Стек

Стек

```
#ifndef _STACK_H_
#define _STACK_H_

const int MAX = 100;

class Stack
{
private:
    int s[MAX], top;

public:
    // конструктор
    Stack() { top = -1; }

    void push(int x);
    int pop();
    bool empty();

    // чтение без удаления
    int read();

    // полная очистка стека
    void clear();
};

#endif // _STACK_H_
```

Данный код должен располагаться в файле stack.h

+ все данные объединились в единую структуру и работают как одно целое

+ запрещен прямой доступ к внутренним данным

Стек

```
#include "stack.h"

void Stack::push(int x)
{
    top++;
    s[top] = x;
}

int Stack::pop()
{
    int x = s[top];
    top--;
    return x;
}

int Stack::read()
{
    return s[top];
}

bool Stack::empty()
{
    return top == -1;
}

void Stack::clear()
{
    top = -1;
}
```

Данный код должен располагаться в файле stack.cpp

Тест класса «Стек»

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

using namespace std;


int main()
{
    Stack stack;
    for (int i = 0; i < 10; i++)
    {
        stack.push(i);
    }
    while(!stack.empty())
    {
        cout << stack.pop() << endl;
    }
    return 0;
}
```

Данный код должен располагаться в файле test.cpp

Очередь

Очередь

queue.h

```
#ifndef _QUEUE_H_
#define _QUEUE_H_

const int MAX = 100;

class Queue
{
private:
    int s[MAX], head, tail;

public:
    // еще один способ инициализации полей
    Queue() : head(0), tail(0) {}

    void push(int x);
    int pop();
    int read();
    bool empty();
};

#endif
```

queue.cpp

```
#include "queue.h"

void Queue::push(int x)
{
    s[tail] = x;
    tail = (tail + 1) % MAX;
}

int Queue::pop()
{
    int x = s[head];
    head = (head + 1) % MAX;
    return x;
}

int Queue::read()
{
    return s[head];
}

bool Queue::empty()
{
    return tail == head;
}
```

Инкапсуляция

- Свойство объекта скрывать свою внутреннюю структуру от внешнего мира называется **инкапсуляцией**

Комплексные числа

Комплексные числа
Процедурный подход

complex_proc.h

```
#ifndef _COMPLEX_H_
#define _COMPLEX_H_

struct Complex
{
     double re, im;
};

// сложение
Complex add(Complex a, Complex b)
{
     Complex result;
     result.re = a.re + b.re;
     result.im = a.im + b.im;
     return result;
}
// вычитание
Complex sub(Complex a, Complex b)
{
     Complex result;
     result.re = a.re - b.re;
     result.im = a.im - b.im;
     return result;
}

#endif
```

test.cpp

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

using namespace std;


int main()
{
    Complex z1, z2, z3;
     z1.re = 1; z1.im = 1;
     z2.re = 2; z2.im = -3.5;

     z3 = add(z1, z2);
     cout << z3.re << "," << z3.im << endl;
     z3 = sub(z1, z2);
     cout << z3.re << "," << z3.im << endl;

    return 0;
}
```

Комплексные числа
Объектный подход

complex_obj.h

```
#ifndef _COMPLEX_H_
#define _COMPLEX_H_

class Complex
{
private:
     double re, im;
public:
     // можно описать несколько конструкторов
     Complex() : re(0), im(0) {}
     Complex(double x, double y)
          : re(x), im(y) {}
     // открываем доступ только на чтение
     double Re() { return re; }
     double Im() { return im; }

     Complex add(Complex b)
     {
          return Complex(re + b.re, im + b.im);
     }
     // аналогично сложению
     Complex sub(Complex b)
     {
          Complex z(this->re - b.re, this->im – b.im);
          return z;
     }
};

#endif
```

test.cpp

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

using namespace std;


int main()
{
     // вызов конструкторов
     Complex z1(1, 1), z2(2, 3.5), z3;

     z3 = z1.add(z2);
     cout << z3.Re() << ","
          << z3.Im() << endl;
     z3 = z1.sub(z2);
     cout << z3.Re() << ","
          << z3.Im() << endl;

     return 0;
}
```

Рациональные числа

Рациональные числа
Процедурный подход

rational_proc.h

```
#ifndef _RATIONAL_H_
#define _RATIONAL_H_

struct Rational
{
     int num, den;
};

// сложение
Rational add(Rational a, Rational b)
{
     Rational result;
     result.num = a.num*b.den + a.den*b.num;
     result.den = a.den * b.den;
     return result;
}


#endif
```

$$\frac{a}{b} * \frac{c}{d} = \frac{a*d+c*b}{b*d}$$

test.cpp

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

using namespace std;


int main()
{
     Rational p, q, r;
     p.num = 1; p.den = 2;
     q.num = 1; q.den = 2;

     r = add(p, q);
     // выведет 4/4
     cout << r.num << "/" << r.den << endl;

     Rational x;
     x.num = 5;
     x.den = 0; // деление на ноль
     r = add(x, q);
     cout << r.num << "/" << r.den << endl;

     return 0;
}
```

Рациональные числа
Объектный подход

rational_obj.h

```
#ifndef _RATIONAL_H_
#define _RATIONAL_H_

class Rational
{
private:
     int num, den;
public:
     Rational() : num(0), den(1) {}
     // опишем конструктор в другом файле
     Rational(int x, int y);
     double getNumerator() { return num; }
     double getDenominator() { return den; }

     Rational add(Rational a, Rational b);

};

#endif
```

rational_obj.cpp

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

// у конструктора не пишут тип возвращаемого
// значения
Rational::Rational(int x, int y)
{
    if (y == 0)
    {
        std::cerr << "Denominator is zero!\n";
    }
     int nod = NOD(abs(x), abs(y));
     num = x / nod; // сокращаем дробь
     den = y / nod;
     if (den < 0)
     {
          num = -num; // пусть знаменатель всегда
          den = -den; // будет больше нуля
     }
}

// сложение
Rational Rational::add(Rational b)
{
     int n = num*b.den + den*b.num;
     int d = den * b.den;
     return Rational(n, d);
}
```

Рациональные числа
Объектный подход

test.cpp

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

using namespace std;


int main()
{
    // вызов конструкторов
    Rational p(1, 2), q(1, 2), r;

    r = p.add(q);
    cout << r.getNumerator() << "/"
         << r.getDenominator() << endl;

    Rational x(5, 0);
    r = x.sub(q);
    cout << r.getNumerator() << "/"
         << r.getDenominator() << endl;

    return 0;
}
```

Класс «Сообщение»

Message

```
#include <iostream>
#include <cstring>
#include <cstdlib>

class Message
{
private:
    char * m_msg; // текущее сообщение
public:
    // Конструктор создает новый контейнер с текстом
    Message(const char * msg);
    // Деструктор ничтожает контейнер
    virtual ~Message();
    // Извлекает сообщение
    const char * getMessage();
private:
    // Реализация алгоритма замены сообщения
    void setMessageImpl(const char * msg);
};
```

Message

```
Message::Message(const char * msg)
{
    m_msg = NULL;
    setMessageImpl(msg);
}


Message::~Message()
{
    if (m_msg != NULL)
    {
        delete [] m_msg;
        m_msg = NULL;
    }
}
```

Ваш класс работает
с динамической памятью?

Это первый признак того,
что нужно описать деструктор!

Message

Написал **new** — напиши **delete**!

```
void Message::setMessageImpl(const char * msg)
{
    if (m_msg != NULL)
    {
        delete [] m_msg;  // удаляем старые данные
    }
    int sz = strlen(msg); // узнаем размер новой строки
    m_msg = new char[sz]; // выделяем память нового размера
    memcpy(m_msg, msg, sz); // копируем данные по байтам
}
```

```
const char * Message::getMessage()
{
    return m_msg;
}
```

Тест класса «Message»

```
int main()
{
    Message helloWorld("Hello, World!");
    cout << helloWorld.getMessage() << endl;

    Message * another = new Message("Another message");
    cout << another->getMessage() << endl;
    delete another;

    return 0;
}
```