Алгоритмы и структуры данных

Семинар 2

Парадигмы программирования

- Процедурное программирование: определите, какие процедуры вам нужны; используйте лучшие из известных вам алгоритмов!

- Модульное программирование: определите, какие модули нужны; поделите программу так, чтобы данные были скрыты в этих модулях

- Абстракция данных: определите, какие типы вам нужны; предоставьте полный набор операций для каждого типа.

Классы

- Определение пользовательских типов
- Работа с пользовательскими типами так же удобна, как и со встроенными
- Класс - это тип, определяемый пользователем

Классы

- Хотим объединить данные и методы работы с ними

```
struct Complex {
    double real;
    double imaginary;
};

Complex number;
number.real = 5;
number.imaginary = 10;
```

- Хотим внести все операции с типом внутрь самого типа!

Классы

```
class Complex {
private:
   double real_;
   double imaginary_;
public:
   ...
};
```

Теперь number.real_ - не скомпилируется!

Классы

```
class Complex {
private:
   double real_;
   double imaginary_;
public:
   double real() const {
      return real_;
   }
   double imaginary() const {
      return imaginary_;
   } // еще добавить abs
};
```

Терминология

- Переменные внутри класса - поля класса.

- Функции, объявленные внутри класса - методы класса

Аргумент this

```
class Complex {
private:
   double real_;
   double imaginary ;
public:
   double real() const { // real(Complex* this)
      return this->real_;
   }
   double imaginary() const { // imaginary(Complex* this)
      return (*this).imaginary_;
   }
};
```

Использование полученного класса

```
Complex number;
std::cout << number.real() << " + i" << number.imaginary();
```

Конструкторы

- Конструктор используется для начальной инициализации объекта.

- Конструктор может как иметь параметры (данные, с помощью которых инициализируются объект), так и не иметь таковых.

- Отдельно выделяют особый тип конструктора - конструктор копирования.

Конструкторы

```
class Complex {
public:
   Complex() {
      real_ = 0.0;
      imaginary_ = 0.0;
   }
   Complex(double r, double i = 0.0) {
      real_ = r;
      imaginary_ = i;
   }
private:
   double real_;
   double imaginary_;
};

Complex number1;
Complex number2(5.0, 10.0);
```

Конструкторы. Списки инициализации

```
class Complex {
public:
   Complex()
      : real_(0.0)
      , imaginary_(0.0)
   {}
   Complex(double r, double i = 0.0)
      : real_(r)
      , imaginary_(i)
   {}
private:
   double real_;
   double imaginary_;
};
```

Конструктор копирования

```
class Complex {
public:
  Complex(const Complex& other)
    : real_(other.real_)
    , imaginary_(other.imaginary_)
  {}
private:
  double real_;
  double imaginary_;
};
```

Деструктор

```
class Complex {
public:
   ~Complex()
   {}
private:
   double real_;
   double imaginary_;
};
```

Чему мы научились?

- Объявлять классы
- Объявлять методы для работы с ними
- Инициализировать объекты нашего нового типа
- Определять поведение при удалении объекта

Прибавление к комплексному числу другого комплексного числа

```
class Complex {
public:
  Complex& add(const Complex& other) {
    real_ += other.real_;
    imaginary_ += other.imaginary_;
    return *this;
  }
private:
  double real_;
  double imaginary_;
};
```

Перегрузка операторов

```
class Complex {
public:
   Complex& operator+=(const Complex& other) {
      real_ += other.real_;
      imaginary_ += other.imaginary_;
      return *this;
   }
private:
   double real_;
   double imaginary_;
};

int main() {
   Complex a;
   Complex b;
   a += b;
   return 0;
}
```

Перегрузка операторов

```
class Complex {
public:
   Complex& operator+=(const Complex& other) {
      real_ += other.real_;
      imaginary_ += other.imaginary_;
      return *this;
   }
private:
   double real_;
   double imaginary_;
};

Complex operator+(const Complex& a, const Complex& b) {
   Complex tmp(a);
   tmp += b;
   return tmp;
}
```

Struct

- Чем же отличаются структуры от классов?

Парадигмы программирования

- Объектно-ориентированное программирование: определите, какой класс вам необходим; предоставьте полный набор операций для каждого класса; общность классов выразите явно с помощью наследования.

Наследование

class <производный класс> : <сп. доступа> <базовый класс>

Производный класс наследует функциональность базового класса в той степени, которую позволяют модификаторы доступа

Модификаторы доступа при наследовании

- private
- protected
- public

Конструкторы и деструкторы при наследовании

- Если у базового и у производного классов имеются конструкторы и деструкторы, то конструкторы выполняются в порядке наследования, а деструкторы — в обратном порядке

Конструкторы и деструкторы при наследовании

```
class Base {
public:
    Base(int i) : i_(i)
    {}
private:
    int i_;
};

class Derived : public Base {
public:
    Derived(int i) : Base(i)
    {}
};

int main() {
    Derived d(1);
    return 0;
}
```

Указатели на производные классы

```
#include <iostream>

class Base {
public:
   void print(){std::cout << "Base\n";}
};

class Derived : public Base {
public:
   void print(){std::cout << "Derived\n";}
};

int main() {
   Derived d;
   Base b = d;
   b.print();
   Base* bp = &d;
   bp->print();
   return 0;
}
```

Виртуальные функции

- Используются для поддержки динамического полиморфизма

- Объявляется в базовом классе и переопределятся в производном

- В базовом классе метод объявляется с ключевым словом virtual, в производном его указывать не требуется

Виртуальные функции

```
#include <iostream>

class Base {
public:
   virtual void print(){std::cout << "Base\n";}
};

class Derived : public Base {
public:
   void print(){std::cout << "Derived\n";}
};

int main() {
   Derived d;
   Base b = d;
   b.print();
   Base* bp = &d;
   bp->print();
   return 0;
}
```

Чистые виртуальные функции

```
#include <iostream>

class Base {
public:
   virtual void print() = 0;
};

class Derived : public Base {
public:
   void print(){std::cout << "Derived\n";}
};

int main() {
   Derived d;
   Base* bp = &d;
   bp->print();
   return 0;
}
```