Лекция 6. Классы

Типы классов: классы значения

- Открытые деструктор, копирующий конструктор; присваивание с семантикой значения
- Нет виртуальных функций
- Используются как конкретные классы, не как базовые
- Чаще размещаются на стеке или являются полем другого класса
- Примеры: `std::vector<T>, std::complex`

Типы классов: базовые классы

- Деструктор:
  - открытый виртуальный (интерфейс), либо
  - защищенный невиртуальный (реализация)
- Закрытые конструктор копирования и оператор присваивания
- Интерфейс определяется виртульными функциями (либо NVI)
- Чаще создаются в куче и держатся с помощью умных указателей.

Типы классов

- Классы свойства (traits):
  - нет состояния, виртуальных функций,
  - только `typedef` и статические функции,
  - обычно не создают объекты этого типа,
  - пример: `std::char_traits<T>`.
- Классы-стратегии
- Классы-исключения

Определение класса

```
1.  class array
2.  {
3.      //...
4.  };
```

- Определение класса является объявлением - может встречаться в разных единицах трансляции (с учетом ODR).
- Можно использовать как struct, так и class. Разница в начальном модификаторе доступа.
- Точка с запятой в конце объявления обязательна!

Функции-члены (member-functions)

```
1.  struct array
2.  {
3.      double* data_;
4.      size_t  size_;
5.  };
6.
7.  void push_back(array& arr, double value)
8.  {delete arr.data_; /*...*/}
9.
10. ////////////////////////////////////////////////////////
11.
12. // array.h
13. struct array
14. {
15.     void push_back(double value);
16.     void resize   (size_t new_size);
17.     //...
18. };
19.
20. // array.cpp – compilation optimization
21. void array::push_back(double value)
22. {
23.     delete data_;
24.     //...
25. }
```

Управление доступом

```
1.  struct array
2.  {
3.      void resize   (size_t);
4.      void push_back(double);
5.
6.  private:
7.      void fill_with(double value);
8.
9.  private:
10.     double* data_;
11.     size_t  size_;
12. };
```

- `struct` – по-умолчанию `public`, `class` – `private`
- К `private` секции имеют доступ только member-функции или друзья.
- Секции могут повторяться в любом порядке.
- Начинайте с `public` секций – интерфейса.

Определение функций в классе

```
1.  // array.h
2.  struct array
3.  {
4.      void resize   (size_t);
5.      void push_back(double)
6.      {
7.          delete data_;
8.          //...
9.      }
10.
11. //...
12.     double* data_;
13. };
14.
15. inline void array::resize(size_t)
16. {
17.     //...
18. }
```

- Для повышения читаемости лучше в объявлении класса оставлять только объявления функций.
- Не смешивайте способы объявления – либо все в заголовочном файле, либо в cpp-шнике.

4 главные функции

- Функции:
  - Конструктор по умолчанию (без параметров)
  - Конструктор копирования
  - Оператор присваивания
  - Деструктор
- Все они создаются автоматически, но могут быть переопределены пользователем.
- Если объявлен хоть один конструктор с параметрами, конструктор по умолчанию не создается автоматически.

Конструкторы

```
 1. struct array_wo_ctor
 2. {
 3.     void init();
 4.     void init(size_t size, double def = 0.);
 5. };
 6.
 7. struct array
 8. {
 9.     array();
10.     array(size_t size, double def = 0.);
11. };
12.
13. void func()
14. {
15.     array_wo_ctor a;
16.     a.init(10, 5.);
17.
18.     array  arr(10, 5.);
19.     array* ptr = new array(10, 5.);
20. }
```

- Конструктор нельзя забыть вызвать или вызвать дважды

Список инициализации

```
1.  array::array(size_t num, double def)
2.      : data_(new double[num]) // exception?(*)
3.      , size_(num)
4.  {
5.      fill_with(def);
6.  }
```

- Поля класса инициализируются в порядке объявления в классе (т.о. во всех конструкторах в одинаковом порядке)
- Поля как объекты создаются до входа в тело конструктора.

Обязательная инициализация полей

```
 1. struct foobar
 2. {
 3.     foobar(int& value)
 4.         : pi_ (3.14)
 5.         , ref_(value)
 6.         , ofs_("~/some.txt")
 7.     {
 8.         name_ = "some";
 9.     }
10.
11. private:
12.     string      name_;
13.     int const   pi_ ;
14.     int&        ref_;
15.     ofstream    ofs_;
16. };
```

- Обязательны для инициализации:
  - константы,
  - ссылки,
  - объекты без конструктора по умолчанию.

Деструктор

```
1.  array::~array()
2.  {
3.      delete data_;
4.  }
```

- Освобождает выделенные объектом ресурсы
- Вызывается
  - при выходе локальной переменной из своей области действия
  - при вызове оператора delete
- Деструкторы полей, если они есть, вызываются после тела деструктора
- (*) Не должен бросать исключений

Конструктор копирования
Оператор присваивания

```
1.  array::array(array const& other)
2.      : data_(new double[other.size_])
3.      , size_(other.size_)
4.  {
5.      std::copy(data_, data_ + size_, other.data_);
6.  }
7.
8.  array& array::operator=(array const& other)
9.  {
10.     if (*other != this) // why?
11.     {
12.         delete data_;
13.         data_ = new double[other.size_];
14.         size_ = other.size_;
15.         std::copy(data_, data_ + size_, other.data_);
16.     }
17.
18.     return *this;
19. }
```

Функция swap

```
1.  // swap is a friend function
2.  void swap(array& lhs, array& rhs)
3.  {
4.      using std::swap;
5.      swap(lhs.data_, rhs.data_);
6.      swap(lhs.size_, rhs.size_);
7.  }
8.
9.  // or exists member-function array::swap
10. void swap(array& lhs, array& rhs)
11. {
12.     lhs.swap(rhs);
13. }
```

- (*) Не должна бросать исключений
- Для всех встроенных типов и STL-типов определена `std::swap`

swap trick

```
1.  // before:
2.  array& array::operator=(array const& other)
3.  {
4.      if (*other != this)
5.      {
6.          delete data_;
7.          data_ = new double[other.size_];
8.          size_ = other.size_;
9.          std::copy(data_, data_ + size_, other.data_);
10.     }
11.
12.     return *this;
13. }
14.
15. // now:
16. array& array::operator=(array other)
17. {
18.     swap(*this, other); // better exception safety?(*)
19.     return *this;
20. }
```

RAII (resource acquisition is initialization)

```
1.  struct out_bin_file
2.  //  : boost::noncopyable
3.  {
4.      out_bin_file(const char* name)
5.          : file_(fopen(name, "w")){}
6.
7.     ~out_bin_file(){ fclose(file_); }
8.
9.  private:
10.     out_bin_file(out_bin_file const&);
11.     out_bin_file& operator=(out_bin_file const&);
12.
13. private:
14.     FILE* file_;
15. };
16.
17. void foo()
18. {
19.     // this file will be always closed
20.     // before returning from these function
21.     out_bin_file file("~/some.txt");
22.
23.     if (...)
24.         return;
25.
26.     throw std::runtime_error("oops...");
27. }
```

Неудачная инициализация*

```
1.
2.  struct out_bin_file
3.      : boost::noncopyable
4.  {
5.      struct file_failure
6.          : std::runtime_error
7.      {
8.          file_failure(string problem)
9.              : std::runtime_error(problem)
10.         {
11.         }
12.     };
13.
14.     out_bin_file(const char* name)
15.         : file_(fopen(name, "w"))
16.     {
17.         if (file_ == nullptr)
18.             throw file_failure("cannot open file " + name);
19.     }
20.
21. private:
22.     FILE* file_;
23. };
```

Преобразование типов

- Конструктор от одного параметра задает неявное преобразование типов. Иногда это удобно (`std::string`), порой – нет.

```
 1.  struct array
 2.  {
 3.      explicit array(size_t num, double def = 0.);
 4.      //...
 5.  };
 6.
 7.  void foo(array const& arr) { /*...*/}
 8.
 9.  void bar()
10.  {
11.      foo(5); // in case of no ‘explicit’
12.  }
```

Статические поля

```
1.  struct socket
2.  {
3.      socket()
4.      {
5.          if (!lib_init)
6.              lib_init = init_socket_lib();
7.      }
8.
9.      static bool lib_init;
10. };
11. //...
12. bool socket::lib_init = false;
13.
14. ////////////////////////////////////////
15. struct singleton
16. {
17.     static singleton* create(/*...*/)
18.     {
19.         static singleton s(/*...*/);
20.         return &s;
21.     }
22.
23. private:
24.     singleton(/*...*/){/*...*/}
25. };
26. //...
27. auto s = singleton::create();
```

`const` member-functions

```
 1. struct array
 2. {
 3.     double& at(size_t i)       {return data_[i];}
 4.     double  at(size_t i) const {return data_[i];}
 5.
 6.     // this type: array const* const
 7.     double  back() const       {return at(size_ - 1);}
 8.
 9. private:
10.     double* data_;
11.     size_t  size_;
12. };
13.
14. void foo(array const& x)
15. {
16.     double value = x.back();
17. }
```

Логическое постоянство

```
1.  struct matrix
2.  {
3.      matrix const& inverted () const
4.      {
5.          return inverted_
6.              ? *inverted_
7.              : (inverted_ = calc_inv());
8.      }
9.
10.     //..
11.     mutable matrix* inverted_;
12. };
```

- Используется чаще всего для различных видов кэша

Время жизни объектов

```
1.  string* foo(bool cond)
2.  {
3.      mapped_file   file("~/data.bin", read_write);
4.      mapped_region reg (file, 0, 0x10000);
5.      // ..
6.
7.      // what if bad_alloc was thrown?
8.      string* strs = new string[15];
9.      return new string("text");
10. }
```

- Локальные объекты удаляются в порядке обратным их созданию
- Объекты в динамической памяти удаляются только после вызова `delete`

Временные объекты

- Если временный объект не инициализирует
  - именованный объект или
  - константную ссылку,

  он удаляется к концу полного выражения.

```
1.  void foo()
2.  {
3.      string h = "Hello, ";
4.      string w = "World!";
5.
6.      const char* str = (h + w).c_str(); // oops!
7.      //str is undefined here
8.  }
```

union

```
1.  struct addr_v4
2.  {
3.      uint8_t b1;
4.      uint8_t b2;
5.      uint8_t b3;
6.      uint8_t b4;
7.  };
8.
9.  union addr
10. {
11.     addr_v4  bytes;
12.     uint32_t value;
13. };
14.
15. void foo()
16. {
17.     addr a;
18.     a.bytes.b1 = 0x54;
19. }
```

- Все поля расположены по одному адресу
- Не может иметь конструкторов/деструкторов

Вложенные типы

```
1.
2.  struct queue
3.  {
4.      typedef int T;
5.
6.  public:
7.      queue();
8.     ~queue();
9.
10.     void push(T);
11.     T    top () const;
12.     void pop ();
13.
14. private:
15.     struct node
16.     {
17.         node* next;
18.         T     data;
19.     };
20.
21. private:
22.     node* head;
23. };
```

- Вложенные типы становятся друзьями внешнему типу

Пример array

```
1.  struct array
2.  {
3.      array();
4.      explicit array(size_t num, double def. = 0);
5.
6.      array(array const& other);
7.      array& operator=(array const& other);
8.
9.      double& at(size_t i);
10.     double  at(size_t i) const;
11.
12.     size_t size () const;
13.     bool   empty() const;
14.
15.     void resize   (size_t);
16.     void push_back(double);
17.
18. private:
19.     void fill_with(double value);
20.
21. private:
22.     double* data_;
23.     size_t  size_;
24. };
```

Рекомендации

- Следуйте принципу «Один объект – одна задача»
- Лучше маленький класс, чем монолитный
  - легче понять и проще использовать
  - монолитные классы часто используют малосвязанные сущности. Но при изменении одной из них, приходится заботится обо всем классе.
- Дополнительную функциональность лучше реализовать через внешние функции

Вопросы?