Тема 12

Перегрузка операций

Что такое операция?

- Каждая операция реализована как вызов функции, так что

```
a = b+c
```

эквивалентно

```
operator = (a, operator + (b, c))
```

- Другой способ задания операции: если a – объект некоторого класса, то

```
a + b
```

эквивалентно

```
a.operator +(b);
```

Пример перегрузки операций

```
a << 2;
```

- если a принадлежит одному из целочисленных типов, то это – операция побитового сдвига влево (результат – a*4, значение a не меняется;
- если a – выходной поток, в этот поток помещается десятичное представление числа 2 (или представление, соответствующее текущему состоянию потока). Поток a изменяется.

Ограничения на перегрузку операций

- обозначения собственных операций вводить нельзя;
- перегрузка операций для стандартных типов не допускается;
- операции «`.`», «`.*`», «`?:`», «`#`», «`##`», «`::`», «**`sizeof`**» не могут быть перегружены;
- при перегрузке операций должно сохраняться число аргументов и приоритеты, принятые для стандартных типов;
- при перегрузке операций нельзя задавать аргументы по умолчанию.

Реализация перегрузки операций при написании классов

Функция-операция может быть определена как:

- метод класса
- дружественная функция класса
- обычная функция.

Операции над классом Point2D

- Сложение и вычитание точек (по правилам сложения и вычитания векторов);
- Унарный минус (по правилам векторов)
- Умножение точки на число и числа на точку (результат – растяжение или сжатие точки);
- Умножение двух точек (результат – скалярное произведение);
- Инкремент и декремент точки (результат – увеличение или уменьшение координат на 1);
- Сравнение на равенство и неравенство двух точек.

Реализация унарных операций

Если унарная операция определяется как метод класса, она описывается без параметров (параметром считается вызвавший ее объект), в противном случае параметром такой функции должна быть ссылка на объект класса (если операнд изменяется) или константная ссылка (в противном случае).

Функция должна возвращать ссылку на объект класса, если результат должен быть L-value. В противном случае результат зависит от сути функции.

Примеры реализации унарных операций для класса Point2D

1. Унарный минус:

```
class Point2D {
…
Point2D operator – () const;
…
};
```

```
Point2D Point2D::operator – () const {
  return Point2D(-x, -y);
}
```

Примеры реализации унарных операций для класса Point2D

2. Инкремент (префиксный) :

Отметим, что операция префиксного инкремента возвращает Lvalue, и что возвращается новое значение

```
class Point2D {
…
Point2D& operator ++ ();
…
};
```

```
Point2D& Point2D::operator ++ () {
  x++;
  y++;
  return *this;
}
```

Примеры реализации унарных операций для класса Point2D

3. Инкремент (постфиксный) :

Отметим, что операция постфиксного инкремента возвращает не-Lvalue, и что возвращается старое значение. Кроме того, добавляется фиктивный параметр для указания типа операции.

```
class Point2D {
…
Point2D operator ++ (int);
…
}
```

```
Point2D Point2D::operator ++ (int a) {
  Point2D p(*this);
  x++;    y++;
  return p;
}
```

Реализация бинарных операций

Если бинарная операция определяется как метод класса, она описывается с одним параметром, соответствующим второму операнду. Первым операндом считается вызвавший ее объект. В противном случае такая функция должна иметь два параметра, соответствующих операндам операции.

Функция должна возвращать ссылку на объект класса, если результат должен быть L-value. В противном случае результат зависит от сути функции.

Примеры реализации бинарных операций для класса Point2D

1. Сложение:

```
class Point2D {
…
Point2D operator + (const Point2D&) const;
…
};
```

```
Point2D Point2D::operator + (const Point2D& p) const
{
  return Point2D(x + p.x, y + p.y);
}
```

Примеры реализации бинарных операций для класса Point2D

2. Скалярное произведение:

```
class Point2D {
…
double operator * (const Point2D&) const;
…
};
```

```
double Point2D::operator * (const Point2D& p) const
{
  return (x*p.x + y*p.y);
}
```

Примеры реализации бинарных операций для класса Point2D

3. Сравнение :

```
class Point2D {
…
bool operator == (const Point2D&) const;
bool operator != (const Point2D&) const;
…
};
```

```
bool Point2D::operator == (const Point2D& p) const
{
  return (x==p.x && y==p.y);
}
bool Point2D::operator != (const Point2D& p) const
{
  return (! this->operator==(p));
}
```

Примеры реализации бинарных операций для класса Point2D

4. Умножение на число :

```
class Point2D {
…
Point2D operator * (double) const;
friend Point2D operator *(double, const Point2D&);
…
};
```

```
Point2D Point2D::operator * (double d) const
{
  return Point2D(d*x, d*y);
}
Point2D operator * (double d, const Point2D& p)
{
   return p.operator *(d);
}
```

Примеры реализации бинарных операций для класса Point2D

4. Вывод в поток:

```
class Point2D {
…
friend ostream& operator <<(ostream&,
                const Point2D&);
…
};
```

```
ostream& operator <<(ostream& s, const Point2D& p)
{
  s << "(" << p.x << ", " << p.y << ")";
  return s;
}
```

Перегрузка операции присваивания

Для вновь создаваемых классов создаётся перегруженная **операция присваивания по умолчанию**. Эта операция копирует все поля объекта из правой части в соответствующие поля объекта из левой части.

Если этого достаточно, то собственную реализацию этой операции можно не писать!

Для класса Point2D достаточно работы стандартной операции присваивания. Для класса Person необходимо писать собственную реализацию, т.к. он захватывает ресурсы во время своего существования.

Схемы работы перегруженной операции присваивания a = b

- проверка на самоприсваивание;
- освобождение ресурсов, полученных объектом a;
- получение ресурсов, закрепленных за объектом b (в случае, если ресурсом является динамическая память – клонирование)

Реализация операции присваивания для класса Person

```
class Person {
…
Person& operator =(const Person&);
…
};
```

```
Person& Person::operator =(const Person& p)
{
  if (this == &p)
    return *this;
  delete [] name;
  name = new char[strlen(p.name)+1];
  strcpy(name, p.name);
  return *this;
}
```

Методы Clone и Erase

Для эффективной реализации работы с ресурсами можно написать два защищённых метода:

- метод, освобождающий все занятые объектом ресурсы (Erase);
- метод, клонирующий ресурсы другого объекта (Clone)

```
class Person {
private:
 void Erase();
 void Clone(const Person&);
…
}
```

Реализация и использование методов Clone и Erase

```
void Person::Erase() {
  delete [] name;
}
void Person::Clone(const Person& p){
  name = new char[strlen(p.name)+1];
  strcpy(name, p.name);
}

Person:: ~Person() {
  Erase();
}
Person::Person(const Person& p): ID(++newID){
  Clone(p);
}
const Person& Person::operator =(const Person& p){
  if (this != &p) {
    Erase(); Clone(p);
  }
  return *this;
}
```

Перегрузка операции приведения типа

- Запись: operator тип()
- Нет возвращаемого значения

```
class Point2D {
…
operator double ();
…
};
```

```
Point2D::operator double() {
  return Module();
}
```

Перегрузка операции приведения типа

После написания этого кода возникают сложности с использованием других перегруженных операций:

```
Point2D p(10,5);
…
cout << 3*p; //Ошибка C2666:
// number overloads have similar conversions

cout << operator*(3, p) //Ошибки нет

cout << 3*(double)p;  // Ошибки тоже нет
```

Перегрузка оператора вызова функции

- **Запись:** тип_возврата **operator()** (формальные параметры);
- Класс, в котором определён хотя бы один оператор вызова функции, называется **функциональным классом**. Функциональный класс может не иметь других полей и методов.
- Функциональные классы используются в алгоритмах из библиотеки STL.

Пример работы с функциональным классом

```
class IsBest {
…
 bool operator () (int a, int b) {
   return (a<b || a%10==0);
 }
…
};
```

```
IsBest best;
…
cout << best(10,4) << best(5,4);
…
int Mas[100];
// заполняем массив
sort(Mas, Mas+100, IsBest);
```

Что такое очередь?

Очередью называется структура данных, содержащая последовательность однотипных элементов и позволяющая эффективно выполнять следующие операции:

- вставка нового элемента в **хвост** очереди;
- просмотр элемента, находящегося в **голове** очереди;
- удаление из очереди элемента, находящегося в голове;
- определение количества элементов в очереди (или определение того, пуста ли очередь)

Что такое очередь?

Над очередью иногда можно выполнить дополнительные операции:

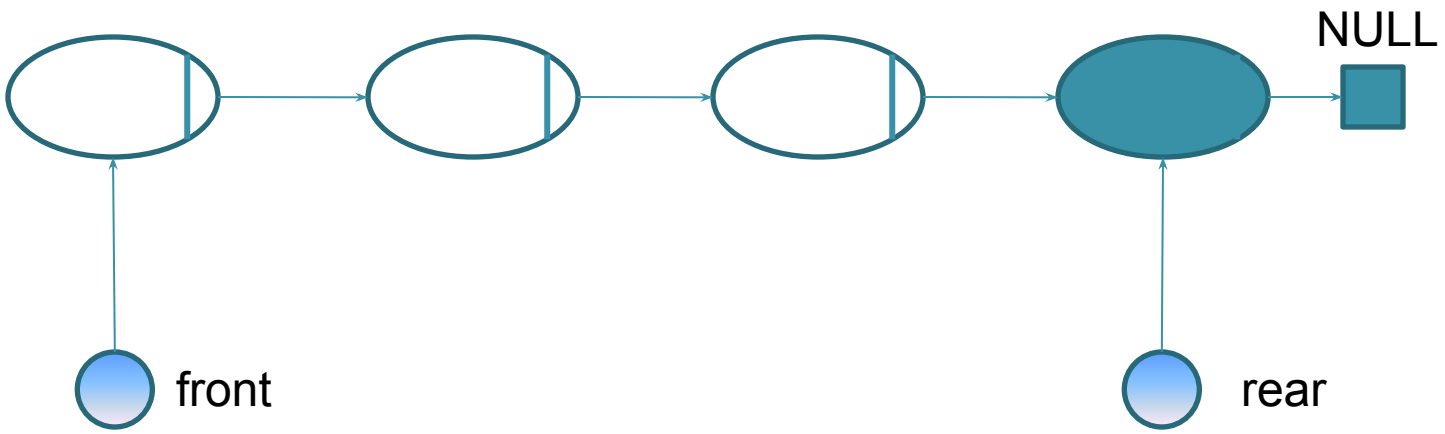
- доступ к элементу очереди по его номеру (элемент, находящийся в голове очереди, имеет номер 0);
- выполнение однотипных действий над всеми элементами очереди

Реализация очереди

Существует несколько способов реализации очереди. Основные способы:

- на массивах (циклическая очередь);
- на линейных списках

Рассмотрим второй способ реализации очереди.



Описание очереди (файл queue.h)

Часть I (предварительные описания)

```
#ifndef __LQueue_defined__
#define __LQueue_defined__

#include <iostream>
using namespace std;

typedef int InfoType;

class LQueue {
 …
};

#endif
```

Описание очереди (файл queue.h)

Часть 2 (защищённые поля и методы)

```
private:
   struct QItem {
    InfoType info;
    QItem* next;
    QItem(InfoType Ainfo): info(Ainfo), next(NULL) {}
   };
   QItem *front, *rear;
   unsigned size;

   void Erase();
   void Clone(const LQueue &);
```

Описание очереди (файл queue.h)

Часть 3 (публичные методы)

```
public:
  LQueue(): front(NULL), rear(NULL), size(0) {};
  LQueue(const LQueue&);
  ~LQueue();
  LQueue& operator = (const LQueue&);

  void Push(InfoType AInfo);
  bool Pop();
  InfoType GetFirst() const;
  bool IsEmpty()const;
  unsigned GetSize() const;

  InfoType operator [] (unsigned) const;
  void Browse(void ItemWork(InfoType)) const;
  void Browse(void ItemWork(InfoType&));
```

Реализация очереди (файл queue.cpp)

Часть I (защищённые методы)

```
void LQueue::Erase() {
  while (Pop());
  size = 0;
}

void LQueue::Clone(const LQueue& Q) {
  //for (unsigned i=0; i<Q.size; i++)
  //  Push(Q[i]);
  QItem *tmp = Q.front;
  for (unsigned i=0; i<Q.size; i++) {
   Push(tmp->info);
   tmp = tmp->next;
  }
}
```

Реализация очереди (файл queue.cpp)

Часть 2 (конструктор копирования, деструктор, оператор присваивания)

```
LQueue::LQueue(const LQueue& Q) {
  size = 0;  Clone(Q);
}

LQueue::~LQueue() {
  Erase();
}

LQueue& LQueue::operator = (const LQueue& Q) {
  if (this != &Q) {
   Erase();
   Clone(Q);
  }
  return *this;
}
```

Реализация очереди (файл queue.cpp)

Часть 3 (метод Push)

```
void LQueue::Push(InfoType Ainfo) {
  QItem* tmp = new QItem(Ainfo);
  if (size>0)
   rear->next = tmp;
  else
   front = tmp;
  rear = tmp;
  size++;
}
```

Реализация очереди (файл queue.cpp)

Часть 4 (метод Pop)

```
bool LQueue::Pop() {
  if (size==0)
   return false;
  QItem *tmp = front;
  front = front->next;
  delete tmp;
  size--;
  if (size==0)
   rear = NULL;
  return true;
}
```

Реализация очереди (файл queue.cpp)

Часть 4 (методы GetFirst и IsEmpty)

```
InfoType LQueue::GetFirst() const {
  if (size==0)
   throw exception("Impossible to execute GetFirst:
  queue is empty");
  return front->info;
}

bool LQueue::IsEmpty() const {
  return (size==0);
}
```

Реализация очереди (файл queue.cpp)

Часть 4 (методы GetSize и operator [])

```
unsigned LQueue::GetSize() const {
  return size;
}

InfoType LQueue::operator [] (unsigned k) const {
  if ((k<0) || (k>=size))
   throw exception("Impossible to execute operator[]:
  invalid index");
  QItem *tmp = front;
  for (unsigned i=0; i<k; i++)
   tmp = tmp->next;
  return tmp->info;
}
```

Реализация очереди (файл queue.cpp)

Часть 4 (другой вариант перегрузки [])

```
const InfoType& LQueue::GetByIndex (unsigned k)
  const
{
  if ((k<0) || (k>=size))
   throw exception("Impossible to execute operator[]:
  invalid index");
  QItem *tmp = front;
  for (unsigned i=0; i<k; i++)
   tmp = tmp->next;
  return tmp->info;
}

InfoType& LQueue::operator [] (unsigned k)
{
    return (InfoType&) GetByIndex(k);
}
```

Реализация очереди (файл queue.cpp)

Часть 5 (перегруженный метод Browse)

```
void LQueue::Browse(void ItemWork(InfoType)) const {
    QItem *tmp = front;
    for (unsigned i=0; i<size; i++) {
        ItemWork(tmp->info);
        tmp = tmp->next;
    }
}

void LQueue::Browse(void ItemWork(InfoType&)) {
    QItem *tmp = front;
    for (unsigned i=0; i<size; i++) {
        ItemWork(tmp->info);
        tmp = tmp->next;
    }
}
```