作业5

1.什么情况下可以通过重载下标操作符“[]”来实现对其元素的访问？

对于由具有线性关系的元素所构成的对象，可通过重载下标操作符“[]”来实现对其元素的访问。

2.对操作符new和delete进行重载会带来什么好处？为什么重载new和delete操作符必须要使用静态成员函数？

系统提供的new和delete操作所涉及的空间分配和释放是通过系统的堆区管理系统来进行的，它要考虑各种大小的空间分配与释放，对某个类而言，效率常常不高。

可以对操作符new和delete进行重载，使得程序能以自己的方式来实现对某个类的动态对象空间的分配和释放功能。

```cpp
#include<iostream>
using namespace std;

class Complex
{
public:
    Complex(double real, double image);
    void Print();
    bool operator==(const Complex& c);
    Complex& operator++();   //前置++
    Complex operator++(int); //后置++
    Complex& operator--();   //前置--
    Complex operator--(int); //后置--
    bool operator>(const Complex& c);
    Complex& operator=(const Complex& c);

    Complex operator+(const Complex& c);
    Complex& operator+=(const Complex& c);
    Complex operator-(const Complex& c);
    Complex& operator-=(const Complex& c);
    Complex operator*(const Complex& c);
    Complex& operator*=(const Complex& c);
    Complex operator/(const Complex& c);
    Complex& operator/=(const Complex& c);
private:
    double _real;
    double _image;
};

class Polynomial
{
    double* pcoefs;
    int* pexps;
    int num_of_items;
    int add(const Polynomial& p, double* coefs, int* exps) const;
    int subtract(const Polynomial& p, double* coefs, int* exps) const;
public:
    Polynomial();
    Polynomial(double coefs[], int exps[], int size);//系数数组、指数数组和项数
    Polynomial(const Polynomial&);
    ~Polynomial();
    Polynomial& operator=(const Polynomial&);
    int degree() const;//最高幂指数
    double evaluate(double x) const;//计算多项式的值
    bool operator==(const Polynomial&) const;
    bool operator!=(const Polynomial&) const;
    Polynomial operator+(const Polynomial&) const;
    Polynomial operator-(const Polynomial&) const;
    Polynomial& operator+=(const Polynomial&);
    Polynomial& operator-=(const Polynomial&);
};
Complex::Complex(double real = 0.0, double image = 0.0)
    :_real(real)
    , _image(image)
{}

void Complex::Print()
{
    if (_image == 0.0)
    {
        cout << _real << endl;
    }
    else
    {
        cout << _real << "+" << _image << "*i" << endl;
    }
}
Complex Complex::operator+(const Complex& c)
{
    Complex tmp;
    tmp._real = _real + c._real;
    tmp._image = _image + c._image;
    return tmp;
}

Complex Complex::operator-(const Complex& c)
{
    Complex tmp;
    tmp._real = _real - c._real;
    tmp._image = _image - c._image;
    return tmp;
}

Complex Complex::operator*(const Complex& c)
{
    Complex tmp;
    tmp._real = _real * c._real - _image * c._image;
    tmp._image = _real * c._image + _image * c._real;
    return tmp;
}

Complex Complex::operator/(const Complex& c)
{
    Complex tmp;
    double t = c._real * c._real + c._image * c._image;
    tmp._real = (_real * c._real - _image * (-c._image)) / t;
    tmp._image = (_real * (-c._image) + _image * c._real) / t;
```

```cpp
    return tmp;

}

Complex& Complex::operator+=(const Complex& c)
{
    _real += c._real;
    _image += c._image;
    return *this;
}

Complex& Complex::operator-=(const Complex& c)
{
    _real -= c._real;
    _image -= c._image;
    return *this;
}

Complex& Complex::operator*=(const Complex& c)
{
    Complex tmp(*this);
    _real = tmp._real * c._real - _image * c._image;
    _image = tmp._real * c._image + tmp._image * c._real;
    return *this;
}

Complex& Complex::operator/=(const Complex& c)
{
    Complex tmp(*this);
    double t = c._real * c._real + c._image * c._image;
    _real = (tmp._real * c._real - tmp._image * (-c._image)) / t;
    _image = (tmp._real * (-c._image) + tmp._image * c._real) / t;
    return *this;
}

bool Complex::operator==(const Complex& c)
{
    return (_real == c._real) &&
        (_image == c._image);
}

Complex& Complex::operator++()  //前置++
{
    _real++;
    _image++;
    return *this;
}

Complex Complex::operator++(int) //后置++
{
    Complex tmp(*this);
    _real++;
    _image++;
    return tmp;
}

Complex& Complex::operator--()   //前置--
{
    _real--;
    _image--;
    return *this;
}

Complex Complex::operator--(int) //后置--
{
    Complex tmp(*this);
    _real--;
    _image--;
    return tmp;
}

bool Complex::operator>(const Complex& c)
{
    return (_real > c._real) &&
        (_image > c._image);
}

Complex& Complex::operator=(const Complex& c)
{
    if (this != &c)
    {
        _real = c._real;
        _image = c._image;
    }
    return *this;
}
int Polynomial::add(const Polynomial& p, double* coefs, int* exps) const
{
    int count = 0, i = 0, j = 0;
    while (i < num_of_items && j < p.num_of_items)
    {
        if (pexps[i] == p.pexps[j])
        {
            if (pcoefs[i] != -p.pcoefs[j])
            {
                coefs[count] = pcoefs[i] + p.pcoefs[j];
                exps[count] = pexps[i];
                count++;
            }
            i++; j++;
        }
        else if (pexps[i] < p.pexps[j])
        {
            coefs[count] = pcoefs[i];
            exps[count] = pexps[i];
            count++; i++;
        }
        else
        {
            coefs[count] = p.pcoefs[j];
            exps[count] = p.pexps[j];
            count++; j++;
        }
    }
    if (i < num_of_items)
        while (i < num_of_items)
        {
            coefs[count] = pcoefs[i];
            exps[count] = pexps[i];
            count++; i++;
        }
    else
        while (j < p.num_of_items)
        {
            coefs[count] = p.pcoefs[j];
            exps[count] = p.pexps[j];
            count++; j++;
        }
    return count;
}
int Polynomial::subtract(const Polynomial& p, double* coefs, int* exps) const
{
    int count = 0, i = 0, j = 0;
    while (i < num_of_items && j < p.num_of_items)
    {
        if (pexps[i] == p.pexps[j])
        {
            if (pcoefs[i] != p.pcoefs[j])
            {
                coefs[count] = pcoefs[i] - p.pcoefs[j];
                exps[count] = pexps[i];
                count++;
            }
            i++; j++;
        }
        else if (pexps[i] < p.pexps[j])
        {
            coefs[count] = pcoefs[i];
            exps[count] = pexps[i];
            count++; i++;
        }
        else
        {
            coefs[count] = -p.pcoefs[j];
            exps[count] = p.pexps[j];
            count++; j++;
        }
    }
    if (i < num_of_items)
        while (i < num_of_items)
        {
            coefs[count] = pcoefs[i];
            exps[count] = pexps[i];
            count++; i++;
        }
    else
        while (j < p.num_of_items)
        {
            coefs[count] = -p.pcoefs[j];
            exps[count] = p.pexps[j];
            count++; j++;
        }
    return count;
}
```

```cpp
Polynomial::Polynomial()
{
    pcoefs = NULL;
    pexps = NULL;
    num_of_items = 0;
}
Polynomial::Polynomial(double coefs[], int exps[], int size)
{
    num_of_items = size;
    pcoefs = new double[num_of_items];
    pexps = new int[num_of_items];
    int i;
    for (i = 0; i < num_of_items; i++)
    {
        pcoefs[i] = coefs[i];
        pexps[i] = exps[i];
    }
    for (i = num_of_items; i > 1; i--)
    {
        bool exchange = false;
        for (int j = 1; j < i; j++)
        {
            if (pexps[j] < pexps[j - 1])
            {
                int temp1 = pexps[j];
                pexps[j] = pexps[j - 1];
                pexps[j - 1] = temp1;
                double temp2 = pcoefs[j];
                pcoefs[j] = pcoefs[j - 1];
                pcoefs[j - 1] = temp2;
                exchange = true;
            }
        }
        if (!exchange) break;
    }
}
Polynomial::Polynomial(const Polynomial& p)
{
    num_of_items = p.num_of_items;
    pcoefs = new double[num_of_items];
    pexps = new int[num_of_items];
    for (int i = 0; i < num_of_items; i++)
    {
        pcoefs[i] = p.pcoefs[i];
        pexps[i] = p.pexps[i];
    }
}
Polynomial::~Polynomial()
{
    delete[]pcoefs;
    delete[]pexps;
    pcoefs = NULL;
    pexps = NULL;
    num_of_items = 0;
}
Polynomial& Polynomial::operator=(const Polynomial& p)
{
    delete[]pcoefs;
    delete[]pexps;
    num_of_items = p.num_of_items;
    pcoefs = new double[num_of_items];
    pexps = new int[num_of_items];
    for (int i = 0; i < num_of_items; i++)
    {
        pcoefs[i] = p.pcoefs[i];
        pexps[i] = p.pexps[i];
    }
    return *this;
}
int Polynomial::degree() const
{
    if (num_of_items == 0)
        return 0;
    else
        return pexps[num_of_items - 1];
}
double Polynomial::evaluate(double x) const
{
    double sum = 0;
    for (int i = 0; i < num_of_items; i++)
    {
        double temp = pcoefs[i];
        for (int j = 0; j < pexps[i]; j++)
            temp *= x;
        sum += temp;
    }
    return sum;
}
bool Polynomial::operator==(const Polynomial& p) const
{
    if (num_of_items != p.num_of_items) return false;
    for (int i = 0; i < num_of_items; i++)
        if (pcoefs[i] != p.pcoefs[i] || pexps[i] != p.pexps[i])
            return false;
    return true;
}
bool Polynomial::operator!=(const Polynomial& p) const
{
    return !(*this == p);
}
Polynomial Polynomial::operator+(const Polynomial& p) const
{
    double* coefs = new double[num_of_items + p.num_of_items];
    int* exps = new int[num_of_items + p.num_of_items];
    int count = add(p, coefs, exps);
    Polynomial temp(coefs, exps, count);
    delete[]coefs;
    delete[]exps;
    return temp;
}
Polynomial Polynomial::operator-(const Polynomial& p) const
{
    double* coefs = new double[num_of_items + p.num_of_items];
    int* exps = new int[num_of_items + p.num_of_items];
    int count = subtract(p, coefs, exps);
    Polynomial temp(coefs, exps, count);
    delete[]coefs;
    delete[]exps;
    return temp;
}
Polynomial& Polynomial::operator+=(const Polynomial& p)
{
    double* coefs = new double[num_of_items + p.num_of_items];
    int* exps = new int[num_of_items + p.num_of_items];
    int count = add(p, coefs, exps);
    delete[]pcoefs;
    delete[]pexps;
    pcoefs = coefs;
    pexps = exps;
    num_of_items = count;
    return *this;
}
Polynomial& Polynomial::operator-=(const Polynomial& p)
{
    double* coefs = new double[num_of_items + p.num_of_items];
    int* exps = new int[num_of_items + p.num_of_items];
    int count = subtract(p, coefs, exps);
    delete[]pcoefs;
    delete[]pexps;
    pcoefs = coefs;
    pexps = exps;
    num_of_items = count;
    return *this;
}
```