Семинар #2: Инкапсуляция. Классные задачи.

Инкапсуляция

Инкапсуляция – это размещение в классе/структуре данных и функций, которые с ними работают. Структуру с методами можно называть классом. Переменные класса называются полями, а функции класса – методами.

Без инкапсуляции:

```cpp
#include <cmath>
#include <iostream>

struct Point
{
    float x, y;
};

float norm(const Point& p)
{
    return std::sqrt(p.x*p.x + p.y*p.y);
}

void normalize(Point& p)
{
    float pnorm = norm(p);
    p.x /= pnorm;
    p.y /= pnorm;
}

Point operator+(const Point& l,
                        const Point& r)
{
    Point result = {l.x + r.x, l.y + r.y};
    return result;
}

int main()
{
    Point p = {1, 2};
    normalize(p);
    std::cout << p.x << " "
              << p.y << std::endl;
}
```

С инкапсуляцией:

```cpp
#include <cmath>
#include <iostream>

struct Point
{
    float x, y;

    float norm() const
    {
        return stdsqrt(x*x + y*y);
    }

    void normalize()
    {
        float pnorm = norm();
        x /= pnorm;
        y /= pnorm;
    }

    Point operator+(const Point& r) const
    {
        Point result = {x + r.x, y + r.y};
        return result;
    }
};

int main()
{
        Point p = {1, 2};
        p.normalize();
        std::cout << p.x << " "
                  << p.y << std::endl;
}
```

Обратите внимание на следующие моменты:

- Методы имеют прямой доступ к полям `x` и `y`. Передавать саму структуру(или ссылку на неё) в методах не нужно.
- Вызов метода класса осуществляется с помощью оператора `.`(точка).
- Спецификатор `const` после объявления метода (например, `float norm() const`) означает, что этот метод не будет менять поля. Желательно указывать этот спецификатор для всех методов, которые не изменяют объект.

– При перегрузке бинарных операций объектом является левый аргумент, а параметром функции – правый. Т.е. `p + q` превратится в `p.operator+(q)`.

Модификаторы доступа `public` и `private`

Модификаторы доступа служат для ограничения доступа к полям и методам класса.

– `public` – поля и методы могут использоваться где угодно

– `private` – поля и методы могут использовать только методы этого класса и друзья (особые функции и классы, объявленные с использованием ключевого слова `friend`)

Конструкторы

Конструктор – это специальный метод, который вызывается автоматически при создании экземпляра класса.

```
struct Point {
private:
    float x, y;
public:
    // Конструктор:
    Point(float ax, float ay)
    {
        x = ax;
        y = ay;
    }
    // другие методы
};

int main()
{
    // Есть несколько разных синтаксов создания экземпляра класса с вызовом конструктора:
    Point a = Point(7, 3);
    Point b(7, 3);
    Point c = {7, 3};
    Point d {7, 3};
    // Все они делают одно и то же - создают переменную на стеке и вызывают конструктор
    // В современном C++ предпочтительным является способ d
}
```

Особым видом конструктора является конструктор копирования:

```
Point(const Point& p)
```

Он используется для создание нового экземпляра класса по уже имеющемуся экземпляру.
Задача 3: Сделайте задание в файле `4point_constructors.cpp`

Ключевое слово `this` и оператор присваивания

Ключевое слово `this` - это указатель на экземпляр класса, который можно использовать в методах этого класса. Оператор присваивания – это просто перегруженный оператор =. Оператор присваивания должен вернуть ссылку на текущий объект, то есть `*this`.

Нужно различать оператор присваивания и вызов конструктора:

```
Point a = Point(7, 3);  // Конструктор ( оператор присваивания не вызывается )
Point b = a;            // Конструктор копирования ( оператор присваивания не вызывается )
Point c;                // Конструктор по умолчанию
c = a;                  // Оператор присваивания
```

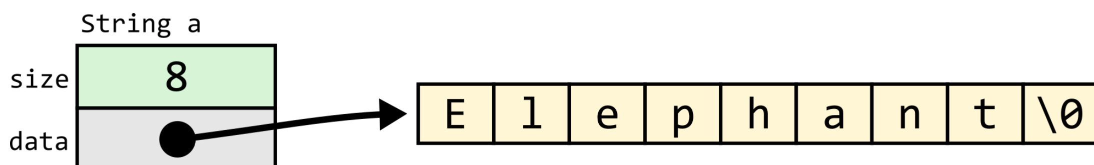
Оператор присваивания должен возвращать ссылку на левый аргумент.

Создаём свой класс строки

Строки в языке C представляют собой просто массивы с элементами типа char(однобайтовое число). Работать с такими строками не очень удобно. Нужно выделять и удалять необходимую память, следить за тем, чтобы строка помещалась в эту память на всём этапе выполнения программы, для работы со этими строками нужно использовать специальные функции из библиотеки string.h. Это всё может привести к ошибкам. В этом разделе мы создадим класс String – класс строки, с которым удобнее и безопаснее работать, чем со строками в стиле C. Заготовка класса выглядит так (Это далеко не самая эффективная реализация строки, более правильная реализация создаётся в примерах кода):

```
#include <cstdlib>

class String
{
private:

    size_t size;
    char* data;

public:

    String(const char* str)
    {
        size = 0;
        while (str[size])
            size++;

        data = (char*)malloc(sizeof(char) * (size + 1));

        for (int i = 0; str[i]; i++)
            data[i] = str[i];
        data[size] = '\0';
    }

    size_t getSize() const
    {
        return size;
    }

    const char* cStr() const
    {
        return data;
    }
};

int main()
{
    String a = "Elephant";
}
```

Схематично это можно представить следующим образом:



Деструктор

В коде выше выделяется память для массива `data`. Эта память выделяется при вызове конструктора (то есть при создании объекта). Однако она нигде не освобождается. Освободить её вручную мы не можем, так как поле `data` является приватным и это бы противоречило принципу сокрытия данных. Эта память должна освобождаться автоматически при удалении объекта.

```
~String()
{
    free(data);
}
```

Перегруженные операторы класса `String`

- **Оператор сложения:** `String operator+(const String& right) const`
  Этот оператор должен создавать новый экземпляр, задавать его поля (в частности придётся выделить память под строку-сумму) и возвращать этот экземпляр.

- **Оператор присваивания сложения:** `String& operator+=(const String& right)`
  Этот оператор не должен создавать новый экземпляр. Он должен изменять левый операнд (т. е. сам объект), и возвращать ссылку на этот объект (т. е. `*this`).

- **Оператор присваивания:** `String& operator=(const String& right)`
  Этот оператор не должен создавать новый экземпляр. Он должен изменять левый операнд (т. е. сам объект), так чтобы он стал идентичен правому. Если размеры строк не совпадают, то в данной реализации строки вам придётся удалить память левой строки и снова выделить память нужного размера. При этом нужно отдельно рассмотреть случай когда левый и правый операнд это один и тот же объект.

  ```
  String a {"Cat"};
  a = a;
  ```

  Конечно, в этом случае ничего удалять не нужно.

- **Оператор сравнения:** `bool operator==(const String& right) const`
  Этот оператор должен сравнивать строки (массивы `data`) и возвращать `true` или `false`.

- **Оператор индексации:** `char& operator[](unsigned int i)`
  Этот оператор должен возвращать ссылку на `i`-ый символ строки.

- **Индексация с проверкой на выход за границы:** `char& at(unsigned int i)`
  Этот метод должен проверять, что индекс `i` не выходит за границы диапазона и, если это так, возвращать ссылку на `i`-ый символ строки. Иначе, этот метод должен печатать сообщение об ошибке и завершать программу.

Раздельная компиляция класса

Методы можно вынести из определения класса следующим образом:

Определение методов в теле класса:

```
#include <cmath>
#include <iostream>


struct Point
{
    float x, y;

    float norm() const
    {
        return std::sqrt(x *x + y * y);
    }

    void normalize()
    {
        float pnorm = norm();
        x /= pnorm;
        y /= pnorm;
    }

    Point operator+(const Point& r) const
    {
        Point result = {x + r.x, y + r.y};
        return result;
    }
};


int main()
{
        Point p = {1, 2};
        p.normalize();
        std::cout << p.x << " "
              << p.y << std::endl;
}
```

Определение методов вне тела класса:

```
#include <cmath>
#include <iostream>

struct Point
{
    float x, y;

    float norm() const;
    void normalize();
    Point operator+(const Point& r) const;
};

float Point::norm() const
{
    return sqrt(x*x + y*y);
}

void Point::normalize()
{
    float pnorm = norm();
    x /= pnorm;
    y /= pnorm;
}

Point Point::operator+(const Point& r) const
{
    Point result = {x + r.x, y + r.y};
    return result;
}

int main()
{
        Point p = {1, 2};
        p.normalize();
        std::cout << p.x << " "
              << p.y << std::endl;
}
```

Теперь эти методы можно скомпилировать отдельно. Для этого их нужно вынести в отдельный компилируемый файл `point.cpp`, а определение класса в отдельный файл `point.h`. Так называемый заголовочный файл `point.h` нужен, так как определение класса нужно и файле `point.cpp` и в файле `main.cpp`. Для компиляции используем:

```
g++ main.cpp point.cpp
```

point.h

```
struct Point {
    float x, y;

    float norm() const;
    void normalize();
    Point operator+(const Point& r) const;
};
```

point.cpp

```
#include <cmath>
#include "point.h"

float Point::norm() const {
    return sqrt(x*x + y*y);
}

void Point::normalize() {
    float pnorm = norm();
    x /= pnorm;
    y /= pnorm;
}

Point Point::operator+(const Point& r) const{
    Point result = {x + r.x, y + r.y};
    return result;
}
```

main.cpp

```
#include <iostream>
#include "point.h"

int main() {
    Point p = {1, 2};
    p.normalize();
    std::cout << p.x << " " << p.y << std::endl;
}
```