Объектно-ориентированное программирование на C++

Причины возникновения объектно-ориентированного программирования

С ростом объема кода программы становится невозможным удерживать в памяти все детали

Необходимо структурировать информацию, выделять главное и отбрасывать несущественное

Этот процесс называется **повышением степени абстракции программы**

Одним из естественных вариантов борьбы со сложностью является **алгоритмическая декомпозиция** -разбиение задачи на подзадачи

Процедурное программирование – подход, при котором исходная задача разбивается на подзадачи

Каждая подзадача оформляется в виде функции

Использование функций - первый шаг к повышению абстракции

Это позволяет отвлечься от деталей ее реализации, поскольку для вызова функции требуется знать только ее интерфейс

Следующий шаг — описание собственных типов данных, позволяющих структурировать и группировать информацию

Процедурное программирование – подход, при котором **функции** и **переменные**, относящиеся к какому-то конкретному объекту свободно располагаются в коде и **никак между собой не связаны**

Следующий шаг к повышению абстракции – **объектно-ориентированный подход**

Объектная декомпозиция – выделение сущностей из предметной области

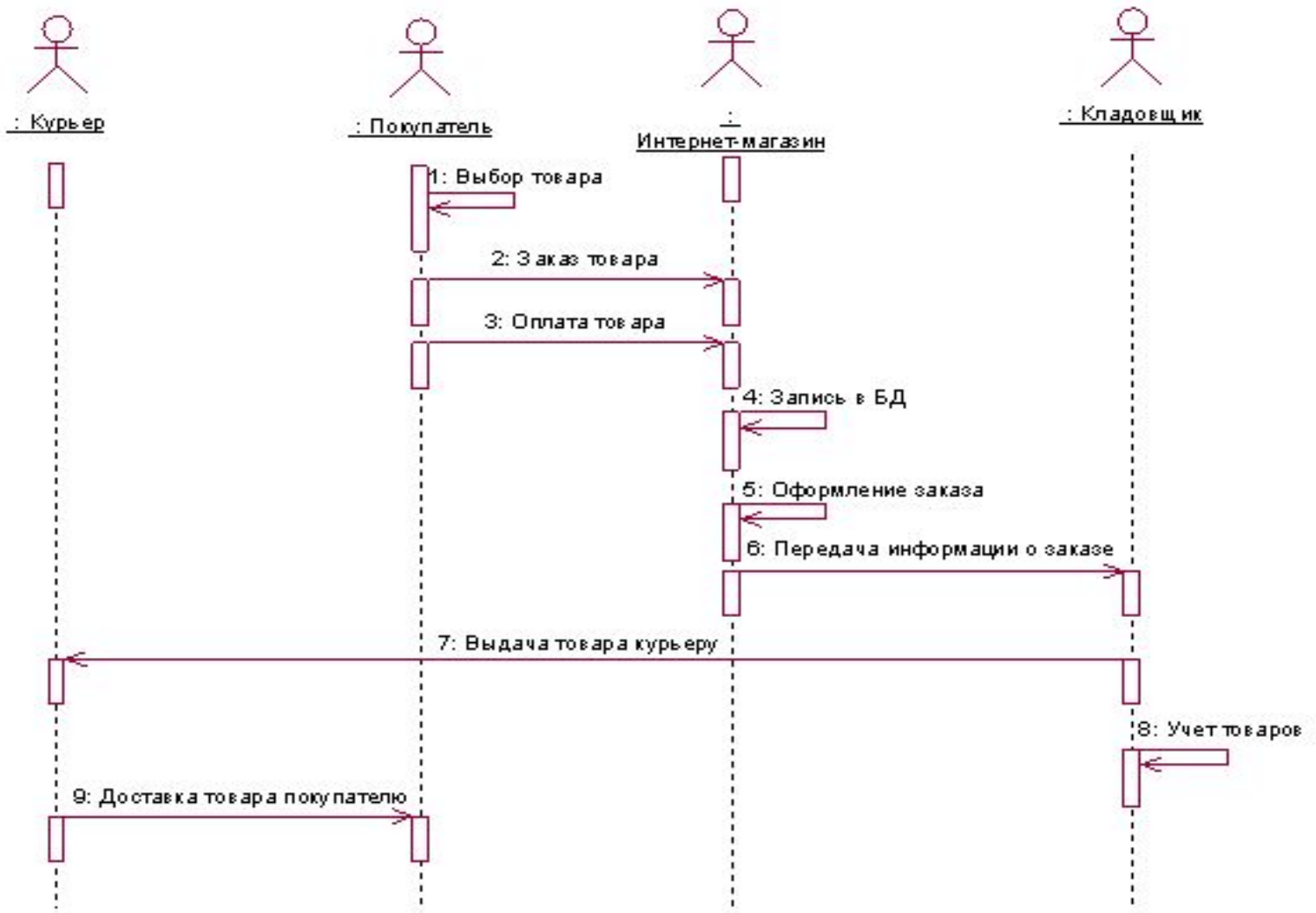
Каждая сущность представляется в коде программы в виде класса

Класс - общее абстрактное описание некоторой сущности

Объектно-ориентированное программирование – подход, при котором объекты реального мира представляются в коде в виде экземпляров классов

Программа представляет собой совокупность **взаимодействующих объектов**, имеющих **состояние** и **поведение**

Интернет-магазин бытовой техники



Объектно-ориентированное программирование – подход, при котором **функции** и **переменные**, относящиеся к конкретному объекту **объединены** в коде и **тесно связаны между собой**

Концепция «черного ящика» является одной из базовых концепций ООП

Снаружи объект принято рассматривать как «черный ящик», т.е. некий прибор с кнопками

Основные понятия ООП

- Инкапсуляция
- Наследование
- Полиморфизм

Инкапсуляция –
это объединение полей и методов объекта в единое целое - **класс**

Синтаксис объявления класса

```
class имя_класса
{
[private | protected | public]:
   тип_поля1 имя_поля1;
   тип_поля2 имя_поля2;
   тип_поля3 имя_поля3;
   ...
   тип1 имя_метода1(список_параметров)
   {
      ...
   }

   тип2 имя_метода2(список_параметров)
   {
      ...
   }
   ...
} [список_переменных];
```

Способы доступа к компонентам класса

- Открытый (public)
- Защищенный (protected)
- Закрытый (private)

Пример объявления класса

```
class Date
{
private: // доступ к этим полям будут иметь только методы класса
    int year;
    int month;
    int day;
public: // открытые методы класса(интерфейсная часть класса)
    void put_date()
    {
        cout << "Year : ";
        cin >> year;
        cout << "Month : ";
        cin >> month;
        cout << "Day : ";
        cin >> day;
    }
    void print_date()
    {
        cout << "Date: " << year << "/"
             << month << "/" << day << endl;
    }
};
```

Важнейшее требование инкапсуляции - скрытие состояния объекта от внешнего мира

Инкапсуляция повышает степень абстракции программы

Данные класса и реализация методов класса находятся ниже уровня абстракции, и для написания программы информация о них не требуется

Инкапсуляция позволяет изменить реализацию класса без модификации основной части программы, если интерфейс остался прежним

Наследование –
это механизм, который позволяет расширять существующие классы, сохраняя их функциональность и добавляя им новые свойства и методы

Полиморфизм –
это механизм, который позволяет использовать одно и тоже имя для решения нескольких технически разных задач

Объект как экземпляр класса – это некоторая уникальная единица, имеющая свои переменные (поля) и функции (методы), эти переменные обрабатывающие

Поля объекта - это переменные, описывающие его состояние, а **методы** - это способ перевести объект из одного состояния в другое

Пример создания объекта класса

```
void main()
{
    Date d; // создание обьекта класса Date
    d.put_date(); // вызов метода класса Date put_date()
    d.print_date(); //вызов метода класса Date print_date()
}
```

Методы-аксессоры

- Инспекторы позволяют получить значения полей
- Модификаторы позволяют установить значения полей

Методы-аксессоры

```
class Date
{
    int year;
    int month;
    int day;
public: // открытые методы класса(интерфейсная часть класса)
    void setYear(int y) // модификатор для поля year
    {
        year = y;
    }
    int getYear() // инспектор для поля year
    {
        return year;
    }
    void setMonth(int m)  // модификатор для поля month
    {
        month = m;
    }
    int getMonth() // инспектор для поля month
    {
        return month;
    }
    void setDay(int d) // модификатор для поля day
    {
        day = d;
    }
    int getDay() // инспектор для поля day
    {
        return day;
    }
};
```

Конструктор -
это специальный метод класса, который вызывается для конструирования объекта в момент его создания

Конструктор не возвращает значение, даже типа void

Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации

Конструктор, вызываемый без параметров, называется конструктором по умолчанию

Параметры конструктора могут иметь любой тип, кроме этого же класса

Если программист не указал ни одного конструктора, компилятор создаст его автоматически

Деструктор –
это специальный метод класса, который вызывается при уничтожении объекта

Деструктор не принимает никаких параметров и не возвращает значений

Класс может иметь только один деструктор

Если деструктор явным образом не определен, компилятор автоматически создаст пустой деструктор

this – константный указатель на объект класса, который неявно передается в каждый нестатический метод класса

Конструктор копирования

Три ситуации, когда новый объект конструируется путем копирования существующего

- Объект создается и инициализируется другим объектом
- Объект передается в функцию по значению
- Объект возвращается из функции по значению

Объект создается и инициализируется другим объектом

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

class MyClass
{
    int n;
public:
    MyClass(int a)  {   n = a;  }
    int getValue()  {   return n;  }
};

void main()
{
    MyClass obj1(100);
    cout << obj1.getValue() << endl;
    MyClass obj2 = obj1;
    cout << obj2.getValue() << endl;
}
```

Объект передается в функцию по значению

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

class MyClass {
    int n;
public:
    MyClass(int a)  {   n = a;  }
    int getValue()  {   return n;   }
};

void f1(MyClass obj) {
    cout << obj.getValue() << endl;
}

void main() {
    MyClass obj(100);
    cout << obj.getValue() << endl;
    f1(obj);
}
```

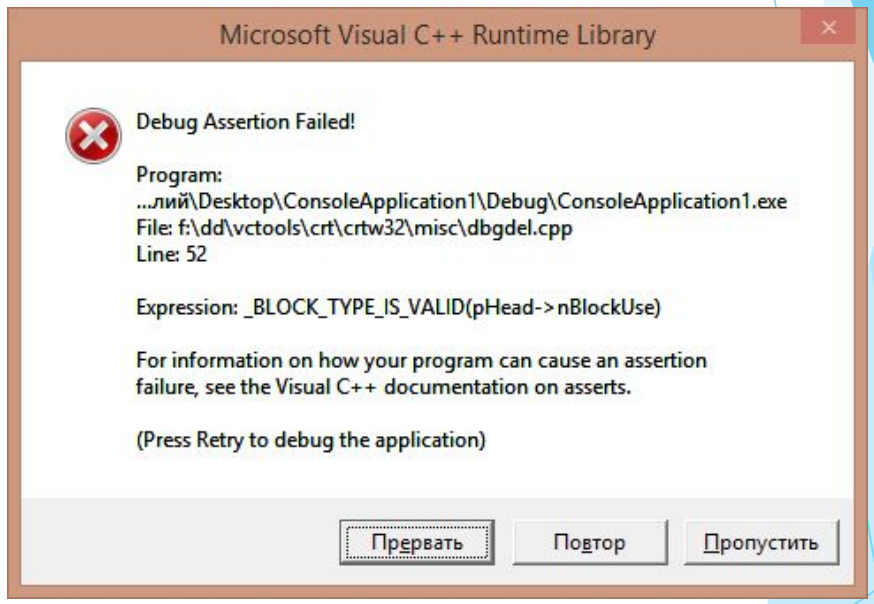
Объект возвращается из функции по значению

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

class MyClass
{
    int n;
public:
    MyClass(int a)  {   n = a;  }
    int getValue()  {   return n;   }
};

MyClass f(){
    MyClass obj(100);
    cout << obj.getValue() << endl;
    return obj;
}

void main(){
    cout << f().getValue() << endl;
}
```

Если в объекте класса есть указатель, адресующий участок динамической памяти, то создается предпосылка возникновения ошибки на этапе выполнения программы

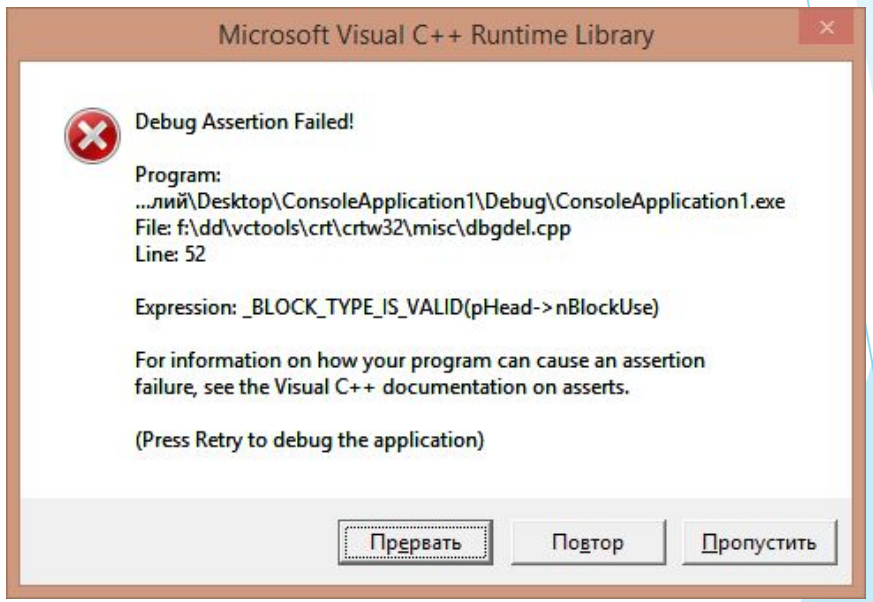
Объект создается и инициализируется другим объектом

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

class MyClass {
private:
    int *p;
public:
    MyClass(int a){ p = new int(a); }
    ~MyClass(){ delete p; }
    int getValue() { return *p; }
};

void main() {
    MyClass obj1(100);
    cout << obj1.getValue() << endl;
    MyClass obj2 = obj1;
    cout << obj2.getValue() << endl;
}
```



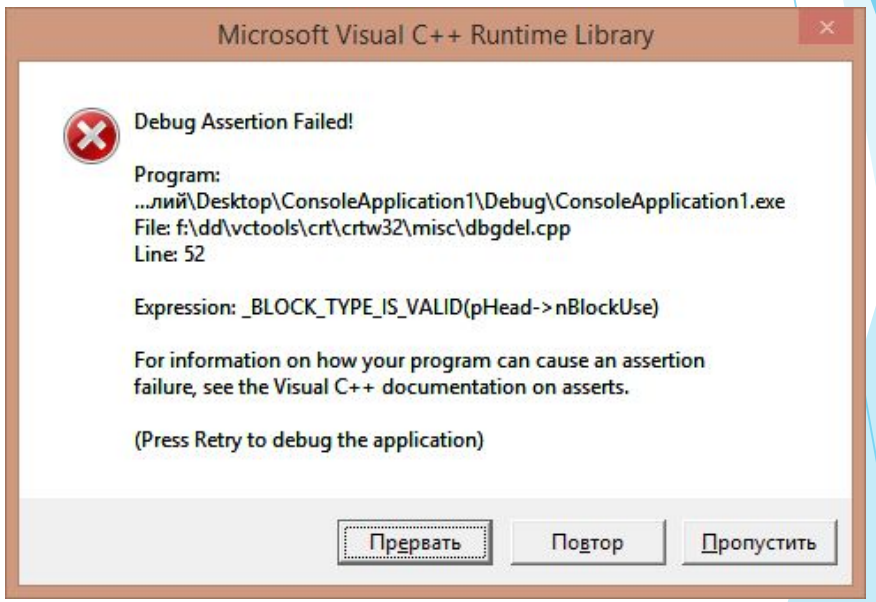
Объект передается в функцию по значению

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

class MyClass {
private:
    int *p;
public:
    MyClass(int a) {    p = new int(a); }
    ~MyClass()  {   delete p;   }
    int getValue()  {   return *p;  }
};

void f(MyClass obj){
    cout << obj.getValue() << endl;
}

void main(){
    MyClass obj(100);
    cout << obj.getValue() << endl;
    f(obj);
}
```

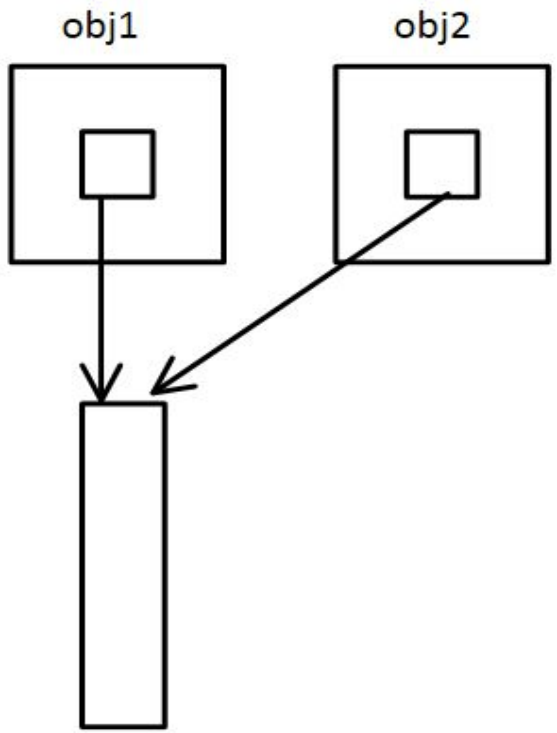


Объект возвращается из функции по значению

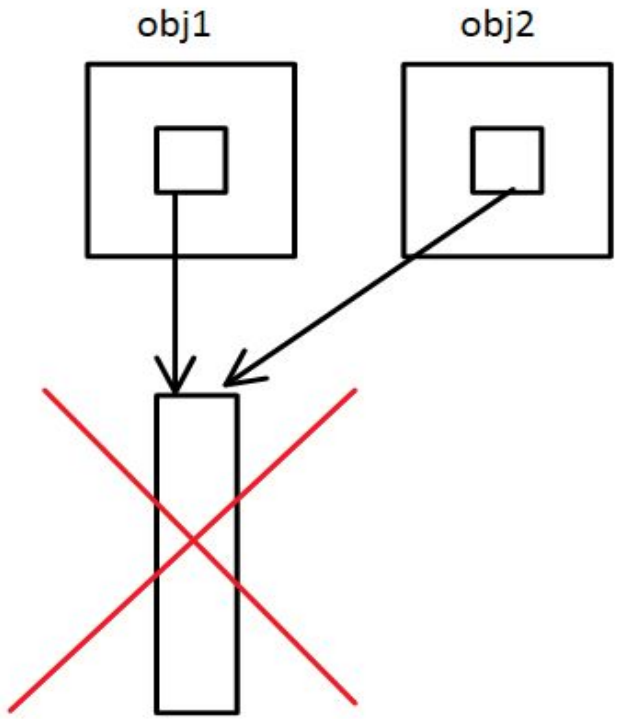
```
#include <iostream>
using namespace std;

class MyClass {
private:
    int *p;
public:
    MyClass(int a)  {   p = new int(a); }
    ~MyClass()  {   delete p;   }
    int getValue()  {   return *p;  }
};

MyClass f(){
    MyClass obj(100);
    cout << obj.getValue() << endl;
    return obj;
}

void main(){
    cout << f().getValue() << endl;
}
```



При поэлементном копировании указатели двух различных объектов будут адресовать один и тот же участок динамической памяти

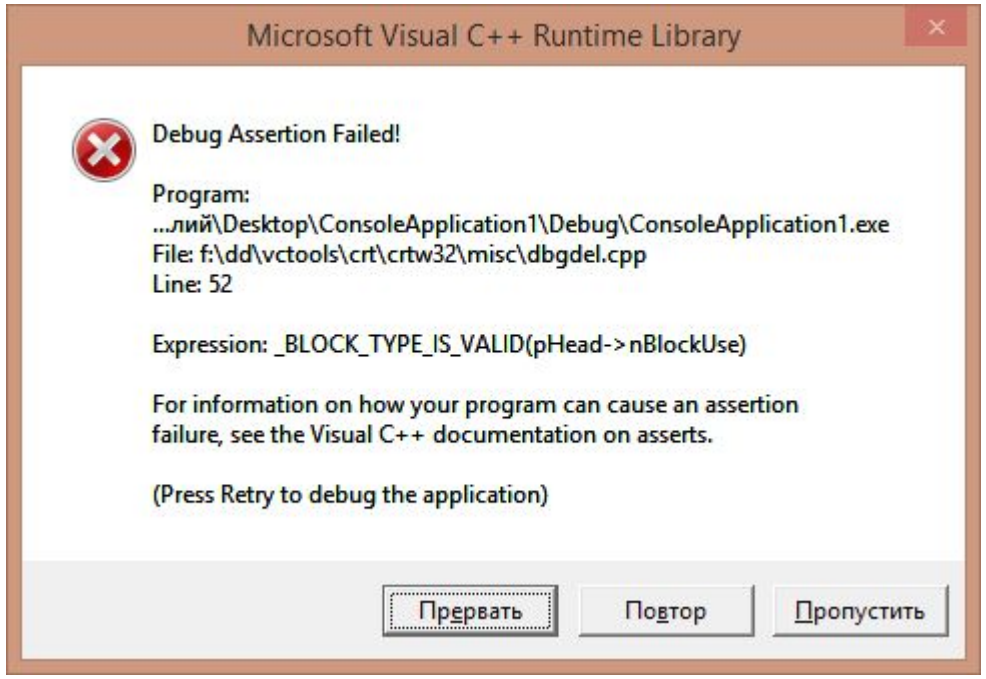


При разрушении одного из объектов сработает деструктор, который освободит динамическую память



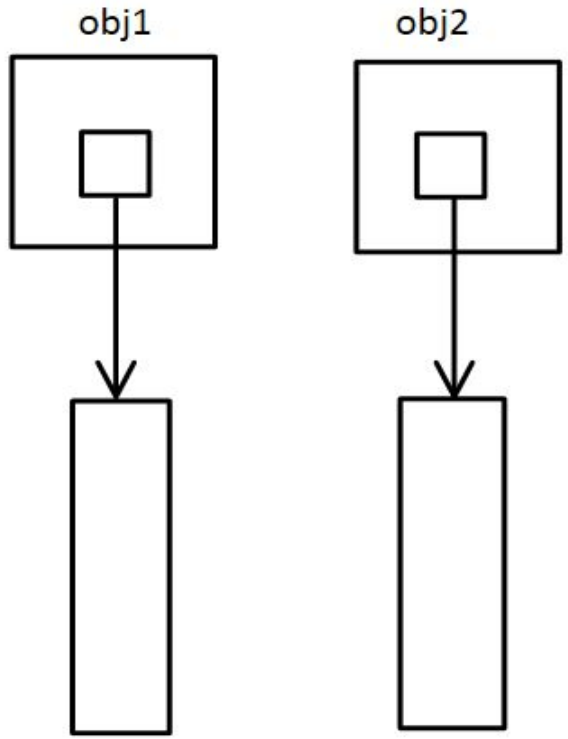
При разрушении второго объекта снова сработает деструктор, который попытается повторно удалить ранее освобожденный участок динамической памяти

Произойдет ошибка этапа выполнения программы



Для решения данной проблемы необходимо обеспечить класс собственной реализацией конструктора копирования

В конструкторе копирования необходимо выделить память для указателя вновь создаваемого объекта



Конструктор копирования

```
class MyClass
{
private:
    int *p;
public:
    MyClass(int a) { ... }
    ~MyClass() { ... }
    int getValue() { ... }

    MyClass(const MyClass & obj)
    {
        p = new int;
        *p = *obj.p;
        cout << "\nCopy-Constructor " << *p << endl;
    }
};
```

Статические элементы класса

Статические поля класса – это глобальные переменные, доступные только в пределах области класса

Статические поля применяются для хранения данных, общих для всех объектов класса

Эти поля существуют для всех объектов класса в единственном экземпляре, то есть не дублируются

Статические поля класса должны быть определены глобально после описания класса

В этом случае под них будет выделено соответствующее количество байт памяти

Статические поля класса

```
class Test
{
public:
    static int count;
    Test()
    {
        count++;
    }
    ~Test()
    {
        count--;
    }
};
int Test::count = 0;
```

Статические поля доступны как через имя класса, так и через имя объекта

```
void main()
{
    Test t1, t2, *p;
    p = &t2;
    cout << Test::count << endl;
    cout << t1.count << endl;
    cout << p->count << endl;
}
```

На статические поля распространяется действие спецификаторов доступа

Память, занимаемая статическим полем, не учитывается при определении размера объекта с помощью операции sizeof

Статические методы предназначены для обращения к статическим полям класса

Статическим методам не передается скрытый указатель на объект класса (this)

Поэтому они могут обращаться непосредственно только к статическим полям и вызывать только статические методы класса

Статические элементы класса

```
class Test{
    int data;
    static int count;
public:
    Test(int d) :data(d) {
        count++;
    }
    static int getObjectCount(){
        return count;
    }
    ~Test()  {
        count--;
    }
};
int Test::count = 0;
```

Обращение к статическим методам производится либо через имя класса, либо через имя объекта

```
void main()
{
    Test t1(10);
    cout << Test::getObjectCount();
    Test t2(20), *p = &t1;
    cout << t2.getObjectCount();
    cout << p->getObjectCount();
}
```

Перегрузка операторов

Перегрузка операторов – возможность интегрировать пользовательский тип данных в язык программирования

Перегрузка операторов дает возможность определить собственные действия для стандартных операций, применяемых к объектам пользовательского типа

Общий синтаксис перегрузки операторов

```
тип operator знак_операции (параметры)
{
    < тело функции >
}
```

Ограничения

- Нельзя перегружать операции для стандартных типов
- Нельзя создавать новые названия операций
- Перегрузка не меняет приоритет операций
- В C++ нет неявной перегрузки операций
- Для той или иной операции нельзя изменить количество операндов

Запрещенные к перегрузке операторы

- :: - оператор разрешения области видимости
- sizeof - оператор определения размера объекта
- . - оператор выбора
- ?: - условный тернарный оператор

Способы перегрузки операторов

- Функция-операция представляется в виде метода класса
- Функция-операция представляется в виде обычной или дружественной функции

Дружественные функции

Функция, объявленная в классе с ключевым словом **friend**, является дружественной по отношению к данному классу

Это означает, что данная функция имеет прямой доступ к скрытым полям класса

Поскольку дружественной функции не передается указатель **this**, то она должна принимать в качестве параметра ссылку на объект класса

Дружественные функции

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

class A
{
    int data;
public:
    A(int d):data(d){}
    friend void someFunc(A& obj);
};

void someFunc(A& obj)
{
    cout << obj.data << endl;
    obj.data += 10;
    cout << obj.data << endl;
}
```

На дружественные функции не распространяются спецификаторы доступа

Одна функция может быть дружественной сразу нескольким классам

Другом может быть не только функция, но и метод другого ранее определенного класса

Дружественные методы

```
class CPoint
{
    int x,y;
public:
    CPoint() : x(0), y(0) {}
    CPoint(int x, int y)
    {
        this->x=x;
        this->y=y;
    }
    friend CRectangle& CRectangle::operator()(CPoint lt, CPoint rb);
};
```

Если все методы какого-либо класса должны иметь доступ к скрытым полям другого, то весь класс объявляется дружественным

Дружественный класс

```
class CPoint
{
    int x,y;
public:
    CPoint() : x(0), y(0) {}
    CPoint(int x, int y)
    {
        this->x=x;
        this->y=y;
    }
    friend CRectangle;
};
```

Важно понимать, что класс сам определяет, какие функции и классы являются дружественными, а какие нет

Шаблоны функций

Шаблоны используются, когда необходимо создать функции, которые применяют один и тот же алгоритм к различным типам данных

Шаблон функции – это обобщенное описание функции

У такой функции хотя бы один формальный параметр имеет обобщенный тип

Определение шаблона функции

```
template <typename T> // или template <class T>
void Swap(T &a, T &b)
{
    T temp; // temp - переменная типа T
    temp = a;
    a = b;
    b = temp;
}
```

Использование шаблона функции

```
void main()
{
    int i = 10;
    int j = 20;
    cout << "i = " << i << ", j = " << j << "\n";
    // генерируется версия void Swap(int &, int &)
    Swap(i, j);
    cout << "i = " << i << ", j = " << j << "\n";
    double x = 27.5;
    double y = 77.7;
    cout << "x = " << x << ", y = " << y << "\n";
    // генерируется версия void Swap(double &, double &)
    Swap(x, y);
    cout << "x = " << x << ", y = " << y << "\n";
}
```

В момент вызова функции компилятор автоматически создает специализированную версию функции, где вместо параметра типа подставляется конкретный тип данных

Этот процесс называется инстанцированием шаблона или созданием экземпляра шаблона

Повторный вызов функции с теми же типами параметров не спровоцирует генерацию дополнительной копии функции, а вызовет уже существующую

Использование шаблона функции

```
void main()
{
    int i = 10;
    int j = 20;
    cout << "i = " << i << ", j = " << j << "\n";
    // генерируется версия void Swap(int &, int &)
    Swap(i, j);
    cout << "i = " << i << ", j = " << j << "\n";
    // вызывается существующая версия
    // void Swap(int &, int &)
    Swap(i, j);
    cout << "i = " << i << ", j = " << j << "\n";
}
```

Определение шаблона функции не вызывает самостоятельную генерацию кода компилятором

По этой причине реализацию шаблонов функций следует размещать в заголовочном файле

Компилятор создает код функции только в момент ее вызова, и генерирует при этом соответствующую версию функции

Каждый параметр типа, который встречается в угловых скобках, должен обязательно появиться в списке параметров функции

В противном случае произойдет ошибка этапа компиляции

Определение шаблона функции

```
template <typename T1, typename T2>
T1 Minimum(T1 a, T2 b)
{
    return (a < b ? a : b);
}
```

Перегруженные шаблоны

```
template <typename T>
void Swap(T &a, T &b)
{
    T temp;
    temp = a;
    a = b;
    b = temp;
}

template <typename T>
void Swap(T a[], T b[], int n)
{
    T temp;
    for (int i = 0; i < n; i++)
    {
        temp = a[i];
        a[i] = b[i];
        b[i] = temp;
    }
}
```

Явная специализация шаблона

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

template <typename T1, typename T2> T1 Minimum(T1 a, T2 b)
{
    return (a < b ? a : b);
}

template <> char* Minimum<char *>(char *a, char *b)
{
    return strcmp(a, b) <= 0 ? a : b;
}

void main()
{
    char* pWindows = "Windows";
    char* pAndroid = "Android";
    cout << "\nMinimum: " << Minimum(pAndroid, pWindows);
}
```

Специализация переопределяет шаблон, а обычная нешаблонная функция переопределяет и специализацию, и шаблон

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

template <typename T1, typename T2> T1 Minimum(T1 a, T2 b) { ... }

template <> char* Minimum<char *>(char *a, char *b) { ... }

char* Minimum(char *a, char *b) { ... }

void main()
{
    char* pWindows = "Windows";
    char* pAndroid = "Android";
    cout << "\nMinimum: " << Minimum(pAndroid, pWindows);
}
```

Важно понимать, что использование шаблонов функций не приводит к уменьшению результирующего объектного кода

Однако экономит время разработки программы

Динамические структуры данных

При решении большинства реальных современных задач для хранения данных приходится использовать динамическую память

До сих пор мы пользовались обычными динамическими массивами

При добавлении и удалении элементов массива приходилось перераспределять память

Динамический массив предполагает такой способ хранения информации, при котором можно получить произвольный доступ к любому элементу массива по индексу

Произвольный доступ к элементам является специфической особенностью массива как динамической структуры данных

Такой способ хранения информации является наиболее часто используемым, но не единственным

Динамическая структура данных – структура данных определенного формата с определенным способом доступа к ее элементам и автоматическим расширением размера при необходимости

Каждая динамическая структура данных имеет определенный набор операций с ее элементами

Наиболее часто используемые динамические структуры данных

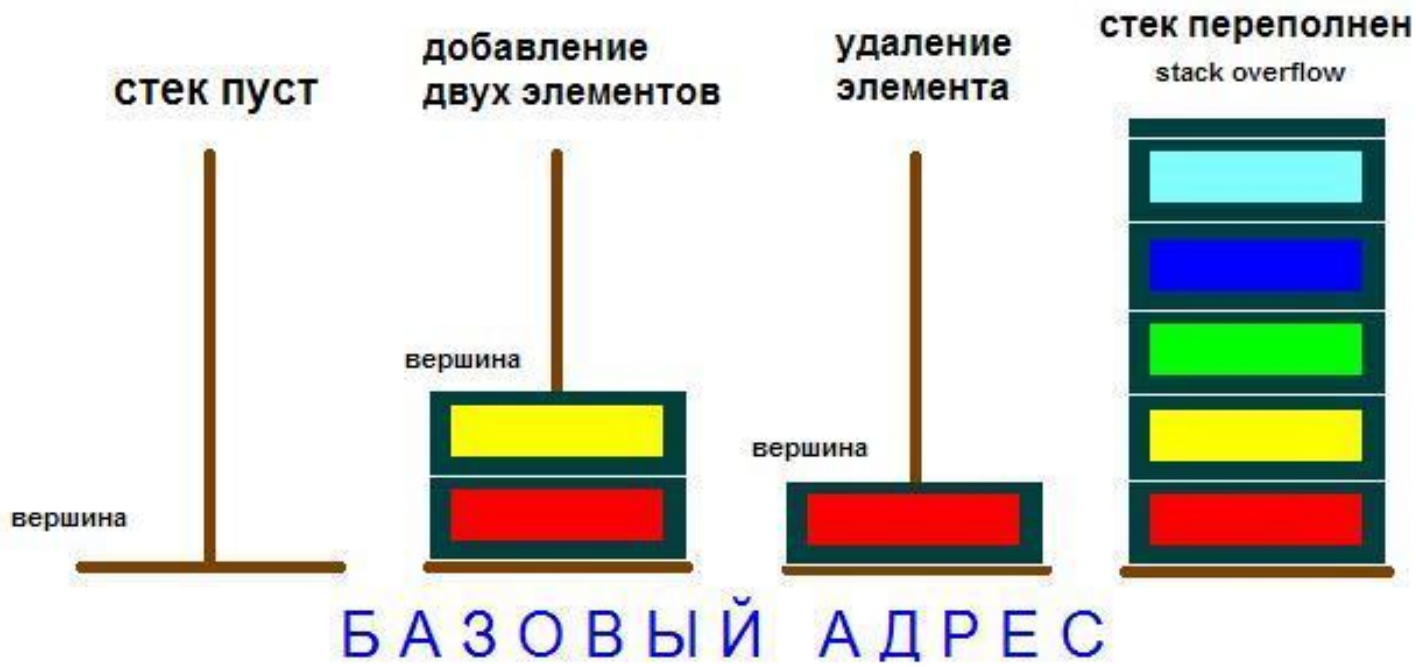
- Стек
- Очередь
- Связный список
- Бинарное дерево

Стек - это динамическая структура данных, которая функционирует по принципу LIFO (**L**ast **I**n **F**irst **O**ut)

Работа стека организована таким образом, что элементы добавляются и удаляются с одного конца, называемого **вершиной стека**

Кроме того, стек обладает **базовым адресом** - начальным адресом, по которому стек размещается в памяти

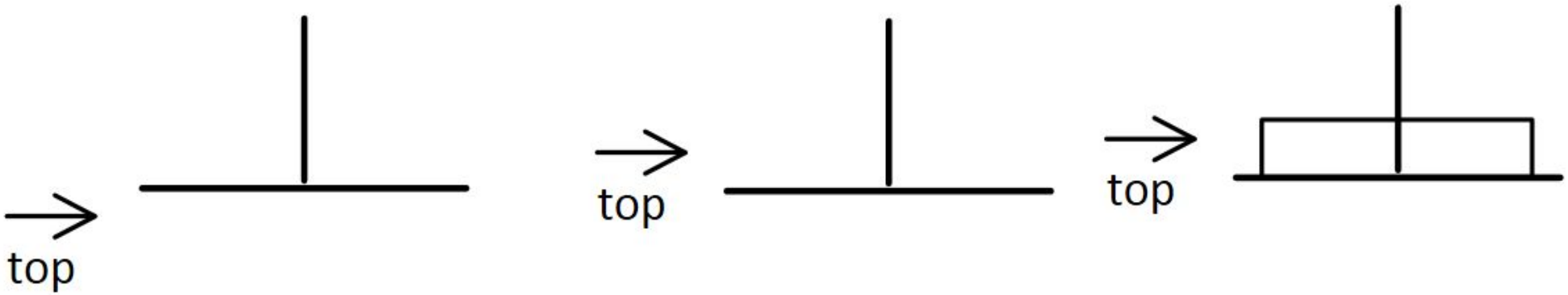
Динамическая структура данных «Стек»



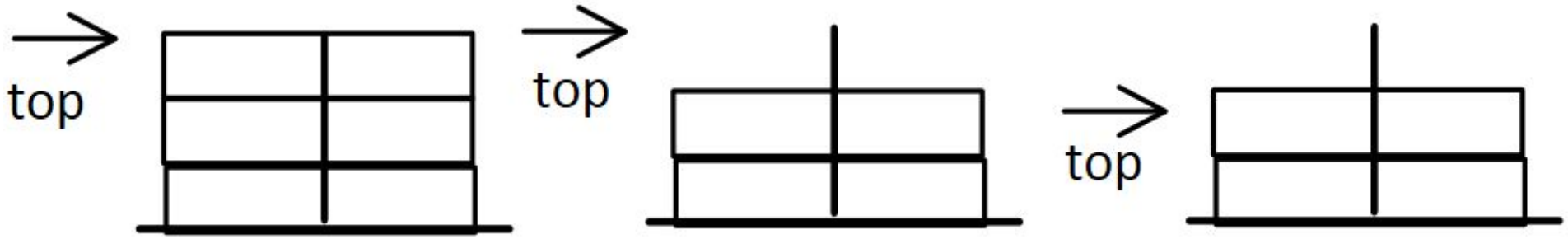
Основные операции над стеком и его элементами

- Добавление элемента в стек
- Удаление элемента из стека
- Просмотр элемента в вершине стека без удаления
- Очистка стека

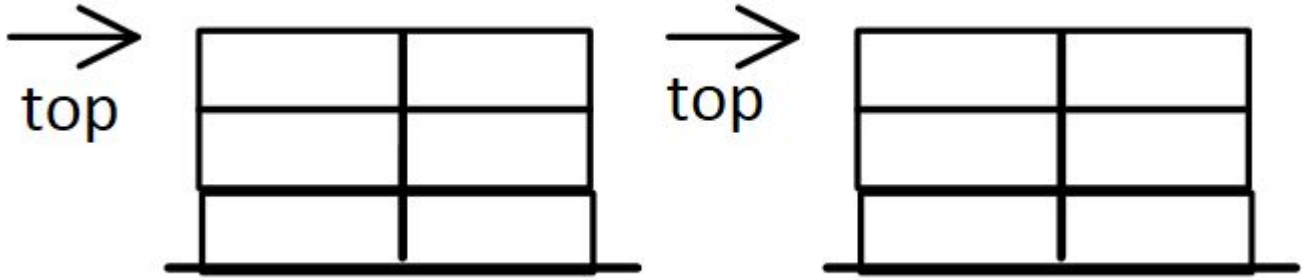
Добавление элемента в стек



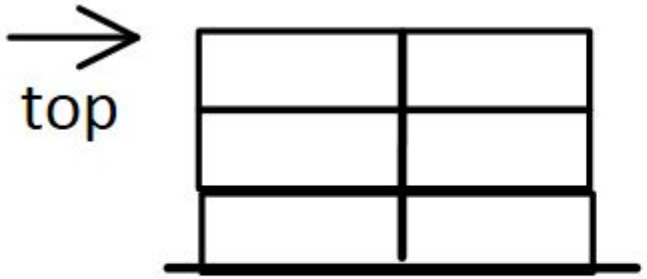
Удаление элемента из стека

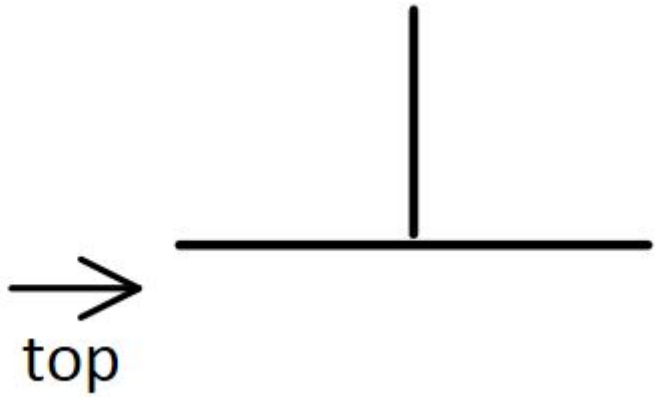


Просмотр элемента в вершине стека без удаления



Очистка стека





Важно понимать, что стек не определяет новый способ хранения данных в памяти, а предоставляет новый способ доступа к данным

Стек широко используются в системном программном обеспечении, включая компиляторы и интерпретаторы

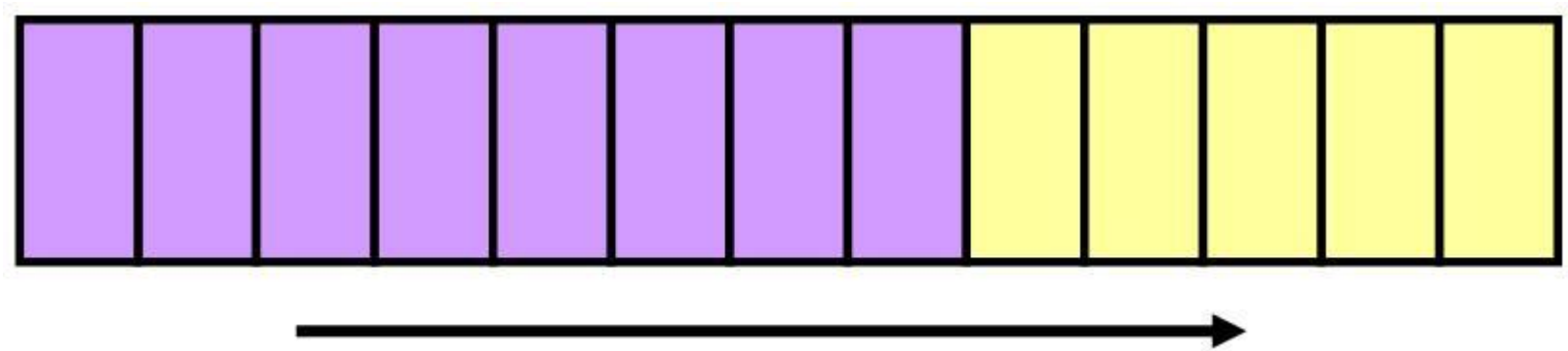
Очередь - это динамическая структура данных, которая функционирует по принципу FIFO (**F**irst **I**n **F**irst **O**ut)

Очередь чаще всего используется в задачах моделирования различных систем обслуживания

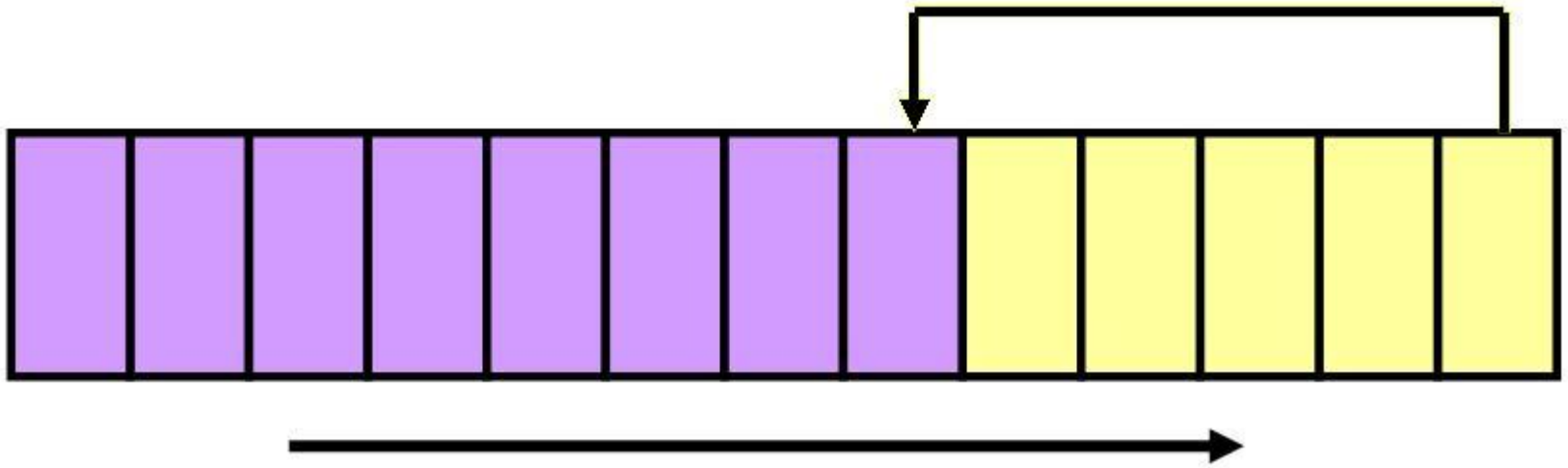
Разновидности очереди

- Классическая очередь
- Кольцевая очередь
- Очередь с приоритетом

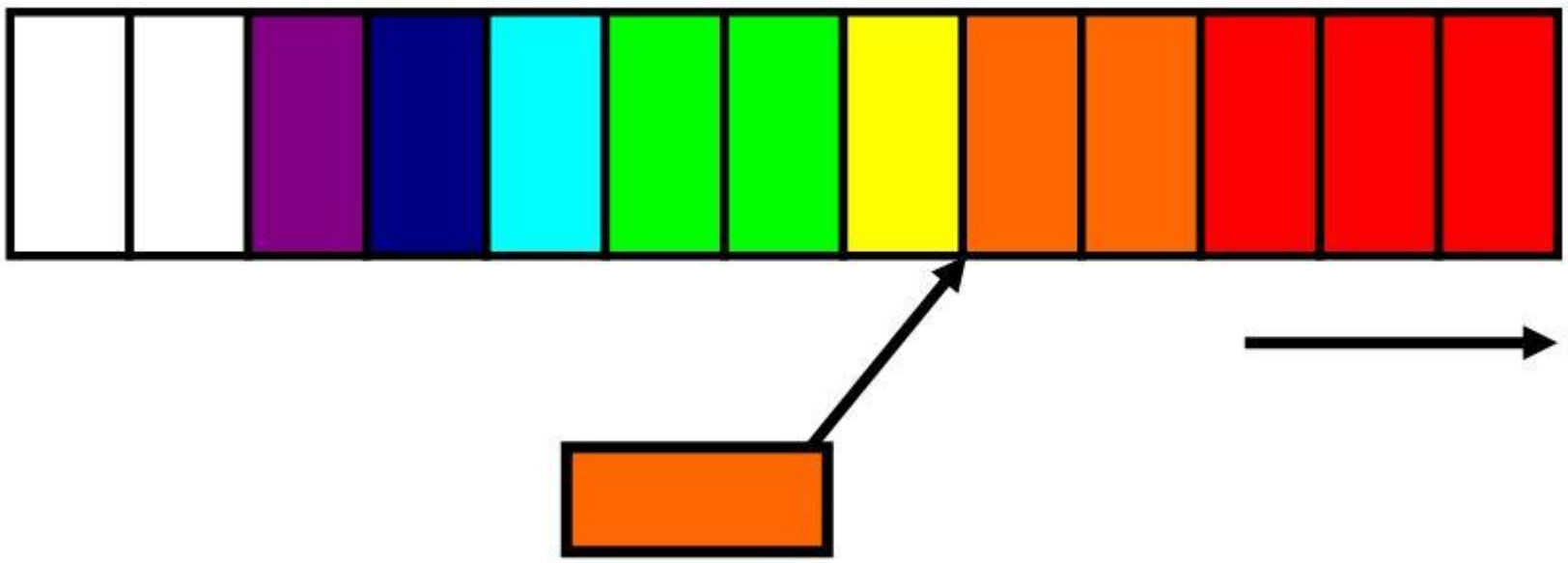
Классическая очередь



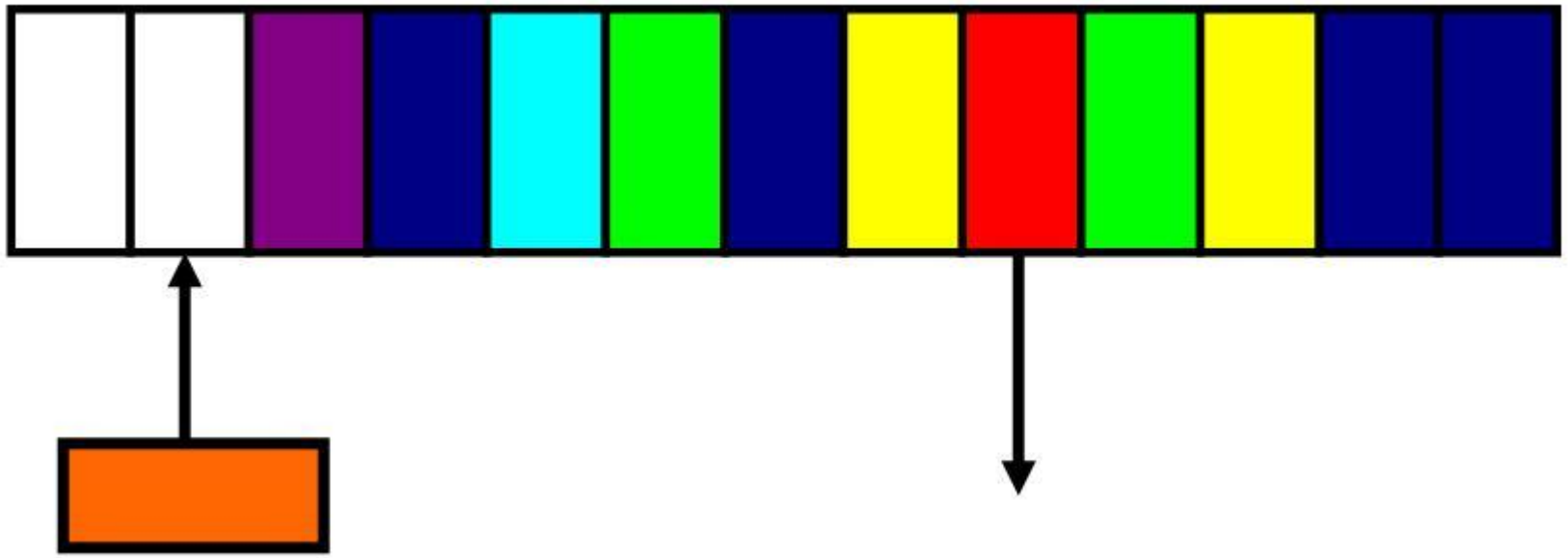
Кольцевая очередь



Очередь с приоритетом



Очередь с приоритетом



Важно понимать, что очередь не определяет новый способ хранения данных в памяти, а предоставляет новый способ доступа к данным

Односвязный список

Основной способ хранения информации, который используется в программах - это массивы

При использовании массивов наиболее эффективно выполняются операции доступа к элементам (чтение/изменение)

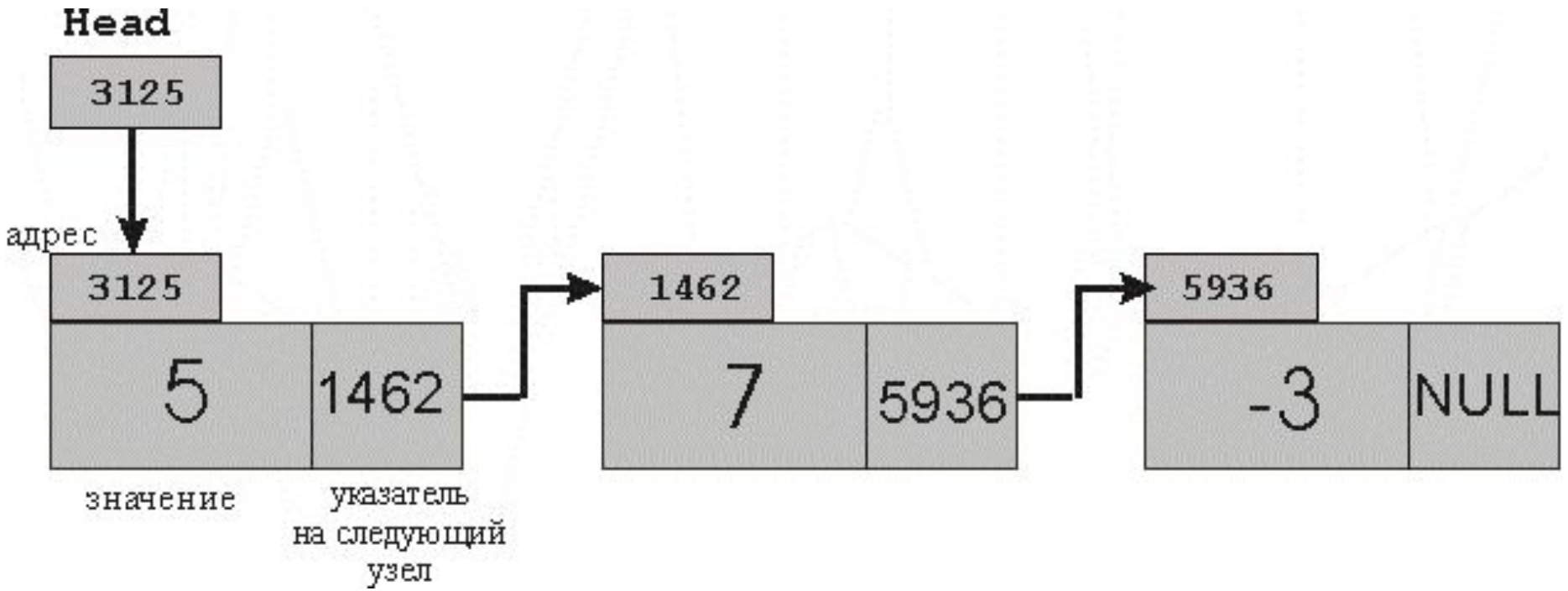
В то время как операции вставки/удаления будут значительно менее эффективны

В задачах, где операции вставки/удаления используются намного чаще, чем операции чтения/изменения, использование массивов оказывается неэффективным

Для решения этой проблемы используются такие динамические структуры данных как **связные списки**

Односвязный список — это набор элементов, связанных между собой с помощью указателя-связки

Односвязный список



Принципиальное отличие списка от массива заключается в том, что элементы списка хранятся не одним блоком в памяти, а каждый отдельно

Список вынужден хранить не только сами данные, но дополнительную служебную информацию

Такой служебной информацией в списке является **указатель на следующий элемент** (next)

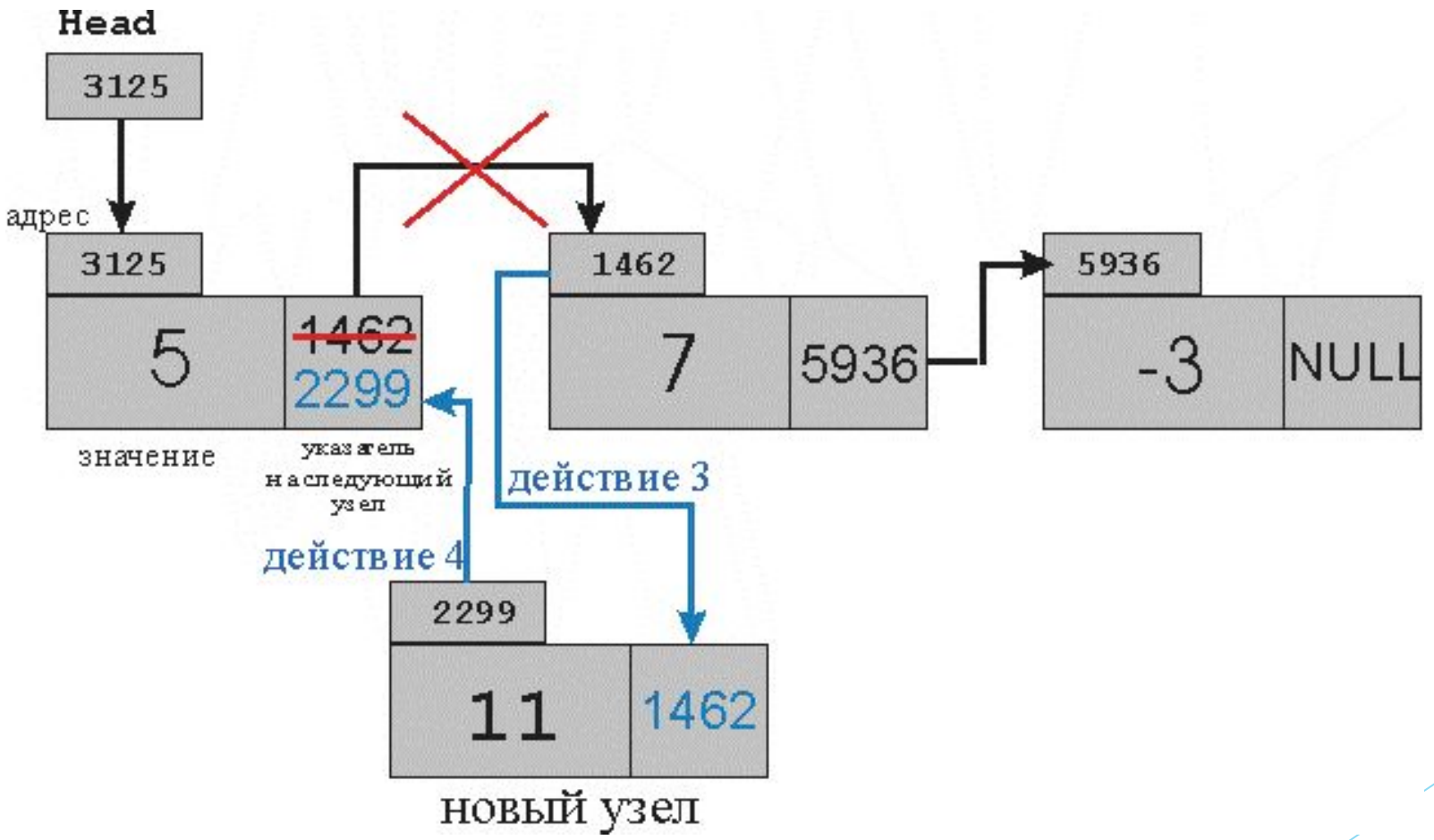
```
template <typename T>
struct ListItem
{
    T item;
    ListItem *next;
};
```

Этот указатель связывает хранящиеся отдельно в памяти элементы списка в единую динамическую структуру

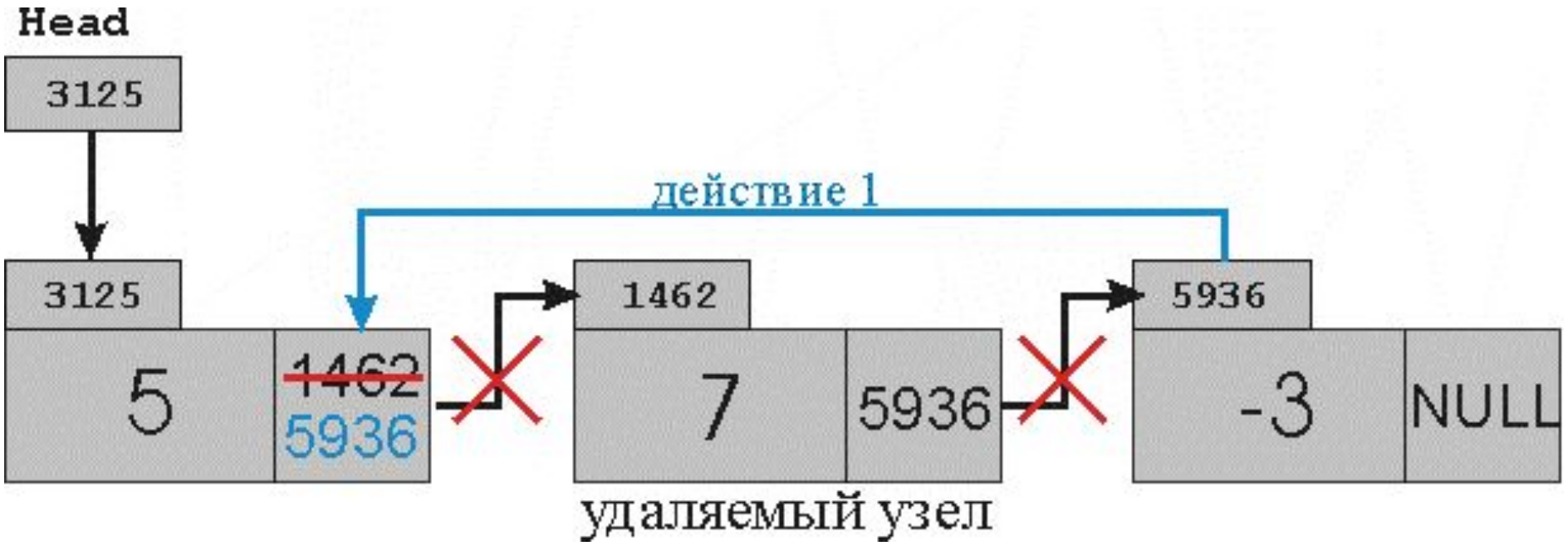
Связь между элементами списка однонаправленная, поэтому список называется **односвязным**

В качестве точки входа в список используется указатель на его первый элемент, называемый **головой списка (head)**

Вставка узла в определенное место списка



Удаление узла из списка



Именно благодаря такой структуре списка, в нем весьма эффективно осуществляются операции вставки/удаления элементов

Однако, скорость доступа к элементам у списка ниже, чем у массива

Двусвязный список

Недостаток односвязного списка – однонаправленная связь между его элементами

Из первого элемента списка попасть во второй можно, а обратно уже нет

Вместо этого придется заново перебирать все элементы, начиная с головы списка, что приводит к потере производительности

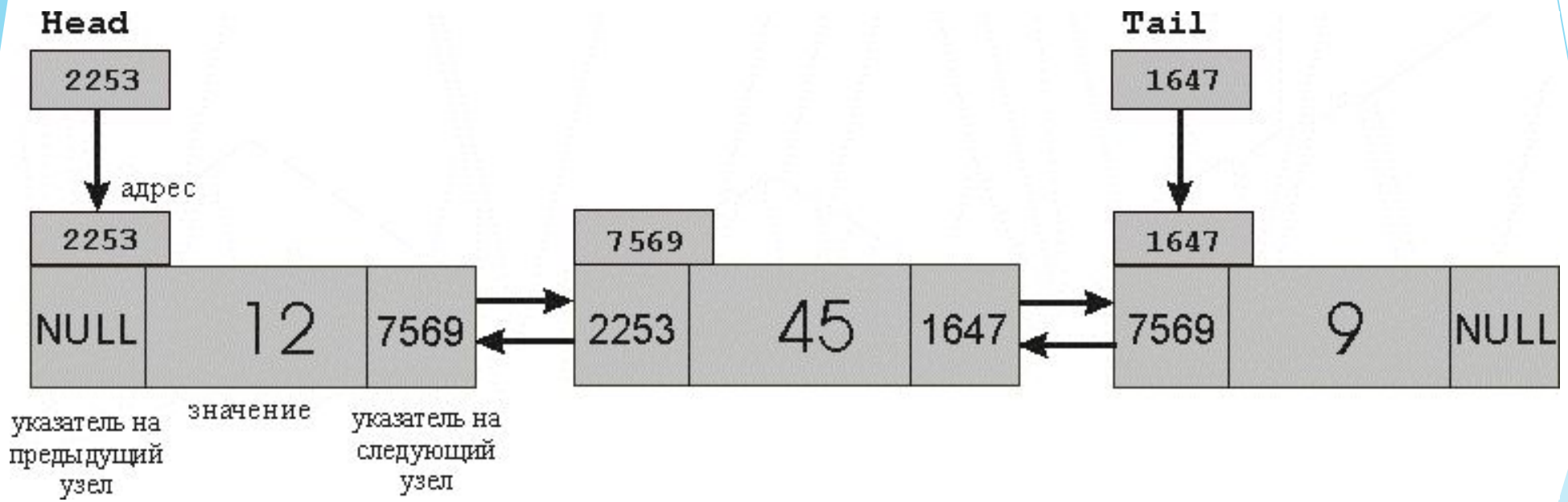
Для решения этой проблемы следует добавить в каждый элемент списка еще одно служебное поле

В этом поле будет храниться **указатель на предыдущий элемент** (previous)

```
template <typename T>
struct ListItem
{
    T item;
    ListItem *next;
    ListItem *prev;
};
```

Тогда из любого элемента списка можно будет попасть не только в следующий за ним элемент в списке, но и в предыдущий

Двусвязный список

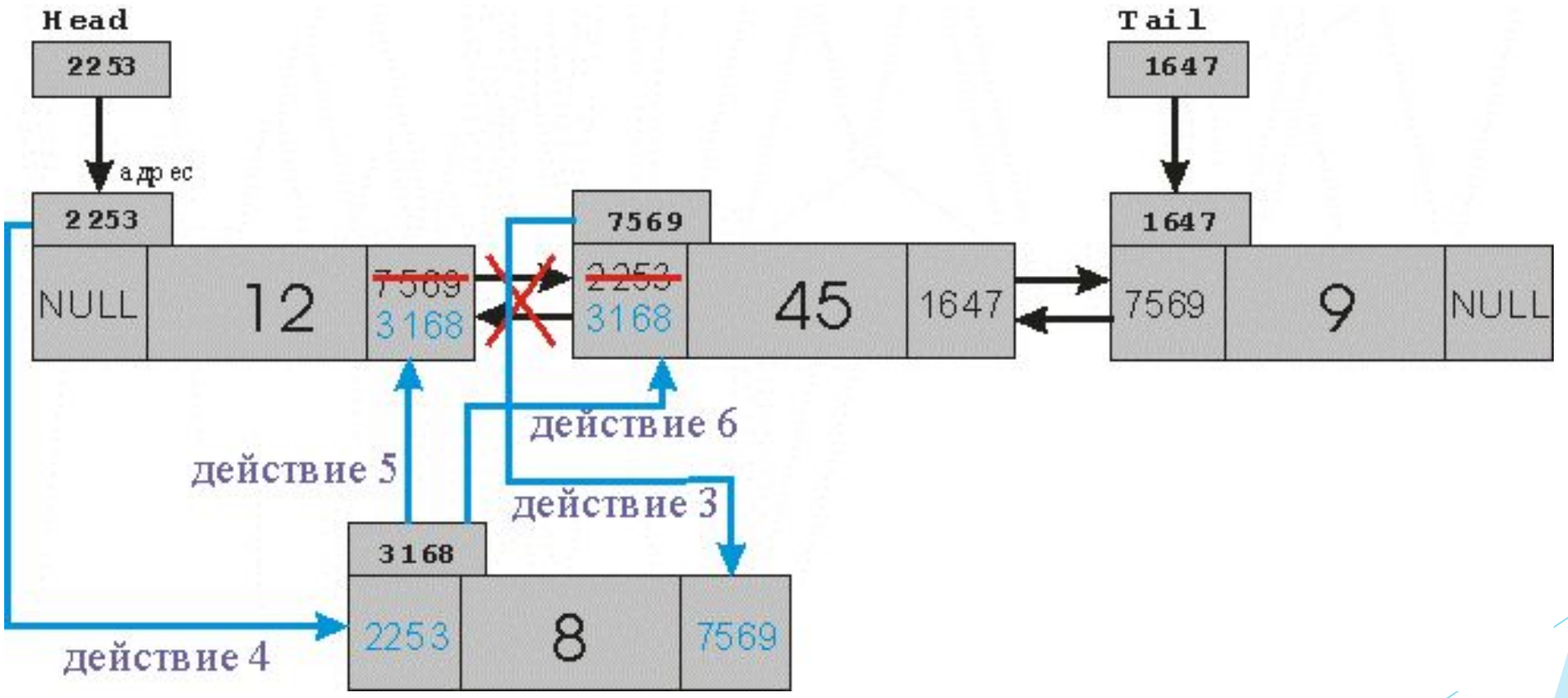


Такая структура предоставляет возможность двунаправленно перебирать список

Первый элемент двусвязного списка называется **головой списка (head)**

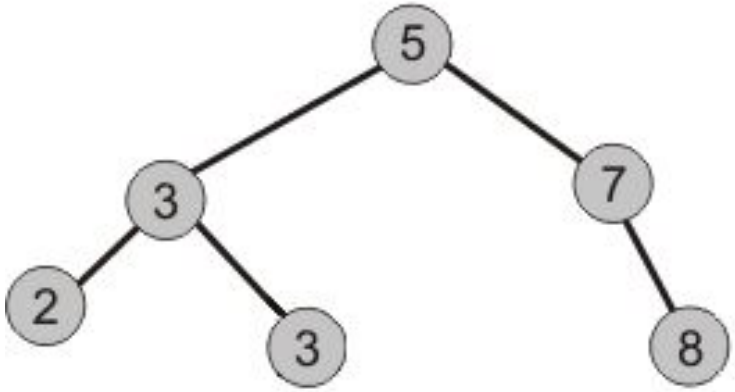
Последний элемент двусвязного списка называется **хвостом списка (tail)**

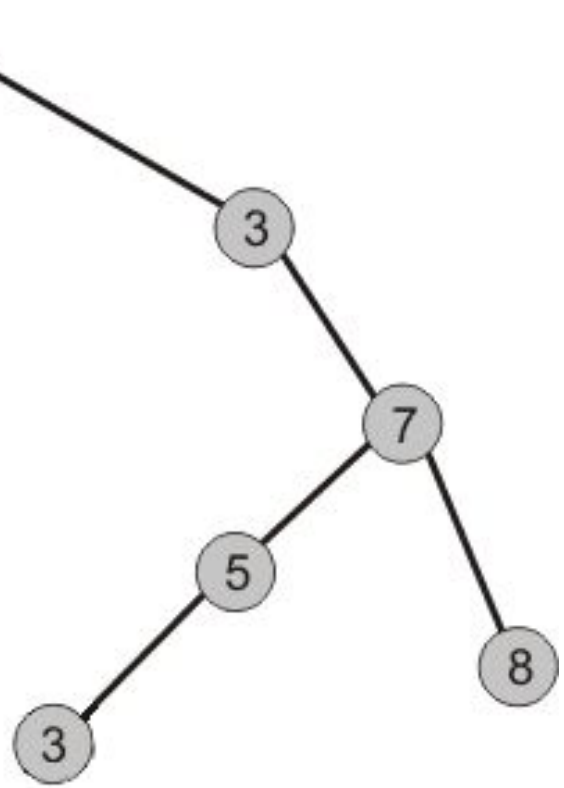
Вставка узла в определенное место двусвязного списка



Удаление узла из двусвязного списка

Бинарные деревья поиска





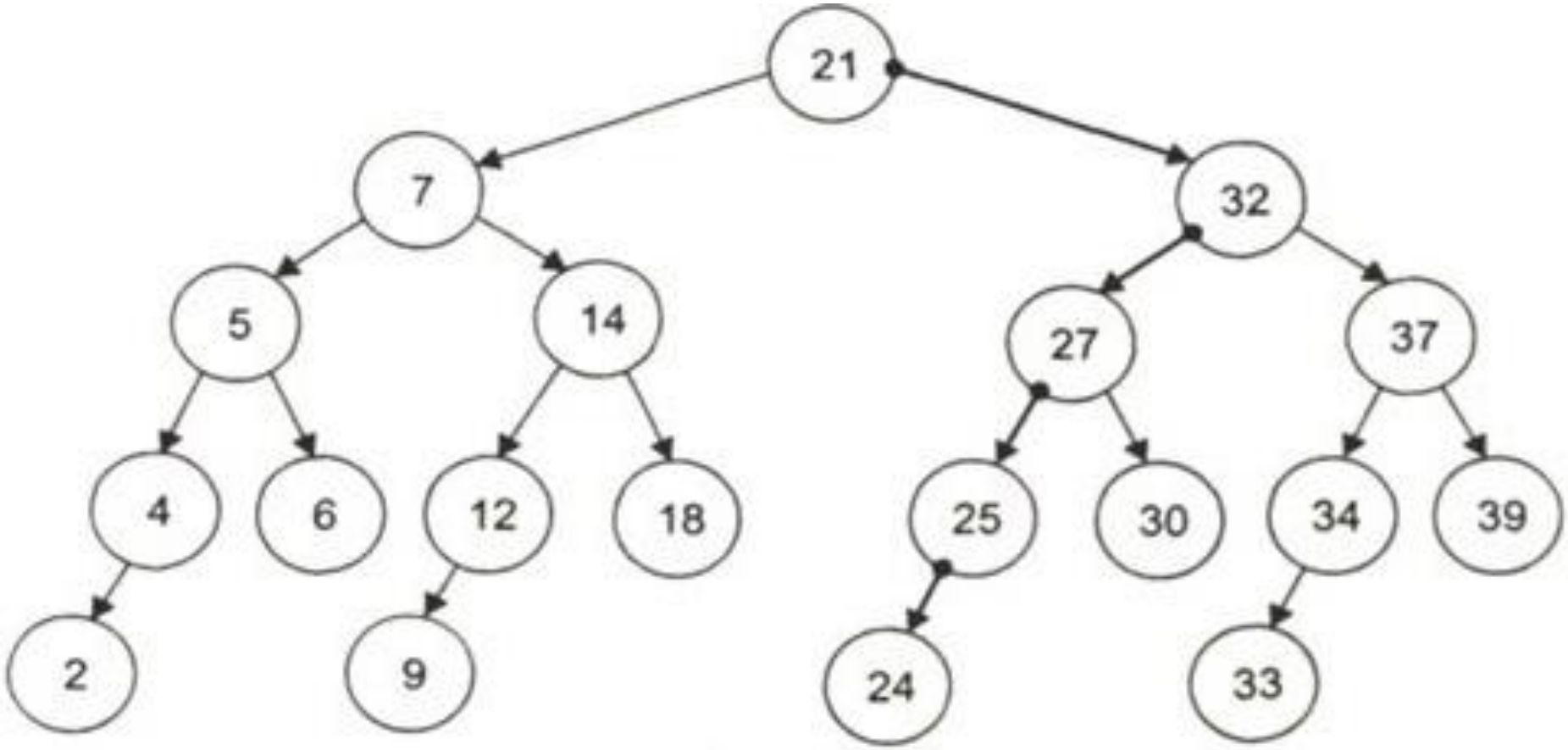
Бинарное дерево – это структурированная совокупность узлов, каждый из которых может иметь одного предка и двух потомков

Под **предком** понимают родительский узел, а **потомком** называют дочерний узел

Бинарное дерево – это динамическая структура данных, в которой элементы изначально упорядочены

Каждый узел бинарного дерева помимо данных имеет **ключ**, который однозначно идентифицирует узел

Бинарное дерево поиска

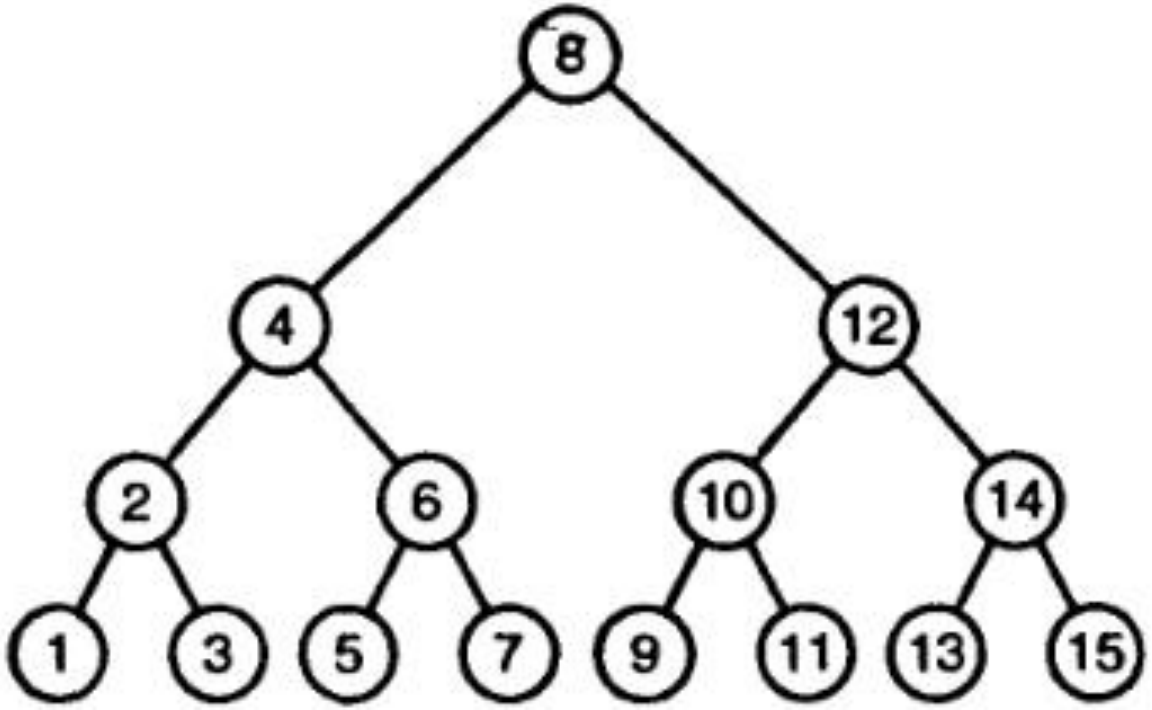


Для каждого узла бинарного дерева в левой ветке содержатся только те ключи, которые имеют значения, меньшие, чем значение данного узла

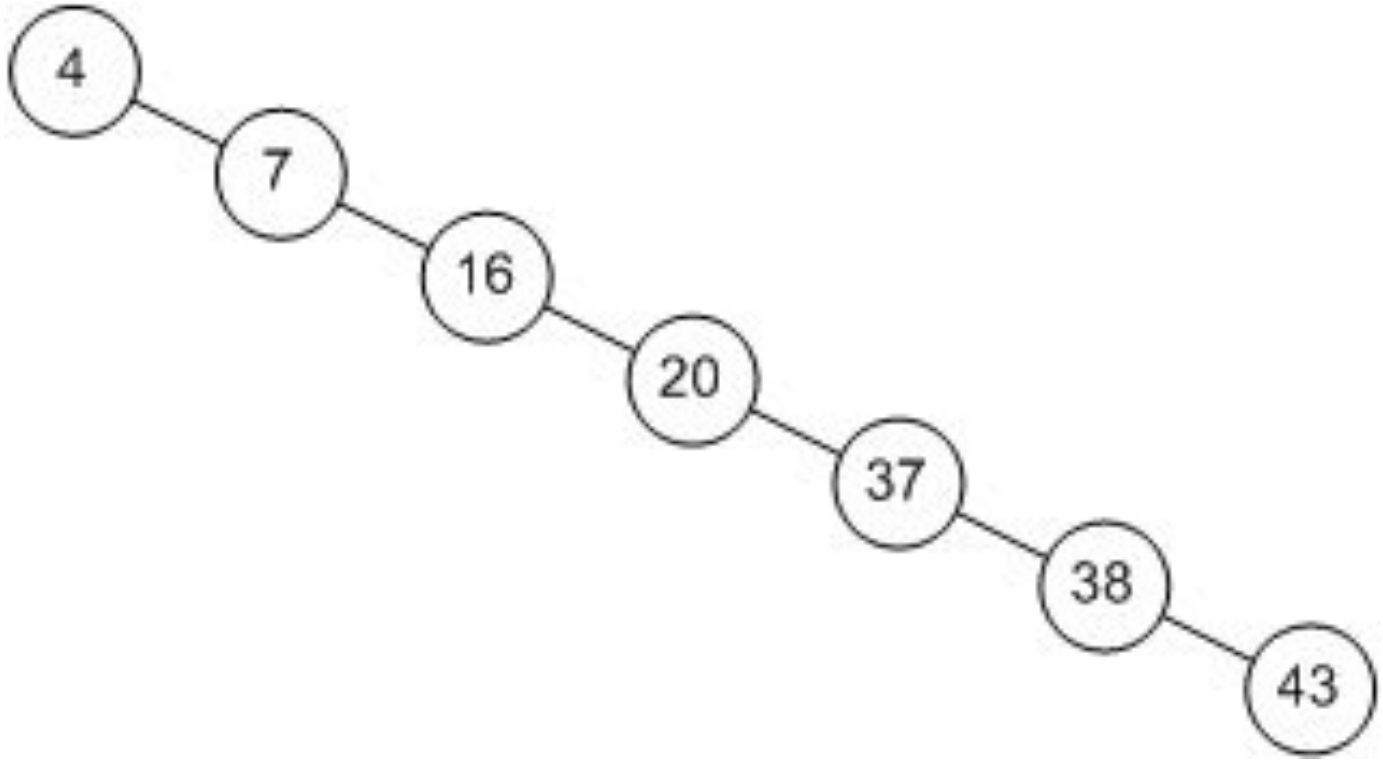
Для каждого узла в правой ветке содержатся ключи, имеющие значения, большие (или равные), чем значение данного узла

Бинарное дерево эффективно используется для поиска информации

Однако, в этом случае бинарное дерево должно быть сбалансированным – расти в ширину, а не в высоту



Если бинарное дерево не сбалансировано, то поиск данных будет достаточно долгим

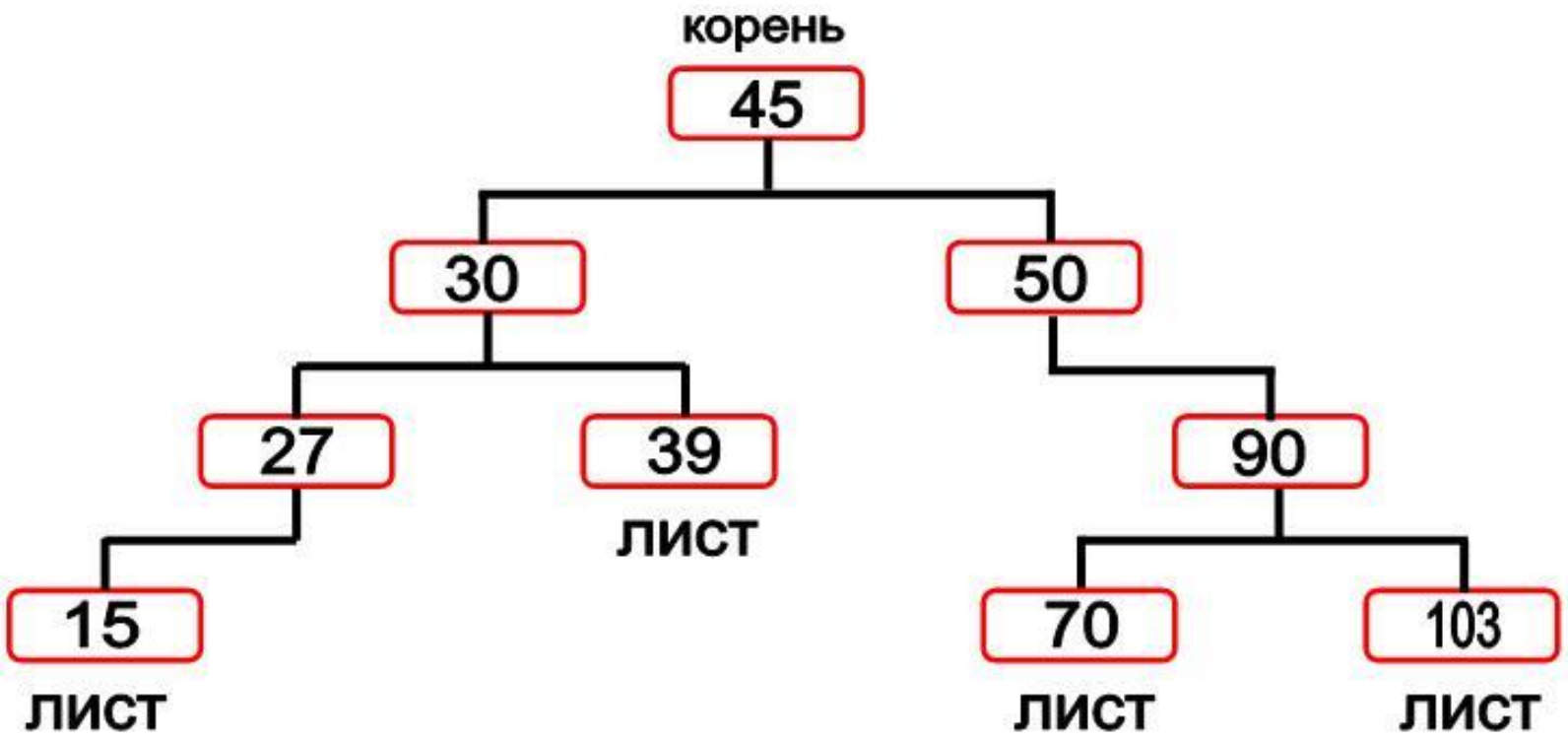


В отличие от массива бинарное дерево хранит отсортированную информацию не в линейном виде, а в иерархическом

Узел, который не имеет предка, является **корнем** бинарного дерева

Узел, который не имеет потомков, называется **листом**

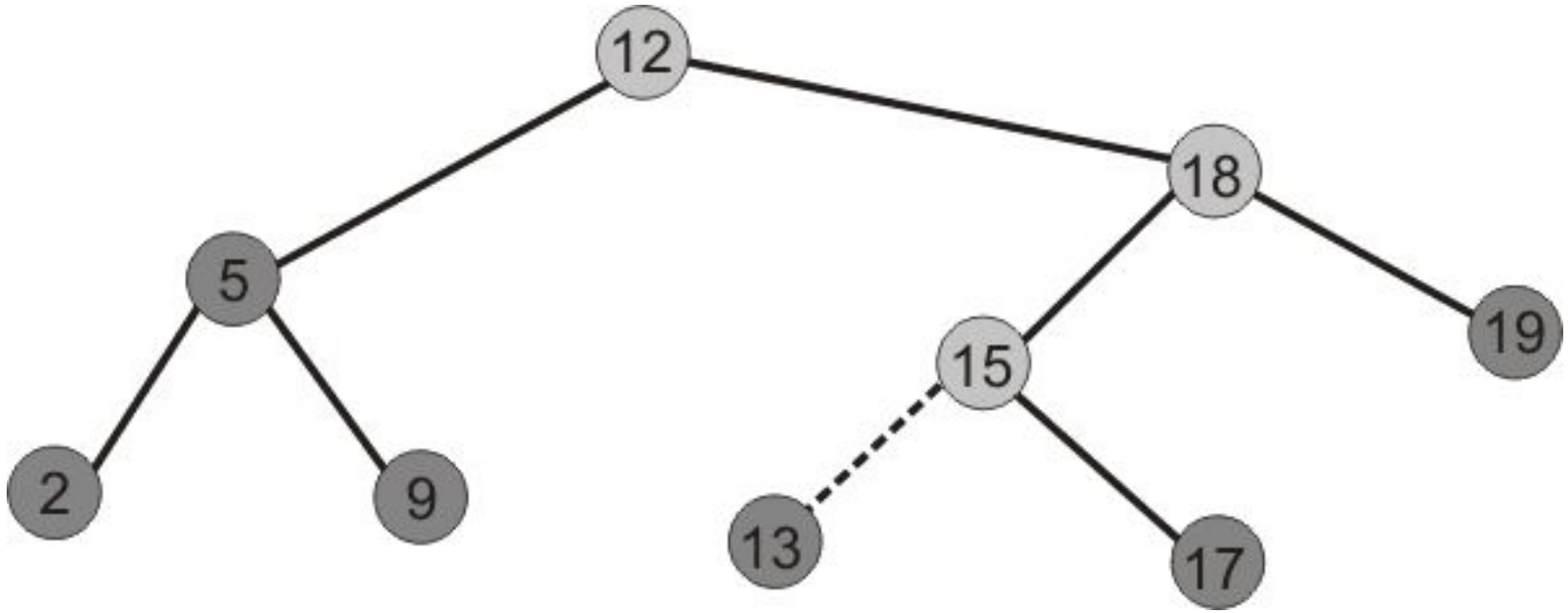
Бинарное дерево поиска



Структура узла бинарного дерева

```
template <typename T>
struct TreeItem
{
    T item;
    int key;
    TreeItem *parent, *left, *right;
};
```

Добавление узла в бинарное дерево



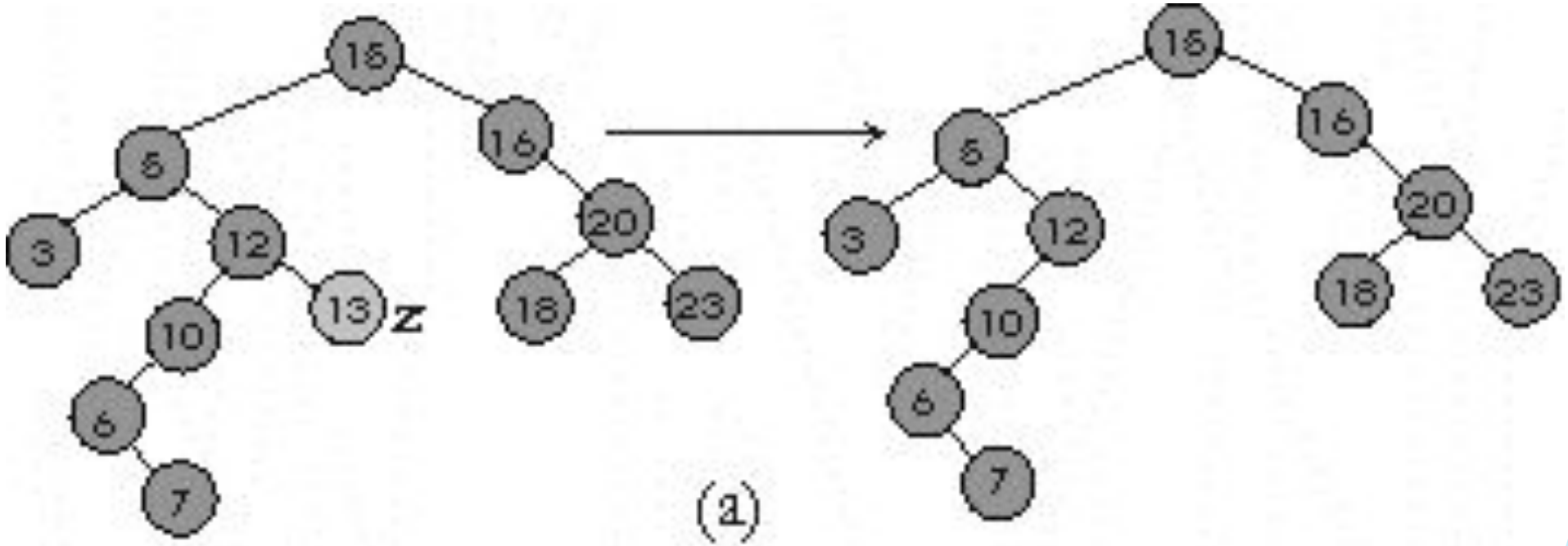
При добавлении узел вставляется в дерево таким образом, что дерево не нужно специально сортировать

Узел сразу вставляется в «правильную» позицию, причем такая позиция единственная

При удалении узла из дерева возможны три ситуации

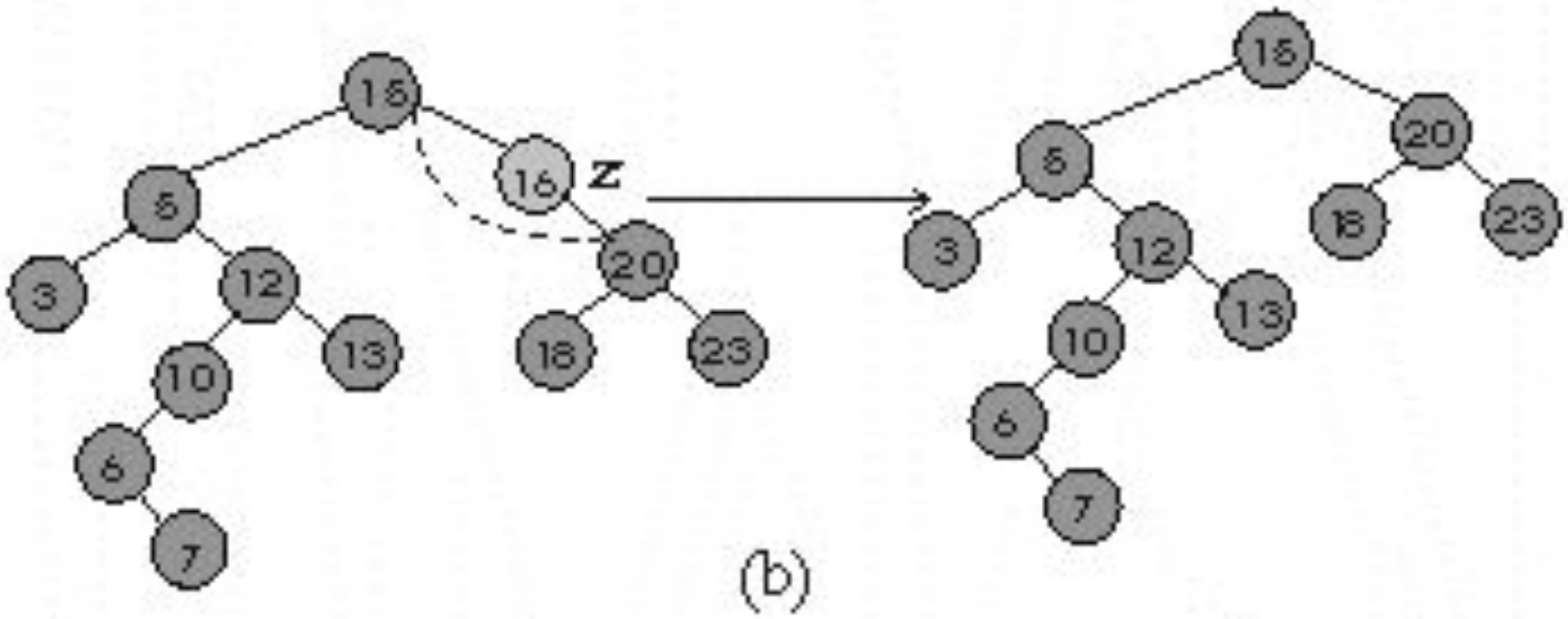
Если удаляемый узел дерева является листом, то удаление такого узла является тривиальной задачей

Удаление узла из дерева



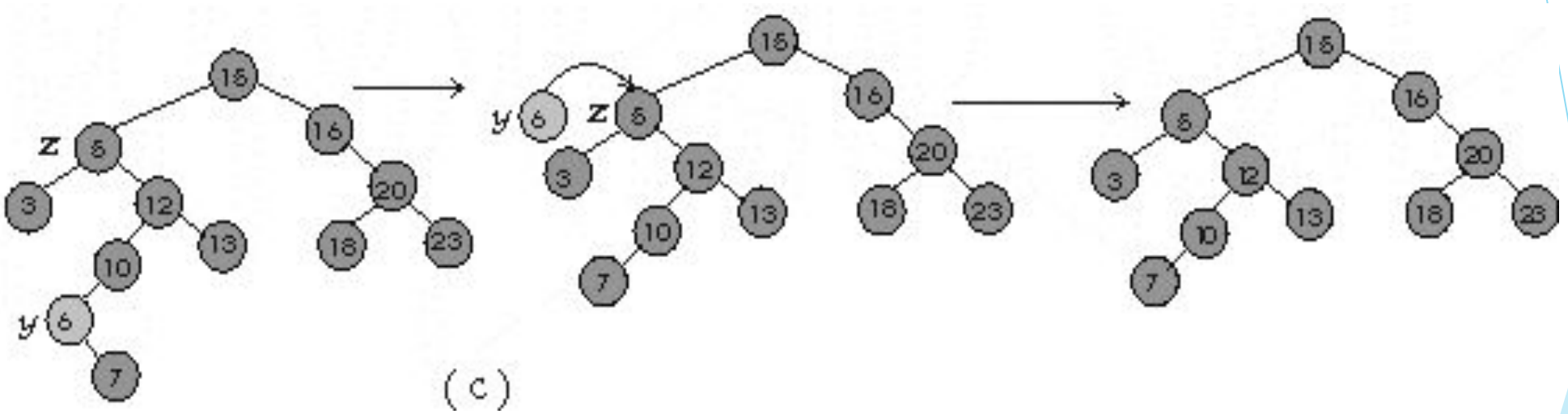
Если удаляемый узел имеет только одного потомка, тогда дочерний узел напрямую соединяется с родительским узлом

Удаление узла из дерева



Если удаляемый узел имеет двух потомков, тогда отыскивается следующий узел дерева и данные из него копируются в удаляемый узел, после чего следующий узел удаляется из дерева

Удаление узла из дерева



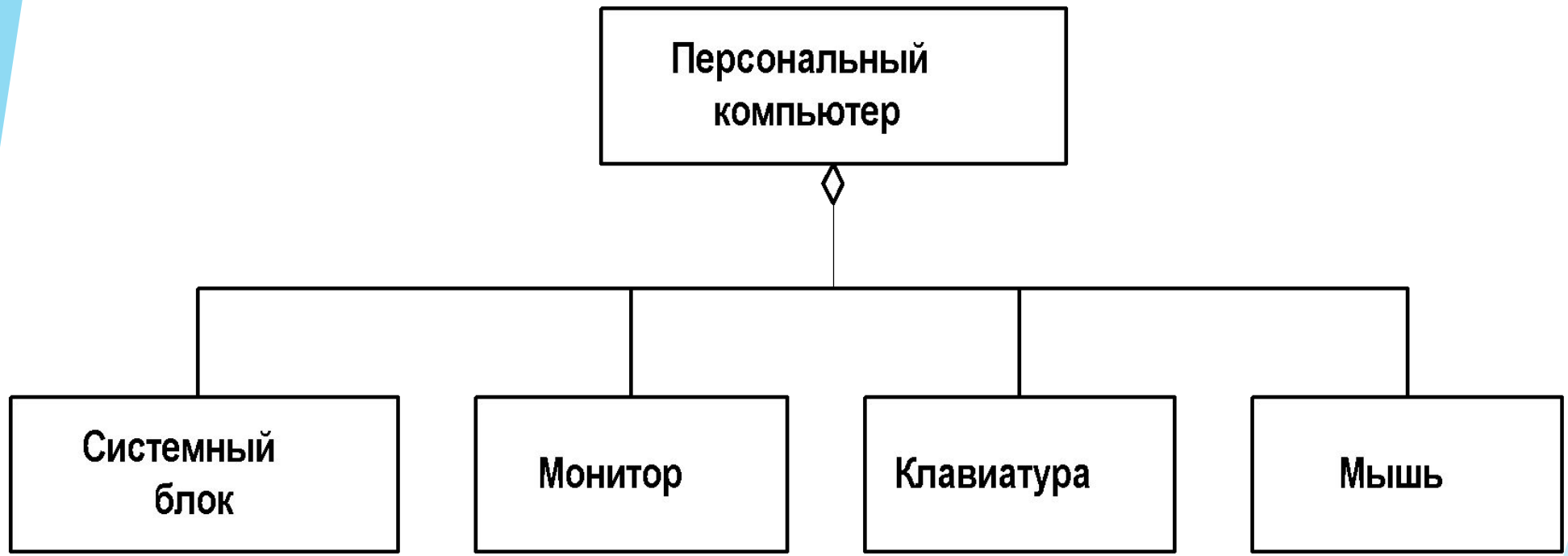
Агрегация и композиция

Агрегирование (агрегация) – это включение объекта (объектов) одного класса в состав объекта другого класса

Агрегация — отношение между двумя равноправными объектами, при котором один объект (контейнер) имеет ссылку на другой объект

Оба объекта могут существовать независимо: если контейнер будет уничтожен, то его содержимое — нет

Пример отношения агрегации



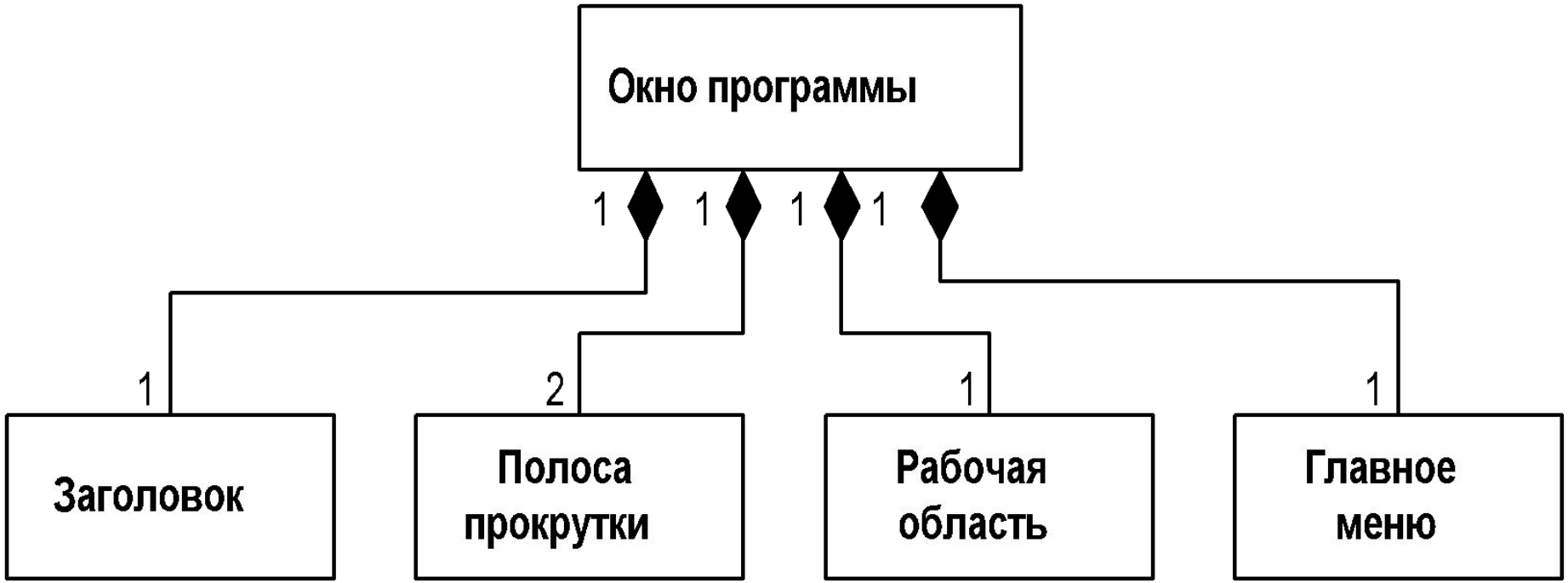
Пример отношения агрегации

```
class Professor;
class Department
{
private:
    Professor* members[5];
};
```

Композиция — более строгий вариант агрегации, когда включаемый объект может существовать только как часть контейнера

Если контейнер будет уничтожен, то и включённый объект тоже будет уничтожен

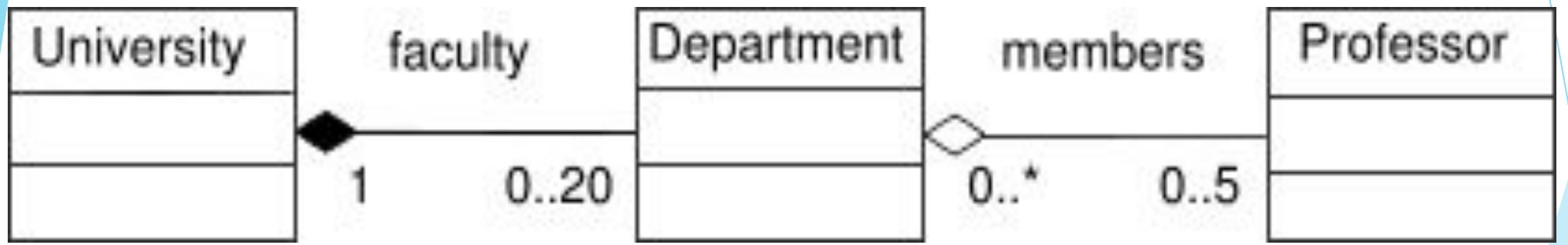
Пример отношения композиции



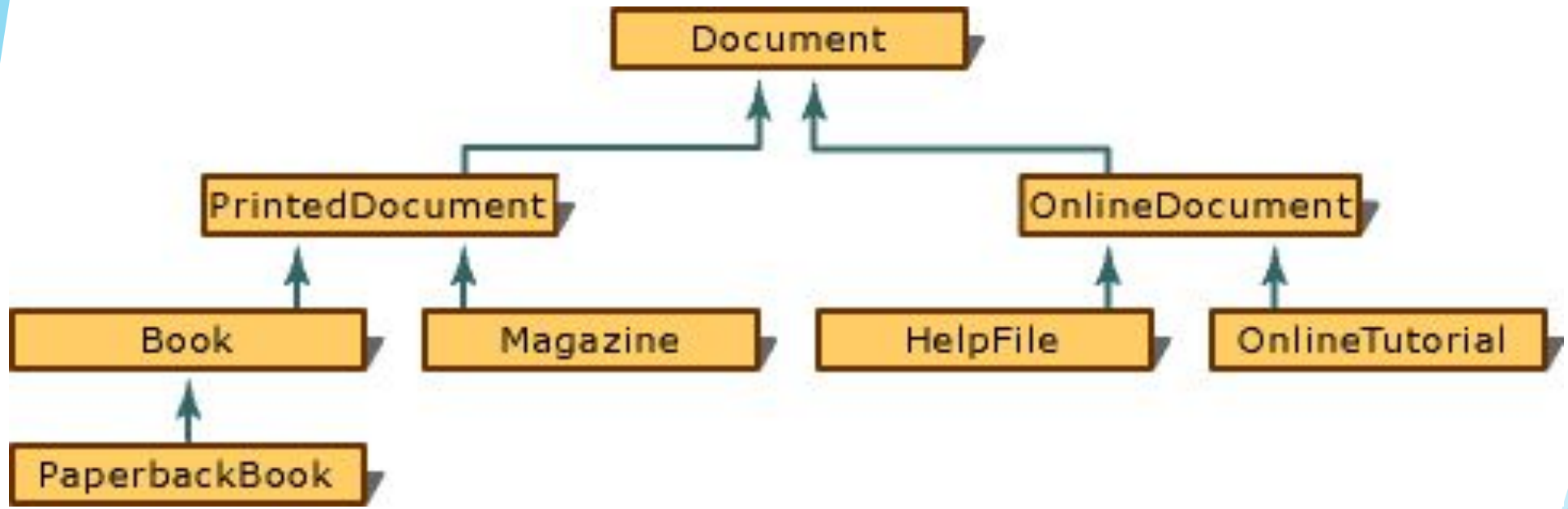
Пример отношения композиции

```
class Department;

class University
{
private:
    Department faculty[20];
};
```

Пример отношения агрегации и композиции



Наследование



Наследование - механизм языка C++, который позволяет расширять существующие классы, сохраняя их функциональность и добавляя им новые свойства и методы

Класс, от которого наследуются называют **базовым классом**

Класс, который наследует некоторый класс называют **классом-наследником** или **производным классом**

Таким образом, наследование позволяет **повторно использовать** уже написанный **код**, расширяя и дополняя его при необходимости новыми функциональными возможностями

Общий синтаксис наследования

class <имя производного класса>:
[спецификатор наследования]
<имя базового класса>
{
<элементы класса>
};

[спецификатор наследования] =
public | protected | private

Спецификатор наследования определяет каким будет в производном классе доступ к полям базового класса

Если спецификатор наследования не указан, то по умолчанию для классов будет **private**, а для структур - **public**

Пример наследования

```
class Person
{
    char name[15];
    char surname[15];
    int age;
    char address[20];
public:
    // методы
};

class Student: public Person
{
    double GPA;
    char group[5];
public:
    // Методы
};
```

Спецификаторы доступа

Спецификатор доступа в базовом классе	Спецификатор наследования	Доступ в производном классе
public	public	public
protected		protected
private		нет доступа
public	protected	protected
protected		protected
private		нет доступа
public	private	private
protected		private
private		нет доступа

Важно понимать, что все унаследованные private-поля хотя и недоступны непосредственно в объектах производного класса, но всё равно содержатся в них

Конструкторы и деструкторы не наследуются

При создании объекта производного класса вызываются конструкторы всех классов иерархии, начиная с самого верхнего и заканчивая конструктором этого производного класса

Деструкторы, соответственно, вызываются в обратном порядке

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

class A
{
public:
    A() {   cout << "Constructor A\n";  }
    ~A(){   cout << "Destructor A\n";   }
};

class B :public A
{
public:
    B() {   cout << "Constructor B\n";  }
    ~B(){   cout << "Destructor B\n";   }
};
```

```
class C :public B
{
public:
    C() {   cout << "Constructor C\n";  }
    ~C(){   cout << "Destructor C\n";   }
};

void main()
{
    A obj1;
    B obj2;
    C obj3;
}
```

```
Constructor A
Constructor A
Constructor B
Constructor A
Constructor B
Constructor C
Destructor C
Destructor B
Destructor A
Destructor B
Destructor A
Destructor A
```

Для инициализации унаследованных полей базового класса в производном классе следует использовать конструктор базового класса

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

class A
{
    int a;
public:
    A()
    {
        cout << "Default constructor A\n";
        a = 0;
    }
    A(int n)
    {
        cout << "Constructor A with parameter\n";
        a = n;
    }
};
```

```
class B :public A
{
    int b;
public:
    B()
    {
        cout << "Default constructor B\n";
        b = 0;
    }
    B(int n, int m) :A(n)
    {
        cout << "Constructor B with parameters\n";
        b = m;
    }
};
```

```
class C :public B
{
    int c;
public:
    C(int n, int m, int k) :B(n, m)
    {
        cout << "Constructor C with parameters\n";
        c = k;
    }
};
```

```
class D :public C
{
    int d;
public:
    D() :C(0, 0, 0)
    {
        cout << "Default constructor D\n";
        d = 0;
    }
    D(int n, int m, int k, int l) :C(n, m, k)
    {
        cout << "Constructor D with parameters\n";
        d = l;
    }
};
```

```
void main()
{
    B obj1;
    B obj2(40, 50);
    cout << endl;
    C obj3; // ошибка компиляции
    // no appropriate default constructor available
    C obj4(10, 20, 30);
    cout << endl;
    D obj5;
    D obj6(1, 2, 3, 4);
}
```

В конструкторе производного класса следует явно инициализировать только новые поля, которые не были унаследованы от базового класса

При помощи списка инициализаторов мы сообщаем компилятору какой именно конструктор следует вызвать

Если необходимо изменить функциональность производного класса относительно базового класса, то используется **переопределение методов**

Если в производном классе имеется метод с таким же именем и сигнатурой, как и в базовом классе, такой метод считается **переопределённым**

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

class A
{
    int a;
public:
    void setA(int k)
    {
        a = k;
    }
    void show()
    {
        cout << a << endl;
    }
};
```

```
class B : public A
{
    int b;
public:
    void setB(int k)
    {
        b = k;
    }
    void show()
    {
        A::show();
        cout << b << endl;
    }
};
```

```
int main()
{
    A obj1;
    obj1.setA(10);
    obj1.show();//10

    B obj2;
    obj2.setA(20);
    obj2.setB(30);
    obj2.show(); //20  30 – метод класса B
    obj2.A::show(); // 20
    obj2.B::show(); //20  30

    return 0;
}
```

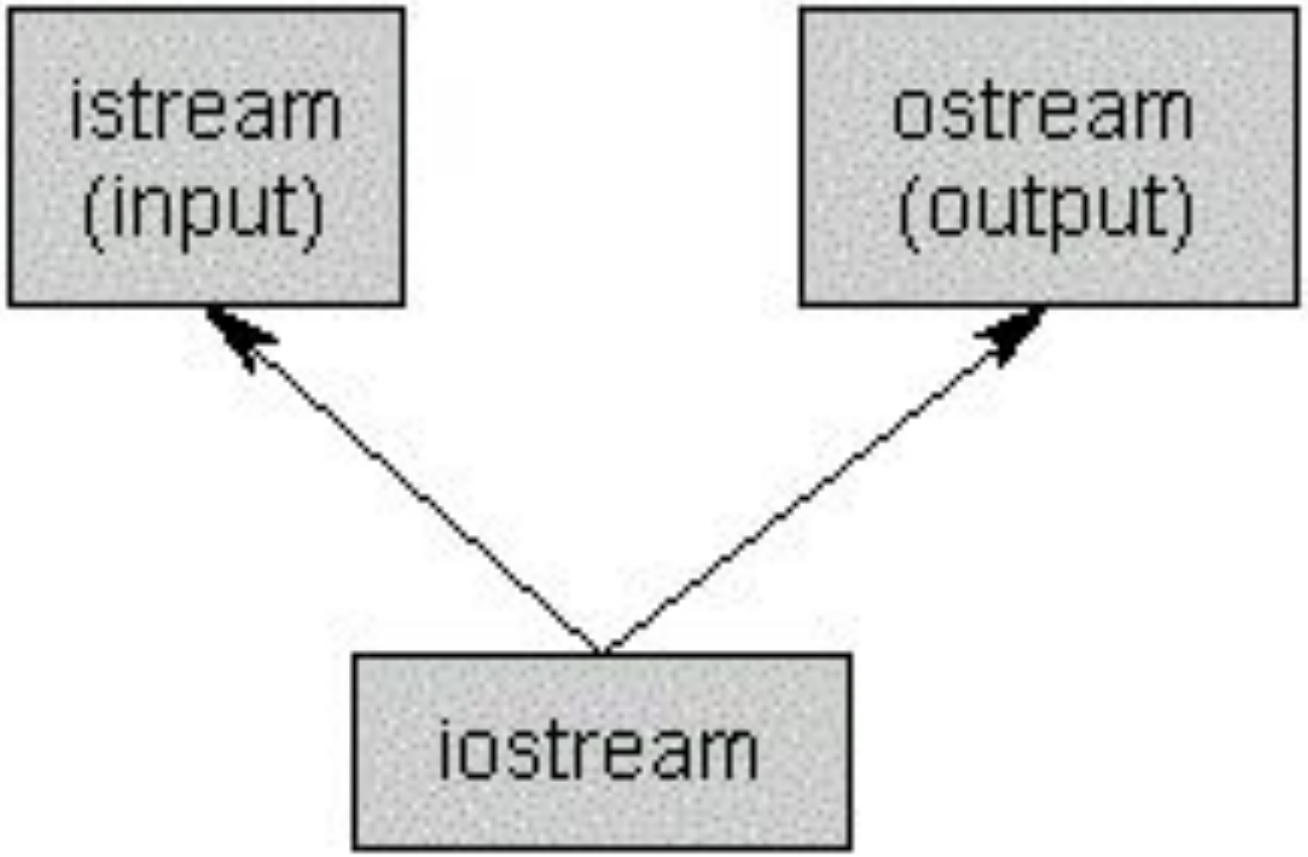
Если в производном классе имеется метод с точно таким же именем как и в базовом классе, но различной сигнатурой, то такой метод называется **замещённым**

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

class A
{
    int a;
public:
    void set()
    {
        a = 10;
    }
    void show()
    {
        cout << a << endl;
    }
};
```

```
class B : public A
{
    int b;
public:
    void set(int k)
    {
        A::set();
        b = k;
    }
    void show()
    {
        A::show();
        cout << b << endl;
    }
};
```

```
int main()
{
    A obj1;
    obj1.set();
    obj1.show(); // 10

    B obj2;
    // obj2.set(); //ошибка компиляции
    obj2.set(50); // obj2.B::set(50);
    obj2.show(); // 10  50
    obj2.A::set();
    obj2.B::set(30);
    obj2.show(); // 10  30
    return 0;
}
```

Множественное наследование



При множественном наследовании производный класс наследуется от нескольких базовых классов

Пример множественного наследования

```
class A { ... };

class B { ... };

class C : public B, public A
{
    int c;
public:
    C(int n, int m, int k) { ... }
    ~C() { ... }
    void ShowC() { ... }
};
```

При множественном наследовании порядок вызова конструкторов базовых классов определяется не списком инициализаторов, а порядком, в котором указаны базовые классы при объявлении производного класса

Порядок вызова конструкторов

```
class A { ... };

class B { ... };

class C : public B, public A
{
    int c;
public:
    C(int n, int m, int k) : A(n), B(m), c(k)
    {
        cout << "\nConstructor C\n";
    }
    ~C() { ... }
    void ShowC() { ... }
};
```

Деструкторы вызываются в порядке, обратном порядку вызова конструкторов

Главной проблемой множественного наследования являются **конфликты имен**

Конфликты имен

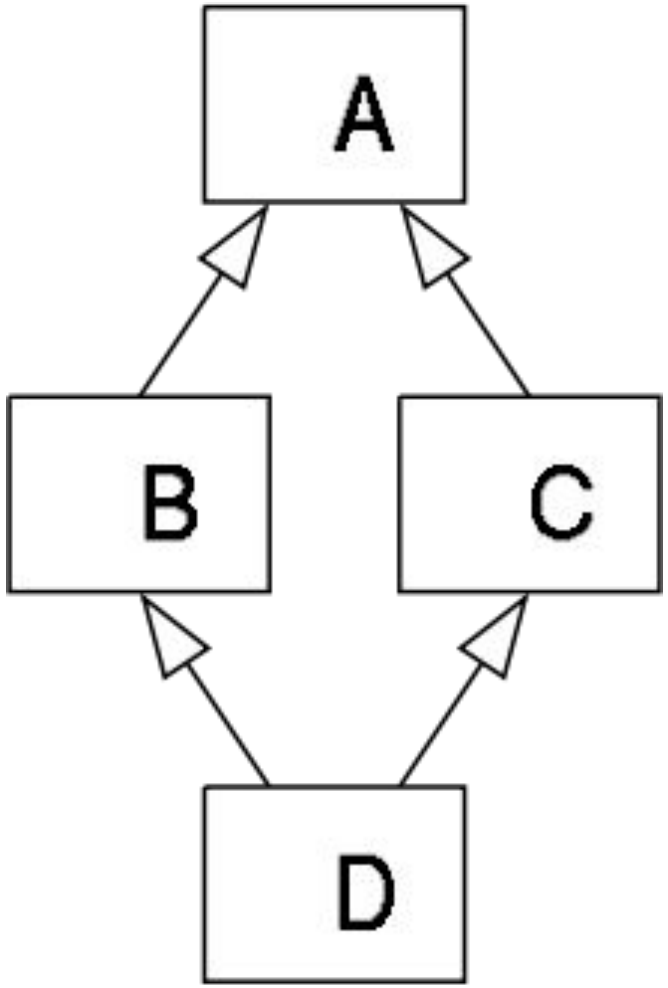
```
class A
{
public:
    int field;
    A() { field = 0; }
};

class B
{
public:
    int field;
    B() { field = 0; }
};

class C : public A, public B
{
public:
    C() { }
};
```

Для решения этой проблемы необходимо использовать операцию разрешения контекста для доступа к полям в производном классе

Решение проблемы конфликта имен

```
class A { ... };

class B { ... };

class C : public A, public B
{
public:
    C()
    {
        A::field = 1;
        B::field = 2;
    }
};
```

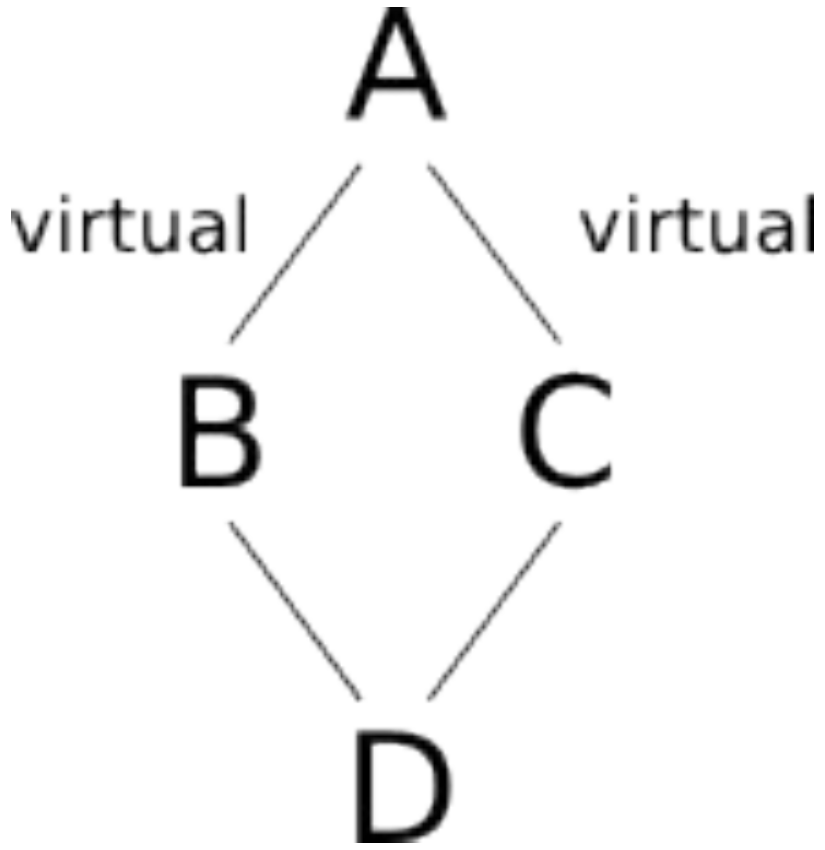
«Ромбовидное» наследование



Проблема «ромбовидного» наследования состоит в том, что в производном классе D дублируется поле из класса A

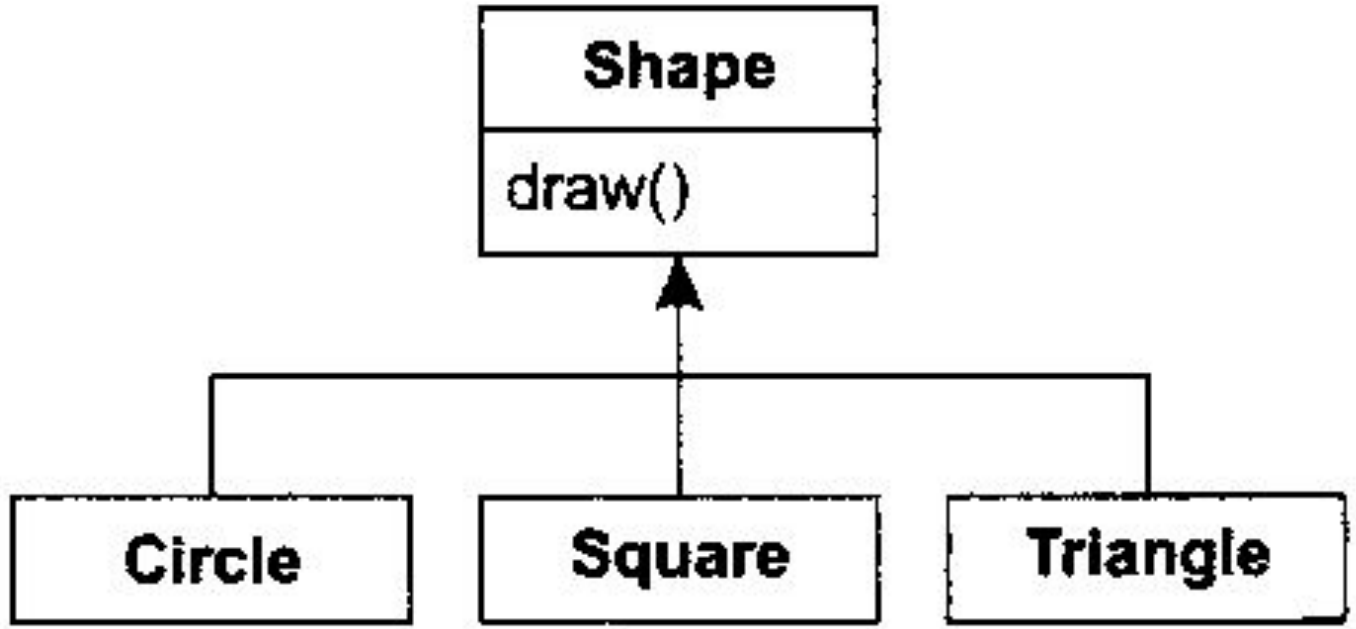
Для решения проблемы необходимо класс A сделать виртуальным базовым классом и обеспечить его конструктором по умолчанию

Решение проблемы «ромбовидного» наследования



При создании объекта производного класса в цепочке вызовов конструкторов сначала вызываются конструкторы по умолчанию виртуальных базовых классов, в порядке следования слева направо, а затем все остальные конструкторы

Полиморфизм



Механизм раннего связывания

```
class figure{
protected:
    double x, y;
public:
    void setDimension(double i, double j = 0) { ... }
    void calculateArea() { ... }
};

class triangle : public figure{
public: void calculateArea() { ... }
};

class rectangle : public figure{
public: void calculateArea() { ... }
};

class circle : public figure{
public: void calculateArea() { ... }
};
```

Механизм раннего связывания

```
void main()
{
    triangle t;
    rectangle r;
    circle c;
    t.setDimension(5.5, 10.0);
    t.calculateArea();
    r.setDimension(5.5, 10.0);
    r.calculateArea();
    c.setDimension(10.0);
    c.calculateArea();
}
```

Поскольку тип объектной переменной известен на этапе компиляции программы, то **связывание** вызова метода с самим кодом метода происходит **на этапе построения программы**

Такое связывание называется **ранним связыванием**

Механизм наследования позволяет записывать адрес объекта производного класса в указатель на базовый класс

```
void Print(figure *p)
{
    p->calculateArea();
}
```

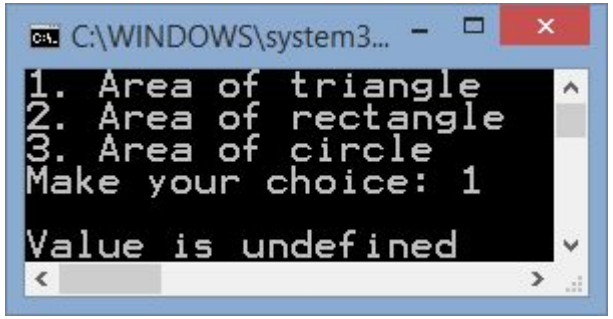
Через указатель на базовый класс можно работать с объектом производного класса, но только с той частью, которая была унаследована из базового

Это позволяет абстрагироваться от конкретного объекта, от конкретной реализации

Это дает возможность выбирать любую реализацию во время выполнения программы

Т.е. работать с любым объектом через указатель на базовый класс

```
figure *p = nullptr;
int choice;
cout << "1. Area of triangle\n2. Area of rectangle\n"
     << "3. Area of circle\n";
cout << "Make your choice: ";
cin >> choice;
switch (choice)
{
case 1:
    p = new triangle;
    p->setDimension(5.5, 10.0);
    break;
case 2:
    p = new rectangle;
    p->setDimension(5.5, 10.0);
    break;
case 3:
    p = new circle;
    p->setDimension(10.0);
    break;
}
Print(p);
```



При работе с объектом производного класса через указатель на базовый класс **связывание** вызова метода с самим кодом метода происходит **на этапе построения программы**

В результате вызывается метод класса, соответствующий типу указателя, а не типу объекта, который адресуется через данный указатель

Это, по-прежнему, **раннее связывание**

При работе с объектом производного класса через указатель на базовый класс желательно, чтобы связывание вызова метода с самим кодом метода происходило на этапе выполнения программы

Необходимо, чтобы вызывался метод в соответствии с типом объекта, а не типом указателя, который адресует данный объект

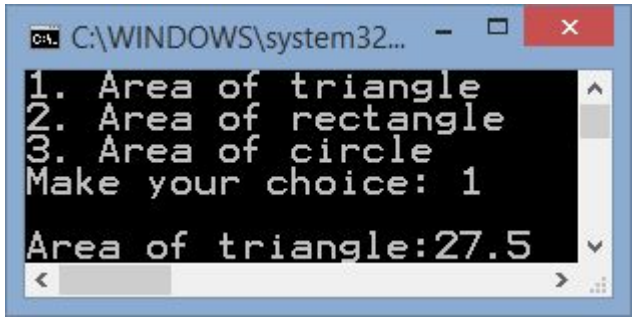
Для решения данной проблемы в базовом классе переопределяемый метод объявляется как **виртуальный**

В производных классах этот виртуальный метод переопределяется

Класс, который содержит хотя бы один виртуальный метод, называется **полиморфным**

```
class figure{
protected:
    double x, y;
public:
    void setDimension(double i, double j = 0) { ... }
    virtual void calculateArea() { ... }
};

class triangle : public figure{
public: virtual void calculateArea() { ... }
};

class rectangle : public figure{
public: virtual void calculateArea() { ... }
};

class circle : public figure{
public: virtual void calculateArea() { ... }
};
```

```
figure *p = nullptr;
int choice;
cout << "1. Area of triangle\n2. Area of rectangle\n"
     << "3. Area of circle\n";
cout << "Make your choice: ";
cin >> choice;
switch (choice)
{
case 1:
    p = new triangle;
    p->setDimension(5.5, 10.0);
    break;
case 2:
    p = new rectangle;
    p->setDimension(5.5, 10.0);
    break;
case 3:
    p = new circle;
    p->setDimension(10.0);
    break;
}
Print(p);
```



Для каждого класса иерархии, в котором определяется или переопределяется виртуальный метод, компилятор создаёт **таблицу виртуальных функций**

В этой таблице содержатся адреса всех виртуальных методов в порядке их описания в классе

Адрес любого виртуального метода имеет в таблице одно и то же смещение для каждого класса в пределах иерархии

Каждый объект полиморфного класса содержит дополнительное поле-указатель **vptr** на таблицу виртуальных функций

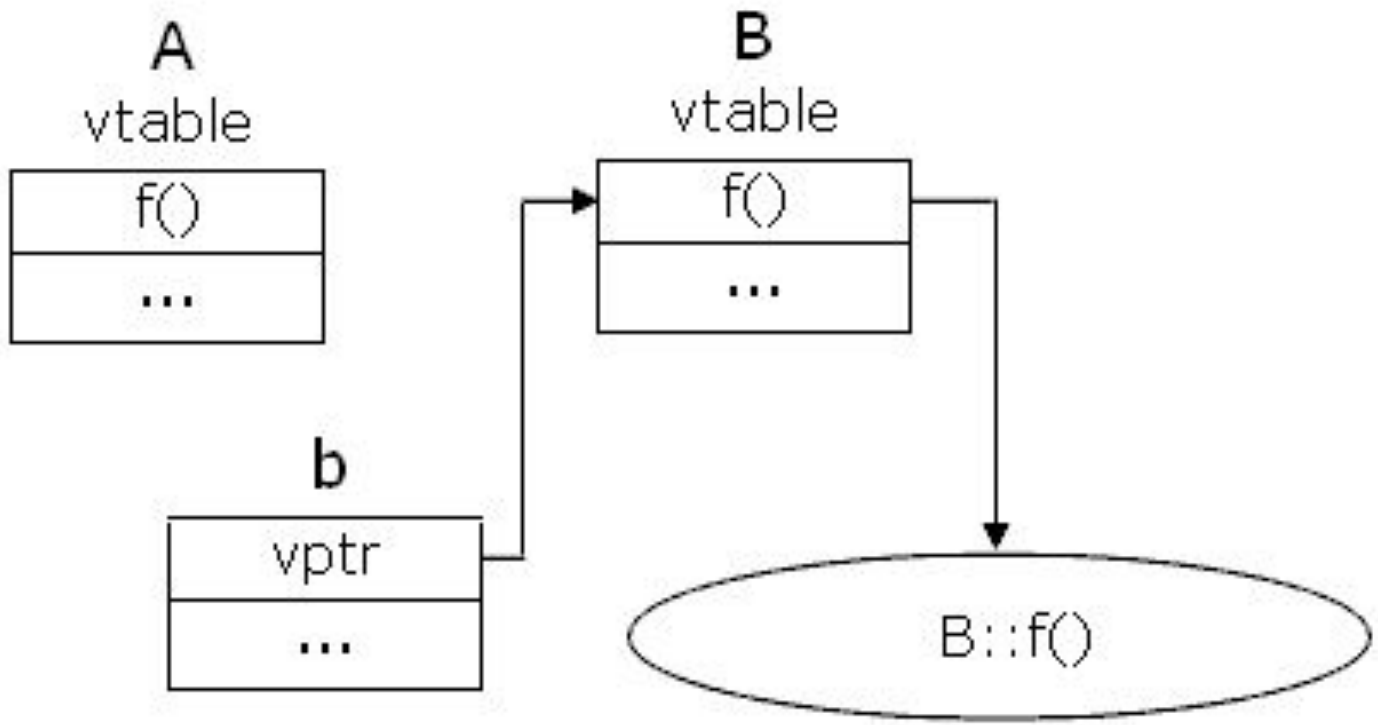
Этот указатель vptr заполняется конструктором при создании объекта

На этапе компиляции программы все обращения к виртуальным методам заменяются на обращения к таблице через vptr объекта

На этапе выполнения в момент обращения к виртуальному методу его адрес выбирается из таблицы и выполняется вызов

В этом и заключается **механизм позднего связывания**

Механизм позднего связывания



Таким образом, вызов виртуального метода, в отличие от обычных методов, выполняется через дополнительный этап получения адреса метода из таблицы виртуальных функций

В производных классах переопределённые виртуальные методы не обязательно объявлять с ключевым словом **virtual**

Отсюда вытекает **правило виртуальности**: метод, объявленный виртуальным в некотором классе, остается виртуальным во всех его потомках

Если в базовом классе имеется виртуальный метод, то в производном классе метод с таким же именем и сигнатурой автоматически становится виртуальным

Виртуальный метод не может быть статическим, однако может быть другом в некотором классе

Объект, который адресуется через указатель или ссылку и имеет хотя бы один виртуальный метод, называется **полиморфным**

Виртуальный метод, объявленный в базовом классе, наследуется производными классами

Поэтому переопределять его следует только в том случае, если необходимо задать отличающиеся действия

```
class base{
public: virtual void WhoIsThis() { ... }
};

class derived1 : public base{
public: void WhoIsThis() { ... }
};

class derived2 : public base{};

void main()
{
    base *ptr;
    base obj1;
    derived1 obj2;
    derived2 obj3;
    ptr = &obj1;
    ptr->WhoIsThis();
    ptr = &obj2;
    ptr->WhoIsThis();
    ptr = &obj3;
    ptr->WhoIsThis();
}
```

```
class base{
public: virtual void WhoIsThis() { ... }
};

class derived1 : public base{
public: void WhoIsThis() { ... }
};

class derived2 : public derived1{ };

void main()
{
    base *ptr;
    base obj1;
    derived1 obj2;
    derived2 obj3;
    ptr = &obj1;
    ptr->WhoIsThis();
    ptr = &obj2;
    ptr->WhoIsThis();
    ptr = &obj3;
    ptr->WhoIsThis();
}
```

Абстрактный базовый класс

Чаще всего базовые классы предназначены для представления общих понятий, которые предполагается конкретизировать в производных классах

Очевидно, что объекты таких классов создавать бессмысленно (например, класс figure)

Базовый класс важен исключительно как **абстрактный класс**, определяющий общие принципы работы его потомков

В языке C++ существует возможность зафиксировать эту абстрактность на уровне самого языка

Для этого используются **чисто виртуальные функции**

Чисто виртуальный метод – это метод, который в базовом классе только объявляется, но не определяется

```
virtual void calculateArea ()=0;
```

```
class figure{
protected:
    double x, y;
public:
    void setDimension(double i, double j = 0) { ... }
    virtual void calculateArea() = 0;
};

class triangle : public figure{
public: void calculateArea() { ... }
};

class rectangle : public figure{
public: void calculateArea() { ... }
};

class circle : public figure{
public: void calculateArea() { ... }
};
```

Базовый класс, который содержит хотя бы один чисто виртуальный метод называется **абстрактным базовым классом**

Абстрактный класс может использоваться только в качестве базового для других классов — объекты абстрактного класса создавать нельзя

Однако можно создавать указатель или ссылку на абстрактный базовый класс

Абстрактный класс можно рассматривать как заготовку, в которой часть функциональности реализована, а оставшаяся часть делегирована потомкам

Важно понимать, что производный класс, который наследуется от абстрактного базового класса, должен переопределить все чисто виртуальные методы

В противном случае производный класс также будет считаться абстрактным

Таким образом, можно создать функцию, параметром которой является указатель на абстрактный класс

На место этого параметра при выполнении программы может передаваться указатель на объект любого производного класса

Это позволяет создавать **полиморфные функции**, работающие с объектом любого типа в пределах одной иерархии

Полиморфная функция

```
void Print(figure *p)
{
    p->calculateArea();
}
```

Таким образом, полиморфизм предполагает один интерфейс и множество реализаций

Динамическая идентификация типов (**Run-Time Type Identification, RTTI**)

Механизм идентификации типа во время выполнения программы позволяет определять, на какой тип в текущий момент времени ссылается указатель

Для доступа к **RTTI** в стандарт языка введен оператор **typeid** и класс **type_info**

Формат оператора typeid:
typeid (тип)
typeid (выражение)

Оператор принимает в качестве параметра имя типа или выражение и возвращает ссылку на объект класса **type_info**, содержащий информацию о типе

```
figure* GeneratorOfFigures()
{
    int n = rand() % 3;
    switch (n)
    {
    case 0: return new triangle(10.5, 6.5);
    case 1: return new rectangle(10.5, 6.5);
    case 2: return new circle(10);
    }
}
```

```
figure *p[20];
int count_triangle = 0, count_rectangle = 0,
    count_circle = 0;
for (int i = 0; i < 20; i++)
{
    p[i] = GeneratorOfFigures();
    if (typeid(*p[i]) == typeid(rectangle))
        count_rectangle++;
    if (typeid(*p[i]) == typeid(triangle))
        count_triangle++;
    if (typeid(*p[i]) == typeid(circle))
        count_circle++;
    cout << typeid(*p[i]).name() << endl;
}
```

Операторы == и != позволяют сравнивать два объекта на равенство и неравенство

Метод **name()** возвращает указатель на строку, представляющую имя типа, описываемого объектом класса **type_info**

Полиморфное приведение типов с использованием оператора **dynamic_cast**

Оператор **dynamic_cast** применяется для преобразования указателей родственных классов иерархии

Чаще всего **dynamic_cast** применяется для преобразования указателя базового класса в указатель на производный

Формат оператора:

dynamic_cast <тип *> (выражение)

При этом во время выполнения программы производится проверка допустимости преобразования

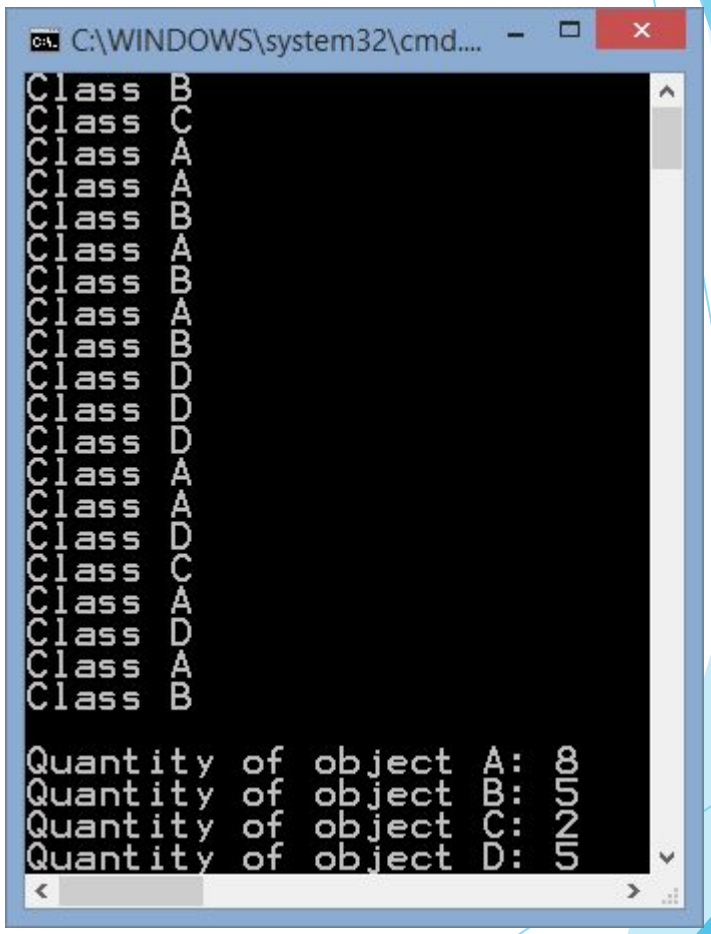
Для того чтобы проверка допустимости могла быть выполнена, аргумент оператора **dynamic_cast** должен быть полиморфного типа, т.е. иметь хотя бы один виртуальный метод

Если преобразование оказалось некорректным, то оно не выполняется, и оператор вернет нулевой указатель

```
class A{
public: virtual void Print() { ... }
};

class B : public A{
public: void Print() { ... }
};

class C : public B{
public: void Print() { ... }
};

class D : public C{
public: void Print() { ... }
};
```

```
A* Generator()
{
    int n = rand() % 4;
    switch (n)
    {
    case 0: return new A;
    case 1: return new B;
    case 2: return new C;
    case 3: return new D;
    }
}
```

```
A *p[20];
int count_A = 0, count_B = 0, count_C = 0, count_D = 0;
for (int i = 0; i < 20; i++)
{
    p[i] = Generator();
    if (dynamic_cast<D*>(p[i]))
        count_D++;
    else
    {
        if (dynamic_cast<C*>(p[i]))
            count_C++;
        else
        {
            if (dynamic_cast<B*>(p[i]))
                count_B++;
            else
                count_A++;
        }
    }
    p[i]->Print();
}
```



Обработка исключительных ситуаций

Исключение – это временный объект, создаваемый программой в случае возникновения ошибки

Исключение будет существовать до тех пор, пока его не обработают

Синтаксис создания (вбрасывания) исключения:

throw значение;

Исключение может быть объектом любого типа - как стандартного, так и пользовательского

Механизм обработки исключений - это способ обработки ошибок средствами языка C++

Синтаксис обработки исключений

```
try
{
 // код, подлежащий проверке на наличие ошибок
    throw <выражение>
//генерация исключения указанного типа
}
catch(<тип_исключения>)
{
    //обработка исключения
}
```

Файловый ввод/вывод средствами языка C++

Поток — это абстрактное понятие, относящееся к любому переносу данных от источника к приемнику

Поток определяется как последовательность байтов и не зависит от конкретного устройства, с которым производится обмен

Физическим устройством может быть клавиатура, монитор, файл, оперативная память, принтер и т.д.

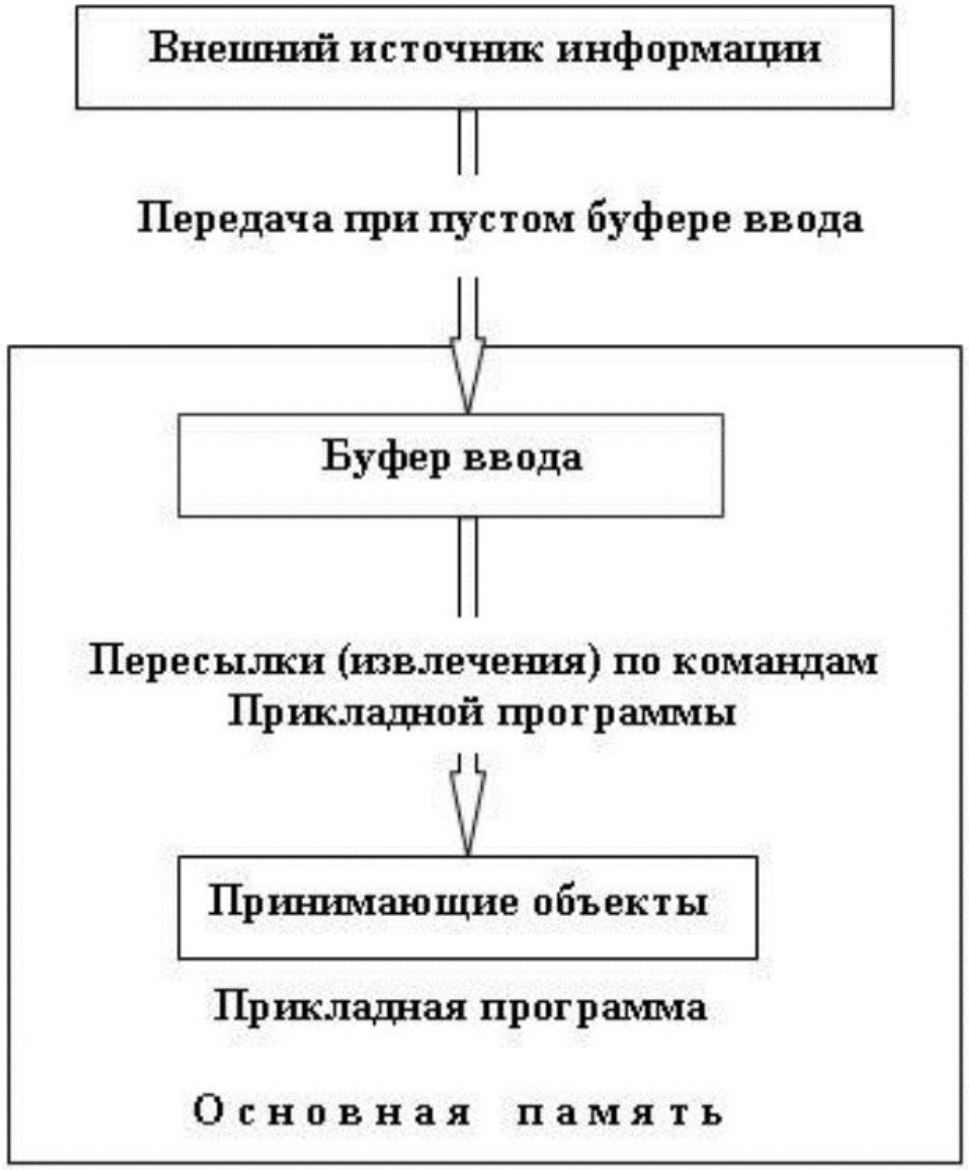
Обмен с потоком для увеличения скорости передачи данных производится, как правило, через специальную область оперативной памяти — **буфер потока**

Фактическая передача данных выполняется при выводе после заполнения буфера, а при вводе — если буфер исчерпан

При этом чтение данных из потока называется **извлечением**, а вывод данных в поток — **включением**

Вывод данных в поток

Чтение данных из потока



По направлению обмена потоки можно разделить на входные, выходные и двунаправленные

Входные потоки - потоки, из которых читаются данные

Выходные потоки - потоки, в которые записываются данные

Двунаправленные потоки - потоки, допускающие как чтение, так и запись

Стандартная библиотека содержит три класса для работы с файлами:

- **ifstream** — класс входных файловых потоков
- **ofstream**— класс выходных файловых потоков
- **fstream** — класс двунаправленных файловых потоков

Эти классы являются производными от классов **istream**, **ostream** и **iostream** соответственно

Объект класса **ifstream** представляет собой входной файловый поток для чтения информации из файла

Объект класса **ofstream** представляет собой выходной файловый поток для записи информации в файл

Объект класса **fstream** представляет собой двунаправленный файловый поток для чтения и записи

Режимы открытия файла

Комбинация флагов ios					Эквивалент
binary	in	out	trunc	app	stdio
		+			"w"
		+		+	"a"
		+	+		"w"
	+				"r"
	+	+			"r+"
	+	+	+		"w+"
+		+			"wb"
+		+		+	"ab"
+		+	+		"wb"
+	+				"rb"
+	+	+			"r+b"
+	+	+	+		"w+b"

Стандартная библиотека C++

Стандартная библиотека шаблонов (STL) (Standard Template Library) - это составная часть стандартной библиотеки языка C++

STL - это набор шаблонных классов и функций общего назначения, реализующих наиболее популярные структуры данных и часто употребительные алгоритмы работы с ними

Архитектура STL была разработана Александром Степановым и Менг Ли

STL состоит из следующих компонентов:

- контейнеры (containers)
- адаптеры (adaptors)
- итераторы (iterators)
- алгоритмы (algorithms)
- функторы (functors)
- предикаты (predicates)

Контейнер предназначен для хранения набора объектов в памяти

Два основных типа контейнеров: **последовательные** и **ассоциативные**

Последовательный контейнер – упорядоченная коллекция, в которой каждый элемент имеет определенную позицию (vector, list)

Позиция зависит от места вставки, но не зависит от значения элемента

Ассоциативный контейнер – коллекция элементов, в которой позиция элемента зависит от его значения и выбранного критерия сортировки (map, set)

Такой контейнер всегда находится в отсортированном состоянии, что ускоряет выполнение поиска

Адаптер – специализированный контейнер, который предоставляет определенный интерфейс для доступа к данным (stack, queue)

Итератор – это специальный указатель, который используется для перемещения по контейнеру и для манипулирования объектами, находящимися в контейнере

Итераторы делятся на 5 категорий:

- итератор ввода
- итератор вывода
- прямой итератор
- двусторонний итератор
- итератор произвольного доступа

Итератор ввода (InputIterator) перемещается только вперед и поддерживает только чтение

Итератор вывода (OutputIterator) перемещается только вперед и поддерживает только запись

Прямой итератор (ForwardIterator) перемещается только вперед, позволяет выполнять чтение и запись

Двусторонний итератор (BidirectionalIterator) - прямой итератор, который поддерживает перебор элементов в обратном порядке

Итератор произвольного доступа (RandomAccessIterator) имеет все свойства двустороннего итератора, а также поддерживает произвольный доступ к элементам

Алгоритмы – функции, которые используются для обработки элементов в контейнере (например, сортировка, поиск)

Алгоритмы используют итераторы

Так как каждый класс контейнера имеет итератор, один и тот же алгоритм может работать с различными контейнерами

Функторы - это объекты, действующие как функции, они могут быть объектами класса или указателями на функцию

Функторы позволяют выполнять конфигурирование алгоритмов для специального использования

Предикаты - функции, которые проверяют, удовлетворяет ли объект заданным критериям, и возвращают значение типа bool

Предикаты используются, например, в поисковых алгоритмах для задания более сложного критерия поиска