Классы и объекты

Объектно-ориентированное программирование — это набор принципов программирования, центральным понятием которого являются понятия объекта и класса.

Класс представляет собой совокупность свойств и методов, общих для группы объектов.

Объект — это экземпляр (представитель) класса.

В C++ структуры также являются и классами, поэтому работа с классом очень похожа на работу со структурой. Однако у класса существует такое понятие, как области видимости элементов[1]. Для класса можно задавать три области видимости:

- `public` — общественная, публичная, открытая. Элементы видны отовсюду (как в структуре).
- `private` — приватная, закрытая. Элементы доступны только внутри класса.
- `protected` — защищенная. Понадобится позднее.

Синтаксис класса:

```
class имя_класса{
private:
    тип_1 имя_1;
    тип_2 имя_2;
    ...
    тип_n имя_n;
public:
    ...
protected:
    ...
};
```

Обратите внимание, после закрывающей скобки обязательно ставится ;

Как правило, из соображений безопасности поля делаются `private` или `protected`.

Для доступа к таким полям используются специальные методы setИмяПоля и getИмяПоля.

```
#include <iostream>
using namespace std;
class my_complex{
     double Re, Im;
     public:
     void setRe(double x){Re=x;}
     void setIm(double y){Im=y;}
     double getRe(){return Re;}
     double getIm(){return Im;}
};
```

[1] Строго говоря, области видимости можно задавать и для структур. Однако по соглашению это не используется. Все элементы структуры по умолчанию имеют область видимости public.

```
void print(my_complex X)
{
     cout<<X.getRe()<<"+"<<X.getIm()<<"i"<<endl;
}
int main() {
 my_complex A;
 A.setRe(3);
 A.setIm(5);
 print(A);
     return 0;
}
```

Мы создали класс my_complex с двумя полями double Re, Im.

Написали методы для получения и установки значений этих полей, а также внешнюю функцию для печати комплексного числа, которую, кстати, можно немного исправить, чтобы вывод был красивым.

Но с классом работать пока не очень удобно, видно, что для того, чтобы создать одно комплексное число, приходится проделать несколько действий.

Конструкторы

Для того, чтобы создавать экземпляры класса, существует замечательный инструмент, который называется Конструктор.

Явно вызвать конструктор нельзя, но можно сделать несколько конструкторов в классе и при создании объекта компилятор сам поймет, который из них нужен. Для этого используется механизм перегрузки. Соответственно требования к параметрам конструктора такие же, как требования к обычным перегружаемым функциям, мы это обсуждали в прошлом семестре.

Нюанс: в C++ конструктор можно считать методом класса особого вида. В других языках программирования конструкторы часто отделяют от методов.

Что мы должны знать о конструкторах?

1. Имя конструктора совпадает с именем класса.
2. Конструктор не возвращает значения, и тип возвращаемого значения для него тоже не пишется.
3. Конструкторов может быть несколько, они различаются параметрами.
4. Конструктор без параметров — это конструктор по умолчанию, если вы его создаете, то он подменяет автоматический конструктор.
5. Конструкторы не наследуются! это уже к следующим темам.

Мы будем использовать три конструктора: по умолчанию, конструктор копирования, конструктор преобразования.

```
my_complex(){Re=0; Im=0;} // пример конструктора по умолчанию
```

Как мы видим, возвращаемого значения нет, типа результата тоже нет, имя совпадает. Внутри мы создаем нулевое значение.

Вопрос студента: если бы мы его не написали, что было бы в полях? Он бы их занулил или просто обьявил?

Ответ: обнулил, но тут 2 нюанса:

1. Неизвестно, как себя поведут другие компиляторы, объект локальный, онлайн-компилятор часто обнуляет создаваемые переменные.

2. Не во всех случаях при создании нового объекта мы должны присвоить нулевые значения полям, даже если этот объект создается по умолчанию.

Для того и нужен собственный конструктор по умолчанию — чтобы гарантированно создать объект с нужными характеристиками, а не полагаться на компилятор. Вы будете решать задачи по вариантам в соответствующей лабораторной работе, в некоторых вариантах использовать нулевые значения точно не получится.

Следующий конструктор — это конструктор преобразования. Он позволяет создавать объект с переданными характеристиками. Это удобнее, чем создавать объект по умолчанию и затем поштучно менять все поля.

```
my_complex(double a, double b){Re=a; Im=b;} // преобразования
```

`return` использовать нельзя, т.к. конструктор не возвращает значения!

Вопрос студента: А как покинуть его в случае выполнения каких-либо условий?

Ответ: А зачем? Чтобы бросить создание объекта? Тут надо исходить из того, что конструктор должен в любом случае создать объект. И если невозможно создать объект с пользовательскими характеристиками, значит, должен быть создан объект с другими характеристиками. Но конструктор по умолчанию вызвать не получится, потому что конструкторы нельзя вызывать.

Преподаватель: какие варианты можете предложить в таком случае?

Студент 1: `return mycomplex().setRe(x).setIm(y)`?[2]

Преподаватель: Что-то слишком сложная конструкция получается. А если полей не два, а, скажем, 50?

Студент 1: Разве конструктор не возвращает объект класса?

Преподаватель: Конструктор ничего не возвращает.

[2] Этот код не будет работать. Подумайте, почему.

Студент 1: `my_complex A = my_complex(); return A;`[3] так тоже нельзя?

Преподаватель: Можно или нет, попробуйте сами. Я только не понимаю, куда вы хотите писать этот пример, потому что внутри конструктора делать return нельзя, а вытаскивать создание объекта из класса — это нарушение принципа инкапсуляции. Про инкапсуляцию позже расскажу.

Студент 2: Прописать конструктор для «непользовательских характеристик»? Закрыть брешь конструкторами.

Преподаватель: Это не решает проблемы, что конструкторы нельзя вызывать явно.

Студент 2: Но их же можно вызывать неявно? Значит надо создать ситуацию, когда конструктор будет гарантировано вызван. Что-то вроде триггера? То, что мы сможем спрогнозировать?

Преподаватель: Да, идея примерно такая. Только конструктор всегда будет вызван, всегда. Программист на это влиять не может. Поэтому и делается дополнительная функция.

Ответ преподавателя: Эта проблема решается следующим образом: создается дополнительная приватная функция, которая заполняет поля значениями, и она вызывается внутри конструктора. Эта функция может возвращать значения и ее можно использовать несколько раз.

Третий конструктор, который мы рассмотрим — это конструктор копирования. Он нужен, чтобы создавать копию существующего объекта.

```
my_complex(my_complex &X); // конструктор копирования
```

Здесь я воспользовалась вторым вариантом описания конструкторов и методов — описала конструктор вне класса, а внутри оставила только заголовок метода.

```
my_complex::my_complex(my_complex &X){
     Re=X.Re;
     Im=X.Im;
}
```

Сам метод описан снаружи.

Чтобы компилятор понимал, что это метод, а не функция, то добавлен префикс:

```
имя класса::
```

Студент 1: Зачем разрешено выносить код?

Ответ преподавателя: Описывать функции вне класса удобнее, когда функции большие. Чтобы было проще работать с классом. А когда мы работаем с многофайловыми

[3] И так тоже нельзя.

проектами, то вынос функций становится по факту обязательным, чтобы отделять интерфейс класса от его реализации. В классе только заголовки методов, а все описания снаружи. Я показываю вам оба способа, потому что для маленьких методов (которые умещаются в одну строчку) часто проще их не выносить, но в реальных проектах обычно выносят, так как скорее это соглашение. Вроде как делать отступы в коде: можно без них, но с ними удобнее всем.

Итак, в лабораторной работе вам нужно будет сделать три конструктора для вашего варианта. У некоторых, возможно, появится еще и четвертый конструктор (второй конструктор преобразования) для удобства работы, но у всех точно будут три штуки.

Деструктор

Он, как можно догадаться из названия, уничтожает объект. Деструктор есть всегда, но вы можете прописать его явно. Для обычных классов и объектов деструктор можно не создавать, т.к. он все равно будет пустым, но деструктор обязательно нужен, если вы работаете с динамической памятью.

Например, в первых двух лабораторных работах мы создавали списки и остальное на обычных структурах. Но можно вспомнить, что в C++ структура по факту тоже класс. В таких случаях при уничтожении объекта надо еще и память за ним почистить, т.е. в случае связного списка сделать деструктор, который будет вызывать функцию, удаляющую весь список. Можно, конечно, и прямо в деструкторе расписать удаление, но зачем дублировать код, если функция уже есть и нужна, например, если вы хотите удалить содержимое объекта без уничтожения?

Но в нашем примере мы работать с динамической памятью не будем, так что я просто покажу, как сделать пустой деструктор.

Итак, имя деструктора — это имя класса, перед которым ставится ~

У деструктора нет аргументов! Он не возвращает значения и не наследуется, так же, как и конструктор.

```
~my_complex(){}; //деструктор
```

Можете самостоятельно попробовать прикрутить деструктор к лабораторным 1 и 2.

Студент 1: А как удалять объект? Или он вызывается автоматически при закрытии программы?

Преподаватель: Да, автоматически.

Студент 1: А если обьект класса динамический?

Преподаватель: Но не только при закрытии программы, а например при выходе из функции, где был объявлен локальный объект. Если объект динамический, он тоже вызывается. И если деструктор не прописан, происходит утечка памяти.

Студент 1: А как вызвать?

Преподаватель: Деструктор? Никак. А если надо удалить объект, то я выше уже написала, что делать.

Студент 1: То есть делать делит после объявления динамического объекта класса не надо?

Преподаватель: Если вы сделали деструктор, который его уничтожает, то не надо. Не сделали – надо. Для того и деструктор чтобы программисту было проще — один раз написал уничтожение объекта и больше не думаешь о том, что его каждый раз надо удалять.

Студент 1: Если единственный указатель перенести с объекта на другой, то объект сам вызовет деструктор?

Преподаватель: Указатель — это не переменная! Если у вас был объект и указатель на него, вы сделали другой указатель, указывающий на этот объект, а старый перенесли или даже приравняли к нулю, ничего с объектом не будет! Пока на объект указывает хотя бы один указатель, он жив. Если, конечно, не пришло время его уничтожить. А вот если у вас на объект указывал один указатель и больше ничего, и вы перенесли указатель на ноль или на другой объект, то первый объект будет висеть в памяти.

Студент 1: `myclass *A= new myclass(); A= new myclass();` Если две этих команды подряд вызвать, первый объект сам вызовет деструктор?

Преподаватель: Нет, не вызовет. Это пример утечки памяти. Надо было сначала вызвать `delete` для A, который запустит деструктор, и только потом выделять новую память.

Статические поля и методы класса

Если вы объявляете какое-то поле с префиксом `static`, оно будет общим для всех объектов этого класса.

Где это можно использовать? Например, в классе для работы с линейным списком. Если сделать в структуре статическое поле для хранения количества элементов в списке, то тогда для любого элемента в списке вы будете видеть сколько в списке элементов, и не надо дополнительную переменную. Соблюдается инкапсуляция, то есть все в одном месте. Есть только одна проблема — количество будет одним и тем же для любой переменном, значит, вы не сможете использовать это больше чем для одного списка.

Решением проблемы будет класс-обертка вокруг структуры для хранения элементов списка, но мы это не будем рассматривать, потому что не успеваем, да и в лабораторных у нас нет такого задания. Просто знайте, что есть такая возможность.

Статистические методы нужны для работы со статическими полями. Подробности можно почитать в учебнике Павловской, стр. 186-187.

Студент 1: К статическому полю можно обратиться без объекта класса без статического геттера?

Преподаватель: А причем тут статичность поля? Оно либо публичное, либо приватное. К публичному можно обратиться отовсюду, к приватному только из класса.

Дружественные функции

Как быть, если очень хочется обратиться к полю, а оно приватное? Можно через геттеры, да. Но иногда их слишком много нужно. тогда можно попробовать чуток нарушить принцип инкапсуляции, который говорит нам о том, что все, что относится к классу, остается внутри класса. Здесь и используются дружественные функции.

Что нам нужно про них знать, кроме того, что они нарушители?

1. Дружественные функции не наследуются.
2. Они обязательно объявляются внутри класса с помощью модификатора `friend`.
3. Они могут иметь доступ к скрытым полям и методам класса.

Давайте попробуем сделать нашу функцию печати дружественной и посмотрим, что изменится.

```
friend void print(my_complex X); // дружественная функция
```

Вот так мы ее объявим внутри класса…

```
void print(my_complex X)
{
     cout<<X.Re<<"+"<<X.Im<<"i"<<endl;
}
```

…а вот так изменилась сама функция

Ну и я ее еще поправлю, чтобы была посимпатичнее.

```
void print(my_complex X)
{
     if (abs(X.Im)<0.0000001) {cout<<X.Re<<endl; return;}
     if (X.Im>0)
          cout<<X.Re<<"+"<<X.Im<<"i"<<endl;
     else
          cout<<X.Re<<X.Im<<"i"<<endl;
}
```

Обратите внимание, сначала я исключаю нулевое значение, не забывая о том, что поля у нас нецелые, а, значит, не могут в точности быть нулевыми, а уже потом вывожу ненулевые числа.

Студент 1: А нельзя на сиаут повесить модификатор «выводить числа всегда со знаками».

Преподаватель: можно, но тогда придется подключить манипуляторы, а вообще суть не в том.

Да, исключаю не просто нулевое значение, а нулевое значение мнимой части, чтобы не таскать некрасивый хвостик.

Перегрузка операций

Вы, возможно, обратили внимание, что я пока не привела ни одного примера функций для работы с комплексными числами. Можно было бы написать функцию `summa()`, которая бы вычисляла сумму двух чисел, но гораздо удобнее было бы сделать так, чтобы сумму можно было посчитать с помощью привычной нам операции + . Вот для этого и используется перегрузка операций.

Базовые принципы перегрузки такие:

1. Должно сохраняться количество аргументов и приоритеты, то есть + будет складывать только два числа, а не пять, и * будет приоритетнее сложения.
2. Нельзя перегрузить операции для стандартных типов данных, только для классов.
3. Перегруженные операции наследуются, кроме операции присваивания.
4. Операции можно перегружать как внутри класса, так и вне его (как дружественные функции). Если пишете операцию внутри класса, у нее будет на один аргумент меньше.

Синтаксис

```
тип operator операция(список аргументов){тело функции}
```

Операции могут быть унарные и бинарные, давайте рассмотрим примеры.

```
friend bool operator ==(const my_complex X, const my_complex Y);
```

Это объявление внутри функции.

```
bool operator ==(const my_complex X, const my_complex Y)
{
     if (abs(X.Re-Y.Re)<0.0000001 && abs(X.Im-Y.Im)<0.0000001)
          return true;
     return false;
}
```

А вот и сам оператор. Программа выводит «yes», так как B и C у нас равны. Аналогичным образом легко перегрузить оператор !=.

С другими операциями сравнения придется сложнее, т.к. это комплексные числа. Их не будем рассматривать.

Теперь попробуем со сложением, но уже внутри класса. Тут у нас будет другой тип результата

```
     my_complex operator +(my_complex X){
          my_complex S;
          S.Re=Re+X.Re;
          S.Im=Im+X.Im;
```

```
        return S;
    }
```
Как видим, сложить получается (компилятор не ругается), а вот присвоить и даже распечатать нет. Надо перегрузить оператор присваивания.

Студент 1: что возвращает оператор присваивания?

Преподаватель: ссылку на объект.

```
    my_complex& operator=(const my_complex &X){
        Re=X.Re;
        Im=X.Im;
        return *this;
    }
```
Давайте разбираться. Понятно, что присваивание будет поэлементным. Тут возникает вопрос, а как возвратить результат? Здесь как раз тот случай, когда надо использовать конструкцию `this`. Это скрытый параметр, он есть у каждого класса. Он скрыто передается в каждый метод

`this` — это указатель на текущий объект. Поэтому правильнее было бы писать не `Re`, а `this->Re`. Но читаемость кода это никак не улучшает, поэтому обычно его опускают, используют только в случаях, когда без него не обойтись. Например, когда его нужно возвратить в качестве результата.

Казалось бы, все хорошо, присваивание работает, можно дописать остальные функции комплексной арифметики и использовать наш класс для решения задач, но мы кое-что забыли: мы забыли исключить самоприсваивание, т.е. когда переменной присваивают саму себя.

```
    my_complex& operator=(const my_complex &X){
        if (this == &X) return *this;
        Re=X.Re;
        Im=X.Im;
        return *this;
    }
```
Вот как будет выглядеть исправленная функция.

Это нужно для более высокой производительности. С двумя числовыми полями оно не будет заметно, а вот с другими типами может.

Студент 1: а теперь `print(B+C);` сработает?

Преподаватель: `print(B+C);` пока не работает, потому что мы не перегрузили скобочки. если хотите сделать чтобы работало, инструкции на странице 196 и далее учебника.

…и избавимся еще и от функции печати, заменив ее на привычное <<

Потому что это тоже операция, и ее тоже можно перегружать. Правда тут придется залезть чуть вперед по темам, поэтому пока просто пример без подробностей, но с ним будет удобнее делать задания лабораторной работы.

```
std::ostream& operator<<(std::ostream& out, const my_complex &X)
{
     if (abs(X.Im)<0.0000001) {out<<X.Re<<endl; return out;}
     if (X.Im>0)
          out<<X.Re<<"+"<<X.Im<<"i"<<endl;
     else
          out<<X.Re<<X.Im<<"i"<<endl;
     return out;
}
```

Это вывод. Операция перегружена как дружественная функция.

```
friend std::ostream& operator<<(std::ostream& out, const
my_complex &X);
```

Это, кстати, общая рекомендация для перегрузки операции вывода.

```
my_complex D;
D=B+C;
cout<<D<<endl;
cout<<B+C<<endl;
```

Мы видим, что сумма в этом случае тоже работает. Здесь мы передаем потоковую переменную: отправляем в нее вывод и ее возвращаем. В таком случае, наверное, стоит удалить из этого метода endl, чтобы печать комплексного числа вообще не отличалась от других.

Студент 1: А когда выделяется память под статические поля? При создании первого объекта класса? Или в начале работы программы? Можно к публичным по имени класса обращаться или нужен экземпляр класса?

Преподаватель: В зависимости от того, когда выделяется память под первый экземпляр класса. Экземпляр нужен, конечно.