++. Мы их разберём также более детально в следующих главах.

Объявление переменных

C++ – язык со строгим контролем типов. Это требует, чтобы каждая переменная была объявлена с её типом до первого использования. Это оповещает компилятор о необходимом размере резервируемой памяти и о том, как интерпретировать значение переменной. Синтаксис объявления новой переменной в C++ прямой: мы просто пишем тип перед именем переменной (то есть, её идентификатором). Например:

```
int a;
float mynumber;
```

Здесь два правильных объявления переменных. Первое объявляет переменную типа int с иденитификатором a. Второе объявляет переменную типа float с идентификатором mynumber. Когда они объявлены, переменные a и mynumber могут использоваться внутри остальной их области применения в программе. Если объявлено более одной переменной с тем же типом, они могут объявляться в одной строке с именами, разделяемыми запятыми. Например:

```
int a, b, c;
```

Этим объявляются три переменные (a, b и c), все они типа int, и это значит то же самое, что:

```
int a;
int b;
int c;
```

Чтобы посмотреть, как выглядит объявление переменной в действии, в программе, давайте рассмотрим весь код C++ примера, предложенного к мысленному эксперименту в начале этой главы:

```
// operating with variables

#include <iostream>
using namespace std;

int main ()
{
  // объявление переменных:
  int a, b;
  int result;

  // процесс:
  a = 5;
  b = 2;
  a = a + 1;
  result = a - b;

  // вывод результата:
```

```
4
```

```
19   cout << result;
20
21   // завершение программы:
22   return 0;
23 }
```

Не беспокойтесь, если что-то, кроме объявления собственно переменной, выглядит для вас немного странно. Всё это будет детально разъяснено в следующих главах.

Инициализация переменных

Когда переменные в примере выше объявлены, они имеют неопределённое значение, пока им впервые не будет присвоено значение. Но можно задать переменной значение при её объявлении. Это называется *инициализацией* переменной.

В C++ есть три способа инициализации переменных. Они все равнозначны и напоминают об эволюции языка в течение многих лет.

Первый, известный как *c-like initialization* (C-подобная инициализация, поскольку она пошла от языка Си), заключается в добавлении знака равенства и значения, которым инициализируется переменная:

```
type identifier = initial_value;
```

Например, для оъявления переменной типа `int` с именеме `x` и начальным значением ноль при объявлении мы можем написать:

```
int x = 0;
```

Второй метод, известный как *constructor initialization* (конструктивная инициализация, введённая языком C++), разрешает вставлять начальное значение между скобками (`()`):

```
type identifier (initial_value);
```
Например:

```
int x (0);
```

Наконец, третий метод, известный как *uniform initialization (единообразная инициализация)*, похож на предыдущий, но использует фигурные скобки (`{}`) вместо обычных (это было введено при ревизии стандарта C++ в 2011):

```
type identifier {initial_value};
```

Например:

```
int x {0};
```

Все три способа инициализации правильны и эквивалентны в C++.

```
1  // инициализация переменных
2
3  #include <iostream>
4  using namespace std;
5
6  int main ()
7  {
```

```
6
```

```
 8   int a=5;     // начальное значение: 5
 9   int b(3);    // начальное значение: 3
10   int c{2};    // начальное значение: 2
11   int result; // начальное значене не определено
12
13   a = a + b;
14   result = a - c;
15   cout << result;
16
17   return 0;
18 }
```

Выявление типа: auto и decltype

Когда инициализируется новая переменная, компилятор может автоматически выяснить тип переменной с помощью инициализатора. Для этого достаточно использовать `auto` в качестве типа, определённого для переменной:

```
1 int foo = 0;
2 auto bar = foo;  // то же, что: int bar = foo;
```

Здесь `bar` декларируется как переменная, имеющая тип `auto`; поэтому тип `bar` – это тип значения, используемый для инициализации: в нашем случае используется тип `foo`, он же `int`.

Переменные, которые не инициализированы, могут также использовать выявление типа спецификатором `decltype`:

```
1 int foo = 0;
2 decltype(foo) bar;  // то же, что: int bar;
```

Здесь `bar` декларируется как имеющий то же тип, что и `foo`.

`auto` и `decltype` – это мощные средства, добавленные в язык недавно. Но выявление типа, вводимого ими, предназначены для использования либо тогда, когда тип не может быть получен другими методами, либо тогда, когда их использование улучшает читаемость кода. Два примера выше не были, скорее всего, одним из этих случаев использования. Более того, они, вероятно, снижают читаемость, поскольку при чтении кода приходится искать тип `foo` для понимания типа `bar`.

Введение в строки

Фундаментальные типы представляют собой самые основные, поддерживаемые компьютерами, для работы на которых код и создаётся. Но одним из основных достоинств языка C++ является его богатый набор сложных типов, для которых фундаментальные типы становятся простыми строительными блоками.

Примером сложного типа является класс `string`. Переменные этого типа способны хранить последовательности символов, такие как слова или предложения. Очень полезная возможность!

Первое отличие от фундаментальных типов данных в порядке объявления и использования объектов (переменных) этого типа, в программу нужно включить заголовочный файл, где определяется этот тип внутри стандартной библиотеки (заголовочный файл `<string>`):

```
1 // my first string
2 #include <iostream>
3 #include <string>
4 using namespace std;
```

```
This is a string
```

```
5
6  int main ()
7  {
8    string mystring;
9    mystring = "This is a string";
10   cout << mystring;
11   return 0;
12 }
```

Как вы можете видеть в примере выше, строки могут инициализироваться любыми допустимыми строковыми символами, подобно тому, как переменные числового типа могут инициализироваться любыми допустимыми числовыми символами. Как и с фундаментальными типами, все форматы инициализации справедливы и для строк:

```
1 string mystring = "This is a string";
2 string mystring ("This is a string");
3 string mystring {"This is a string"};
```

Со строками могут также выполняться все базовые операции, что и с фундаментальными типами, подобно объявлению без начального значения и изменения значения в процессе выполнения:

```
1  // my first string
2  #include <iostream>
3  #include <string>
4  using namespace std;
5
6  int main ()
7  {
8    string mystring;
9    mystring = "This is the initial
10 string content";
11   cout << mystring << endl;
12   mystring = "This is a different
13 string content";
14   cout << mystring << endl;
     return 0;
   }
```

```
This is the initial string content
This is a different string content
```

Примечание: вставка `endl` – это манипулятор ends the line, конец строки (печать символа новой строки и очистка потока).

Класс `string` принадлежит к *compound type (сложный тип)*. Как вы могли видеть в примере выше, *сложные типы* используются так же, как и *фундаментальные типы*: используется тот же синтаксис при объявлении переменных и их инициализации.

Более детально со стандартом строк C++ можно ознакомиться по ссылке класс `string`.

Постоянные

Постоянные – это выражения с фиксированным значением.

Литералы

Литералы – это наиболее явный вид констант. Они используются для установки определённых значений внутри исходного кода программы. Мы уже использовали некоторые из них в предыдущих главах, задавая значения переменным или устанавливая сообщения, которые мы хотели бы, чтобы наша программа распечатала, например, когда мы пишем:

```
a = 5;
```

`5` в этой части кода было *literal constant (литеральная постоянная)*.

Литеральные константы могут классифицироваться в: целых, с плавающей точкой, символьных, строковых, булевых, указательных и определённых пользователем литералах.

Целые числа

```
1776
707
-273
```

Это числовые константы, которые идентифицируют целые значения. Заметьте, что они не в кавычках или других специальных символах; это просто последовательность цифр, представляющая целое десятичное число; например, `1776` всегда представляет значение *одна тысяча семьсот семдесят шесть*.

В дополнение к десятичным числам (тех, что мы используем каждодневно) C++ позволяет использовать восьмеричные числа (с основанием 8) и шестнадцатеричные числа (с основанием 16) как литеральные константы. Для восьмеричных литералов цифры предваряются символом `0` (ноль). А для шестнадцатеричных предваряются символом `0x` (ноль, x). Например, следующие литеральные константы все эквивалентны:

```
75          // десятичное
0113        // восьмеричное
0x4b        // шестнадцатеричное
```

Всё это представляет одно число: 75 (семдесят пять) выраженное десятичными, восьмеричными и шестнадцатеричными цифрами соответственно.

Эти литеральные константы имеют свой тип, как и переменные. По умолчанию целые литералы имеют тип `int`. Однако к целым литералами могут быть добавлены некоторые суффиксы, задающие разные целые типы:

Суффикс	Модификатор типа
`u` *or* `U`	`unsigned`
`l` *or* `L`	`long`
`ll` *or* `LL`	`long long`

Беззнаковые (unsigned) могут комбинироваться с любыми другими двумя для формирования `unsigned long` или `unsigned long long`.

Например:

```
75          // int
75u         // unsigned int
75l         // long
75ul        // unsigned long
75lu        // unsigned long
```

Во всех случаях выше суффикс может быть задан с использованием либо прописных, либо строчных букв.

Числа с плавающей точкой

Они представляют действительные числа в десятичном и/или экспоненциальном виде. Они могут включать либо десятичную точку, символ `e` (выражающий «*десять в степени X*», где *X* – это целое, следующее за символом `e`), либо и то, десятичную точку, и другое, символ `e`:

```
3.14159     // 3.14159
6.02e23     // 6.02 x 10^23
1.6e-19     // 1.6 x 10^-19
3.0         // 3.0
```

Вот четыре числа в десятичном виде в C++. Первое число – это PI, второе – одно из чисел Авогадро, третье – электрический заряд электрона (очень маленькое число) – все они приблизительные – а последнее число *три* выражено как числовой литерал с плавающей точкой.

По умолчанию для литералов с плавающей точкой предназначен тип `double`. Литералы с плавающей точкой типов `float` или `long double` могут задаваться добавлением одного из следующих суффиксов:

Суффикс	Тип
`f` *or* `F`	`float`
`l` *or* `L`	`long double`

Например:

```
3.14159L    // long double
6.02e23f    // float
```

Любая буква, которая может быть частью числовой константы с плавающей точкой (`e`, `f`, `l`) может записываться как в нижнем, так и в верхнем регистре, что не меняет сущности.

Символьные и строковые литералы

Символьные и строковые литералы заключаются в кавычки:

```
'z'
'p'
"Hello world"
"How do you do?"
```

Первые два выражения представляют *отдельные буквенные литералы*, а следующие два представляют *строковые литералы*, состоящие из нескольких символов. Заметьте, что для представления единственного символа мы заключаем его в одиночные кавычки ('), а для представления строки (обычно содержащей более одного символа) мы используем двойные кавычки (").

И одиночные символы, и строковые литералы требуют кавычек вокруг них, чтобы отличить их от возможных идентификаторов переменных или зарезервированных ключевых слов. Отметьте различия между этими двумя выражениями:

```
x
'x'
```

Здесь одиночное `x` будет относиться к идентификатору, такому как имя переменной или сложному типу, тогда как `'x'` (заключённое в одиночные кавычки) будет относиться к литералу буквы `'x'` (символу, представляющему букву *x* в нижнем регистре).

Символьные и строковые литералы могут также представлять специальные символы, что трудно или невозможно выразить иначе в исходном коде программы подобно символу новой строки (`\n`) или табуляции (`\t`). Все эти специальные символы предваряются символом обратной косой черты (`\`).

Вот список одиночных символов escape codes:

Escape code	Описание
`\n`	Новая линия
`\r`	Возврат каретки
`\t`	Табуляция
`\v`	Вертикальная табуляция
`\b`	Возврат на символ
`\f`	Прогон формы (перевод страницы)
`\a`	Сигнал (гудок)
`\'`	Одиночная кавычка (')
`\"`	Двойная кавычка (")
`\?`	Знак вопроса (?)
`\\`	Обратная косая черта (\)

Например:

```
'\n'
'\t'
"Left \t Right"
"one\ntwo\nthree"
```

В компьютере символы представлены как числовые коды: чаще всего используются одно из расширений символов ASCII системы кодирования (см. ASCII code). Символы могут быть также представлены литералами, использующими их числовой код, записанный после обратной косой черты (`\`) в восьмеричном (на базе 8) или шестнадцатеричном (на базе 16) числовом виде. Для восьмеричных значений число следует за косой чертой, а шестнадцатеричному значению предшествует символ `x` за обратной косой чертой (например: `\x20` или `\x4A`).

Несколько строковых литералов могут объединяться вместе в один строковый литерал, просто отделите их одним или несколькими пробелами, в том числе табуляцией, новой строкой и другими подходящими символами пробелов. Например:

```
1 "this forms" "a single"     " string "
```

```
2 "of characters"
```

Строковый литерал выше эквивалентен:

```
"this forms a single string of characters"
```

Заметьте, что пробелы внутри кавычек – это часть литерала, а вне кавычек нет.

Некоторые программисты используют трюк для включения длинного строкового литерала в несколько строк: в C++ обратная косая черта (\) в конце строки обосновывает символ *line-continuation (продолжение строки)*, который связывает обе эти строки в одну. Таким образом, следующий код:

```
1 x = "string expressed in \
2 two lines"
```

эквивалентен:

```
x = "string expressed in two lines"
```

Все символьные литералы и строковые литералы, описанные выше, состоят из символов типа `char`. Другие типы символов могут задаваться с использованием одного из префиксов:

Префикс	**Тип символа**
`u`	`char16_t`
`U`	`char32_t`
`L`	`wchar_t`

Заметьте, что в отличие от суффиксов для целых литералов эти префиксы *чувствительны к регистру*: строчные для `char16_t` и прописные для `char32_t` и `wchar_t`.

Для строковых литералов помимо вышеуказанных `u`, `U` и `L` есть два дополнительных префикса:

Префикс	**Описание**
`u8`	Строковый литерал в кодировке UTF-8
`R`	Строковый литерал – это необработанная строка

В необработанной строке обратные косые и кавычки, одинарные и двойные, все будут правильными символами; содержание литерала ограничено началом `R"`*`sequence`*`(`и окончанием`)`*`sequence`*`"`, где *`sequence`* – это любая последовательность символов (включая пустую последовательность). Содержание строки – это всё, что лежит *внутри скобок*, игнорируя сами ограничители *sequence*. Например:

```
1 R"(string with \backslash)"
2 R"&%$(string with \backslash)&%$"
```

Обе строки выше эквивалентны `"string with \backslash"`. Префикс `R` можно комбинировать с другими префиксами, такими как `u`, `L` или `u8`.

Другие литералы

В C++ есть литералы трёх ключевых слов: `true`, `false` и `nullptr`:

- `true` и `false` – это два возможных значения для переменной типа `bool`.
- `nullptr` – это значение *null pointer (нулевой указатель)*.

```
bool foo = true;
bool bar = false;
int* p = nullptr;
```

Типизированные константы выражений

Иной раз удобно давать имена константам:

```
const double pi = 3.1415926;
const char tab = '\t';
```

Позже их можно использовать вместо литералов, для чего они и определялись:

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

const double pi = 3.14159;
const char newline = '\n';

int main ()
{
  double r=5.0;               //
radius
  double circle;

  circle = 2 * pi * r;
  cout << circle;
  cout << newline;
}
```

```
31.4159
```

Определения препроцессора (#define)

Другой механизм именовать константы – это использовать определения препроцессора. Они имеют следующую форму:

```
#define identifier replacement
```

После этой директивы любое появление `identifier` в коде программы интерпретируется как `replacement`, где подстановка (*replacement*) – это любая последовательность символов (до конца строки). Это замещение выполняется препроцессором, и происходит перед компиляцией программы. Таким образом, это приводит к чему-то вроде слепой замены: правильность типов или синтаксиса не проверяется в любом случае.

Например:

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

```
31.4159
```

```
3
4  #define PI 3.14159
5  #define NEWLINE '\n'
6
7  int main ()
8  {
9    double r=5.0;    // radius
10   double circle;
11
12   circle = 2 * PI * r;
13   cout << circle;
14   cout << NEWLINE;
15
16 }
```

Заметьте, что строка `#define` – это директива препроцессора, и, таким образом, является однострочной инструкцией, которая, в отличие от выражений C++, не требует точки с запятой (;) в конце; директива распространяется автоматически до конца строки. Если точка с запятой есть в строке, она будет частью последовательности замещения (*replacement*) и также появится во всех случаях подстановки.

Операторы

Когда мы ознакомились с переменными и константами, мы можем начать оперировать с ними, используя *операторы*. То, что дальше – это полный список операторов. В данный момент, похоже, нет нужды знать их все, но они все перечислены здесь, чтобы на них можно было ссылаться.

Оператор присваивания (=)

Оператор присваивание присваивает значение переменной.

```
x = 5;
```

Это выражение присваивает целое значение `5` переменной `x`. Операция присваивания всегда имеет место справа налево, и никогда не наоборот:

```
x = y;
```

Это выражение присваивает переменной `x` значение, содержащееся в переменной `y`. Значение `x` в этот момент теряется и заменяется значением `y`.

Учтите, что мы присвоили значение `y` переменной `x` только в момент операции присваивания. То есть, если `y` изменится позже, это не изменит значение `x`.

Например, давайте взглянем на следующий код – я включил изменение содержимого переменных в комментарии:

```
1  // assignment operator
2  #include <iostream>
3  using namespace std;
4
5  int main ()
6  {
7    int a, b;         // a:?,  b:?
8    a = 10;           // a:10, b:?
9    b = 4;            // a:10, b:4
10   a = b;            // a:4,  b:4
11   b = 7;            // a:4,  b:7
12
13   cout << "a:";
14   cout << a;
15   cout << " b:";
16   cout << b;
17 }
```

```
a:4 b:7
```

Эта программа выводит на экран окончательные значения `a` и `b` (4 и 7 соответственно). Заметьте, что `a` не была затронута последней модификацией `b`, хотя мы объявили `a = b` ранее.

Операции присваивания – это выражения, которые могут вычисляться. Что означает, что присваивание само имеет значение, и для фундаментальных типов это значение присваивается при операции. Например:

```
y = 2 + (x = 5);
```

В этом выражении `y` присваивается результат сложения 2 и значение другого выражения присваивания (которое имеет само значение 5). Это приблизительно эквивалентно:

```
x = 5;
y = 2 + x;
```

С окончательным результатом присваивания 7 для `y`.

Следующее выражение также правильно в C++:

```
x = y = z = 5;
```

Это присваивает 5 всем трём переменным: `x`, `y` и `z`; всегда справа налево.

Арифметические операторы (+, -, *, /, %)

Пять арифметических операций, поддерживаемые в C++, это:

оператор	описание
+	сложение
-	вычитание
*	умножение
/	деление
%	остаток

Операции сложения, вычитания, умножения и деления соотносятся буквально с их математическими операторами. Последний *modulo operator*, представленный знаком процента (`%`), даёт остаток от деления двух чисел. Например:

```
x = 11 % 3;
```

результат в переменной `x` – это 2, поскольку при делении 11 на 3 получается 3 с остатком 2.

Сложное присваивание (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)

Операторы сложного присваивания изменяют текущее значение переменной, производя над ней операции. Они эквивалентны присваиванию результата операции первому операнду:

выражение	эквивалентно...
`y += x;`	`y = y + x;`
`x -= 5;`	`x = x - 5;`
`x /= y;`	`x = x / y;`
`price *= units + 1;`	`price = price * (units+1);`

и так для всех сложных операторов присваивания. Например:

```
// compound assignment operators
#include <iostream>
using namespace std;

int main ()
{
```

```
5
```

```
  int a, b=3;
  a = b;
  a+=2;        // эквивалентно a=a+2
  cout << a;
}
```

Инкремент и декремент (++, --)

Некоторые выражения можно существенно сократить: оператор увеличения (`++`) и оператор уменьшения (`--`) увеличивает или уменьшает на единицу значение, хранимое в переменной. Они эквиваленты `+=1` и `-=1` соответственно. То есть:

```
++x;
x += 1;
x = x+1;
```

все эти примеры эквивалентны по функциональности; они увеличивают на единицу значение `x`.

В ранних компиляторах Си три предыдущих выражения могли производить разный исполняемый код, зависящий от того, какой из вариантов использован. Сегодня оптимизация кода обычно производится автоматически компилятором, так что все три выражения дадут одинаковый исполняемый код.

Особенность этих операторов в том, что можно использовать оба и как префикс, и как суффикс. Это означает, что они могут быть записаны либо перед именем переменной (`++x`), либо после него (`x++`). Хотя в простом выражении, похожем на `x++` или `++x`, оба имеют совершенно одинаковое значение; в других выражениях, где вычисляется результат операций инкремента или декремента, они могут сильно различаться. В случае, если оператор используется с префиксом (`++x`), выражение рассчитывается, когда значение `x` уже увеличено. С другой стороны, в случае, когда используется суффикс (`x++`), значение также увеличивается, но выражение вычисляется до увеличения x. Отметьте разницу:

Пример 1	Пример 2

`x = 3;`
`y = ++x;`
`// x содержит 4, y содержит 4`
	`x = 3;`
	`y = x++;`
	`// x содержит 4, y содержит 3`

В *Примере 1* значение, присваиваемое `y`, это значение после того, как `x` было увеличено. А в *Примере 2* значение `x` было увеличено после присвоения.

Операторы отношения и сравнения (==, !=, >, <, >=, <=)

Два выражения могут сравниваться, используя операторы отношений и эквивалентности. Например, чтобы узнать, равны ли два значения или одно из них больше, чем другое.

Результат этой операции либо истина (true), либо ложь (false), то есть, это булевы значения.

Операторы отношений в C++:

оператор	описание
==	Эквивалентно
!=	Не эквивалентно
<	Менее, чем
>	Более, чем
<=	Менее, чем или эквивалентно

>=	Более, чем или эквивалентно

Вот несколько примеров:

```
(7 == 5)      // получится false
(5 > 4)       // получится true
(3 != 2)      // получится true
(6 >= 6)      // получится true
(5 < 5)       // получится false
```

Конечно, сравниваться могут не только числовые константы, но и любые значения, включая, конечно, переменные. Положим, что `a=2`, `b=3` и `c=6`, тогда:

```
(a == 5)      // получится false, поскольку a не равно 5
(a*b >= c)    // получится true, поскольку (2*3 >= 6) это true
(b+4 > a*c)   // получится false, поскольку (3+4 > 2*6) это false
((b=2) == a) // получится true
```

Будьте внимательны! Оператор присваивания (оператор `=`, с одним значком равно) – это не то же самое, что оператор эквивалентности (оператор `==`, с двумя значками равно); первый (`=`) присваивает значение справа переменной, которая слева, а другой (`==`) сравнивает, будут ли значения по обе стороны от оператора равны. Поэтому в последнем выражении (`(b=2) == a`) мы сначала присваиваем значение `2` переменной `b`, а затем мы сравниваем её с `a` (которая тоже хранит значение 2), получая `true`.

Логические операторы (!, &&, ||)

Оператор «`!`» – это оператор C++ для булева оператора NOT (НЕ). Он имеет только один операнд справа, и инвертирует его, производя `false`, если его операнд `true`, и `true`, если операнд `false`. По существу, этот оператор возвращает противоположное булево значение, имевшееся у операнда. Например:

```
!(5 == 5)    // получится false, поскольку выражение справа (5 == 5) true
!(6 <= 4)    // получится true, поскольку (6 <= 4) будет false
!true        // получится false
!false       // получится true
```

Логические операторы `&&` и `||` используются, когда оцениваются два выражения для получения единственного результата отношения. Оператор `&&` относится к булевой логической операции AND (И), которая даёт `true`, если оба оперенда `true`, и `false` в противном случае. Следующая таблица показывает результат операции `&&` применительно к выражению `a&&b`:

&& OPERATOR (and)		
a	**b**	**a && b**
true	true	true
true	false	false
false	true	false
false	false	false

Оператор `||` относится к булевой логической операции OR (ИЛИ), которая даёт `true`, если хотя бы один из операндов `true`, а результат `false` будет тогда, когда оба операнда `false`. Вот возможные результаты `a||b`:

|| OPERATOR (or)		
a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

Например:

```
( (5 == 5) && (3 > 6) )  // получится false ( true && false )
( (5 == 5) || (3 > 6) )  // получится true ( true || false )
```

При использовании логических операторов C++ вычисляет только то, что необходимо слева направо до формирования результата отношения, игнорируя остальное. Поэтому в последнем примере (`(5==5)||(3>6)`) C++ вычисляет вначале, будет ли `5==5` означать `true`, а если так, то никогда не станет проверять, будет ли `3>6` означать `true` или нет. Это известно как *short-circuit evaluation (закороченное вычисление)*, и работает оно так же для следующих операторов:

оператор	short-circuit
`&&`	Если выражение слева `false`, комбинированный результат будет `false` (выражение справа не рассматривается).
`||`	Если выражение слева `true`, комбинированный результат будет `true` (выражение справа не рассматривается).

Самое важное, когда выражение справа имеет побочный эффект, такой как изменение значения:

```
if ( (i<10) && (++i<n) ) { /*...*/ }   // заметьте, что условие инкрементирует i
```

Здесь комбинированное выражение будет увеличивать `i` на единицу, но только, если условие слева `&&` будет `true`, поскольку в противном случае условие справа (`++i<n`) не будет рассматриваться.

Условный тернарный оператор (?)

Условный оператор вычисляет выражение, возвращая одно значение, если результат вычисления `true`, и другое, если выражение `false`. Его синтаксис:

```
condition ? result1 : result2
```

Если условие `true`, результат становится `result1`, а иначе `result2`.

```
7==5 ? 4 : 3      // получится 3, поскольку 7 не эквивалентно 5.
7==5+2 ? 4 : 3    // получится 4, поскольку 7 эквивалентно 5+2.
5>3 ? a : b       // получится значение a, поскольку 5 больше, чем 3.
a>b ? a : b       // получится a или b, в зависимости, что больше.
```

Например:

```
// conditional operator
#include <iostream>
using namespace std;

```

```
7
```

```
5  int main ()
6  {
7    int a,b,c;
8
9    a=2;
10   b=7;
11   c = (a>b) ? a : b;
12
13   cout << c << '\n';
14 }
```

В этом примере `a` было равно 2, а `b` было 7, и выражение было определено (`a>b`) как не `true`, так что первое значение, заданное после знака вопроса, было отвергнуто в пользу второго значения (после двоеточия), которое было `b` (со значением 7).

Оператор запятой (,)

Оператор запятой (`,`) используется для разделения двух или более выражений, которые там, где ожидается только одно выражение. Когда несколько выражений рассчитываются, только самое правое выражение будет результирующим.

Например, следующий код:

```
a = (b=3, b+2);
```

начинается с присваивания значения 3 переменной `b`, а затем присваивается `b+2` переменной `a`. Таким образом, в завершение переменная `a` будет содержать значение 5, хотя переменная `b` будет иметь значение 3.

Побитовые операторы (&, |, ^, ~, <<, >>)

Побитовые операторы модифицируют переменные, учитывая битовый набор, который представляет значения, хранящиеся в них.

оператор	asm эквивалент	описание
&	AND	Побитовое AND
|	OR	Включающее побитовое OR
^	XOR	Исключающее побитовое OR
~	NOT	Унарное дополнение (инверсия битов)
<<	SHL	Сдвиг битов влево
>>	SHR	Сдвиг битов вправо

Операторы явного приведения типов

Операторы явного приведения типов позволяют преобразовать значение данного типа в значение другого типа. Есть несколько способов сделать это в C++. Простейший, который был унаследован от языка Си, предварить выражение для преобразования в новый тип заключением в скобки (()):

```
1 int i;
2 float f = 3.14;
3 i = (int) f;
```

Предыдущий код преобразует число с плавающей точкой `3.14` в целое значение (`3`); остаток теряется. Здесь оператор приведения был `(int)`. Другой способ сделать это в C++ – это использовать предварение функциональной записи выражения, предназначенного для преобразования, типом и заключением выражения в скобки:

```
i = int (f);
```

Оба способа приведения типов допустимы в C++.

sizeof

Этот оператор принимает один параметр, который может быть либо типом, либо переменной, а результатом станет размер в байтах этого типа или объекта:

```
x = sizeof (char);
```

Здесь `x` принимает значение `1`, поскольку `char` – это тип с размером в один байт.

Значение, возвращаемое `sizeof` – это константа в процессе компиляции, так что это всегда определено до выполнения программы.

Другие операторы

Позже в этом руководстве мы познакомимся ещё с операторами подобно упоминавшимся указателям или специфическим для объектно-ориентированного программирования.

Приоритет операторов

Единичное выражение может иметь много операторов. Например:

```
x = 5 + 7 % 2;
```

В C++ выражение выше всегда даст 6 для переменной `x`, поскольку оператор `%` имеет приоритет выше, чем оператор `+`, и всегда вычисляется раньше. Части выражения могут заключаться в скобки для переопределения этого порядка приоритета, или чтобы сделать яснее желаемый результат. Отметьте разницу:

```
1 x = 5 + (7 % 2);     // x = 6 (то же, что и без скобок)
2 x = (5 + 7) % 2;     // x = 0
```

От наибольшего до наименьшего приоритета операторы C++ работают в следующем порядке:

Уровень	Группа приоритета	Оператор	Описание	Группирование
1	Scope	`::`	Определитель границы	Слева направо
2	Postfix (unary)	`++ --`	Постфиксный инкремент / декремент	Слева направо
		`()`	Функциональные формы	
		`[]`	Список индексов	
		`. ->`	Доступ к членам класса	
3	Prefix (unary)	`++ --`	Префиксный инкремент / декремент	Справа налево
		`~ !`	побитовое NOT / логическое NOT	
		`+ -`	Унарный перфикс	
		`& *`	ссылка / разыменование	
		`new delete`	размещение / освобождение	
		`sizeof`	Параметр блока	
		(*type*)	Си стиль приведения типа	
4	Pointer-to-member	`.* ->*`	Доступ к указателю	Слева направо
5	Arithmetic: scaling	`* / %`	Умножение, деление, остаток	Слева направо
6	Arithmetic: addition	`+ -`	Сложение, вычитание	Слева направо
7	Bitwise shift	`<< >>`	Сдвиг влево, сдвиг вправо	Слева направо
8	Relational	`< > <= >=`	Операторы сравнения	Слева направо
9	Equality	`== !=`	Равно / не равно	Слева направо
10	And	`&`	Побитовое AND	Слева направо
11	Exclusive or	`^`	Побитовое XOR	Слева направо
12	Inclusive or	`|`	Побитовое OR	Слева направо
13	Conjunction	`&&`	Логическое AND	Слева направо
14	Disjunction	`||`	Логическое OR	Слева направо
15	Assignment-level expressions	`= *= /= %= += -= >> = <<= &= ^= |=`	Присваивание / сложное присваивание	Справа налево
		`?:`	Условный оператор	
16	Sequencing	`,`	Разделительная запятая	Слева направо

Когда в выражении два оператора с одинковым уровнем приоритета, группирование (*grouping)* определяет, который из них выполняется первым: либо слева на право, либо справа налево.

Заключение всех подформул в скобки (даже тех, что не требуют этого по соображениям приоритета) улучшает читаемость кода.

Базовый ввод/вывод

Примеры программ в предыдущих разделах мало взаимодействовали с пользователем, если вообще это делали. Они просто выводили простые значения на экран. Но стандартная библиотека поддерживает много дополнительных способов для взаимодействия с пользователем через её возможности вввода/вывода. В этом разделе будет представлена краткая инструкция по применению наиболее полезных из них.

C++ использует удобную абстракцию, названную *streams (потоки)* для выполнения операций последовательного ввода и вывода в такой среде, как экран, клавиатура или файл. Поток (*stream)* – это объект, куда программа может либо вставить, либо откуда может извлечь символы. Нет необходимости знать детали о среде, ассоциированной с потоком или любыми его внутренними описаниями реализации. Всё, что нам следует знать, это то, что потоки – источники/места назначения символов, и что эти символы поставляются/получаются последовательно (то есть, один за другим).

Стандартная библиотека определяет поддержку объектов потока, которые могут быть использованы для доступа, что подразумевает стандартные источники и места назначения для символов, поддержанные оборудованием, где работает программа:

поток	описание
`cin`	Стандартный поток ввода
`cout`	Стандартный поток вывода
`cerr`	Стандартный поток ошибок (вывода)
`clog`	Стандартный поток регистрации (вывода)

Мы собираемся более детально рассмотреть только `cout` и `cin` (читается си-аут и си-ин, стандартные потоки вывода и ввода); `cerr` и `clog` – это тоже потоки вывода, которые работают подобно `cout`, но с отличием, касающимся идентификации потока для специальных целей: сообщений об ошибках и журналирования. Которые во многих случаях, чаще всего при установке оборудования, реально делают то же самое: они выводят на экран, хотя они могут быть и индивидуально перенаправлены

Стандартный вывод (cout)

Для большинства программ стандартным оборудованием для вывода по умолчанию является экран, а для объекта потока C++ определение для доступа - это `cout`.

Для форматированных операций вывода используется `cout` совместно с *оператором вставки* (*insertion operator)*, который записывается как `<<` (то есть, два заначка «меньше, чем»).

```
1 cout << "Output sentence"; // печатает Output sentence на экране
2 cout << 120;               // печатает число 120 на экране
3 cout << x;                 // печатает значение x на экране
```

Оператор `<<` вставляет данные, которые следуют за ним, в поток, предшествующий ему. В примере выше он вставляет литерал строку `Output sentence`, число `120` и значение переменной `x` в стандартный выходной поток `cout`. Заметьте, что предложение в первом случае заключено в двойные кавычки ("), поскольку этот литерал – строка, тогда как в последнем примере `x` таковым не является. Двойные кавычки – это то, что вносит различие. Когда текст заключён в кавычки, он печатается буквами, а когда их нет, текст интерпретируется как идентификатор переменной, и вместо текста выводится значение переменной. Например, эти две фразы имеют разный результат:

```
1 cout << "Hello";  // печатает Hello
2 cout << Hello;    // печатает содержимое переменной Hello
```

Несколько вставок операторов (<<) можно связать в едином выражении:

```
cout << "This " << " is a " << "single C++ statement";
```

Это последнее выражение будет печатать текст `This is a single C++ statement`. Подобное связывание вставок полезно при чередовании литералов и переменных в едином выражении:

```
cout << "I am " << age << " years old and my zipcode is " << zipcode;
```

Придав переменной *age* значение 24 и переменной *zipcode* значение 90064, мы получим вывод предыдущего выражения:

```
I am 24 years old and my zipcode is 90064
```

А вот то, что не будет сделано автоматически, не будет добавлен конец строки, если только не сделать этого. Например, положим, следующие два выражения вводятся в `cout`:

```
cout << "This is a sentence.";
cout << "This is another sentence.";
```

Ввод будет сделан в одну строку без разрыва между предложениями, подобно следующему:

```
This is a sentence.This is another sentence.
```

Чтобы предотвратить это, нужно вставить символ новой строки в то место, где строки должны прерваться. В C++ символ новой строки может быть задан как `\n` (то есть, обратная косая черта с последующей строчной `n`). Например:

```
1 cout << "First sentence.\n";
2 cout << "Second sentence.\nThird sentence.";
```

Это создаст следующий вывод:

```
First sentence.
Second sentence.
Third sentence.
```

Альтернативно можно использовать и манипулятор `endl` для этой же цели. Например:

```
1 cout << "First sentence." << endl;
2 cout << "Second sentence." << endl;
```

Что напечатает:

```
First sentence.
Second sentence.
```

Манипулятор `endl` создаёт символ новой строки, как и вставка `'\n'`, но он также и меняет поведение: буфер потока (если есть) очищается, что означает – вывод запрашивает физическую запись в устройство, если это уже не сделано. Это влияет, в основном, на *полную буферизацию* потока, а `cout` (как правило) не *полностью буферизует* поток. Следовательно, не плохая идея использовать `endl` только тогда, когда нужна очистка потока, а `'\n'` в противном случае. Имейте в виду, что операция очистки привносит некоторые издержки, а некоторые устройства и тормозят вывод.

Стандартный ввод (cin)

В большинстве программ по умолчанию стандартным устройством ввода служит клавиатура, а в C++ объектом потока, определённым для доступа к ней, является `cin`.

Для форматирования операции `cin` используется совместно с оператором ввода, который записывается как `>>` (то есть, два значка «больше, чем»). За этим оператором следует переменная, в которой сохранятся вводимые данные. Например:

```
1 int age;
2 cin >> age;
```

Первое утверждение объявляет переменную целого типа `int` , названную `age`, а второе извлекает из `cin` значение, хранимое в ней. Эта операция заставляет программу ждать ввода из `cin`; обычно это означает, что программа будет ждать, что пользователь введёт какую-то последовательность с клавиатуры. В данном случае отметьте, что символы с клавиатуры передаются программе только тогда, когда нажата клавиша `ENTER` (или `RETURN`). В тот момент, когда встречается оператор `cin`, программа будет ждать столько, сколько нужно, чтобы какой-то ввод был осуществлён.

Оператор ввода `cin` использует тип переменной после оператора `>>`, чтобы определить, как ему интерпретировать символы, прочитываемые при вводе. Если это целое, предполагаемый формат – это последовательность цифр, если же строка, последовательность символов и т.д.

```
1  // i/o example
2
3  #include <iostream>
4  using namespace std;
5
6  int main ()
7  {
8    int i;
9    cout << "Пожалуйста, введите целое
10 число: ";
11   cin >> i;
12   cout << "Число, которое вы ввели "
13 << i;
14   cout << ", а его удвоение " << i*2
   << ".\n";
   return 0;
}
```

```
Пожалуйста, введите целое число: 702
Число, которое вы ввели 702, а его
удвоение 1404.
```

Примечание: вы можете использовать приведённые в руководстве программы, но будьте осторожны с текстом на русском языке, который при переводе сделан для лучшего понимания происходящего, но может неверно отображаться в консоли.

Как вы видите, вывод из `cin` означает выполнение задания по получению ввода от стандартного устройства ввода довольно просто и ясно. Но этот метод также имеет и большой изъян. Что случится, если пользователь введёт нечто, что не получится распознать как целое? В этом случае операция ввода провалится. И это, по определению, позволит программе продолжать работать без задания значения переменной `i`, производя непредсказуемый результат, если переменная `i` используется позже.

Это очень примитивное поведение программы. Большинство программ ожидаемо ведут себя вне зависимости от того, что введут пользователи, поддерживая надлежащим образом неверные значения. Только очень простые программы будут полагаться на значения, полученные непосредственно от `cin` без дальнейшей проверки. Чуть позже мы увидим, как можно использовать *строковые потоки* (*stringstreams)* для улучшения контроля над пользовательским вводом.

Ввод `cin` можно тоже сцеплять, запрашивая более одного данного в единственном выражении:

```
cin >> a >> b;
```

Это эквивалентно:

```
cin >> a;
cin >> b;
```

В обоих случаях пользователь обязан ввести два значения, одно для переменной `a`, а второе для переменной `b`. Любого рода пробелы используются для разделения двух последовательных операций ввода; это может быть и пробел, и табуляция, и символ новой строки.

cin и строки

Оператор ввода может использоваться с `cin` для получения строк символов так же, как и для ввода фундаментальных типов данных:

```
string mystring;
cin >> mystring;
```

Однако ввод с `cin` всегда учитывает пробелы (собственно пробелы, табуляции, новая строка...), как прерывание вводимого значения, и, таким образом, ввод строки означает всегда ввод одного слова, а не фразы или целого предложения.

Чтобы получить целую строку из `cin`, есть функция, названная `getline`, которая берёт поток (`cin`) в качестве первого аргумента, а строковую переменную в качестве второго. Например:

```
// cin with strings
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string mystr;
  cout << "Как ваше имя? ";
  getline (cin, mystr);
  cout << "Привет " << mystr <<
".\n";
  cout << "Какая ваша любимая
команда? ";
  getline (cin, mystr);
  cout << "Мне нравится " << mystr <<
" тоже!\n";
  return 0;
}
```

```
Как ваше имя? Homer Simpson
Привет Homer Simpson.
Какая ваша любимая команда? The
Isotopes
Мне нравится The Isotopes тоже!
```

Заметьте, как в обоих вызовах `getline`, мы используем одинаковый идентификатор строки (`mystr`). Что программа делает при втором вызове, это просто замещение предыдущего содержания новым, введённым во второй раз.

Стандартное поведение, которое большинство пользователей ожидает от консольной программы, это то, что каждый раз, когда программа запрашивает от пользователя ввод, пользователь

вводит что-то в поле, а затем нажимает ENTER (или RETURN). Скажем так, обычно ввод предполагает строки в консольных программах, что может быть выполнено с использованием getline для получения ввода от пользователя. Таким образом, пока не возникнет проблем, вы должны использовать getline для ввода в ваших консольных программах вместо ввода из cin.

Строковый поток (stringstream)

Стандартный файл заголовка <sstream> определяет тип, называемый stringstream, который позволяет строки обрабатывать как поток, и этим операциям ввода или вставки из/в строки так же, как выполняются cin и cout. Эти средства весьма полезны для конвертации строк в числовые значения и наоборот. Например, в плане извлечения целого из строки, мы можем написать:

```
string mystr ("1204");
int myint;
stringstream(mystr) >> myint;
```

Это декларирует string с инициализацией значением "1204" и объявляет переменную типа int. Затем третья строка использует эту переменную для извлечения из stringstream созданной строки. Эта часть кода хранит числовое значение 1204 в переменной, названной myint.

```
// stringstreams
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

int main ()
{
  string mystr;
  float price=0;
  int quantity=0;

  cout << "Введите цену: ";
  getline (cin,mystr);
  stringstream(mystr) >> price;
  cout << "Введите количество: ";
  getline (cin,mystr);
  stringstream(mystr) >> quantity;
  cout << "Общая стоимость: " <<
price*quantity << endl;
  return 0;
}
```

```
Введите цену: 22.25
Введите количество: 7
Общая стоимость: 155.75
```

В этом примере мы получаем числовые значения из *standard input* косвенно: вместо извлечения числовых значений прямо из cin мы берём строку из него в строковый объект (mystr), а затем мы извлекаем значения из этой строки в переменные price и quantity. Поскольку это числовые переменные, к ним могут применяться арифметические операции, такие как умножение, чтобы получить общую стоимость.

С этим подходом получения всей строки и извлечения её содержимого, мы разделяем процесс получения пользователем ввода и интерпретации его данных, что даёт процессу ввода быть тем, что ожидает пользователь, а в то же время мы усиливаем контроль над преобразованием содержимого в полезные данные с помощью программы.

Структура программы

Операторы и управление потоком

Простые C++ операторы – это индивидуальные инструкции программы подобные объявлению переменной и выражениям, показанным в предыдущих разделах. Они всегда завершаются точкой с запятой (`;`) и выполняются в том порядке, в каком они появляются в программе.

Но программы не ограничиваются линейной последовательностью операций. В процессе выполнения программа может повторять сегменты кода, может принимать решения и осуществлять ветвление. Для этих целей C++ поддерживает операции управления потоком, которые служат для задания того, что должно быть сделано нашей программой, когда и при каких обстоятельствах.

Многие из операций управления потоком в этих разделах требуют основных (под)операций в качестве части их синтаксиса. Подобный оператор может быть либо простым C++ выражением, как единичная инструкция, прекращаемая точкой с запятой (`;`), так и сложным выражением. Сложное выражение – это группа операций, каждая завершается своей точкой с запятой, но всё собирается вместе в один блок, заключённый в фигурные скобки `{}`:

```
{ statement1; statement2; statement3; }
```

Весь блок определяет один оператор (составленный из нескольких операторов). Поскольку общее выражение является частью синтаксиса оператора управления потоком, это может быть и простым выражением, и сложным.

Операторы выбора: if и else

Ключевое слово `if` используется для выполнения выражения или блока, если, и только если, условие выполняется. Синтаксис:

```
if (условие) выражение
```

Здесь `условие` – это выражение, которое вычисляется. Если это `условие` истинно, `выражение` выполняется. Если оно ложно, `выражение` не выполняется (оно просто игнорируется), а программа продолжается дальше, после всего блока оператора.

Например, следующий фрагмент кода выводит сообщение `(x is 100)` только в том случае, если значение, которое хранится в переменной `x`, действительно 100:

```
1 if (x == 100)
2   cout << "x is 100";
```

Если `x` не равно 100, эта операция игнорируется, и ничего не будет выведено.

Если вы хотите включить более одного выражения для выполнения, когда условие выполняется, эти выражения должны быть заключены в фигурные скобки (`{}`), формируя блок:

```
1 if (x == 100)
2 {
3    cout << "x is ";
4    cout << x;
5 }
```

Как правило, отступы и переход на новую строку в коде никак не сказываются на результате, так что код выше эквивалентен:

```
if (x == 100) { cout << "x is "; cout << x; }
```

Выбор выражения с `if` может также задать то, что случится, когда условие не выполняется, если использовать ключевое слово `else` для вставки альтернативного выражения. Синтаксис таков:

```
if (условие) выражение1 else выражение2
```

где `выражение1` выполняется в случае истины при проверке условия, а если нет, то выполняется `выражение2`.

Например:

```
if (x == 100)
  cout << "x is 100";
else
  cout << "x is not 100";
```

Это выводит `x is 100`, если действительно x имеет значение 100, но если это не так, выводится `x is not 100`.
Структура из нескольких if + else может быть составлена с целью проверки нескольких значений. Например:

```
if (x > 0)
  cout << "x is positive";
else if (x < 0)
  cout << "x is negative";
else
  cout << "x is 0";
```

Этим ввыводится, будет ли x положительно, отрицательно или равно нулю, как результат структуры из двух if-else. Вновь, вполне возможно выполнить более одного выражения для каждого случая, если группировать их в фигурных скобках `{}`.

Повторяющиеся выражения (циклы)

Циклы повторяют выражения некоторое количество раз, или пока не будет выполнено некоторое условие. Они вводятся ключевыми словами `while`, `do` и `for`.

Цикл while

Простейшей разновидностью цикла является цикл while. Его синтасис:

```
while (выражение) операция
```

Цикл while просто повторяет `операцию` пока `выражение` истинно. Если после выполнения `операции`, `выражение` перестало быть истинным, цикл заканчивается, а программа продолжает выполнение оставшегося кода после цикла. Например, давайте взглянем на обратный отсчёт, использующий цикл while:

```
// custom countdown using while
#include <iostream>
using namespace std;

int main ()
{
  int n = 10;

```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
liftoff!
```

```
  while (n>0) {
    cout << n << ", ";
    --n;
  }

  cout << "liftoff!\n";
}
```

Первое выражение в `main` задаёт *n* значение 10. Это первое число при обратном отсчёте. Затем начинается цикл while: если это значение удовлетворит условию `n>0` (*n* больше нуля), тогда блок, следующий за условием, выполняется и повторяется до тех пор, пока условие (`n>0`) остаётся истинным.

Весь процесс предыдущей программы можно интерпретировать согласно следующему описанию (начинающемуся с `main`):

1. `n` получает значение
2. Условие в `while` проверяется (`n>0`). В этом месте есть две возможности:
   - условие истинно: выражение выполняется (до шага 3)
   - условие ложно: выражение игнорируется, и программа продолжается после него (до шага 5)
3. Выполняются выражения:
   `cout << n << ", ";`
   `--n;`
   (выводится значение `n` и `n` уменьшается на 1)
4. Конец блока. Программа возвращается автоматически к шагу 2.
5. Продолжается программа сразу после блока:
   выводится `liftoff!`, и программа завершается.

Смысл обсуждения цикла *while* в том, что цикл должен завершиться, то есть, выражение в теле цикла должно менять значение, проверяемое в условии, так, чтобы заставить его стать ложным в какой-то момент. В противном случае цикл будет продолжаться бесконечно. В нашем случае цикл содержит `--n`, что уменьшает значение переменной (`n`), оцениваемое в условии, на единицу – это, в конечном счёте, сделает условие (`n>0`) ложным после некоторого числа проходов цикла. Более точно, после 10 проходов `n` становится равно 0, сделав условие ложным, что и завершает цикл while.

Заметьте, что сложность этого цикла банальна для компьютера, а весь обратный отсчёт выполняется мгновенно без каких-либо пауз между элементами счёта (если вам интересно, можете посмотреть `sleep_for` – пример обратного отсчёта с паузами).

Цикл do-while

Очень похожий цикл – цикл do-while, синтаксис которого:

```
do выражение while (условие);
```

Он ведёт себя подобно циклу *while*, исключая то, что `условие` проверяется после выполнения `выражения`, а не до этого, что гарантирует хотя бы одно выполнение `выражения`, даже если `условие` никогда не выполняется. Например, следующая программа повторяет любой текст, вводимый пользователем, пока тот не введёт goodbye:

```
// echo machine
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string str;
```

```
Введите текст: hello
Вы ввели: hello
Введите текст: who's there?
Вы ввели: who's there?
Введите текст: goodbye
Вы ввели: goodbye
```

```
  do {
    cout << "Введите текст: ";
    getline (cin,str);
    cout << "Вы ввели: " << str <<
'\n';
  } while (str != "goodbye");
}
```

Цикл *do-while* обычно применяется вместо цикла *while*, когда нужно, чтобы `выражение` выполнилось хотя бы один раз. А проверка условия в конце цикла подразумевает, что то, что проверяется в условии, определялось внутри самого цикла. В предыдущем примере ввод пользователем внутри блока определяет, будет ли цикл завершён. Таким образом, даже если пользователь хочет завершить цикл как можно раньше, введя `goodbye`, блок внутри цикла должен выполниться хотя бы однажды, чтобы осуществить ввод, а условие, фактически, может быть определено после выполнения блока.

Цикл for

Цикл `for` придуман для повторения выражения заданное количество раз. Его синтаксис:

```
for (инициализация; условие; инкремент) выражение;
```

Подобно циклу *while* этот цикл повторяет `выражение` пока `условие` истинно. Но дополнительно цикл *for* предоставляет место для хранения `инициализации` и выражения `инкремента`, осуществляемого до начала первого выполнения цикла и после каждого прохода соответственно. Пэтому весьма полезно в качестве переменных счётчика выбирать то, что проверяется в `условии`.

Всё это работает следующим образом:

1. `инициализация` выполняется. Обычно это объявление переменной счётчика и задание ей некоторого начального значения. Это выполняется одномоментно в начале цикла.
2. `условие` проверяется. Если оно истинно, цикл продолжается; иначе цикл завершается, а `выражение` пропускается до шага 5.
3. `выражение` выполняется. Как обычно, это может быть либо единственное выражение, либо блок выражений, заключённых в фигурные скобки `{ }`.
4. `инкремент` выполняется, а цикл возвращается к шагу 2.
5. цикл завершается: выполнение продолжается со следующего за циклом выражения.

Вот пример обратного счётчика, использующего цикл *for*:

```
// countdown using a for loop
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--) {
    cout << n << ", ";
  }
  cout << "liftoff!\n";
}
```

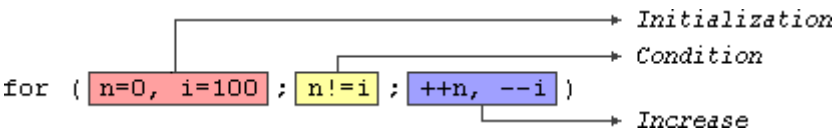
```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
liftoff!
```

Три поля в цикле *for* не обязательны. Они могут оставаться пустыми, но в любом случае требуется точка с запятой между ними. Например, `for (;n<10;)` – это цикл без *инициализации* и *инкремента* (эквивалентно циклу *while*); а `for (;n<10;++n)` – это цикл с *инкрементом*, но без *инициализации* (возможно, по причине того, что переменная была уже *инициализрована* перед циклом). Цикл без *условия* эквивалентен циклу, где условие всегда `true` (то есть, бесконечный цикл).

Поскольку каждое поле исполняется в отдельное время в жизни цикла, может быть полезным выполнять более одного выражения и для *инициализации*, и для *условия*, и для *выражения*. К сожалению, нет выражений кроме простейших, а их нельзя заменить блоками. Однако в выражениях можно использовать оператор запятой (`,`): этот оператор – разделитель выражений, и он может разделять множество выражений, где обычно ожидается только одно. Например, используя его, можно для цикла *for* поддерживать две переменные счёта, их инициализации и инкремента обеих:

```
for ( n=0, i=100 ; n!=i ; ++n, --i )
{
   // whatever here...
}
```

Этот цикл будет выполенен 50 раз, если ни `n` ни `i` не меняются внутри цикла:



`n` начинается со значения 0, а `i` с 100, условие `n!=i` (то есть, `n` не равно `i`). Поскольку `n` увеличивается на единицу, а `i` уменьшается на единицу при каждом проходе, условие цикла станет ложным после 50и повторений, когда и `n` и `i` станут равны 50.

Цикл for на основе интервала

Цикл *for* в этом случае имеет другой синтаксис, который используется исключительно с интервалами:

```
for ( объявление : интервал ) выражение;
```

Эта разновидность цикла for повторяется по всем элементам в `интервале`, где `объявление` задаёт некоторую переменную, способную принимать значения элементов в этом интервале. Интервалы – это последовательности элементов, включая массивы, контейнеры и любые другие типы, поддерживаемые функциями `begin` и `end`. Большинство этих типов пока не встречались в этом руководстве, но мы уже знакомились, по крайней мере, с одним из типов, строками, которые представляют последовательность символов.

Пример цикла for на основе интервала, использующий строки:

```
// range-based for loop
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string str {"Hello!"};
  for (char c : str)
  {
    std::cout << "[" << c << "]";
  }
  std::cout << '\n';
}
```

```
[H][e][l][l][o][!]
```

Заметьте, что перед двоеточием (`:`) в цикле объявляется переменная `char` (элементы строки все типа `char`). Затем мы используем переменную `c` в блоке выражений, представляющих значения каждого из элементов в интервале.

Этот цикл автоматический, он не требует явного объявления какой-либо переменной счётчика.

Циклы, основанные на интервале, обычно могут также использовать отслеживание типа элементов с помощью `auto`. Как правило, циклы, основанные на интервале как выше, можно записывать и так:

```
for (auto c : str)
  std::cout << "[" << c << "]";
```

Здесь тип `c` автоматически определяется типом элементов в `str`.

Операторы прыжков (Jump)

Инструкции прыжка (Jump) позволяют изменить ход программы, выполняя переход в заданное ими место.

Инструкция break

`break` завершает цикл, даже если условие для его завершения не было выполнено. Это может быть использовано для прерывания бесконечного цикла или для ускоренного завершения до естественного окончания цикла:

```
// break loop example
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--)
  {
    cout << n << ", ";
    if (n==3)
    {
      cout << "countdown aborted!";
      break;
    }
  }
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, countdown
aborted!
```

Инструкция continue

Инструкция `continue` приводит к тому, что программа пропускает оставшуюся часть цикла при текущем проходе, как если бы был достигнут конец блока выражений, заставляя начать следующий проход цикла. Например, давайте пропустим число 5 в нашем обратном счётчике:

```
// continue loop example
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--) {
    if (n==5) continue;
    cout << n << ", ";
  }
  cout << "liftoff!\n";
}
```

```
10, 9, 8, 7, 6, 4, 3, 2, 1,
liftoff!
```

Edit & Run

Оператор goto

`goto` позволяет сделать абсолютный прыжок в другое место программы. Этот безусловный прыжок игнорирует уровень вложенности и не вызывает автоматической очистки стека. Поэтому, подобную возможность следует использовать осторожно и желательно в том же блоке выражений, и, главным образом, при наличии локальных переменных.

Точка назначения идентифицируется *меткой* (*label)*, которая затем используется в качестве аргумента для оператора `goto`. *Метка* создаётся, как подходящий идентификатор, сопровождаемый двоеточием (`:`).

`goto` обычно считается элементом низкого уровня, не имеющим практического применения в современной парадигме высокоуровневого программирования, используемой в C++. Но в качестве примера предлагаем код программы обратного счётчика с циклом, использующим goto:

```
1  // goto loop example
2  #include <iostream>
3  using namespace std;
4
5  int main ()
6  {
7    int n=10;
8  mylabel:
9    cout << n << ", ";
10   n--;
11   if (n>0) goto mylabel;
12   cout << "liftoff!\n";
13 }
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
liftoff!
```

Другое выражение выбора: switch.

Синтаксис переключателя (switch) немного своеобразный. Его назначение – проверить значение среди нескольких постоянных констант. Это похоже на связку выражений `if-else`, но ограничивается константами. Самый типичный синтаксис это:

```
switch (выражение)
{
  case константа1:
     группа-выражений-1;
     break;
  case константа2:
     группа=выражений-2;
     break;
  .
  .
  .
  default:
     предопределённая-группа-выражений
}
```

Работает это следующим образом: `switch` выполняет `выражение` и проверяет, будет ли результат эквивалентен `константа1`; если это так, выполняется `группа-выражений-1` пока не будет найдена инструкция `break`. Когда обнаруживается инструкция `break`, программа перепрыгивает на конец всего оператора `switch` (закрывающая скобка).

Если выражение не равно `константа1`, тогда проверяется `констранта2`. Если обнаруживается равенство, выполняется `группа=выражений-2` пока не дойдёт до `break`, когда осуществляется прыжок до конца switch.

Наконец, если значение выражения не сходится с ранее заданными константами (может быть любое их количество), программа выполняет выражения после метки `default:` если она есть (то есть, она не обязательна).

Оба следующих фрагмента кода обнаруживают одинаковое поведение, демонстрируя эквивалентность if-else и switch операторов:

switch пример	if-else эквивалент

`switch (x) {`
`  case 1:`
`  cout << "x is 1";`
`  break;`
`  case 2:`
`  cout << "x is 2";`
`  break;`
`  default:`
`  cout << "value of x unknown";`
`  }`
	`if (x == 1) {`
	`  cout << "x is 1";`
	`}`
	`else if (x == 2) {`
	`  cout << "x is 2";`
	`}`
	`else {`
	`  cout << "value of x unknown";`
	`}`

Инструкция `switch` имеет довольно специфический синтаксис, унаследованный от ранних версий первых компиляторов Си, поскольку используются метки вместо блоков. Во многих типичных примерах использования (показанных выше) это означает, что приходится использовать выражение `break` после каждой группы выражений для каждой метки. Если `break` не включено, все выражения, следующие за case (включая то, что под любыми другими метками), также выполняются, пока не будет достигнут конец блока оператора switch или инструкции перехода (такой как `break`).

Если в примере выше теряется инструкция `break` после первой группы для случая один, программа не перепрыгнет автоматически к концу блока switch после вывода `x is 1`, и будет вместо этого продолжать выполнение выражений для случая два (то есть, выведет и `x is 2`). И так будет продолжаться до встречи с `break` или концом блока `switch`. Это делает лишним заключение выражений для каждого случая в скобки `{}`, и также может быть полезно выполнить ту же группу выражений для разных допустимых значений. Например:

```
1 switch (x) {
2   case 1:
3   case 2:
4   case 3:
5     cout << "x is 1, 2 or 3";
6     break;
7   default:
8     cout << "x is not 1, 2 nor 3";
9   }
```

Заметьте, что оператор `switch` ограничен при сравнении вычисленных выражений с метками тем, что они – константы. Нельзя использовать переменные или интервалы в качестве меток, поскольку они не являются правильными постоянными выражениями в C++.

Для проверки интервалов или значений, которые не являются постоянными, лучше использовать каскадную конструкцию из выражений `if` и `else if`.

Функции

Функции позволяют структурировать программы блоками кода, выполняющими индивидуальные задачи.

В C++ функция – это группа инструкций, которая имеет имя и к которой можно обращаться из разных точек программы. Наиболее общий синтаксис определения функции выглядит так:

```
type name ( parameter1, parameter2, ...) { statements }
```

Где:

- `type` – это тип значения, возвращаемого функцией.
- `name` – это идентификатор, по которому можно вызывать функцию.
- `parameters` (столько, сколько нужно): каждый параметр состоит из типа с последующим идентификатором, каждый параметр отделяется от других запятой. Каждый из параметров выглядит так же, как обычное объявление переменной (например: `int x`), и фактически действует внутри функции, как обычная переменная, но локальная, работающая внутри функции. Назначение параметров – позволить передать аргументы в функцию из места, откуда она вызывается.
- `statements` – это тело функции. Это блок выражений (инструкций), окаймлённый фигурными скобками { }, который задаёт то, что функция реально делает.

Давайте взглянем на пример:

```
1  // function example
2  #include <iostream>
3  using namespace std;
4
5  int addition (int a, int b)
6  {
7    int r;
8    r=a+b;
9    return r;
10 }
11
12 int main ()
13 {
14   int z;
15   z = addition (5,3);
16   cout << "Результат " << z;
17 }
```

```
Результат 8
```

Эта программа разделена на две функции: `addition` и `main`. Помните, что не имеет значения порядок, в котором функции определяются, в C++ программа всегда стартует с вызова функции `main`. Фактически `main` – это только функция, вызываемая автоматически, а код в других функциях выполняется только в том случае, когда функция вызывается из `main` (непосредственно или косвенно).

В примере выше функция `main` начинается с объявления переменной `z` типа `int`, а сразу за этим она выполняет первый вызов функции: она вызвает функцию `addition`. Вызов функции следует структуре очень похожей на её объявление. В примере выше вызов функции `addition` может быть сравнён с её определением несколькими строками выше:

```
int addition (int a, int b)
               ↑     ↑
z = addition (  5  ,  3  );
```

Параметры в объявлении функции имеют прямое соответствие с аргументами, передаваемыми при вызове функции. Вызов передаёт два значения 5 и 3 в функцию; это соотносится с параметрами a и b, объявленными для функции addition.

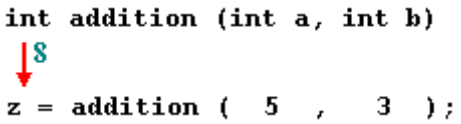
В этом месте, где функция вызывается из main, управление передаётся функции addition. Выполнение функции main приостанавливается, а возобновляется только после завершения работы функции addition. В тот момент, когда функция вызвается, значение обоих аргументов (5 и 3) копируются в локальные переменные int a и int b внутри функции.

Затем внутри addition другая локальная переменная, объявленная в ней (int r), с помощью выражения r=a+b, как результат a плюс b, присваивается r; в нашем случае, когда a равно 5 и b равно 3, r присваивается 8.

Заключительное выражение в функции:

```
return r;
```

Завершение работы функции addition возвращает управление обратно в то место программы, где функция вызвалась, в нашем случае в функцию main. Именно в этот момент программа переключает своё внимание на main, возвращаясь точно в ту точку, где была прервана вызовом функции addition. Но добавим, что addition возвращала тип, вычисленный при вызове и обретший значение. Это значение было задано в выражении return, завершившем addition: в нашем частном случае значение локальной переменной r, которая к этому времени для выражения return получила значение 8.



Таким образом, вызов addition – это выражение со значением, возвращаемым функцией, в нашем случае это значение 8, которое присвается переменной z. Это выглядит так, как если бы весь вызов функции (addition(5,3)) был заменён возвращаемым ею значением (то есть, 8).

Функция main просто выводит это значение с помощью:

```
cout << "The result is " << z;
```

Функция может вызываться множество раз в программе, а её аргументы не ограничены только литералами:

```
1  // function example
2  #include <iostream>
3  using namespace std;
4
5  int subtraction (int a, int b)
6  {
7    int r;
8    r=a-b;
9    return r;
10 }
11
12 int main ()
13 {
14   int x=5, y=3, z;
15   z = subtraction (7,2);
16   cout << "The first result is " << z
```

```
The first result is 5
The second result is 5
The third result is 2
The fourth result is 6
```

```
<< '\n';
  cout << "The second result is " <<
subtraction (7,2) << '\n';
  cout << "The third result is " <<
subtraction (x,y) << '\n';
  z= 4 + subtraction (x,y);
  cout << "The fourth result is " <<
z << '\n';
}
```

Подобно функции `addition` из предыдущего примера в этом определяется функция `subtract`, которая просто возвращает разность между её двумя параметрами. В этот раз `main` вызывает эту функцию несколько раз, демонстрируя возможные способы вызова функции.

Давайте проверим каждый из этих вызывов, держа в памяти то, что каждый вызов функции – это некое выражение, которое вычисляет возвращаемое значение. Вновь, вы может думать о ней, как о замещении её вызова возвращаемым значением:

```
z = subtraction (7,2);
cout << "The first result is " << z;
```

Если вы замените вызов функции значением, которое она возвращает (т.е., 5), то получите:

```
z = 5;
cout << "The first result is " << z;
```

С той же самой процедурой мы можем интерпретировать:

```
cout << "The second result is " << subtraction (7,2);
```

как:

```
cout << "The second result is " << 5;
```

поскольку 5 – это значение, возвращаемое `subtraction (7,2)`.

В случае:

```
cout << "The third result is " << subtraction (x,y);
```

аргументы, передаваемые функции subtraction – это переменные, а не литералы. И это тоже правильно, это тоже прекрасно работает. Функция вызывается со значениями `x` и `y`, имеющими на момент вызова: 5 и 3 соответственно, и возвращает в качестве результата 2.

Четвёртый вызов опять похож:

```
z = 4 + subtraction (x,y);
```

В этом вызове функции было только добавлено сложение и операнд в операции сложения. Вновь, результат то же, что и при замещении функции возвращаемым значением её работы: 6. Заметьте, что благодаря перестановочным свойствам сложения, выражение выше можно переписать так:

```
z = subtraction (x,y) + 4;
```

С тем же результатом. И ещё заметьте, что нет необходимости в точке с запятой после вызова функции, но, как всегда, в конце всего выражения это необходимо. Вновь, за этим легче просматривается логика, если заменить вызов функции возвращаемым значением:

```
z = 4 + 2;     // same as z = 4 + subtraction (x,y);
z = 2 + 4;     // same as z = subtraction (x,y) + 4;
```

Функция без типа. Использование void

Синтаксис, показанный выше для функций, это:

```
type name ( argument1, argument2 ...) { statements }
```

Требуемое объявление должно начинаться с типа. Это тип значения, возвращаемого функцией. Но что если функции не нужно возвращать значение? В этом случае используется тип `void`, который является специальным типом для представления отсутствия значения. Например, функция, которая просто выводит сообщение на экран, не нуждается в возвращении значения:

```
// void function example
#include <iostream>
using namespace std;

void printmessage ()
{
  cout << "I'm a function!";
}

int main ()
{
  printmessage ();
}
```

```
I'm a function!
```

`void` может также использоваться в списке параметров функции, явно показывая, что функция не принимает при вызове параметров. Например, `printmessage` может объявляться как:

```
void printmessage (void)
{
  cout << "I'm a function!";
}
```

В C++ пустой список параметров может использоваться вместо `void`, имея то же смысл, но использование `void` в аргументах было распространено в языке Си, где это требуется.

Нечто, что всегда не является необязательным – это скобки за именем функции, они обязательны и при объявлении функции, и при её вызове. И даже если функция не имеет параметров, хотя бы пустая пара скобок всегда добавляется к имени функции. Вот как вызывается функция `printmessage` в примере ранее:

```
printmessage ();
```

Скобки отличают функцию от другого рода объявлений или выражений. Следующее написание не сможет вызвать функцию:

```
printmessage;
```

Возвращаемое значение функции main

Вы могли заметить, что возвращаемое значение функции `main` это `int`, но большинство примеров в этом и предыдущих разделах в действительности не возвращает никакого значения из `main`.

Хорошо, вот в чём подвох: если выполнение функции `main` завершается нормально без появления выражения `return`, компилятор берёт на себя завершение функции скрытым выражением return:

```
return 0;
```

Заметьте, что это относится только к функции `main` и по историческим причинам. Все другие функции, что-то возвращающие, должны завершаться правильным выражением `return`, которое включает возвращаемое значение, даже если это и не используется.

Когда `main` возвращает ноль (либо явно, либо неявно), это интерпретируется окружением как успешное завершение программы. Функцией `main` могут возвращаться другие значения, а окружение даёт каким-то образом доступ к этим значениям вызвающей программе, хотя подобное поведение и не обязательно, и не всегда переносимо между платформами. Значения для `main`, которые каким-то образом гарантировано будут интерпретированы для всех платформ, это:

значение	описание
0	Успешное завершение программы
EXIT_SUCCESS	Завершение успешно (как и выше). Это значение определено в заголовочном файле <cstdlib>.
EXIT_FAILURE	Программа завершилась неудачно. Это значение определено в заголовочном файле <cstdlib>.

Поскольку наявное выражение `return 0;` для функции `main` является исключением, просто трюком, некоторые авторы считают хорошим правилом явно записывать это выражение.

Передача аргументов по значению и по ссылке

В функциях, которые мы видели раньше, аргументы всегда передавались *по значению*. Это означает, что когда функция вызывается, то, что передаётся в функцию, есть значения тех аргументов на момент вызова, которые копируются в переменные, представляющие параметры функции. Например, возьмём:

```
1 int x=5, y=3, z;
2 z = addition ( x, y );
```

В этом случае функции сложения передаются 5 и 3, которые есть копия значений `x` и `y` соответственно. Эти значения (5 и 3) используются для инициализации набора переменных, как параметров функции в её определении. Но любые изменения этих переменных внутри функции не скажутся на значениях этих переменных x и y вне её, поскольку переменные x и y сами не были переданы в функцию при её вызове, но только копии этих переменных.

```
int addition (int a, int b)
               ↑      ↑
z = addition (  5   ,   3  );
```

В некоторых случаях может оказаться полезным доступ из функции к внешним переменным. Чтобы это осуществить, аргументы могут передаваться в функцию *по ссылке*, а не *по значению*.

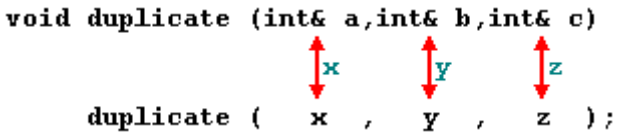
Например, функция `duplicate` в коде ниже дублирует значение трёх её аргументов, что приводит к изменению переменных, использованных в качестве аргументов при вызове:

```
// передача параметров по ссылке
#include <iostream>
using namespace std;

void duplicate (int& a, int& b, int& c)
{
  a*=2; // вспомните, a=a*2
  b*=2; // вспомните, b=b*2 и т.д.
  c*=2;
}

int main ()
{
  int x=1, y=3, z=7;
  duplicate (x, y, z);
  cout << "x=" << x << ", y=" << y << ", z="
<< z;
  return 0;
}
```

```
x=2, y=6, z=14
```

Чтобы получить доступ к её аргументам, функция объявляет свои параметры как *ссылки*. В C++ ссылки обозначаются амперсандом (`&`), следующим за типом параметра, как сделано у параметров, принимаемых функцией `duplicate`, в примере выше.

Когда переменная передаётся *по ссылке*, то, что передаётся, уже не копия, а сама переменная. Переменная, идентифицированная по параметру функции, становится, каким-то образом, связана с аргументом, передаваемым в функцию, а любое её изменение соответствующей локальной переменной в функции отражается на переменной, передаваемой в качестве аргумента при вызове функции.



Фактически `a`, `b` и `c` становятся псевдонимами аргументов, передаваемых в функцию при вызове (`x`, `y` и `z`), а любые изменения `a` в функции реально меняют переменную `x` вне функции. Любые изменения `b` модифицируют `y`, а изменения `c` меняют `z`. Вот почему, когда в примере выше функция `duplicate` меняет значения переменных `a`, `b` и `c`, затрагиваются значения `x`, `y` и `z`.

Если бы вместо определения функции duplicate как:

```
void duplicate (int& a, int& b, int& c)
```

было определение без знаков амперсанда:

```
void duplicate (int a, int b, int c)
```

то переменные передавались бы не *по ссылке*, а *по значению*, создавая копии этих переменных. В этом случае вывод программы показал бы значения переменных `x`, `y` и `z` без изменений (т.е., 1, 3 и 7).

Соображения эффективности и ссылки const

Вызовы функции с параметрами, передаваемыми по значению, приводит к копированию переменных. Это относительно экономная операция для фундаментальных типов как `int`, но если параметры оказываются больших составных типов, это может стать накладно. Например, взгляните на следующую функцию:

```
string concatenate (string a, string b)
{
  return a+b;
}
```

Эта функция берёт строки в качестве параметров (по значению) и возвращает результат их объединения. При передаче аргументов по значению, функция принудительно копирует аргументы `a` и `b`, передаваемые ей при вызове. А если это длинные строки, то будет копирование большого количества данных только для вызова функции.

Но этого копирования можно вполне избежать, если оба параметра сделаны *ссылками*:

```
string concatenate (string& a, string& b)
{
  return a+b;
}
```

Аргументы по ссылке не требуют копирования. Функция оперирует непосредственно со строками (псевдонимами), передаваемыми в качестве аргументов, и, чаще всего, это может означать передачу некоторых указателей в функцию. Благодаря этому версия `concatenate`, принимающая ссылки, более эффективна, чем версия, которая принимает значения, поскольку нет нужды копировать громоздкие строки.

С другой стороны, функции со ссылочными параметрами обычно воспринимаются как фунции, которые изменяют передаваемые аргументы, поскольку это то, для чего ссылочные параметры и предназначены.

Есть решение для функции, гарантирующее, что ссылочные параметры не будут модифицированы этой функцией. Это можно сделать, квалифицируя параметры как константы:

```
string concatenate (const string& a, const string& b)
{
  return a+b;
}
```

Квалифицируя их как `const`, функция запрещает изменять значения и `a`, и `b`, но может действительно получить доступ к значениям как ссылкам (псевдонимам аргументов) без необходимости копирования строк.

Таким образом, `const` ссылки поддерживают функциональность подобную передаче аргументов по значению, но повышают эффективность для параметров больших типов. Это то, почему они очень популярны в C++ для аргументов сложных типов. Заметьте, что при этом для большинства фундаментальных типов разницы в эффективности нет, а в некоторых случаях ссылки const могут оказаться и менее эффективными!

Функции Inline

Вызов функции обычно несёт некоторые издержки (использование стека для аргументов, прыжки и т.д.), таким образом, для очень коротких функций, может быть, более эффективно просто вставить код функции туда, откуда функция вызывается, вместо того, чтобы выполнять процесс

формального вызова функции.

Предварение функции объявлением `inline` передаёт компилятору информацию, что встроенное расширение предпочтительнее, чем обычный механизм вызова функции для конкретной функции. Это совсем не меняет поведение функции, но просто предлагает компилятору вставить код, генерируемый функцией, в то место, где функция вызывается, вместо обращения к обычному вызову функции.

Например, функция concatenate, показанная выше, может быть объявлена inline как:

```
inline string concatenate (const string& a, const string& b)
{
  return a+b;
}
```

Это информирует компилятор, что когда `concatenate` вызывается, для программы предпочтительнее, чтобы функция была встроена как расширение, а не выполнялся обычный вызов функции. `inline` задаётся только при объявлении функции, но никак не при её вызове.

Заметьте, что большинство компиляторов уже оптимизируют код, чтобы сгенерировать `inline` функции, когда они выглядят подходящими для увеличения эффективности, даже если они не отмечены спецификатором `inline`. Таким образом, этот спецификатор просто показывает компилятору, что предпочтительнее так обойтись с этой функцией, хотя компилятор не обязан это делать, а свободен в выборе и иного подхода к оптимизации. В C++ оптимизация – это задача, делегированная компилятору, который вправе генерировать любой код, основываясь на результирующем поведении программы, заданном кодом.

Предопределённые значения в параметрах

В C++ функции могут иметь необязательные параметры, для которых не требуются аргументы при вызове. В этом случае, например, функция с тремя параметрами может быть вызвана, имея только два. Для этого функция включит предопределённое значение для этого последнего параметра, который используется функцией при вызове с несколькими аргументами. Например:

```
// default values in functions
#include <iostream>
using namespace std;

int divide (int a, int b=2)
{
  int r;
  r=a/b;
  return (r);
}

int main ()
{
  cout << divide (12) << '\n';
  cout << divide (20,4) << '\n';
  return 0;
}
```

```
6
5
```

В этом примере два вызова функции `divide`. При первом вызове:

```
divide (12)
```

вызов передаёт только один аргумент в функцию, хотя функция имеет два параметра. Для этого

случая функция присваивает второму параметру значение 2 (обратите внимание на определение функции, которое объявляет второй параметр как `int b=2`). Таким образом, результат становится равен 6.

При втором вызове:

```
divide (20,4)
```

вызов передаёт два аргумента в функцию. Таким образом, предопределённое значение для `b` (`int b=2`) игнорируется, а `b` принимает значение, передаваемое через аргумент, это 4, что приводит к результату 5.

Объявление функций

В C++ идентификатор может использоваться в выражениях только после того, как он был объявлен. Например, некоторая переменная `x` не может использоваться до её объявления в выражении, таком как:

```
int x;
```

Это же относится и к функциям. Функции не могут вызываться до их объявления. По этой причине во всех предыдущих примерах функций они всегда определяются до функции `main`, которая является функцией, из которой вызываются все остальные функции. Если функция `main` была определена до других функций, это нарушает правило, что функции должны объявляться до их использования, вследствие чего они не будут компилироваться.

Прототип функции может объявляться без полного определения функции, дав достаточно деталей, чтобы позволить при вызове функции опознать все типы переменных в функции. Естественно, функция будет определена где-то ещё, похоже, позже в коде. Но, по крайней мере, объявленная так функция может уже быть вызвана.

Объявление будет включать все используемые типы (возвращаемый тип и типы аргументов), используя тот же синтаксис, что использован в определении функции, но с замещением тела функции (блок выражений) завершающей точкой с запятой.

Список параметров не нуждается во включении имён, но только их типов. Имена параметров могут всё-таки быть заданы, но они не обязательны, и нет необходимости соответствия их в определении функции. Например, функция, названная `protofunction` с двумя параметрами int, может быть объявлена любым из этих выражений:

```
1 int protofunction (int first, int second);
2 int protofunction (int, int);
```

Как бы то ни было, включение имени для каждого параметра всегда улучшает читабельность объявления.

```
1  // declaring functions prototypes
2  #include <iostream>
3  using namespace std;
4
5  void odd (int x);
6  void even (int x);
7
8  int main()
9  {
10   int i;
11   do {
```

```
Please, enter number (0 to exit): 9
It is odd.
Please, enter number (0 to exit): 6
It is even.
Please, enter number (0 to exit):
1030
It is even.
Please, enter number (0 to exit): 0
It is even.
```

```
12      cout << "Please, enter number (0
13 to exit): ";
14      cin >> i;
15      odd (i);
16    } while (i!=0);
17    return 0;
18 }
19
20 void odd (int x)
21 {
22    if ((x%2)!=0) cout << "It is
23 odd.\n";
24    else even (x);
25 }
26
27 void even (int x)
28 {
29    if ((x%2)==0) cout << "It is
   even.\n";
   else odd (x);
   }
```

Этот пример, конечно, не является примером эффективности. Вы можете, возможно, написать свою версию этой программы, имеющую вдвое меньше строк. В любом случае, этот пример иллюстрирует, как функции могут быть объявлены до их определения:

Следующие строки:

```
1 void odd (int a);
2 void even (int a);
```

декларируют прототипы функций. Они уже содержат всё, что необходимо для их вызова: их имена, типы их аргументов, и их возвращаемые типы (`void` в нашем случае). С размещением объявлений этих прототипов функции могут вызываться (в `main`) до того, как будут полностью определены.

Но объявление функций до их определения не только полезно при реорганизации порядка функций в коде. В некоторых случаях, как в этом частном примере, требуется хотя бы одна из деклараций, поскольку `odd` и `even` вызваются совместно; есть вызов `even` в `odd` и вызов `odd` в `even`. И, таким образом, нет способа структурировать код, поскольку функция `odd` определена до определения `even`, а `even` перед `odd`.

Рекурсивность

Рекурсивность – это свойство, позволяющее функции вызвать саму себя. Это полезно для некоторых задач, таких как сортировка элементов или вычисления факториала чисел. Например, в плане получения факториала числа (`n!`) математической формулой будет:

`n! = n * (n-1) * (n-2) * (n-3) ... * 1`
Более конкретно, `5!` (факториал числа 5) будет:

`5! = 5 * 4 * 3 * 2 * 1 = 120`
А рекурсивная функция для вычисления этого в C++ будет:

```
1 // factorial calculator
2 #include <iostream>
3 using namespace std;
4
```

```
9! = 362880
```

```
5  long factorial (long a)
6  {
7    if (a > 1)
8     return (a * factorial (a-1));
9    else
10    return 1;
11 }
12
13 int main ()
14 {
15   long number = 9;
16   cout << number << "! = " <<
17 factorial (number);
18   return 0;
   }
```

Заметьте, как в функции factorial мы включаем вызов её самой, но только в том случае, если передаваемый аргумент был больше, чем 1. Так как в противном случае функция будет выполнять бесконечный цикл, в котором по достижении 0 она будет продолжать перемножение всех отрицательных чисел (возможно, провоцируя переполнение стека из-за того же в процессе работы).

Перегрузка и шаблоны

Перегруженные функции

В C++ две разные функции могут иметь одинаковые имена, если их параметры различаются либо тем, что они имеют разное количество параметров, либо тем, что их параметры имеют разный тип. Например:

```
1  // overloading functions
2  #include <iostream>
3  using namespace std;
4
5  int operate (int a, int b)
6  {
7    return (a*b);
8  }
9
10 double operate (double a, double b)
11 {
12   return (a/b);
13 }
14
15 int main ()
16 {
17   int x=5,y=2;
18   double n=5.0,m=2.0;
19   cout << operate (x,y) << '\n';
20   cout << operate (n,m) << '\n';
21   return 0;
22 }
```

```
10
2.5
```

В этом примере есть две функции, названные `operate`, но одна из них имеет два параметра типа `int`, тогда как другая имеет их типа `double`. Компилятор знает, к какой из них обращаться при вызове в каждом случае, проверяя тип передаваемых им аргументов. Если вызов с двумя аргументами типа `int`, компилятор вызывает одну функцию, которая имеет два параметра типа `int`, а если вызывается с двумя типа `double`, он вызывает ту, что с двумя параметрами типа `double`.

В этом примере обе функции различаются совершенно разным поведением, версия `int` перемножает её аргументы, а та, что `double` версия, делит их. Это, конечно, не самая хорошая идея. От двух функций с одинаковыми именами обычно ожидается хотя бы одинаковое поведение, но в нашем примере демонстрируется, что это совсем не обязательно. Две перегруженные функции (т.е., две функции с одинаковыми именами) имеют вполне различаемые определения; они для всех целей разные функции, но им не повезло иметь одинаковые имена.

Заметьте, что функции не могут быть определены как перегруженные только по типу возвращаемого ими значения. Хотя бы один из их параметров должен быть другого типа.

Шаблоны функции

Перегруженные функции могут иметь и одинаковое определение. Вот пример:

```
1  // overloaded functions
2  #include <iostream>
3  using namespace std;
4
5  int sum (int a, int b)
6  {
```

```
30
2.5
```

```
  return a+b;
}

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

int main ()
{
  cout << sum (10,20) << '\n';
  cout << sum (1.0,1.5) << '\n';
  return 0;
}
```

Здесь `sum` перегруженная функция с разными типами параметров, но с совершенно одинаковым телом.

Функция `sum` может быть перегружена множеством типов, и она должна быть чувствительна к ним всем, чтобы иметь одинаковое тело функции. Для случаев, таких как этот, C++ имеет возможность определения функций с базовым типом, известным как *шаблон функции* (*function templates)*. Определение шаблона функции следует тому же синтаксису, что и обычная функция, за исключением того, что она предваряется ключевым словом `template` и рядом параметров, заключённых в угловые скобки <>:

```
template <template-parameters> function-declaration
```

Параметры шаблона – это последовательность параметров, разделённых запятыми. Эти параметры могут быть базовыми типами шаблона, задаваемые либо `class`, либо `typename` ключевыми словами, сопровождаемыми идентификаторами. Идентификатор может быть затем использован в объявлении функции, как это делается в обычной функции. Например, базовая `sum` функция может быть определена как:

```
template <class SomeType>
SomeType sum (SomeType a, SomeType b)
{
  return a+b;
}
```

Безразлично, будет ли базовый тип иметь ключевое слово `class` или `typename` в списке агрументов шаблона (они 100% синонимы в объявлении шаблона).

В коде выше объявление `SomeType` (базовый тип в параметрах шаблона, заключенный в угловые скобки) позволяет использовать `SomeType` в любом месте в определении функции, равно как и любой другой тип. Это может быть использовано как тип параметра, как возвращаемый тип или при объявлении новой переменной этого типа. Во всех случаях он представляет базовый тип, который был определён на момент обработки шаблона.

Для иллюстрации темы шаблона можно применить шаблон к созданию функции, использующей обычные типы или переменные в качестве параметров этого шаблона. Это выполняется вызовом *шаблона функции* (*function template)* с тем же синтаксисом, что и при вызове обычной функции, но с заданием аргументов шаблона, заключённых в угловые скобки:

```
name <template-arguments> (function-arguments)
```

Например, шаблон функции `sum`, определённой выше, может быть вызван:

```
x = sum<int>(10,20);
```

Функция `sum<int>` – это только одна из возможных демонстраций шаблона функции `sum`. В этом

случае, используя `int` в качестве аргумента шаблона при вызове, компилятор автоматически представляет версию `sum`, где при каждой встрече с `SomeType` происходит замена на `int`, как если бы имело место определение:

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

Давайте рассмотрим конкретный пример:

```
// function template
#include <iostream>
using namespace std;

template <class T>
T sum (T a, T b)
{
  T result;
  result = a + b;
  return result;
}

int main () {
  int i=5, j=6, k;
  double f=2.0, g=0.5, h;
  k=sum<int>(i,j);
  h=sum<double>(f,g);
  cout << k << '\n';
  cout << h << '\n';
  return 0;
}
```

```
11
2.5
```

В данном случае мы использовали `T` в качестве имени параметра шаблона вместо `SomeType`. Это не играет роли, а `T` довольно распространённое имя параметра шаблона для базовых типов.

В примере выше мы видели, что шаблон функции `sum` использован дважды. В первый раз он использован с типом `int`, а второй раз с аргументом типа `double`. Компилятор это обработал, а затем вызвал в обоих случаях подходящую версию функции.

Заметьте также, как `T` использовалось для объявления локальной переменной этого (базового) типа в `sum`:

```
T result;
```

Таким образом, *result* будет переменной того же типа, что и параметры `a` и `b`, равно как и тип, возвращаемый функцией.

В этом специфическом случае, где базовый тип `T` используется как параметр для `sum`, компилятор способен отследить тип данных даже автоматически без получения их явного задания в угловых скобок. Поэтому, вместо явного указания аргументов шаблона:

```
k = sum<int> (i,j);
h = sum<double> (f,g);
```

есть возможность просто написать:

```
k = sum (i,j);
h = sum (f,g);
```

без типа, заключённого в угловые скобки. Конечно, для этого тип должен быть точно выражен. Если `sum` вызывается с аргументами разных типов, компилятор не сможет автоматически определить тип `T`.

Шаблоны – это мощное и универсальное средство. Можно иметь множество параметров шаблона, а функция может использовать помимо того и обычные, не указанные шаблоном типы. Например:

```
// function templates
#include <iostream>
using namespace std;

template <class T, class U>
bool are_equal (T a, U b)
{
  return (a==b);
}

int main ()
{
  if (are_equal(10,10.0))
    cout << "x and y are equal\n";
  else
    cout << "x and y are not
equal\n";
  return 0;
}
```

```
x and y are equal
```

Заметьте, что в этом примере используется автоматчиское определение параметров шаблона при вызове `are_equal`:

```
are_equal (10,10.0)
```

что эквивалентно:

```
are_equal<int,double>(10,10.0)
```

Поскольку в C++ целые литералы без суффикса (подобно `10`) всегда типа `int`, а литералы с плавающей точкой без суффикса (как `10.0`) всегда типа `double`, здесь нет места сомнениям, и, следовательно, аргументы шаблона могут быть пропущены при вызове.

Нестандартные аргументы шаблона

Параметры шаблона включают не только типы `class` или `typename`, но могут также включать в выражения обычные типы:

```
// template arguments
#include <iostream>
using namespace std;

template <class T, int N>
T fixed_multiply (T val)
```

```
20
30
```

```
{
  return val * N;
}

int main() {
  std::cout <<
fixed_multiply<int,2>(10) << '\n';
  std::cout <<
fixed_multiply<int,3>(10) << '\n';
}
```

Второй аргумент шаблона функции `fixed_multiply` имеет тип `int`. И он выглядит как параметр функции обычного типа, и может действительно использоваться обычным образом.

Но здесь есть и главное отличие: значения параметров шаблона определяются во время компиляции, чтобы генерировать разные экземпляры функции `fixed_multiply`. Таким образом, значение этого аргумента не передаётся при работе: два вызова `fixed_multiply` в `main` по существу вызывают две версии функции, первая всегда умножает на два, а другая всегда умножает на три. Из этих соображений второй аргумент шаблона должен быть константой (он не может передать переменную).

Видимость имён

Границы

Именованные сущности, такие как переменные, функции и сложные типы, должны быть объявлены до их использования в C++. Точка в программе, где эти объявления имеют место, влияет на видимость этих объявленных объектов:

Сущности, объявленные вне любого блока, имеют *глобальные границы* (*global scope)*, означающие, что их имена работают в любом месте кода. Когда же объекты объявляются внутри блока, такого как функция или отдельное выражение, они имеют *границы блока* (*block scope)*, и имеют видимость внутри отдельного блока, в котором они объявлены, но не вне его.

Переменные с границами блока известны как *локальные переменные* (*local variables)*.

Например, переменная, объявленная в теле функции – это *локальная переменная*, которая действенна до конца функции (т.е., до скобки `}`, которая закрывает определение функции), но не вне её:

```
int foo;         // глобальная переменная

int some_function ()
{
  int bar;       // локальная переменая
  bar = 0;
}

int other_function ()
{
  foo = 1;  // ok: foo является глобальной переменной
  bar = 2;  // неверно: bar является локальной в её функции
}
```

В каждых границах имя может представлять только один объект. Например, не может быть двух переменных с одинаковыми именами в одинаковых границах:

```
int some_function ()
{
  int x;
  x = 0;
  double x;   // неверно, имя уже использовано в этих границах
  x = 0.0;
}
```

Видимость объекта в *границах блока* работает до конца блока, включая внутренние блоки. Тем не менее, внутренний блок, поскольку это другой блок, может повторять имя, содержащееся во внешних границах, адресуясь к разным объектам. В этом случае имя будет относиться к разным объектам только во внутреннем блоке, скрывая объект, именованный вне блока. Будучи вне блока, он будет адресоваться к оригинальному объекту. Например:

```
// inner block scopes
#include <iostream>
using namespace std;

int main () {
  int x = 10;
  int y = 20;
  {
```

```
Внутренний блок:
x: 50
y: 50
внешний блок:
x: 10
y: 50
```

```
    int x;  // ok, внутренние гранцы
    x = 50; // значение внутреннему x
    y = 50; // значение внешнему y
    cout << "внутренний блок:\n";
    cout << "x: " << x << '\n';
    cout << "y: " << y << '\n';
  }
  cout << "внешний блок:\n";
  cout << "x: " << x << '\n';
  cout << "y: " << y << '\n';
  return 0;
}
```

Заметьте, что `y` не скрыто во внутреннем блоке, а, следовательно, присвоение значения `y` всё ещё доступно для внешней переменной.

Переменные, объявленные в объявлениях, представляющих блок, таких как параметры функции и переменные, объявленнные в циклах и условиях (таких, как объявленные для for или if) являются локальными для блока, их представляющего.

Пространство имён

Только один объект может существовать с индивидуальным именем в индивидуальных границах. Временами это становится проблемой для локальных имён, поскольку блоки имеют тенденцию быть относительно короткими, а имена в них имеют специфические свойства, такие как именование переменных счётчика, аргументы и т.п.

Но нелокальные имена предвносят больше возможностей для коллизии имён, в частности с учётом того, что библиотеки могут объявлять много функций, типов, переменных, которые по своей природе не будут локальными, а некоторые и базовыми.

Пространство имён позволяет нам группировать именованные объекты, которые иначе будут иметь *глобальные границы* (*global scope)*, в узких границах, давая им *границы пространства имён* (*namespace scope)*. Это позволяет организовать элементы программ в разные логические границы, обозначенные именами.

Синтаксис для объявления пространства имён:

```
namespace identifier
{
  named_entities
}
```

Здесь `identifier` является каким-то правильным идентификатором, а `named_entities` является набором переменных, типов и функций, которые включены в пространство имён. Например:

```
namespace myNamespace
{
  int a, b;
}
```

В этом случае переменные `a` и `b` – это обычные переменные, объявленные внутри пространства имён, называнного `myNamespace`.

Эти переменные могут быть доступны в их пространстве имён обычным образом по их

идентификатору (либо a, либо b), но если доступ осуществлён извне пространства имён myNamespace, они должны быть соответствующим образом квалифицированы с помощью оператора границ ::. Например, для доступа к предыдущим переменным за пределами myNamespace оин должны квалифицироваться:

```
myNamespace::a
myNamespace::b
```

Пространство имён обычно полезно для того, чтобы избежать коллизий имён. Например:

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

namespace foo
{
  int value() { return 5; }
}

namespace bar
{
  const double pi = 3.1416;
  double value() { return 2*pi; }
}

int main () {
  cout << foo::value() << '\n';
  cout << bar::value() << '\n';
  cout << bar::pi << '\n';
  return 0;
}
```

```
5
6.2832
3.1416
```

В данном случае есть две функции с одинаковыми именами: value. Одна определена в пространстве имён foo, и другая в bar. Но не будет ошибок переопределения, благодаря пространствам имён. Заметьте, как pi появляется неквалифицированным образом в пространстве имён bar (только как pi), и вновь появляется в main, но здесь нуждается в квалификации как bar::pi.

Пространства имён могут разветвляться: два сегмента кода могут быть объявлены в одном пространстве имён:

```
namespace foo { int a; }
namespace bar { int b; }
namespace foo { int c; }
```

Здесь объявляются три переменные: a и c в пространстве имён foo, тогда как b находится в пространстве имён bar. Пространство имён может простираться даже на разные единицы трансляции (т.е., на разные файлы исходного кода).

using

Ключевое слово using вводит имя в текущую область объявлений (такую, как блок), избегая необходимости квалифицировать имя. Например:

```
// using
#include <iostream>
```

```
5
2.7183
```

```
using namespace std;

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using first::x;
  using second::y;
  cout << x << '\n';
  cout << y << '\n';
  cout << first::y << '\n';
  cout << second::x << '\n';
  return 0;
}
```

```
10
3.1416
```

Заметьте, как в `main` переменая `x` (без какого-либо квалификатора имени) адресуется к `first::x`, тогда как `y` ссылается на `second::y`, равно как и задано объявлением `using`. Переменные `first::y` и `second::x` могут быть доступны, но требуют полной квалификации имён.

Ключевое слово `using` может также использоваться в качестве директивы для ввода всего пространства имён:

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

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using namespace first;
  cout << x << '\n';
  cout << y << '\n';
  cout << second::x << '\n';
  cout << second::y << '\n';
  return 0;
}
```

```
5
10
3.1416
2.7183
```

В данном случае, объявляя, что мы используем пространство имён `first`, всё прямое

использование `x` и `y` без квалификации имён будет рассматривать их в пространстве имён `first`.

`using` и `using namespace` имеют законность только в тех блоках, в которых они объявляются, или во всём файле исходного кода, если они используются непосредственно в глобальных границах. Например, будет возможно первое использование объектов одного пространства имён, а затем другое разделение кода в разных блоках:

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

namespace first
{
  int x = 5;
}

namespace second
{
  double x = 3.1416;
}

int main () {
  {
    using namespace first;
    cout << x << '\n';
  }
  {
    using namespace second;
    cout << x << '\n';
  }
  return 0;
}
```

```
5
3.1416
```

Псевдонимы пространства имён

Существующее пространство имён может быть замещено новым именем с помощью следующего синтаксиса:

```
namespace new_name = current_name;
```

Пространство имён std

Все объекты (переменные, типы, постоянные и функции) библиотеки стандарта C++ объявляются в пространстве имён `std`. Большая часть примеров в этом руководстве фактически включают следующую строку:

```
using namespace std;
```

Это вводит прямую доступность всех имен пространства `std` в коде. В руководстве это сделано для облегчения понимания и уменьшения длины примеров, но многие программы предпочитают квалифицировать каждый из элеменов стандартной библиотеки, используемых в их программах. Например, вместо:

```
cout << "Hello world!";
```

часто можно видеть:

```
std::cout << "Hello world!";
```

Будут ли элементы пространства имён `std` введены с помощью объявления `using` или полной квалификацией при каждом использовании, никоим образом не изменит поведение или эффективность результирующей программы. Это в большей мере вопрос стиля, хотя для проектов со смешанными библиотеками предпочтительнее использовать явную квалификацию.

Классы памяти

Хранилище переменных с помощью *global* или *namespace scope* (глобальное или в границах пространства имён) существует на протяжении всей работы программы. Это известно как *static storage (статическое хранилище)*, и это контрастирует с хранением *local variables (локальных переменных)*, которые объявляются в блоке. Подобный механизм известен как автоматическое хранение. Хранилище для локальных переменных доступно в блоке, в котором они объявлены. После чего эти хранилища могут использоваться для локальных переменных какой-то другой функции или использоватья как-то иначе.

Но есть другая существенная разница для переменных между *static storage (статическим хранением)* и *automatic storage (автоматическим хранением)*:
- Переменные с *static storage* (такие, как глобальные переменные), которые не инициализированы явно, инициализируются автоматически нулём.
- Переменные с *automatic storage* (такие, как локальные переменные), которые не инициализированы явно, остаются не инициализированы, то есть, имеют неопределённое значение.

Например:

```
1  // static vs automatic storage
2  #include <iostream>
3  using namespace std;
4
5  int x;
6
7  int main ()
8  {
9    int y;
10   cout << x << '\n';
11   cout << y << '\n';
12   return 0;
13 }
```

```
0
4285838
```

Реальный вывод может варьироваться, но только значение `x` будет гарантировано равно нулю. `y` может реально содержать любое значение (включая нуль).

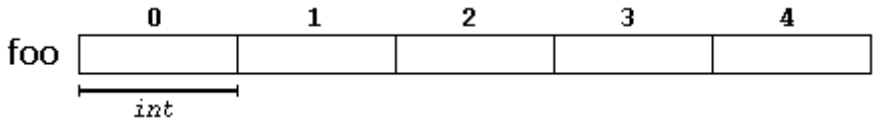
Сложные типы данных

Массивы

Массив – это ряд элементов одного и того же типа, размещённый в последовательных ячейках памяти, которые могут быть индивидуально достижимы через добавление индекса к уникальному идентификатору.

Это означает, например, что пять значений типа `int` могут объявляться в качестве массива без необходимости объявлять пять разных переменных (каждое со своим собственным идентификатором). Вместо этого, используя массив, пять значений `int` хранятся в последовательных ячейках памяти, и все пять могут быть достигнуты при использовании того же идентификатора с подходящим индексом.

Например, массив, содержащий пять целых значений типа `int`, названный `foo`, может быть представлен как:



где каждая пустая клетка представляет элемент массива. В данном случае это значения типа `int`.

Эти элементы обозначены числами от 0 до 4, где 0 первое, а 4 последнее. В C++ первый элемент в массиве всегда нумеруется нулём (а не единицей) вне зависимости от длины массива.

Подобно обычной переменой массив должен быть объявлен до его использования. Типичное объявление массива в C++ выглядит так:

```
type name [elements];
```

где `type` – подходящий тип (как `int`, `float`...), `name` – подходящий идентификатор, а `elements` – это поле (которое всегда в квадратных скобках `[]`), задающее длину массива в терминах количества элементов.

Таким образом, массив `foo` с пятью элементами типа `int` может быть объявлен как:

```
int foo [5];
```

Заметьте: поле `elements` в квадратных скобках `[]`, представляющее количество элементов массива, должно быть *constant expression (постоянное выражение)*, поскольку массивы являются блоками статической памяти, чей размер должен быть определён во время компиляции до выполнения программы.

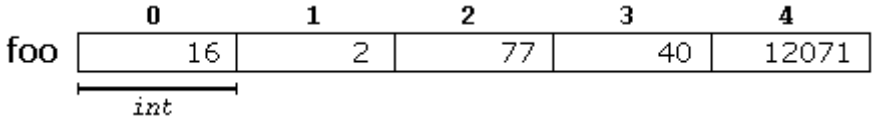
Инициализация массивов

По умолчанию обычные массивы *локальной области*, *local scope* (например, объявленные внутри функции), остаются без инициалилзации. Это означает, что ни одному из элементов не было присвоено обычное значение; их содержимое не определено в момент объявления массива.

Но элементам массива могут быть присвоены начальные значения при объявлении, если заключить эти значения инициализации в фигурные скобки {}. Например:

```
int foo [5] = { 16, 2, 77, 40, 12071 };
```

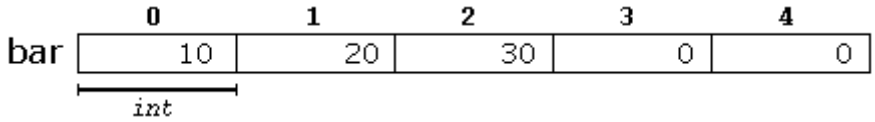
Это выражение объявляет массив, который может быть представлен следующим образом:



Количество значений в скобках `{}` не должно быть больше, чем количество элементов в массиве. Например, в примере выше массив `foo` был объявлен с 5 элементами (как задано числом, заключённым в квадратные скобки `[]`), а фигурные скобки `{}` содержать ровно 5 значений, по одному для каждого элемента массива. Если заявлено меньше значений, оставшимся элементам присваиваются их значения по умолчанию (что для основных типов означает, что элементы заполняются нулями). Например:

```
int bar [5] = { 10, 20, 30 };
```

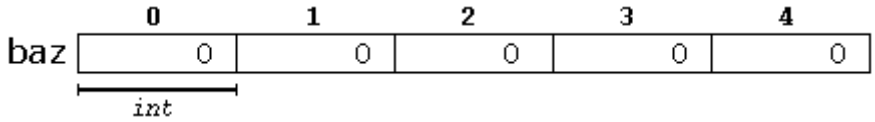
Это создаст массив подобный этому:



Инициализация может даже не иметь значений, а только скобки:

```
int baz [5] = { };
```

Этим создаётся массив пяти `int` значений, где каждому присваивается значение нуля:



Когда значения инициализации предоставляются массиву, C++ позволяет оставить квадратные скобки пустыми `[]`. В этом случае компилятор автоматически определит размер массива, совпадающий с числом значений, заключённых между фигурными скобками `{}`:

```
int foo [] = { 16, 2, 77, 40, 12071 };
```

После этого объявления массив `foo` будет 5 `int` длиной, поскольку мы предложили 5 начальных значений.

Наконец, эволюция C++ привела к выбору *универсальной инициализации* и массивов. Поэтому больше нет необходимости для значка равенства между объявлением и инициализатором. Оба выражения ниже эквивалентны:

```
1 int foo[] = { 10, 20, 30 };
2 int foo[] { 10, 20, 30 };
```

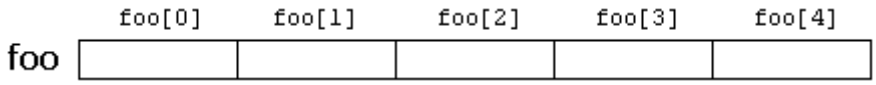
Статические массивы и те, что объявляются прямо в пространстве имён (вне каких-либо функций), всегда инициализируются. Если явный инициализатор на задан, все элементы инициализируются по умолчанию (нулями для базовых типов).

Доступ к значениям в массиве

Значение любого элемента в массиве можно получить так же, как и значение обычной переменной того же типа. Синтаксис таков:

```
name[index]
```

Следуя предыдущему примеру, где `foo` имеет 5 элементов, а каждый из этих элементов типа `int`, имена, которые можно использовать для ссылки на каждый из элементов, следующие:



Например, следующее выражение отправляет на хранение значение 75 в третий элемент `foo`:

```
foo [2] = 75;
```

и, например, следующим копируется значение третьего элемента `foo` в переменную `x`:

```
x = foo[2];
```

Следовательно, выражение `foo[2]` само является переменной типа `int`.

Заметьте, что третий элемент `foo` задаётся как `foo[2]`, поскольку первый – это `foo[0]`, второй – `foo[1]`, а, значит, третий `foo[2]`. Из этих соображений последний элемент будет `foo[4]`. Таким образом, если мы напишем `foo[5]`, мы попытаемся добраться до шестого элемента `foo`, что в действительности превышает размер массива.

В C++ синтаксически правильно превышать допустимый предел диапазона индексов массива. Это может создавать проблемы, поскольку доступ к элементам вне диапазона не вызывает ошибки при компиляции, но вызовет ошибку при работе программы. Причина такого положения дел будет показан в следующей главе, представляющей указатели.

В этом месте важно ясно различать два использования квадратных скобок `[]` относительно массивов. Они выполняют две разные задачи: одна – задать размер массива при его декларации; а вторая задать индекс для конкретного элемента массива, при доступе к нему. Не путайте эти два использования скобок `[]` с массивом.

```
1 int foo[5];         // объявление нового массива
2 foo[2] = 75;        // доступ к элементу массива
```

Основная разница в том, что объявлению предшествует тип элементов, тогда как при доступе этого нет.

Некоторые другие операции с массивами:

```
1 foo[0] = a;
2 foo[a] = 75;
3 b = foo [a+2];
4 foo[foo[a]] = foo[2] + 5;
```

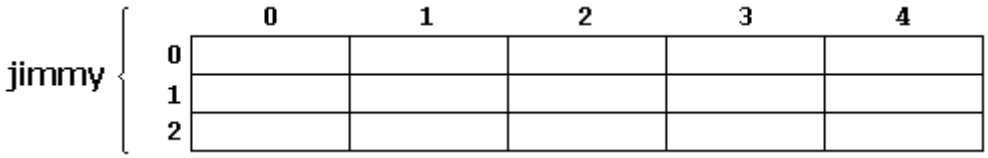
Например:

```
// arrays example
#include <iostream>
using namespace std;

int foo [] = {16, 2, 77, 40, 12071};
int n, result=0;

int main ()
{
  for ( n=0 ; n<5 ; ++n )
  {
    result += foo[n];
  }
  cout << result;
  return 0;
}
```

```
12206
```

Многомерные массивы

Многомерные массивы могут объявляться как «массивы массивов». Например, двумерный массив может быть представлен, как двумерная таблица, составленная из элементов, все из которых одного и того же типа данных.

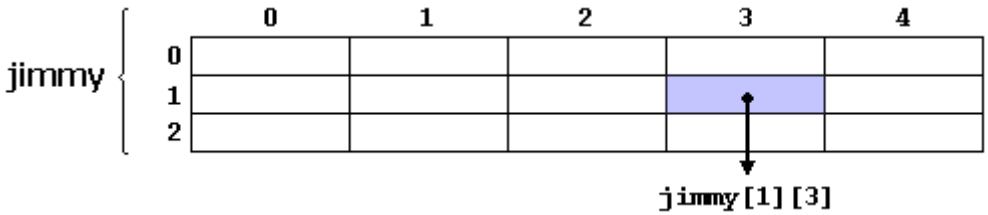


`jimmy` представляет двумерный массив из 3 элементов (0,1,2), каждый из которых имеет 5 элементов (0,1,2,3,4) типа `int`. Синтаксис C++ для этого:

```
int jimmy [3][5];
```

И, например, ссылка на второй элемент по вертикали и четвёртый по горизонтали в выражении будет:

```
jimmy[1][3]
```



(помните, что индексы массива всегда начинаются с нуля).

Многомерные массивы не ограничиваются двумя индексами (т.е., двумя измерениями). Они могут иметь столько индексов, сколько необходимо. Хотя будьте внимательны: количество памяти, требуемое массивом, увеличивается экспоненциально с каждым измерением. Например:

```
char century [100][365][24][60][60];
```

объявляет массив с элементами типа `char` для каждого в сотне. Это более, чем 3 миллиарда `char`! Так что, это объявление будет занимать более 3 гигабайт памяти!

В завершение, многомерные массивы – это только абстракция для программистов, поскольку такие же результаты могут быть получены с помощью простых массивов, если умножать их индексы:

```
1 int jimmy [3][5];   // эквивалентно следующему
2 int jimmy [15];     // (3 * 5 = 15)
```

Есть одно в этом отличие от многомерных массивов – компилятор автоматически запоминает глубину каждого из воображаемых измерений. Следующие два фрагмента кода дают одинаковые результаты, но один использует двумерный массив, а другой использует простой массив:

многомерный массив

```
#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT][WIDTH];
int n,m;

int main ()
{
  for (n=0; n<HEIGHT; n++)
    for (m=0; m<WIDTH; m++)
    {
      jimmy[n][m]=(n+1)*(m+1);
    }
}
```

псевдо-многомерный массив

```
#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT * WIDTH];
int n,m;

int main ()
{
  for (n=0; n<HEIGHT; n++)
    for (m=0; m<WIDTH; m++)
    {
      jimmy[n*WIDTH+m]=(n+1)*(m+1);
    }
}
```

Ни один из двух фрагментов кода выше не выводит что-то на экран, но оба присваивают значения блоку памяти, названному jimmy следующим образом:

jimmy

	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

Заметьте, что код использует константы для ширины и высоты вместо использования их непосредственного числового значения. Это делает код лучше читаемым и позволяет изменить код легче, исправив только в одном месте.

Массивы как параметры

Где-то нам может понадобиться передать массив в функцию как параметр. В C++ нет возможности передавать целый блок памяти, представляющий массив, в функцию непосредственно как аргумент. Но то, что вместо этого можно передать, это адрес. На практике это даёт тот же эффект, притом передача осуществляется много быстрее, и эта операция много результативнее.

При применении массива в качестве параметра для функции тип параметра должен быть объявлен как массив, но с пустыми квадратными скобками, где пропущен реальный размер массива. Например:

```
void procedure (int arg[])
```

Эта функция имеет параметр типа "array of `int`" (массив данных целого типа), названный `arg`. В порядке передачи в эту функцию массив объявлен как:

```
int myarray [40];
```

и при вызове будет достаточно написать:

```
procedure (myarray);
```

Вот полный пример:

```
// arrays as parameters
#include <iostream>
using namespace std;

void printarray (int arg[], int
length) {
  for (int n=0; n<length; ++n)
    cout << arg[n] << ' ';
  cout << '\n';
}

int main ()
{
  int firstarray[] = {5, 10, 15};
  int secondarray[] = {2, 4, 6, 8,
10};
  printarray (firstarray,3);
  printarray (secondarray,5);
}
```

```
5 10 15
2 4 6 8 10
```

В коде выше первый параметр (`int arg[]`) принимает любой массив, у которого элементы типа `int` вне зависимости от его длины. Из этих соображений мы включили второй параметр, который поясняет функции длину каждого массива, которой мы передаём ей как первый параметр. Это позволяет циклу, который выводит на экран массив, понять количество итераций для переданного массива, без выхода за пределы диапазона.

В объявлении функции также можно включать многомерные массивы. Формат для параметра трёхмерного массива будет:

```
base_type[][depth][depth]
```

Например, функция с многомерным массивом в качестве аргумента будет:

```
void procedure (int myarray[][3][4])
```

Обратите внимание, что первые скобки `[]` остаются пустыми, тогда как следующие задают размеры их соответствующих измерений. Это необходимо компилятору, чтобы определить глубину каждого из дополнительных измерений.

В какой-то мере передача массива в качестве аргумента всегда ведёт к потере размерности. Причина в том, что есть исторические мотивы – массивы не могут копироваться непосредственно, поэтому в реальности передаются через указатели. Это общий источник ошибок для начинающих программистов. Хотя ясное понимание указателей, описанных далее, должно помочь в этом.

Библиотечные массивы

Массивы, обсуждённые выше, непосредственно встроены в язык и унаследованы от Си. Они отличные функции, но при ограничении на копирование и легком разведении на указатели, они, возможно, чрезмерно страдают от оптимизации.

Чтобы преодолеть некоторые из этих результатов со встроенными в язык массивами, C++ предлагает альтернативный тип массива, такой как стандартный контейнер. Это тип шаблона (класс шаблона, фактически), определённый в заголовочном файле `<array>`.

Контейнеры – это библиотека функций, которые выходят за рамки этого руководства, и по этой причине этот класс не рассматривается здесь в деталях. Достаточно сказать, что они оперируют схожим образом со встроенными массивами, за исключением того, что могут копироваться (в действительности довольно расточительная операция копирования целого блока памяти, так что следует её использовать осторожно) и переходить к указателям только тогда, когда их явно к этому понуждают (посредством их члена `data`).

Только в качестве примера – вот две версии одинакового образца использования встроенного в язык массива, описанного в этой главе, и контейнера в библиотеке:

массив, встроенный в язык

```
#include <iostream>

using namespace std;

int main()
{
  int myarray[3] = {10,20,30};

  for (int i=0; i<3; ++i)
    ++myarray[i];

  for (int elem : myarray)
    cout << elem << '\n';
}
```

массив как библиотечный контейнер

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

int main()
{
  array<int,3> myarray {10,20,30};

  for (int i=0; i<myarray.size(); ++i)
    ++myarray[i];

  for (int elem : myarray)
    cout << elem << '\n';
}
```

Как вы видите, обе разновидности массивов используют одинаковый синтаксис для доступа к их элементам: `myarray[i]`. За исключением этого основная разница лежит в объявлении массива и включении дополнительного заголовочного файла для *библиотеки массивов*.

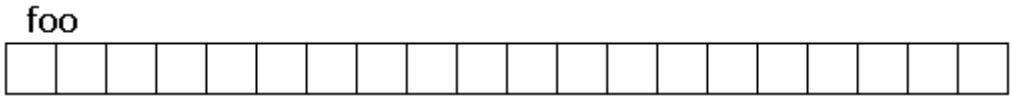
Последовательности символов

Класс `string` был затронут в предыдущих разделах. Это достаточно мощный класс для поддержки и операций с последовательностями символов. Однако, поскольку строки – это фактически последовательности символов, мы можем представить их также как простые массивы элементов типа `char`.

Например, следующий массив:

```
char foo [20];
```

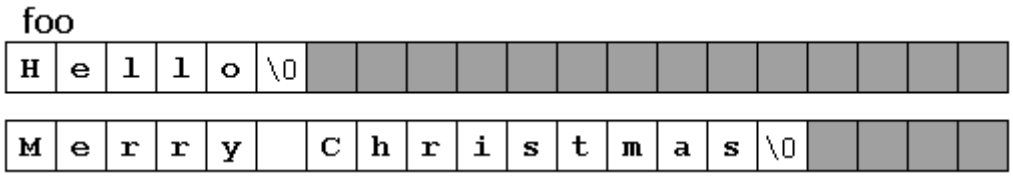
это массив, который может хранить до 20 элементов типа `char`. Его можно представить так:



Таким образом, этот массив имеет ёмкость для хранения последовательности до 20 символов. Но эта ёмкость не обязательно должна полностью использоваться: массив может размещать более короткую последовательность. Например, в каком-то месте программы либо последовательность `"Hello"`, либо последовательность `"Merry Christmas"` может храниться в `foo`, поскольку обе поместятся в последовательности ёмкостью в 20 символов.

Обычно конец строки, представленный в последовательности символов, задаётся специальным символом: *нулевым символом*, литеральное значение которого может записываться как `'\0'` (обратная косая черта, нуль).

В этом случае массив из 20 элементов типа `char`, названный `foo`, может быть представлен при хранении последовательности символов `"Hello"` и `"Merry Christmas"` как:



Заметьте, как после собственно содержимого строк был добавлен нулевой символ (`'\0'`), чтобы обозначить конец последовательности. Ячейки серого цвета представляют элементы `char` с неопределённым значением.

Инициализация последовательности символов с нуль-символом

Поскольку массивы символов – это обычные массивы, они следуют тем же правилам. Например, при инициализации массива символов предопределённой последовательностью мы можем сделать это так же, как для любого другого массива:

```
char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

Выше объявлен массив из 6 элементов типа `char` с начальными символами, формирующими слово `"Hello"` плюс *нулевой символ* `'\0'` в конце.

Но массив символов может быть инициализирован иначе: используя *строку литералов* непосредственно.

В выражениях, использованных в некоторых примерах предыдущих глав, литералы строк уже показаны несколько раз. Это задаётся заключением текста в двойные кавычки ("). Например:

```
"the result is: "
```

Это *строковый литерал*, возможно, использованный в некоторых предыдущих примерах.

Последовательность символов в двойных кавычках (") – это *литеральные константы*. И их тип фактически массив символов, завершающихся нуль-символом. Это означает, что строка литералов всегда имеет символ *null* (`'\0'`), автоматически добавляемый в конце.

Таким образом, массив элементов char, названный `myword`, может инициализироваться последовательностью символов с нуль-символом одним из двух способов:

```
char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
char myword[] = "Hello";
```

В обоих случаях массив символов `myword` объявляется с размером в 6 элементов типа `char`: 5 символов, составляющие слово "`Hello`", плюс завершающий *null* символ (`'\0'`), который задаёт конец последовательности, а во втором случае, когда используются двойные кавычки ("), нуль-символ добавляется автоматически.

Заметьте, что здесь мы обсуждали инициализацию массива символов в момент его объявления, но не обсуждали присвоения значений ему позже (когда он уже был декларирован). Фактически, поскольку строковые литералы относятся к обычным массивам, они имеют те же ограничения и им не могут присваиваться значения.

Выражения (когда myword уже был объявлен, как выше), такие как:

```
myword = "Bye";
myword[] = "Bye";
```

не будут правильны, тогда как следующее будет:

```
myword = { 'B', 'y', 'e', '\0' };
```

Это происходит потому, что массивам не могут присваиваться значения. Но заметьте, что каждому элементу массива можно присваивать значение индивидуально. Например, следующее будет корректно:

```
myword[0] = 'B';
myword[1] = 'y';
myword[2] = 'e';
myword[3] = '\0';
```

Строки и последовательности символов с null-завершением

Простой массив с нуль-символом в последовательности символов – это обычный тип, используемый в языке Си для представления строк (это то, почему они известны как *C-strings, Си-строки*). В C++, хотя стандартная библиотека определяет специфический тип для строк (класс `string`), обычные массивы с нулевым символом в последовательности символов (C-strings) всё ещё остаются обычным способом представления строк в языке. Фактически строковые литералы всё ещё всегда производят последовательности с нуль-символами, а не `string` объекты.

В страндартной библиотеке оба представления для строк (C-strings и библиотечные строки) сосуществуют, и большинство функций, требующих строк, будут перегруженными для поддержки

обоих.

Например, `cin` и `cout` поддерживают null-завершение последовательности непосредственно, позволяя им напрямую извлекаться из `cin` или вставляться в `cout`, просто как строки. Например:

```
// strings and NTCS:
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  char question1[] = "What is your
name? ";
  string question2 = "Where do you
live? ";
  char answer1 [80];
  string answer2;
  cout << question1;
  cin >> answer1;
  cout << question2;
  cin >> answer2;
  cout << "Hello, " << answer1;
  cout << " from " << answer2 <<
"!\n";
  return 0;
}
```

```
What is your name? Homer
Where do you live? Greece
Hello, Homer from Greece!
```

В этом примере оба массива символов используют последовательность с нулевым символом, и используются строки. Они полностью взаимозаменяемы в их совместном использовании в `cin` и `cout`, но здесь заметна ощутимая разница при их объявлении: массивы имеют фиксированный размер, что нужно при задании, явном или неявном, и при декларировании. `question1` имеет размер строго 20 символов (включая завершающий нуль-символ), а `answer1` имеет размер 80 символов, тогда как при объявлении строк размер не задан. Это происходит благодаря тому факту, что строки имеют динамическое определение размера при работе программы, а размер массивов определяется при компиляции до начала работы программы.

В любом случае последовательности с нулевым символом и строки легко трансформируются из одного представления в другое.

Последовательность с Null-завершающим символом может быть преобразована в строку неявно, а строка может преобразовываться в последовательность с нуль-символом при использовании либо `string` функции-члена `c_str`, либо `data`:

```
char myntcs[] = "some text";
string mystring = myntcs;  // конвертирует c-string в string
cout << mystring;          // выводит как библиотечную строку
cout << mystring.c_str();  // выводит как c-строку
```

(заметьте: и `c_str`, и `data` члены `string` эквивалентны)

Указатели

В предыдущих главах переменные объяснялись как области компьютерной памяти, которые могут быть доступны по их идентификатору (их имени). Таким способом программа не нуждается в заботе о физическом адресе данных в памяти, она просто использует идентификатор, где бы ей ни понадобилось сослаться на переменную.

Для C++ программ память компьютера подобна последовательности ячеек памяти, каждая размером по одному байту и каждая с уникальным адресом. Эти однобайтовые ячейки памяти упорядочены так, чтобы позволить данным быть представленными более чем одним байтом, занимая ячейки памяти, которые имеют соседние адреса.

Таким образом, каждая ячейка может быть легко локализована в памяти значением своего уникального адреса. Например, ячейка памяти с адресом `1776` всегда следует непосредственно за ячейкой с адресом `1775` и предшествует ячейке `1777`, и ровно тысяча ячеек памяти будет после `776`, и ровно тысяча ячеек памяти будет перед `2776`.

Когда объявляется переменная, память должна хранить присвоенное ей значение в заданном месте памяти (по адресу в памяти). Обычно C++ программы не активно определяют адреса памяти, где хранятся переменные. К счастью, эта задача остаётся за окружением, где выполняется программа – обычно это операционная система, которая решает, в частности, вопрос использования памяти в момент работы программы. Однако программе может быть полезен доступ к адресам переменных в процессе выполнения, для того, чтобы получить доступ к ячейкам данных, которые находятся в определенном положении по отношению к ним.

Оператор получения адреса (&)

Адрес переменной может быть получен предварением имени переменной знаком амперсанда (`&`), известного как *address-of operator (оператор получения адреса)*. Например:

```
foo = &myvar;
```

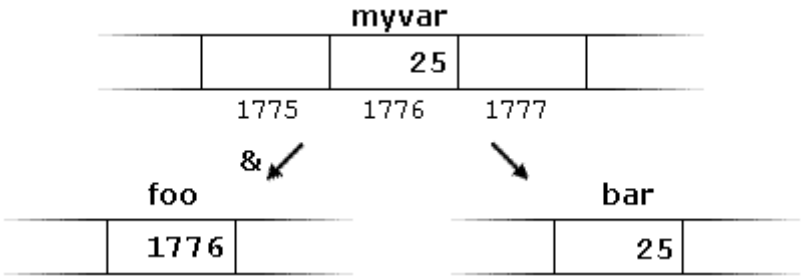
Этим адрес переменной `myvar` присваивается `foo`. Предваряя имя переменной `myvar` оператором *получения адреса* (`&`), мы больше не присваиваем содержание самой переменной `myvar` переменной `foo`, но только её адрес.

Реальный адрес переменной в памяти не может быть известен до выполнения программы, но давайте предположим в порядке помощи в уяснении некоторых концепций, что `myvar` при работе программы располагается в памяти по адресу `1776`.

Для этого случая разберём следующий фрагмент кода:

```
1 myvar = 25;
2 foo = &myvar;
3 bar = myvar;
```

Значения, содержащиеся в каждой из переменных после выполнения этого фрагмента, показано на следующей диаграмме:



Вначале мы присваиваем значение `25` переменной `myvar` (переменнной, чей адрес в памяти мы

предположили, что он будет 1776).

Второе выражение присваивает foo адрес myvar, который мы предположили как 1776.

Наконец, третье выражение присваивает значение, содержащееся в myvar переменной bar. Это стандартная операция присваивания, как мы это делали не раз в предыдущих главах.

Основная разница между вторым и третьим выражениями – это появление оператора *взятия адреса* (&).

Переменная, которая хранит адрес другой переменной (подобно foo в предыдущем примере), это то, что в C++ названо *pointer (указатель)*. Указатели – очень мощное средство языка, которое имеет множество применений в программировании низкого уровня. Чуть позже мы увидим, как объявлять и использовать указатели.

Оператор разыменовывания (*)

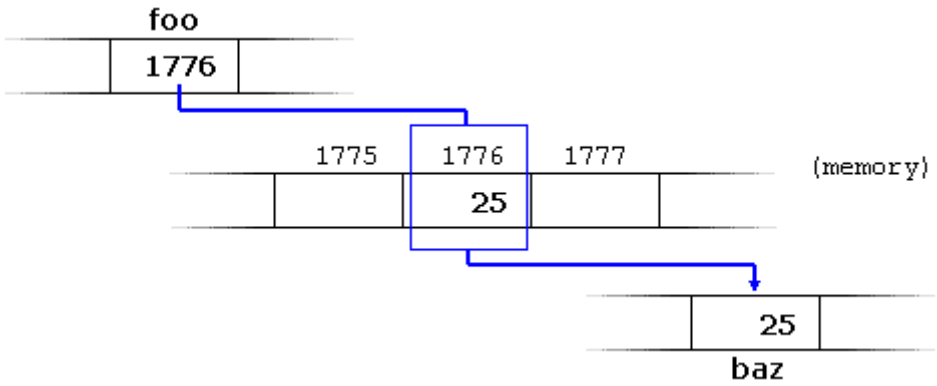
Как мы только что установили, переменная, хранящая адрес другой переменной, называется *указатель*, *pointer*. Указатели, как говорят, «указывают на» переменную, чей адрес они хранят.

Интересное свойство указателей в том, что они могут обращатья к переменной, на которую указывают, непосредственно. Это осуществляется предварением имени указателя *оператором разыменовывания* (*dereference operator)* (*). Оператор сам может читаться как «значение, указанное им».

Таким образом, следующее со значением предыдущего примера, выражение будет:

```
baz = *foo;
```

Это можно прочитать так: "baz эквивалентно значению, указываемому foo", и выражение будет реально присваивать значение 25 переменной baz, поскольку foo это 1776, а значение указанное адресом 1776 (следуя примеру выше) будет 25.



Очень важно уяснить разницу: foo ссылается на значение 1776, тогда как *foo (со значком звёздочки *, предшествующим идентификатору) указывает на значение, хранимое по адресу 1776, которое в данном случае 25. Отметьте разницу включения или не включения *оператора разыменовывания* (ниже добавлен поясняющий комментарий для каждого из этих двух выражений, как они должны читаться):

```
1 baz = foo;   // baz эквивалентно foo (1776)
2 baz = *foo;  // baz эквивалентно значению, указанному foo (25)
```

Операторы указателя и разыменовывания, таким образом, комплементарны:

- & это оператор взятия адреса, и может читаться просто как «адрес…»

- * это оператор разыменовывания, и может читаться как «значение, указанное…»

Итак, они имеют как бы противоположное значение: адрес, получаемый с помощью &, может разыменовываться с помощью *.

Выше мы использовали следующие два оператора присваивания:

```
myvar = 25;
foo = &myvar;
```

Сразу после этих двух выражений, все следующие дадут истинный результат:

```
myvar == 25
&myvar == 1776
foo == 1776
*foo == 25
```

Первое выражения совершенно ясно, утверждая, что операция присвоения, выполненная с myvar, была myvar=25. Второе использует оператор получения адреса (&), который возвращает адрес myvar, присваивая значение 1776. Третий – это сама очевидность, поскольку второе выражение было истинно, а операция присвоения, выполненная для foo, была foo=&myvar. Четвёртое выражение использовало *оператор разыменовывания* (*), который может читаться как «значение, указанное…», а значение, указанное foo, действительно 25.

Итак, после всего этого вы можете также сделать вывод, что поскольку адрес, указанный foo, остаётся неизменным, следующее выражение тоже истинно:

```
*foo == myvar
```

Объявление указателей

Благодаря возможности указателя непосредственно ссылаться на значение, на которое он указывает, указатель имеет разные свойства, когда указывает на char, или когда указывает на int или float. Поскольку есть различия, необходимо знать тип. А для этого объявление указателя включает тип данных, на которые указывает указатель.

Объявление указателя имеет следующий синтаксис:

```
type * name;
```

где type – это тип данных, на которые указывает указатель. Это не тип самого укзателя, но именно тип указываемых данных. Например:

```
int * number;
char * character;
double * decimals;
```

Это три объявления укзателей. Каждое из них принадлежит указателю на разные типы данных, но фактически все они указатели и все они, похоже, занимают одинаковое место в памяти (размер, занимаемой указателем памяти, зависит от платформы, на которой работает программа). Тем не менее, данные, на которые они указывают, не занимают одинковое место, не будучи однотипными: первый указывает на int, второй на char, а последний на double. Таким образом, хотя эти три примера переменных все указатели, они имеют разные типы: int*, char*, и double* соответственно, в зависимости от типа, на который указыают.

Заметьте, что звёздочка (*), используемая при объявлении указателя, означает только, что это указатель (это часть его спецификатора сложного типа), и не следует её путать с *оператором разыменовывания*, показанным чуть ранее, который тоже записывается со звёздочкой (*). Они просто две разные вещи, представленные одним и тем же значком.

Давайте взглянем на примеры указателей:

```
// my first pointer
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue, secondvalue;
  int * mypointer;

  mypointer = &firstvalue;
  *mypointer = 10;
  mypointer = &secondvalue;
  *mypointer = 20;
  cout << "firstvalue is " <<
firstvalue << '\n';
  cout << "secondvalue is " <<
secondvalue << '\n';
  return 0;
}
```

```
firstvalue is 10
secondvalue is 20
```

Обратите внимание, что ни `firstvalue`, ни `secondvalue` непосредственно не получают какого-либо значения в программе, оба оказываются со значениями, полученными косвенно при использовании `mypointer`. Вот как это случилось:

Первое, `mypointer` присвоен адрес firstvalue с использованием оператора получения адреса (`&`). Затем значению, указываемому `mypointer`, присваивается величина `10`. Поскольку в этот момент `mypointer` указывает на место расположения в памяти `firstvalue`, этим фактически модифицируется значение `firstvalue`.

В плане демонстрации того, что указатель может указывать на разные переменные в процессе своей работы в программе, пример повторяет процесс с `secondvalue` и тем же указателем `mypointer`.

Вот пример несколько более продуманный:

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue = 5, secondvalue =
15;
  int * p1, * p2;

  p1 = &firstvalue;  // p1 = адрес
firstvalue
  p2 = &secondvalue; // p2 = адрес
secondvalue
  *p1 = 10;          // значение,
указываемое p1 = 10
  *p2 = *p1;         // значению,
указываемому p2, присвоено значение,
```

```
firstvalue is 10
secondvalue is 20
```

```
19 укзаываемое p1
20   p1 = p2;           // p1 = p2
(значение указателя копируется)
   *p1 = 20;          // значение,
указываемое p1 = 20

   cout << "firstvalue is " <<
firstvalue << '\n';
   cout << "secondvalue is " <<
secondvalue << '\n';
   return 0;
}
```

Каждая операция присваивания выше включает комментарий, показывающий, как каждая из строк должна восприниматься: то есть, заменяя амперсанд (`&`) на «адрес…», а звёздочку (`*`) на «значение, указываемое…».

Заметьте, что есть выражения с указателями `p1` и `p2` с и без *оператора разыменовывания* (`*`). Значение выражения, использующего *оператор разыменовывания* (*), очень отличается от того, которое не использует этот оператор. Когда оператор предваряет имя указателя, выражение ссылается на указываемое значение, а когда имя указателя появляется без этого оператора, оно ссылается на значение самого указателя (то есть, на адрес, используемый указателем).

Ещё одно, что может привлечь ваше внимание в строке:

```
int * p1, * p2;
```

Этим объявляются два указателя, использованные в предыдущем примере. Но отметьте, что есть звёздочка (`*`) для каждого указателя, в плане того, что оба имеют тип `int*` (указатель на `int`). Это требуется из-за правила предшествования. Заметьте, что если бы вместо этого был код:

```
int * p1, p2;
```

`p1` был бы действительно типа `int*`, но `p2` был бы типа `int`. Пробелы не играют роли в этом смысле. Но как бы то ни было, просто не забывайте добавлять одну звёздочку на указатель – этого достаточно для большинства случаев использования указателей, если вы заинтересованы в объявлении множества указателей в одном выражении. Или даже лучше: используйте разные выражения для каждой переменной.

Указатели и массивы

Концепция массивов связана с концепцией указателей. Фактически, массивы работают очень похоже на то, как это делают указатели, если рассматривать их первые элементы. И действительно, массивы всегда могут легко преобразовываться в указатели подходящего типа. Например, рассмотрим два объявления:

```
1 int myarray [20];
2 int * mypointer;
```

Следующая операция присваивания будет справедлива:

```
mypointer = myarray;
```

После чего `mypointer` и `myarray` будут эквивалентны, и будут иметь очень похожие свойства.

Основная разница останется в том, что указателю `mypointer` могут быть присвоены разные адреса, тогда как массиву `myarray` никогда не может быть присвоено ничего, и это всегда будет представлять тот же блок из 20 элементов типа `int`. Так что, следующее присваивание не будет верным:

```
myarray = mypointer;
```

Давайте взглянем на пример, который смешивает массивы и указатели:

```
1  // more pointers
2  #include <iostream>
3  using namespace std;
4
5  int main ()
6  {
7    int numbers[5];
8    int * p;
9    p = numbers;  *p = 10;
10   p++;  *p = 20;
11   p = &numbers[2];  *p = 30;
12   p = numbers + 3;  *p = 40;
13   p = numbers;  *(p+4) = 50;
14   for (int n=0; n<5; n++)
15     cout << numbers[n] << ", ";
16   return 0;
17 }
```

```
10, 20, 30, 40, 50,
```

Указатели и массивы поддерживают тот же набор операций с одинковым значением. Разница в основном будет в том, что указателям могут быть присвоены новые адреса, тогда как массивам нет.

В разделе о массивах скобки (`[]`) объяснялись заданием индекса элемента в массиве. Хорошо, фактически эти скобки есть оператор разыменовывания, и известный как *offset operator (оператор смещения)*. Скобки разыменовывают переменную, за которой следуют, подобно тому, как это делает оператор `*`, но они также добавляют число в скобках для адресации того, что будет разыменовываться. Например:

```
1 a[5] = 0;        // a [смещение на 5] = 0
2 *(a+5) = 0;      // указатель на (a+5) = 0
```

Эти два выражения эквивалентны и правильны, не только если `a` является указателем, но также и тогда, когда `a` массив. Помните, что если есть массив, его имя может быть использовано подобно указателю на его первый элемент.

Инициализация указателя

Указатели могут инициализироваться, чтобы задать местоположение в момент определения:

```
1 int myvar;
2 int * myptr = &myvar;
```

Результирующее состояние переменных после этого кода такое же, как после:

```
1 int myvar;
2 int * myptr;
3 myptr = &myvar;
```

Когда указатели получают начальное значение, то, что ими инициализируется – это адрес, на который они указывают (т.е., `myptr`), но никак не значение, на которое указывается (т.е., `*myptr`). Таким образом, код выше не будет спутан с:

```
1 int myvar;
2 int * myptr;
3 *myptr = &myvar;
```

Который, по-любому, не имеет глубокого смысла (и не будет правильным кодом).

Звёздочка (`*`) в объявлении указателя (строка 2) только показазывает, что это указатель, но это не оператор разыменовывания (как в строке 3). В обоих случаях используется одинаковый значок: `*`. Как всегда, пробелы не имеют значения и никогда не меняют значения выражения.

Указатели могут быть инициализированы либо адресом переменной (так, как в случае выше), либо значением другого указателя (или массива):

```
1 int myvar;
2 int *foo = &myvar;
3 int *bar = foo;
```

Арифметика указателей

Поведение арифметических операций с указателями мало отличается от их поведения с обычными целыми типами. Чтобы начать с чего-то – разрешены только операции сложения и вычитания, остальные операции лишены смысла с точки зрения указателей. Но и сложение, и вычитание имеют некоторые отличия в поведении с указателями, из-за размеров типов данных, на которые указывают указатели.

Когда обсуждались базовые типы данных, мы видели, что типы имеют разные размеры. Например, `char` всегда имеют размер в 1 байт, `short` обычно больше одного байта, а `int` и `long` ещё больше; конкретный их размер зависит от системы. Давайте представим, что в данной системе `char` занимает 1 байт, `short` занимает 2 байта, а `long` все 4.

Положим теперь, что мы определяем три указателя в рамках этого компилятора:

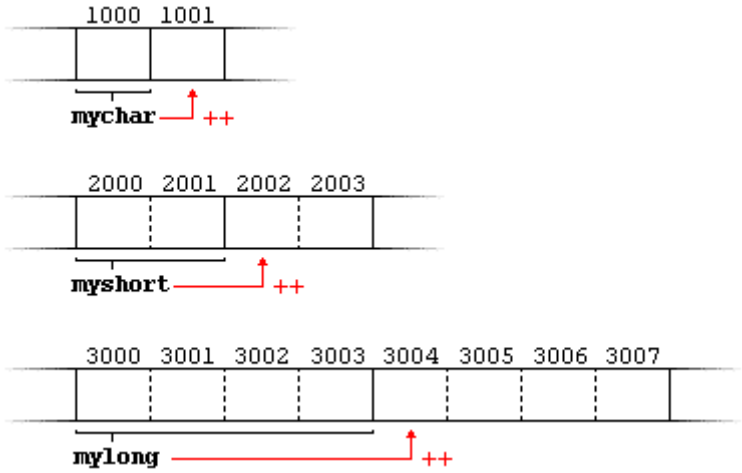
```
1 char *mychar;
2 short *myshort;
3 long *mylong;
```

и что мы знаем, что они указывают на расположение в памяти по адресам `1000`, `2000` и `3000` соответственно.

Поэтому, если мы напишем:

```
1 ++mychar;
2 ++myshort;
3 ++mylong;
```

`mychar`, как ожидалось, будет иметь значение 1001. Но, что не так очевидно, `myshort` будет содержать значение 2002, а `mylong` иметь значение 3004, хотя каждый из указателей был инкрементирован только один раз. Причина в том, что при добавлении единицы к указателю, указатель должен указывать на следующий элемент того же типа, а, следовательно, размер в байтах типа, на который указывается, добавляется к указателю.



Это приложимо и к сложению, и к вычитанию любого числа из указателя. Это произойдёт так же, как если бы мы написали:

```
mychar = mychar + 1;
myshort = myshort + 1;
mylong = mylong + 1;
```

Касательно операторов инкремента (`++`) и декремента (`--`), они оба могут использоваться как в префиксном, так и в суффиксном виде в выражениях с небольшим отличием в поведении: при префиксном использовании инкремент происходит до выполнения выражения, а в суффиксном инкремент имеет место после выполнения выражения. Это также относится к выражениям и инкрементирования и декрементирования указателей, которые могут быть частью более сложных выражений, включая операторы разыменовывания (`*`). Помня правила приоритета операторов, мы можем переиначить так, что постфиксные операторы, такие как инкремент и декремент, имеют больший приоритет, чем префиксные операторы, такие как оператор разыменовывания (`*`). Таким образом, следующее выражение:

```
*p++
```

выполняется как `*(p++)`. И что происходит – инкремент значения `p` (поскольку это теперь указатель на следующий элемент), но, поскольку `++` используется как постфикс, всё выражение выполняется как оригинальное значение, указываемое указателем (адрес, на который он указывает до выполнения инкремента).

В сущности, есть четыре возможных комбинации оператора разыменовывания с префиксными и суффиксными версиями оператора инкремента (это же применимо и к оператору декремента):

```
*p++   // то же, что *(p++): инкремент указателя, разыменовывание не
инкрементированного адреса
*++p   // то же, что *(++p): инкремент указателя, разыменовывание
инкрементированного адреса
++*p   // то же, что ++(*p): разыменовывание указателя, инкремент значения,
на которое указывает указатель
(*p)++ // разыменовывание указателя, последующий инкремент значения, на
которое оно указывает
```

Типичное, но не в таком простом выражении, применение этих операторов:

```
*p++ = *q++;
```

Поскольку ++ имеет более высокий приоритет, чем *, оба и p, и q инкрементируются, но, поскольку оба оператора инкремента (++) используются как постфиксные, а не префиксные, значение, присваиваемое *p, есть *q до того, как и p, и q инкрементируются. А затем оба инкрементируются. Это будет приблизительно эквивалентно:

```
*p = *q;
++p;
++q;
```

Как всегда, скобки уменьшают беспорядок, добавляя удобочитаемости выражениям.

Указатели и постоянные

Указатели могут использоваться для доступа к переменной по её адресу, и этот доступ может включать модификацию указываемого значения. Но также можно объявлять указатели, которые могут дать доступ к указываемым значениям для их чтения, но не для их модификации. Для этого достаточно квалифицировать тип указателя как const. Например:

```
int x;
int y = 10;
const int * p = &y;
x = *p;       // ok: читаем p
*p = x;       // ошибка: изменение указываемого p, заявленного константой
```

Здесь p указывает на переменную, но указатель на неё заявлен как const, что означает, что он может читать указываемое им значение, но не модифицировать это значение. Заметьте также, что выражение &y типа int*, но это присваивается указателю типа const int*. Что позволяет следующее: указатель на не константу может быть неявно конвертирован в указатель на константу. Но не наоборот! В качестве предохранителя – указатели на const не конвертируемы неявно в указатели на не-const.

Один из вариантов использования указателей на элементы const – применить их в качестве параметров функции: функция, которая принимает параметром указатель на не-const может изменять значение, передаваемое в качестве аргумента, тогда как функция, которая принимает указатель на const как параметр, не может.

```
// pointers as arguments:
#include <iostream>
using namespace std;

void increment_all (int* start, int*
stop)
{
  int * current = start;
  while (current != stop) {
    ++(*current);  // increment value
pointed
    ++current;     // increment
pointer
  }
}

void print_all (const int* start,
const int* stop)
{
  const int * current = start;
  while (current != stop) {
    cout << *current << '\n';
```

```
11
21
31
```

```
23      ++current;     // increment
24 pointer
25    }
26 }
27
28 int main ()
29 {
     int numbers[] = {10,20,30};
     increment_all (numbers,numbers+3);
     print_all (numbers,numbers+3);
     return 0;
   }
```

Заметьте, что `print_all` использует указатели на постоянные элементы. Эти указатели указывают на постоянное содержимое, которое они не могут менять, но они сами не являются постоянными: т.е., указатели могут инкрементироваться или им могут присваиваться разные адреса, хотя они и не могут изменять содержимое, на которое они указывают.

И это второе измерение постоянства, добавляемое указателям: указатели тоже могут быть сами постоянными. Это задаётся добавлением *const* к типу указателя (после звёздочки):

```
1 int x;
2       int *       p1 = &x;  // не постоянный указатель на не постоянное int
3 const int *       p2 = &x;  // не постоянный уазатель на постоянное int
4       int * const p3 = &x;  // постоянный уазатель на не постоянное int
5 const int * const p4 = &x;  // постоянный указатель на постоянное int
```

Синтаксис с `const` и указателями безусловно сложнее, а определение случаев, когда каждый из вариантов применим наилучшим образом, требует определённого опыта. В любом случае важно получить постоянство с указателями (и ссылками) скорее раньше, чем позже, но вы не должны слишком беспокоиться, чтобы охватить всё, если это только первый опыт знакомства со смесью `const` и указателей. Больше случаев использования этого будет показано в следующих главах.

Чтобы добавить немного больше путаницы к синтаксису `const` с указателями, квалификатор `const` может либо предшествовать, либо следовать за типом указателя, с точно таким же смыслом:

```
1 const int * p2a = &x;  //   не постоянный указатель на постоянное int
2 int const * p2b = &x;  // также не постоянный указатель на постоянное int
```

Как и с пробелами вокруг звёздочек, порядок *const* в этом случае просто вопрос стиля. Эта глава использует префикс `const`, скорее, из исторических соображений, что, вероятно, должно быть расширено, но оба варианта полностью эквивалентны. Достоинства каждого стиля всё ещё интенсивно обсуждаются в интернете.

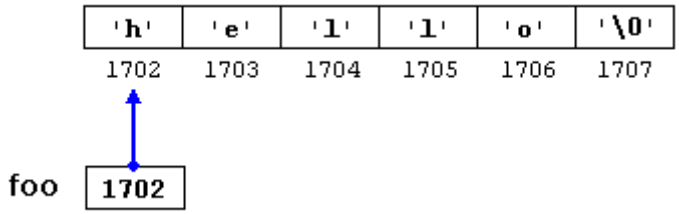
Указатели и строковые литералы

Как указывалось ранее, *строковые литералы* – это массивы, содержащие последовательности символов с нуль-символом. В предыдущих разделах строковые литералы были использованы для непосредственной вставки в `cout`, для инициализации строк и инициализации массивов символов.

Но они могут быть доступны и напрямую. Строковые литералы – это массивы подходящего типа для хранения всех его символов плюс завершающий нуль-символ. А каждый из элементов имеет тип `const char` (как литералы, они не могут быть изменены). Например:

```
const char * foo = "hello";
```

Этим объявляется массив с литералами, представляющими "hello", и затем указатель на его первый элемент, присвоенный foo. Если мы представим, что "hello" хранится в памяти, начинающейся с адреса 1702, мы можем изобразить предыдущее объявление как:



Заметьте, что здесь foo – это указатель, и он имеет значение 1702, а не 'h' или "hello", хотя 1702 действительно начальный адрес.

Указатель foo указывает на последовательность символов. А, поскольку указатели и массивы ведут себя схожим образом в выражениях, foo может использоваться для доступа к символам тем же образом, что и массив с последовательностью, завершающейся нуль-символом. Например:

```
1 *(foo+4)
2 foo[4]
```

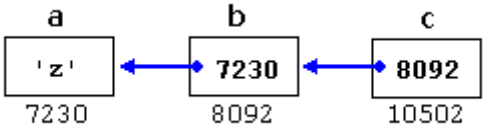
Оба выражения имеют значения 'o' (пятый элемент массива).

Указатели на указатели

C++ позволяет использовать указатели, которые указывают на указатели, что, в свою очередь, указывает на данные (или даже на другие указатели). Синтаксис просто требует звёздочки (*) для каждого уровня абстракции в объявлении указателя:

```
1 char a;
2 char * b;
3 char ** c;
4 a = 'z';
5 b = &a;
6 c = &b;
```

Так, предположительно случайный выбор расположения в памяти для каждой переменной 7230, 8092 и 10502, может быть представлен так:



Со значением каждой переменной, представленной в её соответствующей ячейке, представлен и её соответствующий адрес в памяти, показанный значением под ними.

Новым в этом примере будет переменная c, которая есть указатель на указатель, и может использоваться на трёх разных уровнях абстракции, каждый из которых относится к разным значениям:

- c это тип char** и значение 8092
- *c это тип char* и значение 7230

- `**c` это тип `char` и значение `'z'`

void указатели

Тип `void` указателя – это специальный тип указателя. В C++ `void` означает отсутствие типа. Таким образом, `void` указатели – это указатели, которые указывают на значение, не имеющее типа (и также не определена длина свойства разыменовывания).

Это даёт `void` указателям больше гибкости, разрешая указывать на любые типы данных, от целых или значений с плавающей точкой до строк символов. Взамен они имеют большие ограничения: данные, указываемые ими, не могут напрямую разыменовываться (что логично, поскольку мы не имеем типа для этой операции), и из этих соображений любой адрес в `void` указателе нуждается в преобразовании в некий другой тип указателя, указывающего на конкретный тип данных до разыменовывания.

Одна из возможностей использования – передача общих параметров в функцию. Например:

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

void increase (void* data, int psize)
{
  if ( psize == sizeof(char) )
  { char* pchar; pchar=(char*)data;
++(*pchar); }
  else if (psize == sizeof(int) )
  { int* pint; pint=(int*)data;
++(*pint); }
}

int main ()
{
  char a = 'x';
  int b = 1602;
  increase (&a,sizeof(a));
  increase (&b,sizeof(b));
  cout << a << ", " << b << '\n';
  return 0;
}
```

```
y, 1603
```

`sizeof` – это оператор, интегрированный в язык C++, который возвращает размер в байтах его аргумента. Для нединамических типов данных это значение будет константой. Таким образом, например `sizeof(char)` есть 1, поскольку `char` всегда имеет размер в один байт.

Неверные указатели и нуль указатели

В принципе указатели предназначены для указания правильных адресов, таких как адреса перменных или адреса элементов в массиве. Но указатель может реально указывать на любой адрес, включая адреса, которые не ссылаются ни на какие правильные элементы. Типичным примером этого являются *неинициализированные указатели* и указатели на несуществующие элементы массива:

```
int * p;           // неинициализированный указатель (локальная переменная)

int myarray[10];
int * q = myarray+20;  // элемент вне границ
```

Ни `p`, ни `q` не указывают на адреса, которые содержали бы значения, но ни одно из выше показанных выражений не приводит к ошибке. В C++ указатели позволяют получить любые адресованные значения вне зависимости от того, есть ли что-то по этому адресу или нет. Что может привести к ошибке – это разыменовывание подобных указателей (т.е., получение актуального значения, на которое они указывают). Доступ с помощью таких указателей приводит к неопределённому поведению, приводящему к ошибкам при работе программы, получающей доступ к каким-то случайным значениям.

Но иногда указателю действительно необходимо явным образом указывать в никуда, и не только на неправильный адрес. Для подобных случаев есть специальное значение, которое любой тип указателя может получить: *null pointer value (значение нуль указателя)*. Это значение может быть выражено в C++ двумя способами: либо целым значением ноль, либо ключевым словом `nullptr`:

```
1 int * p = 0;
2 int * q = nullptr;
```

Здесь и `p`, и `q` есть *нуль-указатели*, означающие, что они прямо указывают в никуда, и они актуально сравниваются одинаково: все *нуль-указатели* при сравнении эквивалентны другим *нуль-указателям*. Это можно увидеть в определении константы `NULL`, используемой в старых кодах для ссылки на значение нуль-указателя:

```
int * r = NULL;
```

`NULL` определено в нескольких файлах заголовков стандартной библиотеки. И определено как альтернативное имя постоянных значений некоторых *нуль-указателей* (как `0` или `nullptr`).

Не путайте *нуль-указатели* с `void` указателями! Нуль-указатель – это значение, которое любой указатель может взять для представления, и что он указывает в «никуда», тогда как `void` указатель – это тип указателя, который может указывать куда-то без задания типа. Один указывает на значение, хранимое в указателе, а другой на тип данных, на которые он указывает.

Указатели на функции

C++ допускает операции с указателями на функции. Типичное использование этого – передача функции, как аргумента, в другую функцию. Указатели на функции объявляются тем же синтаксисом, что и обычные объявления функций, исключая, что имя функции берётся в скобки () и перед именем вставляется звёздочка (*):

```
1  // pointer to functions
2  #include <iostream>
3  using namespace std;
4
5  int addition (int a, int b)
6  { return (a+b); }
7
8  int subtraction (int a, int b)
9  { return (a-b); }
10
11 int operation (int x, int y, int
12 (*functocall)(int,int))
13 {
14   int g;
15   g = (*functocall)(x,y);
16   return (g);
17 }
18
19 int main ()
```

```
8
```

```
{
  int m,n;
  int (*minus)(int,int) =
subtraction;

  m = operation (7, 5, addition);
  n = operation (20, m, minus);
  cout <<n;
  return 0;
}
```

В примере выше `minus` является указателем на функцию, которая имеет два параметра типа `int`. Он непосредственно инициализируется, чтобы указать на функцию `subtraction`:

```
Int (* minus)(int,int) = subtraction;
```

Динамическая память

В программах, показанных в предыдущих главах, вся память нуждалась в определени до того, как программа будет работать, что осуществляется через определение всех нужных переменных. Но могут быть случаи, когда память, нужная программе, может определиться только при работе программы. Например, когда нужная память зависит от ввода пользователя. В этих случаях программе нужно динамическое распределение памяти, для которого в язык C++ встроены операторы `new` и `delete`.

Операторы new и new[]

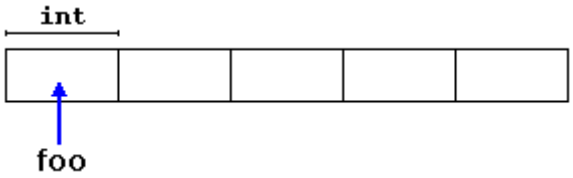
Динамическое выделение памяти осуществляется с помощью оператора `new`. `new` сопровождается спецификатором типа данных и, если требуется последовательность элементов более одного, их количество задаётся в квадратных скобках `[]`. Оператор возвращает указатель на начало нового блока памяти. Синтаксис такой:

```
pointer = new type
pointer = new type [number_of_elements]
```

Первое выражение, используемое для выделения памяти, содержит один элемент типа `type`. Второй используется для определения блока (массива) элементов типа `type`, где `number_of_elements` – это целое значение, представляющее количество этих элементов. Например:

```
1 int * foo;
2 foo = new int [5];
```

В этом случае система динамически резервирует пространство для пяти элементов типа `int` и возвращает указатель на первый элемент последовательности, что присваивается `foo` (указатель). Таким образом, теперь `foo` указывает правильный сегмент памяти с пространством для размещения пяти элементов типа `int`.



Здесь `foo` – это указатель, в результате первый элемент, указываемый `foo`, может быть достигнут либо с помощью выражения `foo[0]`, либо выражения `*foo` (оба эквивалентны). Второй элемент может быть достигнут либо с помощью `foo[1]` , либо `*(foo+1)`, и т.д.

Есть существенная разница между объявлением обычного массива и динамическим выделением блока памяти с использованием `new`. Самая большая разница в том, что размер обычного массива должен быть *постоянным выражением*, то есть, его размер должен быть определён в момент разработки программы, до её запуска, тогда как динамическое выделение выполняется оператором `new`, позволяя обращаться к памяти в процессе работы программы, используя любые переменные значения в качестве размера.

Динамическая память, запрашиваемая нашей программой, выделяется системой из области свободной памяти (heap). Однако компьютерная память – это ограниченный ресурс, и он может быть полностью использован. Таким образом, нет гарантий, что все требования по выделению памяти оператором `new` будут обязательно выполнены системой.

C++ поддерживает два стандартных механизма для проверки, что выделение прошло успешно:

Один из них – поддержка исключений. При использовнии этого метода отправляется исключение типа `bad_alloc`, если выделение потерпело неудачу. Исключения – это мощный механизм C++, который будет пояснён в этом руководстве позже. Но сейчас вы должны знать, что если

отправлено такое исключение, если оно не поддержано специальным средством поддержки, выполнение программы будет прервано.

Этот метод исключений используется по умолчанию для `new`, и это тот, что используется при декларациях вида:

```
foo = new int [5];  // если нет места, вбрасывается исключение
```

Другой метод известен как `nothrow`, и вот, что происходит при его применении: при отказе в выделении памяти вместо вывода `bad_alloc` исключения или прерывания программы указатель, возвращаемый оператором `new`, становится *нуль-указателем*, а программа продолжает работу обычным образом.

Этот метод может быть задан использованием специального объекта, названного `nothrow` и объявленнного в файле заголовка `<new>`, как аргумент для `new`:

```
foo = new (nothrow) int [5];
```

В этом случае, если выделение блока памяти невозможно, отказ программы может быть обнаружен при проверке `foo`, не стал ли указтель *null*:

```
1 int * foo;
2 foo = new (nothrow) int [5];
3 if (foo == nullptr) {
4   // ошибка выделения памяти. Принимайте меры.
5 }
```

Этот `nothrow` метод, похоже, производит менее эффективный код, чем исключения, поскольку он предполагает явную проверку значения указателя, возвращаемоего после каждого распределения памяти. Таким образом, механизм исключения обычно предпочтительнее хотя бы для критичных выделений памяти. Тем не менее, большинство примеров используют `nothrow` механизм из-за его простоты.

Операторы delete и delete[]

В большинстве случаев динамически выделенная память нужна только в заданный период времени в программе. А когда она больше не нужна, память может быть освобождена, став доступна для других динамических запросов. Для этой цели служит оператор `delete`, чей синтаксис:

```
1 delete pointer;
2 delete[] pointer;
```

Первое выражение освобождает память, выделенную единственну элементу, использовавшему `new`, а второе освобождает память, выделенную массиву элементов, использовавшему new и размер в квадратных скобках (`[]`).

Значение, передаваемое в качестве аргумента в `delete`, будет либо указателем на блок памяти, прежде выделенный с помощью `new`, либо *нуль-указателем* (в случае *нуль-указателя* `delete` не производит эффекта).

```
1 // rememb-o-matic
2 #include <iostream>
3 #include <new>
4 using namespace std;
```

```
How many numbers would you like to
type? 5
Enter number : 75
Enter number : 436
```

```
5
6  int main ()
7  {
8    int i,n;
9    int * p;
10   cout << "How many numbers would you
11 like to type? ";
12   cin >> i;
13   p= new (nothrow) int[i];
14   if (p == nullptr)
15     cout << "Error: memory could not
16 be allocated";
17   else
18   {
19     for (n=0; n<i; n++)
20     {
21       cout << "Enter number: ";
22       cin >> p[n];
23     }
24     cout << "You have entered: ";
25     for (n=0; n<i; n++)
26       cout << p[n] << ", ";
27     delete[] p;
28   }
   return 0;
}
```

```
Enter number : 1067
Enter number : 8
Enter number : 32
You have entered: 75, 436, 1067, 8,
32,
```

Заметьте, что значение в квадратных скобках в выражении new – это значение переменной, введённой пользователем (`i`), а не постоянное выражение:

```
p= new (nothrow) int[i];
```

Всегда есть возможность, что пользователь введёт значение для `i` столь большое, что система не сможет выделить для него достаточно памяти. Например, если я попробую задать значение 1 миллиард для вопроса "How many numbers", моя система не сможет выделить столько памяти для программы, и я получу текстовое сообщение, приготовленное нами для этого случая (`Error: memory could not be allocated`).

Есть хорошая практика для программ – всегда поддерживать сбои при выделении памяти либо проверкой значения указателя (при `nothrow`), либо захватом надлежащих исключений.

Динамическая память в C

C++ интегрирует операторы `new` и `delete` для работы с динамической памятью. Но это не доступно в языке Си, вместо этого используется библиотечное решение функциями `malloc`, `calloc`, `realloc` и `free`, определёнными в заголовочном файле `<cstdlib>` (известном как `<stdlib.h>` в Си). Эти фукнции так же доступны в C++ и могут использоваться для выделения и освобождения динамической памяти.

Отметьте для себя, что блоки памяти, выделяемые этими функциями, не обязательно совместимы с теми, что выделяются оператором `new`, так что их не стоит смешивать, каждый из них должен быть поддержан своим собственным набором функций или операторов.

Структуры данных

Структуры данных

Структура данных – это группа элементов данных, собранных вместе под одним именем. Эти элементы данных, известные как *члены* (*members)*, могут иметь разные типы и разную длину. Структуры данных в C++ используют следующий синтаксис:

```
struct type_name {
member_type1 member_name1;
member_type2 member_name2;
member_type3 member_name3;
.
.
} object_names;
```

Где `type_name` является именем типа структуры, `object_name` может быть набором правильных идентификаторов для объектов, которые имеют тип этой структуры. Внутри фигурных скобок `{}` есть список членов данных, каждый из которых с типом и правильным идентификатором, представленным его именем.

Например:

```
1 struct product {
2   int weight;
3   double price;
4 } ;
5
6 product apple;
7 product banana, melon;
```

Здесь объявлен тип структуры, названной `product`, и определено, что она имеет два члена: `weight` и `price`, каждый из которых разного фундаментального типа. Эта декларация создаёт новый тип (`product`), который затем используется для объявления трёх объектов (переменных) таких типов: `apple`, `banana` и `melon`. Заметьте, как объявляется тип `product`, он используется, как любой другой тип.

Сразу в конце определения `struct` и перед закрывающими точкой с запятой (`;`) может использоваться необязательное поле `object_names`, чтобы непосредственно объявить объекты этой структуры. Например, объекты структуры типа `apple`, `banana` и `melon` могут быть объявлены при определении структуры данных:

```
1 struct product {
2   int weight;
3   double price;
4 } apple, banana, melon;
```

В этом случае на месте `object_names` заданы объекты, а имя типа (`product`) становится необязательно: `struct` требует либо `type_name`, либо хотя бы одного имени в `object_names`, но не обязательно обоих.

Важно ясно понимать разницу между тем, что есть имя типа структуры (`product`), и что такое объект этого типа (`apple`, `banana` и `melon`). Многие объекты (такие как `apple`, `banana` и `melon`) могут быть объявлены из единственного типа структуры (`product`).

Когда три объекта определённого типа структуры объявляются (`apple`, `banana` и `melon`), их члены достижимы напрямую. Синтаксис для этого простой – добавление точки (`.`) между именем

объекта и именем члена. Например, мы можем оперировать с любым из этих элементов, как если бы они были стандартными переменными их соответствующих типов:

```
apple.weight
apple.price
banana.weight
banana.price
melon.weight
melon.price
```

Каждый пример из показанного выше имеет тип данных, соответствующий тому члену, на который ссылается: `apple.weight`, `banana.weight` и `melon.weight` все типа `int`, а `apple.price`, `banana.price` и `melon.price` типа `double`.

Вот реальный пример типов структур в действии:

```
// example about structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct movies_t {
  string title;
  int year;
} mine, yours;

void printmovie (movies_t movie);

int main ()
{
  string mystr;

  mine.title = "2001 A Space
Odyssey";
  mine.year = 1968;

  cout << "Enter title: ";
  getline (cin,yours.title);
  cout << "Enter year: ";
  getline (cin,mystr);
  stringstream(mystr) >> yours.year;

  cout << "My favorite movie is:\n ";
  printmovie (mine);
  cout << "And yours is:\n ";
  printmovie (yours);
  return 0;
}

void printmovie (movies_t movie)
{
  cout << movie.title;
  cout << " (" << movie.year <<
")\n";
}
```

```
Enter title: Alien
Enter year: 1979

My favorite movie is:
 2001 A Space Odyssey (1968)
And yours is:
 Alien (1979)
```

Пример показывает, как члены объектов действуют подобно обычным переменным. Например, член `yours.year` имеет правильный тип переменной `int`, а `mine.title` правильный тип

переменной `string`.

Но объекты `mine` и `yours` также переменые, имеющие тип (типа `movies_t`). Например, оба были преданы в функцию `printmovie` точно так же, как если бы были простыми переменными. Таким образом, одна из возможностей структур данных – это способность ссылаться и на оба их члена индивидуально, и всю структуру в целом. В обоих случаях используется тот же самый идентификатор: имя структуры.

Поскольку структуры – это типы, они могут также быть использованы как тип массивов для создания таблиц или баз данных:

```
// array of structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct movies_t {
  string title;
  int year;
} films [3];

void printmovie (movies_t movie);

int main ()
{
  string mystr;
  int n;

  for (n=0; n<3; n++)
  {
    cout << "Enter title: ";
    getline (cin,films[n].title);
    cout << "Enter year: ";
    getline (cin,mystr);
    stringstream(mystr) >>
films[n].year;
  }

  cout << "\nYou have entered these
movies:\n";
  for (n=0; n<3; n++)
    printmovie (films[n]);
  return 0;
}

void printmovie (movies_t movie)
{
  cout << movie.title;
  cout << " (" << movie.year <<
")\n";
}
```

```
Enter title: Blade Runner
Enter year: 1982
Enter title: The Matrix
Enter year: 1999
Enter title: Taxi Driver
Enter year: 1976

You have entered these movies:
Blade Runner (1982)
The Matrix (1999)
Taxi Driver (1976)
```

Указатели на структуры

Подобно другим типам структуры могут указываться их собственным типом указателей:

```
struct movies_t {
  string title;
  int year;
```

```
};

movies_t amovie;
movies_t * pmovie;
```

Здесь amovie есть объект типа структуры movies_t, а pmovie – это указатель, указывающий на объекты типа структуры movies_t. Таким образом, следующий код будет правильным:

```
pmovie = &amovie;
```

Значению указателя pmovie будет присвоен адрес объекта amovie.

Теперь, давайте взглянем на другой пример, который смешивает указатели и структуры, и будет использован для представления нового оператора: оператора стрелки (->):

```
// pointers to structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct movies_t {
  string title;
  int year;
};

int main ()
{
  string mystr;

  movies_t amovie;
  movies_t * pmovie;
  pmovie = &amovie;

  cout << "Enter title: ";
  getline (cin, pmovie->title);
  cout << "Enter year: ";
  getline (cin, mystr);
  (stringstream) mystr >> pmovie->year;

  cout << "\nYou have entered:\n";
  cout << pmovie->title;
  cout << " (" << pmovie->year << ")\n";

  return 0;
}
```

```
Enter title: Invasion of the body snatchers
Enter year: 1978

You have entered:
Invasion of the body snatchers (1978)
```

Оператор стрелки (->) – это оператор разыменовывания, который используется исключительно с указателями на объекты, которые имеют своих членов. Этот оператор используется для доступа к членам объекта непосредственно по их адресам. Например, в примере выше:

```
pmovie->title
```

будет для всех применений эквивалентно:

```
(*pmovie).title
```

Оба выражения, `pmovie->title` и `(*pmovie).title` правильны, и оба обращаются к члену `title` структуры данных, указанному с помощью указателя, названного `pmovie`. И это точно нечто иное, чем:

```
*pmovie.title
```

что, в свою очередь, эквивалентно:

```
*(pmovie.title)
```

Этим осуществляется доступ к значению, указываемому гипотетическим указателем члена, названного `title` объекта структуры `pmovie` (чего не скажешь, поскольку `title` не тип указателя). Следующая таблица суммирует возможные комбинации операторов для указателей членов структуры:

Выражение	Что вычисляется	Эквивалент
`a.b`	Член `b` объекта	
`a->b`	Член `b` объекта, указываемого `a`	`(*a).b`
`*a.b`	Значение, указываемого членом `b` объекта `a`	`*(a.b)`

Вложенные структуры

Структуры тоже могут быть вложенны так, что элементы структуры сами по себе будут другими структурами:

```
struct movies_t {
  string title;
  int year;
};

struct friends_t {
  string name;
  string email;
  movies_t favorite_movie;
} charlie, maria;

friends_t * pfriends = &charlie;
```

После предыдущих деклараций все следующие выражения будут правильны:

```
charlie.name
maria.favorite_movie.title
charlie.favorite_movie.year
pfriends->favorite_movie.year
```

(где, кстати, последние два выражения ссылаются на одни и те же члены).

Другие типы данных

Псевдонимы типа (typedef / using)

Псевдоним типа (*type alias*) – это другое имя, которым может быть идентифицирован тип. В C++ любой правильный тип может быть переименован так, чтобы на него можно было ссылаться с помощью другого идентификатора.

В C++ есть два варианта синтаксиса для создания таких псевдонимов типа: первый наследован от языка Си с помощью ключевого слова `typedef`:

```
typedef existing_type new_type_name ;
```

где `existing_type` – это любой тип, базовый или сложный, а `new_type_name` – это идентификатор с новым именем, данным этому типу данных.

Например:

```
1 typedef char C;
2 typedef unsigned int WORD;
3 typedef char * pChar;
4 typedef char field [50];
```

Здесь объявлены четыре псевдонима типа: `C`, `WORD`, `pChar` и `field` как `char`, `unsigned int`, `char*` и `char[50]` соответственно. Когда эти псевдонимы объявлены, они могут использоваться в любых объявлениях подобно любым другим правильным типам:

```
1 C mychar, anotherchar, *ptc1;
2 WORD myword;
3 pChar ptc2;
4 field name;
```

Позже второй синтаксический вариант определения псевдонима типа был введён в язык C++:

```
using new_type_name = existing_type ;
```

Например, те же псевдонимы типов, что выше, могут быть определены так:

```
1 using C = char;
2 using WORD = unsigned int;
3 using pChar = char *;
4 using field = char [50];
```

Оба определения псевдонимов и `typedef`, и `using` семантически эквивалентны. Разница только в том, что `typedef` имеет некоторые ограничения в области шаблонов (templates), которых `using` лишено. Таким образом, `using` более общий вариант, хотя `typedef` имеет долгую историю и, возможно, шире представлено в существующих кодах.

Заметьте, что ни `typedef`, ни `using` не создают новых типов данных. Они только создают синонимы существующих типов. Что означает – тип `myword` выше, объявленный как тип `WORD`, может так же хорошо подразумеваться как тип `unsigned int`; фактически это неважно, поскольку оба ссылаются на тот же самый тип данных.

Псевдонимы типа могут использоваться для уменьшения длины для длинных или путаных имён типов, но они более полезны в качестве инструмента для абстрагирования программ от подразумеваемых типов, которые они используют. Например, использование псевдонима `int` для

ссылки на частную разновидность параметра вместо прямого использования `int` позволяет легче заместить его типом `long` (или другим типом) в более поздних версиях без необходимости менять каждый образец, где это использовано.

Объединения (Unions)

Объединения позволяют одной «порцией» памяти обеспечивать доступ к разным типам данных. Их декларация и использование похожи на структуры, но их функционирование абсолютно иное:

```
union type_name {
  member_type1 member_name1;
  member_type2 member_name2;
  member_type3 member_name3;
  .
  .
} object_names;
```

Здесь создаётся новый тип объединения, идентифицированный как `type_name`, в котором все элементы его членов занимают одно физическое место в памяти. Размер этих типов определяется по наибольшему элементу из членов объединения. Например:

```
1 union mytypes_t {
2   char c;
3   int i;
4   float f;
5 } mytypes;
```

объявляется объект (`mytypes`) с тремя членами:

```
1 mytypes.c
2 mytypes.i
3 mytypes.f
```

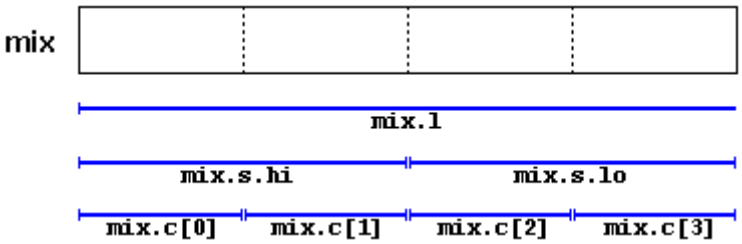
Каждый из этих членов разного типа данных. Но поскольку на них всех ссылка в одно и то же место памяти, изменение одного из членов объединения скажется на значениях всех из них. Невозможно хранить разные значения в них так, чтобы каждый не зависел от остальных.

Одно из использований объединения – это возможность доступа к значениям либо полностью, либо как к массиву, либо как к структуре меньших элементов. Например:

```
1 union mix_t {
2   int l;
3   struct {
4     short hi;
5     short lo;
6     } s;
7   char c[4];
8 } mix;
```

Если мы примем, что система, где работает программа, имеет тип `int` размером в 4 байта, а тип `short` в 2 байта, объединение, определённое выше, позволяет получить доступ к одинаковым группам из 4 байтов: `mix.l`, `mix.s` и `mix.c`. Их мы можем использовать согласно с тем, как мы хотим получить доступ к этим байтам: как если бы они были единичными значениями типа `int`, либо как если бы они были сдвоенными значениями типа `short`, либо как к массиву `char` элементов, соответственно. Пример смешанных типов, массивов и структур в объединении

демонстрирует разные способы доступа к данным. Для систем с прямым порядком даных (little-endian) это объединение может быть представлено как:



Получаемый аргумент и порядок членов объединения в памяти зависит от системы, и есть возможность создать компактные, переносимые решения.

Анонимные объединения

Когда объединения являются членами класса (или структуры), они могут декларироваться без имени. Этот случай становится *анонимным объединением*, а его члены доступны непосредственно из объектов по их именам членов. Например, посмотрите разницу между этими двумя объявлениями структуры:

Структура с обычным объединением	**Структура с анонимным объединением**

`struct book1_t {`
`  char title[50];`
`  char author[50];`
`  union {`
`  float dollars;`
`  int yen;`
`  } price;`
`} book1;`
	`struct book2_t {`
	`  char title[50];`
	`  char author[50];`
	`  union {`
	`  float dollars;`
	`  int yen;`
	`  };`
	`} book2;`

Только одна разница между двумя типами в том, что в первом член объединения имеет имя (`price`), тогда как во втором имени нет. Это сказывается на способе доступа к членам `dollars` и `yen` объекта этого типа. Для объекта первого типа (с обычным объединением), это будет:

```
1 book1.price.dollars
2 book1.price.yen
```

тогда как для объекта второго типа (анонимного объединения), это будет:

```
1 book2.dollars
2 book2.yen
```

Вновь, помните, что поскольку это член объединения (а не член структуры), члены `dollars` и `yen` на самом деле в одном месте памяти, так что они не могут использоваться для хранения двух разных значений одновременно. `price` может быть набором `dollars` или `yen`, но не обоих одновременно.

Перечисляемые типы (enum)

Перечисляемые типы – это типы, которые определяются набором пользовательских идентификаторов, известных как *enumerators (перечислители)*, в качестве возможных значений. Объекты этих *перечисляемых типов* могут принимать любые из этих перечислителей в качестве значения.

Их синтаксис:

```
enum type_name {
  value1,
  value2,
  value3,
  .
  .
} object_names;
```

Этим создаётся тип `type_name`, который может принимать любые из `value1`, `value2`, `value3`, ... как значение. Объекты (переменные) этого типа могут прямо быть представлены как `object_names`.

Например, новый тип переменной, названной `colors_t`, может быть определён для хранения colors следующими объявлениями:

```
enum colors_t {black, blue, green, cyan, red, purple, yellow, white};
```

Заметьте, что эта декларация не включает другие типы, как основные, так и сложные, в этом определении. Можно сказать и иначе, так создаётся полностью новый тип данных «с нуля» без опоры на любые другие существующие типы. Возможные значения этих переменных нового типа `color_t`, которые они могут получить, это перечислители из списка в скобках. Например, если `colors_t` перечисляемого типа объявляется, следующие выражения будут справедливы:

```
1 colors_t mycolor;
2
3 mycolor = blue;
4 if (mycolor == green) mycolor = red;
```

Значения *перечисляемых типов*, объявляемые с помощью `enum`, неявно конвертируемы в целый тип `int`, и наоборот. Фактически элементы, такие как `enum`, всегда внутренне определяются как эквивалент целых чисел, какими они становятся по псевдониму. Если не задано иное, эквивалент целого значения для первого доступного значения будет `0`, эквивалент для второго `1`, для третьего `2` и т.д. Таким образом, в типе данных `colors_t`, определённом выше, `black` будет эквивалентно `0`, `blue` будет эквивалентно `1`, `green` эквивалентно `2` и т.д.

Специальные целые значения могут быть заданы для любого допустимого значения в перечисляемом типе. И если постоянное значение, которое следует за ним, не получило собственного значения, то автоматически подразумевается, что это то же значение плюс один. Например:

```
1 enum months_t { january=1, february, march, april,
2                 may, june, july, august,
3                 september, october, november, december} y2k;
```

В этом случае переменная `y2k` перечисляемого типа `months_t` может содержать любое из 12 допустимых значений от `january` до `december`, и есть эквивалент значениям между `1` и `12` (не между `0` и `11`, поскольку `january` было сделано равно `1`).

Поскольку перечисляемые типы, объявляемые с помощью `enum`, внутренне конвертируемы в `int`, и каждое значение перечислителя фактически типа `int`, нет способа разглядеть `1` за `january` – они строго те же самые значения того же типа. Основания для этого унаследованы от языка Си.

Перечисляемые типы с enum class

Но в C++ есть возможность создавать реальные `enum` типы, которе не будут внутренне конвертируемы в `int`, которые не будут иметь значения перечислятеля типа `int`, но собственно `enum` тип, сохраняя безопасность типа. Они декларируются с `enum class` (или `enum struct`) вместо простого `enum`:

```
enum class Colors {black, blue, green, cyan, red, purple, yellow, white};
```

Каждое из значений перечислителя `enum class` типа нуждается в том, чтобы быть в границах его типа (это, в общем-то, возможно и для `enum` типов, но там это не обязательно). Например:

```
1 Colors mycolor;
2
3 mycolor = Colors::blue;
4 if (mycolor == Colors::green) mycolor = Colors::red;
```

Перечисляемые типы, объявленные с `enum class`, также имеют больше управления над их базовыми типами. Это может быть любой целостный тип данных, такой как `char`, `short` или `unsigned int`, который, в сущности, применяется для определения размера типа. Это задаётся двоеточием и базовым типом, следующими за перечисляемым типом. Например:

```
enum class EyeColor : char {blue, green, brown};
```

Здесь `Eyecolor` – это явный тип с тем же размером `char` (1 байт).

Классы

Классы (I)

Классы – это расширенная концепция *структур данных*: подобно структурам данных, они могут содержать члены-данные, но могут также содержать функции в качестве членов.

Объект – это экземпляр реализации класса. В терминах переменных класс будет типом, а объект будет переменной.

Классы определяются с использованием либо ключевого слова `class`, либо ключевого слова `struct`, со следующим синтаксисом:

```
class class_name {
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  ...
} object_names;
```

Где `class_name` является правильным идентификатором для класса, а `object_names` не обязательный список имён объектов класса. Тело объявления может содержать *члены*, которые могут быть декларациями данных или функций и необязательных *спецификаторов доступа*.

Классы имеют тот же формат, что и явные *структуры данных*, исключая то, что они могут включать функции и иметь такие новые сущности, как *спецификаторы доступа*. *Спецификаторы доступа* – это следующие три ключевых слова: `private`, `public` и `protected`. Эти спецификаторы изменяют права доступа для членов, которые следуют за ними:

- `private` (частный) члены класса доступны только из других членов этого же класса (или из их «*друзей*, *friends*»).
- `protected` (защищённый) члены доступны из других членов того же класса (или их «*друзей*, *friends*»), но также из членов их производных классов.
- Наконец, `public` (общие) члены доступны из любого места, где виден объект.

По умолчанию все члены класса, объявленные с ключевым словом `class`, имеют *private* доступ для всех его членов. Таким образом, любой член, который заявлен перед другими *спецификаторами доступа*, имеет автоматически *private* доступ. Например:

```
1 class Rectangle {
2     int width, height;
3   public:
4     void set_values (int,int);
5     int area (void);
6 } rect;
```

Декларирован класс (т.е., тип), названный `Rectangle`, и объект (т.е., переменная) этого класса, названный `rect`. Этот класс содержит четыре члена: два члена данных типа `int` (член `width` и член `height`) с доступом *private access* (поскольку *private* – уровень доступа по умолчанию) и две функции-члены (*member functions*) с *public access*: функции `set_values` и `area`, для которых сейчас включены их объявления, но не их определения.

Отметьте разницу между *class name (имя класса)* и *object name (имя объекта)*: в предыдущем примере `Rectangle` было *именем класса* (т.е., типом), тогда как `rect` было объектом типа `Rectangle`. Та же связь `int` и `a` имеет быть в следующей декларации:

```
int a;
```

где `int` – это имя типа (класс), а `a` – это имя переменной (объект).

После объявления `Rectangle` и `rect` любые *public*-члены объекта `rect` могут быть доступны, как если бы они были обычными функциями или обычными переменными, просто добавлением точки (`.`) между *именем объекта* и *именем члена*. Это влечёт тот же синтаксис, что и для доступа к членам явных сруктур данных. Например:

```
rect.set_values (3,4);
myarea = rect.area();
```

Те члены `rect`, которые не могут достигаться извне класса, это `width` и `height`, поскольку они имеют *private* доступ, на них могут ссылаться только другие члены внутри того же класса.

Вот полный пример класса Rectangle:

```
// classes example
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area() {return width*height;}
};

void Rectangle::set_values (int x,
int y) {
  width = x;
  height = y;
}

int main () {
  Rectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
  return 0;
}
```

```
area: 12
```

Этот пример возвращает нас к *оператору границ* (*scope operator)* (`::`, два двоеточия), показанному в предыдущих главах в связи с пространстовом имён. Здесь он используется при задании функции `set_values` для определения членов класса вне самого класса.

Заметьте, что определение функции-члена `area` было включено непосредственно внутри определения класса `Rectangle`, придавая ей крайнюю простоту. Наоборот, `set_values` только объявляется её прототипом внутри класса, тогда как определение дано вне класса. В этом внешнем определении оператор границ (`::`) используется для указания, что функция определена членом класса `Rectangle`, а не как обычная функция не член класса.

Оператор границ (`::`) задаёт класс, принадлежность к которому декларирована, гарантируя те же свойства границы, как если бы функция определялась непосредственно включением в определение класса. Например, функция `set_values` в предыдущем примере имеет доступ к переменным `width` и `height`, которые являются *private*-членами класса `Rectangle`, и сама доступна из других членов класса, таких же, как она.

Единственная разница между определением функции-члена полностью внутри определения класса и включением там её объявления, и определением позже, вне класса, в том, что в первом случае функция автоматически рассматривается компилятором как *inline* функция-член, тогда как во втором случае как обычная (не *inline*) функция-член класса. Это не сказывается на поведении, но только на возможности оптимизации компилятором.

Члены `width` и `height` имеют *private* доступ (вспомните, что если не задано обратное, все члены класса, определённые с ключевым словом `class`, имеют *private* доступ). При их объявлении *private* доступ извне класса не разрешён. Это имеет смысл, поскольку мы уже определили функцию-член для задания значений этим членам внутри объекта – функции-члена `set_values`. Таким образом, оставшаяся часть программы не нуждается в прямом доступе к ним. Возможно, в таком простом примере, как этот, трудно увидеть, чем ограниченный доступ к этим переменным может быть полезен, но в больших проектах может стать очень важным, чтобы значения не менялись непредвиденным образом (непредвиденным с точки зрения объекта).

Наиболее важное свойство класса в том, что он является типом, и, таким образом, мы можем объявить множество его объектов. Например, следуя предыдущему примеру класса `Rectangle`, мы могли бы объявить объект `rectb` в дополнение к объекту `rect`:

```
// example: one class, two objects
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area () {return
width*height;}
};

void Rectangle::set_values (int x,
int y) {
  width = x;
  height = y;
}

int main () {
  Rectangle rect, rectb;
  rect.set_values (3,4);
  rectb.set_values (5,6);
  cout << "rect area: " <<
rect.area() << endl;
  cout << "rectb area: " <<
rectb.area() << endl;
  return 0;
}
```

```
rect area: 12
rectb area: 30
```

В этом частном случае класс (тип объектов) – это `Rectangle`, представленный двумя образцами (т.е., объектами): `rect` и `rectb`. Каждый из них имеет свои собственные переменные-члены и функции-члены.

Заметьте, что вызов `rect.area()` не даёт того же результата, что вызов `rectb.area()`. Происходит это потому, что каждый объект класса `Rectangle` имеет свои собственые переменные `width` и `height`, как и они – некоторым образом – имеют также свои функции-члены `set_value` и `area`, которые оперируют с собственными переменными-членами.

Классы позволяют программировать, используя объектно-ориентированное программирование: данные и функции вместе являются членами объекта, исключая необходимость передавать и нести поддержку или другие переменные состояния в качестве аргументов для функций, поскольку они часть объекта, чьи члены вызываются. Заметьте, что не было передано аргументов при вызове ни `rect.area`, ни `rectb.area`. Эти функции-члены непосредственно использовали данные членов их соответствующих объектов `rect` и `rectb`.

Конструкторы

Что произойдёт в предыдущем примере, если мы вызовем функцию-член `area` перед вызовом `set_values`? Непредсказуемый результат, поскольку членам `width` и `height` никогда не присваивалось значение.

Чтобы избежать этого, класс может включать специалную функцию, названную *конструктор* (*constructor)*, которая автоматически вызвается, где бы ни создавался новый объект этого класса, позволяя классу инициализировать переменные-члены или выделить место для их хранения.

Эта функция *constructor* объявляется подобно обычной функции-члену, но с именем, которое совпадает с именем класса и без возвращаемого типа, и даже без `void`.

Класс `Rectangle` выше может легко быть улучшен добавлением конструктора:

```
1  // example: class constructor
2  #include <iostream>
3  using namespace std;
4
5  class Rectangle {
6      int width, height;
7    public:
8      Rectangle (int,int);
9      int area () {return
10 (width*height);}
11 };
12
13 Rectangle::Rectangle (int a, int b) {
14   width = a;
15   height = b;
16 }
17
18 int main () {
19   Rectangle rect (3,4);
20   Rectangle rectb (5,6);
21   cout << "rect area: " <<
22 rect.area() << endl;
23   cout << "rectb area: " <<
   rectb.area() << endl;
   return 0;
   }
```

```
rect area: 12
rectb area: 30
```

Результат этого примера идентичен тому, что показан в предыдущем примере. Но теперь класс `Rectangle` не имеет функции-члена `set_values`, а вместо неё имеет конструктор, который выполняет схожее действие: он инициализирует значения `width` и `height` с помощью аргументов, передаваемых ему.

Заметьте, что эти аргументы передаются конструктору в тот момент, когда создаются объекты этого класса:

```
1 Rectangle rect (3,4);
2 Rectangle rectb (5,6);
```

Конструкторы не могут вызываться явно, как если бы они были обычными функциями-членами. Они выполняются только однажды, когда создаётся объект этого класса.

Заметьте, что ни объявление прототипа конструктра (внутри класса), ни последующее определение конструктора не возвращают значения, даже `void`: конструктор никогда не возвращает значения, он просто инициализирует объект.

Перегружаемые конструкторы

Подобно другим функциям конструктор тоже может быть перегружаем разными версиями, получающими разные параметры: с разным числом параметров и/или разными типами параметров. Компилятор будет автоматически вызывать тот, чьи параметры соответствуют аргументам:

```
// overloading class constructors
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle ();
    Rectangle (int,int);
    int area (void) {return
(width*height);}
};

Rectangle::Rectangle () {
  width = 5;
  height = 5;
}

Rectangle::Rectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  Rectangle rect (3,4);
  Rectangle rectb;
  cout << "rect area: " <<
rect.area() << endl;
  cout << "rectb area: " <<
rectb.area() << endl;
  return 0;
}
```

```
rect area: 12
rectb area: 25
```

В примере выше два объекта класса `Rectangle` конструируются: `rect` и `rectb`. `rect` конструируется двумя аргументами, как и в примере ранее.

Но в этом примере также представлен специальный вид конструктора: *предопределённый конструктор*. *Предопределённый конструктор* – это конструктор, который не принимает параметры, и это сделано специально, поскольку он вызывается тогда, когда объект объявляется, а не инициализируется какими-либо аргументами. В примере выше *предопределённый конструктор* вызвается для `rectb`. Заметьте, что `rectb` даже не конструируется с пустым набором параметров. Фактически пустые скобки не могут использоваться для вызова предопределённого конструктора:

```
Rectangle rectb;   // ok, предопределённый конструктор вызвается
Rectangle rectc(); // oops, предопределённый конструктор НЕ вызывается
```

Это происходит потому, что пустой набор в скобках будет осуществлять декларацию `rectc` как функцию, вместо того, чтобы объявить объект: это будет функция, которая не принимает аргументы и возвращает значение типа `Rectangle`.

Унифицированная инициализация

Способ вызова конструкторов через заключение их аргументов в скобки, как показано выше, известен как *функциональная форма*. Но конструкторы могут таже вызываться с использованием другого синтаксиса:

Во-первых, конструкторы с единственным параметром могут вызываться, используя синтаксис инициализации переменной (знак равно с последующим аргументом):

```
class_name object_name = initialization_value;
```

Позже в C++ ввели возможность вызова конструкторов с использованием *унифицированной инициализации*, *uniform initialization*, которая, в сущности, та же самая, что и функциональная форма, но использует фигурные скобки (`{}`) вместо обычных (`()`):

```
class_name object_name { value, value, value, ... }
```

Не обязательно, но здесь можно включить знак равенства перед скобками.

Вот пример с четырьмя способами конструировать объекты класса, чей конструктор принимает единственный параметр:

```
// classes and uniform initialization
#include <iostream>
using namespace std;

class Circle {
    double radius;
  public:
    Circle(double r) { radius = r; }
    double circum() {return
2*radius*3.14159265;}
};

int main () {
  Circle foo (10.0);   // функциональная
форма
  Circle bar = 20.0;   // начальное
присвоение
  Circle baz {30.0};   // унифицированная
инициализация
  Circle qux = {40.0}; // POD-подобная

  cout << "foo's circumference: " <<
foo.circum() << '\n';
  return 0;
}
```

```
foo's circumference: 62.8319
```

Преимущество унифицированной инициализации над функциональной формой в том, что в отличие от обычных скобок, фигурные скобки нельзя перепутать с объявлением функции, следовательно, это может использоваться исключительно для вызова предопределённых конструкторов:

```
Rectangle rectb;   // вызов предопределённого конструктора
Rectangle rectc(); // объявление функции (НЕ вызов конструктора)
Rectangle rectd{}; // вызов предопределённого конструктора
```

Выбор синтаксиса для вызова конструкторов в значительной мере вопрос стиля. Большинство существующих кодов использует функциональную форму. Но некоторые современные стили проповедуют преводсходство унифицированной инициализации над остальными, хотя и в этом

есть потенциальный подвох из-за выбора `initializer_list` в качестве её типа.

Инициализация членов в конструкторах

Когда конструктор используется для инициализации других членов, эти другие члены могут быть инициализированы непосредственно, не пребегая к декларции в теле. Это выполняется вставкой пред телом конструктора двоеточия (:) и списка инициализации для членов класса. Например, рассмотрим класс со следующей декларацией:

```
class Rectangle {
    int width,height;
  public:
    Rectangle(int,int);
    int area() {return width*height;}
};
```

Конструктор этого класса будет определён, как обычно:

```
Rectangle::Rectangle (int x, int y) { width=x; height=y; }
```

Но он может быть также определён, используя *инициализацию членов*, как:

```
Rectangle::Rectangle (int x, int y) : width(x) { height=y; }
```

И даже так:

```
Rectangle::Rectangle (int x, int y) : width(x), height(y) { }
```

Заметьте, что в последнем случае конструктор не делает ничего, кроме инициализации его членов, поскольку тело функции пусто.

Для членов основных типов не делается разница, каким из способов выше конструктор определён, поскольку они не инициализируются по умолчанию, но для объектов членов (тех, чей тип – это класс), если они не инициализированы после двоеточия, они конструируются по умолчнию.

Сконструированные по умолчанию все члены класса могут или могут не всегда быть пригодны: в некоторых случаях это расточительно (когда члены впоследствии переинициализируются иначе в конструкторе), но в ряде других случаев конструирование по умолчанию даже невозможно (когда класс не имеет предопределённого конструктора). В этих случаях члены должны инициализироваться в инициализационном списке членов. Например:

```
// member initialization
#include <iostream>
using namespace std;

class Circle {
    double radius;
  public:
    Circle(double r) : radius(r) { }
    double area() {return
radius*radius*3.14159265;}
};

class Cylinder {
    Circle base;
    double height;
```

```
foo's volume: 6283.19
```

```
16   public:
17     Cylinder(double r, double h) :
18 base (r), height(h) {}
19     double volume() {return
20 base.area() * height;}
21 };
22
23 int main () {
24   Cylinder foo (10,20);
25
     cout << "foo's volume: " <<
   foo.volume() << '\n';
     return 0;
   }
```

В этом примере класс `Cylinder` имет объект-член, чей тип другого класса (тип `base` – это `Circle`). Поскольку объекты класаа `Circle` могут конструироваться только с параметром, конструктор `Cylinder` нуждается в вызове `base` конструктора, и единственый способ сделать это – в *списке инициализации членов*.

Подобная инициализация также использует синтаксис унифицированного инициализатора, используя фигурные скобки `{}` вместо обычных `()`:

```
Cylinder::Cylinder (double r, double h) : base{r}, height{h} { }
```

Указатели на классы

Объекты также могут указываться указателями: после объявления класс становится правильным типом, так что может использоваться в качестве типа, указываемого указателем. Например:

```
Rectangle * prect;
```

это указатель на объект класса `Rectangle`.

Подобно явным структурам данных члены объектов могут достигаться непосредственно через указатель, если использовать оператор стрелку (`->`). Вот пример с некоторыми возможными комбинациями:

```
1  // pointer to classes example
2  #include <iostream>
3  using namespace std;
4
5  class Rectangle {
6    int width, height;
7  public:
8    Rectangle(int x, int y) : width(x), height(y) {}
9    int area(void) { return width * height; }
10 };
11
12
13 int main() {
14   Rectangle obj (3, 4);
15   Rectangle * foo, * bar, * baz;
16   foo = &obj;
17   bar = new Rectangle (5, 6);
18   baz = new Rectangle[2] { {2,5}, {3,6} };
19   cout << "obj's area: " << obj.area() << '\n';
```

```
20    cout << "*foo's area: " << foo->area() << '\n';
21    cout << "*bar's area: " << bar->area() << '\n';
22    cout << "baz[0]'s area:" << baz[0].area() << '\n';
23    cout << "baz[1]'s area:" << baz[1].area() << '\n';
24    delete bar;
25    delete[] baz;
26    return 0;
27  }
```

Этот пример показывает использование нескольких операторов для операций над объектами и указателями (операторы `*`, `&`, `.`, `->`, `[]`). Они могут интерпретироваться так:

выражение	**можно прочитать как**
`*x`	указывается `x`
`&x`	адрес `x`
`x.y`	член `y` объекта `x`
`x->y`	член `y` объекта, указанного `x`
`(*x).y`	член `y` объекта, указанного `x` (эквивалентно предыдущему)
`x[0]`	первый объект, указываемый `x`
`x[1]`	второй объект, указываемый `x`
`x[n]`	(`n+1`)й объект, указываемый `x`

Большая часть этих выражений была представлена в предыдущих главах. Глава о массивах хорошо представляет оператор смещения (`[]`), а глава о явных стуктурах данных представляет оператор стрелку (`->`).

Классы, определённые с struct и union

Классы могут определяться не только ключевым словом `class`, но также ключевыми словами `struct` и `union`.

Ключевое слово `struct` обычно используется для объявления явной структуры данных, но может также использоваться для объявления классов, которые имеют функции-члены с тем же синтаксисом, который используется с ключевым словом `class`. Единственная разница между ними в том, что члены классов, объявленных ключевым словом `struct`, имеют `public` доступ по умолчанию, тогда как члены классов, объявленных ключевым словом `class`, имеют `private` доступ по умолчанию. Для всех других случаев оба ключевых слова в этом смысле эквивалентны.

Наоборот, концепция *union* отличается от тех классов, что объявлены с помощью `struct` и `class`, поскольку объединения (unions) хранят только один член данных единовременно, но, тем не менее, они также классы и могут, таким образом, содержать функции-члены. Предопределённый доступ в классах *union* это `public`.

Классы (II)

Перегружаемые операторы

Классы, в сущности, определяют новые типы для использования в C++ коде. А типы в C++ не только взаимодействуют с кодом посредством конструкторов и присвоения. Они также взаимодействуют через операторы. Например, возьмём следующие операции над базовыми типами:

```
int a, b, c;
a = b + c;
```

Здесь к разным переменным базового типа (`int`) поименяется оператор сложения, а затем оператор присваивания. Для основных арифметических типов значение подобных операций обычно очевидно и однозначно, но они могут не быть таковыми для некоторых типов класса. Например:

```
struct myclass {
  string product;
  float price;
} a, b, c;
a = b + c;
```

Здесь не очевидно, какой результат операции сложения `b` и `c` получится. Фактически этот изолированный код приведёт к ошибке при компиляции, поскольку тип `myclass` не имеет определённого поведения для сложения. Однако C++ позволяет большинству операторов быть перегруженными, так что их поведение может быть определено для любых типов, включая классы. Вот список всех операторов, которые могут перегружаться:

Перегружаемые операторы												
+	-	*	/	=	<	>	+=	-=	*=	/=	<<	>>
<<=	>>=	==	!=	<=	>=	++	--	%	&	^	!	|
~	&=	^=	|=	&&	||	%=	[]	()	,	->*	->	new
delete		new[]		delete[]								

Операторы перегружаются значением функции `operator`, которая является обычной функцией со специфическими именами: их имена начинаются с ключевого слова `operator` в сопровождении значка оператора, который перегружен. Синтаксис таков:

```
type operator sign (parameters) { /*... body ...*/ }
```

Например, *векторы в декартовой системе координат* – это набор из двух координат: `x` и `y`. Операция сложения двух *векторов* определена как сложение двух `x` координат вместе и двух координат `y`. Например, сложение двух *векторов* `(3,1)` и `(1,2)` даст результат `(3+1,1+2) = (4,3)`. Это должно реализоваться в C++ следующим кодом:

```
// overloading operators example
#include <iostream>
using namespace std;

class CVector {
  public:
    int x,y;
    CVector () {};
    CVector (int a,int b) : x(a),
y(b) {}
```

```
4,3
```

```
    CVector operator + (const
CVector&);
};

CVector CVector::operator+ (const
CVector& param) {
  CVector temp;
  temp.x = x + param.x;
  temp.y = y + param.y;
  return temp;
}

int main () {
  CVector foo (3,1);
  CVector bar (1,2);
  CVector result;
  result = foo + bar;
  cout << result.x << ',' << result.y
<< '\n';
  return 0;
}
```

Если вас смущает столь обильное появление `CVector`, вспомните, что некоторые из них относятся к имени класса (т.е., к типу) `CVector`, а другие являются функциями с этим именем (то есть, конструкторы, которые должны иметь то же имя, что и класс). Например:

```
CVector (int, int) : x(a), y(b) {}  // имя функции CVector (конструктор)
CVector operator+ (const CVector&); // функция, которая возвращает CVector
```

Функция `operator+` класса `CVector` перегружает оператор сложения (`+`) этим типом. Когда она объявлена, эта функция может быть вызвана либо неявно, используя оператор, либо явно, используя её функциональное имя:

```
c = a + b;
c = a.operator+ (b);
```

Оба выражения эквивалентны.

Перегруженный оператор – это только обычная функция, которая может иметь любое поведение. На самом деле нет требований, чтобы операция, выполняемая этими перегруженными носильщиками, относилась к математике или обычному смыслу оператора, хотя это настоятельно рекомендуется. Например, класс, который перегружает `operator+` для вычитания, или тот, что перегружает оператор `operator==` для заполнения объекта нулями, полностью правильны, но использование такого класса может оказаться сложным.

Ожидаемый параметр для функции-члена перегруженной операциями как `operator+` – это, естественно, операнд справа от оператора. Это свойственно всем двоичным операторам (и с операндом слева, и с операндом справа). Но операторы могут появиться в иных формах. Вот сводная таблица с параметрами, требуемыми каждому из разных операторов, которые могут быть перегружены (пожалуйста, замените `@` оператором в каждом случае):

Выражение	Оператор	Функция-член	Не функция-член
`@a`	`+ - * & ! ~ ++ --`	`A::operator@()`	`operator@(A)`
`a@`	`++ --`	`A::operator@(int)`	`operator@(A,int)`
`a@b`	`+ - * / % ^ & | < > == != <= >= << >> && || ,`	`A::operator@(B)`	`operator@(A,B)`

a@b	= += -= *= /= %= ^= &= |= <<= >>= []	A::operator@(B)	-
a(b,c...)	()	A::operator()(B,C...)	-
a->b	->	A::operator->()	-
(TYPE) a	TYPE	A::operator TYPE()	-

Где a – это объект класса A, b – объект класса B, а c является объектом класса C. TYPE – это любой тип (перегруженные операторы преобразуют в тип TYPE).

Заметьте, что некоторые операторы могут быть перегружены в двух формах: либо как функция-член, либо как не функция-член. Первый случай был использован в примере выше для operator+. Но некоторые операторы могут таже быть перегружены как не функции-члены. В этом случае оператор функции принимает объект подходящего класса в качестве первого аргумента.

Например:

```
1  // non-member operator overloads
2  #include <iostream>
3  using namespace std;
4
5  class CVector {
6    public:
7      int x,y;
8      CVector () {}
9      CVector (int a, int b) : x(a),
10 y(b) {}
11 };
12
13
14 CVector operator+ (const CVector&
15 lhs, const CVector& rhs) {
16   CVector temp;
17   temp.x = lhs.x + rhs.x;
18   temp.y = lhs.y + rhs.y;
19   return temp;
20 }
21
22 int main () {
23   CVector foo (3,1);
24   CVector bar (1,2);
25   CVector result;
26   result = foo + bar;
27   cout << result.x << ',' << result.y
   << '\n';
   return 0;
   }
```

```
4,3
```

Ключевое слово this

Ключевое слово this представляет указатель на объект, чья функция-член выполняется. Это используется в функциях-членах класса для ссылки на сам объект.

Одно из использований может быть проверено, если параметр, передаваемый в функцию-член, есть сам объект. Например:

```
1  // example on this
2  #include <iostream>
3  using namespace std;
4
```

```
yes, &a is b
```

```
class Dummy {
  public:
    bool isitme (Dummy& param);
};

bool Dummy::isitme (Dummy& param)
{
  if (&param == this) return true;
  else return false;
}

int main () {
  Dummy a;
  Dummy* b = &a;
  if ( b->isitme(a) )
    cout << "yes, &a is b\n";
  return 0;
}
```

Есть также используемая в `operator=` функции-члене, которая возвращает объекты по ссылке. Следуя примерам *векторов в декартовой системе координат*, показанным выше, их функция `operator=` могла быть определена как:

```
CVector& CVector::operator= (const CVector& param)
{
  x=param.x;
  y=param.y;
  return *this;
}
```

Фактически, эта функция очень похожа на код, который компилятор генерирует неявно для класса при `operator=`.

Статические члены

Класс может содержать статические члены либо данных, либо функций.

Статические члены данных класса также известны, как «переменные класса», поскольку есть только одна общая переменная для всех объектов этого самого класса, совместно использующих эту переменную: т.е., значение не отличается у одного объекта класса от другого.

Например, это может быть использовано для переменной в классе, которая может содержать счётчик с числом объектов этого класса, которые находятся как в следующем примере:

```
// static members in classes
#include <iostream>
using namespace std;

class Dummy {
  public:
    static int n;
    Dummy () { n++; };
    ~Dummy () { n--; };
};

int Dummy::n=0;

int main () {
```

```
7
6
```

```
  Dummy a;
  Dummy b[5];
  Dummy * c = new Dummy;
  cout << a.n << '\n';
  delete c;
  cout << Dummy::n << '\n';
  return 0;
}
```

Фактически статические члены имеют те же свойства, что и переменные не члены, но они находятся в границах класса. Из этих соображений, и для того чтобы предохранить их от объявления несколько раз, они не могут быть инициализированы непосредственно в классе, но нуждаются в инициализации где-нибудь вне его. Как в предыдущем примере:

```
int Dummy::n=0;
```

Поскольку это значение общей переменной для всех объектов того же класса, на него можно ссылаться, как на члена любого объекта этого класса или даже непосредственно по имени класса (конечно, это правильно только для статических членов):

```
cout << a.n;
cout << Dummy::n;
```

Эти два вызова выше ссылаются на ту же переменную: статическая переменная `n` в классе `Dummy` обобщается для всех объектов этого класса.

И вновь, это похоже на переменные не члены, но с именем, которое требуется для доступа подобно членам класса (или объектам).

Классы могут иметь и статические функции-члены. Это представляет одно и то же: члены класса, которые являются общими для всех объектов этого класса, действуя так же, как не функции-члены, но доступные подобно членам класса. Поскольку они похожи на не функции-члены, они не доступны не статическим членам класса (ни членам-переменным, ни членам-функциям). И они не могут использовать ключевое слово `this`.

Const функции-члены

Когда объект класса квалифицируется как `const` объект:

```
const MyClass myobject;
```

доступ к его данным-членам извне класса ограничен только чтением, как если бы все его данные-члены были `const` для доступа к ним извне класса. Заметьте, что конструктор всё ещё вызвается и позволяет инициализировать и модифицировать эти данные члены:

```
// constructor on const object
#include <iostream>
using namespace std;

class MyClass {
  public:
    int x;
    MyClass(int val) : x(val) {}
    int get() {return x;}
};
```

```
10
```

```
11
12 int main() {
13   const MyClass foo(10);
14 // foo.x = 20;            // неверно:
15 x нельзя модифицировать
16   cout << foo.x << '\n';  // ok:
17 член-данные x может читаться
   return 0;
}
```

Функции-члены `const` объектов могут только вызываться, если они сами заданы как `const` члены; в примере выше член `get` (который не задан как `const`) не может быть вызван из `foo`. Чтобы задать, что член есть `const` член, ключевое слово `const` должно следовать за прототипом функции после скобок для её параметров:

```
int get() const {return x;}
```

Заметьте, что `const` может использоваться для квалификации типа, возвращаемого функцией-членом. Это `const` не то же самое, что задаёт член как `const`. Оба независимы и расположены в разных местах прототипа функции:

```
1 int get() const {return x;}         // const функция-член
2 const int& get() {return x;}        // функция-член возвращает const&
3 const int& get() const {return x;} // const функция-член, возвращающая
  const&
```

Функция-член, заданная в качестве `const`, не может изменять не статические данные-члены, и не может вызвать другие не-`const` функции-члены. В основном `const` члены не будут модифицировать соостояние объекта.

`const` объекты ограничены в доступе только членов, помеченных как `const`, но не-`const` объекты не ограничены, и, следовательно, им могут быть доступны и `const`, и не-`const` члены одинаково.

Вы можете подумать, что как бы вы изредка не собирались объявить `const` объекты, и пометить все члены не модифицируемыми объектами как *const*, это не стоит усилий, но *const* объекты теперь обобщены. Большинство функций принимает классы как параметры, реально берёт их как ссылки на `const`, и, таким образом, функции могут только иметь доступ к их `const` членам:

```
1  // const objects
2  #include <iostream>
3  using namespace std;
4
5  class MyClass {
6      int x;
7    public:
8      MyClass(int val) : x(val) {}
9      const int& get() const {return
10 x;}
11 };
12
13 void print (const MyClass& arg) {
14   cout << arg.get() << '\n';
15 }
16
17 int main() {
18   MyClass foo (10);
```

```
10
```

```
  print(foo);

  return 0;
}
```

Если в этом примере `get` не была задана как `const` член, вызов `arg.get()` в функции `print` не будет возможен, поскольку `const` объекты имеют доступ только к `const` функциям-членам.

Функции-члены могут быть перегружены их постоянством: т.е., класс может иметь две функции-члена с одинаковыми сигнатурами, исключая, что одна – это `const`, а другая нет: в этом случае `const` версия вызвается только тогда, когда объект сам есть `const`, а не-`const` версия вызвается тогда, когда объект сам есть не-`const`.

```
// overloading members on constness
#include <iostream>
using namespace std;

class MyClass {
    int x;
  public:
    MyClass(int val) : x(val) {}
    const int& get() const {return
x;}
    int& get() {return x;}
};

int main() {
  MyClass foo (10);
  const MyClass bar (20);
  foo.get() = 15;         // ok:
get() returns int&
// bar.get() = 25;        // not
valid: get() returns const int&
  cout << foo.get() << '\n';
  cout << bar.get() << '\n';

  return 0;
}
```

```
15
20
```

Класс шаблонов (templates)

Аналогично тому, как мы можем создать шаблоны функций, мы можем создать шаблоны классов, позволяя классам иметь члены, которые используют парметры шаблонов как типов. Например:

```
template <class T>
class mypair {
    T values [2];
  public:
    mypair (T first, T second)
    {
      values[0]=first; values[1]=second;
    }
};
```

Класс, который мы только что определили, служит для хранения двух элементов любого правильного типа. Например, если мы хотим объявить объект этого класса для хранения двух целых значений типа `int` со значениями 115 и 36 мы должны написать:

```
mypair<int> myobject (115, 36);
```

Тот же класс может быть использован для создания объектов, хранящих любые другие типы, как:

```
mypair<double> myfloats (3.0, 2.18);
```

Конструктор – это только функция-член в предыдущем шаблоне класса, и он был определён *inline* в определении самого класса. В случае, когда функция-член определяется вне определения шаблона класса, это будет предварено `template <...>` префиксом:

```
// class templates
#include <iostream>
using namespace std;

template <class T>
class mypair {
    T a, b;
  public:
    mypair (T first, T second)
      {a=first; b=second;}
    T getmax ();
};

template <class T>
T mypair<T>::getmax ()
{
  T retval;
  retval = a>b? a : b;
  return retval;
}

int main () {
  mypair <int> myobject (100, 75);
  cout << myobject.getmax();
  return 0;
}
```

```
100
```

Заметьте, синтаксис определения функции-члена `getmax`:

```
template <class T>
T mypair<T>::getmax ()
```

Запутались множеством `T`? Есть три `T` в этом объявлении: первый – это параметр шаблона. Второй `T` относится к типу, возвращаемому функцией. А третий `T` (тот, что между угловыми скобками) – также требование: устанавливает, что этот параметр шаблона функции также параметр шаблона класса.

Специализация шаблона

Есть возможность определить разные реализации шаблона, когда задаваемый тип передаётся как аргумент шаблона. Это называется *специализацией шаблона*.

Например, давайте представим, что у нас есть очень простой класс, который назван `mycontainer`. Пусть класс хранит один элемент любого типа и имеет только одну функцию-член, названную `increase`, которая, в свою очередь, увеличивает своё значение. Но мы обнаружим,

что когда она хранит один элемент типа `char`, было бы гораздо удобнее иметь совсем другую реализацию с функцией-членом `uppercase`, так что мы решаем объявить шаблон класса со специализацей для этого типа:

```
// template specialization
#include <iostream>
using namespace std;

// class template:
template <class T>
class mycontainer {
    T element;
  public:
    mycontainer (T arg)
{element=arg;}
    T increase () {return ++element;}
};

// class template specialization:
template <>
class mycontainer <char> {
    char element;
  public:
    mycontainer (char arg)
{element=arg;}
    char uppercase ()
    {
      if
((element>='a')&&(element<='z'))
      element+='A'-'a';
      return element;
    }
};

int main () {
  mycontainer<int> myint (7);
  mycontainer<char> mychar ('j');
  cout << myint.increase() << endl;
  cout << mychar.uppercase() << endl;
  return 0;
}
```

```
8
J
```

Вот синтаксис, используемый для специализации шаблона класса:

```
template <> class mycontainer <char> { ... };
```

Прежде всего, обратите внимание, что мы предваряем имя класса ключевым словом `template<>`, включающим пустой список параметров. Это сделано потому, что все типы известны, и не требуется аргументов шаблона для этой спецификации. Но пока это всё ещё специализация шаблона класса, и по этой причине требуется подобная отметка.

Но более важным становится префикс `<char>`, специализирующий параметр, который следует за именем шаблона класса. Этот параметр специализации сам идентифицирует тип, для которого шаблон класса специализируется (`char`). Заметьте разницу между оригинальным шаблоном класса и специализированным:

```
template <class T> class mycontainer { ... };
template <> class mycontainer <char> { ... };
```

Первая строка – это оригинальный шаблон, а вторая строка - специализированный.

Когда мы объявляем специализацию для шаблона класса, мы должны только определить его члены, даже если это идентично с оригинальным шаблоном класса, поскольку не происходит «наследования» членов от оригинального шаблона.

Специальные члены

[ВНИМАНИЕ: эта глава требует ясного понимания *динамически выделяемой памяти*]

Специальные функции-члены – это функции-члены, которые неявно определены как члены классов при определённых условиях. Их шесть:

функции-члены	типичная форма для класса C:
Педопределённый конструктор	`C::C();`
Деструктор	`C::~C();`
Конструктор копии	`C::C (const C&);`
Присвоение копии	`C& operator= (const C&);`
Конструктор перемещения	`C::C (C&&);`
Присвоение перемещения	`C& operator= (C&&);`

Давайте проверим каждую из них:

Предопределённый конструктор

Предопределённый конструктор – это конструктор, вызваемый тогда, когда объекты класса объявляются, но не инициализруются какими-нибудь аргументами.

Если определение класса не имеет конструкторов, компилятор определяет для класса неявно заданный *предопределённый конструктор*. Таким образом, декларируется класс подобный этому:

```
1 class Example {
2   public:
3     int total;
4     void accumulate (int x) { total += x; }
5 };
```

Компилятор решил, что `Example` имеет *преопределённый конструктор*. В итоге объекты этого класса могут конструироваться простым объявлением их без каких-либо аргументов:

```
Example ex;
```

Но если класс имеет некоторый конструктор, принимающий какое-то число явно декларированных параметров, компилятор не предоставляет больше *предопределённый конструктор*, и не позволяет больше объявление новых объектов этого класса без аргументов. Например, следующий класс:

```
1 class Example2 {
2   public:
3     int total;
4     Example2 (int initial_value) : total(initial_value) { };
5     void accumulate (int x) { total += x; };
6 };
```

Здесь мы имеем объявленный конструктор с параметром типа `int`. Поэтому следующее объявление объекта будет корректно:

```
Example2 ex (100);   // ok: вызываем конструктор
```

Но следующее:

```
Example2 ex;          // неверно: нет предопределённого конструктора
```

будет неверно, поскольку класс был объявлен с явным конструктором, принимающим один аргумент, а замещающий неявный *предопределённый конструктор* ничего не принимает.

Таким образом, если объекты этого класса нуждаются в конструировании без аргументов, подходящий *предопределённый конструктор* тоже должен быть объявлен в классе. Например:

```
1  // classes and default constructors
2  #include <iostream>
3  #include <string>
4  using namespace std;
5
6  class Example3 {
7      string data;
8    public:
9      Example3 (const string& str) :
10 data(str) {}
11     Example3() {}
12     const string& content() const
13 {return data;}
14 };
15
16 int main () {
17   Example3 foo;
18   Example3 bar ("Example");
19
20   cout << "bar's content: " <<
   bar.content() << '\n';
     return 0;
   }
```

```
bar's content: Example
```

Здесь `Example3` имеет *предопределённый конструктор* (т.е., конструктором без параметров), определённый как пустой блок:

```
Example3() {};
```

Это позволяет объектам класса `Example3` конструироваться без аргументов (похожим образом `foo` было объявлено в этом примере). Обычно *предопределённый конструктор*, подобный этому, неявно определён для всех классов, которые не имеют иных конструкторов, и, значит, не требуют явного определения. Но в этом случае `Example3` имеет другой конструктор:

```
Example3 (const string& str);
```

А когда любой конструктор явно объявлен в классе, неявный *предопределённый конструктор* автоматически не поддерживается.

Деструктор

Деструкторы выполняют функцию противоположную *конструкторам*: они ответственны за необходимую очистку, в которой класс нуждается, когда его время жизни истекает. Классы, которые мы определяли в предыдущих главах, не занимали каких-либо ресурсов, а, значит, не требовали реальной очистки.

Но теперь давайте представим, что для класса из последнего примера выделяется динамическая память для хранения строки, которую он имеет как член данных. В этом случае было бы очень полезно иметь функцию, вызваемую автоматически по завершению существования объекта, чтобы освободить память, занимаемую им. Чтобы это сделать, мы используем *деструктор*. Деструктор – это функция-член очень похожая на *предопределённый конструктор*: она не принимает аргументы и ничего не возвращает, даже `void`. Она также использует имя класса, как своё собственное имя, но предваряет его значком тильды (`~`):

```
// destructors
#include <iostream>
#include <string>
using namespace std;

class Example4 {
    string* ptr;
  public:
    // constructors:
    Example4() : ptr(new string) {}
    Example4 (const string& str) :
ptr(new string(str)) {}
    // destructor:
    ~Example4 () {delete ptr;}
    // access content:
    const string& content() const
{return *ptr;}
};

int main () {
  Example4 foo;
  Example4 bar ("Example");

  cout << "bar's content: " <<
bar.content() << '\n';
  return 0;
}
```

```
bar's content: Example
```

При конструировании `Example4` выделяется место хранения для `string`. Хранилище, которое освобождается впоследствии с помощью деструктора.

Деструктор объекта вызвается в конце его цикла существования. В данном случае `foo` и `bar` выполнили это в конце функции `main`.

Конструктор копии

Когда объекту передаётся именованный объект его собственного типа в качестве аргумента, вызывается его *конструктор копии*, чтобы сконструировать копию.

Конструктор копии – это конструктор, у которого первый парметр имеет тип *ссылки на сам класс* (возможно, квалифицированный как `const`) и который может быть вызыван с единственным аргументом этого типа. Например, для класса `MyClass` *конструктор копии* имеет следующую сигнатуру:

```
MyClass::MyClass (const MyClass&);
```

Если класс не имеет индивидуальных определений ни *копии*, ни *перемещения* конструктора (ни присвоения), поддерживается неявный *конструктор копии*. Этот конструктор копии просто выполняет копирование его собственных членов. Например, как для класса:

```
class MyClass {
  public:
    int a, b; string c;
};
```

Неявный *конструктор копии* определяется автоматически. Определение предполагает для этой функции выполнение *малой копии* приблизительно эквивалентной:

```
MyClass::MyClass(const MyClass& x) : a(x.a), b(x.b), c(x.c) {}
```

Этот предопределённый *конструктор копии* может подойти для нужд многих классов. Но *малые копии* – это только копии членов класса, а это, возможно, не то, что мы ожидали бы для классов подобных классу `Example4`, который мы определили выше, поскольку он содержит указатели, поддерживающие его хранилища. Для этого класса выполнение *малой копии* означает, что копируется значение указателя, но не само содержимое. Это означает, что оба объекта (копия и оригинал) будут совместно использовать единственный `string` объект (они оба будут указывать на тот же объект), и в каком-то месте (при деструкции) оба объкта попытаются очистить тот же блок памяти, что может привести к отказу программы при работе. Это можно исправить определением следующего специализированного *конструктора копии*, который выполняет *глубокую копию*:

```
// copy constructor: deep copy
#include <iostream>
#include <string>
using namespace std;

class Example5 {
    string* ptr;
  public:
    Example5 (const string& str) :
ptr(new string(str)) {}
    ~Example5 () {delete ptr;}
    // copy constructor:
    Example5 (const Example5& x) :
ptr(new string(x.content())) {}
    // access content:
    const string& content() const
{return *ptr;}
};

int main () {
  Example5 foo ("Example");
  Example5 bar = foo;

  cout << "bar's content: " <<
bar.content() << '\n';
  return 0;
}
```

```
bar's content: Example
```

Глубокая копия, выполняемая этим *конструктором копии*, резервирует хранилище для новой строки, которая инициализрована для содержания копии оригинального объекта. Так оба объекта (копия и оригинал) имеют индивидуальные копии содержимого, которое хранится в разных местах.

Присвоение копии

Объекты не только копируются конструктором при инициализации, они также копируются при любой операции присвоения. Посмотрите разницу:

```
MyClass foo;
MyClass bar (foo);     // инициализация объекта: вызвается конструктор копии
MyClass baz = foo;     // инициализация объекта: вызвается конструктор копии
foo = bar;             // объект уже инициализирован: вызывается присвоение
копии
```

Заметьте, что `baz` инициализируется при конструировании, используя *знак равенства*, но это не операция присвоения! (хотя очень на неё похожа). Объявление объекта не является операцией присваивания, это только другой синтаксис для вызова одноаргументных конструкторов.

Присвоение `foo` является операцией присваивания. Ни один объект здесь не объявляется, но операция выполняется с существующим объектом `foo`.

Оператор присваивания копии – это перегрузка `operator=`, который принимает *значение ссылки* самого класса как параметра. Возвращаемое значение обычно - это ссылка на `*this` (хотя это не требуется). Например, для класса `MyClass`, *присваивание копии* может иметь следующую сигнатуру:

```
MyClass& operator= (const MyClass&);
```

Оператор присваивания копии является также *специальной функцией* и также неявно определяется, если класс не имеет ни индивидуальной *копии*, ни определения присваивания *перемещения* (ни объявления конструктора перемещения).

Но вновь, *неявная* версия выполняет *малую копию*, которая подходит для многих классов, но не для классов с указателями на объекты, хранение которых они поддерживают, как в случае `Example5`. В этом случае не только класс подвергается риску удаления указываемого объекта дважды, но присваивание создаёт утечку памяти, не удаляя объект указываемый объектом до присваивания. Эта проблема может быть решена с помощью *присваивания копии*, которое удаляет предыдущий объект и выполняет *глубокое копирование*:

```
Example5& operator= (const Example5& x) {
  delete ptr;                      // удаляет текущую указываемую строку
  ptr = new string (x.content());  // резервирует пространство для новой
строки и копирует
  return *this;
}
```

Или даже лучше, поскольку член `string` не постоянная, возможно повторно использовать тот же объект `string`:

```
Example5& operator= (const Example5& x) {
  *ptr = x.content();
  return *this;
}
```

Конструктор перемещения и присваивания

Подобно копированию перемещение также использует значение объекта, чтобы задать значение другого объекта. Но не так, как при копировании, содержимое реально переносится от одного объекта (источника) к другому (адресату): источник теряет это содержимое, которое захватывается адресатом. Это перемещение происходит только тогда, когда источник значения является *неименованным объектом*.

Неименованные объекты – это объекты, которые носят временный характер, так что им даже не дают имени. Типичными примерами *объектов без имени* являются возвращаемые значения функций или приведение типов.

Используя значение временного объекта, такого как те, что инициализируют другой объект или присваивают значение, можно обойтись без реального копирования. Объект никогда не собираются использовать для чего-либо ещё, следовательно, его значение может *перемещаться в* объект назначения. В этих случаях запускается *конструктор перемещения* или *присваивание перемещения*.

Конструктор перемещения вызывается, когда объект инициализируется конструктором, используя безымянных «временщиков». Таким же образом *присваивание перемещения* вызывается, когда объекту присваивается значение временного объекта без имени:

```
MyClass fn();             // функция, возвращающая объект MyClass
MyClass foo;              // предопределённый конструктор
MyClass bar = foo;        // конструктор копии
MyClass baz = fn();       // конструктор перемещения
foo = bar;                // присваивание копии
baz = MyClass();          // присваивание перемещения
```

Оба значения, возвращаемые `fn`, и значение, сконструированное `MyClass`, временные и без имени. В этих случаях нет нужды делать копии, поскольку безымянные объекты очень недолговечны и могут быть окупированы другими объектами, когда операция более эффективна.

Конструктор перемещения и *присвоение перемещения* – это члены, которые принимают параметр типа *r-значения ссылки на класс* (*rvalue reference to the class)*, собственно:

```
MyClass (MyClass&&);             // конструктор перемещения
MyClass& operator= (MyClass&&);  // присваивание перемещения
```

Ссылка r-значения задаётся следующими за типом двумя амперсандами (`&&`). В качестве параметра *ссылка r-значения* совпадает с аргументами «временных» этого типа.

Концепция перемещения наиболее полезна для объектов, которые обслуживают хранение того, что используют; таких как объекты, которым выделяется место с new и delete. Для таких объектов *копирование* и *перемещение* реально разные операции:
- Копирование из A в B означает, что выделяется новое место в памяти для B, а затем всё содержимое A копируется в новое место, отведённое для B.
- Перемещение из A в B означает, что память, уже отведённая для A, передаётся B без выделения какого-либо нового хранилища. Это просто копирование указателя.

Например:

```
// move constructor/assignment
#include <iostream>
#include <string>
using namespace std;

class Example6 {
    string* ptr;
  public:
    Example6 (const string& str) :
ptr(new string(str)) {}
    ~Example6 () {delete ptr;}
    // move constructor
    Example6 (Example6&& x) :
ptr(x.ptr) {x.ptr=nullptr;}
    // move assignment
```

```
foo's content: Example
```

```
16      Example6& operator= (Example6&&
17  x) {
18        delete ptr;
19        ptr = x.ptr;
20        x.ptr=nullptr;
21        return *this;
22      }
23      // access content:
24      const string& content() const
25  {return *ptr;}
26      // addition:
27      Example6 operator+(const
28  Example6& rhs) {
29        return
30  Example6(content()+rhs.content());
31      }
32  };
33
34
35  int main () {
36    Example6 foo ("Exam");
37    Example6 bar = Example6("ple");
    // move-construction

    foo = foo + bar;
    // move-assignment

    cout << "foo's content: " <<
    foo.content() << '\n';
    return 0;
    }
```

Компиляторы уже оптимизируют многочисленные случаи, когда формально требуется вызов конструктора перемещения, в то, что известно как *оптимизация возвращаемого значения* (*Return Value Optimization)*. Более заметно, когда значение, возвращаемое функцией, используется для инициализации объекта. В таких случаях *конструктор перемещения* может на самом деле никогда не вызваться.

Заметьте, что даже при том, что ссылки r-значения могут использоваться для типа любого параметра функции, изредка полезно использовать что-то другое, чем *конструктор перемещения*. Ссылки r-значения мудрёны, и их использование без особой необходимости может стать источником ошибок, которые крайне трудно отследить.

Скрытые члены

Шесть специальных функций-членов, описанных выше, это члены, неявно декларированные в классе при некоторых условиях:

Функция-член	неявно определены:	предопределённое определение:
Предопределённый конструктор	если нет других конструкторов	ничего не делает
Деструктор	если нет деструктора	ничего не делает
Конструктор копии	если нет конструктора перемещения и нет присваивания перемещения	копирует все члены
Присваивание копии	если нет конструктора перемещения и нет присваивания перемещения	копирует все члены

Конструктор перемещения	если нет деструктора, нет конструктора копии и нет ни присваивания копии, ни перемещения	перемещает все члены
Присваивание перемещения	если нет деструктора, нет конструктора копии и нет ни присваивания копии, ни перемещения	перемещает все члены

Заметьте, что не все *специальные функции-члены* неявно определены в тех же случаях. Это, в основном, связано с обратной совместимостью со структурами Си и ранними версиями C++, и, фактически, включает некоторые устаревшие случаи. К счастью, каждый класс может выбрать явно, какие из этих членов будут содержаться с их предопределением, а какие удалены с помощью ключевых слов `default` и `delete` соответственно. Синтакис один из:

```
function_declaration = default;
function_declaration = delete;
```

Например:

```
1  // default and delete implicit
2  members
3  #include <iostream>
4  using namespace std;
5
6  class Rectangle {
7      int width, height;
8    public:
9      Rectangle (int x, int y) :
10 width(x), height(y) {}
11     Rectangle() = default;
12     Rectangle (const Rectangle&
13 other) = delete;
14     int area() {return width*height;}
15 };
16
17 int main () {
18   Rectangle foo;
19   Rectangle bar (10,20);
20
     cout << "bar's area: " <<
   bar.area() << '\n';
     return 0;
   }
```

```
bar's area: 200
```

Здесь `Rectangle` может конструироваться либо с двумя `int` аргументами, либо быть *предопределённо сконструирован* (без аргументов). Однако он не может быть *сконструирован копированием* из другого `Rectangle` объекта, поскольку эта функция была удалена. Таким образом, принимая объекты последнего примера, следующее выражение не будет правильно:

```
Rectangle baz (foo);
```

Однако выражение может быть сделано совершенно правильным, если определить его конструктор копии как:

```
Rectangle::Rectangle (const Rectangle& other) = default;
```

Что будет, в сущности, эквивалентно:

```
Rectangle::Rectangle (const Rectangle& other) : width(other.width),
height(other.height) {}
```

Заметьте, что ключевое слово `default` не определяет функцию-член эквивалентную *предопределённому конструктору* (т.е., где *предопределённый конструктор* означает конструктор без параметров), но эквивалентную конструктору, который будет определён неявно, если не будет удалён.

В основном – и для будущей совместимости – классам, которые явно определяют один конструктор копии/перемещения или одно присваивание копии/перемещения, но не оба, рекомендовано задавать либо `delete`, либо `default` для других специальных функций-членов, которые они явно не определяют.

Дружба и наследование

Функции-друзья (Friend)

В принципе, *private* и *protected* члены класса не могут быть доступны извне того класса, в котором они объявлены. Однако это правило не применимо к «*друзьям*».

Друзья – это функции класса, объявленные с ключевым словом `friend`.

Функции не члены могут иметь доступ к *private* и *protected* членам класса, если они объявлены *друзьями* класса. Это делается включением декларации этих внешних функций внутри класса с предшествующим ключевым словом `friend`:

```
// friend functions
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle() {}
    Rectangle (int x, int y) : width(x), height(y)
{}
    int area() {return width * height;}
    friend Rectangle duplicate (const Rectangle&);
};

Rectangle duplicate (const Rectangle& param)
{
  Rectangle res;
  res.width = param.width*2;
  res.height = param.height*2;
  return res;
}

int main () {
  Rectangle foo;
  Rectangle bar (2,3);
  foo = duplicate (bar);
  cout << foo.area() << '\n';
  return 0;
}
```

```
24
```

Функция `duplicate` – это *друг* класса `Rectangle`. Значит, функция `duplicate` способна получить доступ к членам `width` и `height` (которые *private*), разным объектам типа `Rectangle`. Заметьте, хотя это не продекларировано ни в `duplicate`, ни в её последующем использовании в `main`, функция-член `duplicate` не рассматривается членом класса `Rectangle`. Нет! Она просто имеет доступ к её *private* и *protected* членам, не будучи членом класса.

Типичное использование фукнций друзей – это операции, которые сопутствуют двум разным классам, при доступе к *private* или *protected* членам обоих.

Классы-друзья

Подобно функциям-друзьям классы-друзья – это классы, чьи члены имеют доступ к *private* или *protected* членам другого класса:

```
// friend class
```

```
16
```

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

class Square;

class Rectangle {
    int width, height;
  public:
    int area ()
      {return (width * height);}
    void convert (Square a);
};

class Square {
  friend class Rectangle;
  private:
    int side;
  public:
    Square (int a) : side(a) {}
};

void Rectangle::convert (Square a) {
  width = a.side;
  height = a.side;
}

int main () {
  Rectangle rect;
  Square sqr (4);
  rect.convert(sqr);
  cout << rect.area();
  return 0;
}
```

В этом примере класс `Rectangle` является другом класса `Square`, позволяя функциям-членам `Rectangle` иметь доступ к *private* и *protected* членам `Square`. Более конкретно, `Rectangle` обращается к переменной-члену `Square::side`, которая описывает сторону квадрата.

Есть и нечто новое в этом примере: в начале программы есть пустая декларация класса `Square`. Это необходимо, поскольку класс `Rectangle` использует `Square` (как параметр в члене `convert`), а `Square` использует `Rectangle` (объявив его другом).

Дружба никогда не передаётся, пока не задана: в нашем примере `Rectangle` рассматривается `Square` классом-другом, но `Square` не рассматривается другом со стороны `Rectangle`. Таким образом, функция-член `Rectangle` может иметь доступ к *protected* и *private* членам `Square`, но не наоборот. Конечно, `Square` может также быть объявлен другом `Rectangle`, если нужно, гарантируя такой доступ.

Другое свойство дружбы – это то, что она не переходит: друг друга не считается другом, пока это не задано явно.

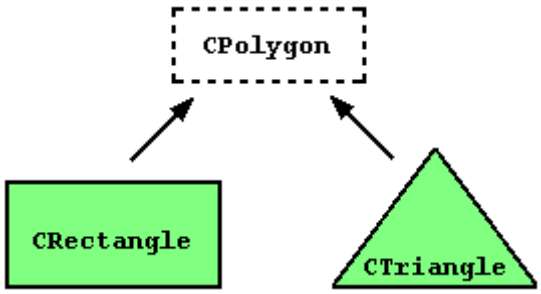
Наследование между классами

Классы в C++ могут быть расширены созданием новых классов, которые сохраняют характеристики базового класса. Этот процесс, известный как наследование, затрагивает *базовый класс* и *производный класс*: *производный класс* наследует членов *базового класса*, поверх которого он может добавить его собственных членов.

Например, давайте представим последовательность классов, чтобы описать две разновидности многоугольников: прямоугольника и треугольника. Эти два многоугольника имеют некоторые общие свойства, такие как нужные для расчёта их площади значения: они оба могут быть

описаны просто высотой и шириной (или основанием).

Это может быть представлено в терминах классов с классом `Polygon`, из которого мы произведём два других: `Rectangle` и `Triangle`:



Класс `Polygon` будет содержать члены, которые являются общими для обоих типов многоугольников. В нашем случае: `width` и `height`. И `Rectangle`, и `Triangle` будут его производными классами с особенностями, которые отличаются в одном типе многоугольника от другого.

Классы, которые являются производными от других, наследуют все доступные члены базового класса. Это означает, что если базовый класс включает член `A`, мы произведём от него класс с другим членом, названным `B`, производный класс будет содежать оба члена: и `A`, и `B`.

Взаимосвязь наследования двух классов объявляется в производном классе. Определение производных классов использует следующий синтаксис:

```
class derived_class_name: public base_class_name
{ /*...*/ };
```

Где `derived_class_name` – это имя производного класса, а `base_class_name` – это имя класса, на котором он базируется. Спецификатор доступа `public` может быть замещён любым другим спецификатором доступа (`protected` или `private`). Этот спецификатор доступа ограничивает уровень доступа к членам, наследуемым из базового класса. Члены с максимальным уровнем доступа унаследованы с этим уровнем, тогда как члены с равным или более строгим уровнем доступа сохраняют их ограниченный уровень в производном классе.

```
1  // derived classes
2  #include <iostream>
3  using namespace std;
4
5  class Polygon {
6    protected:
7      int width, height;
8    public:
9      void set_values (int a, int b)
10       { width=a; height=b;}
11   };
12
13 class Rectangle: public Polygon {
14   public:
15     int area ()
16       { return width * height; }
17   };
18
19 class Triangle: public Polygon {
20   public:
21     int area ()
22       { return width * height / 2; }
23   };
```

```
20
10
```

```
24
25 int main () {
26   Rectangle rect;
27   Triangle trgl;
28   rect.set_values (4,5);
29   trgl.set_values (4,5);
30   cout << rect.area() << '\n';
31   cout << trgl.area() << '\n';
32   return 0;
33 }
```

Объекты классов `Rectangle` и `Triangle` каждый содержат члены, унаследованные от `Polygon`. Это: `width`, `height` и `set_values`.

Спецификатор доступа `protected`, использованный в классе `Polygon`, похож на `private`. Его единственное отличие обнаруживается в фактическом наследовании. Когда клас наследуется другим, члены производного класса могут иметь доступ к *protected* членам, унаследованным из базового класса, но не к его *private* членам.

Декларированием `width` и `height` как `protected` вместо `private` эти члены также становятся доступны из производных классов `Rectangle` и `Triangle` вместо того, чтобы иметь доступ только из членов `Polygon`. Если бы они были *public*, они были бы доступны отовсюду.

Мы можем суммировать разные типы доступа согласно тому, какие функции могут получить доступ в следующих случаях:

Доступ	public	protected	private
члены того же класса	yes	yes	yes
члены производного класса	yes	yes	no
не члены	yes	no	no

Здесь «не члены» дают любой доступ извне класса, такой как из `main`, из другого класса или из функции.

В примере выше члены, унаследованные `Rectangle` и `Triangle`, имеют тот же допуск, какой они имели в их базовом классе `Polygon`:

```
1 Polygon::width           // protected доступ
2 Rectangle::width         // protected доступ
3
4 Polygon::set_values()    // public доступ
5 Rectangle::set_values()  // public доступ
```

Это имеет место, поскольку отношение наследования было объявлено с использованием ключевого слова `public` в каждом из произоводных классов:

```
class Rectangle: public Polygon { /* ... */ }
```

Ключевое слово `public` после двоеточия (`:`) показывает наиболее доступный уровень членов, наследуемых из класса, который следует за двоеточием (в нашем случае `Polygon`), который будет из производного класса (в нашем случае `Rectangle`). Поскольку `public` уровень наибольшего уровня доступа, заданием этого ключевого слова производный класс будет наследовать все члены с тем же уровнем, который они имели в базовом классе.

С `protected` все `public` члены базового класса наследуются как `protected` в производном классе. И наоборот, если задан наиболее закрытый уровень доступа (`private`), все члены

базового класса наследуются как `private`.

Например, если дочерним был класс, произведённый от материнского, который мы определили так:

```
class Daughter: protected Mother;
```

То будет установлено `protected` как менее закрытый уровень доступа для членов `Daughter`, которые унаследованы от материнского класса. Это так, все члены, которые были `public` в `Mother`, станут `protected` в `Daughter`. Конечно, это не исключит для `Daughter` декларирования его собственных *public* членов. Это *менее закрытый уровень доступа* только устанавливается для членов, унаследованных от `Mother`.

Если не задан уровень доступа при наследовании, компилятор присваивает *private* для классов, объявленных с ключевым словом `class`, и *public* для того, что декларировано с помощью `struct`.

Что наследуется из базового класса?

В принципе, производный класс наследует каждого члена базового класса, исключая то:

- Что это конструктор и деструктор.
- Что это оператор присваивания членов (*operator*=).
- Что это друзья.
- Что это *private* члены.

Хотя конструкторы и деструкторы базового класса не наследуются в производном классе, они всё ещё вызваются конструктором производного класса. Пока иное не задано, конструкторы производных классов вызвают предопределённые конструкторы их базовых классов (т.е., конструктор не принимает аргументов), которые должны выполняться.

Вызов предопределённого конструктора базового класса возможен с использованием того же синтаксиса, что и при инициализации переменных-членов в списке инициализации:

```
derived_constructor_name (parameters) : base_constructor_name (parameters)
{...}
```

Например:

```
1  // constructors and derived classes
2  #include <iostream>
3  using namespace std;
4
5  class Mother {
6    public:
7      Mother ()
8        { cout << "Mother: no
9  parameters\n"; }
10     Mother (int a)
11       { cout << "Mother: int
12 parameter\n"; }
13 };
14
15 class Daughter : public Mother {
16   public:
17     Daughter (int a)
18       { cout << "Daughter: int
19 parameter\n\n"; }
20 };
```

```
Mother: no parameters
Daughter: int parameter

Mother: int parameter
Son: int parameter
```

```
21
22 class Son : public Mother {
23   public:
24     Son (int a) : Mother (a)
25       { cout << "Son: int
26 parameter\n\n"; }
27 };
28
29 int main () {
30   Daughter kelly(0);
     Son bud(0);

     return 0;
   }
```

Отметьте разницу между тем, как вызвается конструктор `Mother`, когда создаётся новый `Daughter` объект, и когда это `Son` объект. Разница возникает благодаря разным объявлениям конструктора в `Daughter` и `Son`:

```
1 Daughter (int a)          // ничего не задано: вызвается предопредённый
2 конструктор
  Son (int a) : Mother (a)  // конструктор задан: вызывается это заданный
  конструктор
```

Множественное наследование

Класс может наследовать более чем от одного класса просто заданием более одного базового класса, которые разделяются запятыми в списке базовых классов (т.е., после двоеточия). Например, если программа задаёт класс для вывода на экран, названный `Output`, и мы хотим, чтобы наши классы `Rectangle` и `Triangle` также наследовали его члены в дополнение к тем, что в `Polygon`, мы должны написать:

```
1 class Rectangle: public Polygon, public Output;
2 class Triangle: public Polygon, public Output;
```

Вот полный пример:

```
 1 // multiple inheritance
 2 #include <iostream>
 3 using namespace std;
 4
 5 class Polygon {
 6   protected:
 7     int width, height;
 8   public:
 9     Polygon (int a, int b) :
10 width(a), height(b) {}
11 };
12
13 class Output {
14   public:
15     static void print (int i);
16 };
17
18 void Output::print (int i) {
19   cout << i << '\n';
20 }
```

```
20
10
```

```
21
22 class Rectangle: public Polygon,
23 public Output {
24   public:
25     Rectangle (int a, int b) :
26 Polygon(a,b) {}
27     int area ()
28       { return width*height; }
29 };
30
31 class Triangle: public Polygon,
32 public Output {
33   public:
34     Triangle (int a, int b) :
35 Polygon(a,b) {}
36     int area ()
37       { return width*height/2; }
38 };
39
40 int main () {
41   Rectangle rect (4,5);
   Triangle trgl (4,5);
   rect.print (rect.area());
   Triangle::print (trgl.area());
   return 0;
}
```

Полиморфизм

Перед тем, как углубиться в эту главу, вы должны иметь хорошее понимание указателей и наследования класса. Если вы не уверены в назначении любого из следующих выражений, вы должны повторить указанные рядом разделы:

Выражение:	Объясняется в:
`int A::b(int c) { }`	Классы
`a->b`	Стурктуры данных
`class A: public B {};`	Дружба и наследование

Указатели на базовый класс

Одной из ключевых особенностей наследования классов является то, что указатель на призводный класс совместим по типу с указателем на его базовый класс. *Полиморфизм* – это искусство использования этого простого, но мощного и многогранного средства.

Пример с *rectangle* и *triangle* классами может быть переписан с помощью указателей, принимая в расчёт вышесказанное:

```
// pointers to base class
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
};

class Rectangle: public Polygon {
  public:
    int area()
      { return width*height; }
};

class Triangle: public Polygon {
  public:
    int area()
      { return width*height/2; }
};

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

```
20
10
```

Функция `main` объявляет два указателя на `Polygon` (названные `ppoly1` и `ppoly2`). Есть

назначенные адреса rect и trgl соответственно, которые являются объектами типа Rectangle и Triangle. Такие присваивания правильны, поскольку оба и Rectangle, и Triangle – это классы производные от Polygon.

Разыменовывание ppoly1 и ppoly2 (с помощью *ppoly1 и *ppoly2) правильное и позволяет нам получить доступ к членам указываемых ими объектов. Например, следующие два выражения будут эквивалентны в предыдущем примере:

```
ppoly1->set_values (4,5);
rect.set_values (4,5);
```

Но, поскольку тип ppoly1 и ppoly2 – это указатели на Polygon (и они не указывают ни на Rectangle, ни на Triangle), а только члены, унаследованные от Polygon, могут быть доступны, но не те, что от производных классов Rectangle и Triangle, вот почему программа выше получает доступ к членам area обоих объектов, используя вместо указателей rect и trgl непосредственно. Указатели на базовый класс не могут получить доступ к членам area.

Член area имел бы доступ с указателями на Polygon, если бы area был членом Polygon, а не членом его производных классов, но проблема в том, что Rectangle и Triangle реализуют разные версии area, вследствие чего нет единой общей версии, которая была бы реализована в базовом классе.

Виртуальные члены

Виртуальный член – это функция-член, которая может быть переопределена в производном классе, чтобы предохранить её свойства от вызова через ссылки. Синтаксис для функции, чтобы объявить её виртуальной, это предварить объявление ключевым словом virtual:

```
// virtual members
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area ()
      { return 0; }
};

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return (width * height / 2);
}
};

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon poly;
```

```
20
10
0
```

```
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  Polygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  cout << ppoly3->area() << '\n';
  return 0;
}
```

В этом примере все три класса (`Polygon`, `Rectangle` и `Triangle`) имеют те же члены: `width`, `height`, и функции `set_values` и `area`.

Функция-член `area` была объявлена как `virtual` в базовом классе, поскольку это позже переопределено в каждом из производных классов. Не виртуальные члены могут также быть переопределены в производных классах, но к не виртуальным членам производных классов не может быть доступа через ссылки базового класса. То есть, если `virtual` удалено из декларации `area` в примере выше, все три вызова `area` вернут ноль, поскольку во всех классах должна вызваться вместо этого версия базового класса.

И, в сущности, то, что делает ключевое слово `virtual`, это позволяет членам производного класса с тем же именем, что и в базовом классе, подходящим образом вызываться по указателю. Более точно: когда тип указателя, это указатель на базовый класс, который указывает на объект производного класса, как в примере выше.

Класс, который объявляет или наследует виртуальную функцию, называется *полиморфный класс*.

Заметьте, что, не смотря на виртуальность одного из его членов, `Polygon` был обычным классом, от которого даже был создан экземпляр объекта (`poly`) с его собственным определением члена `area`, который всегда возвращает 0.

Абстрактные базовые классы

Абстрактные базовые классы – это нечто очень похожее на класс `Polygon` в предыдущем примере. Это классы, которые могут быть использованы только как базовые классы, следовательно, им разрешено иметь виртуальные функции-члены без определения (известные, как чисто виртуальные функции). Синтаксис – это замещение их определения с помощью `=0` (и знак равенства, и нуль):

Абстрактный базовый класс `Polygon` может вглядеть похожим на такой:

```
// abstract class CPolygon
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area () =0;
};
```

Заметьте, что `area` не имеет определения; это было заменено `=0`, что делает объект *чисто виртуальной функцией*. Классы, которые содержат хотя бы одну *чисто виртуальную функцию*, известны как *абстрактные базовые классы*.

Абстрактные базовые классы не могут использоваться для создания экземпляров (реализации)

объектов. Таким образом, эта последняя версия абстрактного базового класса `Polygon` не может быть использована для объявления объектов, подобных этим:

```
Polygon mypolygon;   // не работает, если Polygon абстрактный базовый класс
```

Но *абстрактный базовый класс* не совсем бесполезен. Он может использоваться для создания указателей на него, чтобы получить все преимущества полиморфизма. Например, следующее объявление указателя будет правильным:

```
Polygon * ppoly1;
Polygon * ppoly2;
```

И указатель может реально быть разыменован, когда указывает на объекты производных (не абстрактных) классов. Вот полный пример:

```
// abstract base class
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area (void) =0;
};

class Rectangle: public Polygon {
  public:
    int area (void)
      { return (width * height); }
};

class Triangle: public Polygon {
  public:
    int area (void)
      { return (width * height / 2);
}
};

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  return 0;
}
```

```
20
10
```

В этом примере объекты разных, но связанных типов, имеют ссылки для использования уникального типа указателя (`Polygon*`), и каждый раз вызвается правильная функция-член, только потому, что они виртуальны. Это может быть реально полезно в некоторых ситуациях. Например, есть возможность членам абстрактного базового класса `Polygon` использовать

специальный указатель `this` для доступа к правильным виртуальным членам, хотя сам класс `Polygon` не имеет реализации функции *this*:

```
// pure virtual members can be called
// from the abstract base class
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area() =0;
    void printarea()
      { cout << this->area() << '\n';
}
};

class Rectangle: public Polygon {
  public:
    int area (void)
      { return (width * height); }
};

class Triangle: public Polygon {
  public:
    int area (void)
      { return (width * height / 2);
}
};

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  return 0;
}
```

```
20
10
```

Виртуальные члены и абстрактные классы придают C++ полиморфные характеристики, наиболее полезные для объектно-ориентированных проектов. Конечно, примеры выше – это очень простые примеры применения, но эти возможности могут использоваться с массивами объектов или динамически располагаемыми объектами.

Вот пример, комбинирующий некоторые из возможностей в последних разделах, такие как динамическая память, инициализация конструктора и полиморфизм:

```
// dynamic allocation and
polymorphism
#include <iostream>
using namespace std;

class Polygon {
  protected:
```

```
20
10
```

```
    int width, height;
  public:
    Polygon (int a, int b) :
width(a), height(b) {}
    virtual int area (void) =0;
    void printarea()
      { cout << this->area() << '\n';
}
};

class Rectangle: public Polygon {
  public:
    Rectangle(int a,int b) :
Polygon(a,b) {}
    int area()
      { return width*height; }
};

class Triangle: public Polygon {
  public:
    Triangle(int a,int b) :
Polygon(a,b) {}
    int area()
      { return width*height/2; }
};

int main () {
  Polygon * ppoly1 = new Rectangle
(4,5);
  Polygon * ppoly2 = new Triangle
(4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  delete ppoly1;
  delete ppoly2;
  return 0;
}
```

Заметьте, что указатели `ppoly`:

```
Polygon * ppoly1 = new Rectangle (4,5);
Polygon * ppoly2 = new Triangle (4,5);
```

декларируются как имеющие тип «указателя на `Polygon`», но объекты были объявлены, как имеющие непосредственно тип производных классов (`Rectangle` и `Triangle`).

Другие возможности языка:

Трансформирования типа

Неявное превращение

Неявное превращение автоматически выполняется, когда значение копируется в совместимый тип. Например:

```
1 short a=2000;
2 int b;
3 b=a;
```

Здесь значение `a` переводится из `short` в `int` без необходимости какого-либо явного оператора. Это известно как *стандартное преобразование*. Стандартное преобразование затрагивает фундаментальные типы данных и позволяет конверсию между числовыми типами (`short` в `int`, `int` в `float`, `double` в `int`...), в или из `bool`, и некоторые преобразования указателей.

Конвертирование в `int` из некоторого меньшего целого типа или в `double` из `float` известно как *поддержка* (*promotion)*, оно гарантирует получение точно того же значения в целевом типе. Другие преобразования между арифметическими типами не всегда способны представить точно такое же значение:

- Если отрицательное целое значение преобразуется в беззнаковый тип, результирующее значение относится к его 2му дополнительному побитовому представлению (т.е., `-1` становится наибольшим значением представляемым типом, `-2` вторым наибольшим, ...).
- Конверсия из/в `bool` подразумевает, что `false` эквивалентно *нулю* (для числовых типов) и *нуль-указателю* (для типов указателей); `true` эквивалентно всем другим значениям и конвертируемо в эквивалент `1`.
- Если преобразование осуществляется из типа с плавающей точкой в целый тип, значение усекается (десятичная часть удаляется). Если этот результат лежит вне диапазона значения представляемого типом, преобразование приведёт к *неопределённому поведению*.
- Иначе, если конверсия происходит между числовыми типами одного рода (целое в целое или значение с плавающей точкой в значение с плавающей точкой), преобразование правильно, но значение *специфически реализуемо* (и может быть непереносимо).

Некоторые из этих преобразований могут повлечь потерю точности, о которой компилятор может синализировать предупреждением. Этого предупреждения можно избежать при явном преобразовании.

Для не фундаментальных типов, массивов и функций осуществляется неявная конвертация в указатели, а указатели, в основном, допускают следующие преобразования:

- *Нуль-указатели* могут конвертироваться в указатели любого типа.
- Указатели в любой тип могут быть конвертированы в `void` указатели.
- Указатели *восходящие* (*upcast)*, указатели на производный класс, могут конвертироваться в указатели *доступного* и *однозначного* базового класса без модификации его `const` или `volatile` квалификаций.

Неявные преобразования с классами

В мире классов неявные преобразования могут управляться значениями трёх функций-членов:

- **Конструкторы с единственным аргументом:** позволяют неявное преобразование из обычных типов в инициализацию объекта.

- **Оператор присваивания:** позволяет неявное преобразование из обычного типа в присваивания.
- **Оператор однотипных преобразований:** позволяет неявное преобразование обычных типов.

Например:

```
// implicit conversion of classes:
#include <iostream>
using namespace std;

class A {};

class B {
public:
  // преобразование из A (конструктор):
  B (const A& x) {}
  // преобразование из A (присваивание):
  B& operator= (const A& x) {return *this;}
  // преобразование в A (однотипный оператор)
  operator A() {return A();}
};

int main ()
{
  A foo;
  B bar = foo;     // вызов конструктора
  bar = foo;       // вызов присваивания
  foo = bar;       // вызов однотипного оператора
  return 0;
}
```

Одинотипный оператор использует обычный синтаксис: он использует ключевое слово `operator`, следующее за целевым типом и пустые скобки. Заметьте, что возвращаемый тип – это целевой тип и, следовательно, не задаётся перед ключевым словом `operator`.

Ключевое слово explicit

При вызове функции язык C++ позволяет одно неявное преобразование, осуществляемое для каждого аргумента. Это может стать проблемой для классов, поскольку это не всегда то, что намечено. Например, если мы добавляем следующую функцию к последнему примеру:

```
void fn (B arg) {}
```

Функция принимает аргумент типа `B`, но это может также хорошо вызываться с объектом типа `A` в качестве аргумента:

```
fn (foo);
```

Это может или не может быть тем, что подразумевалось. Но в данном случае это может быть предотвращено маркировкой соответствующего конструктора ключевым словом `explicit`:

```
// explicit:
#include <iostream>
using namespace std;
```

```
4
5  class A {};
6
7  class B {
8  public:
9    explicit B (const A& x) {}
10   B& operator= (const A& x) {return *this;}
11   operator A() {return A();}
12 };
13
14 void fn (B x) {}
15
16 int main ()
17 {
18   A foo;
19   B bar (foo);
20   bar = foo;
21   foo = bar;
22
23 //  fn (foo);  // not allowed for explicit ctor.
24   fn (bar);
25
26   return 0;
27 }
```

Дополнительно конструкторы, маркированные с `explicit`, не могут вызываться синтаксисом, похожим на присваивание. В примере выше `bar` не может быть сконструировано с помощью:

```
B bar = foo;
```

Однотипные функции-члены (те, что описаны в предыдущих разделах) могут также быть заданы как `explicit`. Это предотвращает неявное преобразование тем же способом, которым заданные с помощью `explicit` конструкторы осуществляют это с целевым типом.

Подбор типа (Type casting)

C++ – это строго типизированный язык. Многие преобразования, особенно те что подразумевают разную интерпретацию значения, требуют явного преобразования, известного в C++ как *подбор типа*. Здесь есть два основных синтаксиса для базового подбора типа: *функциональный* и *с-подобный*:

```
1 double x = 10.3;
2 int y;
3 y = int (x);    // функциональная нотация
4 y = (int) x;    // нотация с-подобного подбора
```

Функциональности этих основных форм подбора типа достаточно для большинства нужд с базовыми типами данных. Однако эти операторы могут быть приложены огульно к классам и указателям на классы, что может давать код, который, будучи синтаксически верным, приведёт к ошибкам при работе. Например, следующий код компилируется без ошибок:

```
1 // class type-casting
2 #include <iostream>
3 using namespace std;
4
5 class Dummy {
```

```
6     double i,j;
7  };
8
9  class Addition {
10    int x,y;
11   public:
12    Addition (int a, int b) { x=a;
13 y=b; }
14    int result() { return x+y;}
15 };
16
17 int main () {
18   Dummy d;
19   Addition * padd;
20   padd = (Addition*) &d;
21   cout << padd->result();
22   return 0;
   }
```

Программа объявляет указатель на `Addition`, но затем присваивает ему ссылку на объект другого не соответствующего типа, используя явный подбор типа:

```
padd = (Addition*) &d;
```

Неограниченный явный подбор типа позволяет конвертировать любой указатель в указатель любого другого типа, независимо от типа того, на что они указывают. Последующий вызов члена `result` вызовет либо ошибку при работе, либо приведёт к какому-то другому неожиданному результату.

В порядке управления этими типами преобразований между классами, мы имеем четыре специфических оператора подбора: `dynamic_cast`, `reinterpret_cast`, `static_cast` и `const_cast`. Их формат – следовать новому типу, заключённому между угловыми скобками, и, непосредственно после, выражению в скобках, которое должно преобразовываться.

```
dynamic_cast <new_type> (expression)
reinterpret cast <new type> (expression)
static_cast <new_type> (expression)
const_cast <new_type> (expression)
```

Традиционный подбор типа, эквивалентный этим выражениям, будет:

```
(new_type) expression
new_type (expression)
```

но каждое из них с собственными специальными характеристиками:

dynamic_cast

`dynamic_cast` может использоваться только с указателями и ссылками на классы (или с `void*`). Его назначение – убедиться в том, что результат преобразования типа указывает на правильный полный объект целевого типа указателя.

Это, естественно, включает в себя *восходящий указатель* (преобразующий из указателя на производный класс в указатель на базовый) так же, как при *неявном преобразовании*.

Но `dynamic_cast` может работать также с *нисходящими* (преобразование из указателя на базовый класс в указатель на производный) полиморфными классами (те, что с виртуальными членами), если – и только если – указываемый объект правильный полный объект целевого типа. Например:

```
// dynamic_cast
#include <iostream>
#include <exception>
using namespace std;

class Base { virtual void dummy() {}
};
class Derived: public Base { int a;
};

int main () {
  try {
    Base * pba = new Derived;
    Base * pbb = new Base;
    Derived * pd;

    pd = dynamic_cast<Derived*>(pba);
    if (pd==0) cout << "Null pointer
on first type-cast.\n";

    pd = dynamic_cast<Derived*>(pbb);
    if (pd==0) cout << "Null pointer
on second type-cast.\n";

  } catch (exception& e) {cout <<
"Exception: " << e.what();}
  return 0;
}
```

```
Null pointer on second type-cast.
```

Примечание о совместимости: этот тип `dynamic_cast` требует *Run-Time Type Information (RTTI, информацию о типе при работе программы)* для отслеживания динамических типов. Некоторые компиляторы поддерживают эту возможность как необязательную, которая невозможна по умолчанию. А это нуждается в разрешении проверки типа при работе, используя `dynamic_cast` для правильной работы с этими типами.

Код выше пытается выполнить две динамические пробы с указателями на объекты типа `Base*` (`pba` и `pbb`) и указателями на объекты типа `Derived*`, но только первая из них успешна. Обратите внимание на их соответствующую инициализацию:

```
Base * pba = new Derived;
Base * pbb = new Base;
```

Хотя оба – это указатели типа `Base*`, `pba` в действительности указывает на объект типа `Derived`, тогда как `pbb` указывает на объект типа `Base`. Таким образом, когда их соответствующий тип отбора выполняется с использованием `dynamic_cast`, `pba` указывает на полный объект класса `Derived`, тогда как `pbb` указывает на объект класса `Base`, который является неполным объектом класса `Derived`.

Когда `dynamic_cast` не может опробовать указатель, поскольку он не полный объект требуемого класса, как при втором преобразовании предыдущего примера, он возвращает *нуль-указатель* для индикации неудачи. Если `dynamic_cast` используется для преобразования в ссылку на тип, а преобразование невозможно, вместо него возникает исключение типа `bad_cast`.

`dynamic_cast` может также выполнять другие неявные пробы, допускаемые с указателями: проба нуль-указателем между типами указателей (даже между несвязанными классами), и опробирование любых указателей любого типа с `void*` указателем.

static_cast

static_cast может выполнять преобразование указателей на связанные классы, не только по *восходящей* (из указателя на производный в указатель на базовый класс), но и по *нисходящей* (из указателя на базовый в укзатель на производный класс). Проверка не выполняется во время работы программы, чтобы гарантировать, что объект конвертирован, фактически, в полный объект целевого типа. Таким образом, это сделано для программистов, чтобы убедиться, что преобразование безопасно. С другой стороны, это не накладывается поверх проверки безопасности типа в dynamic_cast.

```
1 class Base {};
2 class Derived: public Base {};
3 Base * a = new Base;
4 Derived * b = static_cast<Derived*>(a);
```

Это будет верным кодом, хотя b будет указывать на неполный объект класса, что должно привести к ошибке при работе программы, если будет разыменовывание.

Таким образом, static_cast способно выполняться с указателем на классы не только при пробразованиях, допустимых неявно, но также при противоположных им преобразованиях.

static_cast также способно выполнять все преобразования, допустимые неявно (не только те, что с указателями на классы), и способно выполнять противоположные им преобразования. Можно:

- Конвертировать из void* в любой тип указателей. В этом случае оно гарантирует, что, если void* значение было получено конвертированием из того же самого типа указателя, результирующее значение указателя является тем же.
- Конвертировать целые, значения с плавающей точкой и перечисляемые типы в перечисляемые типы.

Дополнительно static_cast может также выполнить следующее:

- Явный вызов конструктора с единственным аргументом или оператором преобразования.
- Конвертировать в *ссылки r-значений*.
- Конвертировать enum class значения в целые или значения с плавающей точкой.
- Конвертировать любой тип в void, оценивая и отбрасывая значение.

reinterpret_cast

reinterpret_cast конвертирует любой тип указателя в любой другой тип указателя, даже для несвязанных классов. Результат операции является просто двоичной копией значения из одного укзателя в другой. Все преобразования укзателей допустимы: ни содержимое указываемого, ни сам тип указателя не проверяются.

Оно может опробовать указатели на или из целых типов. Формат, в котором это целое значение представляет собой указатель, является платформо-зависимым. Единственной гарантией будет то, что проба указателя на целый тип подтверждает полную вместимость содержимого (такого как intptr_t), и гарантирует способность обратной пробы в правильный указатель.

Преобразования, которые могут быть выполнены reinterpret_cast, но не static_cast, есть низкоуровневые операции, основанные на ином толковании двоичного представления типов, которые во многих случаях реализуются в системо-зависимом коде и, значит, несовместны. Например:

```
1 class A { /* ... */ };
2 class B { /* ... */ };
3 A * a = new A;
4 B * b = reinterpret_cast<B*>(a);
```

Этот код компилируется, хотя он не имеет смысла, поскольку теперь `b` указывает на объект полностью несвязанного и, похоже, несовместимого класса. Разыменовывание `b` опасно.

const_cast

Этот тип опробывания обрабатывает константы объектов, указываемых указателями, либо для задания, либо для удаления. Например, в порядке передачи константы указателем в функцию, которая предполагает аргумент не константу:

```
1  // const_cast
2  #include <iostream>
3  using namespace std;
4
5  void print (char * str)
6  {
7    cout << str << '\n';
8  }
9
10 int main () {
11   const char * c = "sample text";
12   print ( const_cast<char *> (c) );
13   return 0;
14 }
```

```
sample text
```

Пример выше гарантированно работает, поскольку функция `print` не вписывает в указываемый объект. Хотя заметьте, что удаление постоянной указываемого объекта фактически делает запись, приводя к *неопределённому поведению*.

typeid

`typeid` позволяет проверить тип выражения:

```
typeid (expression)
```

Этот оператор возвращает ссылку на постоянный объект типа `type_info`, который определён в стандартном заголовочном файле `<typeinfo>`. Значение, возвращаемое `typeid`, может быть сравнимо с другим значением, возвращаемым `typeid`, с использованием операторов `==` и `!=`, или может служить для получения символа нуля, завершающего последовательность, представляющую тип данных, или имени класса, используя его `name()` член.

```
1  // typeid
2  #include <iostream>
3  #include <typeinfo>
4  using namespace std;
5
6  int main () {
7    int * a,b;
8    a=0; b=0;
9    if (typeid(a) != typeid(b))
10   {
11     cout << "a and b are of different
12 types:\n";
13     cout << "a is: " <<
14 typeid(a).name() << '\n';
15     cout << "b is: " <<
16 typeid(b).name() << '\n';
     }
     return 0;
   }
```

```
a and b are of different types:
a is: int *
b is: int
```

Когда `typeid` применяется к классам, `typeid` использует RTTI для отслеживания типа динамических объектов. Когда `typeid` применяется к выражению, чей тип является полиморфным классом, результатом будет тип наиболее полного производного объекта:

```
// typeid, polymorphic class
#include <iostream>
#include <typeinfo>
#include <exception>
using namespace std;

class Base { virtual void f(){} };
class Derived : public Base {};

int main () {
  try {
    Base* a = new Base;
    Base* b = new Derived;
    cout << "a is: " <<
typeid(a).name() << '\n';
    cout << "b is: " <<
typeid(b).name() << '\n';
    cout << "*a is: " <<
typeid(*a).name() << '\n';
    cout << "*b is: " <<
typeid(*b).name() << '\n';
  } catch (exception& e) { cout <<
"Exception: " << e.what() << '\n'; }
  return 0;
}
```

```
a is: class Base *
b is: class Base *
*a is: class Base
*b is: class Derived
```

Заметьте: строка, возвращённая членом `name` типа `type_info`, зависит от специфики реализации вашего компилятора и библиотеки. Это не обязательно простая строка с её характерным именем типа, как в компиляторе, использованном для выполнения этого вывода.

Заметьте, что тип с `typeid` подразумевает для указателей, что они сами указатели на тип (и `a`, и `b` являются типом `class Base *`). Однако, когда `typeid` применяется к объектам (подобно `*a` и `*b`) `typeid` принимает их динамический тип (т.е., тип их наболее полного производного объекта).

Если тип, вычисляемый `typeid`, является указателем, предваряемым оператором разыменовывания (`*`), и этот указатель имеет значение нуль, `typeid` выводит `bad_typeid` исключение.

Исключения

Исключения предоставляют возможность регировать на исключительные ситуации (подобно ошибкам во время работы) в программах, передавая управление специальным функциям, называемым *поддержками* (*handlers*).

Для перехвата исключений часть кода размещается под проверкой исключения. Это делается с помощью заключения в скобки порции кода в *блок-пробу* (*try-block*). Когда возникает исключительная ситуация внутри этого блока, вбрасывается исключение, что передаёт управление поддержке исключения. Если исключение не вбрасывается, код продолжается нормально и все поддержки игнорируются.

Исключение вбрасывают, используя ключевое слово `throw` внутри блока `try`. Поддержки исключения объявляются с помощью ключевого слова `catch`, которое должно размещаться сразу за `try` блоком:

```
1  // exceptions
2  #include <iostream>
3  using namespace std;
4
5  int main () {
6    try
7    {
8      throw 20;
9    }
10   catch (int e)
11   {
12     cout << "An exception occurred.
13 Exception Nr. " << e << '\n';
14   }
15   return 0;
   }
```

```
An exception occurred. Exception Nr.
20
```

Код под поддержкой исключения заключён в блок `try`. В этом примере код просто вбрасывает исключение:

```
throw 20;
```

Выражение `throw` принимает один параметр (в нашем случае целое значение `20`), который передаётся как аргумент в поддержку исключения.

Поддержка исключения декларирована с помощью ключевого слова `catch` сразу после закрывающей скобки блока `try`. Синтаксис для `catch` похож на обычную функцию с одним параметром. Тип этого параметра очень важен, поскольку тип аргумента, передаваемый выражением `throw`, проверяется, и только в случае их совпадения исключение захватывается поддержкой.

Множественные поддержки (т.е., `catch` выражения) могут соединяться в цепочки; каждая из них с разными типами параметра. Только поддержка, чей тип аргумента совпадает с типом исключения, заявленным в `throw` выражении, выполняется.

Если многоточие (`...`) используется в качестве параметра `catch`, поддержка будет захватываться любым исключением, не важно, какой тип исключения вбрасывается. Это может использоваться в качестве поддержки по умолчанию, которая захватывается всеми иключениями, не захваченными другими поддержками:

```
1  try {
2    // code here
3  }
4  catch (int param) { cout << "int exception"; }
5  catch (char param) { cout << "char exception"; }
```

```
6 catch (...) { cout << "default exception"; }
```

В данном случае последняя поддержка будет захвачена любым вброшенным исключением с типом, который не `int` и не `char`.

После того, как исключение было поддержано программой, исключение подытоживается блоком *try-catch*, а не выражением `throw`!

Есть возможность вкладывать `try-catch` блоки внутрь внешних `try` блоков. В этих случаях у нас есть возможность, что внутренний `catch` блок опередит в исключении внешний уровень. Это происходит с выражением `throw;` без аргументов. Например:

```
1  try {
2    try {
3      // code here
4    }
5    catch (int n) {
6      throw;
7    }
8  }
9  catch (...) {
10   cout << "Exception occurred";
11 }
```

Спецификация исключений

Прежние коды могли содержать *динамические спецификации исключения*. Теперь ими пренебрегают в C++, но они всё ещё поддерживаются. *Динамическая спецификация исключения* следует за объявлением функции, добавляя спецификатор `throw` к ней. Например:

```
double myfunction (char param) throw (int);
```

Здесь объявлена функция, названная `myfunction`, которая принимает один аргумент типа `char` и возвращает значение типа `double`. Если эта функция вбрасывает исключение какого-то другого типа, а не `int`, вызывается функция `std::unexpected` вместо поиска поддержки или вызывается `std::terminate`.

Если спецификатор `throw` остаётся пустым без типа, это означает, что `std::unexpected` вызывается для любого исключения. Функции без спецификатора `throw` (обычные функции) никогда не вызвают `std::unexpected`, но следуют обычному пути поиска поддержки для их исключения.

```
1 int myfunction (int param) throw(); // все исключения вызвают unexpected
2 int myfunction (int param);         // поддержка нормальных исключений
```

Стандартные исключения

Стандартная библиотека C++ предлагает базовый класс, который специально разработан для объявления объектов, которые будут вброшены как исключения. Он назван `std::exception` и определён в `<exception>` файле заголовков. Этот класс имеет виртуальные функции-члены, названные `what`, которые возвращают последовательности символов с нуль-завершением (типа `char *`), и которые могут быть переписаны в производных классах для наполнения неким описанием исключений.

```
1 // using standard exceptions
```

```
My exception happened.
```

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

class myexception: public exception
{
  virtual const char* what() const
throw()
  {
    return "My exception happened";
  }
} myex;

int main () {
  try
  {
    throw myex;
  }
  catch (exception& e)
  {
    cout << e.what() << '\n';
  }
  return 0;
}
```

Мы поместили поддержку, которая ловит объекты исключения по ссылке (обратите внимание на амперсанд `&` после типа), таким образом, это обслуживает и классы, производные от `exception`, подобно нашему `myex` объекту типа `myexception`.

Все вбросы исключений компонентами стандартной библиотеки C++ вбрасывают исключения, произведённые от этого класса `exception`. Это:

исключение	описание
`bad_alloc`	вбрасывается `new` при ошибке выделения
`bad_cast`	вбрасывается `dynamic_cast` при неудаче в динамической пробе
`bad_exception`	вбрасывается некоторыми спецификаторами динамического исключения
`bad_typeid`	вбрасывается `typeid`
`bad_function_call`	вбрасывается пустыми `function` объектами
`bad_weak_ptr`	вбрасывается `shared_ptr`, когда передаётся плохой `weak_ptr`

Также производными от `exception` заголовочный файл `<exception>` определяет два базовых типа исключения, которые могут наследоваться пользовательскими исключениями для сообщения об ошибках:

исключение	описание
`logic_error`	ошибка, относящаяся к внутренней логике программы
`runtime_error`	ошибка, обнаруженная при работе программы

Типичный пример, где стандартные исключения нуждаются в проверке – это выделение памяти:

```
// bad_alloc standard exception
#include <iostream>
#include <exception>
using namespace std;

```

```
6  int main () {
7    try
8    {
9      int* myarray= new int[1000];
10   }
11   catch (exception& e)
12   {
13     cout << "Standard exception: " <<
14 e.what() << endl;
15   }
16   return 0;
   }
```

Исключение, которое может быть захвачено поддержкой исключения в этом примере – это `bad_alloc`. Поскольку `bad_alloc` является производной от стандартного базового класса `exception`, оно может быть захвачено (захваченное по ссылке, захватывает все связанные классы).

Директивы препроцессора

Директивы препроцессора – это строки, включаемые в код программы, предварённые знаком (#). Эти строки не являются выражениями программы, но директивами для *препроцессора*. Препроцессор проверяет код перед началом реальной компиляции и выполняет все эти директивы до того, как любой код реально будет сгенерирован для обычных выражений.

Эти *директивы препроцессора* простираются только на одну строку кода. Как только обнаруживается символ новой строки, директива препроцессора заканчивается. Точка с запятой отсутствует в конце директивы. Единственный способ продолжить директиву препроцессора более, чем на одну строку, это предварить символ перехода на новую строку с помощью обратной косой черты (\).

macro определения (#define, #undef)

Чтобы определить препроцессорный макрос, мы можем использовать `#define`. Синтаксис:

```
#define identifier replacement
```

Когда препроцессор встречает эту директиву, он замещает её любым обнаружением `identifier` в остальной части кода с помощью `replacement`. Это `replacement` может быть выражением, утверждением, блоком или просто чем-то ещё. Препроцессор не понимает правильности C++, он просто замещает любое обнаружение `identifier` с помощью `replacement`.

```
1 #define TABLE_SIZE 100
2 int table1[TABLE_SIZE];
3 int table2[TABLE_SIZE];
```

После замещения препроцессором `TABLE_SIZE` код становится эквивалентен:

```
1 int table1[100];
2 int table2[100];
```

`#define` может работать с параметрами для определения макроса функции:

```
#define getmax(a,b) a>b?a:b
```

Это будет замещено любым обнаружением `getmax` с последующими двумя аргументами через выражение в *replacement*, но также замещается каждый аргумент его идентификатором, точно как вы бы ожидали, если бы это была функция:

```
1  // function macro
2  #include <iostream>
3  using namespace std;
4
5  #define getmax(a,b) ((a)>(b)?(a):(b))
6
7  int main()
8  {
9    int x=5, y;
10   y= getmax(x,2);
11   cout << y << endl;
12   cout << getmax(7,x) << endl;
13   return 0;
14 }
```

```
5
7
```

Определённый макрос не сказывается на структуре блока. Макрос остаётся до тех пор, пока он не будет отменён с помощью `#undef` директивы препроцессора:

```
#define TABLE_SIZE 100
int table1[TABLE_SIZE];
#undef TABLE_SIZE
#define TABLE_SIZE 200
int table2[TABLE_SIZE];
```

Этим сгенерируется тот же код:

```
int table1[100];
int table2[200];
```

Определения макро-функции принимают два специальных оператора (`#` и `##`) в замещающей последовательности:

Если оператор `#` используется перед параметром, используемым в замещающей последовательности, этот параметр замещается строковым литералом (как если бы он был заключён в двойные кавычки)

```
#define str(x) #x
cout << str(test);
```

Это будет транслировано в:

```
cout << "test";
```

Оператор связывает два аргумента, не оставляя пробелов между ними:

```
#define glue(a,b) a ## b
glue(c,out) << "test";
```

Это также будет транслировано в:

```
cout << "test";
```

Поскольку препроцессорные замещения происходят до какой-либо проверки C++ синтаксиса, макро определения могут реализовать некоторые хитрости. Но будьте внимательны: код, который очень полагается на заполнении макросами, хуже читать, поскольку ожидаемый синтаксис во многих случаях отличается от обычных выражений, ожидаемых программистами в C++.

Включение условий (#ifdef, #ifndef, #if, #endif, #else и #elif)

Эти директивы позволяют включать или исключать часть кода программы, если выполняются некоторые условия.

`#ifdef` позволяет разделам программы быть скомпилированными только, если макрос, который задан в качестве параметра, был определён, не важно, каково его значение. Например:

```
#ifdef TABLE_SIZE
int table[TABLE_SIZE];
#endif
```

В данном случае строка кода `int table[TABLE_SIZE];` будет компилироваться, если `TABLE_SIZE` было предварительно определено с помощью `#define` вне зависимости от его значения. Если это не было определено, строка не будет включена в компиляцию программы.

`#ifndef` служит для прямо противоположного: код между `#ifndef` и `#endif` директивами компилируется только, если заданный идентификатор не был ранее определён. Например:

```
#ifndef TABLE_SIZE
#define TABLE_SIZE 100
#endif
int table[TABLE_SIZE];
```

В этом случае, если с появлением в этом куске кода макрос `TABLE_SIZE` не был совсем определён, это будет определено значением 100. Если это уже существует, то будет сохранять предыдущее значение, поскольку `#define` директива не будет выполняться.

`#if`, `#else` и `#elif` (т.е., "else if") директивы служат для задания некоторых условий, которые должны быть выполнены, чтобы часть кода, окружаемая ими, была скомпилирована. Условия, которые следуют за `#if` или `#elif`, могут вычислять только постоянные выражения, включая макро выражения. Например:

```
#if TABLE_SIZE>200
#undef TABLE_SIZE
#define TABLE_SIZE 200

#elif TABLE_SIZE<50
#undef TABLE_SIZE
#define TABLE_SIZE 50

#else
#undef TABLE_SIZE
#define TABLE_SIZE 100
#endif

int table[TABLE_SIZE];
```

Заметьте, что вся структура `#if`, `#elif` и `#else`, связанная в цепочку, завершается с помощью `#endif`.

Поведение `#ifdef` и `#ifndef` может также быть замещено использованием специальных операторов `defined` и `!defined` соответственно, для любых директив `#if` или `#elif`:

```
#if defined ARRAY_SIZE
#define TABLE_SIZE ARRAY_SIZE
#elif !defined BUFFER_SIZE
#define TABLE_SIZE 128
#else
#define TABLE_SIZE BUFFER_SIZE
#endif
```

Контроль строк (#line)

Когда мы компилируем программу, и возникают какие-то ошибки в процессе компиляции, компилятор показывает сообщение об ошибке со ссылкой на имя файла, где обнаружена ошибка, и на номер строки, чтобы легче было найти код, генерирующий ошибку.

Директива `#line` позволяет нам контролировать и номер строки в коде, и имя файла, появление которого мы ожидали, когда возникла ошибка. Её формат есть:

```
#line number "filename"
```

Где `number` – это номер новой строки, который будет присвоен следующей строке кода. Номера удачных строк будут увеличиваться один за другим от этой точки.

`"filename"` не является обязательным параметром, который позволяет переопределить имя файла, который будет показан. Например:

```
1 #line 20 "assigning variable"
2 int a?;
```

Этот код генерирует ошибку, которая будет показана как ошибка в файле `"assigning variable"`, строка 20.

Директива ошибки (#error)

Эта директива прерывает процесс компиляции, когда она обнаруживается, генерируя ошибку компиляции, которая может быть задана, как её параметр:

```
1 #ifndef __cplusplus
2 #error A C++ compiler is required!
3 #endif
```

Этот пример прерывает процесс компиляции, если имя макроса `__cplusplus` не определено (это имя макроса определено по умолчанию во всех C++ компиляторах).

Включение исходного файла (#include)

Эта директива была постоянно использована в других разделах этого руководства. Когда препроцессор встречает директиву `#include`, он замещает её полным содержимым заданного файла заголовка или просто файла. Есть два способа использовать `#include`:

```
1 #include <header>
2 #include "file"
```

В первом случае задан *заголовочный файл* между угловым скобками `<>`. Так используется включение заголовочных файлов, предоставляемых реализацией, таких как заголовочные файлы стандартной библиотеки (`iostream`, `string`,...). Будет ли файл заголовков реальным файлом или существовать в какой-то другой форме, *определённой для реализации*, в любом случае он будет правильно включён этой директивой.

Синтаксис, использованный во второй директиве `#include`, применяет кавычки и включает *файл*. *Файл* ищется на манер *implementation-defined (определённый для реализации)*, который обычно включает текущий путь. В случае, когда файл не найден, компилятор интерпретирует директиву как включение *заголовка*, как если бы кавычки (`""`) были замещены угловым скобками (`<>`).

Pragma директива (#pragma)

Эта директива используется для задания иных опций компилятору. Эти опции являются специфическими для платформы и компилятора, который вы используете. Обратитесь к руководству или справке по вашему компилятору за дополнительной информацией о возможных параметрах, которые вы можете определить через `#pragma`.

Если компилятор не поддерживает специальные аргументы для `#pragma`, он игнорирует их, при этом синтаксической ошибки генерироваться не будет.

Предопределённые макро имена

Следующие имена макросов всегда определены (они все начинаются и заканчиваются двумя символами подчёркивания __):

макрос	значение
`__LINE__`	Целое значение, представляющее текущую строку исходного кода в компилируемом файле.
`__FILE__`	Строковый литерал, содержащий предполагаемое имя компилируемого файла исходного кода.
`__DATE__`	Строковый литерал в форме "Mmm dd yyyy", содержащий дату, когда процесс компиляции начался.
`__TIME__`	Строковый литерал в форме "hh:mm:ss", содержащий время, когда процесс компиляции начался.
`__cplusplus`	Целое число. Все компиляторы C++ имеют эту константу, определённую для некоторых значений. Это значение зависит от версии стандарта, поддерживаемого компилятором: • **199711L**: ISO C++ 1998/2003 • **201103L**: ISO C++ 2011 Компиляторы, не соответствующие требованиям, определяют эту константу как такое же значение максиму в пять цифр. Заметьте, что многие компиляторы не полностью приспособлены, а, следовательно, будут иметь определение этой константы не как значения выше.
`__STD_HOSTED__`	`1`, если реализация является *реализацией под управлением* (со всеми доступными стандартными заголовочными файлами) `0` в противном случае.

Следующие макросы являются не обязательными для объявления, и обычно зависят от доступности этой возможности:

макрос	значение
`__STDC__`	В C: если определено в `1`, реализация подстраивается к стандарту C. В C++: определённая реализация.
`__STDC_VERSION__`	В C: • **199401L**: ISO C 1990, Дополнение 1 • **199901L**: ISO C 1999 • **201112L**: ISO C 2011 В C++: определённая реализация.
`__STDC_MB_MIGHT_NEQ_WC__`	`1`, если многобайтовая кодировка может дать символ другого значения в символьных литералах

`__STDC_ISO_10646__`	Значение в форме `yyyymmL`, задающее дату стандарта Unicode, следующую за кодировкой `wchar_t` символов
`__STDCPP_STRICT_POINTER_SAFETY__`	`1`, если реализация имеет *строгий указатель безопасности* (см. `get_pointer_safety`)
`__STDCPP_THREADS__`	`1`, если программа может иметь более одного потока

Обычная реализация может определять дополнительные константы.

Например:

```
1  // standard macro names
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7    cout << "This is the line number
8  " << __LINE__;
9    cout << " of file " << __FILE__
10 << ".\n";
11   cout << "Its compilation began "
12 << __DATE__;
13   cout << " at " << __TIME__ <<
   ".\n";
     cout << "The compiler gives a
   __cplusplus value of " <<
   __cplusplus;
     return 0;
   }
```

```
This is the line number 7 of file
/home/jay/stdmacronames.cpp.
Its compilation began Nov  1 2005
at 10:12:29.
The compiler gives a __cplusplus
value of 1
```

Edit & Run

C++ Стандартная библиотека

Ввод/вывод с файлами

C++ поддерживает следующие классы для выполнения вывода и ввода символов в/из файлов:

- **ofstream**: Класс потока для записи в файлы
- **ifstream**: Класс потока для чтения из файлов
- **fstream**: Класс потока и для чтения, и для записи из/в файл.

Эти классы произведены непосредственно или косвенно от классов `istream` и `ostream`. Мы уже использовали объекты этих типов (этих классов): `cin` является объектом класса `istream`, а `cout` – объект класса `ostream`. Таким образом, мы уже использовали классы, которые относятся к нашим файловым потокам. И, фактически, мы могли использовать наши файловые потоки тем же способом, каким мы уже использовали `cin` и `cout`, только с единственной разницей, что мы связываем эти потоки с физическими файлами. Давайте взглянем на пример:

```
1  // basic file operations
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main () {
7    ofstream myfile;
8    myfile.open ("example.txt");
9    myfile << "Writing this to a
10 file.\n";
11   myfile.close();
12   return 0;
   }
```

```
[file example.txt]
Writing this to a file.
```

Этот код создаёт файл с именем `example.txt` и вставляет фразу в него так же, как мы это делали, используя `cout`, но использовали файловый поток `myfile` вместо этого.

Но давайте сделаем это пошагово:

Открывание файла

Первая операция обычно выполняется на объекте одного из этих классов, связывая его с реальным файлом. Эта процедура известна как *open a file*. Открыть файл – представлен в программе *потоком*, *stream* (т.е., объект одного из этих классов; в предыдущем примере это был `myfile`) и любая операция ввода или вывода, выполняемая на этом объекте потока, будет приложена к физическому файлу, связанному с ним.

В порядке открытия файла с объектом потока мы используем его член-функцию `open`:

```
open (filename, mode);
```

Где `filename` – это строка, представляющая имя файла, который должнен быть октрыт, а `mode` – это не обязательный параметр с комбинацией следующих флагов:

`ios::in`	Открыть для операций ввода.
`ios::out`	Открыть для операций вывода.
`ios::binary`	Открыть в двоичном режиме.

ios::ate	Установить начальную позицию в конце файла. Если этот флаг не установлен, начальная позиция задаётся с начала файла.
ios::app	Все операции вывода выполняются в конце файла, добавляя содержимое к текущему содержимому файла.
ios::trunc	Если файл открывается для операций вывода и уже существует, его предыдущее содержание удаляется и замещается новым.

Все эти флаги могут быть скомбинированны, используя побитовый оператор OR (|). Например, если мы хотим открыть файл example.bin в двоичном режиме, чтобы добавить данные, мы должны сделать это следующим вызовом функции-члена open:

```
1 ofstream myfile;
2 myfile.open ("example.bin", ios::out | ios::app | ios::binary);
```

Каждая из функций-членов open класса ofstream, ifstream и fstream имеет предопределённый режим, который использован, если файл открыт без второго аргумента:

класс	параметр предопределённого режима
ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

Для ifstream и ofstream классов режимы ios::in и ios::out добавляются автоматически и соответственно присутствуют. Даже если режим ими не включён, он передаётся как второй аргумент в функцию-член open (флаги комбинируются).

Для fstream предопределённое значение применятся, только если функция вызывается без какого-либо заданного значения для параметра режима. Если функция вызывается с каким-то значением в этом параметре, предопределённый режим отменяется, а не комбинируется.

Файловый поток, открытый в *двоичном режиме*, выполняет операции вввода и вывода независимо от любого анализа формата. Не двоичные файлы известны как *текстовые файлы*, и некоторые преобразования могут происходить, благодаря форматированию некоторых специальных символов (подобных символам новой линии и возврата каретки).

Поскольку первая задача, которая выполняется с файловым потоком, это обычно открывание файла, эти три класса включают конструктор, который автоматически вызывает функцию-член open и имеет точно те же параметры, что и этот член. Таким образом, мы должны также декларировать предыдущий myfile объект и сопровождать оператор открывания в нашем предыдущем примере, написав:

```
ofstream myfile ("example.bin", ios::out | ios::app | ios::binary);
```

Комбинация конструктора объекта и потока открывания в одном утверждении. Обе формы для открывания файла правильны и эквивалентны.

Чтобы проверить, будет ли потоковый файл открыт успешно, вы можете сделать это вызовом члена is_open. Эта функция-член возвращает значение bool как true в случае, если действительно потоковый объект связан с открытым файлом, или false в противном случае:

```
if (myfile.is_open()) { /* ok, proceed with output */ }
```

Закрывание файла

Когда мы закончили с операциями ввода и вывода с файлом, мы должны закрыть его так, чтобы операционная система была извещена об этом, и чтобы её ресурсы стали вновь доступны. Для этого мы вызваем потоковую функцию-член `close`. Эта функция-член сбрасывает связанные с процессом буферы и закрывает файл:

```
myfile.close();
```

Когда эта функция-член вызывается, объект потока может быть использован заново для открывания другого файла, а файл становится доступен вновь для открывания в другом процессе.

В случае, когда объект разрушается, оставаясь связан с открытым файлом, деструктор автоматически вызвает функцию-член `close`.

Текстовые файлы

Текстовые потоковые файлы – это те, где `ios::binary` флаг не включён в их режим открывания. Эти файлы разработаны для хранения текста и, следовательно, все значения, которые вводятся или выводятся из/в них, могут претерпевать некоторые преобразования формата, которые не обязательно относятся к их литеральным двоичным значениям.

Запись операций с текстовыми файлами выполняется так же, как мы оперировали с `cout`:

```
// writing on a text file
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile ("example.txt");
  if (myfile.is_open())
  {
    myfile << "This is a line.\n";
    myfile << "This is another
line.\n";
    myfile.close();
  }
  else cout << "Unable to open file";
  return 0;
}
```

```
[file example.txt]
This is a line.
This is another line.
```

Чтение из файла может быть выполено тем же способом, которым мы выполняли `cin`:

```
// reading a text file
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {
  string line;
  ifstream myfile ("example.txt");
  if (myfile.is_open())
  {
    while ( getline (myfile,line) )
    {
```

```
This is a line.
This is another line.
```

```
14        cout << line << '\n';
15      }
16      myfile.close();
17    }
18
19    else cout << "Unable to open file";
20
21    return 0;
22  }
```

Этот последний пример читает текстовый файл и печатает его содержимое на экране. Мы создали цикл while, который читает файл строка за строкой, используя `getline`. Значение, возвращаемое `getline` – это ссылка на сам потоковый объект, который когда исполняется как булево выражение (как в этом цикле *while*) является `true`, если поток готов для выполнения дополнительных операций, и `false`, если либо достигнут конец файла, либо обнаруживается какая-то другая ошибка.

Проверка флагов состояния

Следующие функции-члены существуют для проверки специальных состояний потока (все они возвращают `bool` значение):

`bad()`
: Возвращает `true`, если операция чтения или записи окончилась неудачей. Например, в случае, когда мы пытаемся записать в файл, который не открыт для записи, или если на устройстве, куда мы пытаемся записывать, не осталось места.

`fail()`
: Возвращает `true` в том же случае, что и `bad()`, но также в случае, когда обнаруживается ошибка формата, подобная извлечению алфавитного символа, когда мы пытаемся прочитать целое число.

`eof()`
: Возвращает `true`, если файл, открытый для чтения, достигает конца.

`good()`
: Это самый основной флаг состояния: он возвращает `false` в том же случае, когда вызов любой предыдущей функции вернул бы `true`. Заметьте, что `good` и `bad` не являются полностью противоположными (`good` проверяет больше флагов состояния сразу).

Функция-член `clear()` может быть использована для сброса флагов состояния.

get и put позиционирование потока

Все потоковые объекты ввода/вывода хранят хотя бы одну внутреннюю позицию:

`ifstream` подобно `istream` хранит внутренне *get position (получить позицию)* с местом элемента, который будет прочитан при следующей операции ввода.

`ofstream` подобно `ostream` хранит внутренне *put position (поместить позицию)* с местом, где следующий элемент должен быть записан.

Окончательно, `fstream` хранит и *get* и *put position* подобно `iostream`.

Эти внутренние позиции потока указывают на место в потоке, где следующие операции чтения или записи будут выполняться. Эти позиции могут быть и наблюдаемы, и модифицируемы, если использовать следующие функции-члены:

tellg() и tellp()

Эти две функции-члена без праметров возвращают значение члена типа `streampos`, который есть тип, представляющий текущие *get position* (в случае `tellg`) или *put position* (в случае `tellp`).

seekg() и seekp()

Эти функции позволяют изменить положение *get* и *put positions*. Обе функции перегружены с двумя разными прототипами. Первая форма:

```
seekg ( position );
seekp ( position );
```

Используя этот прототип, указатель потока обращается к своей абсолютной позиции `position` (вычисляемой от начала файла). Тип для этого парметра – это `streampos`, который является тем же типом, что и возвращаемый функциями `tellg` и `tellp`.

Другая форма этих функций есть:

```
seekg ( offset, direction );
seekp ( offset, direction );
```

Используя этот прототип, *get* или *put position* устанавливает значение смещения относительно специальных точек, определяемых параметром `direction`. `offset` имеет тип `streamoff`. А `direction` имеет тип `seekdir`, который оказывается *enumerated type (перечисляемый тип)*, определяющий точку, от которой рассчитывается смещение, и которая может принять любое из следующих значений:

`ios::beg`	offset рассчитывается от начала потока
`ios::cur`	offset рассчитывается от текущей позиции
`ios::end`	offset рассчитывается от конца потока

Следующий пример использует функции-члены, которые мы только что видели, для получения размера файла:

```
// obtaining file size
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  streampos begin,end;
  ifstream myfile ("example.bin",
ios::binary);
  begin = myfile.tellg();
  myfile.seekg (0, ios::end);
  end = myfile.tellg();
  myfile.close();
  cout << "size is: " << (end-begin)
<< " bytes.\n";
  return 0;
}
```

```
size is: 40 bytes.
```

Отметьте тип, который мы использовали для переменных `begin` и `end`:

```
streampos size;
```

`streampos` – это специфический тип, использованный для буфера и позиционирования в файле, и тип, возвращаемый `file.tellg()`. Значения этого типа могут безопасно вычитаться из других значений того же типа, и могут конвертироваться в целый тип, достаточно большой для размещения размера файла.

Эти потоковые функции позиционирования используют два обычных типа: streampos и streamoff. Эти типы также определены, как типы членов потокового класса:

Тип	Тип члена	Описание
streampos	ios::pos_type	Определён как fpos<mbstate_t>. Он может быть конвертирован в/из streamoff, и значения могут быть добавлены или вычтены из значениай этих типов.
streamoff	ios::off_type	Это псевдоним одного из базовых целых типов (таких как int или long long).

Каждый тип члена выше – это псевдоним его эквивалента не члена (они точно те же типы). Не имеет значения, какой используется. Типы членов более универсальны, поскольку они одинаковы для всех потоковых объектов (даже с потоками, использующими экзотические типы символов), но типы не членов широко используются в существующих кодах из исторических соображений.

Двоичные файлы

Для двоичных файлов чтение и запись данных с операторами извлечения и включения (<< и >>) и функциями подобными getline не эффективно, поскольку мы не нуждаемся в форматировании любых данных, и данные, похоже, не сформированы в строки.

Потоковые файлы включают две функции-члена, специально разработанные для чтения и записи двоичных данных последовательно: write и read. Первая (write) – это функция-член ostream (наследованная ofstream). А read – функция-член istream (наследованная ifstream). Объекты класса fstream имеют и то, и другое. Их прототипы:

```
write ( memory_block, size );
read ( memory_block, size );
```

Где memory_block имеют тип char* (указатель на char) и представляют адрес массива байтов, где хранятся прочитываемые элементы данных или откуда берутся элементы данных для записи. Параметр size – это целое значение, которое задаёт число символов для чтения или записи из/в блок памяти.

```
// reading an entire binary file
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  streampos size;
  char * memblock;

  ifstream file ("example.bin",
ios::in|ios::binary|ios::ate);
  if (file.is_open())
  {
    size = file.tellg();
    memblock = new char [size];
    file.seekg (0, ios::beg);
    file.read (memblock, size);
    file.close();

    cout << "the entire file content
is in memory";

    delete[] memblock;
  }
```

```
the entire file content is in memory
```

```
25    else cout << "Unable to open file";
      return 0;
}
```

В этом примере весь файл читается и хранится в блоке памяти. Давайте посмотрим, как это делается:

Вначале файл открывается с флагом `ios::ate`, который означает, что указатель *get* будет позиционироваться в конце файла. Так, когда мы вызываем член `tellg()`, мы непосредственно получим размер файла.

Когда мы получили размер файла, мы запрашиваем место блока памяти достаточно большое для размещения всего файла:

```
memblock = new char[size];
```

Сразу после этого мы продолжим установку *get position* в начало файла (помните, что мы открывали файл с его указателем на конец), затем мы читаем весь файл и, наконец, закрываем его:

```
1 file.seekg (0, ios::beg);
2 file.read (memblock, size);
3 file.close();
```

В этой точке мы можем оперировать с данными, полученными из файла. Но наша программа просто анонсирует, что содержимое файла в памяти, и затем завершается.

Буферы и синхронизация

Когда мы оперируем с потоковыми файлами, есть связь с внутренним буфером объекта типа `streambuf`. Этот объект-буфер может представлять блок памяти, который действует как посредник между потоком и физическим файлом. Например, с `ofstream` каждый раз вызывается функция-член `put` (который пишет единственный символ). Символ может быть вставлен в этот буфер-посредник вместо начала записи непосредственно в физический файл, с которым поток ассоциирован.

Операционная система может также определить другие слои буферизации для чтения и записи файлов.

Когда буфер очищен, все данные, содержащиеся в нём, записываются в физическую среду (если это выходной поток). Этот процесс называется *синхронизацией* и имеет место при любых следующих обстоятельствах:

- **Когда файл закрывается:** перед закрыванием файла все буферы, которые не были очищены, синхронизируются и все отложенные данные записываются или читаются в физической среде.
- **Когда буфер полон:** буферы имеют некоторый размер. Когда буфер полон, он автоматически синхронизируется.
- **Явно, с манипуляторами:** когда некоторые манипуляторы используются с потоками, имеет место явная синхронизация. Эти манипуляторы: `flush` и `endl`.
- **Явно, с функцией-членом sync():** вызывая потоковую функцию-член `sync()`, приводящую к прямой синхронизации. Эта функция возвращает значение `int` эквивалентное `-1`, если поток не имеет связанного буфера или в случае неудачи. Иначе (если буфер потока был успешно синхронизирован) она возвращает `0`.

Заключение или Qt злоключения

Я, по меньшей мере, дважды внимательно прочитал рассказ о языке C++: когда переводил и когда убирал многочисленные опечатки (их не должно быть слишком много). Эффект не заставил себя ждать – появилось ощущение, что мне всё стало понятно.

Бороться с подобным ощущением можно только одним способом – начать что-то делать. При переводе я использовал программу Code::Blocks в консольном варианте. Она позволяет создавать и приложения для Windows. Но далее, чем создание рабочего окна, которое я получил в готовом виде, я не продвинулся – первый звонок, что не всё так благополучно.

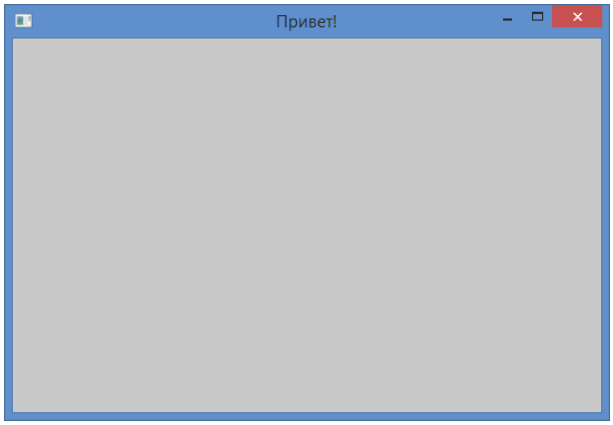


Рис. 8. Рабочее поле программы, полученное в Code::Blocks

Однако я, как и многие, помню серию программ для создания приложений от Microsoft, где многое можно сделать достаточно просто. И я не считаю зазорным воспользоваться удобной для выполнения задачи средой разработки. Правда, я хочу использовать свободную программу Qt Creator, входящую в состав Qt (сейчас версия 5.4).

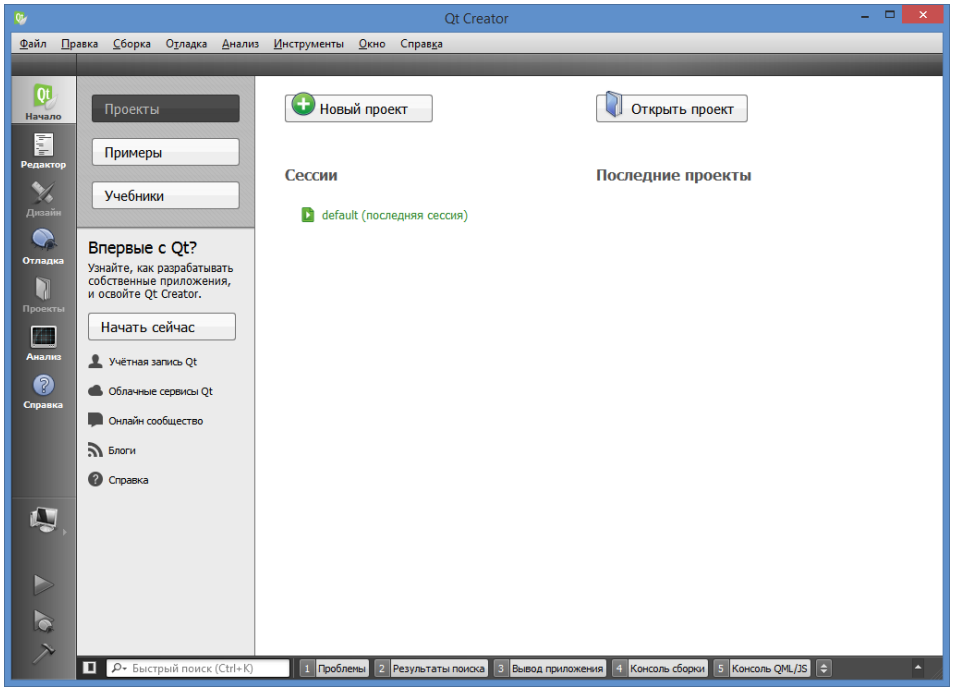


Рис. 9. Первый запуск программы Qt Creator

При создании нового проекта в диалоговых окнах можно выбрать тип приложения:

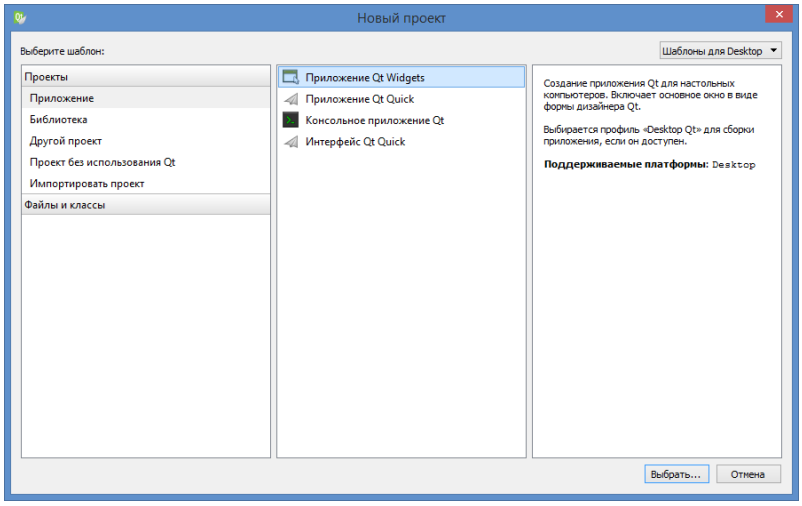


Рис. 10. Выбор типа приложения

В следующих диалогах задаются имя проекта, место его размещения и ряд дополнительных свойств проекта. Завершив эти настройки, мы получим заготовку будущей программы.

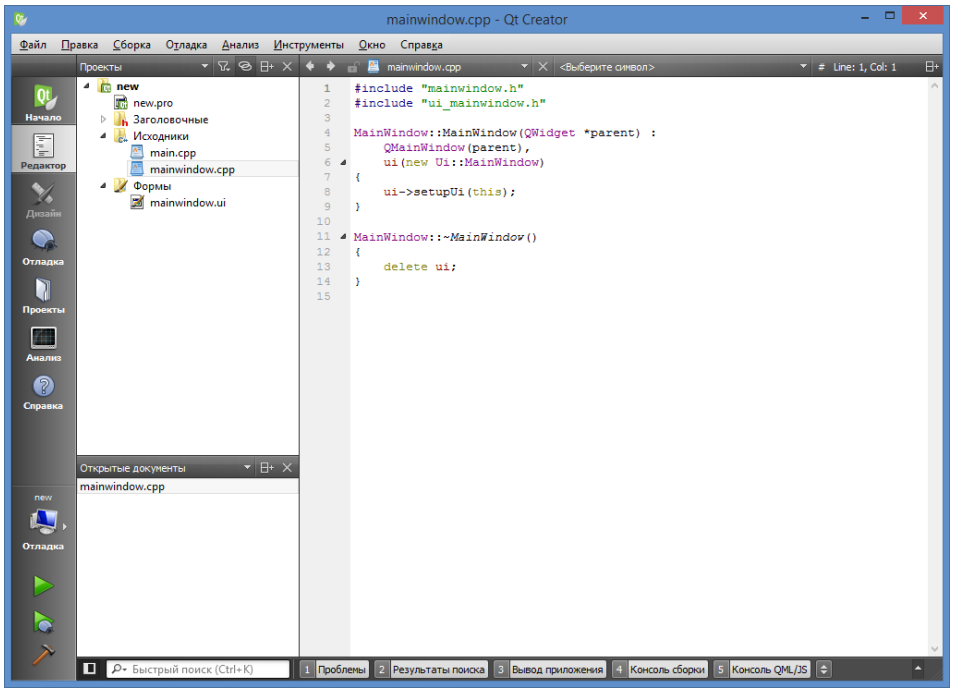


Рис. 11. Заготовка будущей программы

Меня, как человека падкого на «всё готовенькое», привлекла готовая форма (mainwindow.ui), двойной щелчок по этому файлу открывает конструктор формы.

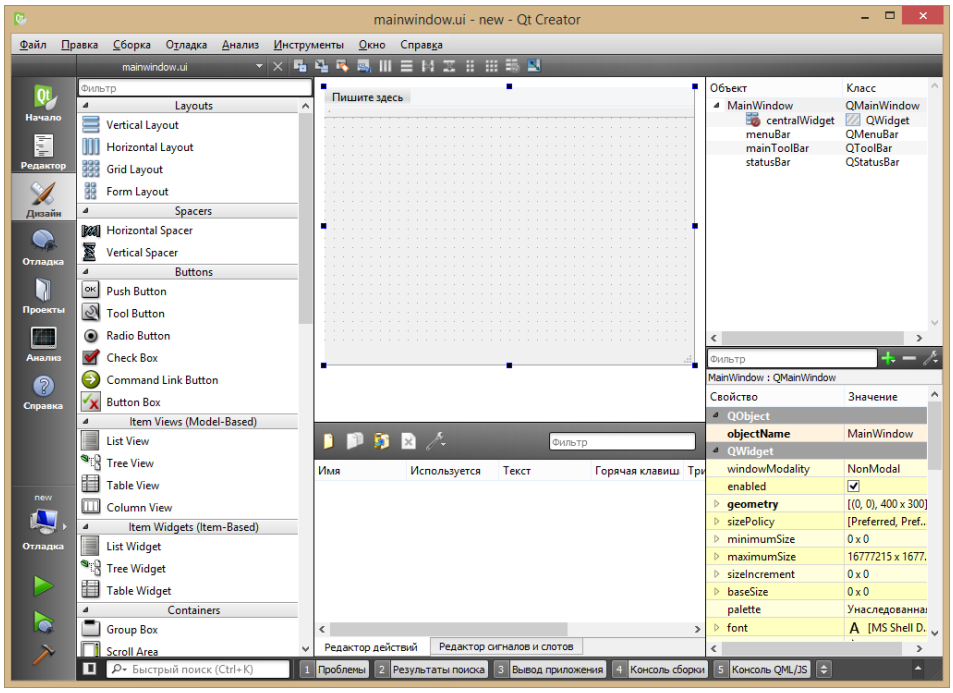


Рис. 12. Конструктор формы Qt Creator

Инструментальная панель, вы можете убедиться сами, предлагает большой набор необходимых для формы компонентов.

Я не умею создавать просто электронные устройства, как не умею создавать просто программы. Мне нужно точно знать, каким должно быть устройство, что и как оно должно делать, то же относится и к программе. Поэтому первое, что следует сделать – это записать, что должна делать программа: после запуска программы она должна позволить обратиться к COM-порту и отправить, скажем, одну букву через порт. Позже, если всё получится, эту программу можно будет расширить. Идея программы в том, что с её помощью (и помощью дополнительного устройства) можно будет управлять с компьютера, например, простым роботом, заставляя его двигаться, поворачивать, останавливаться. У моего компьютера есть один COM-порт, но я хотел бы, чтобы программа работала и на моём планшете, где нет COM-порта, но можно использовать переход USB-COM. На планшете у меня операционная система Windows, так что мне проще использовать одну и ту же программу в двух случаях. А дополнительное устройство, подключаемое к COM-порту, может быть преобразователем команд от компьютера в радиокоманду или, что проще, ИК-команду.

Первое, что нужно сделать – это добавить на форму три кнопки (button). Сделать это не сложно – подцепить мышкой Push Button, перенести на форму и оставить там.

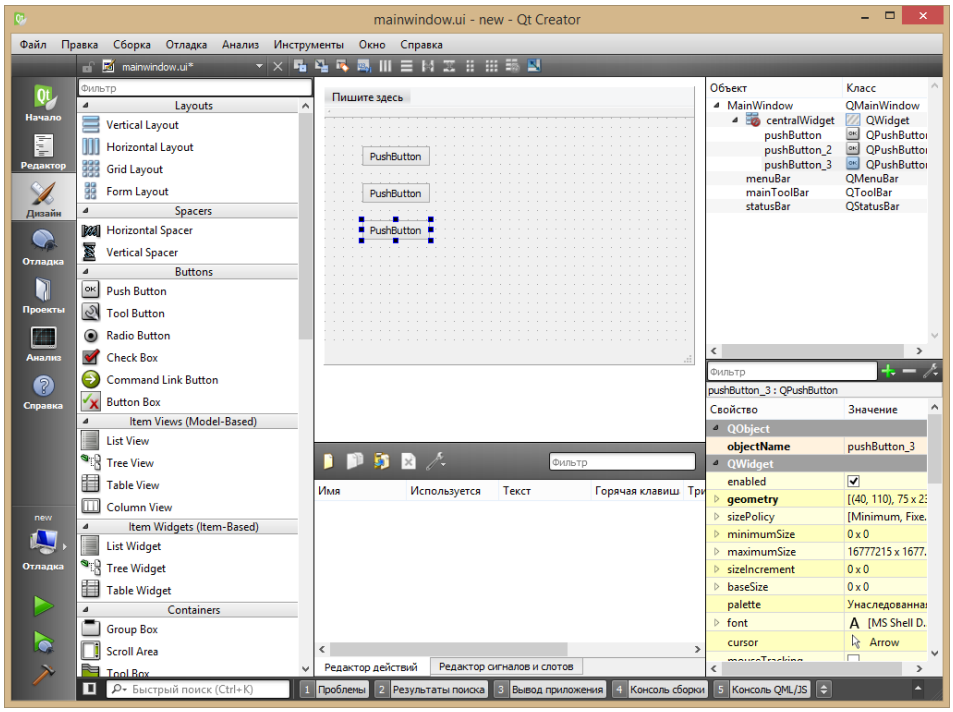


Рис. 13. Заполнение формы

Верхняя кнопка, я назову её «Открыть порт», предназначена для связи программы с портом, следующая пошлёт команду, а последняя закроет порт. По щелчку верхней кнопки следует открыть и настроить COM-порт. В этом месте возникает некоторое затруднение, связанное с воспоминанием о том, как это можно было сделать в Visual Basic. Но помогает и воспоминание о том, что Qt (если это было Qt) поддерживает конструкцию слотов и сигналов. Щёлкнув правой кнопкой мыши по кнопке на панели, можно найти в выпадающем меню:

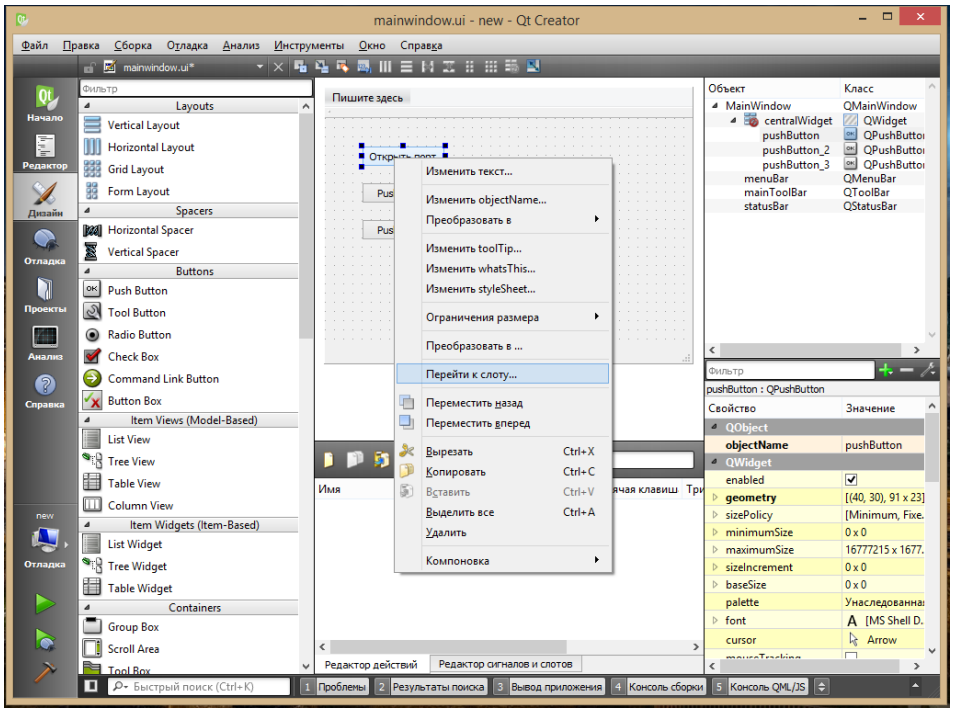


Рис. 14. Переход к сигналам слота

В сигналах слота есть нужный мне сигнал щелчка:

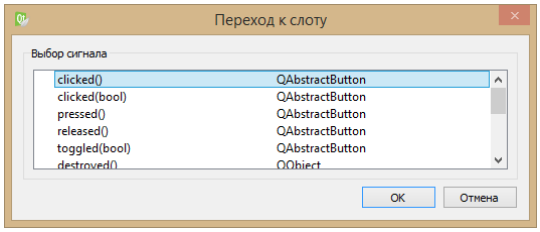


Рис. 15. Добавление в программу обработки щелчка по кнопке

Кнопка **ОК** диалогового окна добавит фрагмент кода в программу:

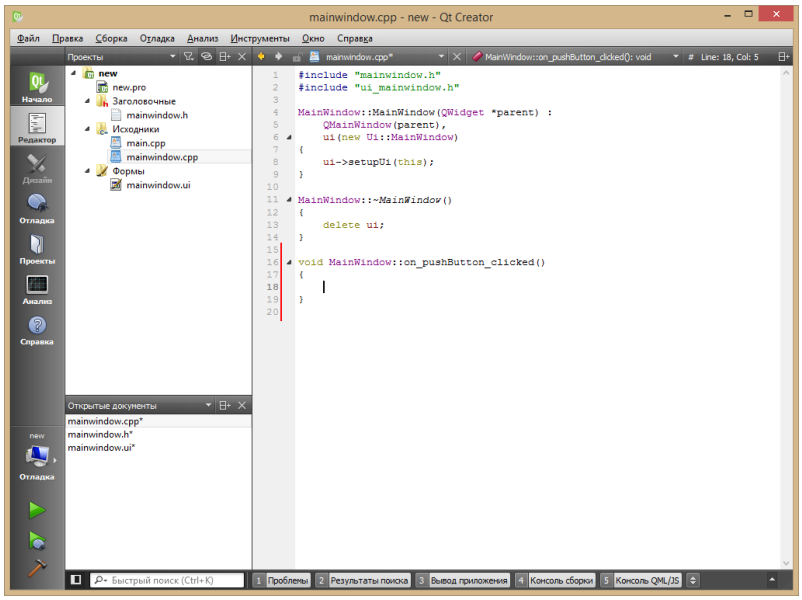


Рис. 16. Автоматическое добавление кода в программу

Согласитесь, что очень удобно, особенно начинающему, так легко создавать нужную форму и добавлять нужные элементы. Но что дальше?

Пока я только (после прочтения руководства) подозреваю, что следует отыскать подходящий класс компонентов. И он есть. Называется он QtSerialPort:

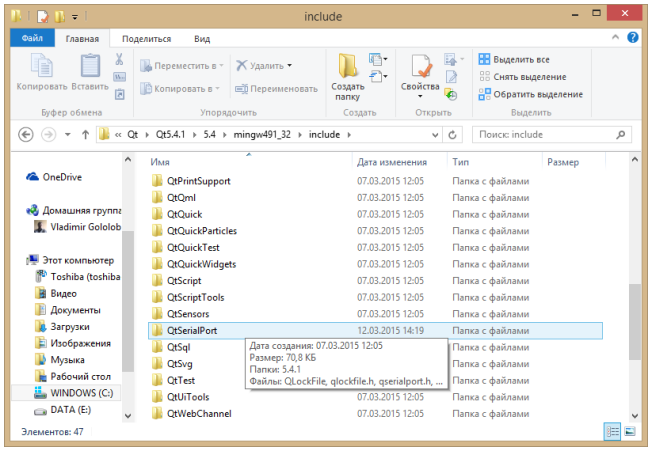


Рис. 17. Последовательный порт в папке include программы Qt

Добавление этого класса элементов происходит ключевым словом #include (это я тоже помню), и я думаю, что следует указать файл заголовка. Кроме того, я помню из переведённого руководства, что класс – это тип для объекта. Поэтому следует объявить объект. Я назову его serial. И попробую задать скорость обмена через COM-порт, используя справку по классу QtSerialPort. Теперь программа принимает вид:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <QtSerialPort/qserialport.h>

QSerialPort serial;

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::on_pushButton_clicked()
{
 serial.BaudRate(Baud9600);
}
```

Но программа Qt Creator при попытке собрать всё (Сборка->Собрать всё), останавливает меня:

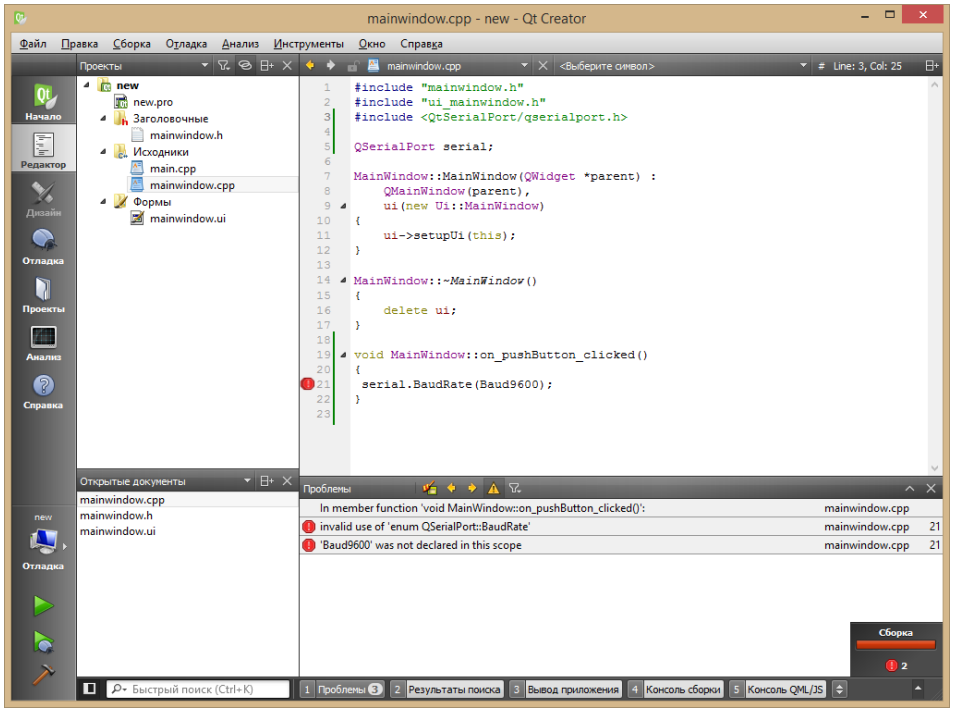


Рис. 18. Первая ошибка

В сообщении об ошибке говорится о неправильном использовании функции-члена класса. Пожалуй, добавим слово установить (set), и пересоберём проект.

Ошибка осталась, теперь речь идёт о том, что сообщалось во второй строке – неверные границы. Что ж, укажем границы. И я хочу добавить имя порта:

```
serial.setPortName("COM1");
serial.setBaudRate(QSerialPort::Baud9600);
```

Однако при попытке собрать проект вновь появляются ошибки, которые мне, признаться, не понятны.

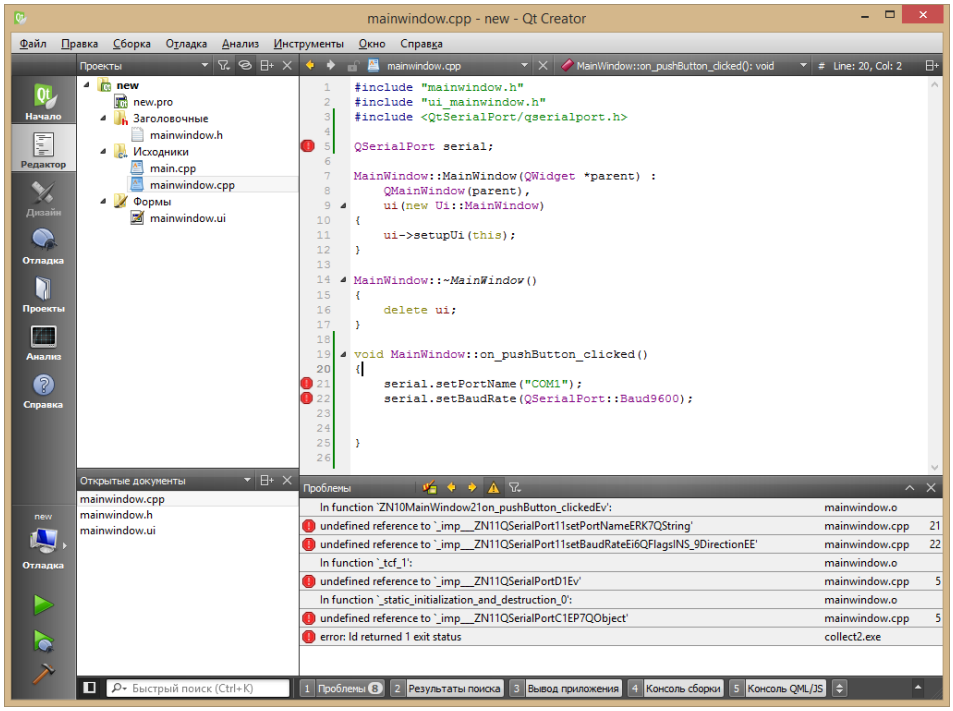


Рис. 19. Появление новых ошибок

Поиск ответа на вопрос, в чём я был неправ (я поискал его в Интернете), приводит к ответу: в файле проекта с расширением *pro* (у меня это файл new.pro) следует добавить одну строку. Вот, как выглядит эта строка, добавленная к предыдущей:

```
QT       += core gui
QT       += serialport
```

Добавленную строку я выделил цветом. Теперь сборка проходит, и можно запустить её в режиме отладки. Но перед этим я хочу добавить остальные настройки порта и команду его открытия. Кроме этого я хочу проверить, открыт ли порт, и если это так, изменить текст на кнопке:

```
void MainWindow::on_pushButton_clicked()
{
serial.setPortName("COM1");
serial.setBaudRate(QSerialPort::Baud9600);
serial.setDataBits(QSerialPort::Data8);
```

```
serial.setParity(QSerialPort::NoParity);
serial.setStopBits(QSerialPort::OneStop);
serial.setFlowControl(QSerialPort::NoFlowControl);
serial.open(QSerialPort::ReadWrite);

if(serial.isOpen()== true)
{
    ui->pushButton->setText("Порт открыт");
}
}
```

Обратите внимание на то, какими командами изменяется текст (это я попутно нашёл в Интернете в предыдущем поиске). И, наконец, чтобы не оставлять порт открытым, добавлю щелчок по третьей кнопке, который должен закрыть порт.

```
void MainWindow::on_pushButton_3_clicked()
{
  serial.close();

  if(serial.isOpen()== false)
  {
      ui->pushButton_3->setText("Порт закрыт");
  }
}
```

Теперь можно проверить, как выглядит программа, если её запустить.

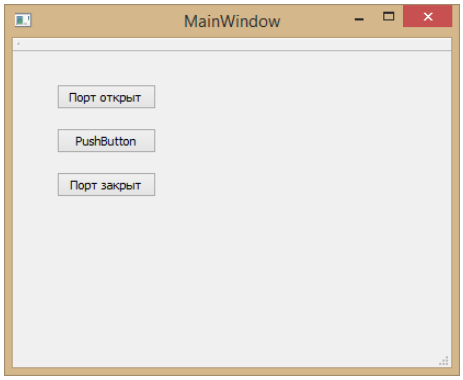


Рис. 20. Первая проверка работы программы

Осталось добавить команду отправки символа команды:

```
void MainWindow::on_pushButton_2_clicked()
{
    serial.putChar(0x41);
}
```

Сообщений об ошибках нет, но я не знаю, отправляется ли латинская буква «A» в порт? Наверное, есть способы проверить это, используя только компьютер, но мне проще подключить приставку-осциллограф, ради которой я, собственно, и покупал планшет с Windows, и посмотреть, доходит ли команда до порта?

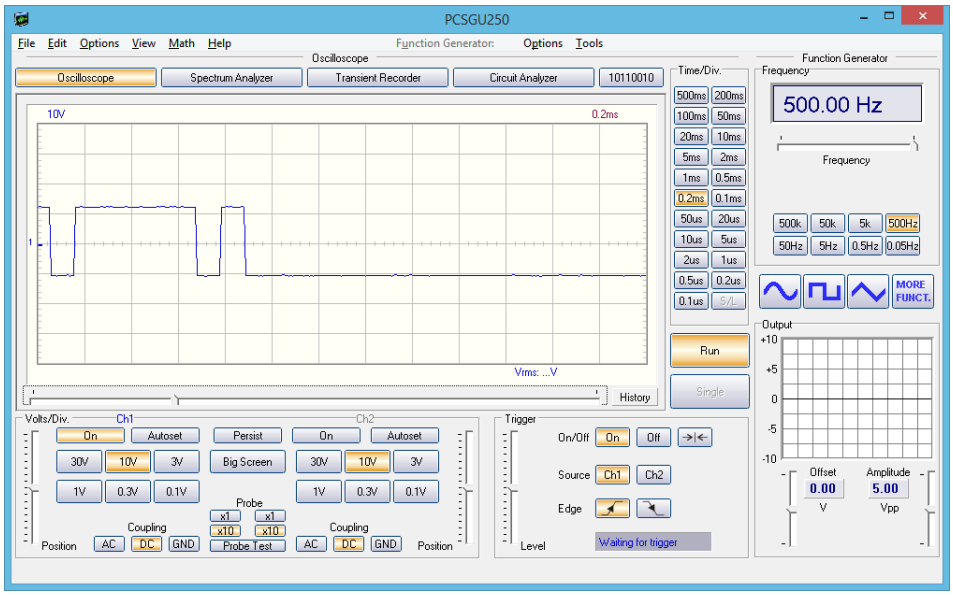


Рис. 21. Сигнал на выходе COM-порта при нажатии на кнопку **Команда**

Итак, ошибаясь и спотыкаясь, я создал первую программу «с картинками». Можно было бы вздохнуть с облегчением, когда бы не очередное «но»! Программа для переноса на другой компьютер должна быть исполняемой.

Для этого есть второй режим работы – первый был режим отладки.

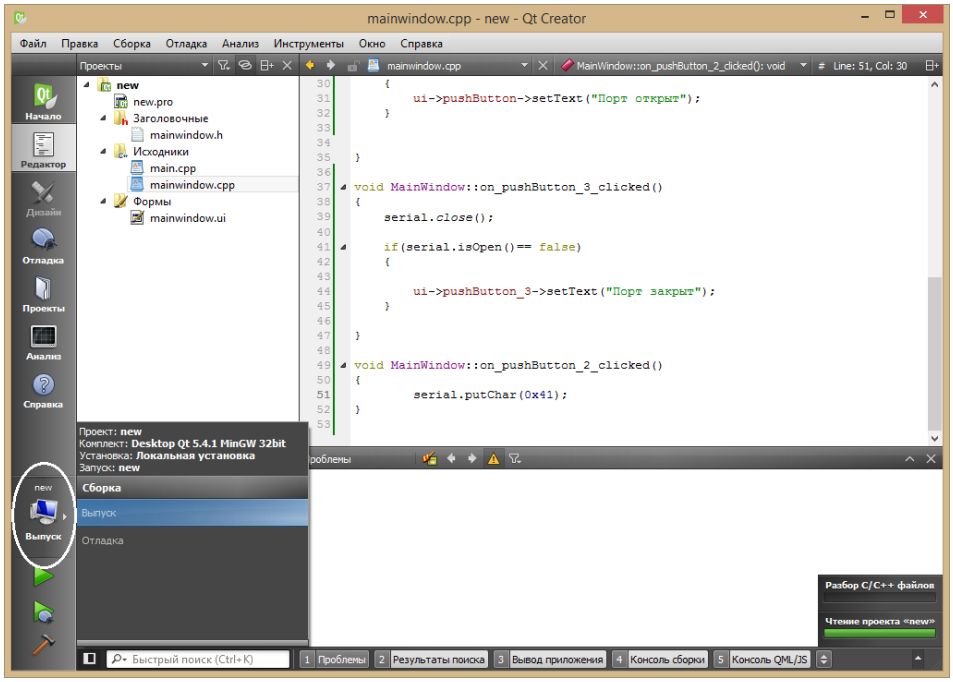


Рис. 22. Переход в режим создания исполняемого файла

Переключив режим (кнопка на инструментальной панели отмечена на рисунке), следует повторить сборку. После этого появится папка с именем проекта и пометкой Releas там, где хранятся папки проектов:

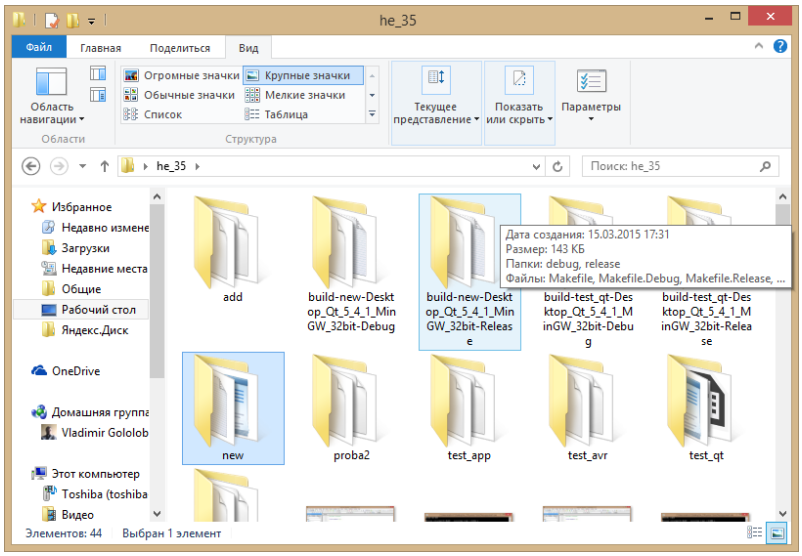


Рис. 23. Папки с проектами

В этой папке находится папка releas с исполняемым файлом проекта. Но радоваться рано. Попробуем запустить файл new.exe.

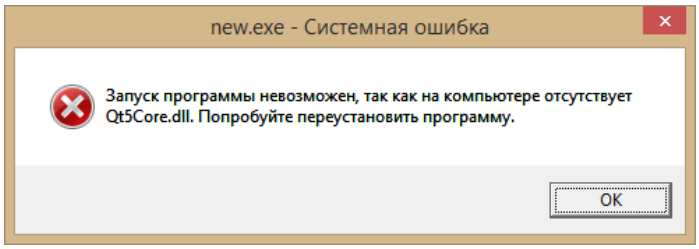


Рис. 24. Сообщение об ошибке при автономном запуске программы

В папке исполняемого файла можно удалить все остальные файлы, но нужно будет добавить ряд файлов dll, о чём гласит сообщение (упоминая только первый файл). Почти все файлы можно будет найти в соответствующей папке программы Qt.

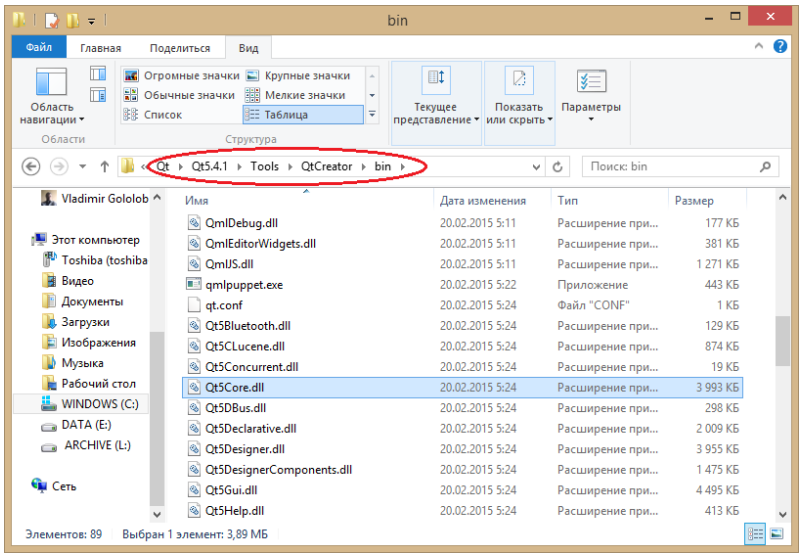


Рис. 25. Место, где находятся многие необходимые файлы

Можно сделать так – добавить файл, запустить программу, найти следующий файл и добавить его. Вот перечень файлов, которые на этом этапе следует добавить:

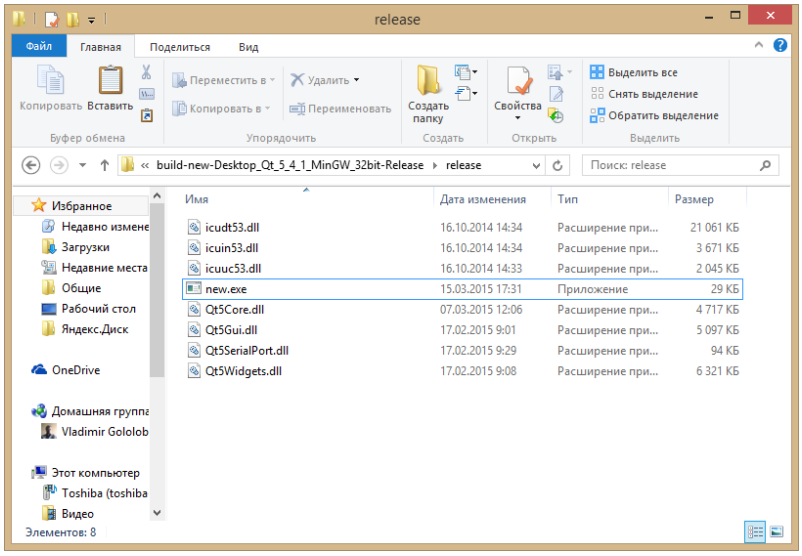


Рис. 26. Набор файлов из Qt

С набором этих файлов заканчивается ясность в том, какие файлы следует добавить. Следующие сообщения теряют ту прямоту, которой отличались предыдущие. Например, такое сообщение:

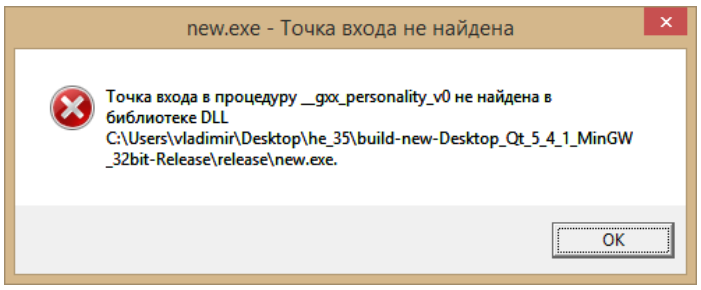


Рис. 27. Сообщение о потерянной точке входа

Проблема в том, что следует добавить ещё ряд файлов. Вот тот набор (я не исключаю, что есть и лишние файлы). Кроме того, в папке *plugins* есть папка *platforms*, в которой два файла:

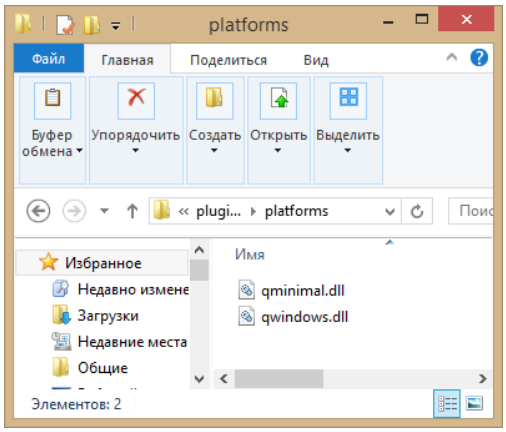


Рис. 28. Содержимое папки platforms

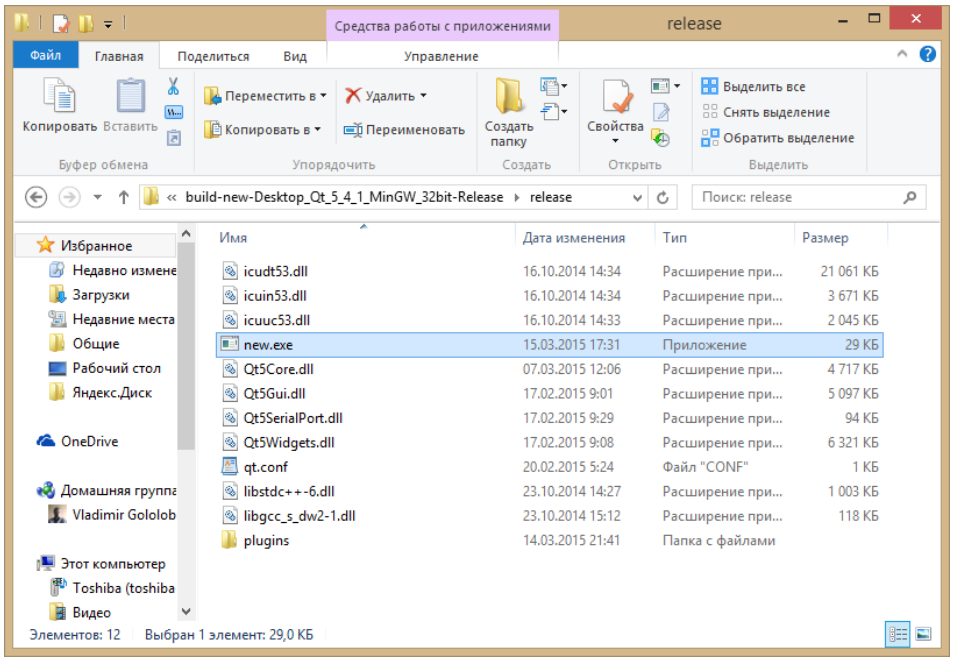


Рис. 29. Вид исполняемого файла с набором дополнительных файлов

Теперь программу можно собрать в отдельную папку, которую можно перенести на другой компьютер. Правда, когда я перенёс её на планшет, потребовалось найти ещё один файл, который я добавил к набору. Весь набор файлов из-за библиотеки стал тяжеловесом, но программа работает и на планшете.

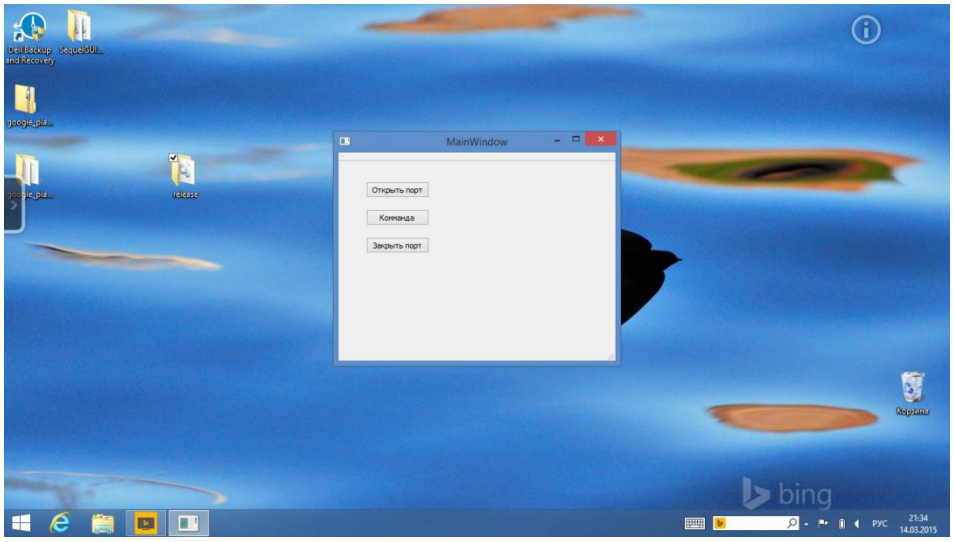


Рис. 30. Работа программы на планшете

У планшета нет COM-порта, таким образом, называть программу работающей на планшете преждевременно, но она запускается и живёт, что от неё пока и требуется.

Для меня этот опыт был очень полезен: мало того, что ты когда-то читал; мало того, что ты прочитал толковое руководство по языку программирования. Нужно каждый день создавать свои программы, чтобы когда-то научиться программировать без подсказок и спотыкания на каждом шагу. Но и подсказки, я считаю, полезны. Особенно в первое время.

P.S. Возвращаясь к Code::Blocks

Не знаю, невезение это или неумение, но первый опыт не задался. Впрочем, по порядку.

Программа Code::Blocks позволяет работать с графической частью создаваемой программы подобно Qt Creator. Достаточно установить wxWidgets (я предпочёл установочный вариант, wxMSW-Setup-3.0.0.exe (Windows Installer), а при создании проекта указать wxWidgets project.

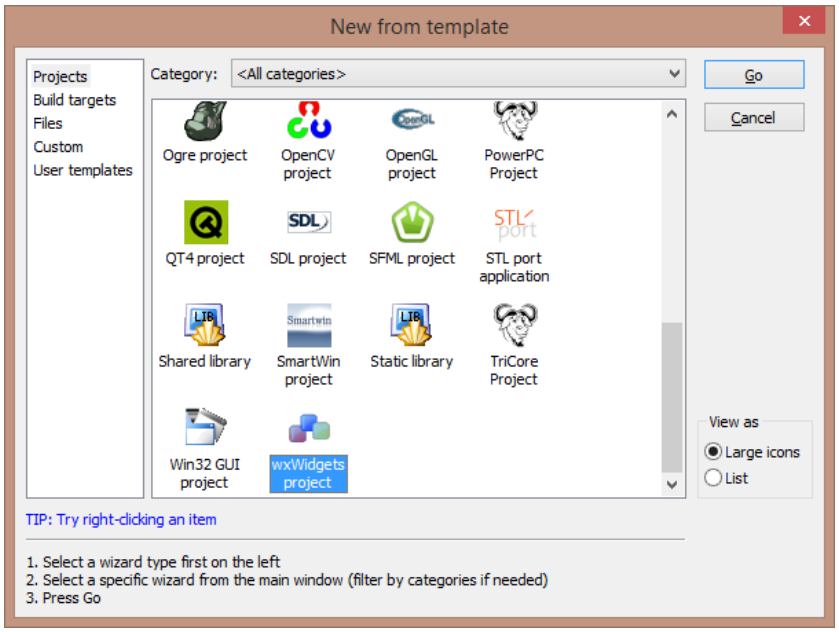


Рис. 31. Создание проекта с графическим построением интерфейса

Помощник создания нового проекта предлагает много настроек, с которыми на первых порах не всё ясно, но в итоге можно получить следующее:

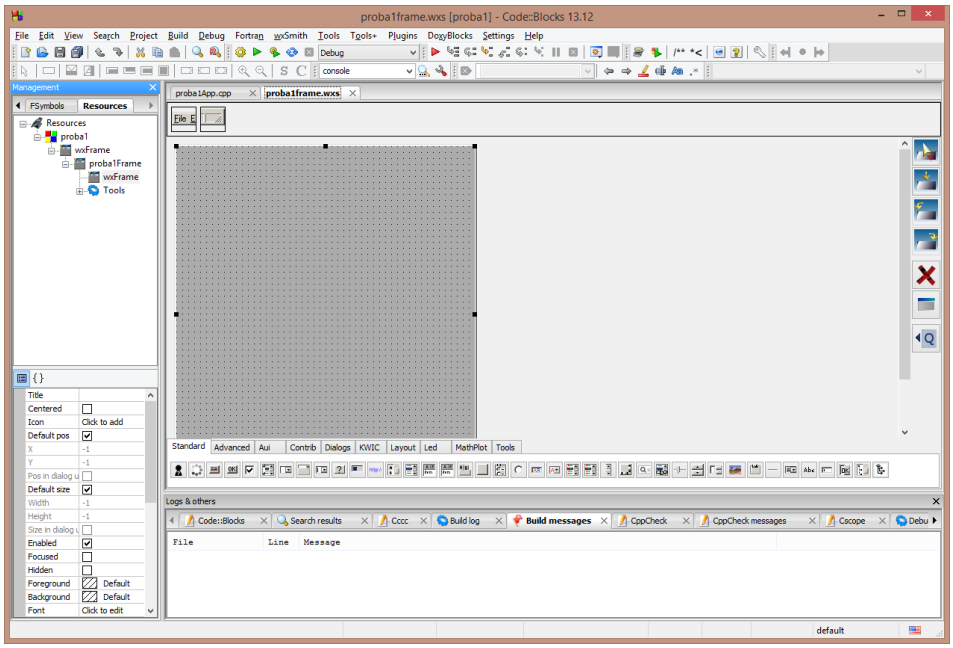


Рис. 32. Работа с формой в Code::Blocks

На форме можно размещать компоненты, которые вы видите в нижней части рабочего поля на нескольких закладках.

Первое, что я пробую сделать – это, используя команду *Build*, запустить сборку формы в режиме отладки.

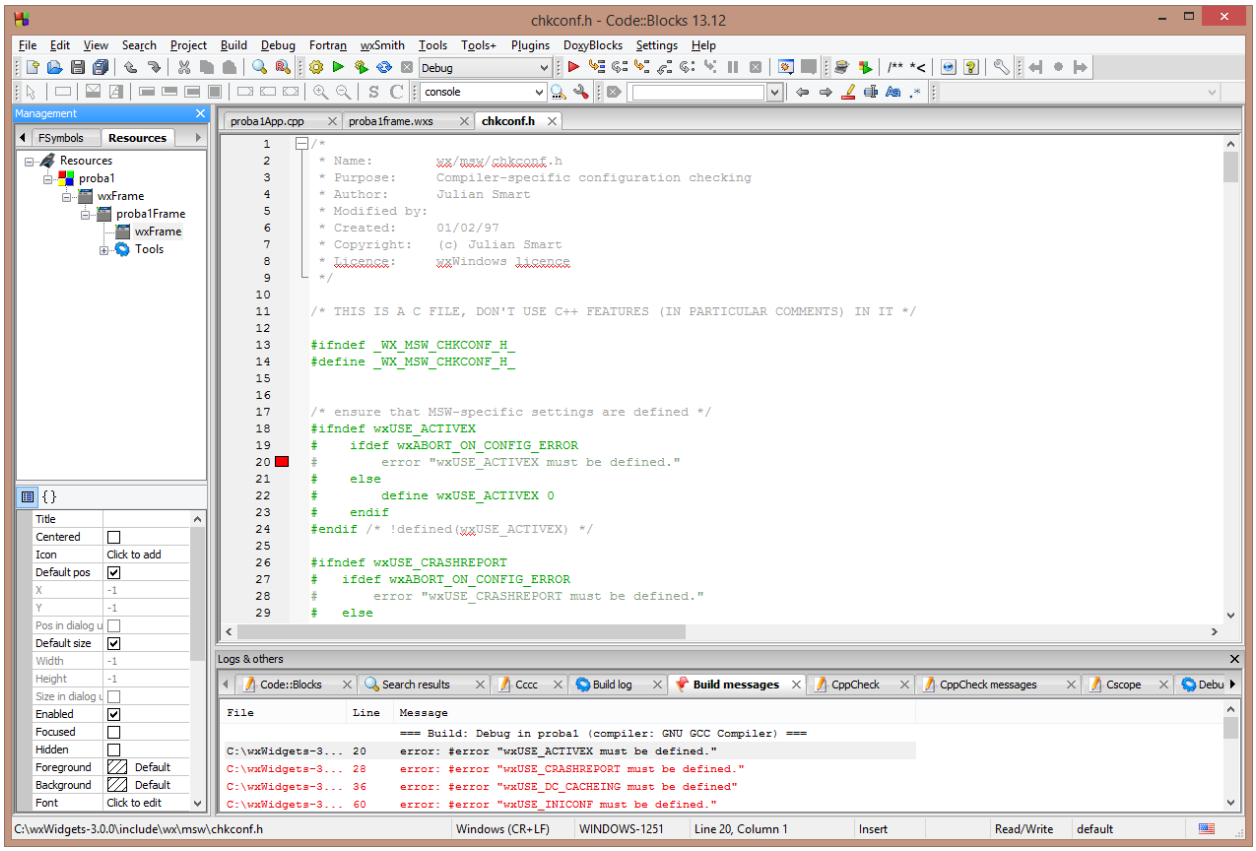


Рис. 33. Появление ошибок при попытке собрать программу и запустить

Сборка завершается многочисленными ошибками, и не создаёт работающую программу. Без единой строки добавленного кода, без добавления каких-либо графических элементов…

Я не ожидал, сознаюсь, подобного эффекта. И мне не хотелось бы об этом говорить, и не собирался я эту часть включать в рассказ, но из песни слова не выкинешь, вдобавок не люблю я недомолвок.

Опуская, как в старом анекдоте, десяток страниц с «цок-цок-цок» по случаю дальней дороги в поисках ответа, в разделе настроек сборки я обратил внимание на следующее:

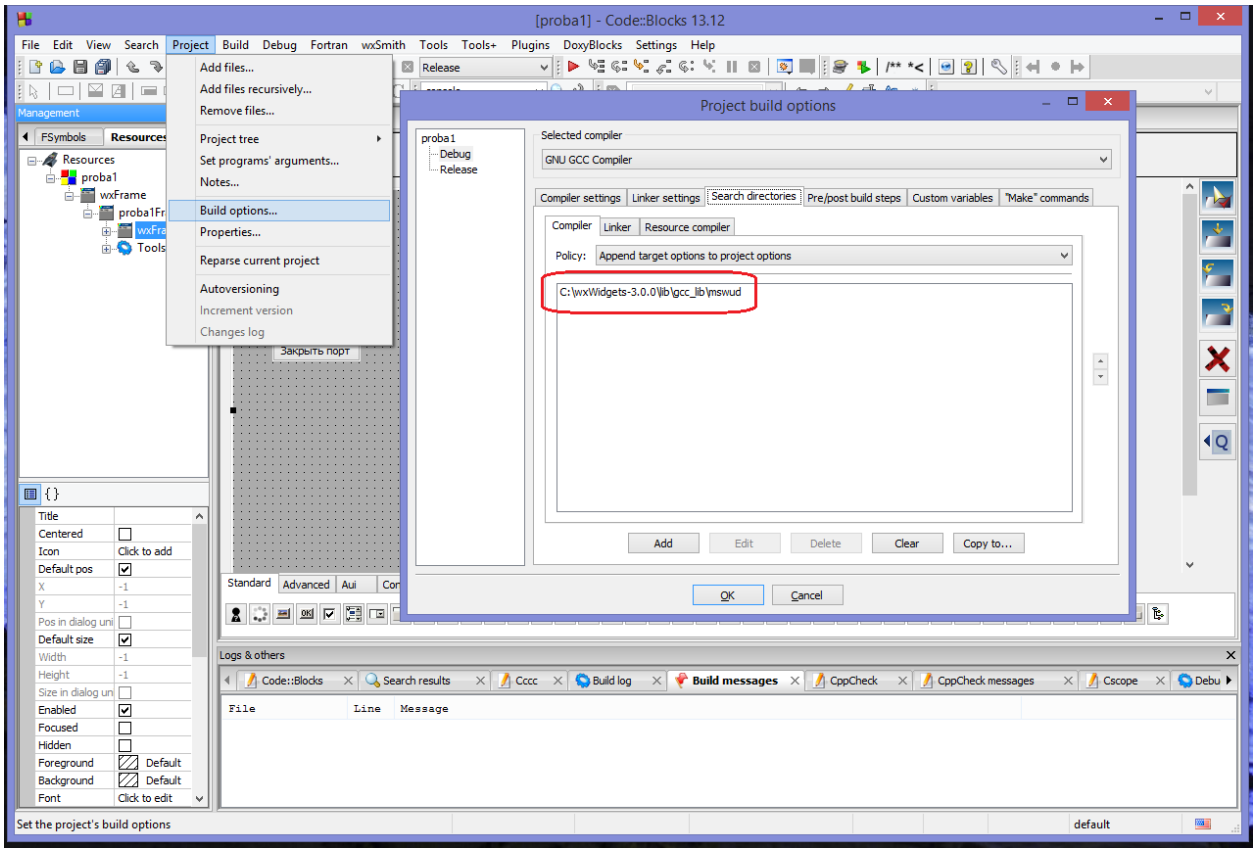


Рис. 34. Раздел настроек сборки проекта

В папке библиотек нет указанной библиотеки gcc_lib. Что справедливо, эту библиотеку предстоит создать. На одном из сайтов: http://habrahabr.ru/post/212027/ , я нашёл набор команд для этой процедуры.

```
cd %WXWIN%\build\msw
mingw32-make -f makefile.gcc clean
mingw32-make -f makefile.gcc BUILD=debug SHARED=0 MONOLITHIC=0 UNICODE=1
WXUNIV=0
mingw32-make -f makefile.gcc BUILD=release SHARED=0 MONOLITHIC=0 UNICODE=1
WXUNIV=0
```

Создание нужной библиотеки производится командами в командной строке операционной системы. Первая команда относится к переходу в папку сборки. Здесь %WXWIN% заменяет путь к этой сборке, если создать эту переменную окружения. Если этого не сделать, можно использовать полный путь.

Перед тем, как использовать вторую команду, автор статьи рекомендует проверить, будет ли работать команда, заменив её такой: mingw32-make –v. Эта команда, если всё хорошо, выведет версию сборки. В том случае, если в ответ вы получаете сообщение об отсутствии файла или команды, следует задать путь к MinGW в системных переменных. У меня в ОС Windows 8.1 к нужным настройкам можно добраться, используя правую выпадающую панель, раздел «Параметры» и пункт «Сведения о компьютере». В открывшемся диалоговом окне справа внизу есть предложение «Изменить параметры». Новое окно имеет ряд закладок, а на закладке «*Дополнительно*» есть кнопка **Переменные среды**.

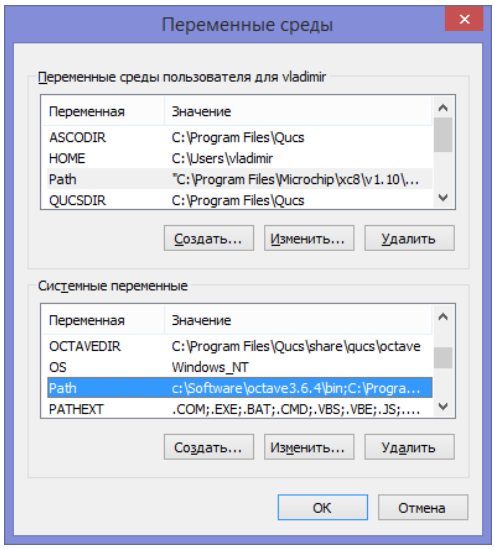


Рис. 35. Добавление пути к исполняемым файлам MinGW

Нажав кнопку **Изменить…**, нужно в конец списка добавить:

```
C:\Program Files\CodeBlocks\MinGW\bin
```

Но это ещё не всё. Хотя вторая команда работает, следующая даёт большое количество ошибок, но не приводит к должному результату. То, как следует поступить, я нашёл на сайте:

https://wiki.wxwidgets.org/Compiling_wxWidgets_with_MinGW

Для меня набор команд превратился в следующий:

```
cd C:\wxWidgets-3.0.0\build\msw
mingw32-make -f makefile.gcc clean
mingw32-make SHELL=CMD.exe -j4 -f makefile.gcc BUILD=debug SHARED=0
MONOLITHIC=0 UNICODE=1 WXUNIV=0
mingw32-make SHELL=CMD.exe -j4 -f makefile.gcc BUILD=release SHARED=0
MONOLITHIC=0 UNICODE=1 WXUNIV=0
```

Здесь –j4 не обязательный, видимо, параметр, но у меня процессор с 4 ядрами, что и указывает этот параметр. Выполнение создания сборки для отладки (debug) занимает достаточно много времени. Я повторил эту операцию несколько раз. Проблема была в отсутствии одного из файлов, без которого операция сборки формы не выполнялась:

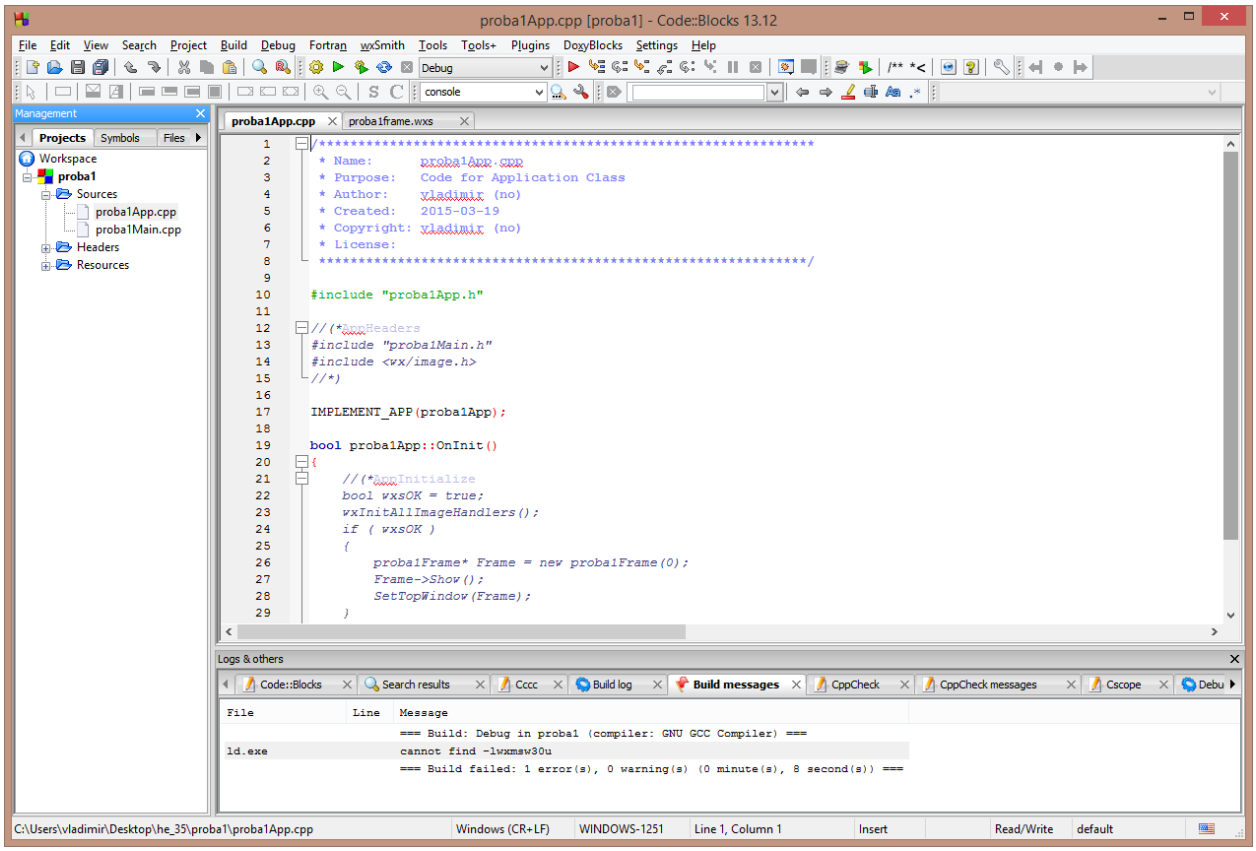


Рис. 36. Сообщение об отсутствии нужного файла

Решение этой проблемы, как мне кажется, удивило не только меня, но и разработчиков. Нашёл я его тоже в Интернете. И суть его в том, что в полученных файлах есть два, имена которых следует поправить: gcc_mswud\corelib_tipwin.o и gcc_mswud\corelib_tipwin.o.d, удалив из имени букву «t». Вот, как должны выглядеть эти файлы:

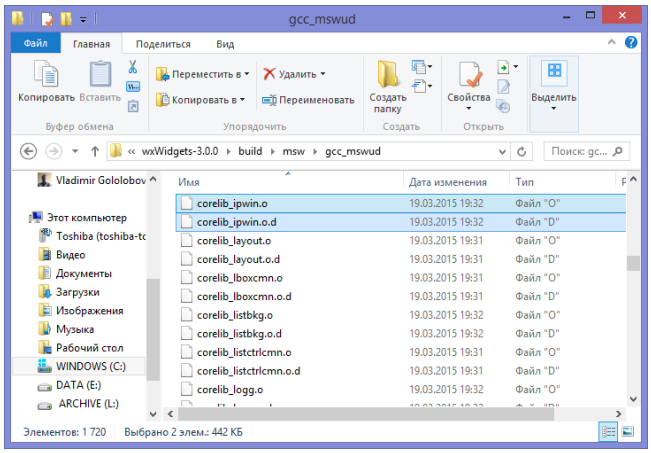


Рис. 37. Местонахождение и исправленные имена файлов

После этой процедуры повторное выполнение команды сборки отладочного набора файлов приводит к успеху. Чему обязано существование этого глюка, я не знаю, но теперь всё работает, что и требовалось. Повторяя процедуру создания нового проекта, я, признаться, сделал ещё одну ошибку – я стараюсь не менять те установки, назначение которых не понимаю. Поэтому в процедуре пропустил выбор дополнительных библиотек, столкнувшись с появлением новых

ошибок при сборке. Повторив создание проекта, я не стал придумывать что-то, только повторил указания автора статьи.

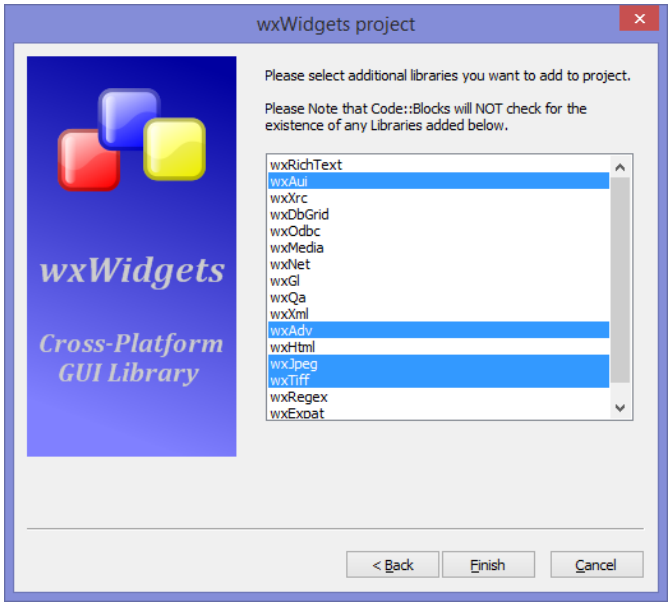


Рис. 38. Добавление библиотек в процедуре создания проекта

Теперь и отладочная сборка, и сборка исполняемого файла проходят без ошибок.

Полученный исполняемый файл не требует, как в предыдущем случае дополнительных файлов, и он меньше по размеру. Его достаточно запустить, как любой исполняемый файл, чтобы получить:

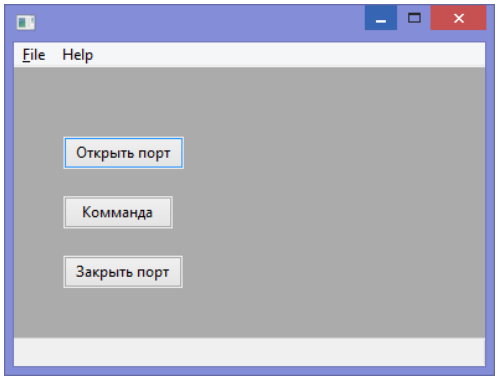


Рис. 39. Работающая программа

Программу можно перенести на другой компьютер, где она тоже работает (есть проблема с надписью на русском, но, видимо, решаемая):

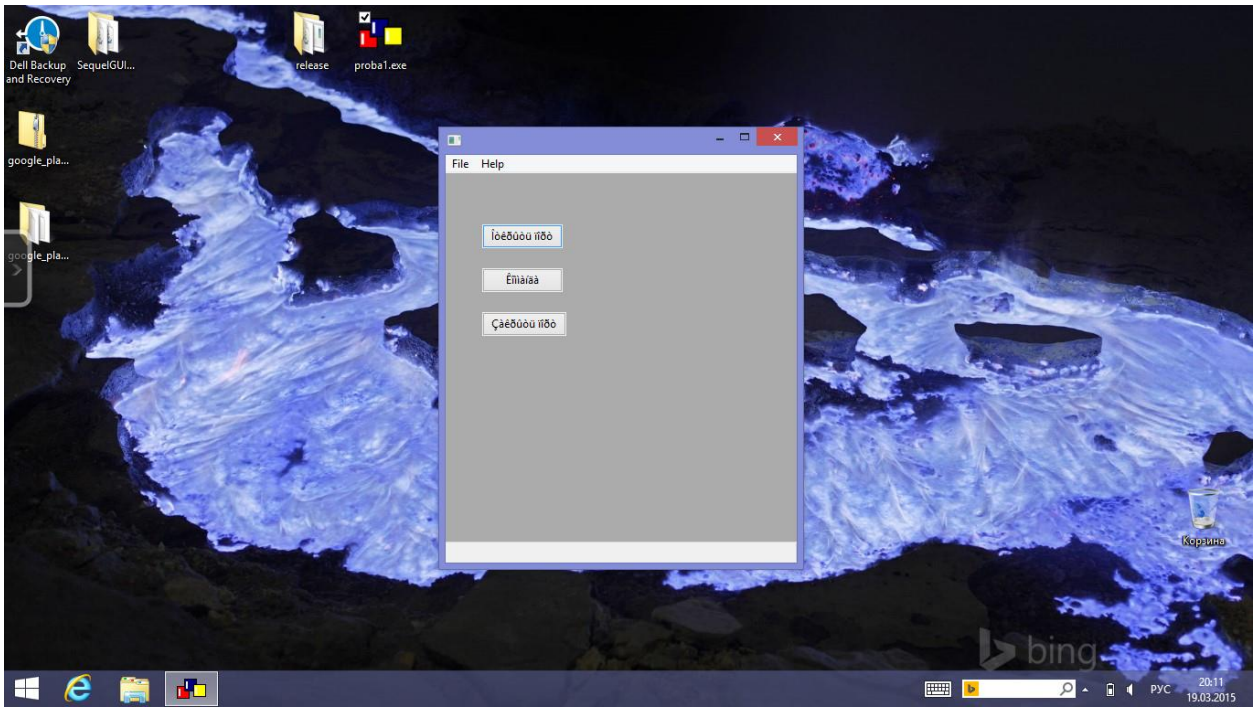


Рис. 40. Работа программы на планшете

Чтобы завершить рассказ, хотелось бы повторить работу с COM-портом, как это было сделано раньше. Но wxWidgets, похоже, работает только с графической стороной программы. Самый простой способ решить проблему – воспользоваться чужой наработкой. Я нашёл необходимое: http://www.codeproject.com/Articles/814411/Using-some-resources-timers-and-COM-ports-in-wxWid

Фраза в тексте: Simple-Serial-communication-utilities, позволяет перейти к загрузке, где можно загрузить: Download rs232 - 3.9 KB

Два файла из разархивированной папки rs232 с именами rs232.c и rs232.h я переношу в папку со своим проектом (слукавив, переименовав первый файл в rs232.cpp). Чтобы добавить эти файлы в проект, достаточно обратиться к разделу *Project->Add Files…* После добавления этих файлов в проект можно использовать новые возможности.

В Code::Blocs для обработки нажатия кнопки достаточно дважды щёлкнуть по кнопке, чтобы получить:

```
void proba1Frame::OnButton1Click(wxCommandEvent& event)
{
}
```

В фигурные скобки осталось записать нужные действия. Три щелчка по трём кнопкам позволяют записать:

```
void proba1Frame::OnButton1Click(wxCommandEvent& event)
{
    RS232_OpenComport(0, 9600);
    Button1 -> SetLabel("Порт открыт");
}

void proba1Frame::OnButton3Click(wxCommandEvent& event)
```

```
{
   RS232_CloseComport(0);
   Button3 -> SetLabel("Порт закрыт");
}

void proba1Frame::OnButton2Click(wxCommandEvent& event)
{
    RS232_SendByte(0, 'A');
    Button2 -> SetLabel("Отправлено");
}
```

Здесь при обращении к COM1 используется его индекс 0. На всякий случай я приведу оба текста файлов, если источник по каким-то причинам окажется недоступен.

rs232.h

```
/*
**************************************************************************
*
* Author: Teunis van Beelen
*
* Copyright (C) 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014
Teunis van Beelen
*
* teuniz@gmail.com
*
**************************************************************************
*
* This program is free software; you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published by
* the Free Software Foundation version 2 of the License.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License along
* with this program; if not, write to the Free Software Foundation, Inc.,
* 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
*
**************************************************************************
*
* This version of GPL is at http://www.gnu.org/licenses/old-licenses/gpl-
2.0.txt
*
**************************************************************************
*/

/* last revision: Januari 31, 2014 */

/* For more info and how to use this libray, visit: http://www.teuniz.net/RS-
232/ */


#ifndef rs232_INCLUDED
#define rs232_INCLUDED

#ifdef __cplusplus
extern "C" {
```

```
#endif

#include <stdio.h>
#include <string.h>



#ifdef __linux__

#include <termios.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <limits.h>

#else

#include <windows.h>

#endif

int RS232_OpenComport(int, int);
int RS232_PollComport(int, unsigned char *, int);
int RS232_SendByte(int, unsigned char);
int RS232_SendBuf(int, unsigned char *, int);
void RS232_CloseComport(int);
void RS232_cputs(int, const char *);
int RS232_IsDCDEnabled(int);
int RS232_IsCTSEnabled(int);
int RS232_IsDSREnabled(int);
void RS232_enableDTR(int);
void RS232_disableDTR(int);
void RS232_enableRTS(int);
void RS232_disableRTS(int);


#ifdef __cplusplus
} /* extern "C" */
#endif

#endif
```

rs232.c

```
/*
**************************************************************************
*
* Author: Teunis van Beelen
*
* Copyright (C) 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014
Teunis van Beelen
*
* teuniz@gmail.com
*
**************************************************************************
*
* This program is free software; you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published by
* the Free Software Foundation version 2 of the License.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License along
* with this program; if not, write to the Free Software Foundation, Inc.,
* 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
*
**************************************************************************
*
* This version of GPL is at http://www.gnu.org/licenses/old-licenses/gpl-
2.0.txt
*
**************************************************************************
*/

/* last revision: Januari 31, 2014 */

/* For more info and how to use this library, visit:
http://www.teuniz.net/RS-232/ */

#include "rs232.h"

#ifdef __linux__   /* Linux */

int Cport[30],
    error;

struct termios new_port_settings,
       old_port_settings[30];

char
comports[30][16]={"/dev/ttyS0","/dev/ttyS1","/dev/ttyS2","/dev/ttyS3","/dev/t
tyS4","/dev/ttyS5",

"/dev/ttyS6","/dev/ttyS7","/dev/ttyS8","/dev/ttyS9","/dev/ttyS10","/dev/ttyS1
1",

"/dev/ttyS12","/dev/ttyS13","/dev/ttyS14","/dev/ttyS15","/dev/ttyUSB0",

"/dev/ttyUSB1","/dev/ttyUSB2","/dev/ttyUSB3","/dev/ttyUSB4","/dev/ttyUSB5",

"/dev/ttyAMA0","/dev/ttyAMA1","/dev/ttyACM0","/dev/ttyACM1",
```

```
"/dev/rfcomm0","/dev/rfcomm1","/dev/ircomm0","/dev/ircomm1"};


int RS232_OpenComport(int comport_number, int baudrate)
{
  int baudr, status;

  if((comport_number>29)||(comport_number<0))
  {
    printf("illegal comport number\n");
    return(1);
  }

  switch(baudrate)
  {
    case      50 : baudr = B50;
                   break;
    case      75 : baudr = B75;
                   break;
    case     110 : baudr = B110;
                   break;
    case     134 : baudr = B134;
                   break;
    case     150 : baudr = B150;
                   break;
    case     200 : baudr = B200;
                   break;
    case     300 : baudr = B300;
                   break;
    case     600 : baudr = B600;
                   break;
    case    1200 : baudr = B1200;
                   break;
    case    1800 : baudr = B1800;
                   break;
    case    2400 : baudr = B2400;
                   break;
    case    4800 : baudr = B4800;
                   break;
    case    9600 : baudr = B9600;
                   break;
    case   19200 : baudr = B19200;
                   break;
    case   38400 : baudr = B38400;
                   break;
    case   57600 : baudr = B57600;
                   break;
    case  115200 : baudr = B115200;
                   break;
    case  230400 : baudr = B230400;
                   break;
    case  460800 : baudr = B460800;
                   break;
    case  500000 : baudr = B500000;
                   break;
    case  576000 : baudr = B576000;
                   break;
    case  921600 : baudr = B921600;
                   break;
    case 1000000 : baudr = B1000000;
                   break;
```

```
    default      : printf("invalid baudrate\n");
                   return(1);
                   break;
  }

  Cport[comport_number] = open(comports[comport_number], O_RDWR | O_NOCTTY |
O_NDELAY);
  if(Cport[comport_number]==-1)
  {
    perror("unable to open comport ");
    return(1);
  }

  error = tcgetattr(Cport[comport_number], old_port_settings +
comport_number);
  if(error==-1)
  {
    close(Cport[comport_number]);
    perror("unable to read portsettings ");
    return(1);
  }
  memset(&new_port_settings, 0, sizeof(new_port_settings));  /* clear the new
struct */

  new_port_settings.c_cflag = baudr | CS8 | CLOCAL | CREAD;
  new_port_settings.c_iflag = IGNPAR;
  new_port_settings.c_oflag = 0;
  new_port_settings.c_lflag = 0;
  new_port_settings.c_cc[VMIN] = 0;      /* block untill n bytes are received
*/
  new_port_settings.c_cc[VTIME] = 0;     /* block untill a timer expires (n *
100 mSec.) */
  error = tcsetattr(Cport[comport_number], TCSANOW, &new_port_settings);
  if(error==-1)
  {
    close(Cport[comport_number]);
    perror("unable to adjust portsettings ");
    return(1);
  }

  if(ioctl(Cport[comport_number], TIOCMGET, &status) == -1)
  {
    perror("unable to get portstatus");
    return(1);
  }

  status |= TIOCM_DTR;    /* turn on DTR */
  status |= TIOCM_RTS;    /* turn on RTS */

  if(ioctl(Cport[comport_number], TIOCMSET, &status) == -1)
  {
    perror("unable to set portstatus");
    return(1);
  }

  return(0);
}


int RS232_PollComport(int comport_number, unsigned char *buf, int size)
{
  int n;
```

```
  n = read(Cport[comport_number], buf, size);

  return(n);
}


int RS232_SendByte(int comport_number, unsigned char byte)
{
  int n;

  n = write(Cport[comport_number], &byte, 1);
  if(n<0)  return(1);

  return(0);
}


int RS232_SendBuf(int comport_number, unsigned char *buf, int size)
{
  return(write(Cport[comport_number], buf, size));
}


void RS232_CloseComport(int comport_number)
{
  int status;

  if(ioctl(Cport[comport_number], TIOCMGET, &status) == -1)
  {
    perror("unable to get portstatus");
  }

  status &= ~TIOCM_DTR;    /* turn off DTR */
  status &= ~TIOCM_RTS;    /* turn off RTS */

  if(ioctl(Cport[comport_number], TIOCMSET, &status) == -1)
  {
    perror("unable to set portstatus");
  }

  tcsetattr(Cport[comport_number], TCSANOW, old_port_settings +
comport_number);
  close(Cport[comport_number]);
}

/*
Constant  Description
TIOCM_LE  DSR (data set ready/line enable)
TIOCM_DTR DTR (data terminal ready)
TIOCM_RTS RTS (request to send)
TIOCM_ST  Secondary TXD (transmit)
TIOCM_SR  Secondary RXD (receive)
TIOCM_CTS CTS (clear to send)
TIOCM_CAR DCD (data carrier detect)
TIOCM_CD  Synonym for TIOCM_CAR
TIOCM_RNG RNG (ring)
TIOCM_RI  Synonym for TIOCM_RNG
TIOCM_DSR DSR (data set ready)

http://linux.die.net/man/4/tty_ioctl
*/
```

```
int RS232_IsDCDEnabled(int comport_number)
{
  int status;

  ioctl(Cport[comport_number], TIOCMGET, &status);

  if(status&TIOCM_CAR) return(1);
  else return(0);
}

int RS232_IsCTSEnabled(int comport_number)
{
  int status;

  ioctl(Cport[comport_number], TIOCMGET, &status);

  if(status&TIOCM_CTS) return(1);
  else return(0);
}

int RS232_IsDSREnabled(int comport_number)
{
  int status;

  ioctl(Cport[comport_number], TIOCMGET, &status);

  if(status&TIOCM_DSR) return(1);
  else return(0);
}

void RS232_enableDTR(int comport_number)
{
  int status;

  if(ioctl(Cport[comport_number], TIOCMGET, &status) == -1)
  {
    perror("unable to get portstatus");
  }

  status |= TIOCM_DTR;    /* turn on DTR */

  if(ioctl(Cport[comport_number], TIOCMSET, &status) == -1)
  {
    perror("unable to set portstatus");
  }
}

void RS232_disableDTR(int comport_number)
{
  int status;

  if(ioctl(Cport[comport_number], TIOCMGET, &status) == -1)
  {
    perror("unable to get portstatus");
  }

  status &= ~TIOCM_DTR;    /* turn off DTR */

  if(ioctl(Cport[comport_number], TIOCMSET, &status) == -1)
  {
    perror("unable to set portstatus");
```

```
  }
}

void RS232_enableRTS(int comport_number)
{
  int status;

  if(ioctl(Cport[comport_number], TIOCMGET, &status) == -1)
  {
    perror("unable to get portstatus");
  }

  status |= TIOCM_RTS;    /* turn on RTS */

  if(ioctl(Cport[comport_number], TIOCMSET, &status) == -1)
  {
    perror("unable to set portstatus");
  }
}

void RS232_disableRTS(int comport_number)
{
  int status;

  if(ioctl(Cport[comport_number], TIOCMGET, &status) == -1)
  {
    perror("unable to get portstatus");
  }

  status &= ~TIOCM_RTS;    /* turn off RTS */

  if(ioctl(Cport[comport_number], TIOCMSET, &status) == -1)
  {
    perror("unable to set portstatus");
  }
}


#else         /* windows */


HANDLE Cport[16];


char comports[16][10]={"\\\\.\\COM1",  "\\\\.\\COM2",  "\\\\.\\COM3",
"\\\\.\\COM4",
                       "\\\\.\\COM5",  "\\\\.\\COM6",  "\\\\.\\COM7",
"\\\\.\\COM8",
                       "\\\\.\\COM9",  "\\\\.\\COM10", "\\\\.\\COM11",
"\\\\.\\COM12",
                       "\\\\.\\COM13", "\\\\.\\COM14", "\\\\.\\COM15",
"\\\\.\\COM16"};

char baudr[64];


int RS232_OpenComport(int comport_number, int baudrate)
{
  if((comport_number>15)||(comport_number<0))
  {
    printf("illegal comport number\n");
    return(1);
```

```
  }

  switch(baudrate)
  {
    case     110 : strcpy(baudr, "baud=110 data=8 parity=N stop=1 dtr=on
rts=on");
                   break;
    case     300 : strcpy(baudr, "baud=300 data=8 parity=N stop=1 dtr=on
rts=on");
                   break;
    case     600 : strcpy(baudr, "baud=600 data=8 parity=N stop=1 dtr=on
rts=on");
                   break;
    case    1200 : strcpy(baudr, "baud=1200 data=8 parity=N stop=1 dtr=on
rts=on");
                   break;
    case    2400 : strcpy(baudr, "baud=2400 data=8 parity=N stop=1 dtr=on
rts=on");
                   break;
    case    4800 : strcpy(baudr, "baud=4800 data=8 parity=N stop=1 dtr=on
rts=on");
                   break;
    case    9600 : strcpy(baudr, "baud=9600 data=8 parity=N stop=1 dtr=on
rts=on");
                   break;
    case   19200 : strcpy(baudr, "baud=19200 data=8 parity=N stop=1 dtr=on
rts=on");
                   break;
    case   38400 : strcpy(baudr, "baud=38400 data=8 parity=N stop=1 dtr=on
rts=on");
                   break;
    case   57600 : strcpy(baudr, "baud=57600 data=8 parity=N stop=1 dtr=on
rts=on");
                   break;
    case  115200 : strcpy(baudr, "baud=115200 data=8 parity=N stop=1 dtr=on
rts=on");
                   break;
    case  128000 : strcpy(baudr, "baud=128000 data=8 parity=N stop=1 dtr=on
rts=on");
                   break;
    case  256000 : strcpy(baudr, "baud=256000 data=8 parity=N stop=1 dtr=on
rts=on");
                   break;
    case  500000 : strcpy(baudr, "baud=500000 data=8 parity=N stop=1 dtr=on
rts=on");
                   break;
    case 1000000 : strcpy(baudr, "baud=1000000 data=8 parity=N stop=1 dtr=on
rts=on");
                   break;
    default      : printf("invalid baudrate\n");
                   return(1);
                   break;
  }

  Cport[comport_number] = CreateFileA(comports[comport_number],
                      GENERIC_READ|GENERIC_WRITE,
                      0,                          /* no share  */
                      NULL,                       /* no security */
                      OPEN_EXISTING,
                      0,                          /* no threads */
                      NULL);                      /* no templates */
```

```
  if(Cport[comport_number]==INVALID_HANDLE_VALUE)
  {
    printf("unable to open comport\n");
    return(1);
  }

  DCB port_settings;
  memset(&port_settings, 0, sizeof(port_settings));  /* clear the new struct
*/
  port_settings.DCBlength = sizeof(port_settings);

  if(!BuildCommDCBA(baudr, &port_settings))
  {
    printf("unable to set comport dcb settings\n");
    CloseHandle(Cport[comport_number]);
    return(1);
  }

  if(!SetCommState(Cport[comport_number], &port_settings))
  {
    printf("unable to set comport cfg settings\n");
    CloseHandle(Cport[comport_number]);
    return(1);
  }

  COMMTIMEOUTS Cptimeouts;

  Cptimeouts.ReadIntervalTimeout         = MAXDWORD;
  Cptimeouts.ReadTotalTimeoutMultiplier  = 0;
  Cptimeouts.ReadTotalTimeoutConstant    = 0;
  Cptimeouts.WriteTotalTimeoutMultiplier = 0;
  Cptimeouts.WriteTotalTimeoutConstant   = 0;

  if(!SetCommTimeouts(Cport[comport_number], &Cptimeouts))
  {
    printf("unable to set comport time-out settings\n");
    CloseHandle(Cport[comport_number]);
    return(1);
  }

  return(0);
}


int RS232_PollComport(int comport_number, unsigned char *buf, int size)
{
  int n;

/* added the void pointer cast, otherwise gcc will complain about */
/* "warning: dereferencing type-punned pointer will break strict aliasing
rules" */

  ReadFile(Cport[comport_number], buf, size, (LPDWORD)((void *)&n), NULL);

  return(n);
}


int RS232_SendByte(int comport_number, unsigned char byte)
{
  int n;
```

```
  WriteFile(Cport[comport_number], &byte, 1, (LPDWORD)((void *)&n), NULL);

  if(n<0)  return(1);

  return(0);
}


int RS232_SendBuf(int comport_number, unsigned char *buf, int size)
{
  int n;

  if(WriteFile(Cport[comport_number], buf, size, (LPDWORD)((void *)&n),
NULL))
  {
    return(n);
  }

  return(-1);
}


void RS232_CloseComport(int comport_number)
{
  CloseHandle(Cport[comport_number]);
}

/*
http://msdn.microsoft.com/en-
us/library/windows/desktop/aa363258%28v=vs.85%29.aspx
*/

int RS232_IsDCDEnabled(int comport_number)
{
  int status;

  GetCommModemStatus(Cport[comport_number], (LPDWORD)((void *)&status));

  if(status&MS_RLSD_ON) return(1);
  else return(0);
}


int RS232_IsCTSEnabled(int comport_number)
{
  int status;

  GetCommModemStatus(Cport[comport_number], (LPDWORD)((void *)&status));

  if(status&MS_CTS_ON) return(1);
  else return(0);
}


int RS232_IsDSREnabled(int comport_number)
{
  int status;

  GetCommModemStatus(Cport[comport_number], (LPDWORD)((void *)&status));

  if(status&MS_DSR_ON) return(1);
  else return(0);
```

```
}


void RS232_enableDTR(int comport_number)
{
  EscapeCommFunction(Cport[comport_number], SETDTR);
}


void RS232_disableDTR(int comport_number)
{
  EscapeCommFunction(Cport[comport_number], CLRDTR);
}


void RS232_enableRTS(int comport_number)
{
  EscapeCommFunction(Cport[comport_number], SETRTS);
}


void RS232_disableRTS(int comport_number)
{
  EscapeCommFunction(Cport[comport_number], CLRRTS);
}

#endif


void RS232_cputs(int comport_number, const char *text)  /* sends a string to
serial port */
{
  while(*text != 0)   RS232_SendByte(comport_number, *(text++));
}
```

Кроме работы с COM-портом была ещё одна проблема – если запускать программу на планшете, надписи, сделанные кириллицей, отображаются неверно.

Но это, как выяснилось, не связано с wxWidgets. Я забыл, что при покупке планшета на нём была установлена англоязычная версия Windows 8. Позже я изменил это, но настройками не занимался. Достаточно было в панели управления зайти в региональные настройки и на закладке «Дополнительно» установить русский язык для программ, не поддерживающих Unicode.

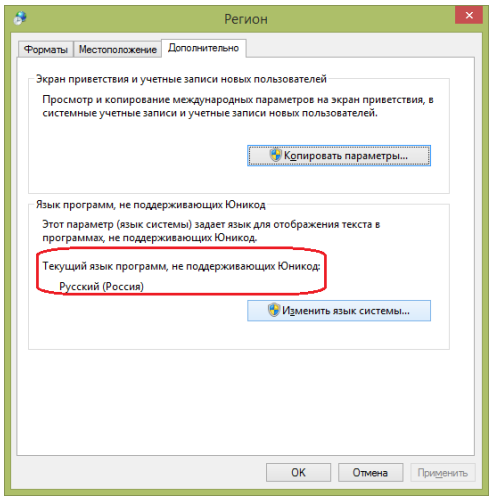


Рис. 41. Изменение параметров системы

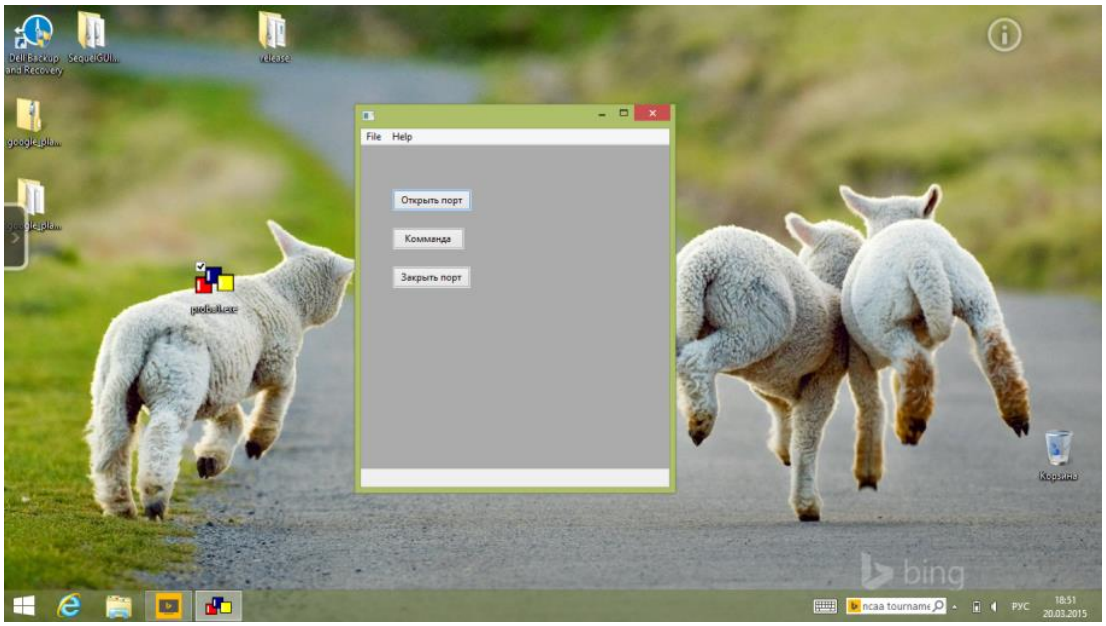


Рис. 41. Работа программы на планшете

После сборки проекта в режиме исполнения (Release) в одноимённой папке можно найти и запустить программу. Размер программы около 7 Мбайт. Я подключу осциллограф к COM-порту основного компьютера, чтобы убедиться в работе порта.

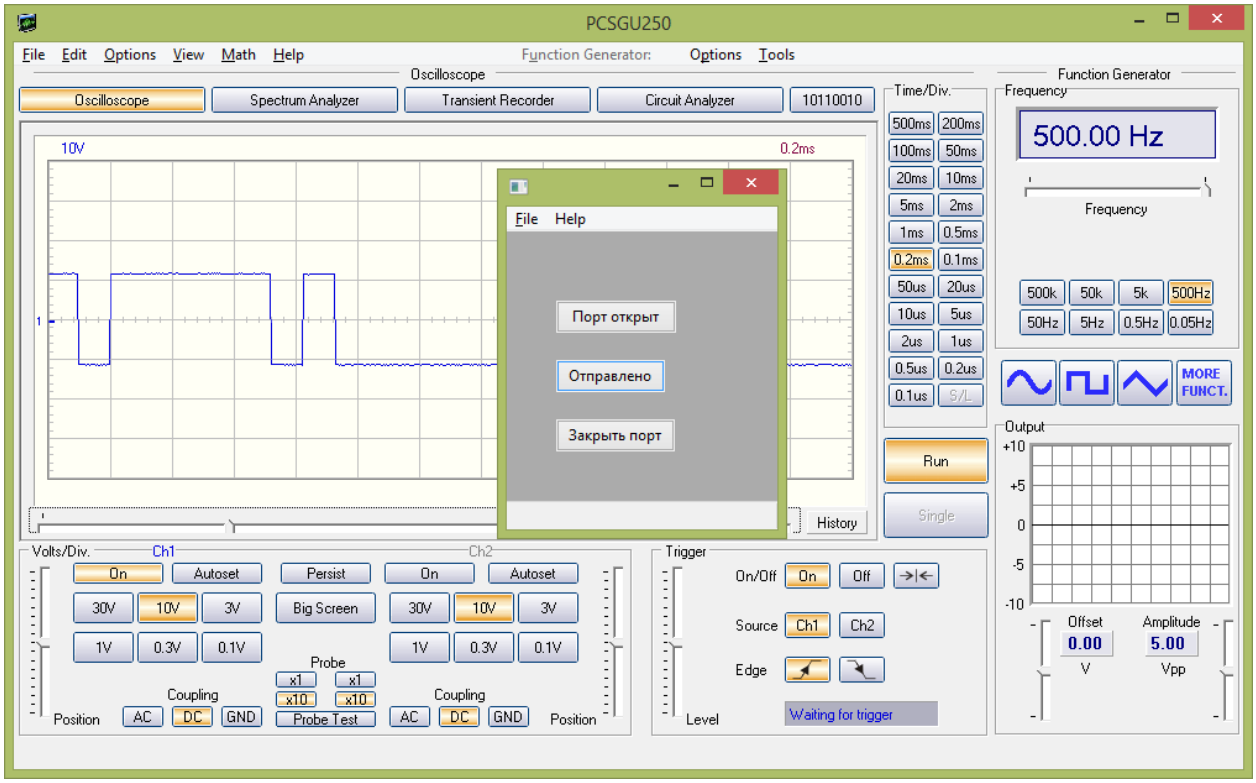


Рис. 42. Проверка работы программы и порта COM1

В этом месте отчего-то вспоминаются слова классика: «Так мало пройдено дорог. Так много сделано ошибок». Что ж, другой классик справедливо сказал: «Учиться, учиться и учиться!».