

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:

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

struct Point {
    float x, y;
};

float norm(const Point& p) {
    return 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:

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

struct Point {
    float x, y;

    float norm() const {
        return 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;
}
```

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)`.

Задача 1: Сделайте задание в файлах `0point_without_encapsulation.cpp`, `1point_with_encapsulation.cpp` и `2point_separate_definition.cpp`

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**)

Задача 2: Сделайте задание в файле `3point_public_private.cpp`

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() {
    // Если 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 должны 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

Задача 4: Сделайте задание в файлах `6point_this.cpp`, `7point_copy_constructor.cpp` и `9point_default.cpp`.

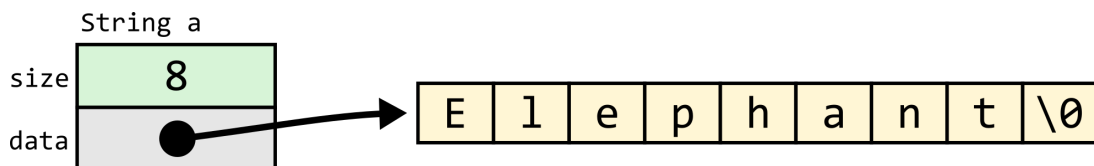
Создаём свой класс строки

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:

```
#include <cstdlib>
class String {
private:
    unsigned int size;
    char* data;
public:
    String(const char* str) {
        // Находим размер строки str (strlen не будем пользоваться)
        size = 0;
        while (str[size])
            size++;
        // Выделяем память
        data = (char*)malloc(sizeof(char) * (size + 1));
        // Копируем массив str в новый массив data
        for (int i = 0; str[i]; i++)
            data[i] = str[i];
        data[size] = '\0';
    }
    unsigned int get_size() const {
        return size;
    }
    const char* c_str() const {
        return data;
    }
};

int main() {
    String a = "Elephant"; // Создаём экземпляр класса String, используя конструктор
}
```

Схематично это можно представить следующим образом:



Задача 5: Сделайте задание в файлах `00string.cpp` и `01string_constructor.cpp`.

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 освободиться автоматически при удалении объекта.

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

Задача 6: Сделайте задание в файлах `02string_destructor.cpp` и `03string_destructor_order.cpp`.

Операторы new и delete

Использовать `malloc` и `free` для создания объектов с конструкторами и деструкторами в куче не получится.

`malloc` просто выделяет память и не вызывает никаких конструкторов

`free` просто освобождает память и не вызывает никаких деструкторов

Для выделения памяти в куче с вызовом конструктора используется оператор `new`:

```
String* p = new String("Hippo");
```

Для освобождения памяти в куче с вызовом деструктора используется оператор `delete`.

`new` = `malloc` + конструктор

`delete` = `free` + деструктор

Задача 7: Сделайте задание в файлах `04string_new_delete.cpp` и `05new_delete.cpp`.

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.

Задача 8: Сделайте задание в файлах `06string_operators.cpp` и `07string_operator[].cpp`.

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 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;  
}
```

Задачи:

1. Раздельно скомпилируйте класс строки. Для этого создайте файлы **String.h** и **String.cpp** и поместите в них объявление класса **String** и определение его методов соответственно.
2. В папке **2image** лежит программа, которая работает с классом **Image** – классом для работы с изображениями в формате **ppm**. Сделайте задание в **main.cpp**.