Объектно-ориентированное программирование в C++

Конструктор копирования

Перегрузка операторов

Копирование объектов

- Копирование данных осуществляется при инициализации, присваивании, передаче параметров и возврате из функции
- Для пользовательских типов (классов\структур) для корректного выполнения копирования внутренних элементов объектов необходимо переопределить *конструктор копирования* и *операцию присваивания*

```
Line line1("Hello world");
// конструктор копирования
Line line2 = line1;
// операция присваивания
line3 = line2;
// конструктор копирования
Show(line1);
```

- Компилятор автоматически создаёт конструктор копирования и операцию присваивания, реализуя «побитовое» копирование внутренних элементов

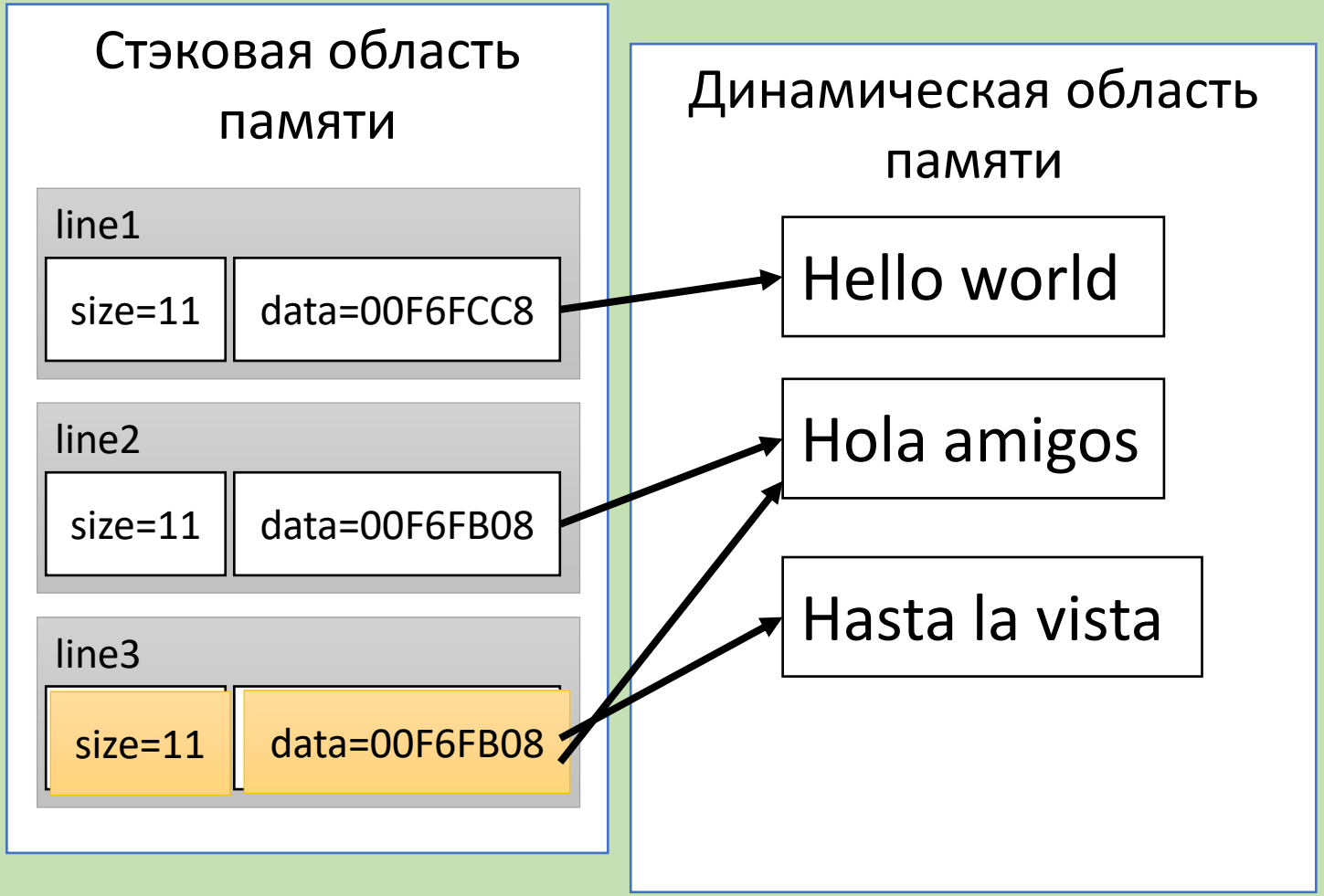
```
class Line {
    private:
        char* data;
        int size;
    public:
        Line(const char* s) : Line() {
            size = strlen(s);
            data = new char[size];
            for(int i=0; i<size; i++) data[i] = s[i];
        }
        Line() : data(nullptr), size(0) {}
        ~Line()  {  delete[] data;  }
        int getSize() const { return size; }
        const char* c_str();
        ..
};
```

Хранение объекта и в stack, и в heap

```
Line line1("Hello world");
Line line2("Hola amigos");
Line line3("Hasta la vista");


// побитовое копирование
line3 = line2;

// побитовое копирование
Show(line1);
```



Конструктор копирования

- Конструктор копирования описывает как создавать объект на основе существующего объекта
- Конструктор копирования принимает один аргумент – типа ссылки на объект данного типа; часто тип аргумента снабжают модификатором **const**

```
// Конструктор копирования
Line::Line(const Line& other)
{
   size = strlen(other.data);
   data = new char[size];
   for (int i = 0; i < size; i++)
        data[i] = other.data[i];
}
```

Операция присваивания: `line3 = line2;`

В операции присваивания важно:

- Устранить возможность утечки памяти (освобождение динам. памяти);
- Проверять на самоприсваивание;

  `line3 = line3;`
- Возвращать ссылку на текущий объект для возможности формирования цепочек с присваиванием;

  `line1 = line2 = line3;`

```
Line& operator= (Line& other)
{
        if (&other == this) return *this;
        delete[] data;
        size = other.size;
        data = new char[size];
        for (int i = 0; i < size; i++)
                data[i] = other.data[i];
         return *this;
}
```

Запрет на копирование

- Существует возможность «запретить» копирование объекта
- «Конструктор копий» и operator= помещаются в закрытую часть объявления класса (можно сделать их пустыми)

```
class Point {
  public :
    Point(int x, int y) : x(x), y(y) {}
  private:
    int x;
    int y;

    void operator=(Point& other) {}
    Point(Point& other) {}
};
```

```
Point p3 = p1 + p2;
p2 = p1;
```

Временные и локальные объекты

- Анонимные объекты

```
cout << Line("all that glitters is gold").c_str() << endl;
```

- Временные объекты порождаются компилятором автоматически

```
// создается временный объект конструктором преобразования
// Line::Line(const char* );
Show("Sometimes all of our thoughts are misgiven");
// создается временный объект для line1 + line2
cout << (line1 + line2 + line3).c_str();
```

- На временные или анонимные объекты нельзя ссылаться неконстантной ссылкой

```
// void Show(const Line& line) или void Show(Line line)
Show(line1 + line2);
```

Значения по умолчанию для аргументов функций

- Достаточно указать значения по умолчанию в объявлении функции

```
void f(int n = 5, const char* s = "no data", float f = 5.0f);
```

- Допустимы вызовы без указания таких аргументов:

```
f();
f(10);
f(15, "big data");
f(20, "", 3.14f);
```

- Аргументы с значениями по умолчанию должны быть «в конце списка аргументов»:

```
void f(int a, int b = 0, int c = 0);
```

- Значения по умолчанию можно использовать и для конструкторов

```
Line(const char* s="\0") : data(nullptr)   { ..}
```

Инициализация элементов класса

- Поля класса могут относится к «сложным» типам с «сложной» инициализацией. В этом случае необходимо использовать *выражения инициализации* в конструкторе типа.

```
class Point {
public :
    Point(int x, int y) : x(x), y(y) {}
private:
    int x;
    int y;
};
```

```
class Segment {
private:
    Point a;
    Point b;
public:
    Segment() : a(0, 0), b(0, 0) {}
};
```

Переопределение операций

```
Point p3 = p1 + p2;     // а что делать?
bool b = line1 > 0;
```

- C++ поддерживает возможность переопределения большинства операций (арифметических, логических, побитовых, сравнения и др.)
- Не поддерживается перегрузка операций:
  - «точка» (выборка элементов класса или структуры)
  - «тернарный оператор» (a ? b : c)
  - “::” (открытие области видимости)
- Специфические операции: ->, [ ], (), (int)
- Для переопределения операции используется «operator»
- Операция может быть элементом класса, *дружественной* функцией или внешней функцией (для некоторых операций разрешено переопределение только внутри пользовательского типа)

Переопределение операций

```
// 1. Внешняя функция
Point operator+(Point& p1, Point& p2)
{
	return Point(p1.getX() + p2.getX(), p1.getY() + p2.getY());
}
```

```
// 2. Дружественная функция
class Point {
  public :
	friend Point operator-(Point&, Point&);
}
Point operator-(Point& p1, Point& p2)
{
    return Point(p1.x - p2.x, p1.y - p2.y);
}
```

Переопределение операций внутри

Переопределять некоторые операции можно только через функции-элементы класса (преобразование типа, [ ], (), ->)

```
// 3. Метод класса
class Point {
  public :
        friend Point operator-(Point&, Point&);
        Point& operator-=(Point& other)
        {
            x -= other.x; y -= other.y;
            return *this;
        }
  ..
};
```

```
p1 +=10;
p2 -= p3;
```

Операция индексирования: char c = line1[0];

- Операция индексирования принимает один аргумент

```
// для класса Line
char operator[]  (int i) const { return (i < size) ? data[i] : 0; }
```

- Аргумент не обязательно должен быть целочисленного типа
- Можно возвращать «копию» элемента или ссылку

```
// для класса Point
int operator[](bool b) const { return b ? x : y; }
int& operator[](bool b)  { return b ? x : y; }
```

Инкремент: Point p2 = p1++;

```
// prefix increment
Point& operator++()
{
    ++x; ++y;
    return *this;
}
//postfix increment
Point operator++(int i)
{
    Point temp(*this);
    ++(*this);
    return temp;
}
```

Перегрузка операции <<, >> cout << p << endl;

- Для возможности вывода на консоль объекта можно перегрузить операцию <<

- Первый аргумент – поток вывода, поэтому функция не может принадлежать пользовательскому типу.
- Функция должна быть внешней (дружественной или «обычной»)

```
friend std::ostream& operator <<(std::ostream& os, const Point & p)
{
    os << p.x << ", " << p.y;
    return os;
}
```