Введение в ООП

Классы и структуры

Парадигма программирования

- **Парадигма программирования** - это совокупность идей и понятий, определяющих стиль написания компьютерных программ.
- Служит для упрощения разработки и поддержки кода программ.
- При проектировании и написании кода парадигма отвечает на вопрос "как?".
- Язык программирования может поддерживать сразу несколько парадигм.

Парадигма программирования: примеры

- **Императивное программирования** - программирование с описанием последовательности инструкций, ветвлений, безусловных переходов, иногда с вызовом подпрограмм.
  Языки: Asm, C/C++, Fortran, Basic, Pascal...
- **Структурное программирование** - программирование с использованием независимых логически законченных процедур/функций.
  Языки: C/C++, Fortran, Basic, Pascal, Java...
- **Объектно-ориентированное программирование** - парадигма основанная на представлении программы в виде совокупности классов, объектов и их взаимодействий.
  Языки: C++, Java, Python...

Основные принципы ООП

- **Абстракция** - выделение наиболее важных свойств объектов реального мира и их оформление в виде атрибутов класса.
- **Инкапсуляция** - объединение данных и методов для работы с ними в рамках одного объекта, возможно с ограничением доступа к деталям реализации (сокрытие данных).
- **Полиморфизм** - свойство системы, позволяющее использовать различные реализации в рамках одного интерфейса (один интерфейс - много реализаций).
- **Наследование** - свойство, позволяющее создавать новый тип данных на основе уже существующего с полным или частичным заимствованием функционала.

Классы и объекты

Класс - описание некоторого концепта из предметной области в виде набора полей и методов для работы с ними (описание нового типа данных).

```
struct Node {
  int value;   // field
  Node* next;  // field
};

struct StackList {
  Node* head = nullptr;  // field

  void Push(int value);  // method declaration
  void Pop();            // method declaration
  int Top();             // method declaration
  void Clear();          // method declaration
};
```

Классы и объекты

Объект - экземпляр класса.

```
StackList stack;  // create object
std::cout << stack.head << '\n';
stack.Push(1);
std::cout << stack.Top() << '\n';
```

```
0
1
```

Стек на массиве: процедурный подход

```
void Push(int value, int* buffer, size_t& size) { buffer[size++] = value; }

void Pop(int* buffer, size_t& size) { --size; }

int Top(int* buffer, size_t size) { return buffer[size - 1]; }

int main() {
  auto stack_buffer = new int[100];  // допустим, переполнения не будет
  size_t stack_size = 0;

  Push(1, stack_buffer, stack_size);
  Push(2, stack_buffer, stack_size);
  Pop(stack_buffer, stack_size);

  std::cout << Top(stack_buffer, stack_size) << '\n';  // 1

  delete[] stack_buffer;
  return 0;
}
```

Стек на массиве: процедурный подход

Нравится ли вам этот код?

-
-
-
-
-

Что такое стек?

-
-
-
-
-

Стек на массиве: процедурный подход

Нравится ли вам этот код?

- Постоянно таскать указатель?
- Постоянно таскать размер?
- Самостоятельно работать с памятью?
- Инициализировать размер?
- В любой момент иметь несанкционированный доступ к данным?

Что такое стек?

- Указатель на массив?
- Сам массив?
- Его размер?
- Набор операций `Pop`, `Push`, `Top`?
- А он существует?

Стек на массиве: ООП подход

```
struct Stack {
  int* buffer;
  size_t size;

  void Init() { buffer = new int[100]; size = 0; }
  void Finalize() { delete[] buf; }
  void Push(int value) { buffer[size++] = value; }
  void Pop() { --size; }
  int Top() { return buffer[size - 1]; }
};

int main() {
  Stack stack;
  stack.Init();
  stack.Push(1);
  std::cout << stack.Top() << '\n';
  stack.Pop();
  stack.Finalize();
}
```

Стек на массиве: ООП подход

Нравится ли вам этот код?

- ~~Постоянно таскать указатель?~~
- ~~Постоянно таскать размер?~~
- ~~Самостоятельно работать с памятью?~~
- ~~Инициализировать размер?~~
- В любой момент иметь несанкционированный доступ к данным?

Что такое стек?

- ~~Указатель на массив?~~
- ~~Сам массив?~~
- ~~Его размер?~~
- Набор операций `Pop`, `Push`, `Top`!

Модификаторы доступа

Модификаторы доступа: синтаксис

```
struct Stack {
// ...
 private:
// ...
 public:
// ...
};
```

Модификаторы доступа

- **public** - *любой* внешний по отношению к классу код имеет доступ к полям и методам.
- **private** - доступ имеют *только* поля и методы *самого класса*, а также дружественные функции и классы (обсудим позже).
- **protected** - то же, что и `private`, но дополнительно доступ получают и наследники класса (обсудим позже в курсе).
- Располагать модификаторы доступа внутри класса можно в любом порядке и в любом количестве.
- Модификатор действует с точки объявления до следующего модификатора (либо до конца класса).

Модификаторы доступа: пример

```
struct S {
  int x;

 private:
  int y;
  int z;

  void f() {
      x = 0;  // ???
      y = 0;  // ???
  }

 public:
  void g() {
      x = 0;  // ???
      z = 1;  // ???
  }
};
```

Модификаторы доступа: пример

```
struct S {
  int x;

 private:
  int y;
  int z;

  void f() {
      x = 0;  // Ok
      y = 0;  // Ok
  }

 public:
  void g() {
      x = 0;  // Ok
      z = 1;  // Ok
  }
};
```

Модификаторы доступа: пример

```
struct S {
  int x;

  void g();
  void h(int);

 private:
  int y;

  void f();
  void h(double);
};
```

```
int main() {
  S s;
  s.x = 0;   // ???
  s.y = 1;   // ???
  s.f();     // ???
  s.g();     // ???
  s.h(0);    // ???
  s.h(0.0);  // ???
}
```

Модификаторы доступа: пример

```
struct S {
  int x;

  void g();
  void h(int);

 private:
  int y;

  void f();
  void h(double);
};
```

```
int main() {
  S s;
  s.x = 0;    // Ok
  s.y = 1;    // CE
  s.f();      // CE
  s.g();      // Ok
  s.h(0);     // Ok
  s.h(0.0);   // CE
}
```

Стек на массиве: ООП подход

```
struct Stack {
  void Init();
  void Finalize();
  void Push(int value);
  void Pop();
  int Top();
  size_t Size();

 private:
  int* buffer;
  size_t size;
};

int main() {
  Stack stack;
  stack.size++;  // CE
}
```

```
error: 'size_t Stack::size' is private within this context
```

Ключевое слово `class`

- Для создания нового типа (класса) вместо `struct` можно использовать ключевое слово `class`.

```
class Stack {
    // ...
};
```

- Классы **полностью** эквивалентны структурам, но есть два нюанса:

1. В классах модификатор доступа по умолчанию - `private`, в структурах - `public`.
2. Классы наследуют по умолчанию приватным образом, структуры - публичным.

- Как правило, предпочитают использовать `class`. Структуры обычно пишут без методов и они состоят только из открытых полей базовых типов (POD-типы).

Определения вне классов

Определения вне классов

- Методы можно определять вне классов, при этом они все еще должны быть предварительно объявлены внутри тела класса

```
struct S {
  int x = 0;

  void f();
};

void S::f() { std::cout << x; }
```

- При определении методов внутри класса они автоматически становятся inline

```
struct S {
  int x = 0;

  void f() { std::cout << x; }  // inline
}
```

Константные поля

Константные поля

- Как и любую переменную поле можно объявить константным.

```
struct S {
  int x;
  const int y = 9;

  void f();
};
```

- Такие поля нужно инициализировать сразу в теле структуры или класса.
- В течение жизни они не меняют своего значения (и ничего с этим не поделать).

Константные поля: пример

```
int n = 0;

struct S {
  int x;
  const int id = n++;

  void f() {
    x += 1;   // ???
    id += 1;  // ???
  }
};
```

```
int main() {
  S a;        // ???
  S b;        // ???
  a.x = 0;    // ???
  b.x = 1;    // ???
  a.id = 0;   // ???
  b.id = 1;   // ???

  return 0;
}
```

Константные поля: пример

```
int n = 0;

struct S {
  int x;
  const int id = n++;

  void f() {
    x += 1;   // Ok
    id += 1;  // CE
  }
};
```

```
int main() {
  S a;        // a.id == 0
  S b;        // b.id == 1
  a.x = 0;    // Ok
  b.x = 1;    // Ok
  a.id = 0;   // CE
  b.id = 1;   // CE

  return 0;
}
```

Константные методы

Константные объекты

- Рассмотрим класс стека. Меняет ли метод `Size()` его состояние?

```
int Stack::Size() {
  return size;
}
```

- Но...

```
Stack s;
// ...
const Stack& cref = s;
cref.Size();  // CE
```

```
error: passing 'const Stack' as 'this' argument discards qualifiers
```

- Получается, у константных объектов нельзя вызывать методов?..

Константные методы

- Все не так плохо - компилятору нужно сообщить, что метод ничего не меняет:

```
int Stack::Size() const {  // <-- !
  return size;
}

cref.Size();  // Ok
```

- Это не только подсказка для компилятора, но и указание, что в методе нельзя изменять поля!

```
int Stack::Size() const {
  return ++size;  // CE
}
```

```
error: increment of member 'Stack::size' in read-only object
```

Константные методы

- Естественно, в константных методах можно вызывать только константные методы (неконстантные могут изменить поля).

```
int Stack::Size() const {
    Pop();  // CE: вызов неконстантного метода в константном
    return size;
}

int Stack::Top() const {
    return buffer[Size() - 1];  // Ok: вызов константного метода
}
```

Константные методы

- Константность - часть сигнатуры, поэтому по константности можно делать перегрузку.

```
int Stack::Top() const {  // вызывается для константных стеков
  return buffer[size - 1];
}

int& Stack::Top() {       // вызывается для неконстантных стеков
  return buffer[size - 1];
}
```

```
Stack a;
const Stack& cref = a;
// ...
a.Top() = 1;  // неконстантный стек теперь позволяет изменять вершину
cref.Top() = 1;  // CE
```

Статические поля и методы

Статические поля и методы

- С помощью ключевого слова `static` некоторые поля и методы можно сделать статическими.

```
struct S {
 int x;
 static int y;

 void f();
 static void g();
};
```

- Такие поля и методы принадлежат не конкретному объекту, а классу в целом.
- К ним можно обращаться не только через объект класса, но и через имя класса с помощью операции `::` (`S::y` или `S::g()`).
- Статические методы могут работать только со статическими полями (в противном случае непонятно, какому объекту принадлежит поле)

Статические поля и методы: пример

```
struct S {
  int x;
  static int y;

  void f() {
    x += 1;  // ???
    y += 1;  // ???
  }

  static void g() {
    x += 1;  // ???
    y += 1;  // ???
  }
};
```

Статические поля и методы: пример

```
struct S {
  int x;
  static int y;

  void f() {
    += 1;  // Ok
    y += 1;  // Ok
  }

  static void g() {
    x += 1;  // CE
    y += 1;  // Ok
  }
};
```

Статические поля и методы: пример

```
struct S {
  int x;
  static int y;

  void f() {
    x += 1;
    y += 1;
  }

  static void g() {
    y += 1;
  }
};
```

```
int main() {
  S a, b;
  a.x = 0; b.x = 1;  // a.x = ???, b.x = ???
  a.y = 0; b.y = 1;  // a.y = ???, b.y = ???
  S::x = 10;         // a.x = ???, b.x = ???
  S::y = 11;         // a.y = ???, b.y = ???
  a.f(); b.f();      // a.x = ???, b.x = ???
  a.g(); b.g();      // a.y = ???, b.y = ???
  S::f();            // a.x = ???, b.x = ???
  S::g();            // a.y = ???, b.y = ???
  return 0;
}
```

Статические поля и методы: пример

```
struct S {
  int x;
  static int y;

  void f() {
    x += 1;
    y += 1;
  }

  static void g() {
    y += 1;
  }
};
```

```
int main() {
  S a, b;
  a.x = 0; b.x = 1;  // a.x = 0  , b.x = 1
  a.y = 0; b.y = 1;  // a.y = 1  , b.y = 1
  S::x = 10;         // CE
  S::y = 11;         // a.y = 11 , b.y = 11
  a.f(); b.f();      // a.x = 1  , b.x = 2
  a.g(); b.g();      // a.y = 15  , b.y = 15
  S::f();            // CE
  S::g();            // a.y = 16  , b.y = 16
  return 0;
}
```

Замечание о статических полях

- Начальное значение статического поля должно быть задано явно.

```
struct S {
  static int x;  // Linker error
};
```

```
error: ld returned 1 exit status
```

- Но его нельзя задавать внутри класса (нарушение ODR).

```
struct S {
  static int x = 0;  // Compilation error
};
```

```
error: ISO C++ forbids in-class initialization of non-const static member 'S::x'
```

Замечание о статических полях

- Исключением (как гласит сообщение об ошибке) являются константные целочисленные статические поля.

```
struct S {
  const static int x = 0; // Ok
};
```

- В общем случае инициализировать статическое поле нужно вне класса и ровно в одном `.cpp` файле.

```
struct S {
  static int x;
};

int S::x = 0;  // Ok
```

Константные статические методы

- Их нет
- И еще, *константные методы не могут изменять константные поля*!

Ключевое слово `this`

Ключевое слово `this`

Значение ключевого слова `this` - адрес текущего объекта.

```
struct S {
  int x;

  void SetX(int value) {
    this->x = value;  // <=> x = value;
  }

  int GetX() {
    return this->x;  // <=> return x;
  }

  S* GetAddress() {
    return this;
  }
};
```

```
S s;
&s == s.GetAddress();  // true
```

Ключевое слово `this`

Обращение к полям и методам неявно происходит через `this`.

`this` имеет тип "указатель на класс", либо "указатель на константный класс", если метод константный.

```
struct S {
  int x;

  void SetX(int value) {
    x = value;  // эквивалентно this->x = value
  }

  void SetX(int value) const {
    x = value;  // CE, так как type(this) == const S*
  }
};
```

Стек на массиве: итог после лекции

```
class Stack {
  int* buffer;
  size_t size;
  static const size_t capacity = 100;

public:
  void Init() { buffer = new int[capacity]; size = 0; }
  void Finalize() { delete[] buffer; }
  void Push(int value) { buffer[size++] = value; }
  void Pop() { if (size > 0) --size; }
  int& Top() { return buffer[size - 1]; }

  int Top() const { return buffer[size - 1]; }
  size_t Size() const { return size; }
  bool Empty() const { return size == 0; }

  static size_t Capacity() { return capacity; }
};
```

Резюме

- ООП - подход к разработке программ, который позволяет естественным образом описывать вычисления в виде набора объектов и их взаимодействий.
- В C++ за ООП в первую очередь отвечают классы и структуры.
- Модификаторы доступа позволяют осуществлять сокрытие деталей реализации.
- Статические поля и методы отвечают за свойства класса в целом, обычные поля и методы - за свойства конкретных объектов.
- Константные методы позволяют работать с объектами, не нарушая их логическую константность.

Логическая и физическая константность

if (remaining_time <= 0) goto seminar;

Логическая и физическая константность

- Объект **логически константен**, если с точки зрения пользователя объект не меняет своего состояния (нельзя отличить объект до и после операции).
- Объект **физически константен**, если его внутреннее представление никак не меняется (не изменилось ни одного бита внутри объекта).
- Пример: для отладки объект логирует информацию о действиях над собой (считает число вызовов метода). С физической точки зрения объект не константа (обновляется счетчик), с логической - константа (пользователь не наблюдает этих изменений).

Логическая и физическая константность

```
class C {
  size_t counter = 0;

public:
  int GetZero() {
    ++counter;
    return 0;
  }
};

const C c;
c.GetZero();
```

- Вопрос: может ли `GetZero()` быть константным?
- Ответ: в таком виде - нет. Он нарушает физическую константность.
- Но можно подсказать компилятору, что логическая константность не нарушается (пользователь никогда не взаимодействует с `counter`).

Ключевое слово `mutable`

- Чтобы сообщить компилятору, что изменение данного поля не влияет на логическую константность объекта, его можно пометить ключевым словом `mutable`.
- `mutable` поля можно изменять в константных методах.

```
class C {
  mutable int counter = 0;

public:
  int GetZero() const {
    ++counter;
    return 0;
  }
};

const C c;
c.GetZero();  // Ok
```

Ключевое слово `mutable`: пример

```
struct S {
    int x = -1;
    const int y = -1;
    mutable int z = -1;
    void f() {
        x = 0;  // ???
        y = 1;  // ???
        z = 2;  // ???
    }
    void g() {}
    void g() const {
        x = 0;  // ???
        y = 1;  // ???
        z = 2;  // ???
    }
};
```

```
int main() {
    const S sc;
    sc.x = 0;  // ???
    sc.y = 1;  // ???
    sc.z = 2;  // ???
    sc.f();    // ???
    sc.g();    // ???

    S s;
    s.f();  // ???
    s.g();  // ???
    return 0;
}
```

Ключевое слово `mutable`: пример

```
struct S {
    int x = -1;
    const int y = -1;
    mutable int z = -1;
    void f() {
        x = 0;  // Ok
        y = 1;  // CE
        z = 2;  // Ok
    }
    void g() {}
    void g() const {
        x = 0;  // CE
        y = 1;  // CE
        z = 2;  // Ok
    }
};
```

```
int main() {
    const S sc;
    sc.x = 0;  // CE
    sc.y = 1;  // CE
    sc.z = 2;  // Ok
    sc.f();    // CE
    sc.g();    // Ok: g() const

    S s;
    s.f();  // Ok
    s.g();  // Ok: g()
    return 0;
}
```