Разбор шаблонов написания классов C++

Далее везде используются обозначения

```
typedef int Type;
typedef double SomeOutputType;
```

Написание простого класса

1. Определить реализуемый класс (выделить из постановки задачи самостоятельный объект, требующий создание класса).

```
class NameClass {

};
```

2. Определить набор полей класса

Определение полей класс можно сделать по постановке задачи или при помощи выделения «главных» характеристик объекта. По умолчанию поля класса private (доступ к ним возможен только внутри класса).

```
class NameClass {
  // поля
  Type example;
};
```

3. Реализовать конструкторы, деструктор

   По умолчанию 3 вида конструктора, может быть больше или меньше (выбирать по постановке задачи). **Конструктор копирования** вызывается в фоновом режиме (например, при передаче объекта в функцию), поэтому в случае более сложных классов данный конструктор **обязателен** к написанию.

   Деструктор необходим, например, если в конструкторах происходило динамическое выделение памяти. В простых случаях деструктор можно не реализовывать или указать его с пустой реализацией (см. ниже).

   Не забывайте указывать для методов модификатор доступа public, иначе пользователь вашего класса не сможет воспользоваться данными методами.

```
class NameClass {
  // поля
  Type example;
public:
  // конструкторы (по умолчанию, инициализации, копирования)
  NameClass() {};
  NameClass(Type _example /*<список параметров, соответствующий набору полей класса>*/)
{};
  NameClass(const NameClass&) {};

  // деструктор
  ~NameClass() {};
};
```

4. Обозначить набор необходимого функционала (в первую очередь вывод на экран для проверки работоспособности). Часть методов можно делать private, чтобы скрыть их от пользователя вашего класса, это могут быть, например, служебные методы).

Написание простого класса с полями разного типа

- Полей в классе может быть сколько угодно.
- В качестве полей класса могут выступать различные типы данных, массивы, структуры, другие классы и т.д.
- Типы могут повторяться.
- При использовании массивов в конструкторах под них должно происходить выделение памяти, а в деструкторах происходить очищение памяти.

Замечание. Обратите внимание на различия в работе с памятью в C и C++.

```
class ClassTemplate {
  int field1;
  double field2;
  std::string field3;
  // ...
  int* fieldN;
  int fieldN_size;

public:
  ClassTemplate() {
    field1 = 0;
    field2 = 0.0;
    field3 = "";
    // ...
    fieldN_size = 10;

    // fieldN = (int*)malloc(fieldN_size * sizeof(int))       // выделение памяти в языке C
    fieldN = new int[fieldN_size];                            // выделение памяти в языке
C++
    for (int i = 0; i < fieldN_size; i++)
      fieldN[i] = 0;
  };

  ClassTemplate(int _field1, double _field2, std::string _field3, /*...*/ int* _fieldN, int
_fieldN_size) {
    field1 = _field1;
    field2 = _field2;
    field3 = _field3;
    // ...
    fieldN_size = _fieldN_size;
    fieldN = new int[fieldN_size];
    for (int i = 0; i < fieldN_size; i++)
      fieldN[i] = _fieldN[i];
  };

  ClassTemplate(const ClassTemplate& current) {
    field1 = current.field1;
    field2 = current.field2;
    field3 = current.field3;
    // ...
    fieldN_size = current.fieldN_size;
    fieldN = new int[fieldN_size];
    for (int i = 0; i < fieldN_size; i++)
      fieldN[i] = current.fieldN[i];
  };
```

```
  ~ClassTemplate() {
    // free(fieldN);     // C
    delete[] fieldN;     // C++
  };

  void print() {
    std::cout << "Field 1: " << field1 << std::endl
      << "Field 2: " << field2 << std::endl
      << "Field 3: " << field3 << std::endl
      << "// ..." << std::endl
      << "Field N: [ ";
    for (int i = 0; i < fieldN_size; i++) {
      std::cout << fieldN[i] << " ";
    }
    std::cout << "]" << std::endl << std::endl;
  };
};
```

Написание классов с наследованием

Иногда возникает необходимость организовывать систему классов с наследованием. Наследование полезно, поскольку оно позволяет структурировать и повторно использовать код, что, в свою очередь, может значительно ускорить процесс разработки. Несмотря на это, наследование следует использовать с осторожностью, поскольку большинство изменений в суперклассе (родительском классе) затронут все подклассы, что может привести к непредвиденным последствиям.

Пример. Рассмотрим образовательный сектор. Пусть есть школьник, студент и преподаватель. Система классов в этом случае выглядит, например, следующим образом:

- Человек (ФИО, дата рождения, паспорт, адрес, СНИЛС) -> Учащийся (предметы, списки оценок по каждому предмету, итоговые оценки) –> Школьник (списки посещаемости, школьные взносы (баланс)),
- Человек (ФИО, дата рождения, паспорт, адрес, СНИЛС) -> Учащийся (предметы, списки оценок по каждому предмету, итоговые оценки) –> Студент (номер зачётки, оценки за экзамены, средний балл, стипендия),
- Человек (ФИО, дата рождения, паспорт, адрес, СНИЛС) -> Преподаватель (предмет, зарплата).

Ввиду количества общих полей и правил работы с ними у класса Школьник и Студент эти поля были вынесены в промежуточный класс Учащийся.

Рассмотрим общую схему наследования.

```
class GrandParent {
protected:
  Type field1;

public:
  GrandParent() {
    field1 = 0.0;
  };

  GrandParent(Type val1) {
    field1 = val1;
  };

  GrandParent(const GrandParent& curr) {
    field1 = curr.field1;
  };
```

```
  void print() {
    std::cout << "Output field 1: " << field1 << std::endl;
  };

private:
  SomeOutputType secretMethod(/* параметры */) { /* реализация */ };
};

class Parent : public GrandParent {
protected:
  Type* field2;
  int size;
public:
  Parent() : GrandParent() {
    size = 10;
    field2 = new Type[size];
    for (int i = 0; i < size; i++) {
      field2[i] = 0.0;
    }
  };

  Parent(Type val1, Type* vals, int _size) : GrandParent(val1) {
    size = _size;
    field2 = new Type[size];
    for (int i = 0; i < size; i++) {
      field2[i] = vals[i];
    }
  };

  Parent(const Parent& curr) : GrandParent(curr.field1) {
    size = curr.size;
    field2 = new Type[size];
    for (int i = 0; i < size; i++) {
      field2[i] = curr.field2[i];
    }
  };

  void print() {
    GrandParent::print();
    std::cout << "Output field 2: [ ";
    for (int i = 0; i < size; i++) {
      std::cout << field2[i] << " " ;
    }
    std::cout << "]" << std::endl;
  };
};

class Child : public Parent {
  Type field3;

public:
  Child() : Parent() {
    size = 10;
    field2 = new Type[size];
    for (int i = 0; i < size; i++) {
      field2[i] = 0.0;
    }
  };

  Child(Type val1, Type* vals, int _size, Type val3) : Parent(val1, vals, _size) {
```

```
    field3 = val3;
  };

  Child(const Child& curr) : Parent(curr.field1, curr.field2, curr.size) {
    field3 = curr.field3;
  };

  void print() {
    Parent::print();
    std::cout << "Output field 3: " << field3 << std::endl;
  };
};
```

Одним из основных преимуществ public-наследования является то, что указатель на классы-наследники может быть неявно преобразован в указатель на базовый класс, то есть:

```
A* a = new B();
```