Список вопросов по дисциплине Программирование, 3 семестр

1. Классы. Назначение секций класса. Конструкторы, списки инициализации.
2. Деструкторы и время жизни объектов класса. Константность методов.
3. Классы. Порядок конструирования объектов класса.
4. Симметричные шифры и их реализация в контексте ООП.
5. Основные принципы создания объектной модели (абстрагирование, инкапсуляция модульность, иерархия, типизация, параллелизм, сохраняемость).
6. Экземпляры класса. Перечислите и охарактеризуйте уровни доступа к членам класса. Классификация методов объекта.
7. Средства инкапсуляции C++. Представление иерархических отношений. Наследование.
8. Перегрузка операторов.
9. Исключения в C++. Обработка исключений.
10. Шаблоны классов и шаблоны функций. Специализация.
11. Понятие юнит тестирования, основные этапы юнит тестирования. Тестовая документация, стратегия тестирования.
12. Реализация в С++: Boost Test Library, Google test, Framework UnitTest. Реализация юнит тестирования на примере другого ЯП.
13. Наследование классов ошибок.
14. Механизм сериализации и десериализации. Примеры на двух языках высокого уровня.
15. Бинарная (базовая) сериализация и десериализация.
16. Json сериализация и десериализация. Использование атрибутов. Использование объекта класса опций.
17. Xml сериализация и десериализация.
18. Принцип реализации дружественных функций. Сравнительный анализ с реализацией на принципах get/set.
19. Наследование и полиморфизм. Использование умных указателей.
20. Понятие многопоточности, сферы применения. Достоинства и недостатки.
21. Понятие потока и процесса. Конкурентность и параллелизм. Методы управления потоками.
22. Основные примитивы синхронизации Mutex, Semaphore, SemaphoreSlim, Barrier, SpinLock, SpinWait, Monitor. Примеры. StopWatch как метод формирования оценки времени.
23. Понятие статической и динамической библиотек. Порядок генерации библиотек и механизма использования. Пример на двух языках программирования.
24. Паттерн программирования Строитель. Пример.
25. Паттерн программирования Фабрика. Пример.
26. Принципы SOLID. Паттерны программирования. Singleton. Ситуации для применения.
27. Безопасность паттерна Singleton, многопоточности, дружественных функций, сериализации\десериализации.
28. **Класс** – это способ описания сущности, определяющий состояние и поведение, зависящее от этого состояния, а также правила для взаимодействия с данной сущностью. С точки зрения программирования класс можно рассматривать как набор данных и функций для работы с ними (методов). С точки зрения структуры программы, класс является сложным типом данных.

Зачем нужно: классы нужны для создания своих структур данных, которые будут содержать какую-то логику обработки. Вся логика хранится в описании класса, при этом оставляя в вызывающей программе лаконичные вызовы, без лишнего кода.

**Секции класса:**

* Секция «public» (общедоступный). К элементам этой секции можно обращаться из любого модуля. Доступ возможен из методов этого класса, его потомков и вообще из любых процедур.
* Секция «protected» (защищённый). К элементам этой секции можно обратиться из модуля, в котором описан класс. В другом модуле доступ возможен из методов «дружественного» класса или его потомков.
* Секция «private» (частный). К элементам этой секции обратиться из других модулей нельзя. Доступ возможен из методов этого же класса, его потомков и процедур, расположенных в данном модуле.

**Конструктор** – это особый вид функции-члена класса, которая автоматически вызывается при создании экземпляра объекта этого класса. Конструкторы обычно используются для инициализации переменных-членов класса соответствующими значениями по умолчанию или пользовательскими значениями или для выполнения любых шагов настройки, необходимых для использования класса (например, открытие файла или базы данных).

В отличие от обычных функций-членов, у конструкторов есть определенные правила того, как они должны называться:

1. конструкторы должны иметь то же имя, что и класс (с такими же заглавными буквами);
2. конструкторы не имеют возвращаемого типа (даже не void).

Списки инициализации представляют перечисления инициализаторов для каждой из переменных и констант через двоеточие после списка параметров конструктора:

Person(string n, int a) : name(n), age(a), ageRef(age)

{

}

2. Деструктор выполняет освобождение использованных объектом ресурсов и удаление нестатических переменных объекта. По сути деструктор - это функция, которая называется по имени класса (как и конструктор) и перед которой стоит тильда (~). Деструктор не имеет возвращаемого значения и не принимает параметров. Каждый класс может иметь только один деструктор.

Деструктор автоматически вызывается, когда удаляется объект. Удаление объекта происходит в следующих случаях:

* когда завершается выполнение области видимости, внутри которой определены объекты
* когда удаляется контейнер (например, массив), который содержит объекты
* когда удаляется объект, в котором определены переменные, представляющие другие объекты
* динамически созданные объекты удаляются при применении к указателю на объект оператора delete

class Person

{

public:

Person(std::string n)

{

name = n;

}

~Person()

{

std::cout << "Destructor called for Person " << name << std::endl;

}

private:

std::string name;

};

int main()

{

Person tom("Tom");

Person \*sam = new Person("Sam");

delete sam; // вызывается деструктор для объекта sam

std::cout << "End of Main" << std::endl;

return 0;

} // вызывается деструктор для объекта tom

В функции main создается переменная tom, которая хранит объект Person. Это обычная переменная, не указатель. И для нее деструктор будет вызываться, когда завершит выполнение та область видимости, где эта переменная определена, то есть функция main.

Также здесь определен указатель bob, который указывает на объект Person. Это динамический объект, который определяется с помощью ключевого слова new. Когда такие объекты выходят из области видимости, то для них автоматически не выполняется деструктор. Поэтому для вызова деструктора и удаления таких объектов применяется оператор delete:

Person \*sam = new Person("Sam");

delete sam; // вызывается деструктор для объекта sam

1. Порядок конструирования объектов класса

class A {}

class B

{

public:

B() = default;

private:

A a;

static A a0;

int i;

static int i0;

}

Из стандарта:

Инициализация в неделегирующем конструкторе производится в следующем порядке:  
■ (13.1) Сначала, и только для конструктора самого производного класса (1.8), инициализируются виртуальные базовые классы в порядке их появления при обходе направленного ациклического графа в направлении «снизу вверх слева направо», где «слева направо» означает очередность появления базовых классов в списке-спецификаторов-баз производного класса.  
■ (13.2) Затем инициализируются непосредственные базовые классы в порядке их упоминания в списке-спецификаторов-баз (независимо от порядка mem-инициализаторов).  
■ (13.3) Далее инициализируются нестатические данные-члены, в порядке их объявления в описании класса (опять-таки, независимо от порядка mem-инициализаторов).  
■ (13.4) Наконец, выполняется составной-оператор тела конструктора. Замечание. Инициализация в порядке объявления обязательна, так как гарантирует разрушение подобъектов базовых классов и членов в порядке, обратном порядку их инициализации.

Т.е. в вашем классе B будет вызван конструктор A, потом - член a; i был бы проинициализирован после a (в порядке объявления), если бы вы указали его инициализатор (или в конструкторе, типа B():i(5){...}, или значение по умолчанию типа int i = 6;).

Статические члены инициализируются до начала работы с классом, однократно.

1. Симметричные шифры и их реализация в контексте ООП

**Симметричное шифрование** — это способ шифрования данных, при котором один и тот же ключ используется и для кодирования, и для восстановления информации.

**Трёхэтапный протокол Шамира:**

Пусть абонент *А* должен передать абоненту *В* секретное сообщение *m*. *А* выбирает случайное большое простое число *p* и открыто передает его абоненту *В*. Затем абонент *А* выбирает два числа *cA* и *dA* такие, что

*cAdA* **mod** (*p*−1) = 1.

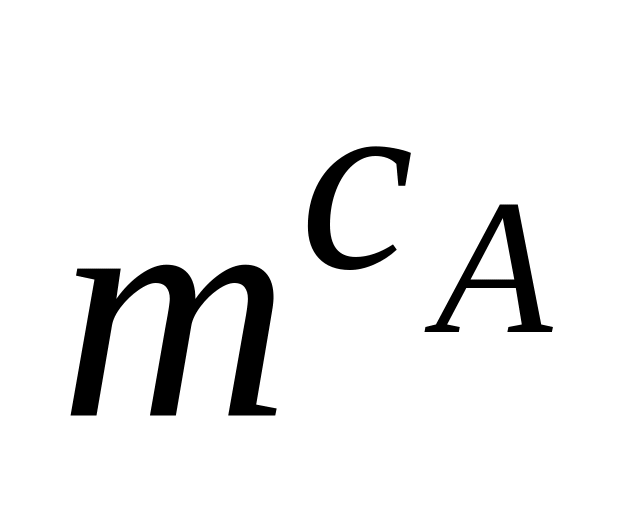
Эти числа *А* держит в секрете. Абонент *В* тоже выбирает два секретных числа *cB* и *dB* такие, что

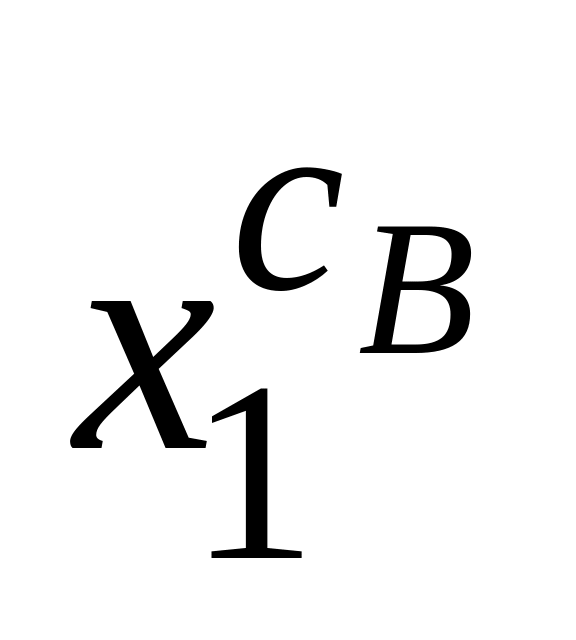
*cBdB* **mod** (*p*−1) = 1.

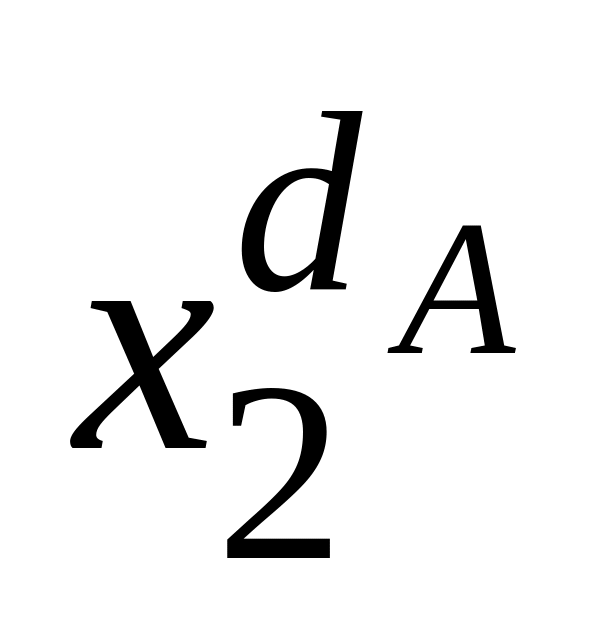
После этого *А* передает свое сообщение *m*, используя трехступенчатый протокол (см. ниже). При этом, если *m*<*p* (*m* рассматривается как число), то сообщение *m* передается целиком, иначе оно представляется в виде *m* = *m*1 *m*2 ... *mt* и каждая часть *mi* < *p* передается отдельно. Для кодирования каждой части *mi* лучше выбирать случайно новые пары (*cAdA*) и (*cBdB*); в противном случае надежность системы понижается.

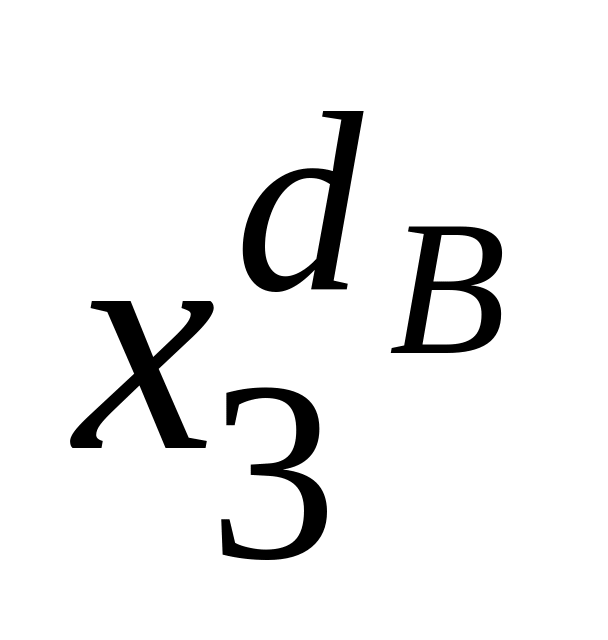
В настоящее время шифр Шамира используется, главным образом, для передачи чисел, например, секретных ключей, значения которых меньше *p*.

***Протокол шифра Шамира*** имеет вид:

**Шаг 1**. *А* вычисляет число *x*1 =  **mod** *p* и передает его абоненту *В*.

**Шаг 2**. *B* вычисляет число *x*2 =  **mod** *p* и передает его абоненту *А*.

**Шаг 3**. *А* вычисляет число *x*3 =  **mod** *p* и передает его абоненту *В*.

**Шаг 4**. *В* вычисляет число *x*4 =  **mod** *p*.

**Криптопротокол Диффи-Хеллмана:**

Пусть p — большое простое число, g — примитивный элемент группы Zp, y=gx mod p, причём p, y и g известны заранее. Функцию y=gx mod p считаем однонаправленной, то есть вычисление функции при известном значении аргумента является лёгкой задачей, а её обращение (нахождение аргумента) при известном значении функции — трудной. (Обратную функцию x = logg y mod p называют функцией дискретного логарифма. В настоящий момент не существует быстрых способов вычисления такой функции для больших простых p.)

Протокол обмена состоит из следующих действий.

1. Алиса выбирает случайное 2 <= a <= p-1

Alice {A = ga mod p} Bob

1. Боб выбирает случайное

Боб вычисляет сессионный ключ

Bob {B = gb mod p} Alice

1. Алиса вычисляет K = Ba mod p

Таким способом создан общий секретный сессионный ключ . За счёт случайного выбора значений  и  в новом сеансе будет получен новый сессионный ключ.

**Криптопротокол Эль Гамаля:**

Для того чтобы генерировать пару ключей (открытый ключ – секретный ключ), сначала выбирают некоторое большое простое число Р и большое целое число G, причем G < P. Числа Р и G могут быть распространены среди группы пользователей.

Затем выбирают случайное целое число X, причем X<P. Число X является секретным ключом и должно храниться в секрете.

Далее вычисляют Y = GX mod P. Число Y является открытым ключом.

Для того чтобы зашифровать сообщение M, выбирают случайное целое число K, 1< K< P –1, такое, что числа К и (Р –1) являются взаимно простыми.

Затем вычисляют числа

a = GK mod P,

b = YK M mod P.

Пара чисел (a,b) является шифртекстом. Заметим, что длина шифртекста вдвое больше длины исходного открытого текста М.

Для того чтобы расшифровать шифртекст (a,b), вычисляют

M = mod P. (\*)

Поскольку

aX≡ GKXmod P,

≡YK≡ GKX≡ M (mod P),

то соотношение (\*) справедливо

**Криптопротокол RSA:**

Первым этапом любого асимметричного алгоритма является создание пары ключей – открытого и закрытого и распространение открытого ключа "по всему миру".

1. **Создание ключей**

Для алгоритма RSA этап создания ключей состоит из следующих операций:

1. Выбираются два очень больших простых числа  and .
2. Вычисляется их произведение , которое называется модулем.
3. Вычисляется значение функции Эйлера от числа :



1. Выбирается произвольное число  ( ), взаимно простое со значением функции .

Число  называется открытой экспонентой

1. С помощью алгоритма Евклида вычисляется число , которое удовлетворяет условию 
2. Пара  публикуется в качестве открытого ключа RSA.
3. Пара  играет роль закрытого ключа RSA и держится в секрете.
4. **Шифрование и расшифрование**

Предположим, отправитель хочет послать получателю сообщение .

Сообщениями являются целые числа в интервале от 0 до , т.е . . На рисунке 1 представлена схема алгоритма RSA.

**Алгоритм Отправителя:**

1. Взять открытый ключ  получателя
2. Взять открытый текст 
3. Зашифровать сообщение с использованием открытого ключа получателя:



**Алгоритм Получателя:**

1. Принять зашифрованное сообщение 
2. Взять свой закрытый ключ 
3. Применить закрытый ключ для расшифрования сообщения:



1. Основные принципы создания объектной модели

Концептуальной основой объектно-ориентированного подхода является объектная модель. Основными принципами ее построения являются:

· абстрагирование (abstraction);

· инкапсуляция (encapsulation);

· модульность (modularity);

· иерархия (hierarchy).

* Абстрагирование — это выделение наиболее важных, существенных характеристик некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, четко определяют его концептуальные границы с точки зрения дальнейшего рассмотрения и анализа, и игнорирование менее важных или незначительных деталей.

Абстрагирование позволяет управлять сложностью системы, концентрируясь на существенных свойствах объекта. Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности его поведения от деталей их реализации. Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу объектно-ориентированного проектирования. Абстракция зависит от предметной области и точки зрения — то, что важно в одном контексте, может быть не важно в другом. Объекты и классы — основные абстракции предметной области.

* Инкапсуляция — физическая локализация свойств и поведения в рамках единственной абстракции (рассматриваемой как «черный ящик»), скрывающая их реализацию за общедоступным интерфейсом.
* Инкапсуляция - это процесс отделения друг от друга отдельных элементов объекта, определяющих его устройство и поведение. Инкапсуляция служит для того, чтобы изолировать интерфейс объекта, отражающий его внешнее поведение, от внутренней реализации объекта. Объектный подход предполагает, что собственные ресурсы, которыми могут манипулировать только операции самого объекта, скрыты от внешней среды. Абстрагирование и инкапсуляция являются взаимодополняющими: абстрагирование фокусирует внимание на внешних особенностях объекта, а инкапсуляция (или иначе ограничение доступа) не позволяет объектам-пользователям различать внутреннее устройство объекта.

Инкапсуляция подобна понятию сокрытия информации (information hiding). Это возможность скрывать многочисленные детали объекта от внешнего мира. Внешний мир объекта — это все, что находится вне его, включая остальную часть системы. Сокрытие информации предоставляет то же преимущество, что и инкапсуляция, - гибкость.

* Модульность — это свойство системы, связанное с возможностью ее декомпозиции на ряд внутренне сильно сцепленных, но слабо связанных между собой подсистем (модулей).

Модульность снижает сложность системы, позволяя выполнять независимую разработку отдельных модулей. Инкапсуляция и модульность создают барьеры между абстракциями.

* Иерархия — это ранжированная или упорядоченная система абстракций, расположение их по уровням.

Основными видами иерархических структур применительно к сложным системам являются структура классов (иерархия по номенклатуре) и структура объектов (иерархия по составу). Примерами иерархии классов являются простое и множественное наследование (один класс использует структурную или функциональную часть соответственно одного или нескольких других классов), а иерархии объектов - агрегация.

Классы можно организовать в виде иерархической структуры, которая по внешнему виду напоминает схему классификации в понятийной логике. Иерархия понятий строится следующим образом. В качестве наиболее общего понятия или категории берется понятие, имеющее наибольший объем и, соответственно, наименьшее содержание. Это самый высокий уровень абстракции для данной иерархии. Затем данное общее понятие конкретизируется, то есть уменьшается его объем и увеличивается содержание. Появляется менее общее понятие, которое на схеме иерархии будет расположено на один уровень ниже исходного. Этот процесс конкретизации понятий может быть продолжен до тех пор, пока на самом нижнем уровне не будет получено понятие, дальнейшая конкретизация которого в данном контексте либо невозможна, либо нецелесообразна

1. Экземпляры класса

**Экземпляр** **класса** – это **класс**, **экземпляр** объекта типа **класс**, или **экземпляр** данного типа **классов** (Но не **экземпляр** этого **класса**). Данные высказывания сделаны в парадигме Аристотеля, где есть типы объектов и **экземпляры** этих типов, но не объекты и **экземпляры** этих объектов.

Перечислите и охарактеризуйте уровни доступа к членам класса.

Все поля, методы и остальные компоненты класса имеют модификаторы доступа. Модификаторы доступа позволяют задать допустимую область видимости для компонентов класса. То есть модификаторы доступа определяют контекст, в котором можно употреблять данную переменную или метод.

В языке C# применяются следующие модификаторы доступа:

* private: закрытый или приватный компонент класса или структуры. Приватный компонент доступен только в рамках своего класса или структуры.
* private protected: компонент класса доступен из любого места в своем классе или в производных классах, которые определены в той же сборке.
* file: добавлен в версии C# 11 и применяется к типам, например, классам и структурам. Класс или структура с такми модификатором доступны только из текущего файла кода.
* protected: такой компонент класса доступен из любого места в своем классе или в производных классах. При этом производные классы могут располагаться в других сборках.
* internal: компоненты класса или структуры доступен из любого места кода в той же сборке, однако он недоступен для других программ и сборок.
* protected internal: совмещает функционал двух модификаторов protected и internal. Такой компонент класса доступен из любого места в текущей сборке и из производных классов, которые могут располагаться в других сборках.
* public: публичный, общедоступный компонент класса или структуры. Такой компонент доступен из любого места в коде, а также из других программ и сборок.

В С++

* private Члены класса, объявленные как private могут использоваться только функциями-членами и друзьями (классами или функциями) класса.
* protected Члены класса, объявленные как protected могут использоваться функциями-членами и друзьями (классами или функциями) класса. Кроме того, они могут использоваться производными классами данного класса.
* public Члены класса, объявленные как public могут использоваться любой функцией.

1. Средства инкапсуляции C++.

По умолчанию, в классе (class) данные и методы приватные (private); они могут быть прочитаны и изменены только классом к которому принадлежат. Уровень доступа может быть изменен при помощи соответствующих ключевых слов которые предоставляет Си++.

В Си++ доступно несколько спецификаторов, и они изменяют доступ к данным следующим образом:

* публичные (public) данные — доступны всем;
* защищенные (protected) — доступны только классу и дочерним классам;
* приватные (private) —доступны только классу которому они принадлежат.

Для краткости, только два уровня (приватный и публичный) будут освещены в примерах.

Наследование

Наследование (inheritance) представляет один из ключевых аспектов объектно-ориентированного программирования, который позволяет наследовать функциональность одного класса или базового класса (base class) в другом - производном классе (derived class).

class Person

{

public:

std::string name; // имя

int age; // возраст

void display()

{

std::cout << "Name: " << name << "\tAge: " << age << std::endl;

}

};

class Employee : public Person

{

public:

std::string company; // компания

};

Стоит учитывать, что конструкторы при наследовании не наследуются. И если базовый класс содержит только конструкторы с параметрами, то производный класс должен вызывать в своем конструкторе один из конструкторов базового класса. Например, добавим в классы выше конструкторы:

#include <iostream>

#include <string>

class Person

{

public:

Person(std::string n, int a)

{

name = n; age = a;

}

void display()

{

std::cout << "Name: " << name << "\tAge: " << age << std::endl;

}

private:

std::string name;

int age;

};

class Employee : public Person

{

public:

Employee(std::string n, int a, std::string c):Person(n, a)

{

company = c;

}

private:

std::string company;

};

int main()

{

Person tom("Tom", 23);

tom.display();

Employee bob("Bob", 31, "Microsoft");

bob.display();

return 0;

}

1. Перегрузка операторов

Перегрузка операторов позволяет определить действия, которые будет выполнять оператор. Перегрузка подразумевает создание функции, название которой содержит слово operator и символ перегружаемого оператора. Функция оператора может быть определена как член класса, либо вне класса.

Перегрузить можно только те операторы, которые уже определены в C++. Создать новые операторы нельзя.

Если функция оператора определена как отдельная функция и не является членом класса, то количество параметров такой функции совпадает с количеством операндов оператора. Например, у функции, которая представляет унарный оператор, будет один параметр, а у функции, которая представляет бинарный оператор, - два параметра. Если оператор принимает два операнда, то первый операнд передается первому параметру функции, а второй операнд - второму параметру. При этом как минимум один из параметров должен представлять тип класса

1. Исключение – это проблема, возникающая во время выполнения программы. Исключение C++ – это реакция на особые обстоятельства, возникающие во время выполнения программы, такие как попытка деления на ноль.

Исключения позволяют передавать управление из одной части программы в другую. Обработка исключений C++ построена на трех ключевых словах: try, catch и throw.

throw-программа выдает исключение, когда появляется проблема. Это делается с помощью ключевого слова throw.

catch-программа ловит исключение с помощью обработчика исключений в том месте программы, где вы хотите обработать проблему. Ключевое слово catch указывает на перехват исключения.

try-блок try определяет блок кода, для которого будут активированы определенные исключения. За ним следует один или несколько блоков catch.

Предполагая, что блок вызовет исключение, метод ловит исключение, используя комбинацию ключевых слов try и catch. Блок try/catch помещается вокруг кода, который может генерировать исключение. Код внутри блока try/catch называется защищенным кодом, а синтаксис использования try/catch выглядит следующим образом:

try {

   // protected code

} catch( ExceptionName e1 ) {

   // catch block

} catch( ExceptionName e2 ) {

   // catch block

} catch( ExceptionName eN ) {

   // catch block

}

1. Шаблоны классов и шаблоны функций

Шаблоны позволяют определить конструкции (функции, классы), которые используют определенные типы, но на момент написания кода точно не известно, что это будут за типы. Иными словами, шаблоны позволяют определить универсальные конструкции, которые не зависят от определенного типа.

Шаблон класса (class template) позволяет задать тип для объектов, используемых в классе.

Для применения шаблонов перед классом указывается ключевое слово template, после которого идут угловые скобки. В угловых скобках после слова typename идет параметр шаблона. Можно определить несколько параметров шаблона, в примере выше применяется только один параметр.

Параметр шаблона представляет произвольный идентификатор, в качестве которого, как правило, применяюся заглавные буквы, например, T. Но это необязательно. То есть в данном случае параметр T будет представлять некоторый тип, который становится известным во время компиляции. Это может быть и тип int, и double, и string, и любой другой тип. И теперь идентификатор счета будет представлять тип, который передается через параметр T.

#include <iostream>

#include <string>

template <typename T>

class Account {

private:

T id;

public:

Account(T id) : id(id)

{ }

T getId() {

return id;

}

};

int main()

{

Account<std::string> acc1("ca-pub1343767");

Account<int> acc2(1234566);

std::cout << "acc1: " << acc1.getId() << std::endl;

std::cout << "acc2: " << acc2.getId() << std::endl;

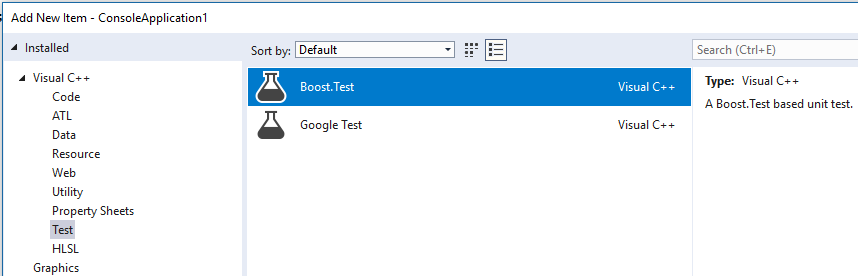
return 0;

}

1. Реализация в С++: Boost Test Library, Google test, Framework UnitTest. Реализация юнит тестирования на примере другого ЯП.

**Создание элемента Boost.Test**

1. Чтобы создать *CPP*-файл для тестов, щелкните правой кнопкой мыши узел проекта в **обозревателе решений** и выберите пункт **Добавить**>**Новый элемент**.
2. В диалоговом окне **Добавление нового элемента** последовательно разверните узлы **Установлено**>**Visual C++**>**Тест**. Выберите **Boost.Test** и затем **Добавить**, чтобы добавить *Test.cpp* в проект.



Новый файл *Test.cpp* содержит пример метода теста. В этот файл можно включить собственные файлы заголовков и записать тесты для приложения.

Этот тестовый файл также использует макросы для определения новой подпрограммы main для конфигураций тестов. Если выполнить сборку проекта сейчас, выводится ошибка LNK2005, например "\_main уже определен в main.obj".

**Создание и изменение конфигураций сборки**

1. Чтобы создать конфигурацию теста, в строке меню выберите **Сборка**>**Диспетчер конфигураций**. В диалоговом окне **Диспетчер конфигураций** выберите в раскрывающемся списке **Активная конфигурация решения** пункт **Создать**. В диалоговом окне **Создание конфигурации решения** введите имя, например "Debug UnitTests". В поле **Копировать параметры из** выберите **Отладка**, а затем нажмите кнопку **ОК**.
2. Исключите тестовый код из конфигураций отладки и выпуска: в **обозревателе решений** щелкните правой кнопкой мыши файл Test.cpp и выберите пункт **Свойства**. В диалоговом окне **Страницы свойств** откройте раскрывающийся список **Конфигурация** и выберите **Все конфигурации**. Выберите **Свойства конфигурации**>**Общие** и откройте раскрывающийся список для свойства **Исключено из сборки**. Выберите **Да**, а затем нажмите кнопку **Применить**, чтобы сохранить изменения.
3. Чтобы включить код теста в конфигурацию "Debug UnitTests", в диалоговом окне **Страницы свойств** выберите **Debug UnitTests** в раскрывающемся списке **Конфигурация**. Выберите **Нет** в свойстве **Исключено из сборки** и нажмите кнопку **ОК** для сохранения изменений.
4. Исключите основной код из конфигурации "Debug UnitTests". В **обозревателе решений** щелкните правой кнопкой мыши файл, содержащий функцию main, и выберите пункт **Свойства**. В диалоговом окне **Страницы свойств** откройте раскрывающийся список **Конфигурация** и выберите **Debug UnitTests**. Выберите **Свойства конфигурации**>**Общие** и откройте раскрывающийся список для свойства **Исключено из сборки**. Выберите **Да**, а затем нажмите кнопку **ОК**, чтобы сохранить изменения.
5. Выберите конфигурацию решения **Debug UnitTests**, а затем создайте проект, чтобы **обозреватель тестов** смог обнаружить этот метод.

При условии, что создаваемое имя конфигурации начинается со слова "Debug" или "Release", соответствующие библиотеки Boost.Test подбираются автоматически.

Шаблон элемента использует вариант Boost.Test с одним заголовком, но вы можете изменить путь в #include, чтобы указать автономный вариант библиотеки. Дополнительные сведения см. в разделе [Добавление директив include](https://learn.microsoft.com/ru-ru/visualstudio/test/how-to-use-boost-test-for-cpp?view=vs-2022#add-include-directives).

**Создание отдельного тестового проекта**

Во многих случаях для тестов проще использовать отдельный проект. Вам не нужно будет создавать специальную конфигурацию теста для своего проекта. Или исключите тестовые файлы из сборок отладки и выпуска.

**Создание отдельного тестового проекта**

1. В **обозревателе решений** щелкните узел решения правой кнопкой мыши и выберите пункты **Добавить**>**Новый проект**.
2. В диалоговом окне **Добавить новый проект** выберите **C++**, **Windows** и **Консоль** в раскрывающихся списках фильтров. Выберите шаблон **Консольное приложение** и нажмите кнопку **Далее**.
3. Присвойте проекту имя и нажмите кнопку **Создать**.
4. Удалите функцию main из *CPP*-файла.
5. Если вы используете версию Boost.Test с одним заголовком или динамическую версию библиотеки, перейдите к разделу [Добавление директив #include](https://learn.microsoft.com/ru-ru/visualstudio/test/how-to-use-boost-test-for-cpp?view=vs-2022#add-include-directives). Если вы используете статическую версию библиотеки, вам нужно выполнить некоторые дополнительные настройки:

а. Чтобы изменить файл проекта, сначала выгрузите его. В **обозревателе решений** щелкните узел проекта правой кнопкой мыши и выберите пункт **Выгрузить проект**. Затем щелкните правой кнопкой мыши узел проекта и выберите **Изменить <имя>.vcxproj**.

b. Добавьте две строки в группу свойств **Глобальные**, как показано ниже:

XMLКопировать

<PropertyGroup Label="Globals">

....

<VcpkgTriplet>x86-windows-static</VcpkgTriplet>

<VcpkgEnabled>true</VcpkgEnabled>

</PropertyGroup>

c. Сохраните и закройте файл *\*.vcxproj*, а затем перезагрузите проект.

d. Чтобы открыть диалоговое окно **Страницы свойств**, щелкните узел проекта правой кнопкой мыши и выберите пункт **Свойства**.

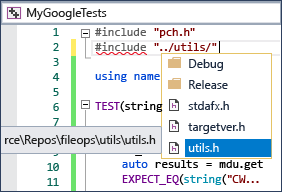
д) Разверните узел **C/C++**>**Создание кода**, а затем выберите элемент **Библиотека времени выполнения**. Выберите **/MTd** в качестве статической библиотеки времени выполнения для отладки или **/MT** в качестве статической библиотеки времени выполнения для выпуска.

е) Разверните узел **Компоновщик**>**Система**. Убедитесь, что для параметра **SubSystem** задано значение **Консоль**.

ж. Нажмите кнопку **ОК**, чтобы закрыть окно страниц свойств.

**Добавление директив include**

1. В *CPP*-файле теста добавьте необходимые директивы #include, чтобы типы и функции программы были доступны коду теста. Если вы используете отдельный тестовый проект, как правило, программа находится в иерархии папок на том же уровне. При вводе #include "../" открывается окно IntelliSense, в котором можно выбрать полный путь к файлу заголовка.



Можно использовать автономную библиотеку с:

C++Копировать

#include <boost/test/unit\_test.hpp>

Или использовать версию с одним заголовком с:

C++Копировать

#include <boost/test/included/unit\_test.hpp>

Затем определите BOOST\_TEST\_MODULE.

Приведенного ниже примера достаточно для обнаружения теста в **обозревателе тестов**.

C++Копировать

#define BOOST\_TEST\_MODULE MyTest

#include <boost/test/included/unit\_test.hpp> //single-header

#include "../MyProgram/MyClass.h" // project being tested

#include <string>

BOOST\_AUTO\_TEST\_CASE(my\_boost\_test)

{

std::string expected\_value = "Bill";

// assume MyClass is defined in MyClass.h

// and get\_value() has public accessibility

MyClass mc;

BOOST\_CHECK(expected\_value == mc.get\_value());

}

**Написание и запуск тестов**

Все готово к написанию и выполнению тестов Boost. Сведения о макросах тестов см. в [документации по библиотеке Boost Test](https://www.boost.org/doc/libs/1_71_0/libs/test/doc/html/index.html). Сведения об обнаружении, выполнении и группировании тестов с помощью **обозревателя тестов** см. в статье [Выполнение модульных тестов с помощью обозревателя тестов](https://learn.microsoft.com/ru-ru/visualstudio/test/run-unit-tests-with-test-explorer?view=vs-2022).

Для описания тестовых случаев применяется макрос BOOST\_AUTO\_TEST\_CASE, содержащий имя и код теста. Код теста содержит специальные макросы, проверяющие соответствие фактических результатов работы функции ожидаемым. Макросы отличаются уровнем предупреждения (CHECK — ошибка, REQUIRE — критическая ошибка, WARN — предупреждение). Среди макросов есть следующие:

1. BOOST\_CHECK(условие) — сообщает об ошибке, если условие ложно;

2. BOOST\_REQUIRE\_EQUAL(аргумент\_1, аргумент\_2) — сообщает о критической ошибке, если аргумент\_1 не равен аргумент\_2;

3. BOOST\_WARN\_MESSAGE(условие, сообщение) — выводит предупреждение с текстом сообщения, если условие ложно;

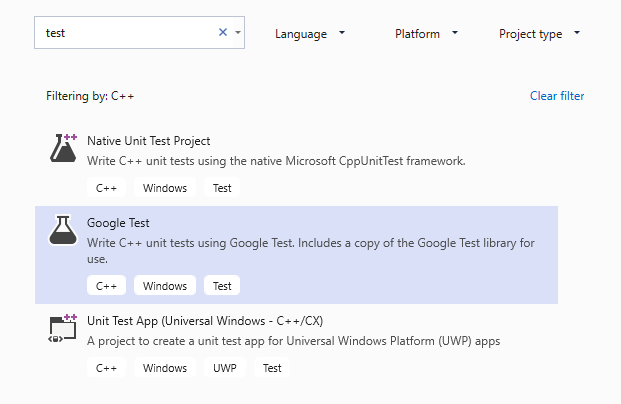
4. BOOST\_CHECK\_NO\_THROW(выражение) — сообщает об ошибке, если при вычислении выражения вырабатывается исключение;

5. BOOST\_CHECK\_THROW(выражение, исключение) — сообщает об ошибке, если при вычислении выражения не вырабатывается исключение требуемого типа;

6. BOOST\_CHECK\_CLOSE\_FRACTION(аргумент\_1, аргумент\_2, погрешность) — проваливает тест если аргумент\_1 не равен аргумент\_2 с заданной погрешностью.

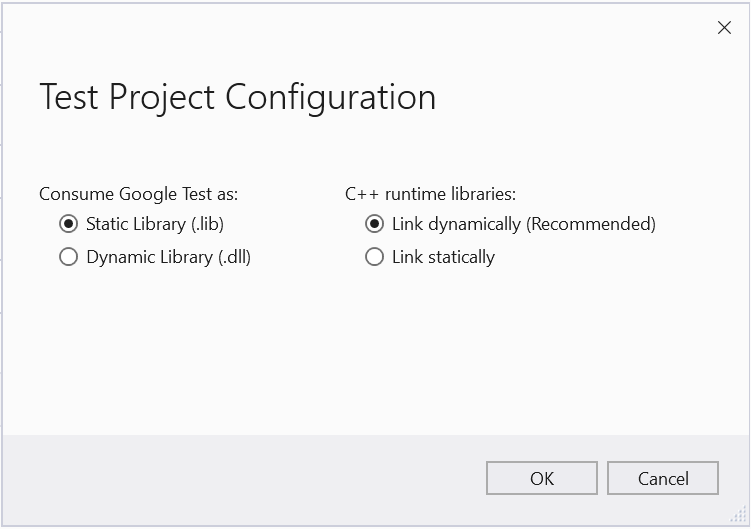
**Добавление проекта Google Test в Visual Studio 2019**

1. В **обозревателе решений** щелкните узел решения правой кнопкой мыши и выберите пункты **Добавить**>**Новый проект**.
2. Задайте **Язык** как **C++** и введите **тест** в поле поиска. Выберите в списке результатов **Проект Google Test**.
3. Укажите имя тестового проекта и нажмите кнопку **ОК**.



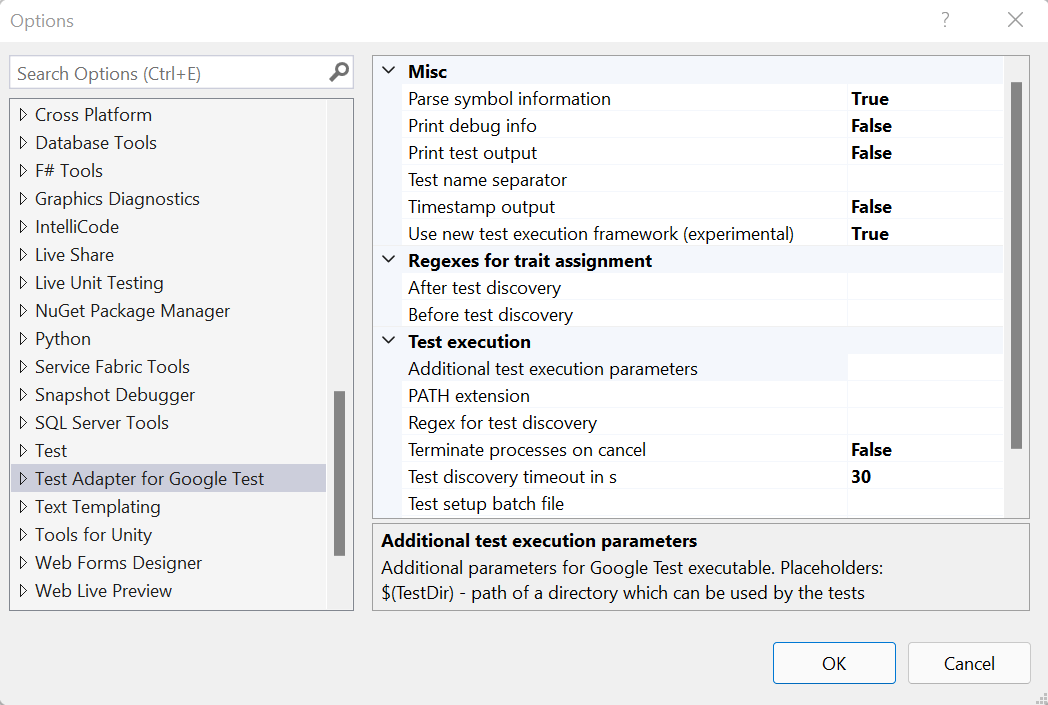
**Настройка тестового проекта**

В открывшемся диалоговом окне **Конфигурация тестового проекта** можно выбрать проект, который необходимо тестировать. При выборе проекта Visual Studio добавляет ссылку на него. Если проект не выбран, необходимо вручную добавить ссылки на проекты, которые следует тестировать. При выборе статического или динамического связывания с двоичными файлами Google Test следует учитывать те же факторы, что и в случае с любой другой программой C++. Дополнительные сведения см. в статье [DLL в Visual C++](https://learn.microsoft.com/ru-ru/cpp/build/dlls-in-visual-cpp).



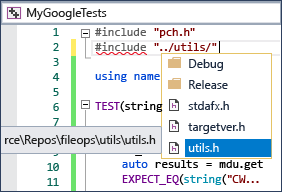
**Настройка дополнительных параметров**

Чтобы настроить дополнительные параметры, в главном меню выберите **Сервис**>**Параметры**>**Адаптер тестов для Google Test**. Дополнительные сведения об этих параметрах см. в документации по Google Test.



**Добавление директив include**

В *CPP*-файле теста добавьте необходимые директивы #include, чтобы типы и функции программы были доступны коду теста. Как правило, программа находится в иерархии папок на один уровень выше. Если ввести #include "../", появится окно IntelliSense, в котором можно выбрать полный путь к файлу заголовка.



Паттерн AAA

• Arrange – секция настройки теста: o Создание переменных o Формирование данных

• Act - секция выполнения действия (вызов метода)

• Assert – секция проверки утверждения (валидация результатов)

• Фатальные (ASSERT\_)

• Не фатальные (EXPECT\_)

Простейшие логические

• ASSERT\_TRUE(condition);

• ASSERT\_FALSE(condition);

Сравнения

• ASSERT\_EQ(actual, expected); — equal (=)

• ASSERT\_NE(val1, val2); — not equal (!=)

• ASSERT\_LT(val1, val2); — less then ( <=)

• ASSERT\_GT(val1, val2); — greater then (>)

• ASSERT\_GE(val1, val2); — greater equal (>=)

• ASSERT\_STREQ(actual, expected); — регистрозависимо

• ASSERT\_STRNE(str1, str2); — регистрозависимо

• ASSERT\_STRCASEEQ(actual, expected); — регистронезависимо

• ASSERT\_STRCASENE(str1, str2); — регистронезависимо

Сравнение чисел с плавающей точкой

• ASSERT\_FLOAT\_EQ(actual, expected); — неточное сравнение float

• ASSERT\_DOUBLE\_EQ(actual, expected); — неточное сравнение double

• ASSERT\_NEAR(val1, val2, abs\_error); — разница между val1 и val2 не превышает погрешность abs\_error

1. Наследование классов ошибок.

Поскольку можно выбрасывать объекты классов, в качестве исключений, а классы могут быть производными от других классов, нам необходимо учитывать, что происходит, когда в качестве исключений мы используем наследованные классы. Оказывается, обработчики исключений будут не только соответствовать классам определенного типа, они также будут соответствовать классам, производным от этого конкретного типа!

1. Механизм сериализации и десериализации

Сериализация имеющихся объектов – это процесс сохранения состояния объекта в последовательность байт. Механизм, который позволят «записывать» полученный прогресс для будущей выгрузки.

Десериализация – восстановление объектов из байт, сохранение которых было произведено ранее. Процедура выгрузки «зафиксированной» информации пользователем.

1. Бинарная (базовая) сериализация и десериализация

Сериализация — это процесс хранения состояния объекта на носителе хранилища. При двоичной сериализации открытые и закрытые поля объекта и имя класса, включая сборку, содержащую класс, преобразуются в поток байтов, который затем записывается в поток данных. После десериализации объекта создается точная копия исходного объекта.

Person person1 = new Person("Tom", 29);

Person person2 = new Person("Bill", 25);

// массив для сериализации

Person[] people = new Person[] { person1, person2 };

BinaryFormatter formatter = new BinaryFormatter();

using (FileStream fs = new FileStream("people.dat", FileMode.OpenOrCreate))

{

// сериализуем весь массив people

formatter.Serialize(fs, people);

Console.WriteLine("Объект сериализован");

}

// десериализация

using (FileStream fs = new FileStream("people.dat", FileMode.OpenOrCreate))

{

Person[] deserilizePeople = (Person[])formatter.Deserialize(fs);

foreach (Person p in deserilizePeople)

{

Console.WriteLine($"Имя: {p.Name} --- Возраст: {p.Age}");

}

}

1. Json сериализация и десериализация

Пространство имен System.Text.Json предоставляет функциональные возможности для сериализации в нотацию объектов JavaScript (JSON) и десериализации объектов из этой нотации. *Сериализация* — это процесс преобразования состояния объекта, то есть значений его свойств в форму, которая может храниться или передаваться. Сериализованная форма не содержит никаких сведений о связанных методах объекта. *Десериализация* воссоздает объект из сериализованной формы.

using System.Text.Json;

// сохранение данных

using (FileStream fs = new FileStream("user.json", FileMode.OpenOrCreate))

{

Person tom = new Person("Tom", 37);

await JsonSerializer.SerializeAsync<Person>(fs, tom);

Console.WriteLine("Data has been saved to file");

}

// чтение данных

using (FileStream fs = new FileStream("user.json", FileMode.OpenOrCreate))

{

Person? person = await JsonSerializer.DeserializeAsync<Person>(fs);

Console.WriteLine($"Name: {person?.Name} Age: {person?.Age}");

}

class Person

{

public string Name { get;}

public int Age { get; set; }

public Person(string name, int age)

{

Name = name;

Age = age;

}

}

using System.Text.Json;

Person tom = new Person("Tom", 37);

var options = new JsonSerializerOptions

{

WriteIndented = true

};

string json = JsonSerializer.Serialize<Person>(tom, options);

Console.WriteLine(json);

Person? restoredPerson = JsonSerializer.Deserialize<Person>(json);

Console.WriteLine(restoredPerson?.Name);

class Person

{

[JsonPropertyName("firstname")]

public string Name { get;}

[JsonIgnore]

public int Age { get; set; }

public Person(string name, int age)

{

Name = name;

Age = age;

}

}

JsonPropertyName – позволяет замещать оригинальные названия свойства

JsonIgnore – позволяет исключать из сериализации определённые свойства

1. Xml сериализация и десериализация

Сериализация представляет собой процесс преобразования объекта в форму, подготовленную для передачи. Например, можно сериализовать объект и передать его по Интернету с использованием протокола HTTP между клиентом и сервером. И наоборот, при десериализации объект воссоздается из потока.

При XML-сериализации в поток XML сериализуются только открытые поля и значения свойств объекта. XML-сериализация не учитывает информацию о типе. Например, если имеется объект **Book**, который существует в пространстве имен **Library**, нет никакой гарантии, что он десериализуется в объект аналогичного типа.

Person[] people = new Person[]

{

new Person("Tom", 37),

new Person("Bob", 41)

};

XmlSerializer formatter = new XmlSerializer(typeof(Person[]));

// ?????????? ??????? ? ????

using (FileStream fs = new FileStream("people.xml", FileMode.OpenOrCreate))

{

formatter.Serialize(fs, people);

}

// ?????????????? ??????? ?? ?????

using (FileStream fs = new FileStream("people.xml", FileMode.OpenOrCreate))

{

Person[]? newpeople = formatter.Deserialize(fs) as Person[];

if(newpeople != null)

{

foreach (Person person in newpeople)

{

Console.WriteLine($"Name: {person.Name} --- Age: {person.Age}");

}

}

}

public class Person

{

public string Name { get; set; } = "Undefined";

public int Age { get; set; } = 1;

public Person() { }

public Person(string name, int age)

{

Name = name;

Age = age;

}

}

1. Принцип реализации дружественных функций. Сравнительный анализ с реализацией на принципах get/set.

Дружественные функции - это функции, которые не являются членами класса, однако имеют доступ к его закрытым членам - переменным и функциям, которые имеют спецификатор private.

Для определения дружественных функций используется ключевое слово friend.

friend void Person::drive(Auto &);

1. Наследование и полиморфизм

В C ++ конструкторы и деструкторы не наследуются. Однако они вызываются, когда дочерний класс инициализирует свой объект. Конструкторы вызываются один за другим иерархически, начиная с базового класса и заканчивая последним производным классом. Деструкторы вызываются в обратном порядке.

**Полиморфизм** — одна из трех основных парадигм ООП. Если говорить кратко, полиморфизм — это способность обьекта использовать методы производного класса, который не существует на момент создания базового.

**Наследование** – это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым или родительским. Новый класс – потомком, наследником или производным классом.

**Указателю** **на** **базовый** **класс** **можно** **присвоить** значение **адреса** **объекта** любого **производного** **класса**, например: monster \*p; // Описывается **указатель** **на** **базовый** **класс** p = new daemon; /\* **Указатель** ссылается на **объект** **производного** **класса** \*/. Вызов методов **объекта** происходит в соответствии с типом **указателя**, а не фактическим типом **объекта**, на который он ссылается, поэтому при выполнении оператора, например.

Вообще говоря, переменная ссылки на объект может ссылаться только на объект своего типа.

Но из этого принципа строгого соблюдения типов в C# имеется одно важное исключение: переменной ссылки на объект базового класса может быть присвоена ссылка на объект любого производного от него класса.

Использование умных указателей

**Умный** **указатель** (англ. **smart** pointer) — идиома косвенного обращения к памяти, которая широко используется при программировании на высокоуровневых языках как: C++, Rust и так далее. Как правило, реализуется в виде специализированного класса (обычно — параметризованного), имитирующего интерфейс обычного **указателя** и добавляющего необходимую новую функциональность (например — проверку границ при доступе или очистку памяти).

Все умные указатели доступны через включение соответствующего хедера (#include <memory>). Для краткости во фрагментах кода внутри статьи это было опущено.

X\* ptr = new X();

if (func()) {

func2();

return;

}

delete ptr;

1. Понятие многопоточности, сферы применения

#Multithreading – мультипоточность

**Многозадачность (multitasking)** — свойство операционной системы или среды выполнения обеспечивать возможность параллельной (или псевдопараллельной) обработки нескольких задач.

**Многопоточность (multithreading)** — свойство платформы (например, операционной системы, виртуальной машины и т. д.) или приложения, состоящее в том, что **процесс**, порождённый в операционной системе, может состоять из нескольких **потоков**, выполняющихся «параллельно», то есть без предписанного порядка во времени. При выполнении некоторых задач такое разделение может достичь более эффективного использования ресурсов вычислительной машины.

Потоки:

* Single-threading – однопоточность(последовательное выполнение), следующая задача не выполнится, пока предыдущая не завершится.
* Hyper – threading – разделение задач на подзадачи для ускорения и оптимизации (не многопоточность).

Hyper – threading - это технология разработанная компанией Intel для повышения производительности процессоров собственного производства.

Принцип действия Hyper-Threading основывается на том, что в каждый момент времени только часть ресурсов процессора используется при выполнении программного кода. Неиспользуемые ресурсы также можно загрузить работой — например, задействовать для параллельного выполнения еще одного приложения (либо другого потока этого же приложения). В одном физическом процессоре Intel Xeon формируются два логических процессора (LP — Logical Processor), которые разделяют между собой вычислительные ресурсы CPU. Операционная система и приложения "видят" именно два CPU и могут распределять работу между ними, как и в случае полноценной двухпроцессорной системы.

Физическое ядро не будет бездействовать, а передаст управление потоку команд второго логического ядра. Таким образом, пока одно логическое ядро ожидает, например, данные из памяти, вычислительные ресурсы физического ядра будут использованы вторым логическим ядром.

* Multi – threading – многопоточность.

Multi – threading – свойство платформы или приложения, состоящее в том, что процесс, порождённый в операционной системе, может состоять из нескольких потоков, выполняющихся «параллельно», то есть без предписанного порядка во времени. При выполнении некоторых задач такое разделение может достичь более эффективного использования ресурсов вычислительной машины.

**Многопоточность** (как доктрину программирования) не следует путать ни с многозадачностью, ни с многопроцессорностью несмотря на то, что операционные системы, реализующие многозадачность, как правило, реализуют и многопоточность.

**Многопоточность имеет большие преимущества:**

* **Увеличение скорости**(по сравнению с использованием обычных процессов). Многопоточность основана на использовании **облегченных процессов (lightweight processes),**работающих в общем пространстве виртуальной памяти. Благодаря многопоточности, не возникает больше неэффективных ситуаций, типичных для классической системе UNIX, в которой каждая команда shell (даже команда вывода содержимого текущей директории **ls**исполнялась как **отдельный процесс**, причем в своем собственном адресном пространстве. В противоположность облегченным процессам, обычные процессы (имеющие собственное адресное пространство) часть называют **тяжеловесными (heavyweight).**
* **Использование общих ресурсов**. Потоки одного процесса используют общую память и файлы.
* **Экономия**. Благодаря многопоточности, достигается значительная экономия памяти, по причинам, объясненным выше. Также достигается и экономия времени, так как переключение контекста на облегченный процесс, для которого требуется только сменить стек и восстановить значения регистров, значительно быстрее, чем на обычный процесс (см. "Методы взаимодействия процессов ").

**Недостатки многопоточности**

* Увеличивает сложность программы.
* Необходима синхронизация общих ресурсов (объектов, данных).
* Трудно отлаживать непредсказуемые результаты
* Создание и синхронизация потоков требует больших затрат ресурсов процессора и памяти.

Пример:

#include <thread>

void function1()

{

cout << this\_thread::get\_id() << endl;

}

thread th1(function1);

th1.detach();//первый вариант завершения потока – завершение нового потока, при завершении основного.

//main thread

th1.join();// второй вариант завершения потока – ожидает завершения thread1.

Thread th1 (function1);

Thread th1 (function1, parameter1, parameter2);

Thread th1 (function1, std::ref(parameter1));//ref – класс ссылок

Thread th1 ([&result](){result = function1(parameter1,patameter2);});

«4 потока -одна консоль» - mutex.

Mutex – приметив синхронизации., обеспечивает взаимное исключение критических участков кода.

#include < thread>

Mutex mtx;

Mtx.lock();

Function1();

Mtx.unlock();

#include < thread>

Mutex mtx;

Void function1() //одновременно выполняется ток один поток, пока один поток не завершит выполнение, второй не начнет работать

{

Mtx.lock();

cout << this\_thread::get\_id() << endl;

mtx.unlock();

};

thread th1(function1);

th1.detach();

thread th2(function1);

th2.detach();

// main thread

#include < thread>

Mutex mtx;

Void function1()

{

Lock\_ guard<mutex>guard(mtx);//объяв без закрытия, закрывается при завершении функции

cout << this\_thread::get\_id() << endl;

};

thread th1(function1);

th1.detach();

thread th2(function1);

th2.detach();

// main thread

Recursive\_mutex rmtx;

Rmtx.lock();

Rmtx.unlock();

Uniqu\_lock<mutex> ul(mtx);

Unique\_lock<mutex> ul(mtx, std::defer\_lock);

Ul.lock();

Ul.unlock();

using System.Threading;

// создаем новый поток

Thread myThread1 = new Thread(Print);

Thread myThread2 = new Thread(new ThreadStart(Print));

Thread myThread3 = new Thread(()=>Console.WriteLine("Hello Threads"));

myThread1.Start(); // запускаем поток myThread1

myThread2.Start(); // запускаем поток myThread2

myThread3.Start(); // запускаем поток myThread3

void Print() => Console.WriteLine("Hello Threads");

**Сферы применения многопоточности**

С развитием компьютерных технологий перечень программ, использующих многопоточность, неуклонно растет. Это дает огромный простор разработчикам для создания нового софта и игр. Например, сейчас каждый современный triple-A проект оптимизирован для многопоточных процессоров, что позволяет наслаждаться игрой, получая высокий уровень fps на многоядерном CPU.

Еще больше распространены многоядерные системы в среде разработчиков. Программы для 3D-моделирования, монтажа видео и создания музыки требуют параллельного выполнения большого количества задач, с чем хорошо справляются системы с Hyper-Threading или SMT. В операционных системах мощность одного потока может тратиться на фоновые задачи (Skype, браузер, мессенджер), в то время как остальные задействуются для тяжелой игры или программы.

1. Понятие потока и процесса

Процесс — экземпляр программы во время выполнения, независимый объект, которому выделены системные ресурсы (например, процессорное время и память). Каждый процесс выполняется в отдельном адресном пространстве: один процесс не может получить доступ к переменным и структурам данных другого. Если процесс хочет получить доступ к чужим ресурсам, необходимо использовать межпроцессное взаимодействие. Это могут быть конвейеры, файлы, каналы связи между компьютерами и многое другое.

Поток использует то же самое пространства стека, что и процесс, а множество потоков совместно используют данные своих состояний. Как правило, каждый поток может работать (читать и писать) с одной и той же областью памяти, в отличие от процессов, которые не могут просто так получить доступ к памяти другого процесса. У каждого потока есть собственные регистры и собственный стек, но другие потоки могут их использовать.

Поток — определенный способ выполнения процесса. Когда один поток изменяет ресурс процесса, это изменение сразу же становится видно другим потокам этого процесса.

**Конкурентность и параллелизм**

**Конкурентность —** это выполнение задач за определённое время (например, есть 5 процессов и все они в сумме выполняются в течение 60 минут по очереди). Важная деталь заключается в том, что задачи необязательно выполняются одновременно, поэтому их можно разделить на более мелкие и чередующиеся.

Проведём аналогию: шеф-повар нарезает лук и иногда проверяет духовку. Ему нужно прекратить нарезать лук, чтобы подойти к духовке, а затем снова начать нарезать и повторять этот процесс до конца приготовления блюда.

**Параллелизм** — это выполнение задач в одно и то же время (например, есть 5 задач, каждая из них выполняется в течение 60 минут). Само название подразумевает, что они выполняются параллельно.

Вернёмся на кухню: теперь у нас есть два шеф-повара. Один следит за духовкой, а второй нарезает лук. Работа разделена, так как теперь на кухне трудятся два повара.

Параллелизм — подкласс конкурентности: перед выполнением нескольких одновременных задач вам нужно их сначала правильно организовать.

Класс Thread определяет ряд методов и свойств, которые позволяют управлять потоком и получать информацию о нем. Основные свойства класса:

* ExecutionContext: позволяет получить контекст, в котором выполняется поток
* IsAlive: указывает, работает ли поток в текущий момент
* IsBackground: указывает, является ли поток фоновым
* Name: содержит имя потока
* ManagedThreadId: возвращает числовой идентификатор текущего потока
* Priority: хранит приоритет потока - значение перечисления ThreadPriority:
* Lowest
* BelowNormal
* Normal
* AboveNormal
* Highest

По умолчанию потоку задается значение Normal. Однако мы можем изменить приоритет в процессе работы программы. Например, повысить важность потока, установив приоритет Highest. Среда CLR будет считывать и анализировать значения приоритета и на их основании выделять данному потоку то или иное количество времени.

* ThreadState возвращает состояние потока - одно из значений перечисления ThreadState:
* Aborted: поток остановлен, но пока еще окончательно не завершен
* AbortRequested: для потока вызван метод Abort, но остановка потока еще не произошла
* Background: поток выполняется в фоновом режиме
* Running: поток запущен и работает (не приостановлен)
* Stopped: поток завершен
* StopRequested: поток получил запрос на остановку
* Suspended: поток приостановлен
* SuspendRequested: поток получил запрос на приостановку
* Unstarted: поток еще не был запущен
* WaitSleepJoin: поток заблокирован в результате действия методов Sleep или Join

В процессе работы потока его статус многократно может измениться под действием методов. Так, в самом начале еще до применения метода Start его статус имеет значение Unstarted. Запустив поток, мы изменим его статус на Running. Вызвав метод Sleep, статус изменится на WaitSleepJoin.

Кроме того статическое свойство CurrentThread класса Thread позволяет получить текущий поток

В программе на C# есть как минимум один поток - главный поток, в котором выполняется метод Main.

1. Основные примитивы синхронизации Mutex, Semaphore, SemaphoreSlim, Barrier, SpinLock, SpinWait, Monitor. Примеры. StopWatch как метод формирования оценки времени.

**Mutex**

Мьютекс (англ. *mutex*, от ***mut****ual****ex****clusion* — «взаимное исключение») — это базовый механизм синхронизации. Он предназначен для организации взаимоисключающего доступа к общим данным для нескольких потоков с использованием барьеров памяти (для простоты можно считать мьютекс дверью, ведущей к общим данным).

**Синтаксис**

* **Заголовочный файл** | **#include** <mutex>
* **Объявление** | **std::mutex** mutex\_name;
* **Захват мьютекса** | mutex\_name**.lock()**;  
  Поток запрашивает *монопольное использование*общих данных, защищаемых мьютексом. Дальше два варианта развития событий: происходит захват мьютекса этим потоком (и в этом случае ни один другой поток не сможет получить доступ к этим данным) или поток блокируется (если мьютекс уже захвачен другим потоком).
* **Освобождение мьютекса**| mutex\_name**.unlock()**;  
  Когда ресурс больше не нужен, текущий владелец должен вызвать функцию разблокирования unlock, чтобы и другие потоки могли получить доступ к этому ресурсу. Когда мьютекс освобождается, доступ предоставляется одному из ожидающих потоков.

**Semaphore**

Примитив синхронизации работы процессов и потоков, в основе которого лежит счётчик, над которым можно производить две атомарные операции: увеличение и уменьшение значения на единицу, при этом операция уменьшения для нулевого значения счётчика является блокирующейся. Служит для построения более сложных механизмов синхронизации и используется для синхронизации параллельно работающих задач, для защиты передачи данных через разделяемую память, для защиты критических секций, а также для управления доступом к аппаратному обеспечению.

#include <iostream>

#include <thread>

#include <chrono>

#include <semaphore>

// глобальные экземпляры двоичных семафоров

// счетчик объектов установлен в ноль

// объекты в несигнальном состоянии

std::binary\_semaphore

smphSignalMainToThread(0),

smphSignalThreadToMain(0);

void ThreadProc()

{

// ждем сигнала от main,

// пытаясь уменьшить значение семафора

smphSignalMainToThread.acquire();

// этот вызов блокируется до тех пор,

// пока счетчик семафора не увеличится в main

std::cout << "[thread] Got the signal\n"; // ответное сообщение

// ждем 3 секунды для имитации какой-то работы,

// выполняемой потоком

using namespace std::literals;

std::this\_thread::sleep\_for(3s);

std::cout << "[thread] Send the signal\n"; // сообщение

// сигнализируем обратно в main

smphSignalThreadToMain.release();

}

int main()

{

// создаем какой-то обрабатывающий поток

std::thread thrWorker(ThreadProc);

std::cout << "[main] Send the signal\n"; // сообщение

// сигнализируем рабочему потоку о начале работы,

// увеличивая значение счетчика семафора

smphSignalMainToThread.release();

// ждем, пока рабочий поток не выполнит свою работу,

// пытаясь уменьшить значение счетчика семафора

smphSignalThreadToMain.acquire();

std::cout << "[main] Got the signal\n"; // ответное сообщение

thrWorker.join();

}

**SemaphoreSlim**

[SemaphoreSlim](https://learn.microsoft.com/ru-ru/dotnet/api/system.threading.semaphoreslim) Класс представляет упрощенный, быстрый семафор, который можно использовать для ожидания внутри одного процесса, когда предполагается, что времена ожидания будут очень короткими. [SemaphoreSlim](https://learn.microsoft.com/ru-ru/dotnet/api/system.threading.semaphoreslim)использует максимально примитивы синхронизации, предоставляемые общеязыковой среды выполнения (CLR). Тем не менее, он также предоставляет неактивно инициализированные дескрипторы ожидания на основе ядра при необходимости поддержки ожидания для нескольких семафоров. [SemaphoreSlim](https://learn.microsoft.com/ru-ru/dotnet/api/system.threading.semaphoreslim)также поддерживает использование токенов отмены, но не поддерживает именованные семафоры или использование дескриптора ожидания для синхронизации.

**Barrier**

Защелки (latches) и барьеры (barriers) – это механизм синхронизации потоков, который позволяет блокировать любое количество потоков до тех пор, пока ожидаемое количество потоков не достигнет барьера. Защелки нельзя использовать повторно, барьеры можно использовать повторно.

Эти механизмы синхронизации используются, когда выполнение параллельного алгоритма можно разделить на несколько этапов, разделённых барьерами. В частности, с помощью барьера можно организовать точку сбора частичных результатов вычислений, в которой подводится итог этапа вычислений. Например, если стоит задача отфильтровать изображение с помощью двух разных фильтров, и разные потоки фильтруют разные части изображения, то перед началом второй фильтрации следует дождаться, когда первая фильтрация будет полностью завершена, то есть все потоки должны дойти до барьера между двумя этапами фильтрации.

Барьер для группы потоков в исходном коде означает, что каждый поток должен остановиться в этой точке и подождать достижения барьера другими потоками группы. Когда все потоки достигли барьера, их выполнение продолжается.

Итак, как пользоваться барьерами? В нашем распоряжении следующие методы:

* arrive(value) уменьшает текущее значение счётчика на value (по умолчанию 1). Поведение не определено, если значение счётчика станет отрицательным. Метод возвращает идентификатор фазы, который имеет тип arrival\_token.
* wait(arrival\_token) блокирует текущий поток до тех пор, пока указанная фаза не завершится. Принимает идентификатор фазы в качестве параметра.
* arrive\_and\_wait() уменьшает текущее значение счётчика на 1 и блокирует текущий поток до тех пор, пока счётчик не обнулится. Эквивалентно вызову wait(arrive());. Поведение не определено, если вызов происходит, когда значение счётчика равно нулю. Поэтому количество потоков, уменьшающих счётчик барьера, не должно быть больше начального значения счётчика.
* arrive\_and\_drop() уменьшает на 1 начальное значение счётчика для следующих фаз, а так же текущее значение счётчика. Поведение не определено, если вызов происходит, когда значение счётчика равно нулю.

int n\_threads;

std::vector<thread\*> workers;

std::barrier task\_barrier(n\_threads);

for (int i = 0; i < n\_threads; ++i) {

workers.push\_back(new thread([&] {

for(int step\_no = 0; step\_no < 5; ++step\_no) {

// выполнение фазы

...

task\_barrier.arrive\_and\_wait();

}

});

}

**SpinLock**

**Спин-блокировка** или **спинлок** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *spinlock* — циклическая блокировка) — низкоуровневый примитив [синхронизации](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D1%85%D1%80%D0%BE%D0%BD%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0))[[1]](https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D0%BD-%D0%B1%D0%BB%D0%BE%D0%BA%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0#cite_note-:0-1), применяемый в многопроцессорных системах для реализации взаимного исключения исполнения [критических участков](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B8%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%81%D0%B5%D0%BA%D1%86%D0%B8%D1%8F) кода с использованием цикла активного ожидания[[2]](https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D0%BD-%D0%B1%D0%BB%D0%BE%D0%BA%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0#cite_note-_45cfdae284faec09-2). Применяется в случаях, когда ожидание захвата блокировки предполагается недолгим[[2]](https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D0%BD-%D0%B1%D0%BB%D0%BE%D0%BA%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0#cite_note-_45cfdae284faec09-2) либо если контекст выполнения не позволяет переходить в заблокированное состояние[[3]](https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D0%BD-%D0%B1%D0%BB%D0%BE%D0%BA%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0#cite_note-:02-3).

Спин-блокировки являются аналогами [мьютексов](https://ru.wikipedia.org/wiki/%D0%9C%D1%8C%D1%8E%D1%82%D0%B5%D0%BA%D1%81), позволяющими тратить меньше времени на процедуру блокировки потока, поскольку не требуется переводить поток в заблокированное состояние. В случае мьютексов может потребоваться задействование планировщика с переводом потока в другое состояние и добавлением его в список потоков, ожидающих разблокировки. Спин-блокировки не задействуют планировщик и используют цикл активного ожидания без изменения состояния потока, что приводит к трате процессорного времени на ожидание освобождения блокировки другим потоком. Типовой реализацией спин-блокировки является простая циклическая проверка переменной спин-блокировки на доступность

struct Node {

int number;

std::atomic\_bool latch;

void add() {

lock();

number++;

unlock();

}

void lock() {

bool unlatched = false;

while(!latch.compare\_exchange\_weak(unlatched, true, std::memory\_order\_acquire));

}

void unlock() {

latch.store(false , std::memory\_order\_release);

}

};

**SpinWait**

[SpinWait](https://learn.microsoft.com/ru-RU/dotnet/api/system.threading.spinwait?view=netstandard-1.0) инкапсулирует общую логику вращения. На однопроцессорных компьютерах доходность всегда используется вместо занятых ожиданий, а на компьютерах с процессорами Intel, используюющими Hyper-Threading технологию, это помогает предотвратить нехватку аппаратных потоков. SpinWait инкапсулирует хорошую смесь спиннинга и истинного урожая.

[SpinWait](https://learn.microsoft.com/ru-RU/dotnet/api/system.threading.spinwait?view=netstandard-1.0) — это тип значения, что означает, что низкоуровневый код может использовать SpinWait, не опасаясь ненужных затрат на выделение ресурсов. SpinWait обычно не полезен для обычных приложений. В большинстве случаев следует использовать классы синхронизации, предоставляемые платформа .NET Framework, например [Monitor](https://learn.microsoft.com/ru-RU/dotnet/api/system.threading.monitor?view=netstandard-1.0).

SpinWait spinWait = new SpinWait();

// ...

spinWait.SpinOnce();

**Monitor**

Этот [Monitor](https://learn.microsoft.com/ru-ru/dotnet/api/system.threading.monitor?view=netcore-2.0) класс позволяет синхронизировать доступ к области кода, принимая и освобождая блокировку для определенного объекта, вызывая [Monitor.Enter](https://learn.microsoft.com/ru-ru/dotnet/api/system.threading.monitor.enter?view=netcore-2.0)[Monitor.TryEnter](https://learn.microsoft.com/ru-ru/dotnet/api/system.threading.monitor.tryenter?view=netcore-2.0) методы и [Monitor.Exit](https://learn.microsoft.com/ru-ru/dotnet/api/system.threading.monitor.exit?view=netcore-2.0). д. Блокировки объектов позволяют ограничить доступ к блоку кода, который обычно называется критическим разделом. Хотя поток владеет блокировкой для объекта, ни другой поток не может получить такую блокировку. Класс также можно использовать [Monitor](https://learn.microsoft.com/ru-ru/dotnet/api/system.threading.monitor?view=netcore-2.0) для обеспечения доступа к разделу кода приложения, выполняемого владельцем блокировки, если другой поток не выполняет код с помощью другого заблокированного объекта.

template<class T>

class monitor

{

public:

template<typename ...Args>

monitor(Args&&... args) : m\_cl(std::forward<Args>(args)...){}

struct monitor\_helper

{

monitor\_helper(monitor\* mon) : m\_mon(mon), m\_ul(mon->m\_lock) {}

T\* operator->() { return &m\_mon->m\_cl;}

monitor\* m\_mon;

std::unique\_lock<std::mutex> m\_ul;

};

monitor\_helper operator->() { return monitor\_helper(this); }

monitor\_helper ManuallyLock() { return monitor\_helper(this); }

T& GetThreadUnsafeAccess() { return m\_cl; }

private:

T m\_cl;

std::mutex m\_lock;

};

**StopWatch как метод формирования оценки времени.**

Класс Stopwatch основан на HPET (High Precision Event Timer, таймер событий высокой точности). Данный таймер был введён фирмой Microsoft, чтобы раз и навсегда поставить точку в проблемах измерения времени. Частота этого таймера (минимум 10 МГц) не меняется во время работы системы. Для каждой системы Windows сама определяет, с помощью каких устройств реализовать этот таймер.  
  
Класс Stopwatch содержит следующие поля:

**private** **const** long TicksPerMillisecond = 10000;

**private** **const** long TicksPerSecond = TicksPerMillisecond \* 1000;

**private** bool isRunning;

**private** long startTimeStamp;

**private** long elapsed;

**private** **static** **readonly** double tickFrequency;

* TicksPerMillisecond — определяет количество DateTime тактов в 1 миллисекунду;
* TicksPerSecond — определяет количество DateTime тактов в 1 секунду;
* isRunning — определяет, запущен ли текущий экземпляр (вызван ли был метод Start);
* startTimeStamp — число тактов на момент запуска;
* elapsed — общее число затраченных тактов;
* tickFrequency — упрощает перевод тактов Stopwatch в такты DateTime.

Статический конструктор проверяет наличие таймера HPET и в случае его отсутствия частота Stopwatch устанавливается равной частоте DateTime.

1. Понятие статической и динамической библиотек

По способу подключения к основной программе библиотеки можно разделить на два типа:

* **Динамические**— это файл с машинным кодом, который подключается во время исполнения. Его в любой момент можно заменить на другие. В этом одновременно и плюс — динамическую библиотеку можно обновить почти без труда, и минус — требуется ровно столько же усилий, чтобы заменить ее на что-нибудь вредоносное.
* **Статические** — это исходный код на языке программы или объектный модуль, который упаковывается в саму программу. Такую библиотеку очень сложно подменить, поэтому, чтобы обновить её, придётся заново компилировать всю программу.

**Порядок генерации библиотек и механизма использования**

**Для С++**

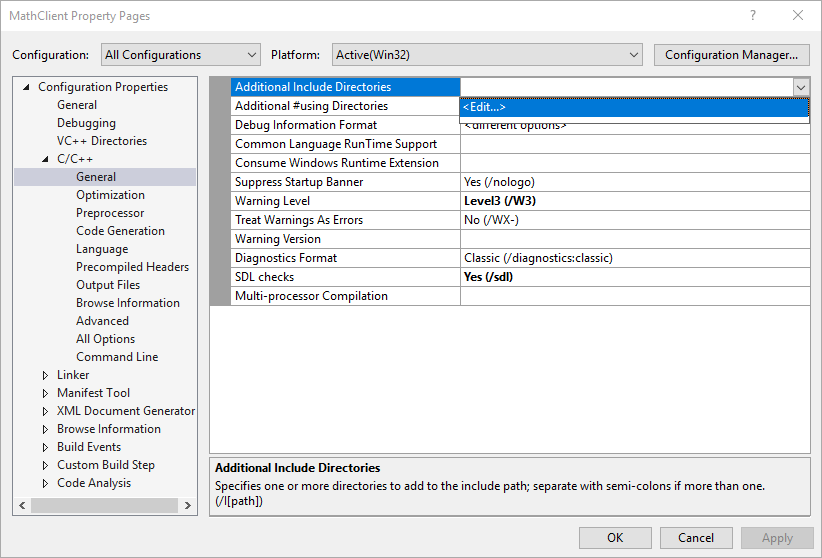
Необходимо вписать в хедер для dll

extern "C++" \_\_declspec(dllexport) для классов

\_\_declspec(dllexport) для функций и методов

**Добавление заголовка библиотеки DLL в путь включения**

1. Щелкните правой кнопкой мыши узел **MathClient** в **обозревателе решений**, чтобы открыть диалоговое окно **Страницы свойств**.
2. В раскрывающемся списке **Конфигурация** выберите пункт **Все конфигурации**, если он еще не выбран.
3. В области слева выберите пункт **Свойства конфигурации** >**C/C++** >**Общие**.
4. На панели свойств щелкните раскрывающийся элемент управления рядом с полем ввода параметра **Дополнительные каталоги включаемых файлов**, а затем щелкните **Правка**.



1. Дважды щелкните в верхней панели диалогового окна **Дополнительные каталоги включаемых файлов**, чтобы включить элемент управления "Поле ввода". Или щелкните значок папки, чтобы создать новую запись.
2. В элементе управления "Поле ввода" укажите путь к расположению файла заголовка **MathLibrary.h**. Чтобы перейти к нужной папке, можно выбрать элемент управления с многоточием ( **...** ).

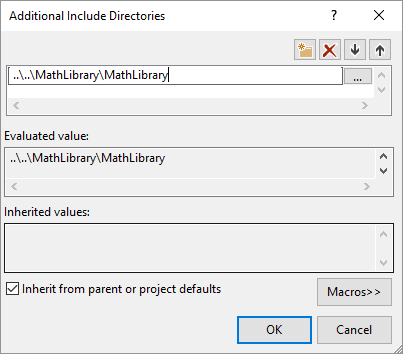
Можно также ввести относительный путь от исходных файлов клиента к папке, содержащей файлы заголовков библиотеки DLL. Если вы следовали инструкциям по размещению клиентского проекта в отдельном решении, отличном от библиотеки DLL, относительный путь должен выглядеть следующим образом:

..\..\MathLibrary\MathLibrary

Если библиотеки DLL и клиентские проекты находятся в одном решении, относительный путь может выглядеть следующим образом:

..\MathLibrary

Если библиотеки DLL и клиентские проекты находятся в других папках, измените относительный путь для соответствия. Или используйте элемент управления "Многоточие" для поиска папки.



1. После ввода пути к файлу заголовка в диалоговом окне **Дополнительные каталоги включаемых файлов** нажмите кнопку **ОК**. В диалоговом окне **Страницы свойств** нажмите кнопку **OK**, чтобы сохранить изменения.

Теперь можно добавить файл **MathLibrary.h** и использовать функции, которые он объявляет, в вашем клиентском приложении. Замените содержимое файла **MathClient.cpp**, используя следующий код:

// MathClient.cpp : Client app for MathLibrary DLL.

// #include "pch.h" Uncomment for Visual Studio 2017 and earlier

#include <iostream>

#include "MathLibrary.h"

int main()

{

// Initialize a Fibonacci relation sequence.

fibonacci\_init(1, 1);

// Write out the sequence values until overflow.

do {

std::cout << fibonacci\_index() << ": "

<< fibonacci\_current() << std::endl;

} while (fibonacci\_next());

// Report count of values written before overflow.

std::cout << fibonacci\_index() + 1 <<

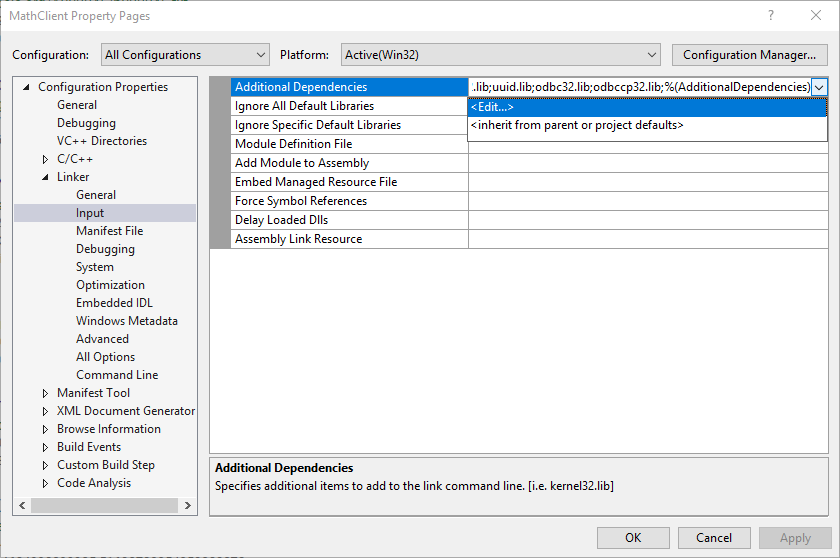
" Fibonacci sequence values fit in an " <<

"unsigned 64-bit integer." << std::endl;

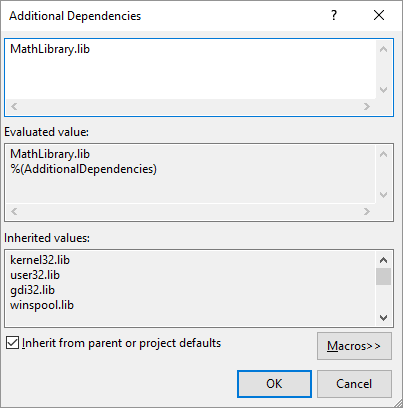
}

**Добавление библиотеки импорта DLL в проект**

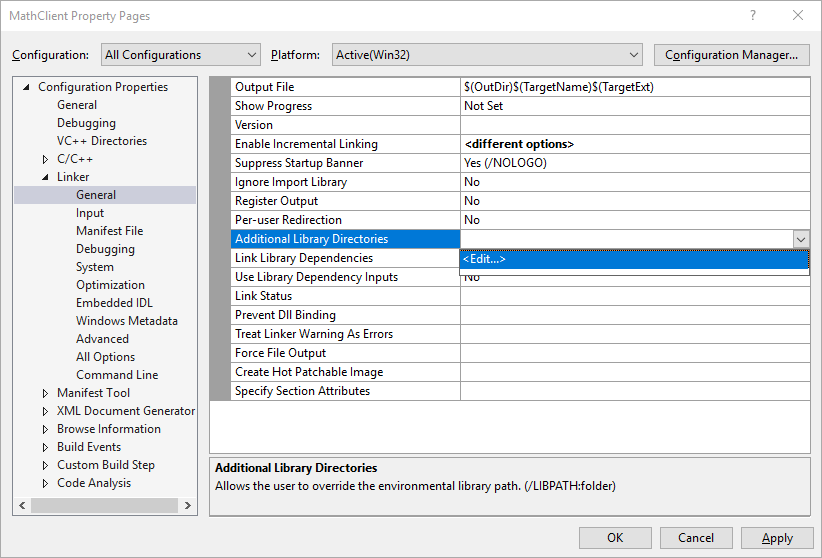
1. Щелкните правой кнопкой мыши узел **MathClient** в **обозревателе решений** и выберите **Свойства**, чтобы открыть диалоговое окно **Страницы свойств**.
2. В раскрывающемся списке **Конфигурация** выберите пункт **Все конфигурации**, если он еще не выбран. Это гарантирует, что любые изменения свойств применяются к сборкам отладки и выпуска.
3. В области слева выберите пункт **Свойства конфигурации** >**Компоновщик**>**Ввод**. На панели свойств щелкните раскрывающийся элемент управления рядом с полем ввода параметра **Дополнительные зависимости**, а затем щелкните **Правка**.



1. В диалоговом окне **Дополнительные зависимости** добавьте *MathLibrary.lib* в список в верхнем элементе управления "Поле ввода".



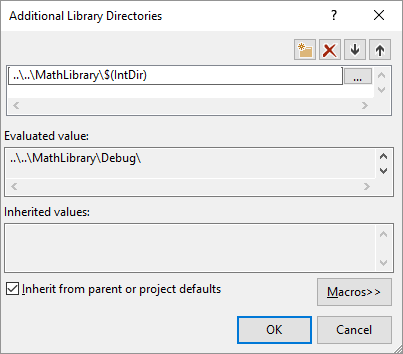
1. Нажмите кнопку **OK**, чтобы вернуться в диалоговое окно **Страницы свойств**.
2. В области слева выберите пункт **Свойства конфигурации** >**Компоновщик**>**Общие**. На панели свойств щелкните раскрывающийся элемент управления рядом с полем ввода параметра **Дополнительные каталоги библиотек**, а затем щелкните **Правка**.



1. Дважды щелкните в верхней панели диалогового окна **Дополнительные каталоги библиотек**, чтобы включить элемент управления "Поле ввода". В элементе управления "Поле ввода" укажите путь к расположению файла **MathLibrary.lib**. По умолчанию он находится в папке с именем *Debug* непосредственно в папке решения DLL. При создании сборки выпуска файл помещается в папку с именем *Release*. Можно использовать макрос $(IntDir), чтобы компоновщик мог найти библиотеку DLL независимо от типа создаваемой сборки. Если вы следовали инструкциям по размещению клиентского проекта в отдельном решении, отличном от проекта DLL, относительный путь должен выглядеть следующим образом:

..\..\MathLibrary\$(IntDir)

Если библиотеки DLL и клиентские проекты находятся в других расположениях, измените относительный путь для соответствия.



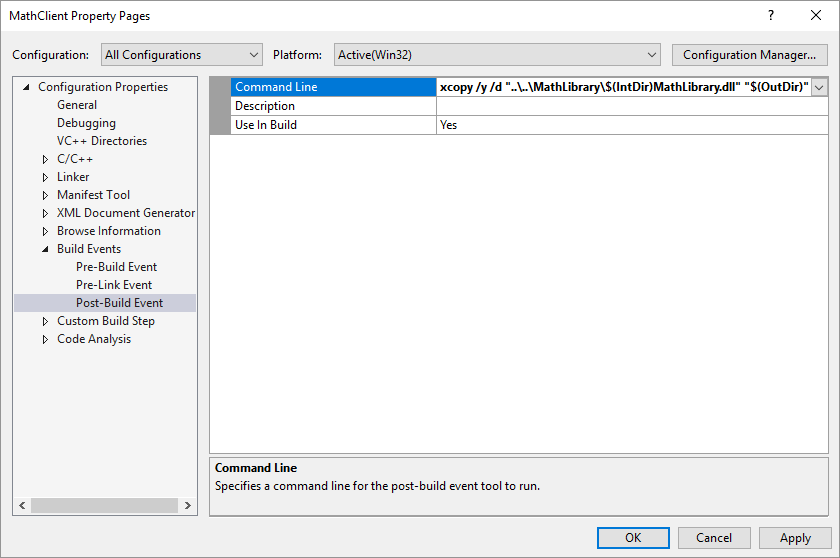
1. Как только вы ввели путь к файлу библиотеки, в диалоговом окне **Дополнительные каталоги библиотек** нажмите кнопку **ОК**, чтобы вернуться в диалоговое окно **Страницы свойств**. Нажмите **ОК**, чтобы сохранить изменения свойств.

**Копирование библиотеки DLL в событие после сборки**

1. Щелкните правой кнопкой мыши узел **MathClient** в **обозревателе решений** и выберите **Свойства**, чтобы открыть диалоговое окно **Страницы свойств**.
2. В раскрывающемся списке **Конфигурация** выберите пункт **Все конфигурации**, если он еще не выбран.
3. В области слева выберите **Свойства конфигурации** >**События сборки** >**Событие после сборки**.
4. В области свойств щелкните элемент управления "Поле ввода" в поле **Командная строка**. Если вы следовали инструкциям по размещению клиентского проекта в отдельном решении, отличном от проекта DLL, введите следующую команду:

xcopy /y /d "..\..\MathLibrary\$(IntDir)MathLibrary.dll" "$(OutDir)"

Если библиотеки DLL и клиентские проекты находятся в других каталогах, измените относительный путь к библиотеке DLL для соответствия.



1. Нажмите кнопку **OK**, чтобы сохранить изменения в свойствах проекта.
2. Паттерн программирования Строитель

Паттерн проектирования Builder

* Паттерн проектирования Builder разработан для обеспечения гибкого решения различных задач создания объектов в объектно-ориентированном программировании.
* Паттерн проектирования Builder позволяет отделить построение сложного объекта от его представления.
* Паттерн Builder создает сложные объекты, используя простые объекты и поэтапный подход.
* Паттерн предоставляет один из лучших способов создания сложных объектов.
* Это один из [паттернов проектирования банды четырех (GoF),](https://en.wikipedia.org/wiki/Design_Patterns) которые описывают, как решать периодически возникающие задачи проектирования в объектно-ориентированном программном обеспечении.
* Этот паттерн полезен для создания разных иммутабельных объектов с помощью одного и того же процесса построения объекта.

Паттерн Builder — это паттерн проектирования, который позволяет поэтапно создавать сложные объекты с помощью четко определенной последовательности действий. Строительство контролируется объектом-распорядителем (director), которому нужно знать только тип создаваемого объекта.

Итак, паттерн проектирования Builder можно разбить на следующие важные компоненты:

* **Product (продукт)** - Класс, который определяет сложный объект, который мы пытаемся шаг за шагом сконструировать, используя простые объекты.
* **Builder (строитель)** - абстрактный класс/интерфейс, который определяет все этапы, необходимые для производства сложного объекта-**продукта**. Как правило, здесь объявляются (абстрактно) все этапы (buildPart), а их реализация относится к классам конкретных строителей (ConcreteBuilder).
* **ConcreteBuilder (конкретный строитель)** - класс-строитель, который предоставляет фактический код для создания объекта-**продукта**. У нас может быть несколько разных **ConcreteBuilder**-классов, каждый из которых реализует различную разновидность или способ создания объекта-**продукта**.
* **Director (распорядитель)** - супервизионный класс, под конролем котрого строитель выполняет скоординированные этапы для создания объекта-продукта. Распорядитель обычно получает на вход строителя с этапами на выполнение в четком порядке для построения объекта-продукта.

#include <iostream>

#include <vector>

// Классы всех возможных родов войск

**class** Infantryman

{

**public**:

**void** info() {

      cout << "Infantryman" << endl;

    }

};

**class** Archer

{

**public**:

**void** info() {

      cout << "Archer" << endl;

    }

};

**class** Horseman

{

**public**:

**void** info() {

      cout << "Horseman" << endl;

    }

};

**class** Catapult

{

**public**:

**void** info() {

      cout << "Catapult" << endl;

    }

};

**class** Elephant

{

**public**:

**void** info() {

      cout << "Elephant" << endl;

    }

};

// Класс "Армия", содержащий все типы боевых единиц

**class** Army

{

**public**:

    vector<Infantryman> vi;

    vector<Archer>      va;

    vector<Horseman>    vh;

    vector<Catapult>    vc;

    vector<Elephant>    ve;

**void** info() {

**int** i;

**for**(i=0; i<vi.size(); ++i)  vi[i].info();

**for**(i=0; i<va.size(); ++i)  va[i].info();

**for**(i=0; i<vh.size(); ++i)  vh[i].info();

**for**(i=0; i<vc.size(); ++i)  vc[i].info();

**for**(i=0; i<ve.size(); ++i)  ve[i].info();

    }

};

// Базовый класс ArmyBuilder объявляет интерфейс для поэтапного

// построения армии и предусматривает его реализацию по умолчанию

**class** ArmyBuilder

{

**protected**:

    Army\* p;

**public**:

    ArmyBuilder(): p(0) {}

**virtual** ~ArmyBuilder() {}

**virtual** **void** createArmy() {}

**virtual** **void** buildInfantryman() {}

**virtual** **void** buildArcher() {}

**virtual** **void** buildHorseman() {}

**virtual** **void** buildCatapult() {}

**virtual** **void** buildElephant() {}

**virtual** Army\* getArmy() { **return** p; }

};

// Римская армия имеет все типы боевых единиц кроме боевых слонов

**class** RomanArmyBuilder: **public** ArmyBuilder

{

**public**:

**void** createArmy() { p = **new** Army; }

**void** buildInfantryman() { p->vi.push\_back( Infantryman()); }

**void** buildArcher() { p->va.push\_back( Archer()); }

**void** buildHorseman() { p->vh.push\_back( Horseman()); }

**void** buildCatapult() { p->vc.push\_back( Catapult()); }

};

// Армия Карфагена имеет все типы боевых единиц кроме катапульт

**class** CarthaginianArmyBuilder: **public** ArmyBuilder

{

**public**:

**void** createArmy() { p = **new** Army; }

**void** buildInfantryman() { p->vi.push\_back( Infantryman()); }

**void** buildArcher() { p->va.push\_back( Archer()); }

**void** buildHorseman() { p->vh.push\_back( Horseman()); }

**void** buildElephant() { p->ve.push\_back( Elephant()); }

};

// Класс-распорядитель, поэтапно создающий армию той или иной стороны.

// Именно здесь определен алгоритм построения армии.

**class** Director

{

**public**:

    Army\* createArmy( ArmyBuilder & builder )

    {

        builder.createArmy();

        builder.buildInfantryman();

        builder.buildArcher();

        builder.buildHorseman();

        builder.buildCatapult();

        builder.buildElephant();

**return**( builder.getArmy());

    }

};

**int** main()

{

    Director dir;

    RomanArmyBuilder ra\_builder;

    CarthaginianArmyBuilder ca\_builder;

    Army \* ra = dir.createArmy( ra\_builder);

    Army \* ca = dir.createArmy( ca\_builder);

    cout << "Roman army:" << endl;

    ra->info();

    cout << "\nCarthaginian army:" << endl;

    ca->info();

    // ...

**return** 0;

}

1. Паттерн программирования Фабрика

Фабричный метод (англ. Factory Method), или виртуальный конструктор (англ. Virtual Constructor) — порождающий шаблон проектирования, предоставляющий подклассам (дочерним классам, субклассам) интерфейс для создания экземпляров некоторого класса. В момент создания наследники могут определить, какой класс создавать. Иными словами, данный шаблон делегирует создание объектов наследникам родительского класса. Это позволяет использовать в коде программы не конкретные классы, а манипулировать абстрактными объектами на более высоком уровне.

#include *<iostream>*

#include *<string>*

**using** **namespace** **std**;

**struct** **Product**{

**virtual** string getName() = 0;

**virtual** ~Product(){}

};

**struct** **ConcreteProductA**: Product{

string getName(){**return** "ConcreteProductA";}

};

**struct** **ConcreteProductB**: Product{

string getName(){**return** "ConcreteProductB";}

};

**struct** **Creator**{

**virtual** Product\* factoryMethod() = 0;

};

**struct** **ConcreteCreatorA**: Creator{

Product\* factoryMethod(){**return** **new** ConcreteProductA();}

};

**struct** **ConcreteCreatorB**: Creator{

Product\* factoryMethod(){**return** **new** ConcreteProductB();}

};

int main()

{

ConcreteCreatorA CreatorA;

ConcreteCreatorB CreatorB;

*// An array of creators*

Creator\*creators[] = {&CreatorA, &CreatorB};

*// Iterate over creators and create products*

**for**(**auto**&& creator: creators){

Product\* product=creator->factoryMethod();

cout << product->getName() << endl;

**delete** product;

}

**return** 0;

}

26. Принципы SOLID.

SOLID   
**Принцип единственной обязанности**

Для каждого класса должно быть определено единственное назначение. Все ресурсы, необходимые для его осуществления, должны быть инкапсулированы в этот класс и подчинены только этой задаче.

**Принцип открытости/закрытости**

Программные сущности должны быть *открыты* для расширения, но *закрыты* для изменений.

**Принцип подстановки Барбары Лисков**

Методы, использующие некий тип, должны иметь возможность использовать его подтипы, не зная об этом.

**Принцип разделения интерфейсов**

Предпочтительнее разделять интерфейсы на более мелкие тематические, чтобы реализующие их классы не были вынуждены определять методы, которые непосредственно в них не используются.

**Принцип инверсии зависимостей**

Система должна конструироваться на основе абстракций «сверху вниз»: не абстракции должны формироваться на основе деталей, а детали должны формироваться на основе абстракций.

**Singleton**

Singleton – порождающий шаблон проектирования, гарантирующий, что в однопоточном приложении будет единственный экземпляр некоторого класса, и представляющий глобальную точку доступа к этому экземпляру.

Одиночка (Singleton, Синглтон) - порождающий паттерн, который гарантирует, что для определенного класса будет создан только один объект, а также предоставит к этому объекту точку доступа.

Когда надо использовать Синглтон? Когда необходимо, чтобы для класса существовал только один экземпляр

Синглтон позволяет создать объект только при его необходимости. Если объект не нужен, то он не будет создан. В этом отличие синглтона от глобальных переменных.

Безопасность:

Проблемы с безопасностью singleton:

1. Рефлексия – изменение во время исполнения класса
2. Десериализация – возможность создать объект
3. Поток и небезопасность -
4. Проблемы с тестированием

Плюсы

* контролируемый доступ к единственному экземпляру;
* уменьшение числа имён;
* допускает уточнение операций и представления;
* допускает переменное число экземпляров;
* большая гибкость, чем у операций класса.

Минусы

глобальные объекты могут быть вредны для объектного программирования, в некоторых случаях приводя к созданию немасштабируемого проекта.

усложняет написание модульных тестов и следование TDD

1. Безопасность паттерна Singleton

Проблемы с безопасностью singleton:

1. Рефлексия – изменение во время исполнения класса
2. Десериализация – возможность создать объект
3. Поток и небезопасность -
4. Проблемы с тестированием

Безопасность многопоточности

**Потокобезопасный**: В определенный момент времени, когда большое количество пользователей (потоков) обращаются к одному и тому же ресурсу, из-за механизма многопоточной работы доступный ресурс может иметь проблемы "загрязнения данных".

Безопасность дружественных функций

С дружественными функциями нет особых рисков безопасности. Дружественные функции являются частью реализации класса, точно так же, как и члены класса; является ли функция членом или другом, ничего не меняет в отношении рисков безопасности (или чего-либо еще).

1. Безопасность сериализации\десериализации

Уязвимости десериализации относятся к категории угроз, связанных с небезопасной обработкой полезных данных запроса. Использующие такие уязвимости злоумышленники могут провести на целевое приложение атаки, приводящие к отказу в обслуживании, раскрытию информации или удаленному выполнению кода в нем.