



сизжу ахуел



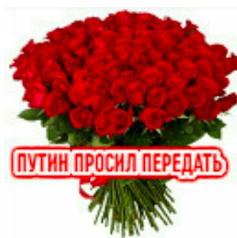
Авторы документа: **Макс Жерлыгин**, **Юля Обыденкова**, **Виталия Овечкин**, **Василиса Васильева**,
Дима Коростелёв, **Саша Марков**, **Андрюша Синявский**, **Ноташа Кошакена** (2085)

1. Ручное управление памятью. Выделение памяти на стеке. Опасности при возвращении адресов локальных переменных. Объект и его свойства. Классы объектов. Пространства имен. Жизненный цикл классов. Конструкторы и деструкторы.
2. Указатели и ссылки. LValue и RValue ссылки. Move-семантика. Константы в C++ (переменные и функции), мутанты.
3. Перегрузка операторов. Синтаксис. Возможные к перегрузке операции. Перегрузка ++ и -- . Перегрузка бинарных операторов.
4. Перегрузка операторов. Перегрузка оператора =. Запрет операторов копирования. Перегрузка (). Функтор. constexpr. Пользовательские литералы.
5. Понятие объектно-ориентированного языка. Объектно-ориентированная декомпозиция. Базовые понятия ООП: абстрагирование. Абстракция сущности, поведения, виртуальной- машины, произвольная. Наследование в C++. Полиморфизм.
6. Понятие объектно-ориентированного языка. Инкапсуляция. Жизненный цикл классов и объектов. Конструкторы и деструкторы.
7. Понятие объектно-ориентированного языка. Протечка абстракции. Примеры, способы устранения.
8. Исключения в C++. std::exception и std::exception_ptr. std::current_exception и std::rethrow_exception. Модификатор noexcept. Метод terminate
9. Шаблоны классов и функций. Сравнение наследования и шаблонов. Параметры шаблонов. Специализация шаблонов.
10. Шаблоны классов и функций. Метафункции. Метафункция возвращающая разный тип данных в зависимости от условия.

11. [Шаблоны классов и функций. SFINAE. std::enable_if.](#)
12. [Шаблоны классов и функций. Variadic template. Хвостовая рекурсия на стадии компиляции.](#)
13. [Шаблоны классов и функций. Реализация tuple.](#)
14. [Лямбда выражения. Функция как параметр, функтор, std::function. Списки захвата лямбда функций. Использование лямбда-выражений в стандартных алгоритмах \(std::transform, std::for_each ...\).](#)
15. [Идиома RAII. Умные указатели. Недостатки «обычных» указателей. Шаблон std::unique_ptr. Семантика перемещения. std::move](#)
16. [Идиома RAII. Умные указатели. Недостатки «обычных» указателей. std::shared_ptr. Наследование и std::shared_ptr. Проблемы при использовании std::shared_ptr. Шаблон std::weak_ptr.](#)
17. [Контейнеры в C++. std::array, std::vector. Итераторы.](#)
18. [Контейнеры в C++. std::stack, std::queue, std::deque. Итераторы.](#)
19. [Контейнеры в C++. std::forward_list, std::list. Итераторы.](#)
20. [Контейнеры в C++. std::unordered_set, std::unordered_map. Итераторы.](#)
21. [Контейнеры в C++. std::unordered_multiset, std::unordered_multimap. Итераторы.](#)
22. [Аллокаторы памяти. Перегрузка оператора new. Простой аллокатор памяти на массиве.](#)
23. [Проектирование структуры классов. Характеристики \(жесткость, хрупкость, повторное использование. Способы улучшения структуры классов. SOLID: Принцип единой ответственности. Пример.](#)
24. [Виды связанности. Метрика cohesion. Метрика coupling. SOLID: Принцип открытости/закрытости. Пример.](#)
25. [Шаблон проектирования template method \(обычный и CRTP\). Шаблон проектирования strategy.](#)
26. [SOLID: Принцип подстановки Барбары Лисков.](#)
27. [Принцип TDA. Принцип Command Query Separation. Закон Постеля.](#)

28. [SOLID: Принцип разделения интерфейсов.](#)
29. [SOLID: Dependency Inversion Principle.](#)
30. [Мультипроцессирование и мультипрограммирование. SMP, MPP и NUMA. Мультипрограммирование. Вытесняющая и не вытесняющая многозадачность. Планировщик задач. Жизненный цикл потока.](#)
31. [std::thread. Функции и функторы как параметры. Использование семантики перемещения в std::thread. Функции из пространства имен std::this_thread.](#)
32. [Потокобезопасность. Реентерабельность. Race condition. Взаимное исключение. std::mutex и std::recursive_mutex.](#)
33. [std::lock_guard и std::unique_lock. Реализация потокобезопасного стека. dead_lock. Просачивание данных за пределы lock_guard](#)
34. [std::future и std::async std::promise. Условные переменные. Закон Амдала.](#)
35. [Неблокирующие алгоритмы. Атомарные типы. CAS операции. Потокобезопасный стек на CAS-операциях.](#)

ООП ебать его в рот. Ответы на вопросы ебаный насрал.



1. Ручное управление памятью.

Выделение памяти на стеке.

Стек — это область оперативной памяти, которая создается для каждого потока. Он работает в порядке LIFO (Last In, First Out), то есть последний добавленный в стек кусок памяти будет первым в очереди на вывод из стека. Каждый раз, когда функция объявляет новую переменную, она добавляется в стек, а когда эта переменная пропадает из области видимости (например, когда функция заканчивается), она автоматически удаляется из стека. Когда стековая переменная освобождается, эта область памяти становится доступной для других стековых переменных.

Из-за такой природы стека управление памятью оказывается весьма логичным и простым для выполнения на ЦП; это приводит к высокой скорости, в особенности потому, что время цикла обновления байта стека очень мало, т.е. этот байт скорее всего привязан к кэшу процессора. Тем не менее, у такой строгой формы управления есть и недостатки. Размер стека — это фиксированная величина, и превышение лимита выделенной на стеке памяти приведет к переполнению стека. Размер задается при создании потока, и у каждой переменной есть максимальный размер, зависящий от типа данных. Это позволяет ограничивать размер некоторых переменных (например, целочисленных), и вынуждает заранее объявлять размер более сложных типов данных (например, массивов), поскольку стек не позволит им изменить его. Кроме того, переменные, расположенные на стеке, всегда являются локальными.

Опасности при возвращении адресов локальных переменных.

Возврат по адресу — это возврат адреса переменной обратно в caller. Подобно передаче по адресу, возврат по адресу может возвращать только адрес переменной. Если попытаться вернуть адрес локальной переменной, то можно получить неожиданные результаты. Например:

```
int* doubleValue(int a)
{
    int value = a * 3;
    return &value; // value возвращается по адресу здесь
} // value уничтожается здесь
```

Value уничтожается сразу после того, как его адрес возвращается в caller. Конечным результатом будет то, что caller получит адрес освобождённой памяти (висячий указатель), что, несомненно, вызовет проблемы.

Возврат по адресу часто используется для возврата динамически выделенной памяти обратно в caller:

```
int* allocateArray(int size)
{
    return new int[size];
}
int main()
{
    int *array = allocateArray(20);
    // Делаем что-нибудь с array
    delete[] array;
    return 0;
}
```

Здесь не возникнет никаких проблем, так как динамически выделенная память не выходит из области видимости в конце блока, в котором объявлена и всё ещё будет существовать, когда адрес будет возвращаться в caller.

Когда использовать возврат по адресу:

- при возврате динамически выделенной памяти;
- при возврате аргументов функции, которые были переданы по адресу.

Когда не использовать возврат по адресу:

- при возврате переменных, которые были объявлены внутри функции (используйте возврат по значению);
- при возврате большой структуры или класса, который был передан по ссылке (используйте возврат по ссылке).

Объект и его свойства.

Объявление класса определяет новый тип, связывающий код и данные между собой. Таким образом, класс является логической абстракцией, а объект — ее физическим воплощением. Иными словами, объект — это экземпляр класса.

Свойства объекта:

1. Состояние

в любой момент времени объект находится

в каком-либо состоянии, которое можно измерить / сравнить / скопировать

2. Поведение

объект может реагировать на внешние события либо меняя свое состояние, либо создавая новые события

3. Идентификация

объект всегда можно отличить от другого объекта

Классы объектов.

Классом называется группа объектов с общей структурой и поведением.

```
class имя-класса {  
    закрытые данные и функции  
    спецификатор_доступа:  
        данные и функции  
    спецификатор_доступа:  
        данные и функции  
    // ...  
    спецификатор_доступа:  
        данные и функции  
} список_объектов;
```

Список объектов указывать не обязательно. Он просто позволяет объявлять объекты класса. В качестве спецификатора доступа используется одно из трех ключевых слов языка C++: public, private, protected.

По умолчанию функции и данные, объявленные внутри класса, считаются закрытыми и доступны лишь функциям — членам этого класса. Спецификатор public открывает доступ к функциям и данным класса из других частей программы. Спецификатор доступа protected необходим только при наследовании классов. Зона влияния

спецификатора доступа простирается до следующего спецификатора или до конца объявления файла.

Пространства имен.

При включении в программу заголовка нового стиля его содержимое погружается в пространство имен `std`. Пространство имен — это просто область видимости. Оно предназначено для локализации имен идентификаторов и предотвращения конфликтов между ними. Элементы, объявленные в одном пространстве имен, отделены от элементов, принадлежащих другому пространству. Изначально имена библиотечных функций просто размещались в глобальном пространстве имен (как в языке C). Однако после появления заголовков нового образца их содержимое стали размещать в пространстве имен `std`.

Жизненный цикл классов.

Класс начинает свой жизненный цикл при его объявлении и вызове конструктора класса. Жизненный цикл класса заканчивается при вызове деструктора, который вызывается при выходе из зоны видимости.

Конструкторы и деструкторы.

ЭТО ИЗ ЛЕКЦИЙ ДЗЮБЫ: Если у класса есть конструктор, он вызывается всякий раз при создании объекта этого класса. Если у класса есть деструктор, он вызывается всякий раз, когда уничтожается объект этого класса.

Объект может создаваться как:

- автоматический, который создается каждый раз, когда его описание встречается при выполнении программы, и уничтожается по выходе из блока, в котором он описан;
- статический, который создается один раз при запуске программы и уничтожается при ее завершении;
- объект в свободной памяти, который создается операцией `new` и уничтожается операцией `delete`;
- объект-член, который создается в процессе создания другого класса или при создании массива, элементом которого он является.

ПРО ДЕКТРУКТОР ИЗ КНИЖКИ:

Антиподом конструктора является деструктор. Во многих ситуациях объект должен выполнить некоторое действие или действия, которые уничтожат его. Локальные объекты создаются при входе в соответствующий блок, а при выходе из него они уничтожаются. При разрушении объекта автоматически вызывается его деструктор. Для этого существует несколько причин. Например, объект должен освободить занимаемую им память или закрыть файл, открытый им ранее. В языке C++ эти действия выполняет деструктор. Имя деструктора совпадает с именем конструктора, но перед деструктором ставится знак тильда `~`. Пример:

```

1  class date
2  {
3      int day, year;
4      char *month;
5  public:
6      date(int d, char* m, int y)
7      {
8          day = d;
9          month = new char[strlen(m)+1];
10         strcpy_s(month, strlen(m)+1,m);
11         year = y;
12     }
13     ~date() { delete[] month; } // деструктор
14 };

```

2. Указатели и ссылки.

Ссылка `reference` — механизм языка программирования (C++), позволяющий привязать имя к значению. В частности, ссылка позволяет дать дополнительное имя переменной и передавать в функции сами переменные, а не значения переменных.

Синтаксически ссылка оформляется добавлением знака `&` (амперсанд) после имени типа. Ссылка на ссылку невозможна.

Ссылка требует инициализации. В момент инициализации происходит привязка ссылки к тому, что указано справа от `=`. После инициализации ссылку нельзя “отвязать” или “перепривязать”.

Любые действия со ссылкой трактуются компилятором как действия, которые будут выполняться над объектом, к которому эта ссылка привязана.

В C и C++ указатель определяется с помощью символа `*` после типа данных, на которые этот указатель будет указывать.

Указатель — старший родственник ссылки. Указатели активно использовались ещё в машинных языках и отсюда были перенесены в C. Ссылки же доступны только в C++.

Указатели — простые переменные. Указатели не “делают вид”, что они — те значения в памяти, к которым они привязаны. Чтобы получить указатель на переменную, нужно явно взять её адрес с помощью оператора `&`. Чтобы обратиться к переменной, на которую указывает указатель, требуется явно разыменовать его с помощью оператора `*`.

Rvalue ссылка – это составной тип, очень похожий на традиционную ссылку в C++. Чтобы различать эти два типа, мы будем называть традиционную C++ ссылку lvalue ссылка. Когда будет встречаться термин ссылка, то это относится к обоим видам ссылок, и к lvalue ссылкам, и к rvalue ссылкам.

По семантике lvalue ссылка формируется путём помещая & после некоторого типа. Rvalue ссылка ведет себя точно так же, как и lvalue ссылка, за исключением того, что она может быть связана с временным объектом, тогда как lvalue связать с временным (не константным) объектом нельзя.

Семантика перемещений (move semantics)

Устранение побочных копий

Копирование может быть дорогим удовольствием. К примеру, для двух векторов, когда мы пишем `v2 = v1`, то обычно это вызывает вызов функции, выделение памяти и цикл. Это, конечно, приемлемо, когда нам действительно нужны две копии вектора, но во многих случаях это не так: мы часто копируем вектор из одного места в другое, а потом удаляем старую копию.

вызов `move()` возвращает значение объекта, переданного в качестве параметра, но не гарантирует сохранность этого объекта. К примеру, если в качестве параметра в `move()` передать `vector`, то можно обоснованно ожидать, что после работы функции от параметра останется вектор нулевой длины, так как все элементы будут перемещены, а не скопированы. Другими словами, перемещение – это считывание со стиранием (destructive read). Функция `move` в действительности выполняет весьма скромную работу. Её задача состоит в том, чтобы принять либо lvalue, либо rvalue параметр, и вернуть его как rvalue без вызова конструктора копирования

кроме переменных в языке программирования C++ можно определять константы. Их значение устанавливается один раз и впоследствии мы его не можем изменить.

Константа определяется практически также, как и переменная за тем исключением, что в начале определения константы идет ключевое слово `const`.

Обычно в качестве констант определяются такие значения, которые должны оставаться постоянными в течение работы всей программы и не могут быть изменены.

Например, если программы выполняет математические операции с использованием числа π , то было бы оптимально определить данное значение как константу, так как оно все равно в принципе неизменно

Объекты методы и ссылки могут быть константными

3. Перегрузка операторов.

Перегрузка операторов - это более удобный способ вызова функций. Функция оператора может быть определена как член класса, либо вне класса.

Синтаксис перегрузки операторов выглядит следующим образом:

тип operator @ (список_параметров-операндов)

{

// тело функции

},

где @ - это идентификатор оператора (например, +, -, <<, >>).

Есть 4 типа перегрузок операторов:

1. Перегрузка обычных операторов + - * / % ^ & | ~ ! = < > += -= *= /= %= ^= &= |= << >> >>= <<= == != <= >= && || ++ --, ->* -> () <=> []
2. Перегрузка операторов преобразования типа
3. Перегрузка операторов аллокации и деаллокации new и delete
4. Перегрузка литералов operator" "

Перегрузка операторов инкремента (++) и декремента (--) довольно-таки проста. Особенность перегрузки этих операторов состоит в том, что нужно перегружать как префиксную (например: ++x, --y) так и постфиксную форму (например: x++, y--). этих операторов.

Поскольку операторы инкремента и декремента являются унарными и изменяют свои операнды, то перегрузку следует выполнять через методы класса.

- Если перегружается префиксная форма оператора ++, то в классе нужно реализовать операторную функцию `operator++()` без параметров;
- Если перегружается постфиксная форма оператора ++, то в классе нужно реализовать операторную функцию `operator++(int d)` с одним параметром. Параметр `d` необходим для указания того, что перегружается именно постфиксная реализация оператора.

пример:

префикс:

```
T operator ++ () {  
x++;  
y++;  
return *this; }
```

постфикс:

```
T operator ++(int) {  
T old = *this  
x++;  
y++;  
return old; }
```

Перегрузка бинарных операторов

Любая бинарная операция @ может быть определена двумя способами: либо как компонентная функция с одним параметром, либо как глобальная (возможно дружественная) функция с двумя параметрами. В первом случае `x @ y` означает вызов `x.operator @(y)`, во втором – вызов `operator @(x, y)`.

Операции, перегружаемые внутри класса, могут перегружаться только нестатическими компонентными функциями с параметрами. Вызываемый объект класса автоматически воспринимается в качестве первого операнда.

Операции, перегружаемые вне области класса, должны иметь два операнда, один из которых должен иметь тип класса.

4. *Определение в вопросе №3*

Перегрузка =

Оператор присваивания обязательно определяется в виде функции класса, потому что он неразрывно связан с объектом, находящимся слева от “=”.

Пример:

```
T& operator = (const T& arg) {
    if(this->x == arg.x && this->y == arg.y) {
        return *this;
    }
    x = arg.x;
    y = arg.y;
    return *this;
}
```

Перегрузка ()

Оператор “[]” ограничен лишь одним параметром. Оператор “()” может принимать разное количество параметров.

Пример:

```
class Matrix {
private:
    double data[5][5];
public:
    double& operator()(int i, int j) {
        return data[i][j];
    }
}
```

Функтор.

Функтор — это сокращение от функциональный объект, представляющий собой конструкцию, позволяющую использовать объект класса как функцию. В C++ для определения функтора достаточно описать класс, в котором переопределен оператор ().

Пример:

```
class functor{
private:
    std::string name;
public:
    functor(const char* name_): name(name_) {}
    void operator()() {
        std::cout << "Hello, " << name << std::endl;
    }
};
```

```
int main() {
    functor f("Maxim");
    f();
}
```

Constexpr

Это спецификатор типа, для обозначения константных выражений, которые могут быть вычислены на этапе компиляции кода. Если значения параметров можно посчитать на этапе компиляции, то возвращаемое значение также должно посчитаться на этапе компиляции

Пример:

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

```
void func() {
    constexpr int a1 = sum (5, 12); // ОК: constexpr-переменная
    int a2 = sum (5, 12); // ОК: функция будет вызвана на этапе компиляции
}
```

Пользовательский литерал

Литерал - это некоторое выражение, создающее объект. В языке C++ существуют литералы для разных встроенных типов: int, double, float, char, unsigned long long, unsigned int в шестнадцатеричном формате, string. Начиная с версии C++ 11, можно определять собственные литералы на основе этих категорий для создания синтаксических ярлыков для общих идиом и повышения безопасности типов. Определяемые пользователем литералы не дают выигрыша в производительности. Они служат, главным образом, для удобства и для определения типов во время компиляции.

Синтаксис:

OutputType operator ""_suffix(InputType a);

Конструктор типа должен так же иметь спецификатор **constexpr**.

Могут иметь следующие параметры:

const char*

unsigned long long int

long double

char, wchar_t, char16_t, char32_t

const char*, std::size_t const wchar_t*, std::size_t const char16_t*, std::size_t const char32_t*, std::size_t

Пример:

```
class Rectangle{
public:
    unsigned long width,height;
    Rectangle(int w,int h) : width(w),height(h) {};
};

Rectangle operator "" _rect (unsigned long long w){
    return Rectangle(w,w);
}

int main(int argc, char** argv) {
    std::cout << "Width=" << (10_rect).width << std::endl;
    return 0;
}
```

5. Понятие объектно-ориентированного языка. ОО-декомпозиция. Базовые понятия ООП: абстрагирование. Абстракция сущности, поведения, виртуальной машины, произвольная. Наследование в C++, полиморфизм

Объектно-ориентированная декомпозиция предписывает разделение модели предметной области на элементы, выбирая в качестве критерия декомпозиции принадлежность этих элементов к различным абстракциям данной предметной области. Прежде чем разделять задачу на шаги, необходимо определить объекты предметной области.

Абстракция выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя.

- 1. Абстракция сущности**
Объект представляет собой полезную модель некой сущности в предметной области
- 2. Абстракция поведения**
Объект состоит из обобщенного множества операций
- 3. Абстракция виртуальной машины**
Объект группирует операции, которые либо вместе используются более высоким уровнем управления, либо сами используют некоторый набор операций более низкого уровня
- 4. Произвольная абстракция**
Объект включает в себя набор операций, не имеющих друг с другом ничего общего

Наследование

В наследственной иерархии общая часть структуры и поведения сосредоточена в наиболее общем суперклассе. По этой причине говорят о наследовании, как об иерархии *обобщение-специализация*.

Суперклассы при этом отражают наиболее общие, а подклассы - более специализированные абстракции, в которых члены суперкласса могут быть дополнены, модифицированы и даже скрыты.

Пример наследования в C++

```
class Animal
{
protected:
    std::string name;
public:
    Animal(const char *val) : name(val){};
    const char *get_name() { return name.c_str(); }
};
class Moose : public Animal
{
public:
    Moose() : Animal("Moose"){};
    void run()
    {
        std::cout << name << " running" << std::endl;
    }
}
```

Если говорить кратко, полиморфизм — это способность объекта использовать методы производного класса, который не существует на момент создания базового.

6. Понятие объектно-ориентированного языка. Инкапсуляция. Жизненный цикл классов и объектов. Конструкторы и деструкторы.

ООП - парадигма программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

Инкапсуляция - это процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение. Инкапсуляция служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации.

Примером инкапсуляции в C++ может служить контроль доступа. Член класса может быть private, protected, public.

private - к члену класса X имеют доступ методы класса X, а также друзья этого класса.
protected - к члену класса X имеют доступ методы класса X и друзья этого класса, а также методы и друзья производных от этого класса.
public - к члену класса X имеют доступ любая функция.

Конструктор - функция, которая запускается при создании объекта.
Деструктор - функция, которая запускается при уничтожении объекта.

Конструктор вызывается, начиная от родителя к наследнику.
Деструкторы наоборот.

Деструктор полиморфного базового класса должен объявляться виртуальным. Только так обеспечивается корректное разрушение объекта производного класса через указатель на соответствующий базовый класс.

7. Понятие объектно-ориентированного языка

В основе объектно-ориентированного программирования лежит понятие объекта - некой субстанции, которая объединяет в себе поля (данные) и методы(выполняемые объектом действия).

Протечка абстракции

Неудобной составляющей работы с коллекциями объектов родительского типа является необходимость приведения родительского типа к типу-наследнику(для выполнения необходимых операций над элементом коллекции). Т.е. мы жертвуем статическим контролем типов.

8. Исключения в C++. `std::exception` и `std::exception_ptr`. `std::current_exception` и `std::rethrow_exception`. Модификатор `noexcept`. Метод `terminate`

Обрабатывать ошибки можно несколькими способами – вернуть как результат функции или как один из параметров, передаваемых по ссылке.

Для реализации механизма обработки исключений в язык C++ введены следующие три ключевых слова: Try, catch, throw.

Пример:

```

#include <cstdlib>
#include <iostream>
#include <exception>
class PrintCheck{
public:
    PrintCheck(int money){
        if(money<0) throw std::exception(); //непонятное исключение
        std::cout << "Total:" << money << std::endl;
    }
};
// Пример очень плохого стиля программирования
int main(int argc, char** argv) {
    int money = 0;
    try{
        std::cin >> money;
        if(money<0) throw std::exception(); // непонятное исключение
        if(money==0) throw 0;
    }catch(std::exception ex){
        std::cout << "Да у тебя кредит!?" << std::endl;
        PrintCheck A(money); // А вот тут исключение уже не ловится
    }catch(int){
        std::cout << "Голытьба!" << std::endl;    }
    std::cout << "Thank you for your money:" << money << std::endl;
    return 0; }

```

Исключение – объект наследник класса `std::exception`, событие, прерывающее работу программы. C++ позволяет создавать исключения любого типа, хотя обычно рекомендуется использовать типы производные от `std::exception`. Исключение в c++ может быть перехвачено обработчиком, который способен перехватить любой тип исключения.

Если созданное исключение имеет тип класса, у которого имеется один или несколько базовых классов, то его могут перехватить обработчики, которые принимают базовые классы этого типа исключения.

Current_exception – данная функция возвращает `exception_ptr`. Если мы находимся внутри блока `catch`, то возвращает `exception_ptr`, который содержит обрабатываемое в данный момент текущим потоком исключение, если вызвать ее вне блока `catch`, то она вернет пустой объект `exception_ptr`.

Rethrow_exception – данная функция бросает исключение, которое содержится в `exception_ptr`. Если входной параметр не содержит исключений, то результат не определен

Модификатор noexcept - используется для того, чтобы задать возможность вызова функцией исключений, данным модификатором разработчик гарантирует, что из функции не будет выброшена исключение, однако, если метод с `noexcept` все таки сгенерирует исключение, то оно перехвачено не будет.

Если для текущего исключения не удастся найти подходящий обработчик, то вызывается переопределенная функция времени выполнения terminate. Действие по

умолчанию для `terminate` заключается в том, что она вызывает функцию `abort`. Если вам необходимо, чтобы перед выходом из приложения функция `terminate` в вашей программе вызывала какую то другую функцию, вызовите функцию `set_terminate` указав в качестве ее аргумента функцию, которую нужно вызвать.

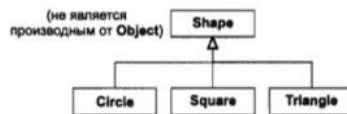
**9.Шаблоны классов и функций. Сравнение наследования и шаблонов.
Параметры шаблонов. Специализация шаблонов.**

Сравнение наследования и шаблонов.

Два вида многократного использования кода

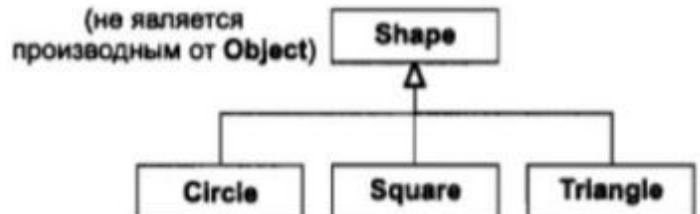
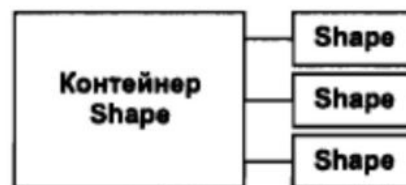
Наследование

- Создаем структуру для работы с «базовым классом»
- Создаем классы-наследники на каждый случай.



Шаблоны

- Описываем «стратегию работы» с «неопределенным» классом.
- Компилятор в момент создание класса по шаблону, сам создает нужный «код» для конкретного класса.



Параметры шаблонов.

- Шаблон – это параметрическая функция или класс.
- Параметром может являться как значение переменной (как в обычных функциях) так и тип данных.
- Параметры подставляются на этапе компиляции программы.
- Подставляя параметры в шаблон – мы конструируем новый тип данных (или функцию, если это шаблон функции)

Перед описанием класса ставим ключевое слово `template <class T>` или `template <typename T>`

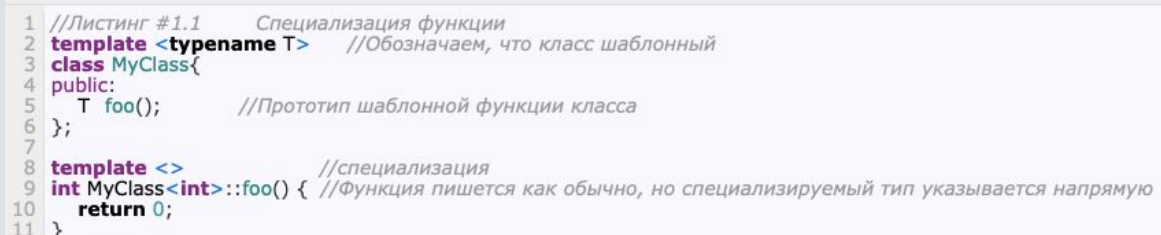
T – используем вместо имени класса, который будет заменяться при создании конкретного экземпляра класса.

```
template <class T> struct print {};  
  
// print– это шаблон  
// print<int> - это класс, сконструированный по шаблону
```

Специализация шаблонов.

Возможность специализации класса существует для того, чтобы мы могли скорректировать поведение чего-либо под нужды неподходящего наиболее общему решению типа.

Чтобы специализировать шаблон класса, следует объявить класс, предварив его конструкцией `template< >`, и указать типы, для которых специализируется шаблон класса. Типы используются в качестве аргументов шаблона и задаются непосредственно после имени класса.



```
1 //Листинг #1.1      Специализация функции  
2 template <typename T> //Обозначаем, что класс шаблонный  
3 class MyClass{  
4 public:  
5     T foo();          //Прототип шаблонной функции класса  
6 };  
7  
8 template <>  
9 int MyClass<int>::foo() { //специализация  
10     return 0;           //Функция пишется как обычно, но специализируемый тип указывается напрямую  
11 }
```

```

1 //Листинг #1.2 Специализация шаблона класса
2 #include <iostream>
3
4 using std::cin;
5 using std::cout;
6
7 /*ШАБЛОННЫЙ КЛАСС*/
8 template <typename T1>
9 class MyClass {
10 public:
11     int foo();
12 };
13
14
15 template <typename T>
16 int MyClass<T>::foo() { //Наиболее общий вариант решения
17     return 0;
18 }
19
20
21 template <>
22 class MyClass<char*> { //специализация
23 public: //шаблонного класса под тип char*
24     int foo();
25 };
26
27 int MyClass<char*>::foo() { //Реализация под специализацию
28     return 0;
29 }
30
31
32 int main() {
33     cin.get();
34 }

```

10. Шаблоны классов и функций. Метафункции. Метафункция возвращающая разный тип данных в зависимости от условия.

Метапрограммирование в с++ - по сути вычисление значений на этапе компиляции.

Основной инструмент- метафункции- шаблоны класса, а значения хранятся в статических переменных или переменных опред. типов (typedef) классов

Template<unsigned int N>

Struct fact{ static const unsigned int value = N*fact(N-1);}

Template<>

Struct fact<0>{static const unsigned}

Мои первые 2 закончены

11. Шаблоны классов и функций

SFINAE - механизм языка с++, связанный с шаблонами и перегрузкой функций.

Аббревиатура SFINAE расшифровывается как substitution failure is not an error.

Правило правило применяется, если **не получается рассчитать окончательные типы аргументов** (провести подстановку шаблонных параметров) **перегруженной шаблонной функции, компилятор не выбрасывает ошибку, а ищет другую подходящую перегрузку**. Ошибка будет в трёх случаях:

- Не нашлось ни одной подходящей перегрузки.

- Нашлось несколько таких перегрузок, и Си++ не может решить, какую взять.
- Перегрузка нашлась, она оказалась шаблонной, и при инстанцировании шаблона случилась ошибка.

Пример использования:

```
int difference(int val1, int val2)
{
    return val1 - val2;
}
template<typename T>
typename T::difference_type difference(const T& val1, const T& val2)
{
    return val1 - val2;
}
```

Шаблон `std::enable_if` действует на правиле SFINAE и позволяет создать ошибку замещения, чтобы включить или отключить определенные перегрузки на основе условия, оцененного во время компиляции.

`std::enable_if` может быть использована:

- в качестве дополнительного аргумента функции (не применимо к перегруженным операторам),
- в качестве возвращаемого типа (не применимо к конструкторам и деструкторам),
- как параметр шаблона класса или функции.

Возможная реализация `std::enable_if`

```
template<bool B, class T = void>
struct enable_if {};

template<class T>
struct enable_if<true, T> { typedef T type; };
```

12. Variadic template. Хвостовая рекурсия на стадии компиляции.

Variadic template.

Шаблоном с переменным числом параметров (*variadic template*) называется шаблон функции или класса, принимающий так называемый *parameter pack*. При объявлении шаблона это выглядит следующим образом

```
template<typename... Args> struct some_type;
```

Такая запись значит то, что шаблон может принять 0 или более типов в качестве своих аргументов. В теле же шаблона синтаксис использования немного другой.

`template<typename... Args> // Объявление`
`void foo(Args... args); // Использование`

Вызов `foo(1,2.3, "abcd")` инстанцируется в `foo<int, double, const char*>(1, 2.3, "abcd")`.

Хвостовая рекурсия на этапе компиляции.

Хвостовая рекурсия — частный случай **рекурсии**, при котором любой рекурсивный вызов является последней операцией перед возвратом из функции.

Для хвостовой рекурсии необходимо описать терминальное состояние, на котором рекурсия будет останавливаться.

Пример хвостовой рекурсии на стадии компиляции:

```
template <class T> void print(const T& t) {
    std::cout << t << std::endl;
}

template <class First, class... Rest>
void print(const First& first, const Rest&... rest) {
    std::cout << first << ", ";
    print(rest...); // хвостовая рекурсия на этапе компиляции
}

template <class T> T add(const T t) {
    return t;
}

template <class First, class... Rest>
First add(const First first, const Rest... rest) {
    return first + (First) add(rest...); // хвостовая рекурсия на этапе компиляции
}

int main(int argc, char** argv) {
    print(10, 20);
    print(100, 200, 300);
    print<int, double, int>(1, 2.0, 3);
    print("first", 2, "third", 3.14159);
    return 0;
}
```

13. Реализация tuple

Кортеж - коллекция элементов с фиксированным размером. Любая связанная пара, тройка, четверка и т.д элементов является кортежем.

В качестве элементов кортежа могут выступать переменные произвольного типа.

Создание кортежа

Путем инстанцирования шаблона `std::tuple<>` можно создавать кортежи :

```
std::tuple<int> t1;  
std::tuple<int, std::string> t2;
```

Можно также задавать значения у элементов кортежа путем их последовательного перечисления в конструкторе :

```
std::tuple<int> t1(1);  
std::tuple<int, std::string> t2(1, "hello");
```

basic_tuple.cpp

```
template <class... Ts> class tuple {  
};  
  
template <class T, class... Ts>  
class tuple<T, Ts...> : public tuple<Ts...> {  
public:  
    tuple(T t, Ts... ts) : tuple<Ts...>(ts...), value(t) {  
    }  
    tuple<Ts...> &next = static_cast<tuple<Ts...>&>(*this);  
    T value;  
};
```

Что внутри?

```
class tuple {  
}  
  
class tuple<const char*> : public tuple {  
    const char* value;  
}  
  
class tuple<uint64_t, const char*> : public tuple<const char*>{  
    uint64_t value;  
}  
  
class tuple<double, uint64_t, const char*> : public  
tuple<uint64_t, const char*>{  
    double value;  
}
```


Вспомогательный тип

```
// специальная структура для определения типа конкретного элемента в
tuple
template <size_t, class> struct elem_type_holder;

// без параметра – это тип базового класса
template <class T, class... Ts> struct elem_type_holder<0, tuple<T, T
s...>> {
    typedef T type;
};

// это тип k-го класса в цепочке наследования
template <size_t k, class T, class... Ts>
    struct elem_type_holder<k, tuple<T, Ts...>> {
        typedef typename elem_type_holder<k - 1, tuple<Ts...>>::type type;
    };
};
```

Получение элемента tuple.cpp

```
template <size_t index, class ...Ts>
typename std::enable_if<index == 0,
    typename elem_type_holder<0, tuple<Ts...>>::type&>::type
    get(tuple<Ts...>& t){
    return t.value;
}

template <size_t index, class T, class ...Ts>
typename std::enable_if<index != 0,
    typename elem_type_holder<index, tuple<T, Ts...>>::type&>::type
    get(tuple<T, Ts...>& t){
    tuple<Ts...> &base = t;
    return get<index-1>(base);
}
```

14. Лямбда выражения. Функция как параметр, функтор, std::function. Списки захвата лямбда функций. Использование лямбда выражений в стандартных алгоритмах (std::transform, std::for_each ...).

Функтор - класс, который содержит перегрузку оператора (), и работает как функция.

Лямбда-выражения - это краткая форма записи анонимных функторов.

```
[](int x){
    std::cout << x << " ";
}
```

Соответствует

```
class lambda {
public:
    void operator()(int x) const {
        std::cout << x << " ";
    }
};
```

Если в лямбда-функции только один return, то тип возвращаемого значения определяется автоматически, но если их несколько, то нужно указать явно.

```
[](int x) -> double {
    if (x < 5) {
        return x + 1.0;
    } else if (x % 2 == 0) {
        return x / 2.0;
    } else {
        return x * x;
    }
}
```

Поскольку лямбда-функции это анонимные функторы, то они могут хранить состояние. Чтобы использовать переменные, объявленные вне лямбды-функции, нужно использовать список захвата.

```
[a, b](int x) {
    return a < x && x < b;
}
```

Эквивалентно

```
class lambda {
    int a, b;
public:
    lambda(int a_, int b_) : a(a_), b(b_) {}

    bool operator()(int x) {
        return a < x && x < b;
    }
};
```

Правила захвата переменных из внешнего контекста:

[] - без захвата переменных из внешней области видимости

[=] - все переменные захватываются по значению
[&] - все переменные захватываются по ссылке
[this] - захват текущего класса
[x, y] - захват x, y по значению
[&x, &y] - захват x, y по ссылке
[x, &y] - захват x по значению, а y по ссылке
[=, &x, &y] - захват всех переменных по значению, кроме x и y, которые захватываются по ссылке
[&, x, y] - захват всех переменных по ссылке, кроме x и y, которые захватываются по значению

std::function является полиморфной оберткой функций для общего использования. Объекты класса std::function могут хранить, копировать и вызывать функции, лямбда-выражения, выражения связывания и другие функциональные объекты.

```
template<class> class function;
```

```
template<class R, class... ArgTypes>  
class function<R(ArgTypes...)>;
```

```
void print_num(int i)  
{  
    std::cout << i << "\n";  
}
```

```
int main()  
{  
    std::function<void(int)> f_display = print_num;  
    f_display(-9);  
}
```

```
template< class InputIt, class UnaryFunction >  
UnaryFunction for_each( InputIt first, InputIt last, UnaryFunction f);
```

По порядку применяет заданный функциональный объект f к результату разыменования каждого итератора в диапазоне [first, last). Если InputIt — изменяемый итератор, то f может изменять элементы диапазона через разыменованный итератор. Если f возвращает результат, то он игнорируется.

```
std::vector<int> nums{3, 4, 2, 9, 15, 267};
```

```
std::for_each(nums.begin(), nums.end(), [](int &n){ n++; });
```

```
template< class InputIt, class OutputIt, class UnaryOperation >
OutputIt transform( InputIt first1, InputIt last1, OutputIt d_first, UnaryOperation unary_op );
```

```
template< class InputIt1, class InputIt2, class OutputIt, class BinaryOperation >
OutputIt transform( InputIt1 first1, InputIt1 last1, InputIt2 first2, OutputIt d_first,
BinaryOperation binary_op );
```

Применяет заданную функцию к одному диапазону и сохраняет результат в другой диапазон, начинающийся с d_first.

В первом варианте унарная операция unary_op применяется к диапазону [first1, last1). Во втором варианте бинарная операция binary_op применяется к элементам из двух диапазонов: [first1, last1) и начинающемуся с first2.

```
std::string s("hello");
std::transform(s.begin(), s.end(), s.begin(), [](unsigned char ch){ return std::toupper(ch) });
```

15. Идиома RAII(Resource Acquisition Is Initialization)

Получение ресурса есть инициализация - программная идиома ООП, смысл которой заключается в том, что с помощью тех или иных программных механизмов получение некоторого ресурса неразрывно совмещается с инициализацией, а освобождение - с уничтожением объекта.

Типичным (хотя и не единственным) способом реализации является организация получения доступа к ресурсу в **конструкторе**, а освобождения - в **деструкторе** соответствующего класса.

Поскольку деструктор автоматической переменной вызывается при выходе из ее области видимости, то ресурс гарантированно освобождается при уничтожении переменной. Это справедливо и в ситуациях, в которых возникают **исключения**.

Умные указатели

Класс, предназначенный для управления динамически-выделенной памятью и обеспечения освобождения выделенной памяти при выходе объекта этого класса из области видимости.

Недостатки “обычных” указателей

1. нет контроля создания и удаления
2. может указывать в неизвестность, **nullptr**
3. может указывать в неизвестность, но чужую

Шаблон `std::unique_ptr`

`#include<memory.h>`

1. нераздельное владение объектом
2. нельзя копировать(только перемещение)
3. размер зависит от пользовательского deleter'a
4. без особой логики удаления издержки чаще всего отсутствуют
5. `std::make_unique` - только в качестве "синтаксического сахара"

Семантика перемещения

Единственные доступные операторы перемещения и копирования:

`unique_ptr& operator= (unique_ptr&& x) noexcept; //копируем из Rvalue`

`assign null pointer(2) unique_ptr& operator= (nullptr_t) noexcept;`

`std::move`

`template<class T>`

`typename std::remove_reference<T>::type&& move(T&& t);`

Возвращает объект Lvalue с помощью шаблона структуры `std::remove_reference`, которая помогает получить тип без ссылок:

`template<class T> struct remove_reference{typedef T type};`

`template<class T> struct remove_reference<&T>{typedef T type};`

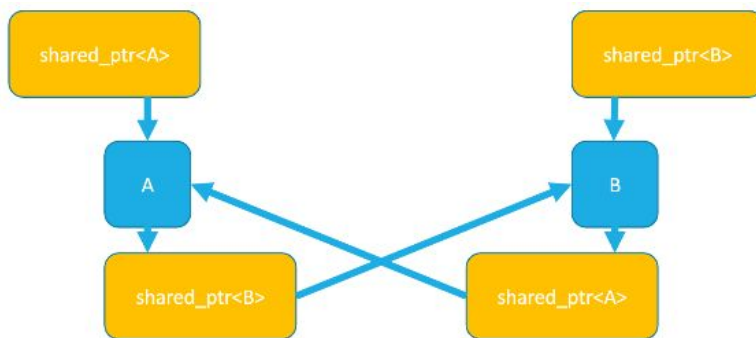
`template<class T> struct remove_reference<&&T>{typedef T type};`

16. Идиома RAII. Умные указатели. Недостатки «обычных» указателей.

`std::shared_ptr`. Наследование и `std::shared_ptr`. Проблемы при использовании `std::shared_ptr`. Шаблон `std::weak_ptr`.

Shared_ptr – предоставляет возможности по обеспечению автоматического удаления объекта за счет подсчета ссылок указателей на объект. Хранит ссылку на один объект, при создании счетчик ссылок на объект увеличивается, при удалении – уменьшается, при достижении счетчиком значения 0 – объект автоматически удаляется. Кроме этого указатель можно копировать с разделением владения, но дешевле перемещать, внутри всегда два указателя(на объект и на счетчик), при использовании `std::make_shared` память выделяется сразу под счетчик и под объект, потокобезопасный, можно создать из `unique_ptr`.

Проблема `shared_ptr` состоит в том, что если зациклить объекты друг на друга, то появится цикл и объект никогда не удалится, так как деструктор не запустится.



Weak_ptr – представляет разделяемое владение, обеспечивает доступ к объекту, только когда он существует, может быть удален кем-то другим, содержит деструктор вызываемый после его последнего использования.

17. Контейнеры в C++. std::array, std::vector. Итераторы.

Контейнер — это объект, который может содержать в себе другие объекты. Существует несколько разных типов контейнеров. Например, класс `vector` определяет динамический массив, `deque` создает двунаправленную очередь, а `list` представляет связный список. Эти контейнеры называются **последовательными контейнерами** (sequence containers), потому что в терминологии STL последовательность — это линейный список.

STL также определяет **ассоциативные контейнеры** (associative containers), которые обеспечивают эффективное извлечение значений на основе ключей. Таким образом, ассоциативные контейнеры хранят пары “ключ/значение”. Примером может служить `map`. Этот контейнер хранит пары “ключ/значение”, в которых каждый ключ является уникальным. Это облегчает извлечение значения по заданному ключу.

std::array

`std::array` - это контейнер, инкапсулирующий массив фиксированного размера.

array предоставляет некоторые возможности стандартных контейнеров, такие как знание собственного размера, поддержка присваивания, итераторы произвольного доступа и т.д.

Можно создавать array нулевой длины (`N == 0`).

```
#include <string>
#include <iterator>
#include <iostream>
#include <algorithm>
#include <array>

int main()
{
    // конструктор использует агрегатный инициализатор
    std::array<int, 3> a1{ {1,2,3} }; // требуются двойные фигурные скобки,
    std::array<int, 3> a2 = {1, 2, 3}; // за исключением операций присваивания
    std::array<std::string, 2> a3 = { {std::string("a"), "b"} };

    // поддерживаются обобщённые алгоритмы
    std::sort(a1.begin(), a1.end());
    std::reverse_copy(a2.begin(), a2.end(), std::ostream_iterator<int>(std::cout, " "));

    // поддерживается ranged for цикл
    for(auto& s: a3)
        std::cout << s << ' ';
}
```

std::vector

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

```
#include <iostream>
#include <vector>

int main ( ) {
    // Создание вектора, содержащего целые числа
    std::vector<int> v = {7, 5, 16, 8};

    // Добавление ещё двух целых чисел в вектор
    v.push_back(25);
    v.push_back(13);

    // Проход по вектору с выводом значений
    for ( int n : v ) {
        std::cout << n << '\n';
    }
}
```

std::vector vector.cpp

1. аналог динамическому массиву Си
2. эмулирует расширяемость
3. добавление в начало неэффективно
4. данные лежат в непрерывной области памяти (в куче)
5. итераторы произвольного доступа
6. инвалидация итераторов почти всегда

Итераторы.

Итераторы

Предоставляет способ последовательного доступа ко всем элементам контейнера, не раскрывая его внутреннего представления.

Зачем

- Составной объект, скажем список, должен предоставлять способ доступа к своим элементам, не раскрывая их внутреннюю структуру.
- Иногда требуется обходить список по-разному, в зависимости от решаемой задачи.
- Нужно, чтобы в один и тот же момент было определено несколько активных обходов списка.

Идея

Основная его идея в том, чтобы за доступ к элементам и способ обхода отвечал не сам список, а отдельный объект - итератор. В классе `Iterator` определен интерфейс для доступа к элементам списка. Объект этого класса отслеживает текущий элемент, то есть он располагает информацией, какие элементы уже посещались.

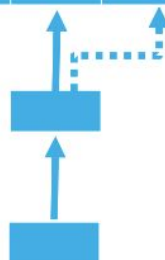
Итератор

Контейнер может иметь произвольную структуру и различные методы доступа:



Итератор указывает на элемент контейнера и знает как перейти к следующему элементу

Программе работает только с итератором и его интерфейсом (++)



Итераторы - итога

1. Итераторы – хранят ссылку на контейнер и даже на определенный элемент в контейнере
2. Итераторы – знают о структуре контейнера
3. Итераторы – предоставляют однотипный интерфейс по доступу к любому контейнеру
4. Итераторы – могут не только получать данные из контейнера, но могут записывать данные в контейнер (зависит от структуры контейнера)
5. Итераторы не могут менять размер контейнера

18. Контейнеры в C++. `std::stack`, `std::queue`, `std::deque`. Итераторы.

Класс `std::queue` -- это контейнер-адаптер, предоставляющий функционал структуры данных типа очередь, работающей по принципу FIFO (first-in, first-out) первым вошел -- первым вышел.

Шаблонный класс `std::queue` действует как класс-обертка. Он предоставляет лишь ограниченный набор функций для контейнера, который лежит внутри. Конкретно, `std::queue` кладет элементы в конец контейнера и извлекает элементы из начала.

`std::deque` (двусторонняя очередь) представляет собой последовательный индексированный контейнер, который позволяет быстро вставлять и удалять элементы с начала и с конца. Кроме того, вставка и удаление с обоих концов двусторонней очереди оставляет действительными указатели и ссылки на остальные элементы.

В отличие от `std::vector`, элементы `deque` не хранятся непрерывно: обычно реализован с помощью набора выделенных массивов фиксированного размера.

Хранилище `deque` обрабатывается автоматически, расширяясь и сужаясь по мере необходимости. Расширение `deque` дешевле, чем расширение `std::vector`, потому что оно не требует копирования существующих элементов в новый участок памяти.

`iterator = RandomAccessIterator`

`const_iterator` = Константный итератор с произвольным доступом

`reverse_iterator` = `std::reverse_iterator<iterator>`

`const_reverse_iterator` = `std::reverse_iterator<const_iterator>`

Класс `stack` является контейнером-адаптером, реализующим функционал стека - в частности, структуру данных FILO (First In Last Out - первый вошел, последний вышел).

19. Контейнеры в C++ . `std::forward_list`, `std::list`. Итераторы.

`std::forward_list` - контейнер, предоставляющий механизм вставки и удаления элементов из контейнера. Быстрый произвольный доступ не поддерживается. Реализован в виде однонаправленного списка и не имеет никаких накладных расходов по сравнению с аналогичной реализацией в C.

Также для перебора и получения элементов можно использовать итераторы. Причем класс `forward_list` добавляет ряд дополнительных функций для получения итераторов: **`before_begin()`** и **`cbefore_begin()`**. Обе функции возвращают итератор (вторая функция возвращает константный итератор `const_iterator`) на несуществующий элемент непосредственно перед началом списка. К значению по этому итератору обратиться нельзя. В отличие от `std::list`, этот тип контейнера не поддерживает двунаправленную итерацию.

Контейнер **`std::list`** задаёт стандартные двунаправленные списки. В эти списки можно быстро вставлять, а также удалять элементы. Однако операция обращения к элементу по номеру долгая.

Использование `list` требует подключения

```
#include <list>
```

Объявляется список так: `list<int> MyL;`

При работе со списком можно воспользоваться указателями:

- `myL.begin()` - указатель на начало списка,
- `myL.end()` - указатель на конец списка,
- `myL.rbegin()` - реверсивный указатель на конец списка,
- `myL.rend()` - реверсивный указатель на начало списка.

Полезна также функция проверки списка на пустоту: `myL.empty()`

Основные функции, для работы с отдельными элементами списка:

- `myL.push_back(e)` - добавить в конец элемент,
- `myL.pop_back()` - удалить последний элемент,
- `myL.front()` - первый элемент списка,
- `myL.back()` - последний элемент списка,
- `myL.insert(i, e)` - вставка элемента в позицию `i`,

- `myL.erase(first,last)` - удаляет последовательность элементов.

При работе со списками, чтобы удобно было по списку ходить, можно пользоваться итераторами, например, `list<int>::iterator i`;

Наконец, рассмотрим пример работы со списком:

```
#include <list> // подключаем list
#include <string> // подключаем строки string
#include <iostream> // подключаем cout
using namespace std; // чтобы не писать std::
int main()
{
    list<string> L; // инициализация list
    list<string>::iterator i;
    L.push_back("January"); // список January
    L.push_back("March"); // список January March
    L.push_back("April"); // список January March April
    i = L.begin();
    i++;
    L.insert(i, "February"); // список January February March April
    L.push_back("June"); // список January February March April June
    i = L.end();
    i--;
    L.insert(i, "May"); // список January February March April May June
    i = L.end();
    i--;
    i--;
    i--;
    L.erase(i); // список January February March May June
    L.pop_back(); // список January February March May
    for(i = L.begin(); i != L.end(); i++)
        cout << (*i) << " ";
    system("pause");
}
```

20. Контейнеры в C++. `std::unordered_set`, `std::unordered_map`. Итераторы.

`Unordered_set`

```
template<
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<Key>
> class unordered_set;
```

Неупорядоченное множество представляет собой ассоциативный контейнер, который содержит множество уникальных объектов типа Key. Поиск, вставка и удаление имеют в среднем константную временную сложность.

Разберём поподробнее все интересные методы и свойства.

Первый параметр — единственный обязательный — это собственно тип того, что мы будем хранить.

Второй параметр — это хэш-функция. Это должен быть объект (структура или функция), у которой оператор (), выполненный от объекта типа Key, возвращает величину типа size_t, равную хэшу объекта.

Третий параметр — это функция сравнения. Так как в одном bucket'e все значения лежат вперемешку, структура должна как-то уметь проверять — очередное значение это то, что надо, или нет? По умолчанию это std::equal_to, который просто вызывает оператор ==, который стоит перегрузить для ваших типов.

Четвёртый параметр — это стандартный для всех STL'ных структур аллокатор.

Пример:

```
using namespace std;
int main() {
    unordered_set<int> S;
    int n, m, t;
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> t;
        S.insert(t);
    }
    cin >> m;
    for (int i = 0; i < m; i++) {
        cin >> t;
        if (S.find(t) != S.end()) cout << "YES" << endl;
        else cout << "NO" << endl;
    }
    return 0;
}
```

Unordered_map

```
template<
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator< std::pair<const Key, T> >
> class unordered_map;
```

Unordered map является ассоциативным контейнером, который содержит пары ключ-значение с уникальными ключами. Поиск, вставка и удаление выполняются за константное время.

Первый параметр - тип ключа.

Второй параметр - Сопоставляемый тип.

Третий параметр- тип объекта хэш-функции.

Четвертый параметр - тип объекта функции сравнения напредмет равенства.

Пятый параметр - Класс распределителя.

21. Контейнеры в C++. unordered_multimap, unordered_multiset, итераторы

unordered_multimap: ассоциативный контейнер, поддерживающий одинаковые ключи, и сопоставляющий ключи со значениями другого типа. Этот класс поддерживает forward-итераторы. Элементы не хранятся в определённой последовательности, но распаханы по бакетам, в зависимости от хэша ключа, что значительно ускоряет поиск элемента.

unordered_multiset: абсолютно та же штука, но хранит не пары ключ-значение, а просто ключи.

Хз, что про них говорить. Наверно про то, что хэш-функция там есть. Итераторы у них - forward итераторы.

22. Аллокаторы памяти. Перегрузка оператора new. Простой аллокатор памяти на массиве.

Аллокатор управляет выделением памяти для контейнера. По умолчанию аллокатором является объект класса `allocator`, но можно определить свой аллокатор.

Операторы `new` и `delete` можно перегрузить. Причины:

Можно увеличить производительность за счет кеширования: при удалении объекта не освобождать память, а сохранять указатели на свободные блоки, используя их для вновь конструируемых объектов.

- Можно выделять память сразу под несколько объектов.
- Можно реализовать собственный сборщик мусора.
- Можно вести лог выделения/освобождения памяти.

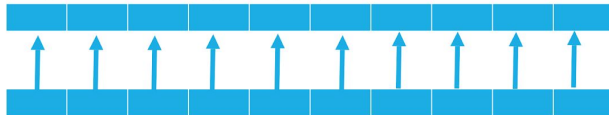
```
void *operator new(size_t size);  
void operator delete(void *p);
```

Оператор `new` принимает размер памяти, которую необходимо выделить, и возвращает указатель на выделенную память.

Оператор `delete` принимает указатель на память, которую нужно освободить.

Simple Allocator [1/4]

Used_blocks: Память для структур Item



Free_blocks: Указатели на свободные блоки

Вначале все блоки свободные. Каждый указатель в структуре free_blocks указывает на некоторый адрес в структуре used_blocks.



Simple Allocator [2/4]

Used_blocks: Память для структур Item



Free_blocks: Указатели на свободные блоки

Когда выделяется память – то возвращается значение последнего указателя в структуре free_blocks на адрес в used_blocks



Simple Allocator [3/4]

Used_blocks: Память для структур Item

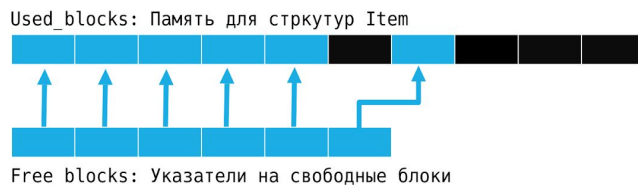


Free_blocks: Указатели на свободные блоки

После того как выделенно 5 объектов – мы имеем вот такую ситуацию



Simple Allocator [4/4]



Это происходит когда мы вызвали оператор delete. В конец массива free_blocks добавляется адрес Освободившегося блока

23. Проектирование структуры классов

Хорошо спроектированный класс:

1. проще и быстрее написать
2. в него быстрее вносить изменения
3. позволяет писать простой и понятный программный код

Плохой дизайн классов появляется не только из-за неправильных решений, принятых при написании программы. Он имеет особенность накапливаться при внесении рутинных изменений, таких как исправление ошибок.

Характеристики

1. жесткость - как просто поменять структуру класса
2. хрупкость - как сложно испортить структуру программы
3. повторное использование - как сложно переиспользовать готовую структуру классов в других программах

Способы улучшения структуры классов

1. Уменьшить связанность кода(слабо-связанный код позволяет вносить изменения в часть программы без влияния на остальной код)
2. Уменьшить сложность кода
3. Повысить читаемость кода(делаем структуру классов однотипной)
4. Повысить тестируемость кода(создаем методы классов таким образом, чтобы их работу можно было легко протестировать)

SOLID: принцип единой ответственности

- На каждый программный объект(функция/класс) должна быть возложена одна единственная ответственность

- Ответственность - это то, какую работу выполняет код с точки зрения пользователя. Т.е. это элементарная осмысленная подзадача в программе.
1. Требования к программе реализуются с помощью набора классов/функций
 2. У каждого класса/функции есть ответственность в решении задачи
 3. Чем больше у класса/функции ответственностей, тем больше вероятность, что его придется менять при изменении требований
 4. Таким образом, появляется вероятность, что при изменении одного класса мы повлияем на реализацию нескольких требований. Это может быть причиной ошибок

Пример(модем)

1. позволяет звонить
2. позволяет передавать данные
3. принимать данные
4. в конце сеанса связи нужно разъединяться

```
bool Deal(const char* value){..}//установка соединения
bool Connected(){...}//проверка соединения
void Hangup(){...}//разрыв соединения
void Send(char Data){...}//отправка данных
char Recieve(){...}//получение данных
```

Изменение задания: теперь модем может отправлять целые числа:

1. меняем сигнатуру двух методов
void Send(int Data){...};
int Recieve(){...};
2. меняем атрибуты класса Modem
3. меняем реализацию конструктора
4. добавляем метод bool NotEmpty(){...};
5. меняем код проверки модема в функции main

24. Виды связанности. Метрика cohesion. Метрика coupling. SOLID: Принцип открытости/закрытости. Пример.

Cohesion – свойство объекта или класса определяющее насколько объект занят своим делом. Если cohesion низкое, то у класса слишком много ответственности, класс делает слишком много различных операций, отчего становится большим, а большой класс тяжело читать, расширять и т.д.

Связанность(coupling) – степень того, насколько сильно модули зависят друг от друга. Измеряется подсчетом количества исходящих связей между пакетами, классами, отдельными функциями.

Loose coupling - принцип, связанности должны быть слабыми, то есть связей между программными сущностями должно быть как можно меньше и они должны быть как

можно слабее. Сильно зависимый код приводит к тому, что изменения в одном классе или методе ведут к необходимости вносить изменения в другие классы или методы. Классы имеющие высокий cohesion обычно слабо-связанные.

(Необязательно) Виды связанности: функционально связанный класс/функция (содержит атрибуты/переменные предназначенные для решения одной единственной задачи), последовательно связанный класс(объекты охватывают подзадачи, для которых выходные данные одной из подзадач являются входными для другой), информационно связанный класс (содержит объекты, использующие одни и те же выходные данные или выходные данные), процедурно связанный класс (класс, объекты которого включены в различные подзадачи, в которых управление переходит от одной подзадачи к следующей), класс с временной связанностью (класс, в котором объекты модуля привязаны к конкретному промежутку времени), класс с логической связанностью (класс, объекты которого содействуют решению одной общей подзадачи, для которой эти объекты отобраны во внешнем по отношению к классу мире), класс со связанностью по совпадению (содержит объекты, которые слабо связаны между собой).

Хорошие связанности – функциональная, последовательная и информационная.

Принцип открытости/закрытости – программные сущности должны быть открыты для расширения и закрыты для модификаций. В случае изменения требований мы не должны добавлять новые классы, расширяющие функциональность программ. Цели принципа – помогает делать базовые алгоритмы неизменными, новые классы-расширения вряд ли добавят ошибок в существующие алгоритмы, уменьшается количество проверок, которое нужно сделать для новых тестов.

Пример с модемом

Теперь нам нужно уметь подключаться не только по номеру телефона, но и по IPv4 адресу. А это уже не строка, а 4 байта.

Нам придется менять IConnection, либо изменяя логику Deal (например, записать IP адрес в строку "10.1.12.45") или добавлять еще один метод в интерфейс и менять структуру хранения данных в IConnection.

У нас плохой дизайн!

```
class IConnection {  
    ...  
    virtual bool Deal(const char* value) {  
        number = value;  
        connected = true;  
        return connected;  
    };  
    ...  
}
```


ОСР: Выносим адрес в контекст!

оср_1.cpp

1. Создаем отдельный класс IAddress;	class IConnection {
2. Делаем в нем виртуальную функцию для получения адреса в виде удобном для драйвера модема (воображаемого);	... virtual bool Deal(std::shared_ptr<IAddress> value) {
3. Делаем у него два наследника AddressPhone и AddressIP;	number = value; connected = true;
4. В наследниках описываем структуру хранения адреса;	return connected; };
5. Теперь изменение типа адреса сводится просто к добавлению нового класса-наследника. Алгоритмы не меняются!	... }

25.Шаблон проектирования template method (обычный и CRTP). Шаблон проектирования strategy.

Шаблонный метод — это поведенческий паттерн проектирования, который определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.

Паттерн TemplateMethod

оср_2_template.cpp

1. Скелет алгоритма определяется в базовом классе. Алгоритм использует еще не определенные методы (которые будут описаны в наследниках)
2. Классы-наследники переопределяют методы, используемые в алгоритме.
3. Таким образом, мы можем менять элементы алгоритма только созданием новых классов-потомков.

```
class Mailer{
protected:
    virtual const char* Greeting() = 0;
    virtual const char* Sign() = 0;
    ...
    const char* ComposeMail(){
        static std::string mail;
        mail = Greeting();
        mail += msg;
        mail += Sign();
        return mail.c_str();
    };
};
```

Curiously Recurring Template Pattern (CRTP) идиома языка C++, состоящая в том, что некоторый класс X наследуется от шаблона класса, использующего X как шаблонный параметр.

Основная форма [\[править | править код \]](#)

```
// The Curiously Recurring Template Pattern (CRTP)
template<class T>
class Base
{
    // methods within Base can use template to access members of Derived
};
class Derived : public Base<Derived>
{
    // ...
};
```

Шаблон проектирования strategy.

Стратегия — поведенческий шаблон проектирования, предназначенный для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости. Это позволяет выбирать алгоритм путём определения соответствующего класса.

Шаблон Strategy позволяет менять выбранный алгоритм независимо от объектов-клиентов, которые его используют.

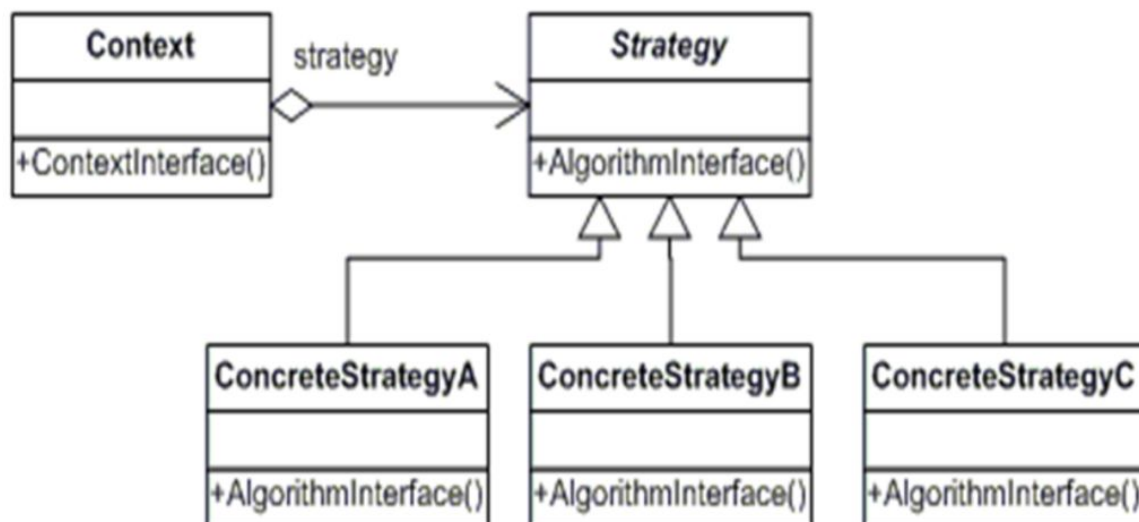
Strategy

Необходимо иметь много алгоритмов решения одной задачи (например, с какими-то особенностями).

Фактически, это механизм подключения plug-in.

Определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.

1. **Strategy** (Compositor) - стратегия:
 - объявляет общий для всех поддерживаемых алгоритмов интерфейс. Класс Context пользуется этим интерфейсом для вызова конкретного алгоритма, определенного в классе ConcreteStrategy;
2. **ConcreteStrategy** - конкретная стратегия:
 - реализует алгоритм, использующий интерфейс, объявленный в классе Strategy;
3. **Context** (Composition)
 - Контекст:
 - конфигурируется объектом класса ConcreteStrategy;
 - хранит ссылку на объект класса Strategy;
 - может определять интерфейс, который позволяет объекту Strategy получить доступ к данным контекста.



26 SOLID принцип Барбары Лисков

Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.

Таким образом, идея Лисков о «подтипе» даёт определение понятия **замещения** — если S является подтипом T, тогда объекты типа T в программе могут быть замещены объектами типа S без каких-либо изменений желательных свойств этой программы (например, **корректность**).

Этот принцип является *важнейшим* критерием для оценки качества принимаемых решений при построении иерархий наследования. Сформулировать его можно в виде простого правила: тип S будет подтипом T *тогда и только тогда*, когда каждому объекту oS типа S соответствует некий объект oT типа T таким образом, что для всех программ P, реализованных в терминах T, поведение P не будет меняться, если oT заменить на oS.

Более простыми словами можно сказать, что поведение наследующих классов не должно противоречить поведению, заданному базовым классом, то есть поведение наследующих классов должно быть ожидаемым для кода, использующего переменную базового типа.

При создании программных систем использование принципов SOLID способствует созданию такой системы, которую будет легко **поддерживать** и расширять в течение долгого времени^[3]. Принципы SOLID — это руководства, которые также могут применяться во время работы над существующим программным обеспечением для его улучшения - например для удаления «**дурно пахнущего кода**».

Стратегии **гибкой** и **адаптивной разработки**^{[en][3]} предполагают написание кода с соблюдением принципов SOLID.

L в слове SOLID отвечает как раз за принцип Лисков

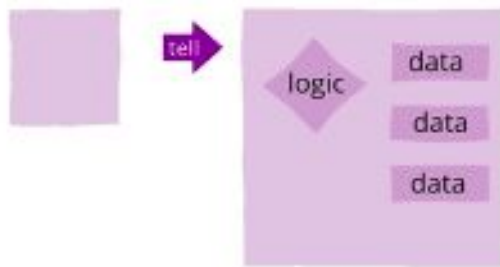
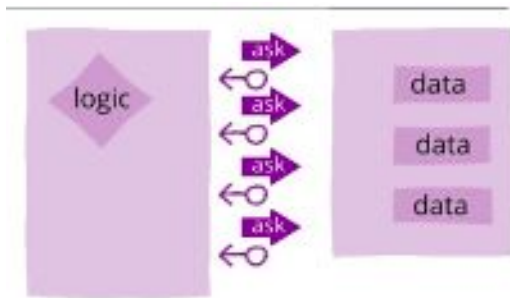
Принцип подстановки Барбары Лисков (*The Liskov Substitution Principle*)

«объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы.» См. также **контрактное программирование**.

Наследующий класс должен дополнять, а не изменять базовый.

27. Принцип TDA. Принцип Command Query Separation. Закон Постеля.

TDA - Tell Don't Ask - один из основополагающих принципов ООП: Необходимо делегировать объекту действия, вместо того, чтобы запрашивать его детали реализации. Это помогает достичь многократного использования класса (поскольку никто не знает его деталей реализации).



tda.cpp пример

```
1  #include <iostream>
2  #include <string>
3  #include <vector>
4
5  class MyCollection;
6
7  class MyPair
8  {
9  private:
10     int id;
11     std::string value;
12     friend MyCollection;
13 public:
14     MyPair(int i, const char* v) : id(i), value(v) {};
15     void Print() { std::cout << "id=" << id << " ,Value=" << value << std::endl; }
16 };
17
18 class MyCollection
19 {
20 private:
21     std::vector<MyPair> vector;
22 public:
23     void Add(MyPair &&other) {
24         vector.insert(vector.end(), other); } // good
25
26     void Add(int id, const char* value) {
27         vector.insert(vector.end(), MyPair(id, value)); } // bad
28
29     void Print() {
30         for (auto i : vector) i.Print(); } // good
31     void Print2() {
32         for (auto i : vector)
33             std::cout << "id=" << i.id << " ,Value=" << i.value << std::endl; }
34
35 };
36
37 int main()
38 {
39     MyCollection collection;
40     // Ask, don't tell;
41     collection.Add(1, "This is 1");
42
43     // Tell, don't ask
44     collection.Add(MyPair(2, "This is 2"));
45
46     collection.Print();
47
48     return 0;
49 }
```

Command Query Separation - принцип разделения методов, которые выполняют какие-либо действия (tell) и методов, которые осуществляют запросы данных (ask).

Операция либо имеет побочные эффекты (команда), либо возвращает значение (запрос), являясь чистой функцией.

Принцип говорит о том, что не должно быть функций, которые и возвращают результат запроса и меняют состояние объекта (имеют побочный эффект).

cqs.cpp пример

```
class MessageStore {  
  
public:  
  
void Save(size_t index, const char* msg) {  
  
... }  
  
const std::string Read(size_t index) {  
  
... } }
```

Принцип Постеля (принцип надежности) известен после интернет-пионера Джона Постеля, который написал в ранней спецификации протокола TCP, что:

TCP должны следовать за общим принципом надежности: будьте консервативны в том, что вы делаете, будьте либеральными в том, что вы принимаете от других.

Другими словами, кодексы, который посылает команды или данные к другим машинам (иди к другим программам на той же самой машине) должен соответствовать полностью техническим требованиям, но кодексы, который получает вход, должен принять вход non-conformant, пока значение четкое.

Среди программистов, чтобы произвести совместимые функции, принцип популяризирован в форме быть контр-вариантом во входном типе и контр-вариантным в типе продукции.

28. SOLID: Принцип разделения интерфейсов

Interface Segregation Principle

Принцип разделения интерфейса.

Много специализированных интерфейсов лучше, чем один универсальный.

Пользователи вашего класса не должны зависеть от методов, которые им не нужно использовать для решения своих задач.

Что плохого в «больших интерфейсах»?

1. Интерфейсы с большим числом методов трудно переиспользовать.
2. **Ненужные методы** приводят к увеличению связанности кода.
3. Дополнительные методы усложняют понимание кода.

Зато если у Вас много небольших интерфейсов можно быть уверенным, что каждый из них реализует одну ответственность!

Принцип разделения интерфейсов говорит о том, что слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические, чтобы программные сущности маленьких интерфейсов знали только о методах, которые необходимы им в работе. В итоге, при изменении метода интерфейса не должны меняться программные сущности, которые этот метод не используют.

29. Dependency Inversion Principle

Dependency Inversion Principle

зависимость от абстракции, а не от реализации

Принцип инверсии зависимостей.

Зависимости внутри системы строятся на основе абстракций. Модули верхнего уровня не зависят от модулей нижнего уровня. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Уменьшаем сильные связи между классами:

1. Два класса сильно связаны, если они зависят друг от друга;
2. Изменения в одном, ведут к изменениям в другом;
3. Сильно связанные классы не могут работать отдельно друг от друга;

Пример:


```

1. class ItemBulka : public IItem {
2. ...
3. };
4. class ItemCoffe : public IItem {
5. public:
6. void print() override {
7.     std::cout << "Item coffe ";
8.     bool hasBulka = false;
9.     for (auto i : items) {
10.         if (dynamic_cast<ItemBulka*> (i.get()))
11.             std::cout << " for your cookie";
12.     }
13.     std::cout << std::endl;
14. }
15. };

```

Печатаем меню в ресторане.

Классы ItemBulka и ItemCoffe – элементы меню.

Если кофе подается с булкой то в названии кофе это отображаем.

Что тут не так:

1. **А что если выпечка это не только ItemBulka?**
Зависимость от класса ItemBulka.
2. **А откуда взялась коллекция items?**
Зависимость от реализации класса Menu.

30. Мультипроцессирование и мультипрограммирование. SMP, MPP и NUMA. Мультипрограммирование. Вытесняющая и не вытесняющая многозадачность. Планировщик задач. Жизненный цикл потока.

Мультипроцессирование - использование нескольких процессоров для одновременного выполнения задач.

Виды мультипроцессирования:

- 1) Различные устройства. Например, одновременная работа центрального процессора и графического ускорителя видеокарты.
- 2) Одинаковые устройства.
 - 2.1) Асимметричное мультипроцессирование. Выделение управляющего и подчиненных устройств.
 - 2.2) Симметричное мультипроцессирование. Использование равноправных устройств.

SMP (shared memory processor).

В таком компьютере несколько процессоров подключены к общей оперативной памяти и имеют равноправный доступ.

По мере увеличения числа процессоров производительность оперативной памяти и шины становится критически важной.

Обычно используются 2-8 процессоров.

MPP (massively parallel processors).

Используются несколько однопроцессорных или SMP-систем, объединяемых с помощью некоторого коммуникационного оборудования в единую сеть. В такой системе оперативная память каждого узла изолирована от других узлов, и для обмена данными требуется пересылка данных по сети. Для MPP систем критической становится среда передачи данных; однако в случае мало связанных между собой мало связанных между собой процессоров возможно одновременное использование большого числа процессоров. Число процессоров в MPP измеряется сотнями и тысячами.

NUMA (non-uniform memory access).

Является компромиссом между SMP и MPP системами: оперативная память является общей и разделяемой между всеми процессорами, но при этом память неоднородна по времени доступа. Каждый процессорный узел имеет некоторый объем оперативной памяти, доступ к которой осуществляется максимально быстро; для доступа к памяти другого узла потребуется значительно больше времени.

Мультипрограммирование - одновременное выполнение нескольких задач на одном или нескольких процессорах.

В мультипрограммировании ключевым местом является способ составления расписания, по которому осуществляется переключение между задачами, а также механизм, осуществляющий эти переключения.

Виды планирования:

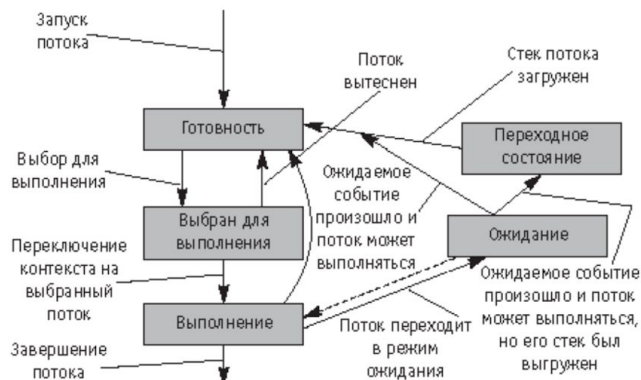
- 1) Статическое. Расписание составляется заранее, до запуска приложений, и ОС в дальнейшем просто выполняет составленную расписание.
- 2) Динамическое. Порядок задач и передачи управления задачам определяется непосредственно во время исполнения.

Виды многозадачности:

- 1) Невытесняющая. Решение о переключении принимает выполняемая в данный момент задача.
- 2) Вытесняющая. Решение принимается ОС, независимо от работы активной в данный момент задачи.

Планировщик задач в некоторый момент перепланирования принимает решение о том, какую именно задачу в следующий момент времени надо начать выполнять. Принципы, по которым назначается момент перепланирования, и критерии, по которым осуществляется выбор задачи, определяют способ реализации многозадачности и его сильные и слабые стороны.

Жизненный цикл потока в Windows



31. std::thread

заголовочный файл <thread>

Может работать с регулярными функциями, лямбдами и функторами, он позволяет передавать любое число параметров в функцию потока

конструктор

template<class Fn, class...Args> explicit thread(Fn&& fn, Args&&...args); // в качестве аргументов функтор и его параметры

Потоки позволяют нескольким фрагментам кода работать асинхронно и одновременно

Функции и функторы как параметры

```
class MyClass{
public:
    void operator()(const char* param){
        std::cout << param;
    }
};

int main(){
    std::thread my_class(MyClass(), "Hi");
    my_class.join();
}
```

Функции из пространства имен std::this_thread

std::thread::hardware_concurrency()

Возвращает количество thread, которые могут выполняться параллельно для данного приложения

std::this_thread::get_id()

Возвращает идентификатор потока

std::this_thread::sleep_for(std::chrono::milliseconds)

Позволяет усыпить поток на время

32. Потокобезопасность. Реентерабельность. Race condition. Взаимное исключение. std::mutex и std::recursive_mutex.

Потокобезопасность – свойство кода, предполагающее его корректное функционирование при одновременно исполнении несколькими потоками. Основные методы достижения – реентерабельность, локальное хранилище потока. Взаимное исключение, атомарные операции.

Реентерабельность – свойство функции предполагающее ее корректное исполнение во время повторного вызова (не должна работать со статическими переменными, не должна возвращать адрес статических переменных, должна работать только с данными переданными вызывающей стороной, не должна модифицировать своего кода, не должна вызывать не реентерабельных программ и процедур

Потоконебезопасный код race_condition_1.cpp

```
void add_function( long * number){
    for( long i=0;i<1000000000L;i++) (*number)++;}

void subst_function( long * number){
    for( long i=0;i<1000000000L;i++) (*number)--;}

int main() {
    long number = 0;
    {
        Scoped_Thread th1(std::move(std::thread(add_function,&number)));
        Scoped_Thread th2(std::move(std::thread(subst_function,&number)));
    }
    std::cout << "Result:" << number << std::endl;
    return 0;
}
```

Состояние гонки вызывает проблему разделения объектов. Когда только читаем, проблем нет, но когда пишем, все может сломаться. Также, неизвестно, когда переключается контекст.

Для синхронизации таких потоков используется мьютекс – базовый элемент синхронизации, представлен в 4 форматах в заголовочном файле mutex.

Mutex – обеспечивает базовые функции lock() и unlock() и не блокируемый метод try_lock().

Timed_Mutex – в отличие от обычного мьютекса имеет еще два метода – `try_lock_for()` и `try_lock_until()` позволяющих контролировать время ожидания вхождения в мьютекс.

Recursive_mutex – может войти сам в себя, т.е. поддерживает многократный вызов `lock()` из одного потока, содержит все функции, что и `mutex`.

33.std::lock_guard и std::unique_lock. Реализация потокобезопасного стека. dead_lock. Просачивание данных за пределы lock_guard

Явная блокировка и разблокировка могут привести к ошибкам, например, если вы забудете разблокировать поток или, наоборот, будет неправильный порядок блокировок — все это вызовет deadlock. Std предоставляет несколько классов и функций для решения этой проблемы.

Классы «обертки» позволяют непротиворечиво использовать мьютекс в RAII-стиле с автоматической блокировкой и разблокировкой в рамках одного блока. Эти классы:

`lock_guard`: когда объект создан, он пытается получить мьютекс, а когда объект уничтожен, он автоматически освобождает мьютекс

`unique_lock`: в отличие от `lock_guard`, также поддерживает отложенную блокировку, временную блокировку, рекурсивную блокировку и использование условных переменных

RAII - lock_guard lock_guard_1.cpp

1. Захват в конструкторе
2. Освобождение в деструкторе
3. Используются методы мьютексов
 - Захват: `void lock();`
 - Освободить: `void unlock();`

RAll - unique_guard

lock_guard_2.cpp

1. То же, что lock_guard
2. Используются методы мьютексов
 - Попытаться захватить: bool try_lock();
 - Захват с ограничением: void timed_lock(...);
3. + Дополнительные функции получения мьютекса, проверки «захваченности»...

dead_lock

Одна из самых неприятных проблем, с которыми приходится столкнуться при программировании многопоточности, это Deadlock. Чаще всего он случается, когда поток уже заблокировал ресурс А, после чего пытается провести блокировку Б, другой же поток заблокировал ресурс Б, после чего пытается заблокировать ресурс А.

dead_lock.cpp

```
std::lock_guard<std::mutex>  
lock(a);  
  
std::lock_guard<std::mutex>  
lock(b);
```

```
std::lock_guard<std::mutex>  
lock(b);  
  
std::lock_guard<std::mutex>  
lock(a);
```

Возникает когда несколько потоков пытаются получить доступ к нескольким ресурсам в разной последовательности.

ПОТОКОБЕЗОПАСНЫЙ СТЕК:

Листинг 6.1. Определение класса потокобезопасного стека

```
#include <exception>

struct empty_stack: std::exception
{
    const char* what() const throw();
};

template<typename T>
class threadsafe_stack
{
private:
    std::stack<T> data;
    mutable std::mutex m;
public:
    threadsafe_stack(){}
    threadsafe_stack(const threadsafe_stack& other)
    {
        std::lock_guard<std::mutex> lock(other.m);
        data=other.data;
    }
    threadsafe_stack& operator=(const threadsafe_stack&) = delete;

    void push(T new_value)
    {
        std::lock_guard<std::mutex> lock(m);
        data.push(std::move(new_value));
    }

    std::shared_ptr<T> pop()
    {
        std::lock_guard<std::mutex> lock(m);
        if(data.empty()) throw empty_stack();
        std::shared_ptr<T> const res(
            std::make_shared<T>(std::move(data.top())));
        data.pop();
        return res;
    }

    void pop(T& value)
    {
        std::lock_guard<std::mutex> lock(m);
        if(data.empty()) throw empty_stack();
        value=std::move(data.top());
        data.pop();
    }
};
```

← 1

← 2

← 3

← 4

← 5

← 6

```

std::lock_guard<std::mutex> lock(m);
if(data.empty()) throw empty_stack();
value=std::move(data.top());
data.pop();
}

```



```

bool empty() const
{

```

Матрица, заданная в 01

224

Проектирование параллельных структур данных...

```

std::lock_guard<std::mutex> lock(m);
return data.empty();
}
};

```

https://books.google.ru/books?id=1UXRAAAQBAJ&pg=PA222&lpg=PA222&dq=%D0%A0%D0%B5%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F+%D0%BF%D0%BE%D1%82%D0%BE%D0%BA%D0%BE%D0%B1%D0%B5%D0%B7%D0%BE%D0%BF%D0%B0%D1%81%D0%BD%D0%BE%D0%B3%D0%BE+%D1%81%D1%82%D0%B5%D0%BA%D0%B0&source=bl&ots=rvFNxgaNRK&sig=ACfU3U1pgjdbg cQa_syp6I31eTVvB80wpA&hl=ru&sa=X&ved=2ahUKEwjYy30f7mAhVVxMQBH22nA14Q6AEwBHoECAoQAQ#v=onepage&q&f=false

(книга, где описан этот стек) страница 223 ..

Просачивание данных за пределы lock_guard

Ошибки в логике потокобезопасного кода, когда к данным получают доступ несколько потоков одновременно, при этом такого поведения быть не должно

34 Условные переменные . Закон Амдала std::future std::async std::promise

Шаблонный класс std::future обеспечивает механизм доступа к результатам асинхронных операций:

- Асинхронные операции (созданные с помощью `std::async`, `std::packaged_task`, или `std::promise`) могут вернуть объект типа `std::future` создателю этой операции.
- Создатель асинхронной операции может использовать различные методы запроса, ожидания или получения значения из `std::future`. Этим методы могут заблокировать выполнение до получения результата асинхронной операции.
- Когда асинхронная операция готова к отправке результата её создателю, она может сделать это, изменив *shared state* (например, `std::promise::set_value`), которое связано с `std::future` создателя.

`async` шаблон функции выполняет функцию `f` асинхронно (возможно, в отдельном потоке) и возвращает `std::future`, что будет в итоге результат этого вызова функции.

1) Ведет себя так же, как `async(std::launch::async | std::launch::deferred, f, args...)`. Иными словами, `f` может быть запущен в новый поток или он может быть запущен синхронно, когда в результате `std::future` запрашивается значение.

2) Вызывает функцию `f` с аргументами `args` в соответствии с определенной политикой запуск `policy`

- Если асинхронный флаг установлен (т.е. `policy & std::launch::async != 0`), то `async` порождает новый поток исполнения, как по `std::thread(f, args...)`, за исключением того, что если функция `f` возвращает значение или генерирует исключение, оно хранится в общем состоянии доступным через `std::future`, что `async` возвращается к вызывающему.
-
-
- Если отложенный флаг установлен (т.е. `policy & std::launch::deferred != 0`), то `async` новообращенных `args...` же, как и конструктором `std::thread`, но не породить новый поток выполнения. Вместо этого, ленивая оценка производится: при первом вызове, не приурочен ждать функции на `std::future`, что `async` возвращается вызывающему вызовет `f(args...)` должны быть выполнены в текущем потоке. Все последующие обращения к той же `std::future` будет возвращать результат сразу.
-
-
- Если оба `std::launch::async` и `std::launch::deferred` флаги установлены в `policy`, это до реализации, выполнять ли асинхронное выполнение или ленивые вычисления.

Возвращаемое значение

- `std::future` со ссылкой на возвращаемого значения функции.
-

Исключения

- Выдает `std::system_error` с ошибкой `std::errc::resource_unavailable_try_again` условию, если запуск политика `std::launch::async` и осуществление не в состоянии начать новую тему.

`std::promise` шаблон класса предоставляет возможности для хранения значения, которое позже приобрел асинхронно через `std::future` объекта


работа непосредственно с потоком и обещанием полностью закрыта высокоуровневой оберткой `async`, и фьючерсы всегда описывают именно в контексте использования функции `async`. В этом есть смысл, потому что иметь дело непосредственно с обещанием программисту приходится только в том редком случае, когда его не устраивает стандартный `async`, и он вынужден писать его аналог вручную.

Опишу, как работают условные переменные:

- Должен быть хотя бы один поток, ожидающий, пока какое-то условие станет истинным. Ожидающий поток должен сначала выполнить `unique_lock`. Эта блокировка передается методу `wait()`, который освобождает мьютекс и приостанавливает поток, пока не будет получен сигнал от условной переменной. Когда это произойдет, поток пробудится и снова выполнится `lock`.
- Должен быть хотя бы один поток, сигнализирующий о том, что условие стало истинным. Сигнал может быть послан с помощью `notify_one()`, при этом будет разблокирован один (любой) поток из ожидающих, или `notify_all()`, что разблокирует все ожидающие потоки.
- В виду некоторых сложностей при создании пробуждающего условия, которое может быть предсказуемым в многопроцессорных системах, могут происходить ложные пробуждения (*spurious wakeup*). Это означает, что поток может быть пробужден, даже если никто не сигнализировал условной переменной. Поэтому необходимо еще проверять, верно ли условие пробуждение уже после то, как поток был пробужден. Т.к. ложные пробуждения могут происходить многократно, такую проверку необходимо организовывать в цикле.
- `condition_variable`: требует от любого потока перед ожиданием сначала выполнить `std::unique_lock`
- `condition_variable_any`: более общая реализация, которая работает с любым типом, который можно заблокировать. Эта реализация может быть более дорогой (с точки зрения ресурсов и производительности) для использования, поэтому ее следует использовать только если необходима те дополнительные возможности, которые она обеспечивает

Закон Амдала (англ. *Amdahl's law*, иногда также *Закон Амдала-Уэра*) — иллюстрирует ограничение роста **производительности вычислительной системы** с увеличением количества **вычислителей**. **Джин Амдал** сформулировал закон в 1967 году, обнаружив простое по существу, но непреодолимое по содержанию ограничение на рост производительности при **распараллеливании вычислений**: «В случае, когда задача разделяется на несколько частей, суммарное время её выполнения на **параллельной системе** не может быть меньше времени выполнения самого медленного фрагмента»^[1]. Согласно этому закону, ускорение выполнения программы за счёт **распараллеливания** её **инструкций** на множестве вычислителей ограничено временем, необходимым для выполнения её последовательных инструкций. Закон Амдала показывает, что прирост эффективности вычислений зависит от алгоритма задачи и ограничен сверху для любой задачи с

$\{\displaystyle \alpha \neq 0\}$

. Не для всякой задачи имеет смысл наращивание числа процессоров в вычислительной системе.

Более того, если учесть время, необходимое для передачи данных между узлами вычислительной системы, то зависимость времени вычислений от числа узлов будет иметь **минимум**. Это накладывает ограничение на **масштабируемость** вычислительной системы, то есть означает, что с определенного момента добавление новых узлов в систему будет *увеличивать* время расчёта задачи.

35. Неблокирующие алгоритмы. Атомарные типы. CAS операции. Потокобезопасный стек на CAS- операциях.

Неблокирующие алгоритмы

1. Без препятствий (Obstruction-Free) - поток совершает прогресс, если не встречает препятствий со стороны других потоков
2. Без блокировок (Lock-Free) - гарантируется системный прогресс хотя бы одного потока
3. Без ожидания (Wait-Free) - каждая операция выполняется за фиксированное число шагов, не зависящее от других потоков

Атомарные типы `#include <atomic>`

`std::atomic<T>`

- операции гарантированно атомарные
- хоть и все еще происходит чтение-модификация-запись
- а реализовано это через блокировки и взаимодействие процессов

Атомарность:

1. Есть разделяемые переменные;
2. Доступ к разделяемым переменным осуществляется без использования механизмов блокировок;
 - 2.1. Переменные можно изменять/читать без появления “состояния гонки”
 - 2.2. Промежуточные состояния изменения переменных “не наблюдаемы”
3. Используется доступ к аппаратным атомарным инструкциям (fetch-and-add, xchg, cmpxchg);

Атомарные типы C++

`#include<atomic>`

```
std::atomic_bool //bool
std::atomic_char //char
std::atomic_schar //signed char
std::atomic_uchar //unsigned char
std::atomic_int //int
std::atomic_uint //unsigned int
std::atomic_short //short
std::atomic_ushort //unsigned short
std::atomic_long //long
std::atomic_ulong //unsigned long
std::atomic_llong //long long
std::atomic_ullong //unsigned long long
std::atomic_char16_t //char16_t
std::atomic_char32_t //char32_t
std::atomic_wchar_t //wchar_t
std::atomic_address //void*
```

Операции

- `bool is_lock_free()` // свободен ли тип от блокировок
- `void store(T val, std::memory_order)` //положить значение
- `T load(std::memory_order)` // достать значение
- `operator T` // автоматическое преобразование типа к T
- `exchange(T val)` // обменять (аналог swap)
- `compare_exchange_strong/weak` // чуть позже

CAS - операции

CAS-операции

- | | |
|---|--|
| <ol style="list-style-type: none">1. CAS — compare-and-set, compare-and-swap
<pre>bool compare_and_set(
 int* адрес_переменной ,
 int старое значение ,
 int новое значение)</pre>2. Возвращает признак успешности операции установки значения3. Атомарна на уровне процессора | <ol style="list-style-type: none">1. Является аппаратным примитивом2. Возможность продолжения захвата примитива без обязательного перехода в режим «ожидания»3. Меньше вероятность возникновения блокировки из-за более мелкой операции4. Быстрая |
|---|--|

Если значение переменной такое, как мы ожидаем — то меняем его на новое;

Основные операции

```
load() //Прочитать текущее значение  
store() //Установить новое значение  
exchange() //Установить новое значение и вернуть предыдущее  
compare_exchange_weak() // см. следующий слайд  
compare_exchange_strong() // compare_exchange_weak в цикле  
fetch_add() //Аналог оператора ++  
fetch_or() //Аналог оператора --  
is_lock_free() //Возвращает true, если операции на данном типе  
неблокирующие
```

Метод `atomic::compare_exchange_weak`

```
bool compare_exchange_weak( Ty& OldValue, Ty NewValue)
/**
Сравнивает значения которые хранится в *this с OldValue.


- Если значения равны то операция заменяет значение, которая хранится в *this на NewValue (*this= NewValue) , с помощью операции read-modify-write.
- Если значения не равны, то операция использует значение, которая хранится в *this, чтобы заменить OldValue (OldValue=*this).


*/
```

Потокобезопасный стек на CAS операциях

```
#include <atomic>
#include <algorithm>
#include <iostream>
#include <memory>
#include <thread>
#include <mutex>
#include <vector>
#include <sstream>

struct sync_stream : std::stringstream {
    inline static std::mutex mtx;
    ~sync_stream() {
        std::lock_guard<std::mutex> lck(mtx);
        std::cout << this->rdbuf();
        std::cout.flush();
    }
};

template <typename T> class stack {
private:
    struct node {
        node* next;
        std::shared_ptr<T> data;
        node(const T& d, node* n = 0): next(n), data(std::make_shared<T>(d)) {}
    };
    std::atomic<node*> head;
public:
```

```

void push(const T& data) {
    node* new_node = new node(data, head.load());
    while (!head.compare_exchange_weak(new_node->next, new_node));
}

std::shared_ptr<T> pop() {
    node* old_head=head.load(); // читаем old_head
    while (old_head && !head.compare_exchange_weak(old_head,
old_head->next));
    // old head может быть удален с момента получения
    return old_head ? old_head->data : std::shared_ptr<T>();
}

};

void pop_and_push(stack<int>* stack,int number)
{
    for(int i=0;i<10;i++) stack->push(i); // fill stack
    for(int i=0;i<10;i++) { // pop stack
        std::shared_ptr<int> res = stack->pop();
        sync_stream{} << ((res.get() !=
nullptr)?(std::to_string(*res.get())):"null");
    }
}

auto main() -> int {
    stack<int> my_stack;
    std::vector<std::thread> threads;
    for(int i=0;i<1000;i++)
threads.push_back(std::thread(pop_and_push,&my_stack,i));
    for(auto &a : threads) a.join();
    return 1;
}

```