Novosibirsk State University
*THE REAL SCIENCE

Программирование на C++ и Python

Лекция 1

От структур к классам

Воробьев Виталий Сергеевич (ИЯФ, НГУ)

Новосибирск, 16 сентября 2020

Цели курса

1. Познакомить с концепциями современного программирования
   - Структуры данных и алгоритмы
   - Парадигмы программирования
2. Дать первоначальные навыки разработки на языках *C++* и Python
3. Познакомить с инструментами для совместной разработки программ

За один семестр невозможно стать
профессиональным программистом (да и не надо!).
После прохождения курса вам будет проще
осваивать эти и другие языки программирования
самостоятельно.

Программа курса

Лекции

C++

1. Объектно-ориентированное программирование I
2. Объектно-ориентированное программирование II
3. Стандартная библиотека
4. Обобщенное и функциональное программирование

Python

1. Введение в Python
2. Стандартная библиотека
3. Вычисления с Scipy I
4. Вычисления с Scipy II

Задачи

C++

1. Потоки ввода-вывода, строки
2. Контейнеры стандартной библиотеки
3. Алгоритмы стандартной библиотеки
4. Объектно-ориентированное программирование
5. Обобщенное программирование

Python

1. Основы языка
2. Стандартная библиотека
3. Работа с numpy
4. Работа с matplotlib
5. *Работа с pygame (не обязательно)*

Ресурсы

1. Сайт курса:
   - https://nsu-programming.github.io
2. Репозиторий с лекциями:
   - https://github.com/NSU-Programming/lectures2020
3. Сайт с баллами за задачи:
   - https://cpp-python-nsu.inp.nsk.su/
4. Telegram-группа (QR-код)

Зачем вам 

- *C++* быстрый и он развивается
  - **Быстрый**: *C++* – основной язык разработки в коммерческих и научных проектах, в которых важна эффективность (OS X, MS Windows, Adobe Photoshop, Tensorflow, Firefox, Chromium, Skype и ещё **очень** много чего)
  - **Развивается**: на современном *C++* можно писать ясный и надёжный код
- Философия *C++*: **не платить за то, что не используешь (zero cost abstractions)**
- Преподаватели этого курса считают, что выпускник физфака НГУ должен иметь представление о написании эффективного кода

Hello!

C

```c
#include <stdio.h>

int main() {
  char str[100];
  printf("What is you name? ");
  scanf("%s", str);
  printf("Hello, %s!", str);
  return 0;
}
```

C++

```cpp
#include <iostream>
#include <string>
using namespace std;
int main() {
  cout << "What is your name? ";
  string name;
  cin >> name;
  cout << "Hello, " << name
       << "!" << endl;
  return 0;
}
```

Структуры в *C*

- В сложных программах неизбежно приходится создавать собственные типы данных. В языке *C* это *структуры*:

```
typedef struct {
  int d;
  int m;
  int y;
} Date;  // тип

Date today;  // экземпляр
today.d = 16;
today.m = 9;
today.y = 2020;
```

```
typedef struct {
  int h;
  int m;
  int s;
} Time;

Time now = {12, 55, 00};
```

Структуры как абстракции

- Структуры позволяют создавать абстракции в программе
- Структура – это контейнер, все действия с которым надо производить самостоятельно
- Если для структуры определить функции, то мы перейдем на новый уровень абстракции – структуры «оживут»
- Получится простейший *класс* – «структура с функциями»

```
void print_date(Date d) {
  printf("Date is %02d/%02d/%04d\n",
    d.d, d.m, d.y);
}
void print_time(Time t) {
  printf("Time is %02d:%02d:%02d\n",
    t.h, t.m, t.s);
}
int main() {
  Date today;
  Time now;
  print_date(today);
  print_time(now);
}
```

Простейший класс

- Определим `Date` и `Time` как классы
- Описание структуры данных (*полей*) и действий, которые можно проводить с данными (*методов*), называется *классом*
- Конкретный экземпляр такой структуры данных называется *объектом*

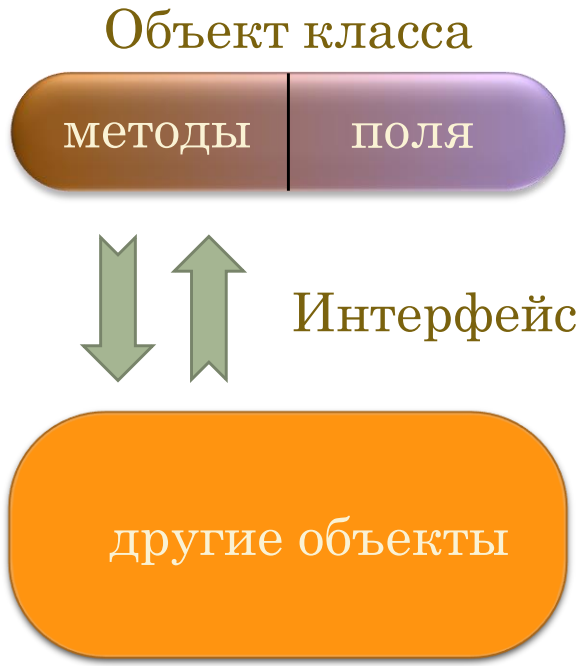
```
class Date {  // класс
public:
  int d, m, y;  // поля класса

  void print() {  // метод класса
    cout << "Date is " << d << '/'
      << m << '/' << y << endl;
  }
};
// аналогично класс Time
int main() {
  Date today = {16, 9, 2020}; // объект
  Time now = {13, 05, 00};
  today.print();
  now.print();
}
```

Инкапсуляция

- Дату можно хранить как три числа: (день, месяц, год), или в виде одного числа (день $\times 10^6$ + месяц $\times 10^4$ + год):

  (16, 9, 2020) или 16092020

- Как правильно? Это решает разработчик класса. Но программа, в которой *используется* этот класс, не должна зависеть от внутреннего представления даты
- Решение: скрыть от пользователя внутреннее представление (обычно скрывают все поля класса) и работать с классом только через разрешенные методы (интерфейс класса)
- *Инкапсуляция* – объединение данных и методов работы с ними и сокрытие внутренних деталей
- Для управления доступом к полям и методам в *C++* используются ключевые слова `private` и `public`



Private и public

- По умолчанию все члены класса скрыты
- Скрывать и объявлять публичными можно как методы, так и поля класса

```
class Date {
  int d, m, y; // private часть

public:
  void print() {
    cout << "Date is " << d << "/"
         << m << "/" << y << endl;
  }
};

int main() {
  Date today = {16, 9, 2020}; // Ошибка!
  today.print();
  today.d = 16; // Ошибка!
}
```

Геттеры и сеттеры

- Поля класса обычно объявляют приватными, а доступ к ним (при необходимости) организуют через специальные методы
- В методах доступа к полям классов можно делать различные проверки
- Такой подход позволяет изменять представление данных класса, не изменяя интерфейс

```
class Date {
  int date; // ddmmyyyy
public:
  void setMonth(int m) {
    if (m > 0 && m < 13) {
      date += (m - getMonth()) * 10000;
    }
  }
  int getMonth() {
    return date / 10000 % 100;
  }
};

int main() {
  Date today;
  today.setMonth(9);
}
```

Перегрузка функций

- В *C++* функция определяются не только именем, но и аргументами
- Можно определить несколько функций с одним и тем же именем, но разными аргументами

```
class Date {
  int d, m, y;

public:
  int month() const { return m; }
  void month(int month) { m = month; }
  void month(const string& month) {/* … */}
};

int main() {
  Date today;
  today.month(9);
  today.month("September");
  cout << "Today month is "
       << today.month() << endl;
}
```

Конструктор и деструктор

- При создании, структуры инициализировались поэлементно. Класс, внутренняя структура которого скрыта, так инициализировать нельзя – мы не знаем какие поля класса нужны и не имеем доступа к ним
- Нужен специальный метод доступа: *конструктор*, который инициализирует объект
- Имя конструктора совпадает с именем класса. В классе может быть несколько конструкторов с различными аргументами
- Раз существует метод инициализации (создания) объекта, должен быть и метод удаления объекта – *деструктор*

```
class Date {
  int packed_date; // ddmmyyyy
public:
  // конструктор по умолчанию
  Date() { packed_date=0; }
  // конструктор
  Date(int day, int month, int year) {
    packed_date = day*1000000+month*10000+year;
  }
  // конструктор
  Date(int pack) {packed_date = pack; }
  // деструктор
  ~Date() = default;
};

int main() {
  Date today(16, 9, 2020);
  Date tomorrow(17092020);
  Date someday;
}
```

Объявление и определение

```
class Date {
  int packed_date; // ddmmyyyy
public:
  Date();
  Date(int); // ddmmyyyy
  Date(int, int, int);
  ~Date() = default;

  int day() const;
  int month() const;
  int year() const;
  void print() const;
};
```

Date.h: *объявление*
(открытый интерфейс)

```
#include "Date.h"
#include <iostream>
using namespace std;
Date::Date() { packed_date = 0; }
Date::Date(int day, int month, int year) {
  packed_date = day * 1000000 + month * 10000 + year;
}
int Date::day() const { return packed_date / 1000000;}
int Date::month() const {
    return packed_date / 10000 % 100;
}
int Date::year() const { return packed_date % 10000; }
void Date::print() const {
  cout << "Date is "
    << day() << '/' << month() << '/' << year() << endl;
}
```

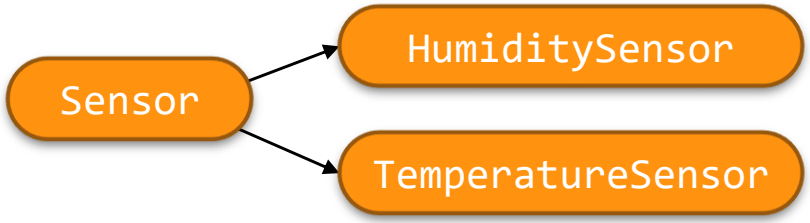
Date.cpp: *определение*
(скрытая реализация)

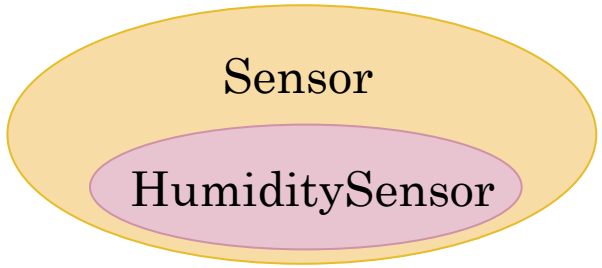
main.cpp: использование 

```
#include "Date.h"

int main() {
  Date d;
}
```

Наследование

- Рассмотрим пример разработки ПО для системы мониторинга окружающей среды. В нашей системе есть сенсоры влажности и температуры
- Для работы с сенсорами мы разработали классы `HumiditySensor` и `TemperatureSensor`. Это разные сенсоры, но у них есть общая функциональность – например, у них есть значение, их можно прочитать и т.п.
- Можно выделить общую функциональность этих классов в *базовый класс* `Sensor` и унаследовать `HumiditySensor` и `TemperatureSensor` от него. Классы-наследники получают всю функциональность базового класса и дополняют её
- Принцип (публичного) наследования в *C++*: множество объектов класса-наследника является подмножеством объектов базового класса. Про каждый объект класса-наследника можно сказать, что он *является* также объектом базового класса. Если эта логика не выполняется, то наследование лучше не применять





```
class Sensor {
  double value;
public:
  void setValue(double);
  double getValue() const;
  void print() const { cout << "Sensor value is " << value << endl;}
};
```

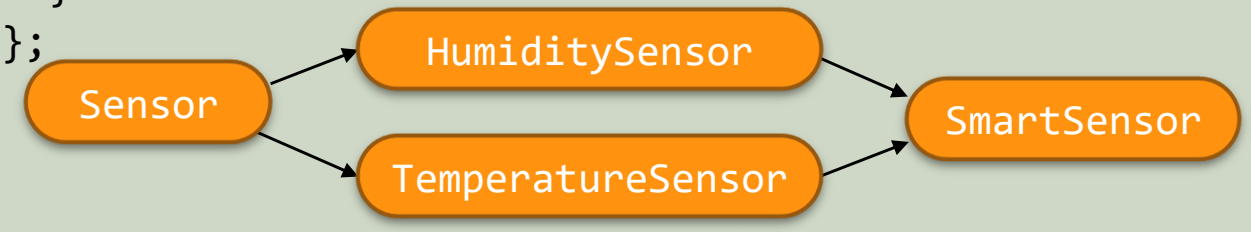
```
class HumiditySensor : public Sensor {
public:
  void measure() { /* do_something */ setValue(x);}
  // переопределение
  void print() const { cout << "Humidity is " << getValue() << endl; }
};
```

```
class TemperatureSensor : public Sensor {
public:
  void measure() { … }
  // переопределение
  void print() const { cout << "Temperature is " << getValue() << endl;}
};
```

Множественное наследование

- Теперь нам необходимо описать многофункциональный сенсор, который может измерять и влажность, и температуру
- Одно из решений – унаследовать класс `SmartSensor` от классов `HumiditySensor` и `TemperatureSensor`. Это не лучшее решение в данном случае по нескольким причинам (одна из них – мы унаследовали базовый класс `Sensor` дважды)

```
class SmartSensor :
  public HumiditySensor, public TemperatureSensor {
public:
   void measure() {
    // вызываем методы базовых классов
    HumiditySensor::measure();
    TemperatureSensor::measure();
  }
};
```




```
int main() {
  SmartSensor s;
  s.measure();
  s.print(); // Ошибка! Какой print() вызвать?
}
```

Множественное наследование

- Теперь нам необходимо описать многофункциональный сенсор, который может измерять и влажность, и температуру
- Одно из решений – унаследовать класс `SmartSensor` от классов `HumiditySensor` и `TemperatureSensor`. Это не лучшее решение в данном случае по нескольким причинам (одна из них – мы унаследовали базовый класс `Sensor` дважды)
- Проще включить объекты простых сенсоров в объект умного сенсора

```
class SmartSensor {
  HumiditySensor hs;
  TemperatureSensor ts;

public:
  void measure() {
    hs.measure();
    ts.measure();
  }
  void print() const {
    hs.print();
    ts.print();
  }
};
```

Указатель на базовый класс

- В любом объекте-наследнике содержится объект базового класса
- Указатель на объект-наследник можно преобразовать в указатель на объект базового класса.

```
HumiditySensor h;
h.setValue(10.0);

h.print();  // Humidity is 10

HumiditySensor *ptr_h = &h;
ptr_h->print();  // Humidity is 10

Sensor *ptr_s = &h;
ptr_s->print();  // Sensor value is 10

ptr_h->measure();  // Ok

ptr_s->measure();  // Error!
                   // В базовом классе нет measure()
```

Полиморфизм

- Наследование в *C++* открывает путь к *полиморфизму* – возможности использования объекта с известным интерфейсом, но неизвестным типом
- Работать с объектом-наследником можно через указатель базового класса
- Для реализации этой идеи необходимы *виртуальные методы*

```
HumiditySensor h;
h.setValue(10.0);
TemperatureSensor t;
t.setValue(25.0);

Sensor* sensors[2] = { &h, &t };

for(int i = 0; i < 2; ++i)
  sensors[i]->print();
// Хотим получить: Humidity is 10
//                 Temperature is 25
// Получим:        Sensor value is 10
//                 Sensor value is 25
```

```
class Sensor {
  double value;
public:
  void setValue(double);
  double getValue() const;
  void print() const { cout << "Sensor value is " << value << endl;}
};
```

```
class HumiditySensor : public Sensor {
public:
  int measure() { /* do_something */ setValue(x);}
  // переопределение
  void print() const { cout << "Humidity is " << getValue() << endl; }
};
```

```
class TemperatureSensor : public Sensor {
public:
  int measure() { … }
  // переопределение
  void print() const { cout << "Temperature is " << getValue() << endl;}
};
```

```cpp
class Sensor {
  double value;
public:
  void setValue(double);
  double getValue() const;
  virtual void print() const {cout << "Sensor value is " << value << endl;}
};
```

```cpp
class HumiditySensor : public Sensor {
public:
  int measure() { /* do_something */ setValue(x);}
  // переопределение
  void print() const override { cout << "Humidity is " << getValue() << endl; }
};
```

```cpp
class TemperatureSensor : public Sensor {
public:
  int measure() { … }
  // переопределение
  void print() const override { cout << "Temperature is " << getValue() << endl;}
};
```

Виртуальные методы

- Виртуальные методы связываются с объектом *во время исполнения* программы (dynamic binding)
- При вызове виртуального метода программа определяет тип объекта и вызывает соответствующий метод

```
HumiditySensor h;
h.setValue(10.0);

h.print();  // Humidity is 10

HumiditySensor *ptr_h = &h;
ptr_h->print();  // Humidity is 10

Sensor *ptr_s = &h;
ptr_s->print();  // Humidity is 10

ptr_h->measure();  // Ok
ptr_s->measure();  // Error!
```

Абстрактные классы

- Базовый сенсор `Sensor` нельзя измерить, поэтому мы не определили для него `measure`. Но мы хотим потребовать, чтобы в классах-наследниках `measure` был определен. Для этого можно использовать чистый виртуальный (абстрактный) метод
- Класс, в котором есть чистые виртуальные методы, называется абстрактным. Объекты таких классов создать невозможно

```
class Sensor {
public:
  virtual void measure() = 0;
};

class HumiditySensor : public Sensor {
public:
   void measure() override { … }
};

class TemperatureSensor : public Sensor {
public:
   void measure() override { … }
};
```

```
class Sensor {
  double value;
public:
  void setValue(double);
  double getValue();
  virtual void print() { cout << "Sensor value is " << value << endl; }
};

class Measurable {
public:
  virtual int measure() = 0;
};

class HumiditySensor : public Sensor, public Measurable {
public:
  int measure() override { … }
  void print() override { cout << "Humidity is " << getValue() << endl; }
};

class TemperatureSensor : public Sensor, public Measurable {
public:
  int measure() override { … }
  void print() override { cout << "Temperature is " << getValue() << endl; }
};
```

Суммируем

- Классы и объекты позволяют создавать абстракции, объединяющие данные (поля класса) и методы работы с ними
- С классом работают с помощью его интерфейса, внутренняя структура класса скрыта (инкапсуляция). Принято разделять файлы с объявлением (.h,.hh) и определением (.cc, .cxx, .cpp) классов
- Для инициализации объектов в классах используются специальные методы (конструкторы)
- Наследование позволяет описать новый класс на основе уже существующего
- Виртуальные методы позволяют использовать интерфейс объекта, не зная его точный тип (полиморфизм)

Ресурсы по *C++*

- https://en.cppreference.com/ – полная документация
- https://isocpp.org/ – Standard *C++* Foundation
- [Поисковик] + https://stackoverflow.com/. Запросы лучше писать по-английски
- https://www.boost.org/ – peer-review библиотеки, многие из которых войдут в стандарт *C++*
- Coursera: специализация Яндекс + ВШЭ и др.
- Задачи online: hackerrank.com и др.
- Книги
  - Bjarne Stroustrup
  - Scott Meyers
  - …

Backup

Стандартная библиотека

isocpp.org

- Стандартная библиотека *C++* содержит множество полезных инструментов

Строки
- string
- regex

Библиотеки C
[21 файл]

Потоки ввода-вывода
- ios
- iostream
- fstream
- iomanip
- sstream
- iosfwd
- ostream
- streambuf

Контейнеры STL
- set
- list
- array
- map
- bitset
- vector
- stack
- queue
- forward_list
- unordered_set
- unordered_map

Численные библиотеки
- complex
- random
- valarray
- numeric

Вспомогательные
- exception
- limits
- new
- typeinfo

Общие
- algorithm
- chrono
- memory
- functional
- iterator
- stdexcept
- tuple
- utility

Многопоточность
- thread
- mutex
- future
- condition_variable

Локализация
- locale
- codesvt

(Перечислены не все заголовочные файлы, но большая их часть)