Лекция 2

ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ C++

Лекция 2. Основы объектно-ориентированного программирования на языке C++

План

1. Введение в язык программирования C++.
2. Основы объектно-ориентированного программирования. Классы и объекты.
3. Инкапсуляция. Наследование. Полиморфизм.
4. Перегрузка операций.
5. Системы программирования на языке C++.

1. Введение в язык программирования C++

C++ – высокоуровневый компилируемый язык программирования общего назначения со статической типизацией. На сегодняшний день C++ является одним из самых популярных и распространенных языков, т. к. он подходит для создания самых различных приложений (операционных систем, прикладных программ, драйверов устройств, серверов, компьютерных игр и т. д.).

Бьёрн Страуструп

Поддерживаемые парадигмы программирования:

- процедурная;
- структурная;
- объектно-ориентированная;
- обобщенная.

C++ был разработан в начале 1980-х годов сотрудником фирмы Bell Labs Бьёрном Страуструпом как объектно-ориентированное расширение языка C. Первое его название «C с классами».

1. Введение в язык программирования C++

Стандарты C++:

- C++98 – первый промышленный стандарт, разработанный комитетом по стандартизации C++ (ISO/IEC JTC1/SC22/WG21 working group) и утвержденный в 1998 году;
- C++03 – второй стандарт, утвержденный в 2003 году (является уточнением стандарта C++98);
- C++11 – утвержден в 2011 году; содержит значительные изменения по сравнению с предыдущими стандартами;
- C++14 – является уточнением стандарта C++11 (утвержден в 2014 году);
- C++17 – последний действующий стандарт (2017 год), содержит ряд улучшений по поддержке параллелизма.

1. Введение в язык программирования C++

C++ имеет мощную стандартную библиотеку (**C++ Standard Library**), представляющую из себя коллекцию классов и функций, написанных на C++.

Она включает в себя:

- стандартную библиотеку языка C;
- работу со строками;
- ввод-вывод;
- многопоточность;
- стандартную библиотеку шаблонов (STL – Standard Template Library);
- набор алгоритмических операций над контейнерами и другими последовательностями и т. п.

1. Введение в язык программирования C++

Не входит в C++ Standard Library, но существенно расширяет функционал языка C++ библиотека Boost.

Boost содержит:

- различные алгоритмы;
- многопоточное программирование;
- математические и численные алгоритмы;
- взаимодействие с другими языками программирования;
- синтаксический и лексический разбор;
- обобщённое программирование;
- структуры данных и многое другое.

2. Основы объектно-ориентированного программирования. Классы и объекты

Объектно-ориентированный подход (ООП) к программированию позволяет разработчику создавать собственные типы данных (классы), более адекватно описывающие предметную область, чем имеющиеся в языке программирования встроенные типы данных.

ООП базируется на следующих фундаментальных понятиях:

- **абстракция** (использование только тех характеристик объекта, которые с достаточной точностью представляют его в данной системе) – модель объекта предметной области, формализуемая в виде *класса*;
- **инкапсуляция** – размещение в одном компоненте данных и методов, которые с ними работают и/или скрытие внутренней реализации от других компонентов;
- **наследование** – механизм, позволяющий типу данных наследовать данные и функциональность некоторого существующего типа, способствуя повторному использованию компонентов программного обеспечения;
- **полиморфизм** – способность функции обрабатывать данные разных типов.

Фактически **класс** – это абстракция, т. е. созданный программистом новый тип данных, описывающий сущности предметной области более точно (наглядно), чем имеющиеся в языке программирования встроенные типы данных.

Объект – это переменная класса, физически существующая в памяти компьютера.

2. Основы объектно-ориентированного программирования. Классы и объекты

В C++ класс описывается следующим образом:

```
class <имя_класса>
{
[private:]
	// Закрытые свойства и методы (доступны только методам класса)
	// ...
[protected:]
	// Защищенные свойства и методы
	// (доступны только методам класса и его наследникам)
	// ...
[public:]
	// Интерфейсная часть класса
	// (свойства и методы, объявленные здесь, доступны в любом месте программы)
	// ...
};
```

2. Основы объектно-ориентированного программирования. Классы и объекты

Например, класс, описывающий автомобиль, может иметь следующий вид:

```
#include <string>

class Car
{
private:
	string model; // Модель автомобиля
	string color;   // Цвет
	int year;        // Год выпуска

public:
	Car(string m, string c, int y)       // Конструктор класса
	{
		model = m;
		color = c;
		year = y;
	}
	~Car(void) {} // Пустой декструктор класса (объявлен, но ничего не делает)
};
```

2. Основы объектно-ориентированного программирования. Классы и объекты

С каждым классом ассоциируются два стандартных метода – **конструктор** и **деструктор**.

Конструктор автоматически вызывается при создании объекта для его первоначальной инициализации. Его имя совпадает с именем класса. Конструкторов может быть несколько. Они должны в этом случае отличаться по количеству и/или типу агрументов.

Например, в классе Car можно объявить еще один «пустой» конструктор следующего вида:

```
class Car
{
//…
public:
	void Car(void)
	{
		model = color = "";
		year = 0;
	}
//...
};
```

2. Основы объектно-ориентированного программирования. Классы и объекты

В приведенных примерах у класса Car объявлены два конструктора. Они отличаются друг от друга количеством аргументов. Например.

```
// …

Car unknown,
      my_car("ZAZ", "green", 1972);


// …
```

В приведенном выше примере при создании объекта (переменной) unknown будет автоматически вызван пустой конструктор (без аргументов), а при создании переменной my_car будет вызван конструктор с тремя аргументами.

2. Основы объектно-ориентированного программирования. Классы и объекты

У класса в C++ может быть еще один специальный тип конструктора – **копирующий**. Он вызывается при первоначальной инициализации объекта в момент его объявления. Например.

```
// …
Car my_car("ZAZ", "green", 1972),
      my_car1 = my_car;   // Здесь вызывается копирующий конструктор
// …
```

Копирующий конструктор объявляется, например, так:

```
// Конструктор копирования
Car(const Car &right)
{
      model = right.model;
      color = right.color;
      year = right.year;
}
```

Таким образом, аргументом копирующего конструктора является константная (неизменяемая) ссылка на ранее созданный объект данного класса, значение которого копируется в текущий вновь создаваемый объект.

2. Основы объектно-ориентированного программирования. Классы и объекты

Деструктор автоматически вызывается тогда, когда объект удаляется из памяти (уничтожается).

```
// ...

int main(void)
{
    Car my_car("Volvo", "white", 2020); ← Здесь вызывается конструктор

    // ...


    return 0; ← Здесь при выходе из функции автоматически вызывается деструктор
}
```

Деструктор у класса в C++ всегда один. Его имя должно начинаться с тильды «~», а дальше оно совпадает с именем класса. Задача деструктора очистить объект, например, освободить динамически выделенную память, закрыть ранее открытые файлы и т. п.

3. Инкапсуляция. Наследование. Полиморфизм

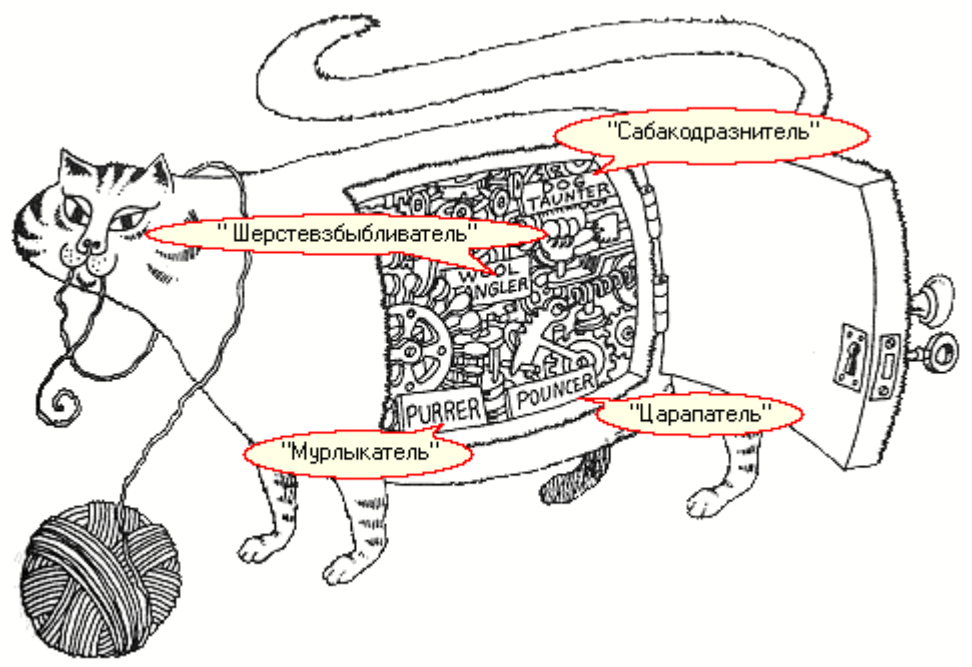
Механизм **инкапсуляции**, как сокрытия данных, реализуется путем использования модификаторов **private – protected – public** при объявлении класса.

Все члены класса, описанные либо сразу после открывающейся фигурной скобки, либо после ключевого слова **private**, являются его закрытой внутренней частью. Доступ к ним возможен только из методов данного класса. В приведенном выше классе Car свойства (переменные) model, color и year меняются только в конструкторе. Попытка напрямую обратиться к ним вызовет ошибку компиляции.

```
// …

Car my_car;


my_car.model = "ZAZ";  ← Здесь компилятор выдаст сообщение об ошибке

// ...
```

3. Инкапсуляция. Наследование. Полиморфизм

Обычно в секции **private** объявляются свойства и методы класса, необходимые для его функционирования, но скрываемые от пользователя, как ненужные ему детали реализации класса.



Для игры с кошкой вовсе не нужно знать, как она устроена (и тем более видеть как выглядят её внутренности :)

3. Инкапсуляция. Наследование. Полиморфизм

В секции **protected** объявляются свойства и методы класса, доступные для его методов и методов классов-наследников.

В секции **public** объявляются члены класса, доступные из любой части программы. Это так называемая интерфейсная часть класса. Она предназначена для взаимодействия с ним.

Например, интерфейсной частью автомобиля являются его руль, педали газа, тормоза и сцепления, а также рычаг переключения скоростей и панель приборов. Детали же реализации автомобиля, такие как его двигатель, шасси, трансмиссия, топливная система и т. п. обычно скрываются от водителя (хотя и не так строго как в ООП).

Детали реализации, обычно скрываемые от пользователя

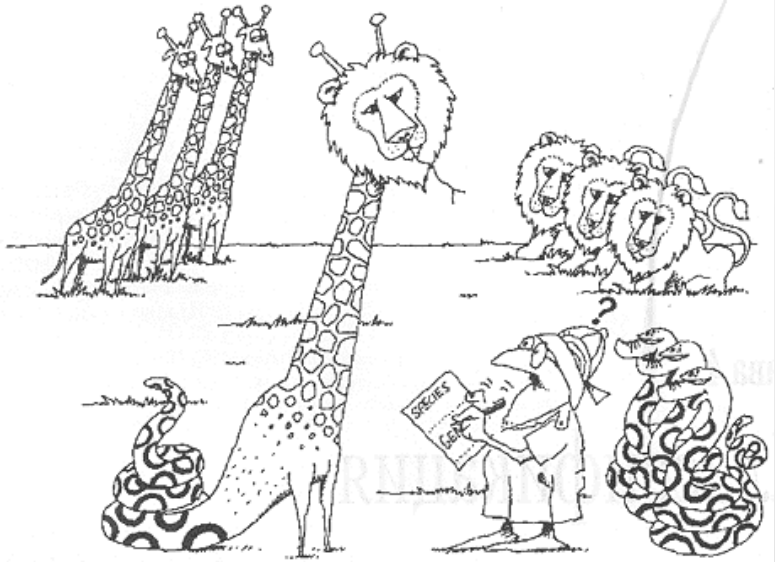
3. Инкапсуляция. Наследование. Полиморфизм

Механизм **наследования** позволяет создавать новые классы (**дочерние** или **производные**) на основе уже существующих (**родительских** или **базовых**). При этом дочерний класс наследует все свойства и методы родительского класса, расширяя его функциональность за счет новых свойств и методов, реализованных в нем.

Наследование позволяет без переписывания больших объемов существующего программного кода создавать новые классы с улучшенной функциональностью.

В C++ класс-наследник создается следующим образом:

```
// Родительский класс
class A
{
	// …
};

// Дочерний класс
class B : public A
{
	// …
};
```



Наследование позволяет приобретать свойства родителей

3. Инкапсуляция. Наследование. Полиморфизм

Рассмотрим следующий пример. Пусть требуется разработать драйверы (программы управления) для разнотипных принтеров одного производителя. Предполагается, что они должны быть максимально унифицированы и различаться только деталями реализации для каждого конкретного типа принтера (матричного, струйного или лазерного). Понятно, что на физическом уровне способ печати у каждого типа принтера отличается. Но все остальные характеристики (поддерживаемый формат бумаги, скорость и качество печати, цветопередача и т. п.) у них могут быть унифицированы.

Матричный принтер

Струйный принтер

Лазерный принтер

3. Инкапсуляция. Наследование. Полиморфизм

Некий абстрактный принтер можно в самом общем виде описать, например, так.

```
//-----------------------------------------------------------------------------
// Абстрактный класс, описывающий печатающее устройство
//-----------------------------------------------------------------------------
class Printer
{
private:
	string format;		// Поддерживаемый формат бумаги (например, A4)
	string model;		// Модель принтера
	// ...
public:
	Printer(string f, string m)	// Конструктор и деструктор
	{
		format = f;
		model = m;
	}
	virtual ~Printer(void) {}
	virtual void print(void) = 0; // Чистая виртуальная функция печати
};
```

3. Инкапсуляция. Наследование. Полиморфизм

Если в классе в начале объявления метода добавлено ключевое слово **virtual**, то это означает, что данный метод является **виртуальным** и может быть переопределен в классах-наследниках. Если объявление виртуального метода заканчивается выражением «= 0;», то он называется **чистой виртуальной функцией**. Наличие чистых виртуальных функций делает класс абстрактным. **Абстрактный класс** определяет интерфейс для переопределения производными классами и создать для него объект нельзя. Если, например, написать в программе следующий код, то компилятор выдаст ошибку об использовании абстрактного класса.

```
#include "Printer.h"
// ...

int main(void)
{
	Printer prn;	// Ошибка непосредственного использования абстрактного класса

	// …
	return 0;
}
```

3. Инкапсуляция. Наследование. Полиморфизм

В приведенном выше примере чистая виртуальная функция **print()** должна реализовать низкоуровневую процедуру печати на принтере конкретной модели. Обычно в результате ее работы создается двоичный файл, содержащий управляющие коды конкретного принтера, который затем копируется в память печатающего устройства и после этого физически начинается процесс печати. Программная реализация данного метода осуществляется в классе-наследнике, разрабатываемом для конкретного типа принтера (или даже модели). Здесь следует отметить, что если от класса будут создаваться наследники, то его деструктор также следует объявлять виртуальным методом. В этом случае при удалении объекта производного класса по цепочке будут вызваны и деструкторы всех его базовых классов, что гарантирует корректное освобождение памяти (и, соответственно, отсутствие утечки памяти!).

Предположим, изготовитель на базе существующего принтера разработал новую усовершенствованную модель, которая обладает рядом дополнительных возможностей. Вместо разработки для него драйвера «с нуля», достаточно создать дочерний класс с переписанной функцией print(), программно реализующей новые возможности печати.

3. Инкапсуляция. Наследование. Полиморфизм

Например.

```
// Класс, реализующий драйвер для модели «Type3»
class PrinterType3 : public Printer
{
public:
    PrinterType3(string f, string m) : Printer(f, m) {} // Вызов конструктора базового класса
    virtual ~PrinterType3(void)
    {
       // ...
    }
    // ...
    void print(void) // Реализация печати для конкретной модели
    {
        // do something
        // ...
    }
    // ...
};
```

3. Инкапсуляция. Наследование. Полиморфизм

Предположим, к компьютеру подключено несколько принтеров. Реализация печати на них в операционной системе может выглядеть так.

```
#include "Printer.h"

// Полиморфная функция печати
void sysPrint(Printer *prn)
{
    prn->print();     // Вывод на печать в конкретное устройство
}

int main()
{
    Printer *prn1 = new PrinterType3("A1", "EpsonDFX8000"),   // Создание объектов
            *prn2 = new PrinterType8("A4", "EpsonLX18");
    // ...
    sysPrint(prn1);       // Печать
    sysPrint(prn2);
    // ...
    delete prn1;          // Освобождение памяти (здесь будут вызваны деструкторы)
    delete prn2;
    return 0;
}
```

3. Инкапсуляция. Наследование. Полиморфизм

В приведенном выше примере реализована функция **sysPrint()**, демонстрирующая работу **полиморфизма** – способности функций обрабатывать данные разных типов. Полиморфизм также позволяет сократить объемы написания программного кода, т. к. для унифицированной обработки данных разных типов можно использовать один и тот же код.

В данном примере функция **sysPrint()** в качестве аргумента получает указатель на базовый абстрактный класс Printer, которым также являются и все его производные классы (PrinterType3 и PrinterType8). В теле этой функции вызывается метод базового класса **print()**, который в родительском классе является чистой виртуальной функцией, но должен быть реализован в каждом дочернем классе. При выполнении кода «prn->print();» будет автоматически определен класс, на объект которого ссылается указатель prn, и вызван соответствующий ему метод печати. Это называется **поздним связыванием**, что означает, что объект связывается с вызовом функции только во время исполнения программы, а не раньше.

4. Перегрузка операций

Правильно спроектированный класс должен быть так же удобен в применении, как и встроенный тип данных. Для этого в классах C++ поддерживается перегрузка операций (один из способов реализации полиморфизма). Под **перегрузкой операций** понимается возможность наличия одинаковых операций (операторов), различающихся только типами обрабатываемых параметров (операндов).

Например, результат вычисления выражения «**a * b**» зависит от типов операндов a и b. Если это числа, то очевидно, что результатом является их произведение. А если это строки? Каким результатом будет вычисление выражения «"Hello world!" * 3»? И корректно ли такое выражение? В языке программирования Python, например, результатом его вычисления является новая строка, образованная повторением заданное количество раз исходной.

```
Type "help", "copyright", "credits" or "license" for more information.
>>> "Hello world!" * 3
'Hello world!Hello world!Hello world!'
>>>
```

Таким образом, например, операция умножения может быть полиморфной, т. е. результат ее выполнения зависит от типов параметров.

Для реализации такой возможности в ООП и поддерживается перегрузка операций.

4. Перегрузка операций

В C++ для перегрузки операций используется специальная функция **operator()**. Рассмотрим следующий пример. Пусть необходимо реализовать векторные вычисления на плоскости. Для этого объявим следующий класс Vector2D.

```
// Вектор на плоскости
class Vector2D
{
private:
    double x, y; // Координаты вектора
public:
    Vector2D(void) : x(0), y(0) {}
    Vector2D(double px, double py) : x(px), y(py) {}
    Vector2D(const Vector2D &r) : x(r.x), y(r.y) {}      // Копирующий конструктор
    ~Vector2D(void) {}
    Vector2D operator = (const Vector2D &r)    // Перегруженный оператор присваивания
    {
        x = r.x;      y = r.y;
        return *this;      // Возврат разыменованного указателя на текущий объект
    }
    friend ostream& operator << (ostream& out, const Vector2D &r) // Перегруженный оператор
    {                                                              // вывода (в стандартный поток)
        out << '(' << r.x << ',' << r.y << ')';
        return out;
    }
};
```

4. Перегрузка операций

В приведенном примере в классе Vector2D перегружены две операции: присваивания («=») и побитового сдвига («<<»), которое теперь стало выводом в стандартный поток.

Отметим, что способ перегрузки операций зависит от того, являются ли они унарными или бинарными. **Унарная операция** имеет один операнд, например, «-a» (изменение знака). **Бинарная** – два операнда, например, «a - b» (отнимание b от a). При перегрузке бинарных операций следует учитывать следующий нюанс. В каждый метод класса неявно передается еще один параметр, содержащий адрес текущего объекта. К нему можно явно обратиться через использование специальной зарезервированной переменной «this», содержащей адрес текущего объекта.

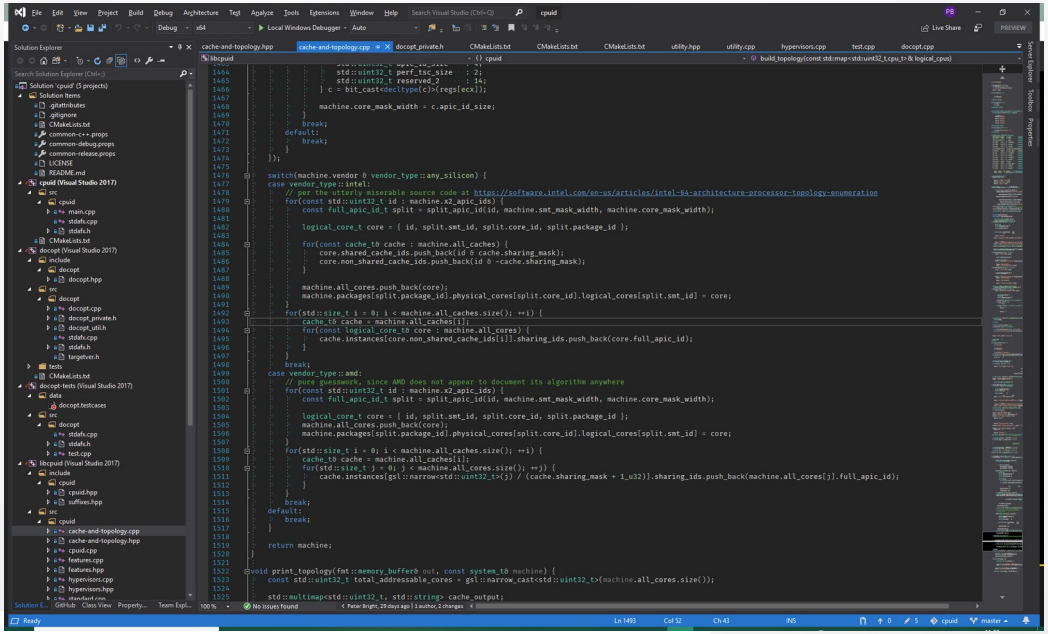
Таким образом, если в классе объявить метод, перегружающий бинарную операцию, то он фактически должен получать не два аргумента (операнда операции), а три (еще и this). Возникшее противоречие устраняется следующим способом: функция, перегружающая бинарную операцию, объявляется не как член класса (его метод), а как так называемая **дружественная функция** (friend), которая формально не является методом класса, но имеет доступ к его закрытым (private) и защищенным (protected) данным.

В приведенном выше примере перегруженная операция присваивания является унарной, а операция вывода в поток – бинарной.

5. Системы программирования на языке C++

Для разработки на C++ используется большое количество различных интегрированных сред разработки (IDE – Integrated Development Environment). Среди самых популярных можно выделить следующие.

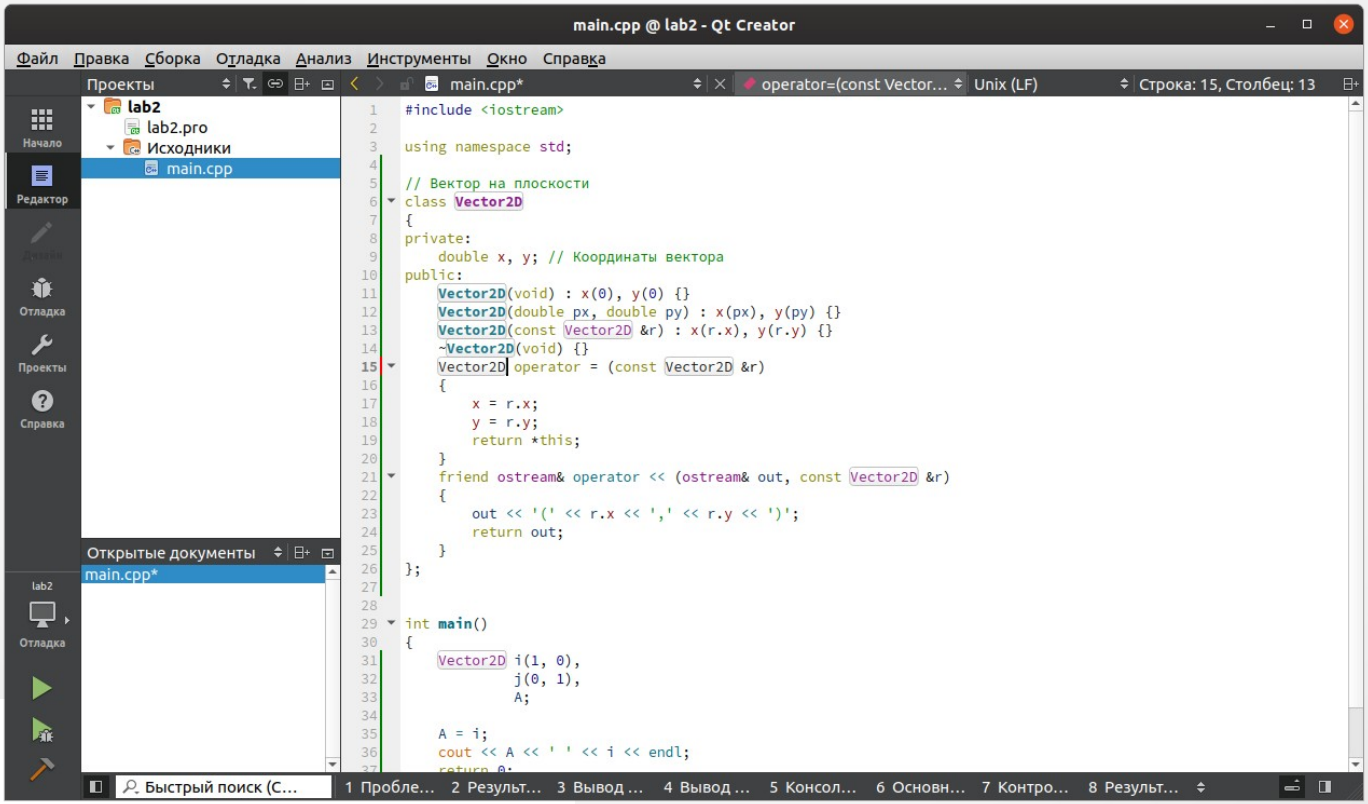
1. **Microsoft Visual Studio** – наиболее популярная IDE разработки программ для семейства операционных систем Windows. Является проприетарной, но для нее существуют и бесплатные версиии с ограниченной функциональностью.



Visual Studio

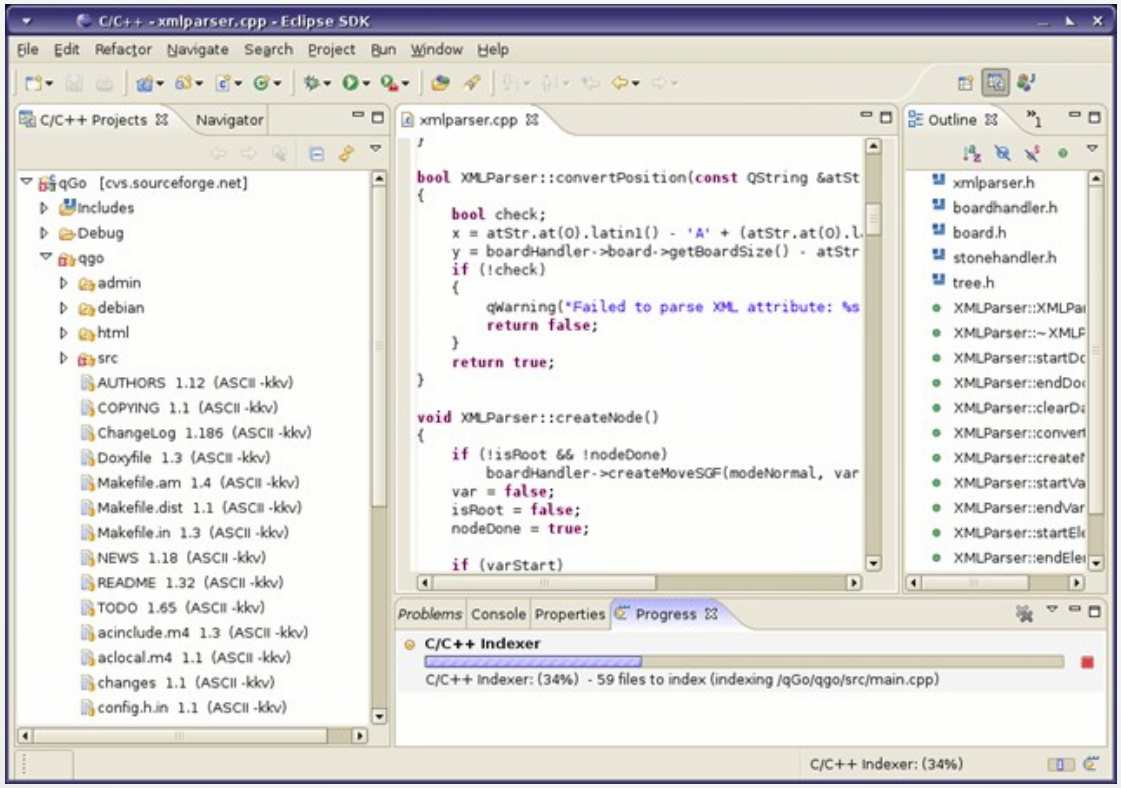
5. Системы программирования на языке C++

2. **Qt Creator** – популярная кроссплатформенная IDE для разработки программ под ОС Windows, Linux и MacOS. Полнофункциональная версия данной среды разработки также является проприетарной, но имеются и бесплатные версиии.



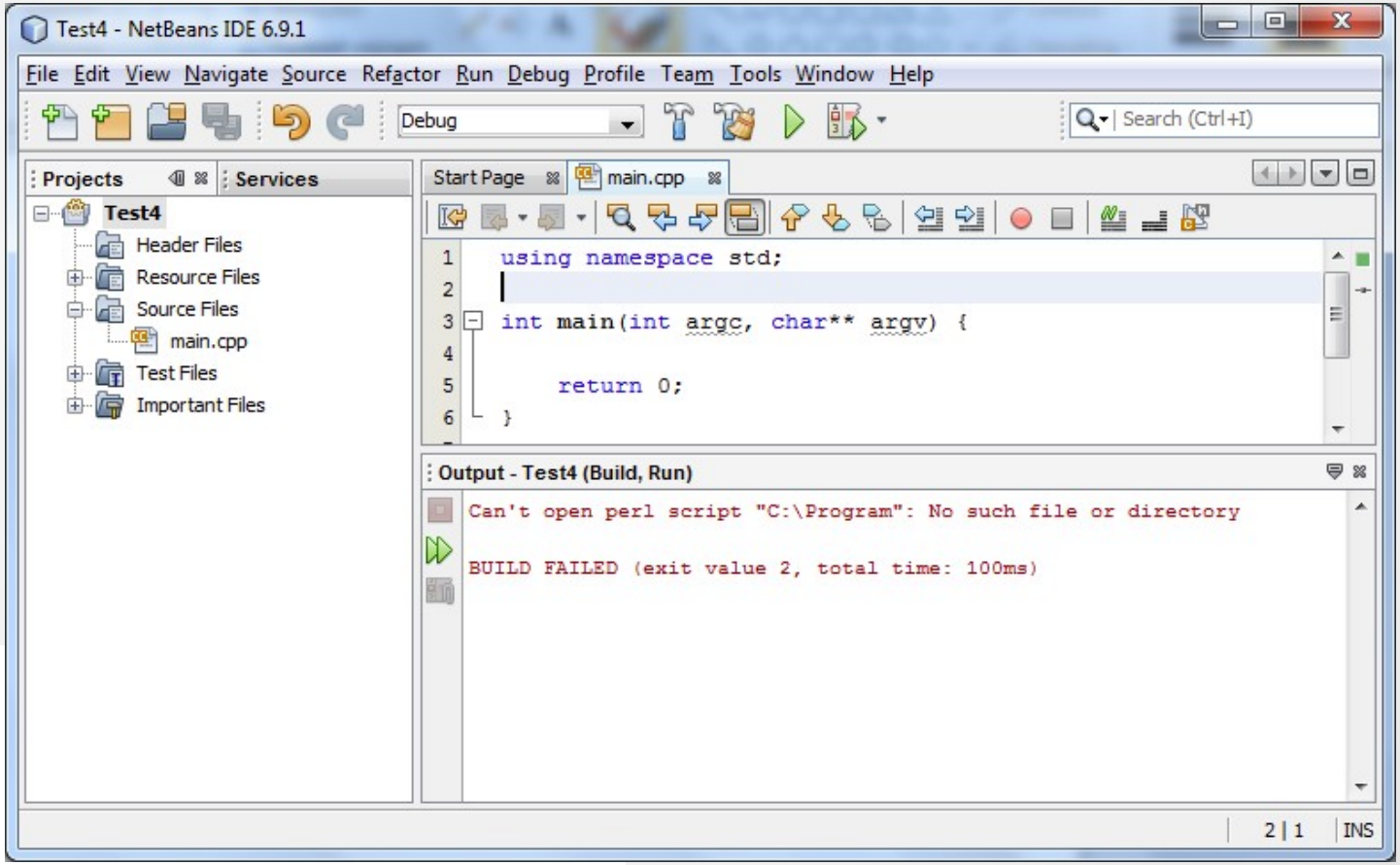
5. Системы программирования на языке C++

3. **Eclipse** – является одной из ведущих IDE для разработчиков на C и C++. Это полностью бесплатный кросплатформенный программный продукт с открытым исходным кодом, работающий со всеми основными ОС.



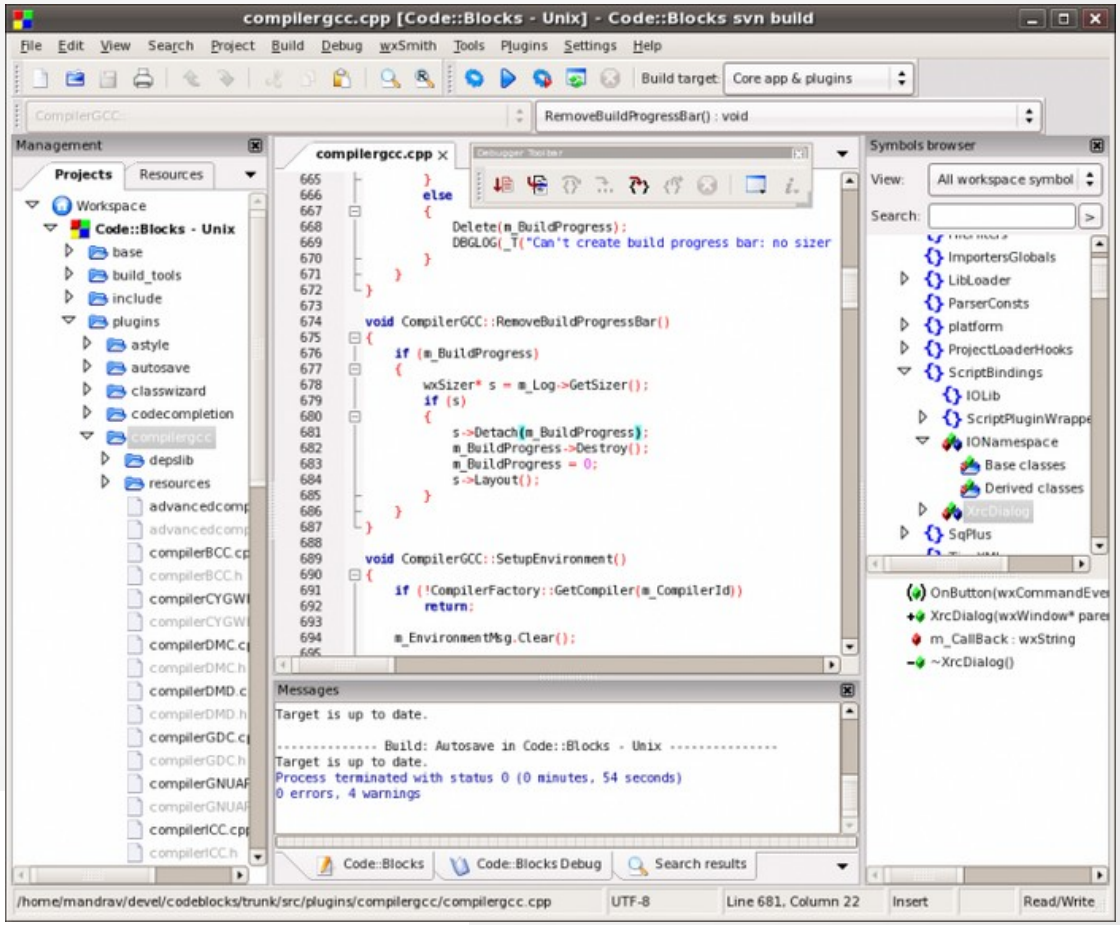
5. Системы программирования на языке C++

4. **NetBeans** – это также одна из популярных среди разработчиков кроссплатформенных IDE для программирования на C++ с открытым исходным кодом.



5. Системы программирования на языке C++

4. **Code::Blocks** – популярная легковесная кроссплатформенная IDE с открытым исходным кодом, распространяемая бесплатно.



Code::Blocks

The open source, cross-platform IDE