

# Программирование и алгоритмизация

Санкт-Петербургский государственный политехнический университет

17 февраля 2015

На Си Вы могли просто делать ошибки, на C++ Вы сможете их также наследовать

Включает в себя:

- Процедурные возможности — акцент делается в основном на реализацию программы посредством эффективных функций. Применяется для небольших программ, где требуется оптимизировать время и объем занимаемой памяти.
- Объектно-ориентированные средства (классы) — время выполнения и объем программы увеличиваются, но программист получает выигрыш при создании больших программных продуктов, который выражается в уменьшении трудоемкости.

Объектно ориентированные средства C++ позволяют разработчику ПО: Создавать собственные пользовательские типы данных (классы) для описания объектов реального мира и использовать их также как и объекты базового типа.

- Инкапсуляция — каждый класс представляет уникальный набор данных и операций (методов) над этими данными. Объект обычно представляется как «Черный ящик» в котором скрыты детали реализации.
- Наследование — возможность добавления в класс новых свойств на основе базовых классов.
- Полиморфизм — получение разного поведения объектов во время выполнения посредством одного и того же кода.

```
struct Date {  
    int year;  
    int month;  
    int day;  
};  
  
int DayOfYear(const Date *date);  
  
//main.cpp  
int main()  
{  
    Date d = {10, 10, 2015};  
    std::cout << DayOfYear(&d);  
}
```

---

# Объектно-ориентированное решение

---

```
class Date {  
int year;  
int month;  
int day;  
public:  
int DayOfYear(const Date *date);  
};
```

```
//main.cpp  
int main()  
{  
Date d; //создание объекта  
std::cout << d.DayOfYear(); //вызов метода класса  
}
```

---

Данные содержатся в самом объекте, как в капсуле. Главный принцип ООП — не получайте посредством объекта данные, необходимые для совершения операции. Вместо этого «попросите» объект содержащий данные сделать эту операцию для Вас. Этот принцип называется делегированием.

# Объявление класса

---

```
class имя_класса {  
    список_членов_класса: переменные (member variables)  
        и методы (member functions)  
};
```

---

Компилятор извлекает из класса следующую информацию:

- новый пользовательский тип данных;
- имена и типы членов класса, включая типы переменных и прототипы функций;
- спецификаторы доступа (ограничения по использованию функций и переменных класса).

# Абстракция

Основное назначение классов — описывать объекты реального мира, а следовательно проектирование класса == моделированию.

# Абстрагируем животное

---

```
enum SEX {MALE, FEMALE};
```

```
class Animal {
    unsigned int m_age;
    bool m_isPet;
    SEX m_sex;
};
```

---

# Нарушение инкапсуляции

---

```
enum SEX {MALE, FEMALE};

struct Animal {
    unsigned int m_age;
    bool m_isPet;
    SEX m_sex;
};

int main()
{
    Animal an;
    an.m_age = 100;
    an.m_isPet = true;
}
```

---

---

```
enum SEX {MALE, FEMALE};

class Animal {
private: //защита данных от изменения
unsigned int m_age;
bool m_isPet;
SEX m_sex;
};

int main()
{
Animal an;
an.m_age = 100; //ошибка компилятора
an.m_isPet = true; //ошибка компилятора
}
```

---

# Изменение защищенных данных

```
enum SEX {MALE, FEMALE};  
  
class Animal {  
private: //защита данных от изменения  
    unsigned int m_age;  
    bool m_isPet;  
    SEX m_sex;  
public:  
    void setAge(int age) { m_age = age; }  
};  
  
int main()  
{  
    Animal an;  
    an.setAge(100); //???  
    an.m_age = 50; //???  
}
```

- Каждый уровень доступа к данным и функциям определяется ключевыми словами `public`, `private`, `protected`.
- Не обязательно для каждого члена класса указывать спецификатор доступа.
- По умолчанию объявлены как `private`.
- Секции с одним и тем же ключевым словом может быть сколько угодно и идти они могут в любом порядке.
- `private` и `protected` члены другого объекта такого же класса будут внутри всех методов класса.

# Создание объекта

Создание экземпляра класса осуществляется также как создание переменной.

---

```
#include "animal.h"
```

```
int main()
{
    Animal a;
}
```

---

# Объявление и определение inline методов

Создание экземпляра класса осуществляется также как создание переменной.

---

```
class Animal {  
public:  
void setPet() { isPet = true; } //неявно  
};  
  
class Animal {  
public:  
inline void setPet(); //явно  
};  
  
inline void Animal::setPet() { isPet = true; }
```

---

# Объявление и определение не-*inline* методов

Создание экземпляра класса осуществляется также как создание переменной.

---

```
//animal.h
```

```
class Animal {  
public:  
void setPet();  
};
```

```
//animal.cpp
```

```
void Animal::setPet() { isPet = true; }
```

---

- указатель `this` формируется компилятором внутри нестатических методов, следовательно использовать его можно только внутри класса.
- для класса А тип указателя `this` — `A* const`.

# Программирование и алгоритмизация

Санкт-Петербургский государственный политехнический университет

24 февраля 2015

При создании объекта требуется:

- выделить под него память;
- проинициализировать его переменные различными значениями.

Для автоматического осуществления перечисленных действий в классах используются конструкторы.

# Конструктор

Метод, который вызывается компилятором автоматически при создании объекта.

- нельзя вызвать явно, как обычный метод;
- имя конструктора всегда совпадает с именем класса;
- у конструктора отсутствует тип возвращаемого значения, даже `void`;
- конструктор не может быть константным, виртуальным или статическим методом.

# Конструктор по умолчанию

Конструктором по умолчанию называется конструктор у которого нет параметров или все параметры имеют значения по умолчанию.

Генерируется конпилятором автоматически, если программист явно не определил свой конструктор.

- вызывает конструкторы для всех встроенных объектов класса;
- вызывает конструктор базового класса (при наследовании).

Компилятор не генерирует default конструктор, если:

- в классе объявлен любой другой конструктор;
- в классе объявлены константные члены данных;
- данный класс является производным, а в базовом классе конструктор объявлен как private.

# Автоматический конструктор по умолчанию(default)

---

```
//animal.h
class Animal
{
    unsigned int age;
};

//main.cpp
#include "animal.h"
Animal aGlobal;

int main()
{
    Animal aLocal;
}
```

---

# Пользовательский конструктор по умолчанию(default)

---

```
//animal.h
class Animal
{
    enum AnimalType { Predator , Herbivous , Unknown };
    unsigned int age;
    AnimalType type;

public:
    Animal()
    { age = 0; type = Unknown; }
};

//main.cpp
#include "animal.h"
int main() {
    Animal aLocal; //Выделена память и вызван
                    //конструктор по умолчанию.
}
```

# Конструктор с параметрами

```
//animal.h
enum AnimalType { Predator , Herbivous , Unknown };
class Animal
{
    unsigned int age;
    AnimalType type;

public:
    Animal(int age , AnimalType t)
    { this ->age = age; type = t; }
};

//main.cpp
#include "animal.h"
int main()
{
    Animal aLocal(10, Predator); //Выделена память и
        вызван конструктор по умолчанию.
    Animal aLocal2; //???
}
```

# Конструктор с одним параметром

---

```
class A
{
int m_a;
public:
A(int a) { m_a = a; }
};

int main()
{
A a = 1;
A a(1);
}
```

---

Если добавить к конструктору ключевое слово `explicit`, то это запретит приводить тип компилятору неявно.

# Конструкторы базовых типов

---

```
int i = 0;
int i1 (0);

class A
{
    int i;
    const int x;

public:
    A() : i(0), x(0) {}
};
```

---

# Перегрузка конструкторов

---

```
class Animal
{
    ...
public:
    Animal() { /*...*/ }
    Animal(int age, AnimalType) { /* ... */ }
};

int main()
{
    Animal an1;
    Animal an2(10, Predator);
}
```

---

# Конструктор с параметрами по умолчанию

---

```
class Animal
{
...
public:
    Animal(int age = 0, AnimalType = Unknown) { /* ...
        ... */ }
    // Animal() {} //FIXME?
};

int main()
{
    Animal an1;
    Animal an2(10);
    Animal an3(10, Predator);
}
```

---

---

```
Animal *p1 = new Animal;  
Animal *p2 = new Animal();  
Animal *p3 = new Animal(10);  
  
delete p1;  
delete p2;  
delete p3;
```

---

- в большинстве случаев вызывается компилятором неявно;
- можно вызвать явно, как метод;
- имя деструктора совпадает с именем класса, но с символом ~;
- не принимает параметров и ничего не возвращает;
- не может быть константным или статическим;
- может быть виртуальным.

# Автоматический деструктор

- вызывает деструкторы встроенных объектов данного класса;
- вызывает деструкторы базовых классов.

# Классы с динамической памятью

---

```
class Animal
{
    ...
    char *name;
public:
    Animal(unsigned int age, AnimalType, const char *
        pName);
```

---

# Плохой способ реализации

```
Animal::Animal(unsigned int age, AnimalType t, char
               *pName)
{
    this->age = age;
    type = t;
    name = pName; //<— Потенциальная ошибка в
                   программе
}
int main() {
    char ar[] = "Sharik";
    Animal a1(1, Unknown, ar);
    Animal a2(1, Unknown, ar); //если мы переименуем
                               животное a2, то a1, тоже изменит имя!
    char *name = new char[64];
    cin >> name;
    Animal a3(1, Unknown, name);
    delete [] name;
    //продолжаем работать с a3 => это приведет к
      краху программы }
```

# Правильный вариант реализации

---

```
Animal::Animal(unsigned int age, AnimalType t, char
               *pName)
{
    this->age = age;
    type = t;
    name = new char[ strlen(pName) + 1 ];
    strcpy(name, pName); //НО, кто очистит
                         динамическую память от поля name ?
}
```

---

---

```
Animal::~Animal()
{
    delete [] name;
}
```

---

Явную реализацию деструктора, необходимо писать каждый раз, когда идет выделение динамической памяти в классе.

# Конструктор копирования

Если нет явной реализации, генерируется компилятором автоматически. Используется для:

---

- \item поэлементного копирования всех полей в классе ;
  - \item вызывает конструктор копирования базового класса ;
  - \item вызывает конструкторы копирования для встроенных объектов .
- 

Компилятор не создает автоматический конструктор копирования, если:

---

- \item в классе объявлены константные члены данных ;
  - \item в классе объявлены ссылки ;
  - \item в базовом классе конструктор копирования находится в секции **private** .
-

# Конструктор копирования

---

```
{  
    Animal a1(10, Predator, "Bobik");  
    Animal a2 = a1; //Вызов конструктора копирования  
    !!!  
    Animal a3(a2);  
} //вызов деструкторов
```

---

ВОПРОС: Какая потенциальная проблема возникнет между объектами a1, a2 и a3, даже если у них корректно реализован конструктор с параметрами и деструктор?

# Реализация правильного конструктора копирования

---

```
Animal::Animal(const Animal &other)
{
    age = other.age;
    type = other.type;
    name = new char[ strlen(other.name) + 1 ];
    strcpy(name, other.name);
}
```

---

Конструктор копирования обычно в качестве параметра принимает константную ссылку, на объект такого же типа как и сам класс.

# Неявный вызов конструктора копирования

---

```
void F(A a)
{
    //вызов конструктора копирования
    //в функции происходит работа с копией
} //вызывается деструктор временного объекта a

A copy()
{
    A temp;
    return temp; //происходит вызов конструктора
                  //копирования
                  //в результате взывающую функцию
                  //передается
                  //копия локального объекта
} //деструктор для объекта temp
```

---

## move конструктор копирования

---

```
class Animal
{
...
public:
    Animal(const Animal&); //обычный конструктор
    //копирования
    Animal(Animal &&); //move конструктор копирования
};

Animal f()
{
    Animal tmp;
    ...

    return tmp; //вызов move конструктора копирования
}
```

---

# Реализация move конструктора копирования

---

```
Animal::Animal(Animal &&other)
{
    age = other.age;
    type = other.type;
    name = other.name; //отбираем указатель на
                        //временный объект
    other.name = nullptr; //присваиваем строке 0, т.к
                          .деструктор все равно удалит эту строку
}
```

---

# Ключевое слово const и классы

const применимо к объектам и методам класса, в качестве принимаемого или возвращаемого значения.

# Константные методы класса

---

```
class A
{
    int m_a;
public:
    int getA() const { //const означение, что все
        поля read-only
        //m_a++
        return m_a;
    }

    void setA(int a) {
        m_a = a;
    }
};
```

---

# Ключевое слово mutable

mutable — означает, что поле может быть изменено из const-метода

---

```
class A
{
    mutable int counter;
    int m_a;
public:
    int getA() const { //const означение, что все
        поля read-only
        //m_a++;
        counter++; //OK
        return m_a;
    }

    void setA(int a) {
        m_a = a;
    }
};
```

# Константные объекты

Для константных объектов можно вызывать, только  
константные методы

---

```
class A
{
public:
    void f1();
    void f2() const;
};

int main()
{
    A a;
    a.f1(); //OK
    a.f2(); //OK

    const A a;
    a.f1(); //???
    a.f2(); //???
}
```

---

```
class A
{
public:
    A(int) {}
};

int main()
{
    A arr[5]; //???
    A arr2[2] = { A(1), A(2) }; //???
    A arr3[5] = { A(1), A(2) }; //???
    A arr4[] = { 1, 2, 3 }; //???

    A *p = new A[10]; //???

    delete [] p; //???
}
```

---

# Массив указателей на объекты

---

```
int main()
{
    A *arr [5]; //???
    A *arr2 [2] = { new A(5), new A(4) };

    delete //???
}
```

---

# Программирование и алгоритмизация

Санкт-Петербургский государственный политехнический университет

03 марта 2015

# Отношения между классами

Бывают следующих типов:

- является (is a). Открытое public-наследование;
- содержит (has a). Нет наследования, объект содержит объект другого класса;
- подобен (as a). Закрытое (private или protected) наследование.

Для корректного построения иерархии, программист должен представлять различия между типами взаимоотношений и применять их исключительно по назначению.

# Виды наследования

- одиночное;
- множественное

# Одиночное наследование

---

```
class A //базовый класс
{ //список членов класса A
};
```

```
class B : public A
{ //производный класс
};
```

---

Такое объявление говорит компилятору, что класс B включает в себя весь класс A, как составную часть. В зависимости от спецификаторов доступа методы класса B имеют право обращаться к членам класса A или нет.

# Спецификаторы при наследовании

Спецификатор при наследовании, позволяет задать для базового класса более жесткие ограничения на свойства видимости переменных.

# Пример наследования спецификаторов

---

```
class A
{ private: int a;
protected: int b;
public: int c;
};
class B : public A //если private , protected ?
{
public: B() {
a = 0;
b = 0;
c = 0; }//???
};
int main()
{ B b;
b.a = 1; //???
b.b = 2; //???
b.c = 3; //??? }
```

# Порядок вызова конструкторов

---

```
class A{};
class B : public A {};
class C : public B {};
C::C()
{
    //вызов конструктора класса B
    //тело конструктора C
}
B::B()
{
    //вызов конструктора класса A
    //тело конструктора B
}
A::A()
{
    //тело конструктора A
}
```

---

# Порядок вызова деструкторов

---

```
class A{};
class B : public A {};
class C : public B {};
C::~C()
{
    //тело деструктора C
} //вызов деструктора класса B
B::B()
{
    //тело деструктора B
} //вызов деструктора класса A
A::A()
{ //тело деструктора A }
```

---

При вызове оператора `delete`, вначале вызываются деструкторы, а потом освобождается динамическая память.

# Иерархия животных

---

```
//animal.h
class Animal
{
protected:
int m_age;
char *m_name;
public:
Animal(int age, const char *name);
};

//dog.h
class Dog : public Animal
{
bool m_hasMaster;
char *m_masterName;
public:
Dog(int age, bool master, const char *name, const
char *mName); }
```

# Передача параметров конструктору базового класса

---

```
//animal.cpp
Animal::Animal(int age, char *name)
{
}

//dog.cpp
Dog::Dog(int age, bool master, const char *name,
          const char *masterName) : Animal(age, name)
{
    m_hasMaster = master;
    m_masterName = new char [strlen(masterName) + 1];
    strcpy(m_masterName, masterName);
}
```

---

# Передача параметров конструктору копирования

---

```
//animal.cpp
Animal::Animal(const Animal &a)
{
    m_age = a.m_age;
    m_name = new char [strlen(a.m_name) + 1];
    strcpy(m_name, a.m_name);
}

//dog.cpp
Dog::Dog(const Dog &d) : Animal(d)
{
    m_hasMaster = d.m_hasMaster;
    m_masterName = new char [strlen(d.masterName) + 1];
    strcpy(m_masterName, d.m_masterName);
}
```

---

Открытое наследование в C++ моделирует следующее утверждение:

- производный класс — разновидность базового класса;
- все что применимо к базовому классу, должно быть применимо к производному классу;
- везде, где может быть использован объект A, может быть использован объект B, поскольку объект B содержит базовую часть A.

# Правила открытого наследования

---

```
class A{};
class B: public A {};

void f1(A, A); //A &, A*
void f2(B, B); //B &, B*
int main()
{
    A a; B b;
    f1( a, b ); //???
    f2( a, b ); //???
}
```

---

---

```
class Animal
{
public:
void Voice() const { cout << "???" ; }
};

class Dog : public Animal
{
void Voice() const { cout << "Gav" ; }
};

class Cat : public Animal
{
void Voice() const { cout << "Miau" ; }
};
```

---

---

```
void F(const Animal *p)
{ p->Voice(); }

int main()
{
Animal a;
Dog b;
Cat c;

F(&a); //???
F(&b); //???
F(&c); //???

}
```

---

# Раннее связывание. Исправление

---

```
class Animal
{
AnimalType type;
};

void F(const Animal *p)
{
if (type == CAT)
static_cast<Cat*>(p)->Voice();
else if (type == DOG)
static_cast<Dog*>(p)->Voice();
else p->Voice();
}
```

---

# Механизм вызова виртуальных функций

- Если в объявлении класса появляется виртуальная функция, то компилятор формирует таблицу виртуальных функций.
- Каждый объект в дополнении к стандартным полям хранения данных содержит поля для хранения указателя на таблицу виртуальных функций — vfptr;

# Механизм позднего связывания

---

```
class Animal
{
public:
virtual void Voice() const { cout << "???" ; }

class Dog : public Animal
{
virtual void Voice() const { cout << "Gav" ; }

class Cat : public Animal
{
virtual void Voice() const { cout << "Miau" ; }
```

---

---

```
void F(const Animal *p)
{ p->Voice(); }

int main()
{
Animal a;
Dog b;
Cat c;

F(&a); //???
F(&b); //???
F(&c); //???

}
```

---

# Виртуальные деструкторы

---

```
class Animal
{ public:
/* virtual */ ~Animal();
};

class Dog: public Animal
{ public:
~Dog();
};

int main()
{
Animal *zoo[] = { new Animal(), new Dog() };
for(int i = 0; i < sizeof(zoo)/sizeof(Animal*); i
++)
{
zoo[i]->Voice();
delete zoo[i];
}
}
```

---

# Спецификатор разрешения области видимости

---

```
class A
{
public:
void F(int);
};

class B : public A
{
public:
void F(int, int);
void F(int);
};

int main()
{
B b;
b.F(1, 2); //???
b.F(3); //???
b.A::F(3); //???
}
```

# Спецификатор разрешения области видимости

Спецификатор разрешения области видимости отменяет полиморфизм

---

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

class B : public A
{ public:
virtual void VF();
};

int main()
{ A *p = new B;
p->VF(); //???
p->A::VF(); //???
}
```

---

```
class A
{ public: void General() { VF(); }
private: virtual void VF() { /* ... */ }
};
class B: public A
{ virtual void VF() { /* ... */ } };
int main()
{
A* pA = new A;
A* pB = new B;
pA->General();
pB->General();
pA->A:: General();
pB->A:: General();
pA->A:: VF();
pB->B:: VF();
}
```

---

```
class A
{
public:
virtual void VF() = 0;
};
```

---

Делает функцию чисто виртуальной. Чисто виртуальный метод может, но не обязан иметь тело.

# Абстрактный класс

- класс содержащий хотя бы одну чисто виртуальную (pure virtual) функцию, называется абстрактным;
- компилятор НЕ ПОЗВОЛЯЕТ создавать объекты такого класса;
- абстрактный класс используется только в качестве базового класса при наследовании;
- компилятор следит, чтобы в производных классах такой метод был реализован;
- можно использовать указатель на абстрактный класс.

# Абстрактный класс

---

```
class Animal
{ public:
void Voice() const = 0;
};

class Dog : public Animal
{ void Voice() const { cout << "Gav"; } };
class Cat : public Animal
{ void Voice() const { cout << "Miau"; } };

int main()
{
Animal animal; //???
Animal *zoo[] = { new Dog, new Cat, new Animal };
//???

Animal *zoo2[] = { new Dog, new Cat };
//FIXME
}
```

---

# Программирование и алгоритмизация

Санкт-Петербургский государственный политехнический университет

17 марта 2015

Ключевое слово предоставляющее возможность обращения к защищенным понятиям класса извне. Может быть полезно при перезагрузке операторов и реализации вспомогательных классов.

## Без friend

---

```
class Rect
{
    int m_left, m_top, m_right, m_bottom;
};

Rect BoundingRect(const Rect &r1, const Rect &r2)
{
    int l = (r1.m_left < r2.m_left) ? r1.m_left : r2.
        m_left; //???
    ...
}
```

---

Секция расположения ключевого слова `friend` не имеет значения, хотя часто для ясности располагают прототип функции в секции `public`.

---

```
class Rect
{
    int m_left, m_top, m_right, m_bottom;
    friend Rect BoundingRect(const Rect &r1, const
        Rect &r2);
};
```

```
Rect BoundingRect(const Rect &r1, const Rect &r2);
{
    int l = (r1.m_left < r2.m_left) ? r1.m_left : r2.
        m_left; //???
    ...
}
```

---

Для всех остальных функций правила доступа к полям остаются прежними.

Дружба не передается по наследству.

---

```
int main()
{
    Rect r;
    r.m_left = 1; //???
}
```

---

# Дружба при наследовании

Друг производного класса имеет доступ к **protected** членам базового класса.

---

```
class A { int m_a1;
protected: int m_a2;
};
class B { int m_b;
    friend void F(B &b);
};
void F(B& b)
{
    std::cout << b.m_a1; //??
    std::cout << b.m_a2; //??
    std::cout << b.m_b; //??
}
```

---

# Дружба между классами

---

```
class Circle { int r;
public:
    Circle(const Rect&r) {
        int w = r.m_right - r.m_left; //???
    }
};
```

---

## friend class

---

```
class Rect {  
    friend class Circle;  
    ...  
};  
Circle::Circle(const Rect &r)  
{  
    int w = r.m_right - r.m_left;  
    int h = r.m_bottom - r.m_top;  
}
```

---

Дружба классов не наследуется.

Дружба классов не является транзитивной.

---

```
class X { friend class Y; };  
class Y { friend class Z; };
```

---

Методы класса Z не имеют права обращаться к защищенным членам класса X.

## friend метод другого класса

---

```
class Rect {
    friend Circle::Circle(const Rect&);
    ...
};

Circle::Circle(const Rect &r)
{
    int w = r.m_right - r.m_left; //???
    int h = r.m_bottom - r.m_top; //???
}
```

---

# Перегрузка операторов

Предоставление возможности обращаться с объектами пользовательского типа также как с переменными базового типа.

---

```
int x, y;  
x + y;  
double x1, y1;  
x1 + y1;
```

---

//выполняются разные низкоуровневые операторы для разных типов: add, fadd.

Для класса A программист должен реализовать функцию с именем operator+.

---

```
A a1(1), a2(2), a3;  
a3 = a1 + a2; //???
```

---

# Специфика перегружаемых операторов

- Нельзя создавать собственные операторы, а можно перегружать только существующие;
- перегруженный оператор действует только по отношению к объектам того класса, для которого он переопределен;
- нельзя менять число operandов оператора;
- перегруженные операторы наследуют приоритеты и ассоциативность от встроенных операторов;
- оператор перегружается только относительно пользовательского типа данных;
- нельзя перегружать операторы: . :: . \* ? :
- как любая другая функция оператор может быть перегружен несколько раз, быть виртуальным или чисто виртуальным;
- не существует ограничений на тип возвращаемого значения.

# Способы перегрузки операторов

Унарные операторы:

- метод класса без параметров;
- глобальная функция с одним параметром.

Бинарные операторы:

- метод класса с одним параметром;
- глобальная функция с двумя параметрами.

## Пример вызова operator+

---

```
A x, y, z;  
z = x + y; //нормальная форма вызова  
z = operator+(x, y); //функциональная форма вызова  
    //глобальной функции  
  
z = x.operator(y); //функциональная форма вызова  
    //метода класса
```

---

# Рекомендации при перегрузке

Всегда стоит предпочтовать перегрузку методом класса, за исключением:

- первый операнд относится к базовому типу, например,  $z = 1 + z$ ;
- тип первого операнда библиотечный;
- операторы  $=$ ,  $()$ ,  $[]$ ,  $->$  могут быть перегружены только методом класса.

# Порядок поиска компилятором перегруженного бинарного оператора

- ① если оба аргумента относятся к базовым типам, то используется встроенный оператор;
- ② если слева стоит операнд пользовательского типа, то компилятор ищет оператор в форме метода класса;
- ③ если перегрузки в форме метода не найдено или слева стоит операнд базового типа, то компилятор ищет перегрузку в форме глобальной функции;
- ④ ошибка при компиляции.

## operator=

Автоматический оператор присваивания генерируется компилятором автоматически.

- поэлементно копирует данные из одного объекта в другой;
- вызывает оператор присваивания базового класса (определенный программистом явно или автоматически);
- вызывает оператор присваивания для встроенных объектов.

Компилятор не генерирует автоматически оператор присваивания в следующих случаях:

- в классе объявлен константный объект;
- в классе объявлена ссылка;
- в базовом классе оператор присваивания private;
- во встроенном объекте оператор присваивания private.

# operator=

Тип возвращаемого оператором присваивания значения.

---

```
class A {  
    int m_a;  
public: ...  
void operator = (const A & other) { m_a = other.m_a  
    ; }  
int main()  
{  
    A a1(1), a2(2), a3;  
    a2 = a1; //???  
    a3 = a2 = a1; //??  
    a3.operator=(a2.operator=(a1));  
}
```

---

Т.к. нет ограничений, то следует учитывать преемственность от базовых типов и предпочитать эффективный вариант.

## operator= возвращаемое значение

---

```
class A {
    int m_a;
public: ...
A operator = (const A & other) { m_a = other.m_a;
    return *this; //???
}
int main()
{
    A a1(1), a2(2), a3;
    a2 = a1; //???
    a3 = a2 = a1; //??
    a3.operator=(a2.operator=(a1));
}
```

---

## operator= возвращаемое значение

---

```
class A {
    int m_a;
public: ...
A& operator = (const A & other) { m_a = other.m_a;
    return *this; //???
}
int main()
{
    A a1(1), a2(2), a3;
    a2 = a1; //???
    a3 = a2 = a1; //??
    a3.operator=(a2.operator=(a1));
}
```

---

# Присваивание и классы с указателями

---

```
class Animal {
    int m_age;
    char *m_name;
public: Animal(int, const char*) { ... };
};

int main()
{
    Animal a1(10, "Bobik");
    Animal a2(15, "Tuzik");
    a1 = a2;
}
```

---

# "Правильный" оператор присваивания

---

```
Animal &Animal::operator=(const Animal &a)
{
    m_age = a.m_age;

    delete [] m_name;
    m_name = new char[strlen(a.m_name)+1];
    strcpy(m_name, a.m_name);
    return *this;
}
```

---

---

```
int main
{
    Animal a1(10, "Bobik");
    Animal a2(15, "Tuzik");
    a1 = a2; //???

    a1 = a1; //???
}
```

---

---

```
Animal &Animal::operator=(const Animal &a)
{
    if(&a == this)
        return *this;

    m_age = a.m_age;
    delete [] m_name;
    m_name = new char[strlen(a.m_name)+1];
    strcpy(m_name, a.m_name);
    return *this;
}
```

---

- оператор присваивания не наследуется;
- если нет явного вызова оператора присваивания базовой части класса, то базовая часть останется прежней;
- может быть `virtual`;

# Оператор присваивания и наследование

---

```
Dog & Dog::operator=(const Dog &r)
{
    if (this != r)
    {
        //копирование базовой части
        Animal::operator=(r);
        *static_cast<Animal*>(this)=r;
        static_cast<Animal&>(*this)=r;

        //копирование производной части
    }
    return *this;
}
```

---

# Конструктор копирования через присваивание

---

```
MyString::MyString( const MyString &r )
{
    m_pStr = 0;
    *this = r;
}
```

---

## move operator=

---

```
int main()
{
    MyString s1("My\u041cMy");
    s1 = MyString("StrStr"); //???
}
```

---

## move operator=

---

```
MyString &MyString :: operator=(MyString &&other)
{
    delete [] m_pStr;
    m_pStr = other.m_pStr;
    other.m_pStr = 0;
    return *this;
}
int main()
{
    MyString s1("My\u041c");
    s1 = std::move(MyString("StrStr"));
}
```

---

# Программирование и алгоритмизация

Санкт-Петербургский государственный политехнический университет

24 марта 2015

# Оператор []

---

```
int main()
{
    MyArray a(10);
    a[0] = 5;
    a.operator[](1) = 65;
}
```

---

# Перегрузка оператора []

---

```
class MyArray
{
int m_size;
int *arr;
};

double &MyArray::operator[](int n)
{
    if (n>=0 && n < m_size)
return &arr[n];
else
{ //генерация исключений
//или
//return &arr[0];
//или
//изменить размер, если оператор находится слева
//от знака равенства.
}
}
```

# Перегрузка оператора []

- перегруженный оператор индексирования должен возвращать не значение элемента, а его адрес, для того чтобы можно было использовать выражение слева от знака равенства;
- для перегруженного оператора индексирования, можно предусмотреть проверку значений;
- для массивов из стандартных типов индекс может быть только целый, а для перегруженного оператора — любым.
- обычно в класс вводят еще один перегруженный константный оператор индексирования.

# константный оператор индексирования

---

```
class A {  
public:  
    double &operator[]( int i );  
    const double &operator[]( int i ) const ;  
};  
  
A a1(10);  
const A a2(10);  
  
int tmp = a1[0]; //???  
tmp = a2[0]; //???  
a1[0] = 5; //???  
a2[0] = 5; //???
```

---

# Перегрузка оператора ++

Как отличить префиксный от постфиксного?

---

```
//A.h
class A {
int m_a;
public:
A(int = 0);
/* FIXME */ operator++(); //префиксной
/*FIXME */ operator++(int unused); //постфиксный
};
//A.cpp
... A::operator++()
{ ++m_a; return ... }

... A::operator++(int) //FIXME
//main.cpp
int main() {
A a1(1), a2;
++a1; //a1.operator++();
a2 = a1++; //a1.operator++(0); }
```

# Перегрузка оператора ++

---

```
//A.cpp
A& A::operator++()
{
    ++m_a;
    return *this;
}

A A::operator++(int)
{
    A a(*this);
    m_a++;
    return a;
}
```

---

# Перегрузка бинарного оператора +

---

```
//A.h
class A {
    int m_a;
public:
    A(int a=0) { m_a = a; }
    A operator+(const A&r);
};

//A.cpp
A A::operator+(const A& r)
{
    return A(m_a + r.m_a);
}
```

---

# Перегрузка бинарного оператора +. Глобальная функция

---

```
//A.h
class A {
int m_a;
public:
A(int a=0) { m_a = a; }
};
A operator+(const A& l, const A& r); //глобальная
функция вне класса

//A.cpp
A operator+(const A& l, const A& r)
{
return A(l.m_a + r.m_a); //???
}
```

---

# Перегрузка бинарного оператора +. Глобальная функция

---

```
//A.h
class A {
    int m_a;
public:
    A(int a=0) { m_a = a; }
    friend A operator+(const A& l, const A& r); // функция - друг
};

A operator+(const A& l, const A& r); // глобальная
// функция вне класса

//A.cpp
A operator+(const A& l, const A& r)
{
    return A(l.m_a + r.m_a); //???
}
```

---

В чем отличие от оператора `+=`?

Как будет выглядеть `operator+` для `MyString`?

# Перегрузка операторов >> и <<

---

```
//C.h
class C
{
    int x; int y;
    friend ostream& operator << (ostream& out, MyClass& C);
    friend istream& operator >> (istream& in, MyClass& C);
};

//C.cpp
ostream& operator << (ostream& out, MyClass& C)
{ return out << "x=" << C.x << "y=" << C.y; }

istream& operator >> (istream& in, MyClass& C)
{ cout << "Enter x:"; in >> C.x;
  cout << "Enter y:"; in >> C.y;
  return in;
}
```

# Внедряемые объекты

Отношение между классами «содержит» (has a).

---

```
//A.h
class A { int m_a;
public:
A(int a=0);
};

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

class C : public B
{ int m_c;
  A m_A;
public:
C(int a, int b, int c);
};

C c(1, 2, 3);
```

Отношение между классами «содержит» (has a).

- компилятор выделяет память для объекта С (с учетом внедренного объекта А);
- вызывает конструктор базового класса В;
- вызывает конструктор внедряемого объекта А;
- вызывает конструктор С.

Разрушение объекта происходит в обратном порядке:  
деструкторы С -> А -> В.

Отношение между классами «содержит» (has a).

---

```
C::C( int a, int b, int c )
{
    //как проинициализировать:
    m_c
    m_b
    m_A
    ???
}
```

---

Рекомендация: в конструктора всегда предпочтите инициализацию вместо присваивания.

Переменные класса инициализируются в порядке объявления в классе, поэтому порядок их следования в списке инициализаторов значения не имеет.

# Ассоциативный массив

- В ассоциативных массивах хранятся пары — ключ/значение.
- Операции поиска, сортировки, добавления, удаления — осуществляются по ключу.
- Ключ в большинстве ассоциативных массивов должен быть уникальный.

---

```
class Pair {
    MyString name; //КЛЮЧ – ФИО
    int phone;
    Pair(): name("NoName"), phone(0) {}
    Pair(const char *key, int num) : name(key), phone(
        num) {}

    bool operator== (const char *key)
    {
        return name == key; //???
    }

    friend class Book;
};
```

---

---

```
class Book {
    Pair **ar;
    int size;
    int capacity;
public:
    Book() { size = capacity = 0; ar = nullptr; }
    int &operator[](const char *);
};
```

---

# Программирование и алгоритмизация

Санкт-Петербургский государственный политехнический университет

07 апреля 2015

# Предварительное неполное объявление класса

Такое объявление называется (forward declaration) и допускает использование ссылки или указателя на пользовательский тип до его полного объявления.

---

```
class A
{
    B* pB; //???
    void F(B &); //???
};

class B
{
    A a;
};
```

---

# Предварительное неполное объявление класса

---

```
class B;
class A
{
    B b; //???
    B* pB; //???
    void F(B &); //???
};

class B
{
    A a;
};
```

---

Двухсвязный список состоит из рекурсивного пользовательского типа данных (структуры или класса) с неполным объявлением, содержащего указатель на предыдущий и следующий элементы.

---

```
class Node
{
    Point m_data;
    Node *pPrev;
    Node *pNext;
};
```

---

# Вложенное объявление вспомогательного класса

---

```
class List
{
    private:
        class Node { ...
            friend class List;
        };
        ...
};
```

---

# Вспомогательный класс обертка

---

```
class Node {
private:
    Point m_data;
    Node *pPrev;
    Node *pNext;
    ~Node();
    Node(Node *p, const Point&);
    friend class List;
};
```

---

# Вспомогательный класс обертка

```
Node::Node( Node *prev , const Point &pt ) : m_data( pt
)
{
    pNext = prev->Next;
    prev->pNext = this;
    pPrev = prev;
    pNext->pPrev = this;
}

Node::~Node()
{
    if( pPrev )
        pPrev->pNext = pNext;
    if( pNext )
        pNext->pPrev = pPrev;
    pPrev = nullptr;
    pNext = nullptr;
}
```

# Класс List

---

```
class List
{
    class Node { ... };
    Node Head; //Node*
    Node Tail;
    unsigned int size;
public:
    List();
    ~List();
    void AddToHead(const Point&);
    bool Remove(const Point&);

};
```

---

# Класс List

---

```
List::List()
{
    Head.pNext = &Tail;
    Tail.pPrev = &Head;
    m_size = 0;
}
void List::AddToHead(const Point &pt)
{
    new Node(&Head, pt);
    m_size++;
}
```

---

# Класс List. Метод Remove

---

```
bool List::Remove(const Point &pt)
{
    Node *ptr = Head.pNext;
    for(int i = 0; i < m_size; i++)
    {
        Node *n = ptr->pNext;
        if(ptr->m_data == pt)
        {
            delete ptr;
            return true;
        }
        ptr = n;
    }
    return false;
}
```

---

- характеризуют количество или взаимосвязь всех существующих на данный момент объектов данного типа;
- имеют отношение к классу в целом, но не являются частью объекта данного класса.

## Специфика:

- Статические переменные класса размещаются в статической области памяти, в независимости от того где располагается объект. Существуют в единственном экземпляре.
- Статическую переменную необходимо явно определить независимо от спецификатора доступа.
- При обращении к статической переменной класса, можно обращаться к ней, как к глобальной переменной заключенной в пространство имен (без создания объекта), но т.к. она входит в класс, можно работать с ней и посредством созданного объекта.

- Не следует инициализировать статические переменные в конструкторе, иначе есть опасность каждый раз при обращении получать одно и тоже значение.
- В константных методах, можно манипулировать статические переменные класса.

# Счетчик объектов

---

```
//A.h
class A {
    int m_a; //обычная переменная класса
public:
    static int count;
    A() { count++; }
    ~A() { count--; }

};

//A.cpp
#include "A.h"

int A::cout = 0; //выделение памяти под переменную
и ее инициализация
//если память не выделить, то будет ошибка при
линковке
```

# Счетчик объектов

---

```
#include "A.h"
int main()
{
    size_t n = sizeof(A); //???
    std::cout << A::count;
    A a1;
    std::cout << a1.count;
    {
        A a2=a1;
        std::cout << a2.count; //???
    }
    std::cout << A::count; //???
}
```

---

# Программирование и алгоритмизация

Санкт-Петербургский государственный политехнический университет

14 апреля 2015

# Встроенные статические объекты

---

```
class A {  
    A a; //ошибка использования неопределенного  
          //класса A  
};  
  
class A {  
    static A a; //OK  
};
```

---

# Альтернатива значениям параметров по умолчанию

Преимущество использования статических полей заключается в том, что такое значение можно заменить в процессе работы программы.

---

```
//Date.h
class Date {
    int day, month, year;
    static Date default; //дата запуска приложения

public:
    Date() { }; //FIXME
};
```

---

# Альтернатива значениям параметров по умолчанию

---

```
#include <ctime>
#include "Date.h"
Date Date::default = Init();
Date Init()
{
    time_t cur;
    time(&cur);
    tm *t = localtime(&cur);
    return Date(t->tm_day, t->tm_month+1, t->tm_year)
};

}
```

---

# Целочисленные статические константы

---

```
class A
{
    int a1;
    const int a2;
    static int a3;
    static const int a4 = 60;
};
```

---

Статические методы являются фактически глобальной функцией, область видимости которой ограничена именем класса. При наличии прав доступа можно просто вызвать из функции: `имя_класса::имя_функции()`;  
При необходимости можно вызвать как обычный метод у объекта.

- в таких функциях указатель `this` не существует;
- обратиться к нестатическим данным класса из этой функции невозможно;
- статическая функция не может быть `virtual`.

# Статические методы и защищенные поля

---

```
//X.h
class X
{
    static int count;
public:
    X() { count++; }
    static int GetCount() { return count; }
};
```

---

# Статические методы и защищенные поля

---

```
#include "X.h"

int X::count; //???
int main()
{
    cout << X::count; //???
    cout << X::GetCount(); //???
    X x1;
    x1.GetCount(); //???
}
```

---

# Обработка исключительных ситуаций

При обработке рассматриваются только ситуации внутреннего характера (нет памяти в области heap, не найден файл, переполнение и т.д.). Ситуация созданная нажатием Ctrl-C считается внешней.

# Обработка исключительных ситуаций

Синтаксис исключений:

---

```
try {  
    ...  
}
```

---

Обозначает контролируемый блок кода.

# Обработка исключительных ситуаций

Генерация исключений:

---

**throw [ выражение ];**

---

# Обработка исключительных ситуаций

Обработка исключений:

---

```
catch(тип имя) { /*тело обработчика*/ }
catch(тип) { /*тело обработчика*/ }
catch(...) { /*тело обработчика*/ }
```

---

# Обработка исключительных ситуаций

Обработка исключений:

---

```
catch( int i ) { //int }
catch( const char* ) { //const char* }
catch( Overflow ) { //класс Overflow }
catch( ... ) { /*обработка всех исключений*/ }
```

---

# Обработка исключительных ситуаций

После обработки исключения управление передается первому оператору находящемуся за блоком исключений. Туда же, минуя код всех обработчиков, передается управление, если исключение в try-блоке не было сгенерировано.

При генерации исключения в C++ происходят следующие действия:

- создается копия параметра `throw` в виде объекта, который существует до тех пор, пока исключение не будет обработано;
- вызываются деструкторы объектов выходящих из области действия;
- передается объект и управление обработчику имеющему совместимый тип с этим объектом.

При генерации исключений:

- создается копия параметра `throw` в виде объекта, который существует до тех пор, пока исключение не будет обработано;
- вызываются деструкторы объектов выходящих из области действия;
- передается объект и управление обработчику имеющему совместимый тип с этим объектом.

Обработчик считается найденным, если:

- тип объекта в обработчеке тот же, что и указан после `throw`, т.е. `T`, `const T`, `T&` или `const T&`, где `T` — тип исключения;
- является производным от указанного в параметре `catch`, если наследования производилось с ключом `public`;
- является указателем который может быть преобразован к нужному типу, например `void*` .

# Обработка исключительных ситуаций

Если происходит вызов непредусмотренного исключения, то вызывается функция `unexpected()`, реализацию которой можно заменить при помощи `set_unexpected()`, если такой функции нету, то вызывается функция `terminate()`, реализацию которой можно заменить при помощи `set_terminate()`, если такой функции нету то происходит вызов функции `abort()`.

# Обработка исключительных ситуаций

Типы исключений которые может генерировать функция, перечисляются после ключевого слова `throw` после прототипа функции.

---

```
void f1() throw (int, const char*) /* может
    генерировать только типов int или char* */
void f2() throw (Oops*) /* исключения типа
    указатель на класс */
```

---

# Обработка исключительных ситуаций

---

```
void f1() throw ()  
{  
    // Тело функции, не порождающей исключений  
}
```

---

# Обработка исключительных ситуаций

При переопределении виртуальной функции можно задавать список исключений такой же или более ограниченный чем в базовом классе.

Если функция вызывает не описанное исключение, вызывается функция `unexpected()`.

# Исключения в конструкторах и деструкторах

---

```
class Vector {
public:
    class Size {};
enum {max = 32000};
Vector(int n)
{ if(n<0 || n > max) throw Size(); }
};

try {
    Vector *p = new Vector(i);
    ...
}
catch(Vector::Size) { //Обработка ошибки размера
    вектора }
```

---

Если в конструкторе генерируется исключение то автоматически вызываются деструкторы для полностью созданных в этом блоке объектов. Например, если исключение было вызвано при создании массива объектов, то деструкторы будут вызваны только для успешно созданных элементов. Если память выделяется динамически с помощью операции new и в конструкторе возникает исключение, память из-под объекта корректно освобождается.

# Стандартные исключения

- `bad_alloc`
- `bad_cast`
- `bad_typeid`
- `bad_exception`

## Файловые потоки:

- ifstream — входной файловый поток
- ofstream — выходной файловый поток
- fstream — двунаправленный

---

```
char buf[100];
ifstream fff("file.txt", ios::in | ios::nocreate);
if(!fff)
cout << "err";
while(!fff.eof())
fff.getline(buf, 100);
```

---

## Строковые потоки:

- `istringstream` — входные строковые потоки;
- `ostringstream` — выходные строковые потоки;
- `stringstream` — двунаправленные строковые потоки.

---

```
#include <sstream>
ostringstream os;
time_t t;
time(&t);
os << "time:" << ctime(&t);
```

---

---

```
friend ostream& operator << (ostream& out, MyClass&
    C)
{ return out << "x=" << C.x << "y=" << C.y; }

friend istream& operator >> (istream& in, MyClass&
    C)
{ cout << "Enter x:"; in >> C.x;
  cout << "Enter y:"; in >> C.y;
  return in;
}
```

---

# Программирование и алгоритмизация

Санкт-Петербургский государственный политехнический университет

05 мая 2015

# Шаблоны классов

---

```
class Queue {
public:
    Queue();
    ~Queue();

    int& remove();
    void add( const int & );
    bool is_empty();
    bool is_full();

private:
};

Queue qi;
```

---

Шаблоны используются для:

- Обобщения действий (шаблоны функций).
- Обобщение наборов данных (шаблоны классов).

Главное достоинство — позволяет уменьшить количество “дублирующегося кода”. Используемые типы становятся известны на этапе компиляции, поэтому компилятор проверяет соответствие (type safe).

# Шаблоны классов

---

```
template <typename Type>
class Queue {
public:
    Queue();
    ~Queue();

    Type& remove();
    void add( const Type & );
    bool is_empty();
    bool is_full();

private:
};

Queue<int> qi;
```

---

**template < список\_параметров\_шаблона >** объявление  
функции или класса .

---

- Параметры — типы. `<typename T>`.
- Параметры — `int`, `enum`, указатель, ссылка `<typename T, int i>`.
- Параметры — шаблоны `<template <typename T> >`.

# Специфика параметров шаблона

- Видя объявление шаблона, компилятор не создает никакого кода. Только встретив обращение к данному шаблону (вызов функции или создание экземпляра класса), компилятор сгенерирует соответствующий код.
- Для обозначения обобщенного параметра рекомендуется использовать ключевое слово template вместо class.
- В качестве обобщенного типа можно задать как имя пользовательского типа данных, так и базового.
- Можно задавать значение параметров шаблона по умолчанию.

Часто говоря о шаблонах употребляют следующие термины:

Инстанцирование — подстановка реальных типов в качестве параметров шаблона.

Специализация — версия шаблона для конкретного набора параметров

# Шаблоны классов

---

```
template<class T = char> class String;  
String ◇ *p;
```

---

# Шаблоны функций

---

```
template<typename Data>
class List
{
public:
void print();
};

template <typename Data> void List<Data>::print();
```

---

# Шаблоны классов

---

```
template< template<typename U> typename V> class C
{
    V<int> y;
    V<int*> y1;
};
```

---

# Шаблоны классов

---

```
#include <iostream>
template <int N>
struct Factorial{
    enum { val = Factorial<N-1>::val * N };
};

template<>struct Factorial<0>{
    enum { val = 1 };
};

int main(){
    std::cout << Factorial<4>::val << "\n"; } //на этапе
                                                 //компиляции
```

---

# Шаблоны функций

---

```
template<class T> void f() { T::x * p; ... }
template<class T> void f() { typename T::x * p; ...
}
```

---

# Специализация шаблонной функции

---

```
template <typename T> const T& min( const T &a,
    const T &b)
{
    return (a < b) ? a : b;
}

// явная специализация
template<> const char * &
min<const char*>(const char * &, const char * &)
{
    return /* FIXME */ ? a : b;
}
```

---

# Пример шаблонного класса

```
template <typename T, int size> class MyArray
{
T ar[size];
public:
/* */ operator [] (int i);
};

template<class T, int size>/> /* MyArray<T,
size >:: operator [](int i)
{ /* FIXME */

}

int main()
{
MyArray <int , 5> a1;
MyArray <Rect , 10> a2;
MyArray <MyString , 2> a3;
```

# Не стоит злоупотреблять шаблонами

```
.. / global.h:145: error: in passing argument 3 of '  
    int vstd :: findPos(const std :: v  
ector<T1, std :: allocator<_CharT> >&, const T2&,  
    Func&) [with T1 = std :: pair<cons  
t CGHeroInstance*, CPath*>, T2 = const  
    CArcmedInstance*, Func = boost :: _bi :: bind_<bool, bool (*)(const std :: pair<const  
    CGHeroInstance*, CPath*>&, const CGHeroIn  
stance* const std :: pair<const CGHeroInstance*,  
    CPath*>::*, const CArcmedInstance*  
const&), boost :: _bi :: list3<boost :: arg<1> (*)(),  
    boost :: _bi :: value<const CGHerol  
nstance*std :: pair<const CGHeroInstance*, CPath  
*>::*>, boost :: arg<2> (*)()> > ]'
```

# Спецификаторы класса памяти

- auto
- register
- static
- extern

# Спецификатор volatile

```
volatile int i;
```

## Операция const\_cast.

---

```
void print(int *p) { cout << *p; }
const int *p;
print(p); //ошибка
int *j = const_cast<int*>(p);
```

---

Операция `dynamic_cast`. Преобразования бывают двух типов:

- повышающее — приведение производного класса к базовому
- понижающее — из базового класса в производный
- перекрестное — приведение между производными типами

# dynamic\_cast

Повышающее преобразование.

---

```
class B{ ... };
class C : public B{ ... };
C *c = new C;
B *b = dynamic_cast<B*>(c); //Эквивалентно B *b = c;
```

---

## dynamic\_cast

Поникающее преобразование. Применяется когда компилятор не может проверить правильность приведения. Для использования данного типа преобразований необходимо включить механизм RTTI.

---

```
class B{ public: virtual void f1() {} };
class C : public B{ public: void f2() {} };
C *c = new C;
B *b = new B;
C *temp = dynamic_cast<C*>(b);
if (temp)
    temp->f2();
```

---

# dynamic\_cast

Перекрестное преобразование.

---

```
class B{ public: virtual void f1() {} };
class C : public B{ public: void f2() {} };
class D : public B{ ... };
D *d = new D;
C *temp = dynamic_cast<C*>(d);
if(temp)
    temp->f2();
```

---

## dynamic\_cast

Для доступа к RTTI введена операция typeid и класс type\_info.

---

```
class B { ... };
class C : public B { ... };
B *p = new C;
if(typeid(*p) == typeid(C))
{
    dynamic_cast<C*>(p)->f2();
    cout << typeid(*p).name();
}
```

---

## static\_cast

Выполняется на этапе компиляции над:

- целыми типами
- целыми и вещественными типами
- целыми и перечисляемыми типами
- указателями и ссылками одной иерархии

## `reinterpret_cast`

Применяется для преобразования не связных между собой типов, например указателей в целые и наоборот.

---

```
char *p = reinterpret_cast<char*>(malloc(10));
```

---

# Программирование и алгоритмизация

Санкт-Петербургский государственный политехнический университет

26 мая 2015

## Операция const\_cast.

---

```
void print(int *p) { cout << *p; }
const int *p;
print(p); //ошибка
int *j = const_cast<int*>(p);
```

---

Операция `dynamic_cast`. Преобразования бывают двух типов:

- повышающее — приведение производного класса к базовому
- понижающее — из базового класса в производный
- перекрестное — приведение между производными типами

# dynamic\_cast

Повышающее преобразование.

---

```
class B{ ... };
class C : public B{ ... };
C *c = new C;
B *b = dynamic_cast<B*>(c); //Эквивалентно B *b = c;
```

---

## dynamic\_cast

Поникающее преобразование. Применяется когда компилятор не может проверить правильность приведения. Для использования данного типа преобразований необходимо включить механизм RTTI.

---

```
class B{ public: virtual void f1() {} };
class C : public B{ public: void f2() {} };
C *c = new C;
B *b = new B;
C *temp = dynamic_cast<C*>(b);
if (temp)
    temp->f2();
```

---

# dynamic\_cast

Перекрестное преобразование.

```
class B{ public: virtual void f1() {} };
class C : public B{ public: void f2() {} };
class D : public B{ ... };
D *d = new D;
C *temp = dynamic_cast<C*>(d);
if(temp)
    temp->f2();
```

## dynamic\_cast

Для доступа к RTTI введена операция typeid и класс type\_info.

---

```
class B { ... };
class C : public B { ... };
B *p = new C;
if(typeid(*p) == typeid(C))
{
    dynamic_cast<C*>(p)->f2();
    cout << typeid(*p).name();
}
```

---

## static\_cast

Выполняется на этапе компиляции над:

- целыми типами
- целыми и вещественными типами
- целыми и перечисляемыми типами
- указателями и ссылками одной иерархии

## `reinterpret_cast`

Применяется для преобразования не связных между собой типов, например указателей в целые и наоборот.

---

```
char *p = reinterpret_cast<char*>(malloc(10));
```

---

Класс `string` входит в стандартную библиотеку C++.

---

```
string s1("Text");
string s2;
```

```
char *str = "Txt";
string s3(str);
```

---

## Функции:

---

```
s.at(1);  
s1.append(s2); // +  
s1.append(s2, 3, 5);  
s1.insert(1, str2, 3, 5);  
s1.replace(1, 2, str2, 5);  
s1.swap(s2);  
s1.c_str(); //возвращает const char*  
s1.find(s2); //возвращает позицию  
s1.compare(1, 2, s2, 1, 2); //аналог strstr
```

---

## Последовательные контейнеры:

- `vector` — структура, эффективно реализующая произвольный доступ к элементам, а также добавление в конец и удаление из конца;
- `dequeue` — структура, эффективно реализующая произвольный доступ к элементам, а также добавление в оба конца и удаление из обоих концов;
- `list` — список, эффективно реализует вставку и удаление элементов в произвольное место, но не имеет произвольного доступа к своим элементам.

---

```
vector<int> v;
v.push_back(5);
v.push_back(6);
v.push_back(7);
for( vector<int>::iterator i = v.begin(); i != v.
    end(); i++)
{ cout << *i << " "; }
```

---

Функции:

---

```
insert() //вставка в произвольное место  
erase() //удаление из произвольного места  
at, [] //произвольный доступ к элементу
```

---

Ассоциативные контейнеры:

- map — словарь
- multimap — словарь с дубликатами

# std::map

---

```
map<int ,int> maps;
maps[100] = 1;
maps[200] = 3;
```

---

# Стандартные имена типов

```
template<typename T> class MyVector {
```

```
    typedef T value_type;
```

```
    ...
```

```
};
```

```
int main()
```

```
{
```

```
    int k1;
```

```
    MyVector<int>::value_type k2;
```

```
}
```

# Цель введения стандартных имен типов

Каждый контейнер:

- с помощью `typedef` дает стандартные имена используемым типам
- определяет эти типы своим способом, наиболее подходящим для реализации

=> Все псевдонимы во внешний мир выглядят **одинаково!**

# Псевдонимы контейнеров STL

<b>value_type</b>	тип элемента контейнера
<b>allocator_type</b>	тип распределителя памяти
<b>size_type</b>	тип индексов, счетчика элементов и т.п. (эквивалент sizeof() элемента)
<b>difference_type</b>	тип результата вычитания адресов двух элементов
<b>iterator</b>	ведет себя подобно value_type*
<b>const_iterator</b>	ведет себя подобно const value_type*
<b>reverse_iterator</b>	просматривает контейнер в обратном порядке ведет себя как value_type*
<b>const_reverse_iterator</b>	ведет себя как const value_type*
<b>reference</b>	ведет себя подобно value_type&
<b>const_reference</b>	ведет себя подобно const value_type&

+ специализированные эквиваленты для некоторых типов контейнеров

# Пример использования псевдонима value\_type

Требуется реализовать функцию которая  
будет суммировать элементы **любого**  
контейнера

# Пример использования псевдонима value\_type

```
template<typename C> typename C::value_type sum( C& c
                                                // typename обязательно
{
    /*typename*/ C::value_type s = C::value_type();
                // typename необязательно
    /*typename*/ C::iterator it = c.begin();
    while(it != c.end() )
    {
        s += *it;
        ++it;
    }
    return s;
}
```

# Пример использования псевдонима value\_type

```
int main()
{
    MyVector<int> v(10,1);
    int res = sum(v); //???

    MyVector<double> v1(10,1.1);
    double res1 = sum(v1); //???

    MyVector<MyString> v2(10, MyString("A"));
    MyString res2 = sum(v2); //???

}
```

# Метод класса - шаблон

- Обычные методы (в частности, конструктор) могут быть шаблонными – базирующимися на другом типе параметра
- Виртуальные методы **не** могут быть шаблонными

# Зачем нужны шаблонные методы класса

```
template <class T, int size> class MyArray
{
    T m_ar[size];
public:
    ...
    void copy(T* p, size_t num)
    {
        int n = (num<size) ? num : size;
        for(int i = 0; i<n; i++) { m_ar[i] = p[i];}
    }
};
```

# Зачем нужны шаблонные методы класса

```
{  
    MyArray<int, 5> a;  
    a[0] = 5.5; //OK  
    int iar[20] = {4,7,...};  
    a.copy(iar,20); //OK  
    double dar[10] = {1.1, 2.2, 3.3,...};  
    a.copy(dar,10); //ошибка  
}
```

# Зачем нужны шаблонные методы класса

```
template <class T, int size> class MyArray
```

```
{
```

```
    T m_ar[size];
```

```
public:
```

```
...
```

```
template<typename Other> void copy(Other* p, size_t num)
```

```
{
```

```
    int n = (num<size) ? num : size;
```

```
    for(int i = 0; i<n; i++) { m_ar[i] = p[i];}
```

```
}
```

```
};
```

# Виртуальные методы шаблонных классов

- шаблоны могут участвовать в наследовании (при этом перемежаться с нешаблонными классами)
- методы шаблонных классов (не шаблонные) могут быть виртуальными

# Виртуальные методы

```
template<typename T> class A{
    T m_a;
public: A(const T& a){m_a = a;}
    virtual void f(){m_a++;}
};

template<typename T> class B:public A<T>{
    T m_b;
public: B(const T& a, const T& b):A<T>(a) {m_b = b;}
    virtual void f(){m_b++;}
};

int main()
{
    B<int>* pB = new B<int>(1,5);
    A<int>* pA = new B<int>(1,5);
    pA->f();
    pB->f();
}
```

# typename

```
class A{
public:
    class B{
        ...
    };
};

template<typename T> class C{
typename T::B* p;
};

C<A> ccc;
```

# Специализация шаблонного класса

- `template<> class MyArray<int,10>;`
- `template class MyArray<int,10>;`
- `template int&`  
`MyArray<int,10>::operator[](int);`

# **STL**

vector

**Header: <vector>**

**Namespace: std**

```
template<class _Ty,  
         class _Ax = allocator<_Ty>>  
    class vector  
    : public _Vector_val<_Ty, _Ax>  
    {...};
```

# Распределитель памяти (allocator<T>)

Отвечает за **выделение, освобождение, перераспределение** памяти.

- В большинстве задач используется значение по умолчанию
- Для специфических задач программисту предоставляется возможность реализовать собственный алокатор

## Класс `_Vector_val`

Содержит встроенный объект  
распределителя памяти (**алокатора**)

# Класс \_Vector\_val

Посредством функциональности базового класса **отделены** друг от друга:

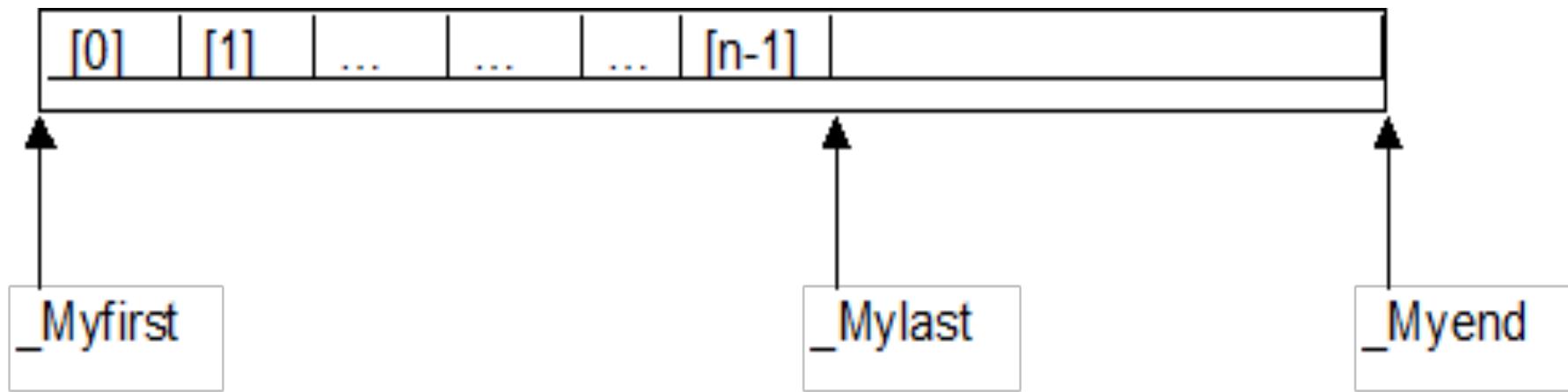
- операции захвата памяти и инициализации
- операции освобождения памяти и деинициализации

А также операции **перераспределения** памяти осуществляются без вызова operator= для элементов контейнера

# Класс \_Vector\_val

Выделение манипуляций с памятью в отдельный класс позволяет использовать **другие механизмы** работы с памятью (не только heap, а также разделяемая память...)

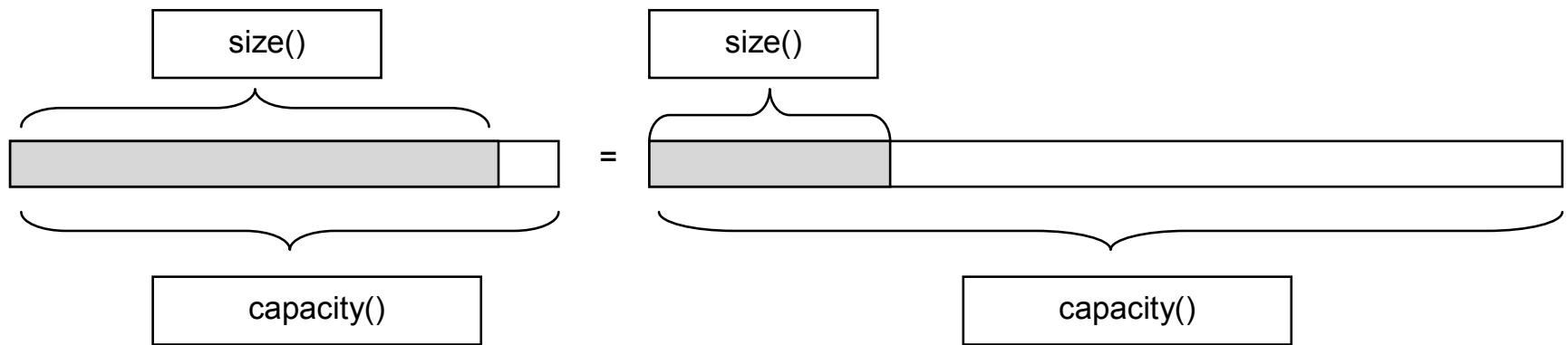
# Внутреннее устройство vector



# Размеры vector

- **size()** //???
- **capacity()** //???

# Оптимизации операций



# Конструкторы

**vector( );**

**explicit vector( size\_type \_Count );**

**vector( size\_type \_Count, const Type& \_Val );**

**vector( const vector& \_Right );**

**template<class InputIterator> vector(**  
**InputIterator \_First, InputIterator \_Last );**

**vector( vector&& \_Right );**

# Методы, связанные с размером

- **size\_type size() const;**
- **size\_type capacity () const;**
- **void reserve(size\_type n);**
- **void resize(size\_type n);**
- **void resize(size\_type n, T& x);**
- **size\_type max\_size() const;**
- **bool empty() const;**

# **resize() и reserve()**

1.

```
vector<int> v;  
v.resize(10);  
size_t n = v.size(); //???
```

2.

```
vector<int> v;  
v.reserve(10);  
size_t n = v.size(); //???
```

# Получение/модификация значений элементов

- reference **at(size\_type pos);**
- reference **operator[](size\_type pos);**
- reference **front();**
- reference **back();**

# at() и operator[]

```
vector<int> v(10,1);
```

- Распечатать значения элементов  
???
- Получить/изменить значение i-ого элемента

```
int i;  
std::cin>>i;
```

# Вставка/удаление последнего элемента

- void **push\_back**(const T& x);
- void **pop\_back**();

1.

```
vector<int> v;  
v.resize(10);  
v.push_back(1);  
size_t n = v.size(); //???
```

2.

```
vector<int> v;  
v.reserve(10);  
v.push_back(1);  
size_t n = v.size(); //???
```

Нет операций с началом  
последовательности

Нет

`push_front(), pop_front()`

# Использование итераторов

- iterator begin();
- const\_iterator begin() const;
- iterator end();
- iterator end() const;
- reverse\_iterator rbegin();
- const\_reverse\_iterator rbegin() const;
- reverse\_iterator rend();
- const\_reverse\_iterator rend() const;

# Пример: прохождение последовательности в обратном порядке с помощью "прямого" итератора

```
char ar[] = "QWERTY";
```

```
//создать вектор, элементы которого должны быть  
копиями символов массива
```

```
...
```

```
//создать итератор для вектора
```

```
...
```

```
//вывести значения элементов вектора с помощью  
итератора
```

# Пример: прохождение последовательности в обратном порядке с помощью "прямого" итератора

```
char ar[] = "QWERTY";
vector<char> v(ar, ar+sizeof(ar) -1);
vector<char>::iterator it=v.end();
while(it != v.begin())
{
    it--;
    cout<<*it<<" ";
}
cout<<endl;
```

# Пример: прохождение последовательности в обратном порядке с помощью «реверсивного» итератора

```
char ar[] = "QWERTY";
vector<char> v(ar, ar+sizeof(ar) -1);
vector<char>::reverse_iterator rit=v.rbegin();
while(rit!=v.rend())
{
    cout<<*rit<<" ";
    rit++; //!!!!
}
cout<<endl;
```

# Замещение

```
void assign(const_iterator first,  
           const_iterator last);
```

```
void assign(size_type n, const T& x);
```

# Замещение

```
vector<int> v1(10,1);
```

```
vector<int> v2(11,12);
```

```
int ar[] = {5,7,1,-6...};
```

```
//заменить элементы v1 на элементы v2
```

```
...
```

```
//заменить элементы v1 на элементы ar
```

# Вставка

**iterator insert(iterator it, const T& x);**

**void insert(iterator it, size\_type n, const T& x);**

**void insert(iterator it, const\_iterator first,  
const\_iterator last);**

# Пример insert()

```
vector<int> v;  
//формируем значения элементов таким  
образом, чтобы содержимое вектора стало  
1,2,3,4  
  
//требуется вставить между элементами  
значение 33 - 1,33,2,33,3,33,4
```

# Приимер insert()

```
vector<int>::iterator it =v.begin();  
for(int i=0; i<v.size(); i++ ) //???  
{  
    //вставка  
}
```

# Приимер insert()

```
vector<int>::iterator it =v.begin();  
int size = v.size();  
for(int i=0; i<size; i++ )  
{  
    ++it;  
    it = v.insert(it, 33);  
    ++it;  
}
```

# удаление

iterator **erase**(iterator it);

iterator **erase**(iterator first, iterator last);

void **clear**();

# Пример erase()

А теперь требуется удалить все 33

# Пример erase()

```
vector<int>::iterator it =v.begin();  
while(it != v.end())  
{  
    if(*it==33)  
    { it = v.erase(it);}  
    else {++it;}  
}
```

# Уменьшение емкости вектора

```
void shrink_to_fit( );
```

# Обмен двух векторов

```
void swap(vector<Type, Allocator>& x);
```

# Создание двухмерных массивов посредством vector

- `vector<vector<int>> vv;`
- или посредством `typedef`:  
**typedef** `vector<int> VECT_INT;`  
`vector<VECT_INT> vv;`

Замечание: в отличие от обычного массива  
строки такого вектора могут быть разной  
длины!

# Примеры

1.???

```
vector< vector<int> > vv1(10, vector<int>(10,1));
```

2.

```
vector< vector<int> > vv2; //???
```

```
vv2[2][5] = 1; //???
```

```
vv2.at(2).at(5) = 1; //???
```

3.

```
int ar[10][3] = {{1,2,3},{4,5,6},...};
```

//Создать вектор векторов таким образом, чтобы элементы каждого вектора стали копиями элементов соответствующей строки массива

# Для итератора vector перегружен operator->

```
class A{ int m_a;
public: A(int a=0){m_a = a;}
    int GetA(){return m_a;}
};

int main()
{
    vector<A> v(10, A(1));
    vector<A>::iterator it = v.begin();
    while(it != v.end())
    {
        std::cout<< (*it).GetA();
        std::cout<< it->GetA();
        ++it;
    }
}
```

# Если вектор содержит указатели

```
{  
    vector<MyString*> v;  
    v.push_back(new MyString("AAA"));  
    v.push_back(new MyString("BBB"));  
    ...  
    //Печать???  
    ...  
}///???
```

# Сравнение списка и вектора

## Достоинства списка:

- Операции вставки/удаления
- Не требуют перераспределения памяти

## Недостатки списка:

- Только последовательный доступ
- Время доступа к разным элементам разное
- Дополнительные затраты памяти на «обертку» для каждого данного

# Шаблон двухсвязного списка

```
template<typename T> class Node{
    Node<T>* pNext, *pPrev;
    T data;
    ...
    template<typename T> friend class List;
};

template<typename T> class List{
    Node<T> Head;
    Node<T> Tail;
    size_t m_size;
    ...
};
```

# Итератор для списка

```
class iterator{
    Node* p;
public:
    iterator(){???
    iterator(Node* pT){???
    T& operator*() {???
    iterator& operator++(){ ???
    iterator operator++(int){ ???
    bool operator==(iterator& r){???
    bool operator!=(iterator& r){???
};
```

# Пример использования итератора

```
MyList<int> l;  
l.push_back(1);  
l.push_front(2);  
l.push_back(3);  
l.push_front(4);  
MyList<int>::iterator it=l.begin();  
for(; it!=l.end(); ++it)  
{  
    cout<<*it<<" ";  
}
```

# Методы списка для получения итераторов

**iterator begin()**

{???

**iterator end()**

{???

# Шаблонный конструктор списка

```
template<typename IT> MyList(IT first, IT last)
```

```
{
```

```
???
```

```
}
```

# Шаблонный конструктор списка

```
template<typename IT> MyList(IT first, IT last)
{
    Head.pNext = &Tail;
    Tail.pPrev = &Head;
    m_size=0;
    while(first != last)
    {
        push_back(*first);
        ++first;
    }
}
```

# Пример создания списка по любой другой последовательности

```
int ar[10] = {5,2,3,...};
```

```
MyList<int> l1(??? );
```

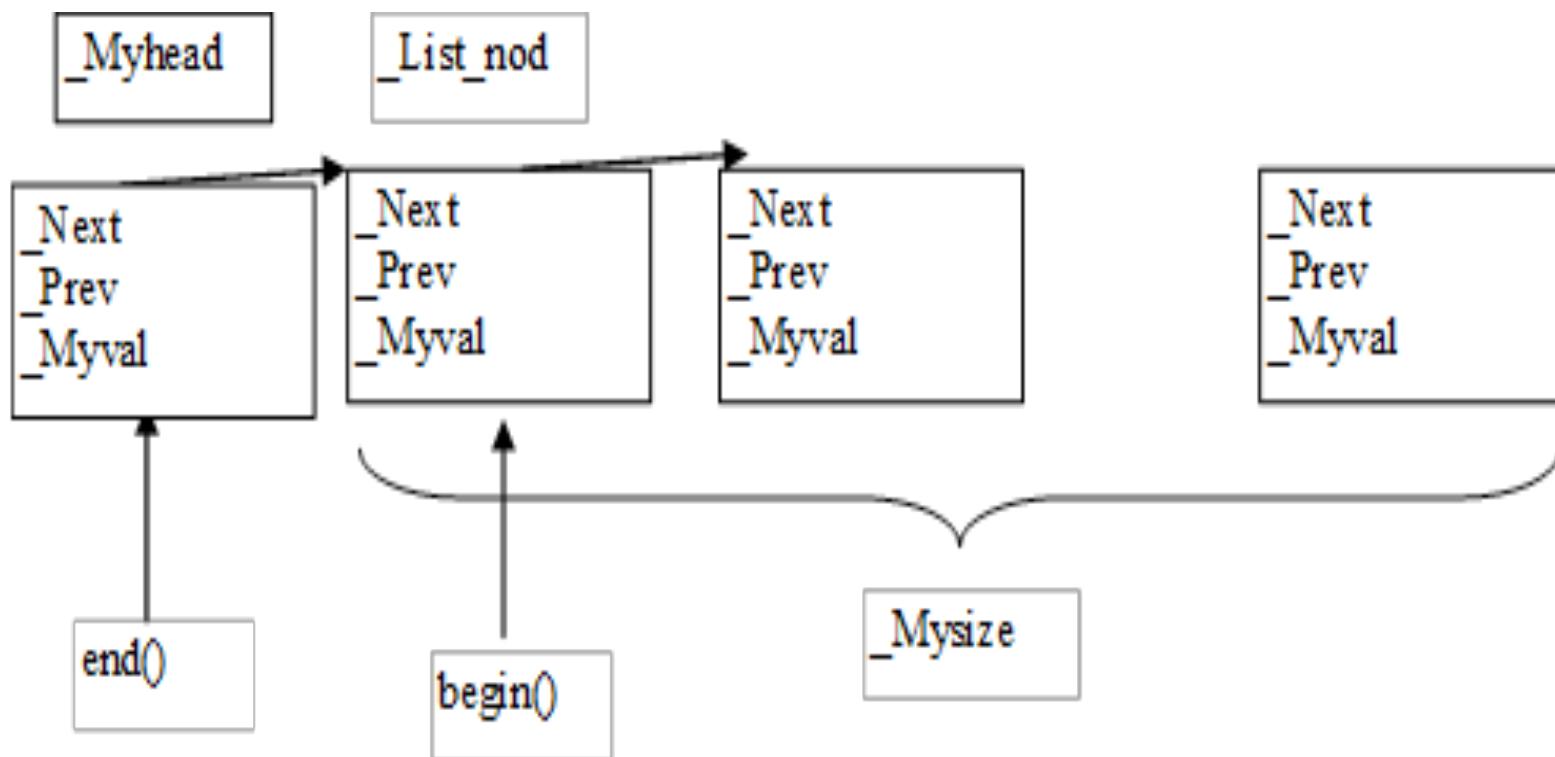
```
MyVector<int> v(10,1);
```

```
MyList<int> l2(??? );
```

# STL

list

# Внутреннее устройство list



Header <list>  
namespace std

```
template<class _Ty,  
         class _Ax = allocator<_Ty>>  
class list : public _List_val<_Ty, _Ax>  
{ // bidirectional linked list  
...  
};
```

# Конструкторы

- `list( );`
- `explicit list( const Allocator& _Al );`
- `explicit list( size_type _Count );`
- `list( size_type _Count, const Type& _Val );`
- `list( size_type _Count, const Type& _Val, const Allocator& _Al );`
- `list( const list& _Right );`
- `template<class InputIterator> list( InputIterator _First, InputIterator _Last );`
- `template<class InputIterator > list( InputIterator _First, InputIterator _Last, const Allocator& _Al );`
- `list( list&& _Right );`

# Нет произвольного доступа

Поэтому **не** реализованы:

- operator[]
- at

# Не требуется резервирование

Поэтому **нет**:

- capacity()
- reserve()

# Специфические для list операции

Перемещение элементов из одного списка в другой без копирования node-ов (**только за счет изменения указателей**)

- `void splice(iterator it, list& x);`
- `void splice(iterator it, list& x, iterator first);`
- `void splice(iterator it, list& x, iterator first, iterator last);`

# splice()

```
list<int> l; //1,2,3,4  
  
list<int> l1; //11,22,33,44  
//1  
l.splice(l.begin(),l1); //11 22 33 44 1 2 3 4  
//2  
l.splice(l.begin(),l1,l1.begin()); //11 1 2 3 4  
//3  
list<int>::iterator it1=l1.begin();  
++it1;  
++it1;  
l.splice(l.begin(),l1,it1,l1.end()); //33 44 1 2 3 4
```

# Специфические для list операции

**сортировка реализована только методом класса!**

- void **sort()**; //по возрастанию (operator<)
- template<class Pred> void **sort(Pred pr);**  
//!pr(\*Pj, \*Pi)

# sort()

1.

```
list<int> l;//6,-2,3,-4,1  
l.sort(); //???
```

2. Как отсортировать по модулю?

# sort()

```
bool Mod(int left, int right)  
{ return abs(left)<abs(right);}
```

```
int main()  
{  
    list<int> l;//6,-2,3,-4,1  
    l.sort(Mod);  
}
```

# Специфические для list операции

## объединение отсортированных списков:

- void **merge**( list<Type, Allocator>& \_Right );
- template<class Traits> void **merge**( list<Type, Allocator>& \_Right, Traits \_Comp );

# merge()

```
list<int> l; //1,2,3,55
```

```
list<int> l1; //11,22,33,44
```

```
l1.merge(l); //???
```

```
//l ???
```

# Операции с началом последовательности

В дополнение к push\_back(), pop\_back():

- void **push\_front**(const T& x);
- void **pop\_front**();

# Специфические для list операции

- void **remove**( const Type& \_Val );
- template<class Predicate> void **remove\_if**(  
Predicate \_Pred )

# `remove()`, `remove_if()`

1.

```
list <int> l; //3,5,-2,5  
l.remove(5);
```

2. Удалить все отрицательные

# remove\_if()

```
bool Neg(int x)
{ return x<0;}
Int main()
{
    list <int> l; //3,5,-2,5
    l.remove_if(Neg);
}
```

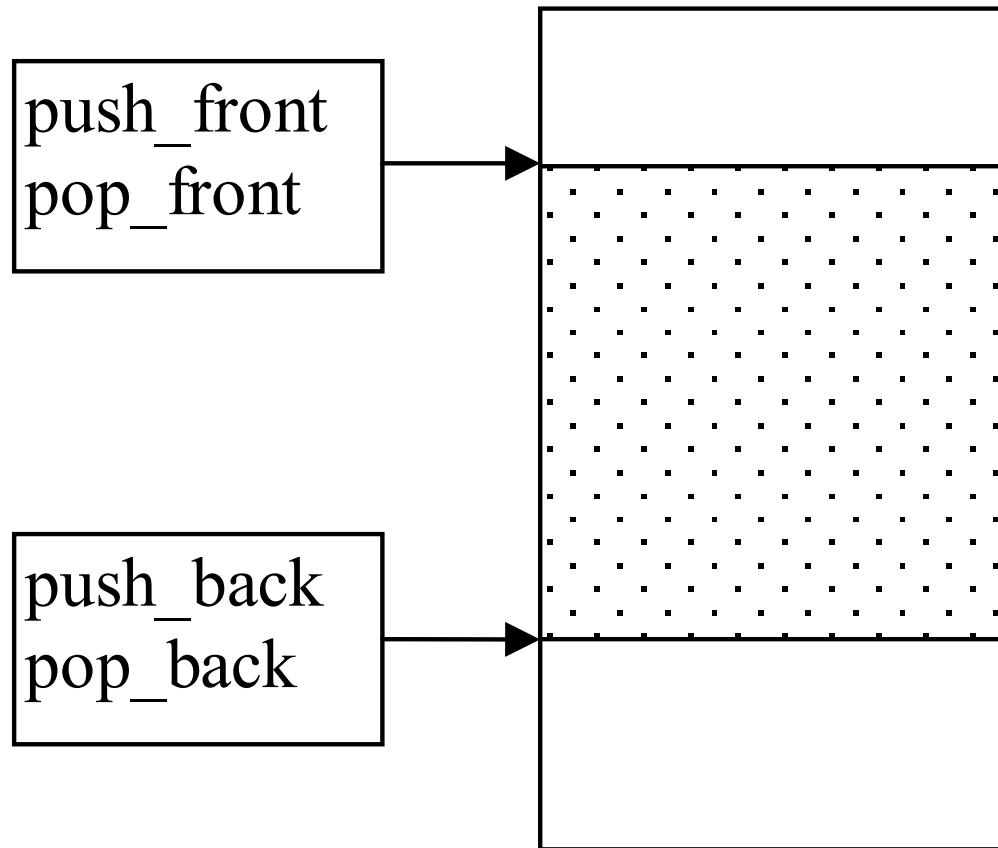
# Специфические для list операции

- void **unique**( );
- template<class BinaryPredicate> void **unique**(  
BinaryPredicate \_Pred );

# STL

## deque

# Назначение deque:



# Требования к deque:

- Обеспечивать **произвольный** доступ к элементам (как vector => at(), operator[]...)
- Эффективная работа с **началом последовательности** (как у list => push\_front(), pop\_front()...)

Header <deque>

namespace std

template<class \_Ty,

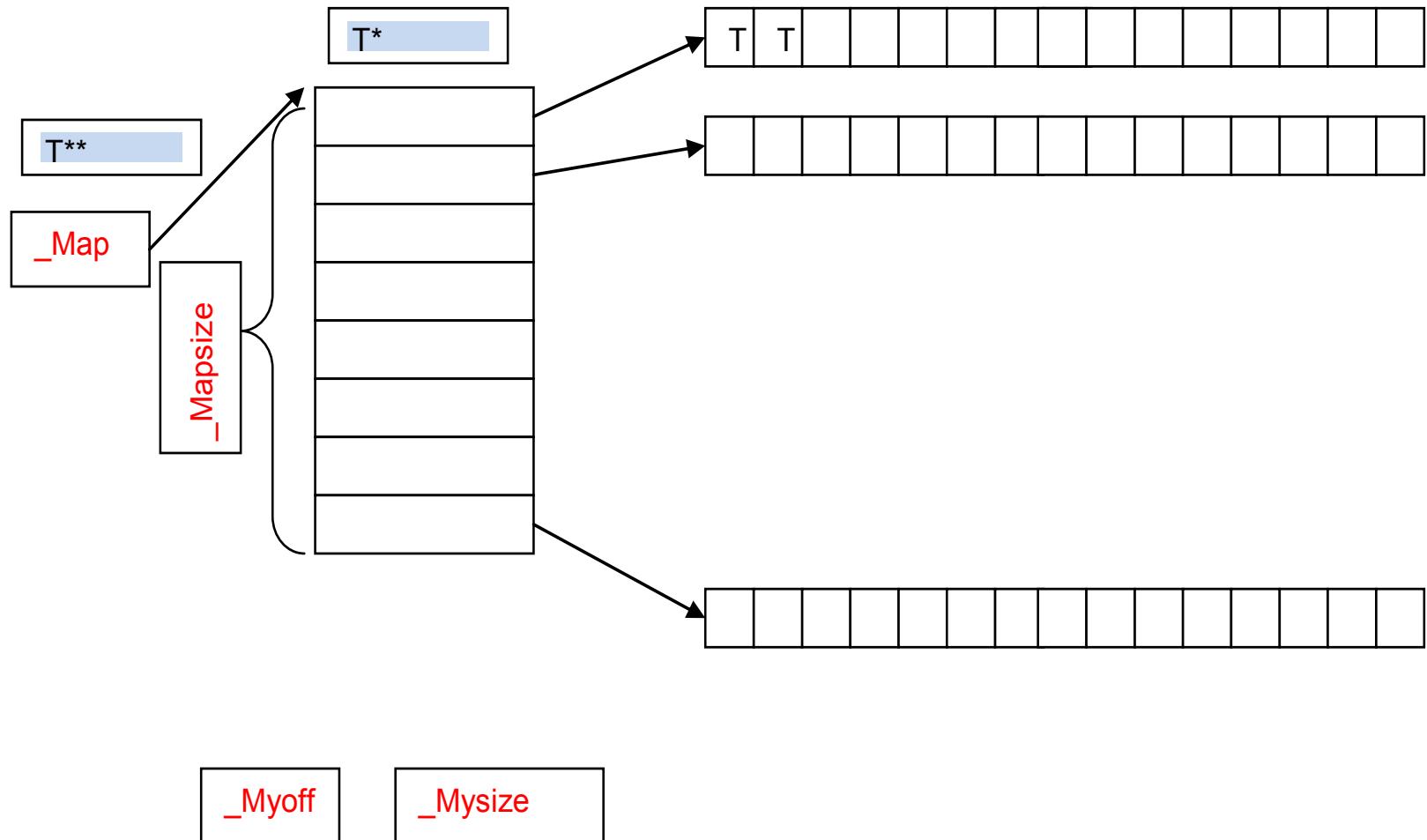
          class \_Ax = allocator<\_Ty> >

class **deque**

      : public \_Deque\_val<\_Ty, \_Ax>

{// circular queue of pointers to blocks

# Внутреннее устройство deque



# Класс deque

```
iterator begin(){ return iterator(_MyOff, this);}
```

```
iterator end()  
{ return iterator(_MyOff + _Mysize, this);}
```

```
T& operator[](int i){ return *(begin() + i);}
```

# Контейнеры

**Контейнер** – это объект, который  
хранит другие объекты и  
контролирует их размещение в  
памяти посредством  
конструкторов, деструкторов и  
методов вставки/удаления

# Коллекция

- это объединение 0 или более элементов, предоставляющее интерфейс, посредством которого можно получать доступ к элементам в модифицирующем или не модифицирующем стиле

# Состав STL

- Контейнеры
- Итераторы
- Аллокаторы
- Обобщенные алгоритмы
- Адаптеры
- Предикаты

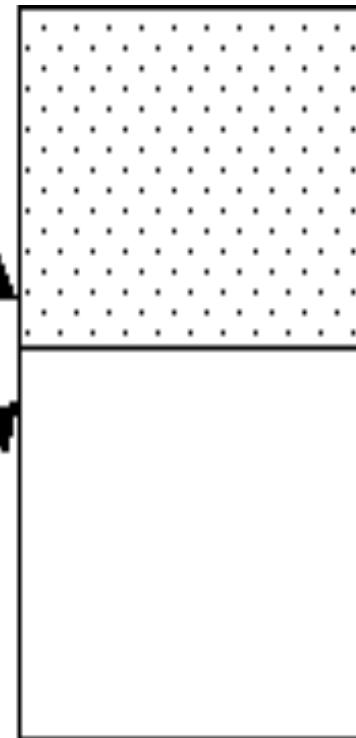
# Контейнеры

- **Последовательные**
  - Базовые (vector, list, deque)
  - Адаптеры базовых (stack, queue, priority\_queue)
- **Ассоциативные** (set, multiset, map, multimap, hash\_map, hash\_set)

# Стек. Назначение

`pop()` – достать можно только последний

`push()` – поместить можно только в конец



LIFO

# Стек – интерфейс для базового контейнера. Header <stack> namespace std

```
template<class T, class C = deque<T> >
class stack{
```

- 1. Запретить все не стековые операции
- 2. Переопределить общепринятые для стека посредством базовой последовательности

# Стек. Адаптер

pop_back() – уничтожает только последний	<b>pop()</b> – ничего не возвращает!!! =>
back() – значение последнего	для получения значения последнего элемента – <b>top()</b>
push_back() – добавляет только в конец	<b>push()</b>

# stack

```
template<class T, class C = deque<T>> class std::stack{  
protected: C c;  
public:  
    typedef typename C::value_type value_type;  
    typedef typename C::size_type size_type;  
    typedef C container_type;  
    explicit stack(const C& a=C()):c(a){}  
    bool empty() const {return c.empty();}  
    size_type size() const {return c.size();}  
  
    value_type& top(){return c.back();}  
    const value_type& top()const {return c.back();}  
  
    void push(const value_type& x) {c.push_back(x);}  
    void pop(){c.pop_back();}  
};
```

# stack

Извне **недоступны**:

- Методы базового контейнера
- Итераторы базового контейнера
- Псевдонимы базового контейнера

# Пример

```
stack<int> s; //???
s.push(1); //size==1
s.push(2); //size==2
s.pop();   // size==1
int tmp = s1.top();      //tmp==1, size==1
if(s1.top() == 1) s1.pop();
```

# Пример

1.

```
vector<int> v(10,1);
```

//создать стек таким образом, чтобы его  
элементы стали копиями элементов вектора

2. Создать пустой стек на базе list

# Пример

1. `vector<int> v(10,1);`  
`stack<int, vector<int> > s2(v); //???`
  
2. `stack<int, list<int> > s3; //???`

# Пример

```
stack<int> s; //???
```

```
s.push(1); //size==1
```

```
s.push(2); //size==2
```

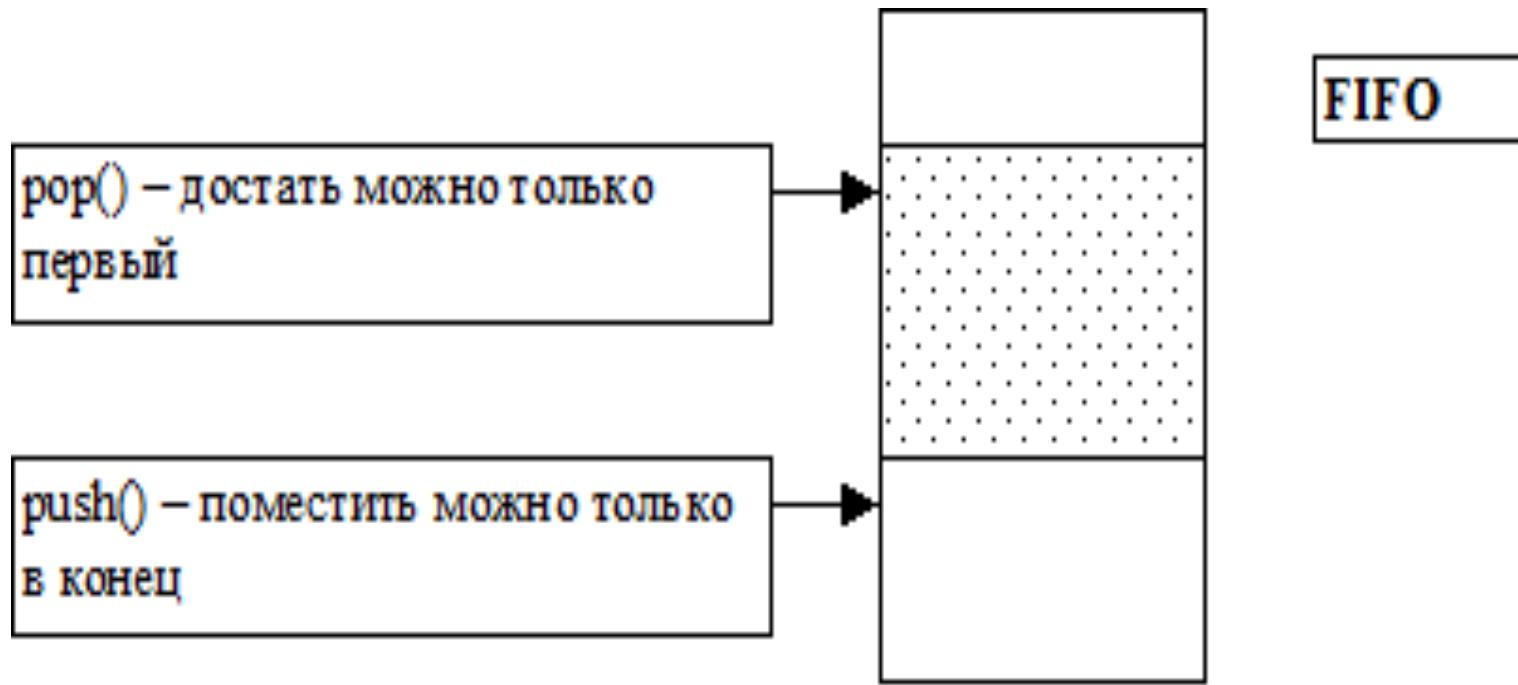
...

//Распечатать значения элементов стека:

# Пример

```
stack<int> s; //???
s.push(1);    //size==1
s.push(2);    //size==2
...
while(s.size()) //или while(!s.empty())
{
    std::cout<<s.top();
    s.pop();
}
std::cout<<s.size(); //???
```

# queue



# queue

```
template<class T, class C = deque<T>> class std::queue{  
protected:  
    C c;  
public:  
    explicit queue (const C& a=C()):c(a){};  
    value_type& front(){return c.front();}  
    value_type& back(){return c.back();}  
    bool empty() const {return c.empty();}  
    size_type size() const {return c.size();}  
  
    void push(const value_type& x) {c.push_back(x);}  
    void pop(){c.pop_front();}  
};
```

# Пример

```
struct Message{...};  
  
void Message_Loop(queue<Message>& q)  
{  
    while(q.empty()==false)  
    {  
        Message& msg = q.front();  
        msg.service();  
        q.pop();  
    }  
}
```

# `priority_queue` – очереди с приоритетами

```
template<typename T,  
        typename C = vector<T>,  
        typename Cmp = less<T> >  
  
class std::priority_queue{  
protected:  
    C c;  
    Cmp cmp; //объект – для формирования критерия  
              вставки  
    ...  
};
```

# Специфика

- Вставка и удаление элементов реализованы посредством `std::make_heap()`,  
`std::push_heap()` и `std::pop_heap()`
- Внедренный объект компаратора используется:  
`if(cmp(<элемент очереди>, <вставляемое значение>))...`
- Для критерия сравнения по умолчанию используется шаблон структуры `std::less`

# less

```
template<typename T> struct less{  
    bool operator()(const T& x, const T& y)  
    { return x < y; }  
};
```

# Heap - сортировка

- реализуется посредством последовательной коллекции (массива)
- Представляет собой линеаризованное бинарное дерево

# Heap - сортировка

Специфика:

- первый элемент всегда является максимальным => идеальная основа для реализации **очереди с приоритетами**;
- добавление и удаление элементов в общем случае производится с логарифмической сложностью

# Алгоритмы STL для работы с кучами

- **make\_heap()** преобразует интервал элементов в кучу;
- **push\_heap()** добавляет новый элемент в кучу;
- **pop\_heap()** удаляет элемент из кучи;
- **sort\_heap()** преобразует кучу в упорядоченную коллекцию (которая после этого **перестает** быть кучей)

# **make\_heap()**

Преобразует последовательность [beg, end) в кучу (сложность линейная - не более  $3*n$  сравнений):

```
void make_heap (RandomAccessIterator beg,  
                RandomAccessIterator end);
```

```
void make_heap (RandomAccessIterator beg,  
                RandomAccessIterator end, BinaryPredicate op);
```

# **make\_heap()**

```
#include <algorithm>
```

```
{
```

```
    int ar[] = {3,5,-1,7,2,5,10,1}; //исходный массив
```

```
    size_t n = sizeof(ar)/sizeof(int);
```

```
    std::make_heap(ar, ar+n);
```

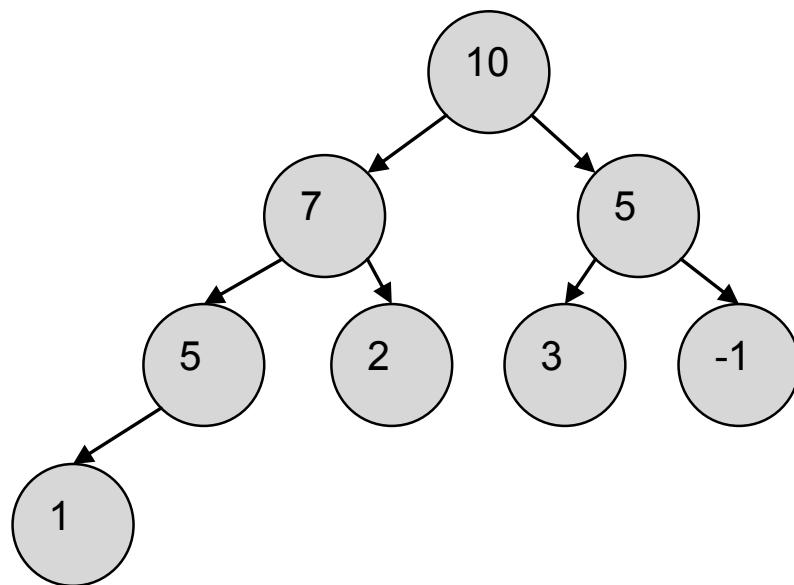
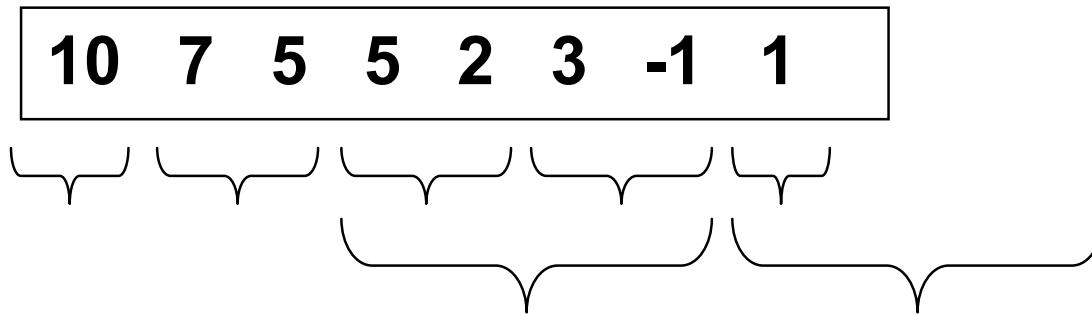
//10 7 5 5 2 3 -1 1 - после сортировки

```
}
```

# `make_heap()`

Значение каждого узла  
бинарного дерева **меньше**  
**или равно** значению  
родительского узла

# **make\_heap()**



# **push\_heap**

Добавляет последний элемент (находящийся перед end) в существующую кучу (в результате [begin, end) становится кучей. Сложность логарифмическая (не более **log(numberOfElements)** сравнений)

- void **push\_heap** (RandomAccessIterator **beg**, RandomAccessIterator **end**)
- void **push\_heap** (RandomAccessIterator **beg**, RandomAccessIterator **end**, BinaryPredicate **op**)

# **push\_heap**

Алгоритм **push\_heap()** переставляет элементы таким образом, что инвариант структуры бинарного дерева (любой узел не больше своего родительского узла) остается неизменным!

# push\_heap

```
int ar[] = {3, 5, -1, 7, 2, 5, 10, 1, 99};
```

```
int n = sizeof(ar)/sizeof(int);
```

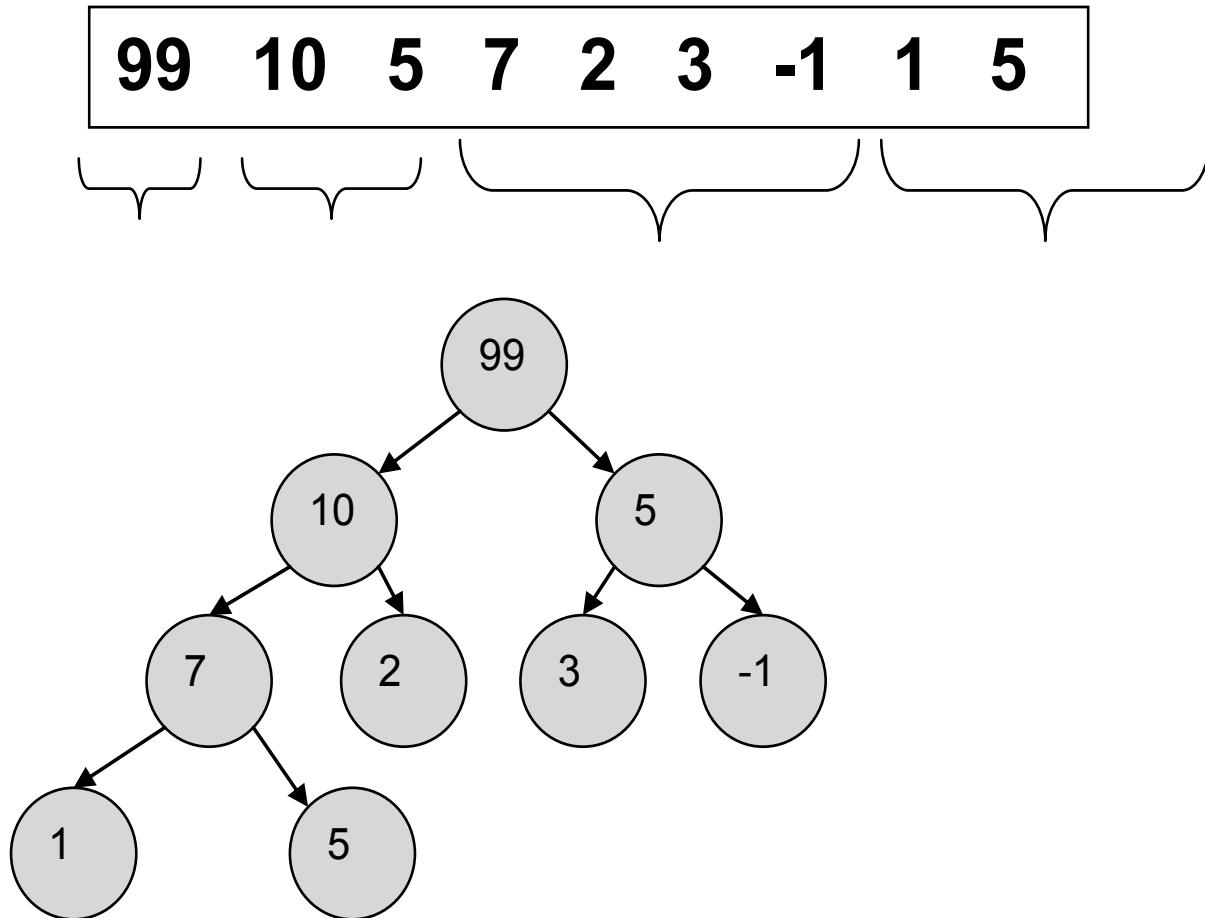
```
std::make_heap(ar,ar+n-1);
```

//**10 7 5 5 2 3 -1 1 99** - после сортировки

```
std::push_heap(ar, ar+n);
```

//**99 10 5 7 2 3 -1 1 5** - после push

# push\_heap



# **pop\_heap**

Перемещает максимальный (то есть первый) элемент кучи [beg,end) в конец и создают новую кучу из оставшихся элементов в интервале [beg,end-1). Сложность логарифмическая (не более  $2 * \log(\text{numberOfElements})$  сравнений)

- **void pop\_heap (RandomAccessIterator beg,  
                  RandomAccessIterator end)**
- **void pop\_heap (RandomAccessIterator beg,  
                  RandomAccessIterator end, BinaryPredicate Op)**

# **pop\_heap**

Алгоритм **pop\_heap()** переставляет элементы таким образом, что инвариант структуры бинарного дерева (любой узел не больше своего родительского узла) остается неизменным!

# **pop\_heap**

```
int ar[] = {3,5,-1,7,2, 5,10,1};
```

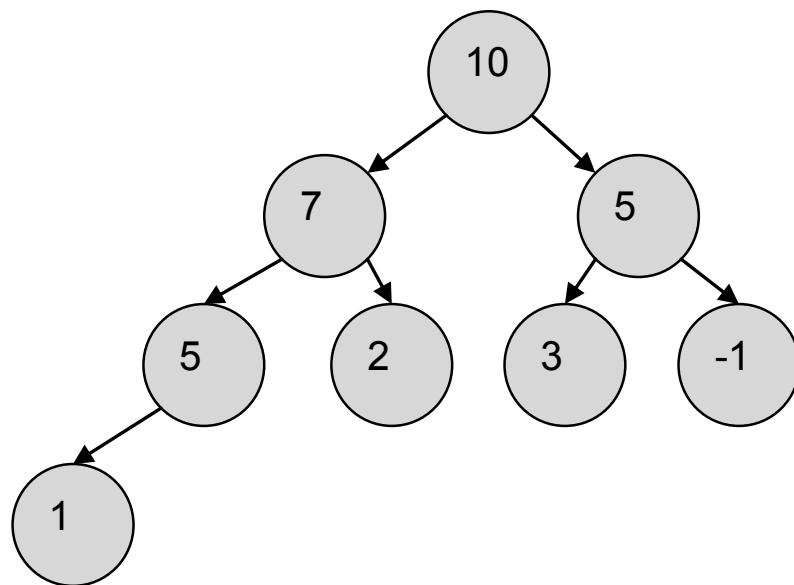
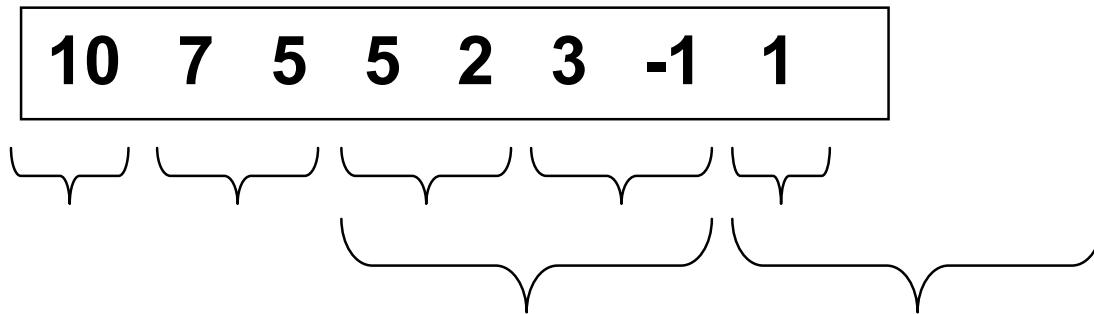
```
size_t n = sizeof(ar)/sizeof(int);
```

```
make_heap(ar,ar+n);
```

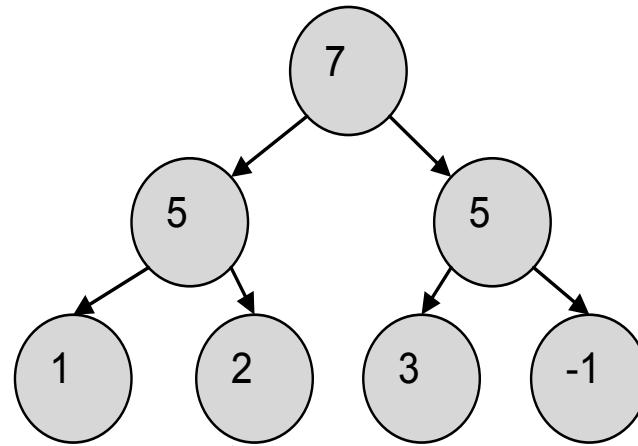
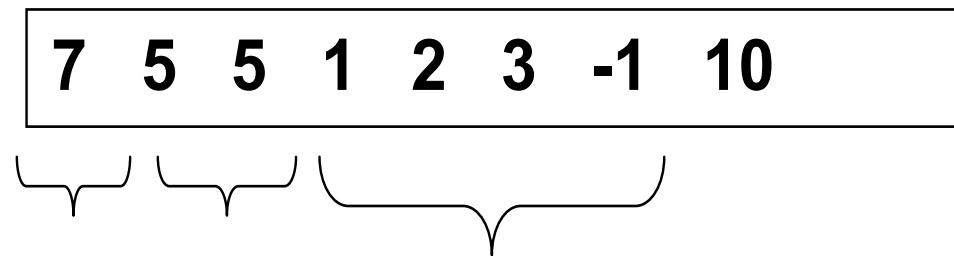
```
pop_heap(ar,ar+n); //7 5 5 1 2 3 -1 10
```

```
pop_heap(ar,ar+n-1); //5 5 3 1 2 -1 7 10
```

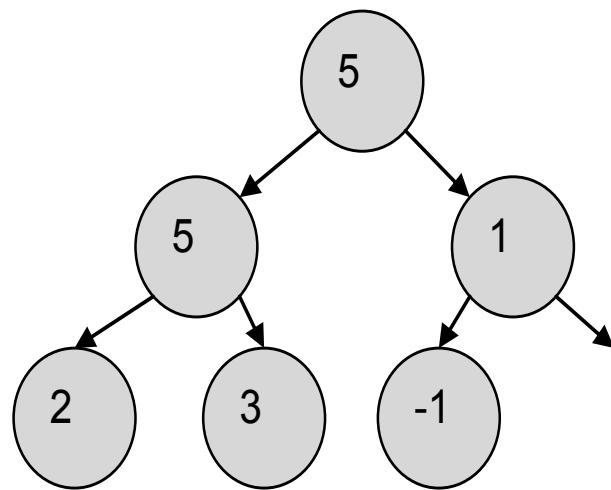
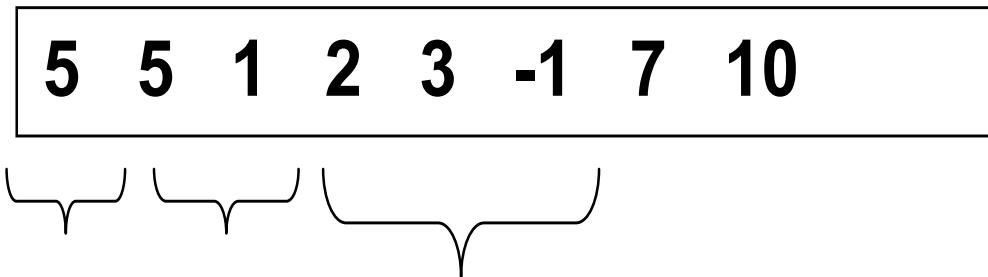
# make\_heap()



# pop\_heap



# pop\_heap



# **sort\_heap**

преобразует кучу [beg,end) в упорядоченный интервал. После вызова интервал **перестает быть кучей**. Сложность логарифмическая (не более `numberOfElements*log(numberOfElements)` сравнений)

- `void sort_heap (RandomAccessIterator beg,  
                  RandomAccessIterator end)`
- `void sort_heap (RandomAccessIterator beg,  
                  RandomAccessIterator end, BinaryPredicate Op)`

# sort\_heap

```
int ar[] = {3,5,-1,7,2, 5,10,1};
```

```
int n = sizeof(ar)/sizeof(int);
```

```
make_heap(ar,ar+n);
```

```
...
```

```
sort_heap(ar,ar+n); // -1,1,2,3,5,5,7,10
```

# priority\_queue – push()

```
void push(const T& t)  
{  
    c.push_back(t);  
    push_heap(c.begin(), c.end(), comp);  
}
```

# priority\_queue – pop()

```
void pop()
{
    pop_heap(c.begin(), c.end(), comp);
    c.pop_back();
}
```

# Пример (использование умолчаний)

```
priority_queue<char> q; //???
q.push('B'); //B
q.push('F'); //FB
q.push('A'); //FBA
q.push('W'); //WFBA
```

//Распечатать значения элементов

...

# Пример (использование умолчаний)

```
priority_queue<char> q;  
q.push('B'); //B  
q.push('F'); //FB  
q.push('A'); //FBA  
q.push('W'); //WFBA  
  
//Распечатать значения элементов  
while(!q.empty())  
{  
    cout<<q.top()<<" ";  
    q.pop();  
}
```

# Пример (явное задание)

```
priority_queue<char,deque<char>, greater<char>> q;  
q.push('B');//B  
q.push('F');//BF  
q.push('A');//ABF  
q.push('W');//ABFW  
while(!q.empty())  
{  
    cout<<q.top()<<" ";  
    q.pop();  
}  
}
```

# Пример (пользовательский критерий)

```
class Nocase{
public:
    bool operator()  (char left, char right)
    { ... }
};

{
    priority_queue<char,vector<char>, Nocase > q;
    q.push('b');//b
    q.push('F');//bF
    q.push('a');//abF
    q.push('W');//abFW
    ...
}
```

# set (множество)

```
namespace std  
#include <set>  
  
template<class _Key,  
         class _Pr = less<_Key>,  
         class _Alloc = allocator<_Key> >  
  
class set : public _Tree<_Tset_traits<_Kty, _Pr, _Alloc, false>>  
{ // ordered red-black tree of key values, unique keys
```

# Специфика set

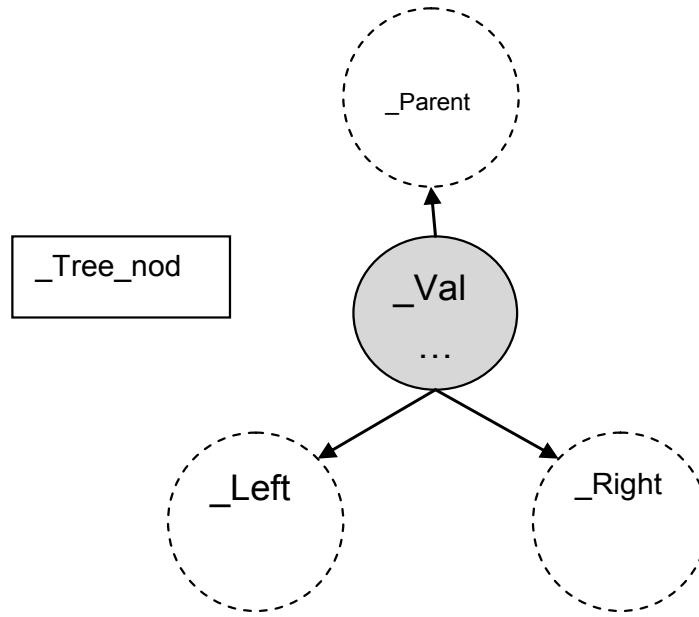
1. Данные всегда хранятся в **упорядоченном** виде! =>  
**поиск** происходит очень **быстро!**

# Специфика set

2. Самый простой из ассоциативных контейнеров, так как **ключ совпадает со значением**

# Специфика set

3. Базовым классом является двоичное сбалансированное **дерево**. Дерево состоит из узлов:



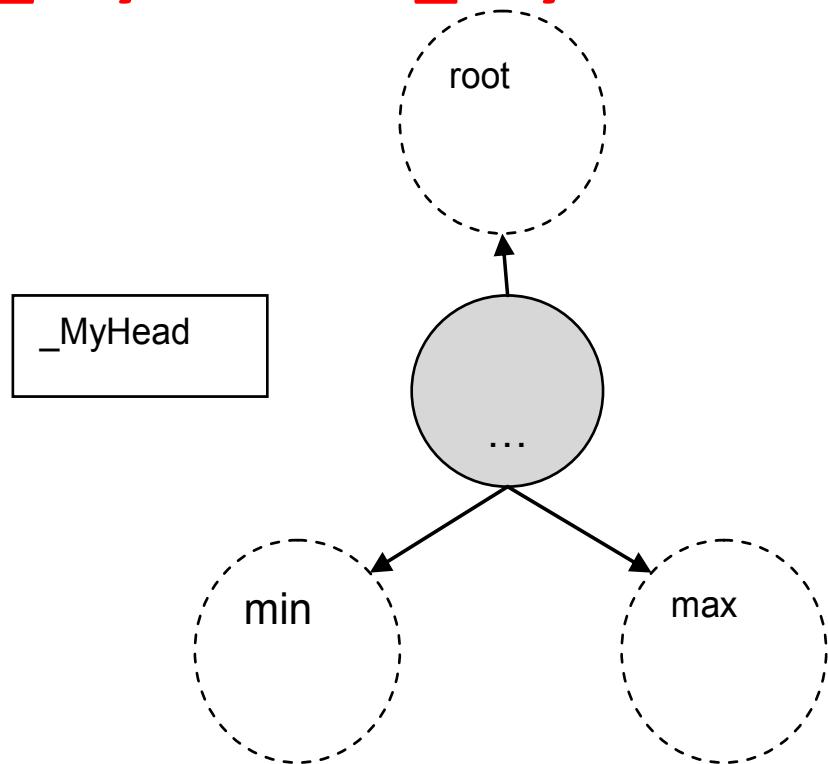
Замечание: тип конкретного используемого дерева зависит от реализации

# Специфика set

4. Значения уникальны => **дубли**  
**игнорируются!**

# Специфика set

5. Данные: фиктивный элемент-страж -  
**\_MyHead + \_Mysize**



# Специфика set

6. Нет операций `push_back()`, так как заполняется в соответствии с критерием упорядоченности => вставка происходит только методом **insert()**

# Пример

```
set <int> s; //???
s.insert(10);
s.insert(2);
s.insert(30);
s.insert(20);
s.insert(1);
s.insert(11);
s.insert(10); //???
//Распечатать значения элементов
...
```

# Продолжение примера

```
for(set<int>::iterator it=s.begin();  
    it!=s.end(); ++it)  
{  
    cout<<*it<<" ";  
}
```

# Специфика set

7. Сложный итератор, который ++ (--) умеет перемещаться на узел с большим значением, выбирая наименьшее из поддеревьев

# Специфика set

8. Итератор предназначен только для чтения.  
Почему???

```
set <int> s;
```

```
s.insert(10);
```

```
...
```

```
set<int>::iterator it=s.begin();
```

```
*it = 33; //???
```

# Специфика set

## 9. Реверсивный итератор

Распечатать set в обратном порядке

# Специфика set

10. Если дерево «вырождается» в **не** сбалансированное, оно **перестраивается!** (то есть изменяется root)

```
set<int> s;  
s.insert(10);  
s.insert(9);  
s.insert(8); //дерево «перестраивается»  
s.insert(7);
```

# Пример

1.

```
int ar1[5] = {1,2,3,4,5};  
set<int> s1(ar1, ar1+5);  
//печатать элементов  
int ar2[5] = {5,4,3,2,1};  
set<int> s2(ar2, ar2+5);  
//печатать элементов
```

# Пример

2.

```
vector<int> v;  
//формирование значений v  
set<int> s1(v.begin(), v.end());  
//печать элементов s1  
set<int> s2(v.rbegin(), v.rend());  
//печать элементов s2
```

# Пример

3.

```
int ar1[10] = {1,2,3,4,5,1,3,5};  
set<int> s(ar1, ar1+10); //???  
//печатать элементов
```

# Пример

## 4. Явное задание критерия упорядочения

```
int ar1[10] = {1,2,3,4,5};  
set<int, greater<int>> s(ar1, ar1+10);  
//печать элементов  
set<int, greater<int>> ::iterator it =  
    s.begin();
```

...

# Пример

5. Пользовательский критерий упорядочения

```
struct Abs{  
    bool operator()(int x, int y){return abs(x)<abs(y);}  
};
```

```
int ar1[10] = {5,-1,-3,2,10};  
set<int, Absset<int, Abs...  
...
```

# Пример

## 6. Упорядоченное чтение строк из файла

```
ifstream f("my.txt");
string word;
set<string> words;
while(f>>word, !f.eof())
{
    words.insert(word);
}
f.close();
```

# Специфические методы set

```
iterator lower_bound(const Key& _Key);  
//возвращает итератор на элемент, значение  
которого >=_Key
```

```
iterator upper_bound(const Key& _Key);  
//возвращает итератор на элемент, значение  
которого >_Key
```

```
pair <iterator, iterator> equal_range (const Key&  
_Key); //возвращает пару итераторов
```

# Пример

```
set<int> s;  
//10, 20, 25, 30  
typedef set<int>::iterator IT;  
IT it1 = s.lower_bound(15); //???  
IT it2 = s.lower_bound(30); //???  
IT it3 = s.upper_bound(15); //???  
IT it4 = s. upper_bound(30); //???  
//Распечатать от it1 до it2  
//Стереть от it3 до it4
```

# multiset (множество с повторениями)

namespace std

#include <set>

```
template<class _Key,  
        class _Pr = less<_Key>,  
        class _Alloc = allocator<_Key> >  
  
class multiset  
: public _Tree<_Tset_traits<_Key, _Pr, _Alloc,  
true>>  
{ // ordered red-black tree of key values, non-unique keys
```

# Специфика multiset

Допускает **дублирование** значений

```
multiset <int> s;
```

```
s.insert( 10 );
```

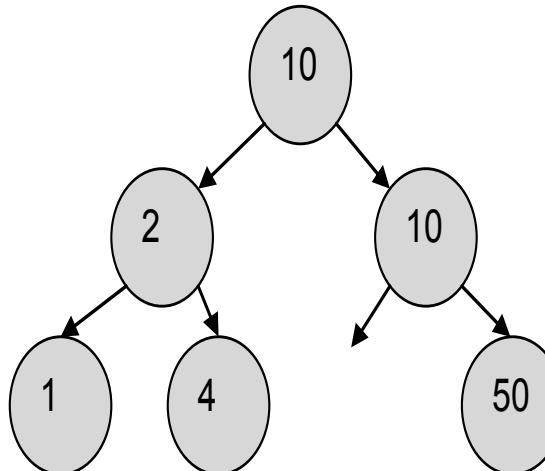
```
s.insert( 2 );
```

```
s.insert( 10 );
```

```
s.insert( 4 );
```

```
s.insert( 1 );
```

```
s.insert( 50 );
```



# Специфика multiset

- Значения могут повторяться
- Метод `find` возвращает итератор на первый элемент с «искомым» значением

# map (словарь)

```
namespace std  
#include <map>  
  
template<class _Key,  
         class _Val,  
         class _Pr = less<_Key>,  
         class _Alloc = allocator<pair<const _Key, _Val>>>  
class map  
: public _Tree<_Tmap_traits<_Key, _Val, _Pr, _Alloc,  
false>>  
{ // ordered red-black tree of {key, mapped} values, unique keys
```

# Специфика тар

1. Настоящий ассоциативный контейнер  
(значение не совпадает с ключом)

# Специфика тар

2. Ключи уникальны => для пары с существующим ключом модифицируется значение

# Специфика тар

3. тар отсортирован по ключам => критерий  
сравнения задается для ключей

# Специфика map

4. Для хранения пары ключ/значение используется шаблон структуры pair:

```
template<typename T1, typename T2> struct pair{
    T1 first;
    T2 second;
    pair(const T1& t1, const T2& t2):first(t1), second(t2){}
};
```

```
#include <utility>
int main()
{
    pair<char,double> p('A',1.1); //ключ/значение
    p.make_pair('B', 2.2); //изменяю оба значения
    p.first = 'C'; //изменяю ключ
    p.second = 3.3;// изменяю значение
}
```

# Специфика map

5. Псевдоним `value_type` сопоставляется  
чему???

При разыменовании итератора получаем  
что???

Можно ли использовать итератор для  
записи???

# Специфика map

## 6. operator[]

# Специфика map

## 7. метод insert()

- принимает пару ключ/значение
- Возвращает пару итератор/bool (true, если пара занесена)

# Специфика map

8. Методы `find()` и `erase()` осуществляют поиск по ключу

# Пример

```
map<string, int> book;  
book[string("Иванов")] = 1111111;  
pair< map<string, int>::iterator, bool> r=  
    book.insert(pair<string, int>("Петров",  
                                2222222));  
  
//Печать записной книжки  
???
```

# Пример

```
map<string, int> book;
book[string("Иванов")] = 1111111;
book[string("Петров")] = 2222222;
...
map<string, int>::iterator it = book.begin();
while(it != book.end())
{
    cout<<(*it).first<<":"<<(*it).second<<endl;
}
```

???

```
map<string, int>::iterator it=
    book.find(string("Петров"));
//проверка – если существует
```

...

```
*it.second = 3333333; //???
*it.first = string("Сидоров"); //???
```

# multimap (словарь с повторениями)

```
namespace std
#include <map>

template<class _Key,
class _Val,
class _Pr = less<_Key>,
class _Alloc = allocator<pair<const _Key, _Val>>>
class multimap
: public _Tree<_Tmap_traits<_Key, _Val, _Pr, _Alloc,
true>>
{ // ordered red-black tree of {key, mapped} values,
non-unique keys}
```

# Специфика multimap

- позволяет дублировать значения ключей, элементы с одинаковыми ключами в контейнере хранятся в порядке занесения
- => не определен operator[]
- добавление элементов с помощью только insert()
- при удалении по ключу ( erase() ) удаляются все элементы, удовлетворяющие условию
- count() – возвращает количество пар, в которых ключ совпадает с указанным значением

# Итераторы

Обеспечивают:

- Получение значений элементов
- Перемещение по последовательности

# Категории итераторов

Тип	Дополнительные операции	Какие контейнеры предоставляют
входной (input)	$x = *i; ++i; i++;$	???
выходной(output)	$*i = x; ++i; i++;$	???
прямой или однонаправленный (farward)	$x = *i; *i = x; ++i; i++;$	???
дву направленный (bidirectional)	как у farward и $-i; i--;$	???
Произвольного доступа (random access)	как у bidirectional и $i+n; i-n; i+=n; i-=n; i < j; i > j;$ $i \leq j; i \geq j;$  При этом: если для вектора желательно сохранить “общность”, то лучше ???, повысить эффективность – ???	???

# Потоковые итераторы (Stream Iterators )

- #include <iterator>
- Цель - представить входные и выходные потоки как **последовательности** – для того, чтобы можно было использовать потоки ввода/вывода в обобщенных алгоритмах, также как списки, вектора...

# **ostream\_iterator**

Специфика:

- `operator++()` – реализован как заглушка
- `operator=` выводит в поток
- конструктор с одним параметром –  
принимает в качестве параметра указанный  
объект потока вывода
- конструктор с двумя параметрами  
принимает в качестве второго параметра –  
разделитель (строка)

# Пример

```
ostream_iterator<int> os(cout);
*os = 1;
//++os;
*os = 2;
```

# **istream\_iterator**

Специфика:

- operator++() осуществляет ввод (НЕ заглушка!)
- программист должен обеспечить место
- default конструктор формирует признак конца ввода
- конструктор с одним параметром принимает ссылку на объект потокового ввода
- Разделители (пробел, tab, enter) отфильтровываются

# Защита от некорректного ввода

```
cin.exceptions(ios::failbit);
istream_iterator<int> is(cin);
int ar[10];
try{
    copy(is, istream_iterator<int>(), ar);
}
catch(ios_base::failure& f)
{
    cout<<f.what();
    while(cin.rdbuf()->in_avail()) { cin.get(); }
    cin.clear();
}
cin.exceptions(ios::goodbit);
```

# Непосредственная работа с буферами ввода/вывода

- `ostreambuf_iterator`
- `istreambuf_iterator`

# Итераторы - адаптеры

- Реверсивные итераторы
- Итераторы вставки (в случае необходимости умеют **расширять** контейнер)

# Итераторы вставки

- `back_insert_iterator` (использует `push_back`)
- `front_insert_iterator` (использует `push_front`)
- `insert_iterator` (использует `insert`)

# Итераторы вставки

Для формирования итераторов вставки STL предоставляет шаблоны глобальных функций:

- **`back_inserter(cont& c);`**
- **`front_inserter(cont& c);`**
- **`inserter(cont& c, iterator it);`**

# Итераторы вставки

Пример:

```
vector<int> v; //пустой!  
back_insert_iterator<vector<int>> it =  
    back_inserter(v);  
*it = 1; //???
```

# Обобщенные алгоритмы

Способы задания предикатов

# for\_each()

Инкапсулирует цикл:

- перебирает элементы последовательности
- и вызывает заданный предикат, отправляя в качестве параметра очередной элемент последовательности

# Использование явного цикла

```
int main()
{
    map<const char*, int> months;
    months[“Январь”] = 31;
    ...
    //Вывести соответствие ????
}
```

# Использование явного цикла

```
int main()
{
    map<const char*, int> months;
    months[“Январь”] = 31;
    ...
    map<const char*, int> ::iterator it =
        months.begin();
    for(;it != months.end(); ++it)
    {
        cout<<(*it).first <<”: ”<<(*it).second<<endl;
    }
}
```

# Ключевое слово **auto** C++11

- тип переменной, объявленной как **auto**, определяется компилятором самостоятельно на основе того, чем эта переменная инициализируется
- **auto**-переменная не может хранить значения разных типов! => после инициализации сменить тип переменной будет уже нельзя (C++ как был, так и остается статически типизированным языком)

# Пример использования auto

Как делать НЕ стоит!

**auto n = 1;**

# Использование явного цикла C++11

```
int main()
{
    map<const char*, int> months;
    months[“Январь”] = 31;
    ...
auto it = months.begin();
for(;it != months.end(); ++it)
{
    cout<<(*it).first <<”: ”<<(*it).second<<endl;
}
}
```

# for\_each()

```
template<class IT, class PRED>
    void for_each(IT first, IT last, PRED F)
{
    while(first != last)
    {
        F(*first);
        ++first;
    }
}
```

# Предикат – глобальная функция

- без модификации последовательности

```
int main()
{
    vector<int> v;
    //заполнение вектора
    //Вывести куб каждого элемента последовательности ???
}
```

# Предикат – глобальная функция

```
void PrintCube(int n) { cout << n * n * n << ' ' ; }
```

```
int main()
{
    vector<int> v;
    //заполнение вектора
    for_each(v.begin(), v.end(), PrintCube) ;
    cout << "\n\n" ; }
```

# Предикат – глобальная функция

- модификация последовательности

```
int main()
{
    list<Point> l;
    //заполнение списка
    //Поменять знак координат ???
}
```

# Предикат – глобальная функция

```
void Neg(Point& pt)
{
    pt=-pt;
}

int main()
{
    list<Point> l;
    //заполнение списка
    for_each(l1.begin(), l1.end(), Neg);
}
```

# Предикат – шаблон глобальной функции

???

# Предикат – шаблон глобальной функции

```
template<typename T> inline void Neg(T& t)
{
    t=-t;
}

int main()
{
    list<Point> l;
    //заполнение списка
    for_each(l.begin(), l.end(), Neg<Point>);
    int ar[] = {5,-3,-10,33};
    for_each(ar, ar+sizeof(ar)/sizeof(int), Neg<int>);

}
```

# `find()`, `find_if()`

Найти элемент последовательности, значение которого равно значению параметра

???

# find()

```
int main()
{
    vector<Point> v;
    ...
    vector<Point>::iterator it = find(v.begin(),
                                         v.end(), Point(1,1));
    //что должно быть определено в классе Point ???
    if(???) cout<<"Not found";
}
```

# Эмуляция алгоритма `find_if()`

```
template<typename IT, typename PRED> IT  
    find_if(IT first, IT last, PRED f)  
{  
    while(first != last)  
    {  
        if(f(*first)==true){return first;}  
        ++first;  
    }  
    return last;  
}
```

# Пример использования алгоритма find\_if()

Найти **все** точки (Point), которые находятся в окружности с радиусом ==10 от начала координат

# Пример использования алгоритма find\_if()

```
inline bool FindPoint(const Point& pt){  
    int tmp = pt.m_x*pt.m_x + pt.m_y*pt.m_y;  
    return tmp *tmp < 10*10;  
}  
int main()  
{  
    vector<Point> v;  
    //...  
    vector< vector<Point>::iterator> vIT;  
    vector<Point>::iterator it = v.begin();  
    while(it != v.end())  
    {  
        it= find_if(it, v.end(), FindPoint) ;  
        if(it != v.end() ) vIT.push(it);  
    }  
    ...  
}
```

# count(), count\_if()

```
template<class In, class T>
```

```
    size_t count(In first, In last, const T& val);
```

# Эмуляция count\_if()

???

# Эмуляция count\_if()

```
template<typename IT, typename PRED> int
    count_if(IT first,IT last,PRED f)
{
    int count=0;
    while(first != last)
    {
        if(f(*first)==true){count++;}
        ++first;
    }
    return count;
}
```

# Пример `count_if()`. Предикат – функциональный объект

Преимущество функционального объекта –  
можно запаковать любое количество  
информации

Задание: посчитать точки, которые отстоят  
меньше, чем на заданное значение от начала  
координат

???

# Предикат – функциональный объект

```
class COUNT_IF{
    int m_d;
public:
    COUNT_IF(int d){m_d = d;}
    bool operator()(const Point& pt){
        return ...<m_d*m_d);}
};
```

# Предикат – функциональный объект

```
vector<Point> v;  
//...  
size_t n = count_if(v.begin(), v.end(),  
                    COUNT_IF(2));
```

# Алгоритмы копирования

copy()

copy\_if()

# copy\_if()

```
template<typename FROM,typename TO,  
         typename PRED>  
void copy_if(FROM first, FROM last, TO res, PRED f)  
{  
    while(first != last)  
    {  
        if(f(*first)==true){*res = *first; ++res;}  
        ++first;  
    }  
}
```

# Пример использования copy\_if()

- скопировать все элементы, кроме совпадающих с указанным значением

# Пример использования copy\_if().

Предикат – шаблон класса

```
template<typename T> class COPY_IF{
```

```
    T m_t;
```

```
public:
```

```
    COPY_IF(const T& t){m_t = t;}
```

```
    bool operator()(const T& t)
```

```
        {return t!=m_t;}
```

```
};
```

# Пример использования copy\_if().

Предикат – шаблон класса

```
vector<int> vi;
```

```
int ar[] = {1,5,-6,1};
```

```
// скопировать все значения, кроме «1»
```

# Пример использования copy\_if().

## Предикат – шаблон класса

```
vector<int> vi;  
int ar[] = {1,5,-6,1};  
copy_if(ar, ar+sizeof(ar)/sizeof(int),  
       ???,  
       COPY_IF<int>(1));
```

```
vector<Point> v;  
//...  
vector<Point> v1;//пустой вектор!!!  
copy_if(v.begin(), v.end(),  
       ???,  
       COPY_IF<Point>(Point(1,1)));
```

# Пример использования copy\_if().

## Предикат – шаблон класса

```
vector<Point> v;
```

```
//сформировать значения
```

```
//вывести на печать, кроме Point(1,1)
```

# Пример использования copy\_if().

Предикат – шаблон класса

```
vector<Point> v;
```

```
//сформировать значения
```

```
//вывести на печать, кроме Point(1,1)
```

```
copy_if(v.begin(), v.end(),
```

```
        ostream_iterator<Point>(cout, " "),
```

```
        COPY_IF<Point>(Point(1,1)));
```

# Пример использования copy\_if().

Предикат – шаблон класса

```
vector<int> v;  
//сформировать значения  
//считать из файла:  
vector<int> v;  
ifstream file("test.txt");  
copy_if(istream_iterator<int> (file),  
        istream_iterator<int>(),  
        inserter(v,v.begin()), COPY_IF<int>(33));
```

# sort()

- Два перегруженных варианта
- Без предиката:
  - сортирует последовательность в порядке возрастания значений элементов
  - чтобы алгоритм работал с пользовательскими типами, в классе должен быть перегружен operator<
- Замечание: для списка сортировка реализована методом класса.

# Пример sort()

```
int ar[5] = {-5,8,1,-10,5};  
sort(ar, ar+5);
```

```
vector<int> v(ar, ar+5);  
sort(v.begin(),v.end()); //в возрастающем порядке  
sort(v.rbegin(),v.rend()); //в убывающем порядке
```

# Пример

```
vector<Point> v;
```

```
//...
```

```
//Отсортировать по удалению от x,y
```

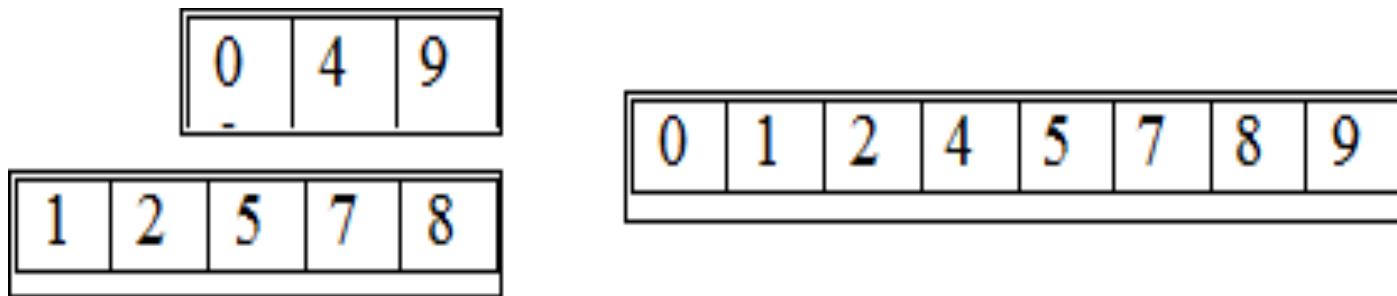
# Предикат

```
class CMP_IF{//по удалению от x,y
    int x,y;
public:
    CMP_IF(int a, int b){x=a; y=b;}
    bool operator()(const Point& pt1, const Point& pt2)
    {
        return ...
    }
};

sort(v.begin(),v.end(),CMP_IF(2,2));
```

# merge()

объединение двух **отсортированных!**  
последовательностей



# merge()

```
vector<int> v;  
//...  
sort(...);
```

```
int ar[5] = {1,2,5,7,8};  
sort(...);
```

//объединить в список ???

# merge()

```
vector<int> v;  
//...  
sort(...);
```

```
int ar[5] = {1,2,5,7,8};  
sort(...);
```

- a) 

```
list<int> l(v.size() + sizeof(ar)/sizeof(int)); // места достаточно  
merge(v.begin(), v.end(), ar, ar+5, l.begin()); // режим замены
```
- б) 

```
list<int> l; // пустой список  
merge(v.begin(), v.end(), ar, ar+5, inserter(l,l.begin())); // режим  
вставки
```

# transform()

```
template<class FROM, class TO, class UnaryF>
TO transform(FROM First1, I FROM Last1, TO
Result, UnaryF f );
```

```
template<class FROM1, class FROM2, class TO,
class BinaryF>
TO transform(FROM1 First1, FROM1 Last1,
FROM2 First2, TO Result, BinaryF f );
```

# Трансформация одной последовательности в другую

```
int a[] = {2,-5,...};
```

```
int b[<столько же>];
```

```
//Требуется: b[i] = -a[i];
```

```
transform(a, a+sizeof(a)/sizeof(int), b,  
         negate<int>() );
```

# Трансформация двух последовательностей в третью

```
vector<int> v;  
//сформировали значения  
  
list<int> l;  
//сформировали значения  
  
deque<int> d;  
//требуется: d[i] = v[i] + l[i];  
//сколько элементов можно использовать?  
transform(..., plus<int>());
```

# Задание предиката

```
vector<double> v;
```

```
//сформировали значения
```

```
list<double> l;
```

```
//требуется: l[i] = A*v[i]*v[i] + B*v[i] +C;
```

# Предикат – метод класса

`mem_fun` – для последовательностей, в которых хранятся указатели

`mem_fun_ref` – для последовательностей, в которых хранятся копии объектов

# mem\_fun и mem\_fun\_ref

mem\_fun\_ref – это шаблон функции, которая:

- Принимает в качестве параметра указатель на метод класса
- А возвращает объект типа mem\_fun\_ref\_t, в котором:
  - Сохраняется указатель на метод класса
  - Перегружен operator()

# Приимер mem\_fun и mem\_fun\_ref

```
class Point{
    int m_x, m_y;
public:
    Point (int x=0, int y=0){m_x = x; m_y = y;}
    bool MEM_F(){return (m_x<0 || m_y<0);}
};
```

```
Point* arptr[] = {new Point(), new Point(1,2), new
Point(3,4)};
int n=count_if(arptr, arptr+sizeof(arptr)/sizeof(Point*),
               mem_fun(&Point::MEM_F));
Point arpt [] = { Point(), Point(1,2), Point(3,4)};
n = count_if(arpt, arpt+sizeof(arpt)/sizeof(Point),
               mem_fun_ref(&Point::MEM_F));
```

# Лямбда-выражения

C++11

# Лямбда-выражениями (функциями)

- называются безымянные локальные функции, которые можно создавать прямо внутри какого-либо выражения
- в C++ — это краткая форма записи **анонимных предикатов**

# Специфика лямбда-выражений

λ-выражение:

- всегда начинается с [] (скобки могут быть непустыми)
- затем идет необязательный список параметров
- параметры можно передавать разными способами (по ссылке, по значению)
- затем непосредственно «тело функции»
- по умолчанию λ-функция возвращает **void** (можно указать возвращаемый тип явно)

# Использование λ -выражений вместо функциональных объектов

```
class PrintCube{  
public: void operator ()(int x) const { cout<< (x*x*x)<<' '; }  
};  
  
int main()  
{  
    int ar[] = {5,-1,4,-7,3};  
    for_each(ar, ar + sizeof(ar)/sizeof(int), PrintCube() );  
    cout<<endl;  
//Использование лямбда-выражений  
    for_each(ar, ar + sizeof(ar)/sizeof(int),  
            [] (int x){cout<< (x*x*x)<<' ';} );  
    cout<<endl;  
}
```

# Встречая лямбда-выражение, компилятор:

- генерирует анонимный класс (или структуру?),
- в котором перегружен operator()
- важно! метод operator() – константный!
- в нашем примере метод:
  - ничего не возвращает (void),
  - принимает по значению очередной элемент последовательности

# Тип возвращаемого λ-выражением значения

- Может формироваться компилятором неявно
- Программист может указать явно

Тип возвращаемого значения  
формируется компилятором **неявно**

// скопировать только отрицательные значения

```
int ar[] = {5,-1,4,-7,3};
```

```
vector<int> v;
```

```
copy_if(ar, ar + sizeof(ar)/sizeof(int), ???,
```

**[](int x){return x<0;}**

```
);
```

# Тип возвращаемого значения указывается явно

```
int ar[] = {5,-1,4,-7,3};  
  
vector<int> v;  
  
copy_if(ar, ar + sizeof(ar)/sizeof(int),  
        back_inserter (v),  
        [](int _x)-> bool{return _x<0;}  
);
```

# Замечания:

- для неявного формирования типа возвращаемого значения в лямбда-функции должна быть только **одна** инструкция return
- если

```
copy_if(ar, ar + sizeof(ar)/sizeof(int),  
       back_inserter(v),  
       [](int x) -> bool  
       {if(x==0) return true; else return x<0;}  
);
```

То тип возвращаемого значения может быть задан только явно!

# Формирование значений переменных анонимного функционального объекта

- Требуется скопировать только те значения,  
которые попадают в указанный диапазон

# Функциональный объект

```
class Range{  
    int lower, upper;  
public:  
    Range(int l, int u):lower(l), upper(u){}  
    bool operator ()(int x) const  
        { return (x > lower) && (x < upper); }  
};
```

# Лямбда-функция

```
int ar[] = {5,-1,4,-7,3};  
int lower=0, upper=10;  
vector<int> v;  
copy_if(ar, ar + sizeof(ar)/sizeof(int),  
       back_inserter (v),  
       [lower, upper](int _x)-> bool  
       {return _x>lower && _x<upper;})  
);
```

# Тип передаваемого параметра

```
int ar[] = {5,-1,4,-7,3};  
for_each(ar, ar + sizeof(ar)/sizeof(int),  
        [](int& x){x++;})  
    ;
```

```
for_each(ar, ar + sizeof(ar)/sizeof(int),  
        [](int x){x++;})  
    ; //???
```

# Модификация переменных анонимного функционального объекта внутри λ-функции

- Метод `operator()`, генерируемый компилятором, является **константным!**
- Как позволить модифицировать переменные класса в константном методе???

# Пример

```
//При каждом копировании увеличиваем верхнюю границу
диапазона
int ar[] = {5,-1,4,-7,3,11};
int lower=0, upper=10;
vector<int> v;
copy_if(ar, ar + sizeof(ar)/sizeof(int), back_inserter (v),
        [lower, upper](int _x) -> bool
{
    if(_x>lower && _x<upper){upper++; //ошибка!
                                return true;}
    else      return false;
}
);
```

# mutable

```
int ar[] = {5,-1,4,-7,3,11};  
int lower=0, upper=10;  
vector<int> v;  
copy_if(ar, ar + sizeof(ar)/sizeof(int), back_inserter (v),  
        [lower, upper](int x)mutable -> bool  
    {  
        if(x>lower && x<upper){upper++; return true;}  
        else return false;  
    }  
); //??? значение upper??? (что будет изменяться?)
```

Формирование в анонимном объекте  
адресов внешних переменных

Посчитать количество скопированных  
элементов

# Формирование в анонимном объекте адресов внешних переменных

```
int ar[] = {5,-1,4,-7,3,11};  
int lower=0, upper=10, count = 0;  
vector<int> v;  
copy_if(ar, ar + sizeof(ar)/sizeof(int), back_inserter (v),  
[lower, upper, count](int _x)mutable -> bool  
{  
    if(_x>lower && _x<upper){count++; return true;}  
    else    return false;  
}  
); //??? значение count???
```

# Формирование в анонимном объекте адресов внешних переменных

```
int ar[] = {5,-1,4,-7,3};  
int lower=0, upper=10, count = 0;  
vector<int> v;  
copy_if(ar, ar + sizeof(ar)/sizeof(int), back_inserter (v),  
        [lower, upper,&count](int x) -> bool  
        { if(x>lower && x<upper){count++;  
          return true;}  
        else return false;  
    }  
); //???
```

# Формирование в анонимном объекте адресов внешних переменных

[x, y]	// захват x и y по значению
[&x, &y]	// захват x и y по ссылке
[x, &y]	// захват x по значению, а y — по ссылке

# Спецификация исключений в λ-функции

- Не генерирует исключений:

```
[] (int x) throw() { ... }
```

- Генерирует bad\_alloc:

```
[] (int x) throw(std::bad_alloc) { ... }
```

# Технология ООП

Санкт-Петербургский государственный политехнический университет

06 сентября 2011

# Стандартные типы данных в C++

- `char` — символ;
- `int` — целое число;
- `float` — число с плавающей точкой;
- `double` — число с плавающей точкой двойной точности;
- `enum` — перечислимый тип задает набор именованных констант целого типа.

Уточняющие описатели типов: short, long и unsigned делают возможными следующие описания типов:

- unsigned char
- short int
- long int
- unsigned short int
- unsigned int
- unsigned long int

# Область действия имен

---

```
int i=16; // Глобальная переменная
void proc()
{
    int i=1;
    {
        int i=2;
        printf ("\n global i=%d , local i=%d" , :: i , i );
    }
}
```

---

3 типа операций:

- унарные
- бинарные
- тернарные

# Унарные

- ~ !	Операции отрицания и дополнения
* &	Разыменование и взятие адреса
sizeof	Определение размера переменной или типа
+	Унарный плюс
++ --	Приращение (increment) и убывание (decrement)

# Бинарные

* / %	Мультипликативные операции
+ -	Аддитивные операции
<< >>	Операции сдвига
< > <= >= == !=	Операции бинарных отношений
&   ^	Побитовые логические операции
&&	Логические операции
,	Операции последовательного вычисления

---

```
float x = -0.1, y = -0.05, z = 0.5;
printf("%ox", x<y<z);
```

---

# Операции присвоения

Все эти операции изменяют значение операнда:

---

`++, --, +=, =, *=, /=, %=, <<=, >>=, &=, |=, ^=`

---

# Операции присвоения

Чему будут равны значения переменных i,j в конце фрагмента?

---

```
int i=0, j=0;  
j = i++; //постинкремент  
i = ++j; //преинкремент
```

---

Для логических операций в языке С используются следующие символы:

- || — логическое ИЛИ (OR);
- && — логическое И (AND);
- ! — логическое НЕ (NOT);

---

```
if (min < n && n < max) // Если n лежит внутри
    диапазона
DoSomething();
```

---

Операции << и >> выполняют сдвиг влево и вправо на указанное во втором операнде число разрядов.

---

```
int i=64;  
i >>= 4;
```

---

- Сдвиг влево переменной устанавливает правые освобождающиеся биты в ноль.
- Сдвиг вправо устанавливает левые биты либо в ноль, либо в единицу в зависимости от типа (беззнаковый — в ноль, знаковый — копией знакового разряда).

# Побитовые операции

Операции `&`, `|`, `^` являются поразрядными, побитовыми (bitwise) и носят названия: И, ИЛИ, исключающее ИЛИ (XOR).

# Побитовые операции

Операция `&` часто используется для снятия определенных битов в переменной, а операция `|` — для установки.

---

```
var |= (1 << i); //установить i-й бит
var &= ~(1 << i); //очистить i-й бит
```

---

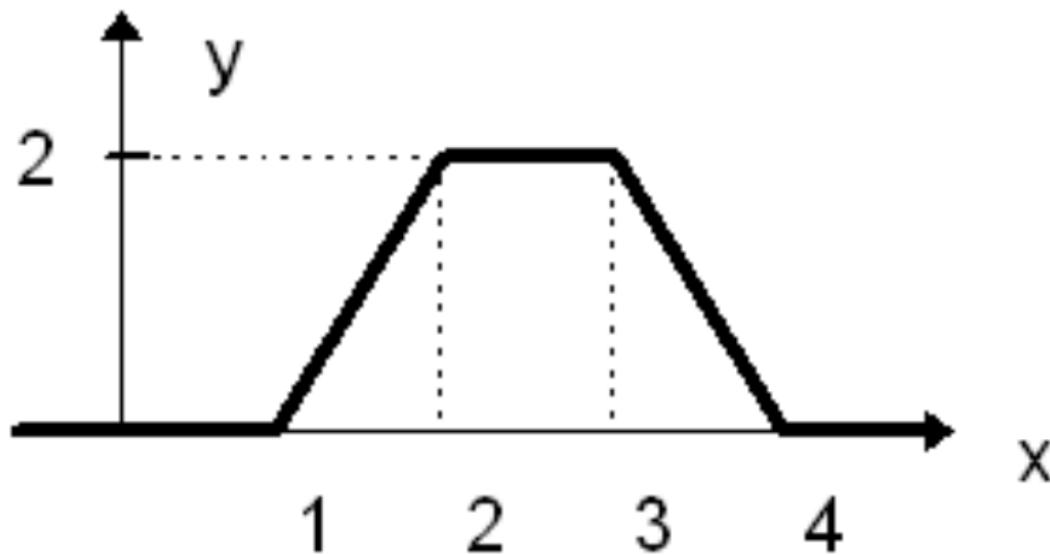
В языках С и С++ существует две разновидности условных операторов или операторов разветвления: if-statement, и switch-statement

---

```
if (условие)
{
    операторы
}
else
{
    операторы
}
```

---

Пусть функция задана в виде графика.



# Условные операторы

Чтобы задать ее с помощью цепочки вложенных условных операторов, определим переменные double x, y.

---

```
if (x>1)
    if (x>2)
        if (x>3)
            if (x>4)
                y=0;
            else
                y=-2*x+8;
            else
                y=2;
        else
            y=2*x-2;
    else
        y=0;
```

---

# Оператор switch

Алгоритм функционирования оператора можно описать в виде схемы:

---

```
switch ( выражение )
{
    case константа1: операторы ;
    case константа2: операторы ;
    .
    .
    .
    default: операторы ;
}
```

---

# Оператор switch

После выполнения одной из ветвей case управление передается в следующую ветвь, поэтому для выхода из switch необходимо использовать один из следующих операторов:

- break;
- goto <метка>;
- continue;
- return;

# Тернарные операции

Тернарная операция имеет следующий вид:

---

(условие) ? expr1 : expr2;

---

# Тернарные операции

---

```
i = x < 0 ? -1 : 1;  
// Равносильно if (x<0) i=-1; else i=1;
```

---

Специфика тернарной операции заключается в том, что она возвращает значение, т.е. она может быть rvalue (right-hand-side value), что позволяет создавать такие конструкции:

---

```
i = x < 0 ? -1 : x > 0 ? 1 : 0;
```

---

# Операторы циклов

Три оператора цикла языков С и С++ могут быть описаны следующей схемой:

---

```
while (условие) {тело цикла}
do {тело цикла} while (условие);
for ( ; ; ) {тело цикла}
```

---

# Операторы циклов

Сколько раз выполнится каждое тело цикла?

---

```
while(0) {тело цикла}  
while (1) {тело цикла}  
do {тело цикла} while(0);
```

---

# Операторы циклов

---

```
int i=1, j=1024;  
while (i < j) i=2*i;
```

---

```
int i=1, j=1024;  
while (i < j) j=2*i;
```

Последовательность выполнения операторов при реализации цикла:

- ① Операторы первой части заголовка (выполняются один раз до входа в цикл).
- ② Вторая часть (проверка условия и выход из цикла в случае его нарушения).
- ③ Операторы тела цикла.
- ④ Операторы третьей части заголовка.
- ⑤ Переход к пункту 2.

# Цикл for

---

```
char sText[1024];
for (int i=0; i<1024; i++)
sText[i] = ' '; // Заполнение строки текста пробелами
```

---

# Задача

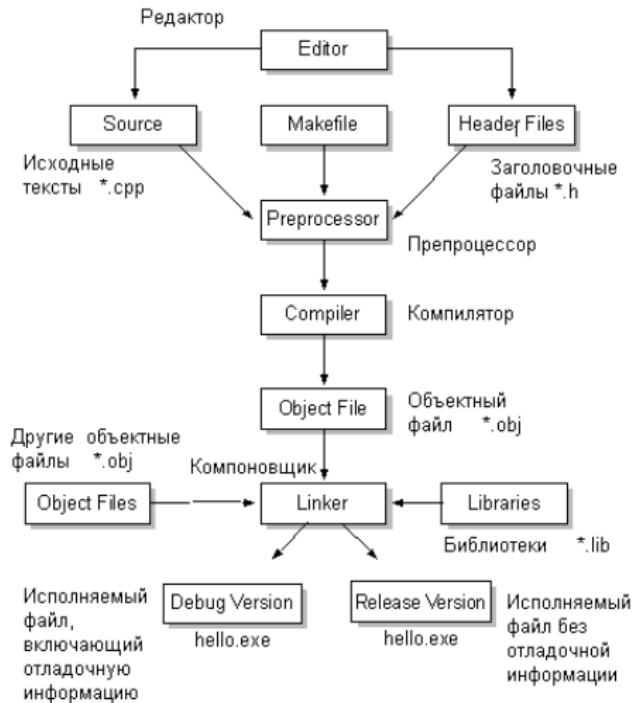
Исправить 1 символ чтобы программа напечатала 20 \*. Именно исправить, а не удалить или добавить, пробел тоже символ.  
Найти 3 решения.

---

```
int main () { int i , N=20; for ( i=0 ; i<N ; i-- )  
{ printf ("*"); } }
```

---

# Последовательность этапов генерации исполняемого кода



Имя массива в языке С фактически является указателем на первый его элемент, который соответствует нулевому значению индекса. Если объявлен массив float a[16];, то справедливо равенство  $a == \&a[0]$ .

---

```
float a[16],*p; // Объявление массива и указателя
for (int i=0; i<16; i++)
    a[i] = float(i*i); // Заполнение массива
p = &a[6]; // p указывает на a[6]
*p = 3.14f; // Равносильно a[6] = 3.14;
p++; // Теперь p указывает на a[7]. Произошел сдвиг
      на 4 байта
a[1] = *(p+3); // Равносильно a[1] = a[10]; В a[1]
                  попадает 100.
a[2] = ++*p; // Равносильно a[2] = ++a[7]; В a[2]
                  попадает 50.
a[3] = *++p; // Равносильно a[3] = a[8]; В a[3]
                  попадает 64.
```

---

Выделим динамически память под матрицу размерностью NxN.

---

```
// Динамическое размещение массива указателей на
// массивы типа double (строки матрицы)
double **a = new double*[n];
// Динамическое размещение массивов строк матрицы (
// переменные типа double)
for (int i=0; i<n; i++)
a[ i ] = new double[ n];
```

---

# Динамическое выделение памяти

```
// Освобождение памяти, занятой строками матрицы
for (i=0; i<n; i++)
    delete [] a[i];
// Освобождение памяти, занятой массивом указателей
// на строки матриц
delete [] a;
```

# Технология ООП

Санкт-Петербургский государственный политехнический университет

13 сентября 2011

# Еще немного об указателях

---

```
double *p1, *p2;  
*p1=12.56e7;  
*p2=8.85e12;
```

---

# Еще немного об указателях

---

```
double *p1,*p2;  
p1 = new double;  
p2 = new double;  
*p1=12.56e7;  
*p2=8.85e12;
```

---

# Константные указатели и указатели на константу

---

```
const char *title = "Diagram"; // Указатель на
    константу
char *const title="Diagram"; // Константный
    указатель
```

---

# Константные указатели и указатели на константу

---

```
const char *title = "Diagram"; // Указатель на
    константу
char *const title="Diagram"; // Константный
    указатель
```

---

Что будет если выполнить:

---

```
title[0] = 'd';
title = new_str;
```

---

В первом и во втором случае?

# Константные параметры

---

```
char* strcpy (char* dest, const char* source);
```

---

# Способы передачи параметров

В языках С и С++, в отличие от многих других языков, передача параметров осуществляется по значению — это значит что в функции происходит работа с копией переменной.

---

```
void func(int x);  
func(value);
```

```
void func(float *x);  
func(array);
```

```
void func(float x[]);  
func(array);
```

```
void func(int &out_x);  
func(value);
```

---

# Возвращаемые значения

---

```
double array[100];  
  
double& getV(int i) { return array[i]; }  
  
getV(1) = 5.0;  
printf("%f\n", getV(1));
```

---

# Указатели на функции

Указатель на любую функцию, которая возвращает вещественное число и требует одно вещественное число в качестве параметра:

---

```
double (*pFunc)(double);
```

---

# Указатели на функции

---

```
#include <math.h>

double (*pFunc)(double);
if(input)
pFunc = sin;
else
pFunc = cos;

double val = pFunc(1.0);
```

---

Языки С и С++ позволяют определить новый тип переменных, присвоив символьное имя какому-то заданному типу.

---

```
typedef int points;
typedef struct Node* NodePtr;
typedef char Msg[100];
```

---

# Структуры

---

```
struct Man
{
    char *Name; // Имя
    int Age; // Возраст
};
```

---

---

```
struct Man m;  
m.Name = new char[100];
```

---

# Массив структур

//TODO: Написать сортировку

---

```
Man ar [3] = {  
    {"Man1", 60},  
    {"Man2", 50},  
    {"Man3", 80}  
};
```

---

Union — это структура, все элементы которой имеют нулевое смещение.

---

```
struct Test
{
    int selector;
    union
    {
        long long_value;
        short short_value;
    }
};
```

---

# Битовые поля

Структура позволяет работать с битовыми полями.

---

<целый тип> [идентификатор] : <размер поля>;

```
struct byte
{
    char a: 1;
    char b: 1;
    char c: 1;
    char d: 1;
    char e: 1;
    char f: 1;
    char g: 1;
    char h: 1;
};
```

---

# Битовые поля

---

```
union chbyte
{
    byte bit;
    char x;
} b;

b.x = 'a';
printf("%lc\n", b.bit.h ? '1' : '0');
```

---

# Перечислимые типы

---

```
typedef enum
{
    One, Two, Three
} Digits;

enum
{
    One = 1, Two, Three
} digit;

digit = Two;
Digits digit2 = digit;
```

---

# Рекурсивные функции

Рекурсивная функция — функция вызывающая саму себя.

Функция вычисляющая число Фибоначчи:

---

```
unsigned long fib (int n)
{
    unsigned long f=0;
    if (n>0)
        if (n==1)
            f = 1;
        else
            f = fib(n-1) + fib(n-2);
    return f;
}
```

---

# Операции с файлами

---

```
char *fn = "test.txt"; // Имя файла
FILE *fp; // Указатель на файловую структуру
if(fopen (fn , "rt") == NULL) printf("File not found
\n");
```

---

# Операции с файлами

- r — Открыть существующий файл только для чтения.
- r+ — Открыть существующий файл для обновления.
- w — Создать файл для записи.
- w+ — Создать новый файл для обновления.
- a — Открыть для записи в конец файла.
- a+ — Открыть для чтения и записи в конец файла.
- b — Открыть файл в двоичном режиме.
- t — Открыть файл в текстовом режиме.

# Операции с файлами

---

```
char str[100];  
  
while (!feof(fp))  
{  
    fgets(str, 100, fp);  
}  
fclose(str);
```

---

# Параметры командной строки

- argc — количество параметров в командной строке программы.
- argv — массив из строк содержащий переданные параметры.

---

```
void main (int argc , char *argv [])
{
char src_filename[20] , dst_filename[20];
if(argc < 3)
printf("error\n");
strncpy(src_filename , argv[1] , 20);
strncpy(dst_filename , argv[2] , 20);
//TODO: написать копирование файлов
}
```

---

Базовые понятия которые имеют существенную роль в ООП:

- инкапсуляция
- наследование
- полиморфизм

# Технология ООП

Санкт-Петербургский государственный политехнический университет

20 сентября 2011

# Спецификаторы для данных и методов

---

```
struct Man
{
    char* Name;
    int Age;
    int age() { return Age; }
};
```

```
Man man;
man.age();
man.Age;
```

---

# Спецификаторы для данных и методов

- private
- public
- protected

---

```
class Man
{
char* Name;
int Age;

public:
int age() { return Age; }
};

Man man;
man.age();
man.Age; //не будет работать
```

---

# Конструкторы и деструкторы

Конструктор — служит для начальной инициализации объекта при его создании.

Деструктор — вызывается при прекращении жизни объекта, используется для освобождения динамически выделенной памятию.

# Конструктор

Носит такое же имя как и класс, не имеет типа возвращаемого значения.

---

```
class Man
{
    char Name[100];
    int Age;

public:
    Man(char *name) { strcpy(Name, name, 100); }
    Man() { strcpy(Name, "test", 100); } //конструктор
           по умолчанию
    Man(int age = 100) { Age = age; }
};
```

---

# Конструктор

---

```
Man::Man(int value1, int value2)
{
    var1 = value1;
    var2 = 5;
}

Man::Man(int var1, int var2) : var1(value1), var2
    (5)
{}
```

---

# Конструктор копирования

Срабатывает при инициализации объекта класса другим объектом.

---

```
class Man
{
    char *Name;

public:
    Man(const &name) { strncpy(Name, name.Name, 100) }
};
```

---

Срабатывает при окончании времени жизни объекта. Не имеет параметров.

---

```
class Man
{
char *Name;

public:
~Man() { delete Name; }
};
```

---

Можно вызвать деструктор явно obj-> Man();

# Указатель `this`

Указатель на текущий объект, используется неявно для обращения к элементам объекта.

---

```
class Man
{
    int Age;

public:
    int age() { return Age; }
    Man *obj() { return this; }
    // int age() { return this->Age; }
};
```

---

# Дружественные функции

Дружественные функции позволяют получить доступ к `private` членам класса.

---

```
class Man
{
int Age;
friend void setAge(Man *, int);

public:
void setAge(int age) { Age = age; }
};

void setAge(Man *man, int age)
{
man->Age = age;
}
```

---

# Дружественные функции

---

```
class Man;
class User
{
public:
void age(Man &, int);
};
class Man
{
int Age;
friend void User::age(Man &, int);
};
void User::age(Man &man, int age)
{
man.Age = age;
}
```

---

# Дружественный класс

Дружественный класс позволяет сделать все функции класса дружественными.

---

```
class Man;
class User
{
public:
void age(Man &, int);
};
class Man
{
int Age;
friend class User;
};
void User::age(Man &man, int age)
{
man.Age = age;
}
```

# Статические члены класса

Члены класса для которых память выделяется один раз при создании класса.

---

```
class Man
{
private:
    static int count;
public:
    static int getCount() { return count; }
};

int x = Man::getCount();
```

---

# Спецификатор const

Используется для предотвращения изменения данных.

---

```
class Man
{
private:
int Age;
public:
void age(int _age) const //нельзя изменять члены
    класса
{
Age = _age; //ошибка
}

void change(const Man& man)
{
man.Age = 5; //ошибка
}
};
```

# Перегрузка операций

В C++ можно перегрузить любые операции, кроме: '.', '.\*', '?:', '::', '#', '###', 'sizeof'.

- Для стандартных типов переопределить операции нельзя.
- Функции-операции не могут иметь аргументов по умолчанию, наследуются за исключением операции '='.
- При перегрузке сохраняется число аргументов, а также приоритет операции.

---

тип **operator** операция (список параметров) {тело функции}

---

# Перегрузка унарных операций

---

```
class Man
{
int Age;
Man & operator ++() { ++Age; return *this; }
};

Man man;
++man;
```

---

# Перегрузка бинарных операций

---

```
class Man
{
int Age;
public:
int getAge() { return Age; }
};

bool operator >(const Man &m1, const Man &m2)
{
if (m1.getAge() > m2.getAge())
return true;
return false;
}
```

---

# Перегрузка бинарных операций

---

```
class Man
{
int Age;

Man operator +(const Man &m2)
{
    Man temp;
    temp.Age = Age + m2.Age;
    return temp;
}
};

Man man1, man2;
man1 = man1 + man2;
```

---

# Операция присваивания

---

```
class Man
{
int Age;
char *Name;

const Man & operator = (const Man &m1) { ... }
};
```

---

Перегрузить для класса массив операцию [].

---

//TODO

---

---

```
class имя : [private | protected | public]  
    базовый_класс
```

---

Простое наследование — наследование от одного родителя.

# Простое наследование

---

```
class Man
{
char *Name;
int Age;
};

class Stud : public Man
{
int Course;
};

class Prof : public Man
{
char * Dept;
};
```

---

# Простое наследование

---

```
class Man
{
char *Name;
int Age;
};

class Stud : public Man
{
int Course;
};

class Prof : public Man
{
char * Dept;
};
```

---

# Простое наследование

---

```
class Man
{
    char *Name;
    int Age;
public:
    int getAge() {return Age;}
};

class Stud : public Man
{
    int Course;
public:
    int getAge() {return getAge();} //FIXME
    Stud() { printf("%d\n", getAge()); }
};

class Prof : public Man
{
```

Конструкторы вызываются от более старшего класса к младшему, деструкторы наоборот.

# Множественное наследование

---

```
class A { ... };
class B : public A { ... };
class C : public A { ... };
class D : public A, public B, public C { ... };
```

---

Объект класса D содержит унаследованные члены класса A в 3х экземплярах.

# Множественное наследование

---

```
class A { ... };
class B : virtual public A { ... };
class C : virtual public A { ... };
class D : public A, public B, public C { ... };
```

---

---

```
Man *man = new Stud;  
man->age();
```

---

Будет вызван метод класса Man. Механизм раннего связывания.

---

```
virtual int age() { ... }
```

```
Man *man, *man2;  
man = new Stud;  
man2 = new Man;
```

```
man->age();  
man2->age();
```

---

Механизм позднего связывания.

- Если в базовом классе метод виртуальный, то в производном классе такой же метод становится автоматически виртуальным.
- Виртуальные методы наследуются.
- Нельзя объявить как static.
- Можно определить виртуальный метод, как «чисто виртуальный» присвоив ему значение 0.

Деструктор базового класса как правило всегда следует делать виртуальным, чтобы избежать утечек памяти.

Абстрактный класс — класс имеющий «чисто виртуальные» функции, объект такого класса не может быть создан, а только унаследован.

# Технология ООП

Санкт-Петербургский государственный политехнический университет

27 сентября 2011

# Шаблоны классов

---

```
class Queue {
public:
    Queue();
    ~Queue();

    int& remove();
    void add( const int & );
    bool is_empty();
    bool is_full();

private:
};

Queue qi;
```

---

# Шаблоны классов

---

```
template <class Type>
class Queue {
public:
    Queue();
    ~Queue();

    Type& remove();
    void add( const Type & );
    bool is_empty();
    bool is_full();

private:
};

Queue<int> qi;
```

---

# Шаблоны классов

---

```
typedef double Type;
template <class Type>
class QueueItem {
public:
private:
    Type item;
    QueueItem *next;
};
```

---

# Шаблоны классов

---

```
template<class T = char> class String;  
String ◇ *p;
```

---

# Шаблоны классов

---

```
template<class List>
class List
{
public:
void print();
};

template <class Data> void List<Data>::print();
```

---

# Шаблоны классов

---

```
template<class List>
class List
{
public:
void print();
};

template <class Data> void List<Data>::print();
```

---

# Шаблоны классов

---

```
template< template<class U> class V> class C {  
    V<int> y;  
    V<int*> y1;  
};
```

---

# Шаблоны классов

---

```
#include <iostream>
template <int N>
struct Factorial{
    enum { val = Factorial<N-1>::val * N };
};

template<>struct Factorial<0>{
    enum { val = 1 };
};

int main(){
    std::cout << Factorial<4>::val << "\n"; } //на этапе
                                                 //компиляции
```

---

# Шаблоны классов

---

```
template<class T> void f() { T::x * p; ... }
template<class T> void f() { typename T::x * p; ...
}
```

---

# Не стоит злоупотреблять шаблонами

```
.. / global.h:145: error: in passing argument 3 of '  
    int vstd :: findPos(const std :: v  
ector<T1, std :: allocator<_CharT> >&, const T2&,  
    Func&) [with T1 = std :: pair<cons  
t CGHeroInstance*, CPath*>, T2 = const  
    CArcmedInstance*, Func = boost :: _bi :: bind_<bool, bool (*)(const std :: pair<const  
    CGHeroInstance*, CPath*>&, const CGHeroIn  
stance* const std :: pair<const CGHeroInstance*,  
    CPath*>::*, const CArcmedInstance*  
const&), boost :: _bi :: list3<boost :: arg<1> (*)(),  
    boost :: _bi :: value<const CGHerol  
nstance*std :: pair<const CGHeroInstance*, CPath  
*>::*>, boost :: arg<2> (*)()> > ]'
```

# Спецификаторы класса памяти

- auto
- register
- static
- extern

## inline-функции

```
inline double Sqr(double x) return x*x;
```

# Спецификатор volatile

```
volatile int i;
```

Операция `const_cast`.

---

```
void print(int *p) { cout << *p; }
const int *p;
print(p); //ошибка
int *j = const_cast<int*>(p);
```

---

# mutable

---

```
class mutable_test
{
    int      data;
    mutable sync  sync_obj;
public:
    void set_data (int i)
    {
        sync_obj.lock ();
        data = i;
        sync_obj.unlock ();
    }
    int get_data () const
    {
        sync_obj.lock ();
        int i = data;
        sync_obj.unlock ();
        return i;
    }
};
```

Операция `dynamic_cast`. Преобразования бывают двух типов:

- повышающее — приведение производного класса к базовому
- понижающее — из базового класса в производный
- перекрестное — приведение между производными типами

# dynamic\_cast

Повышающее преобразование.

---

```
class B{ ... };
class C : public B{ ... };
C *c = new C;
B *b = dynamic_cast<B*>(c); //Эквивалентно B *b = c;
```

---

## dynamic\_cast

Поникающее преобразование. Применяется когда компилятор не может проверить правильность приведения. Для использования данного типа преобразований необходимо включить механизм RTTI.

---

```
class B{ public: virtual void f1() {} };
class C : public B{ public: void f2() {} };
C *c = new C;
B *b = new B;
C *temp = dynamic_cast<C*>(b);
if (temp)
    temp->f2();
```

---

# dynamic\_cast

Перекрестное преобразование.

---

```
class B{ public: virtual void f1() {} };
class C : public B{ public: void f2() {} };
class D : public B{ ... };
D *d = new D;
C *temp = dynamic_cast<C*>(d);
if(temp)
temp->f2();
```

---

## dynamic\_cast

Для доступа к RTTI введена операция typeid и класс type\_info.

---

```
class B { ... };
class C : public B { ... };
B *p = new C;
if(typeid(*p) == typeid(C))
{
    dynamic_cast<C*>(p)->f2();
    cout << typeid(*p).name();
}
```

---

Выполняется на этапе компиляции над:

- целыми типами
- целыми и вещественными типами
- целыми и перечисляемыми типами
- указателями и ссылками одной иерархии

## reinterpret\_cast

Применяется для преобразования не связных между собой типов, например указателей в целые и наоборот.

---

```
char *p = reinterpret_cast<char*>(malloc(10));
```

---

# Технология ООП

Санкт-Петербургский государственный политехнический университет

4 октября 2011

# Обработка исключительных ситуаций

При обработке рассматриваются только ситуации внутреннего характера (нет памяти в области heap, не найден файл, переполнение и т.д.). Ситуация созданная нажатием Ctrl-C считается внешней.

# Обработка исключительных ситуаций

Синтаксис исключений:

---

```
try {  
    ...  
}
```

---

Обозначает контролируемый блок кода.

# Обработка исключительных ситуаций

Генерация исключений:

---

**throw [ выражение ];**

---

# Обработка исключительных ситуаций

Обработка исключений:

---

```
catch(тип имя) { /*тело обработчика*/ }
catch(тип) { /*тело обработчика*/ }
catch(...) { /*тело обработчика*/ }
```

---

Обработка исключений:

---

```
catch( int i ) { //int }
catch( const char* ) { //const char* }
catch( Overflow ) { //класс Overflow }
catch( ... ) { /*обработка всех исключений*/ }
```

---

# Обработка исключительных ситуаций

После обработки исключения управление передается первому оператору находящемуся за блоком исключений. Туда же, минуя код всех обработчиков, передается управление, если исключение в try-блоке не было сгенерировано.

При генерации исключения в C++ происходят следующие действия:

- создается копия параметра `throw` в виде объекта, который существует до тех пор, пока исключение не будет обработано;
- вызываются деструкторы объектов выходящих из области действия;
- передается объект и управление обработчику имеющему совместимый тип с этим объектом.

При генерации исключений:

- создается копия параметра `throw` в виде объекта, который существует до тех пор, пока исключение не будет обработано;
- вызываются деструкторы объектов выходящих из области действия;
- передается объект и управление обработчику имеющему совместимый тип с этим объектом.

# Обработка исключительных ситуаций

Обработчик считается найденным, если:

- тип объекта в обработчке тот же, что и указан после `throw`, т.е. `T`, `const T`, `T&` или `const T&`, где `T` — тип исключения;
- является производным от указанного в параметре `catch`, если наследования производилось с ключом `public`;
- является указателем который может быть преобразован к нужному типу, например `void*` .

# Обработка исключительных ситуаций

Если происходит вызов непредусмотренного исключения, то вызывается функция `unexpected()`, реализацию которой можно заменить при помощи `set_unexpected()`, если такой функции нету, то вызывается функция `terminate()`, реализацию которой можно заменить при помощи `set_terminate()`, если такой функции нету то происходит вызов функции `abort()`.

# Обработка исключительных ситуаций

Типы исключений которые может генерировать функция, перечисляются после ключевого слова `throw` после прототипа функции.

---

```
void f1() throw (int, const char*) /* может
    генерировать только типов int или char* */
void f2() throw (Oops*) /* исключения типа
    указатель на класс */
```

---

# Обработка исключительных ситуаций

---

```
void f1() throw ()  
{  
    //Тело функции, не порождающей исключений  
}
```

---

# Обработка исключительных ситуаций

При переопределении виртуальной функции можно задавать список исключений такой же или более ограниченный чем в базовом классе.

Если функция вызывает не описанное исключение, вызывается функция `unexpected()`.

# Исключения в конструкторах и деструкторах

---

```
class Vector {
public:
    class Size {};
enum {max = 32000};
Vector(int n)
{ if(n<0 || n > max) throw Size(); }
};

try {
    Vector *p = new Vector(i);
    ...
}
catch(Vector::Size) { //Обработка ошибки размера
    вектора }
```

---

Если в конструкторе генерируется исключение то автоматически вызываются деструкторы для полностью созданных в этом блоке объектов. Например, если исключение было вызвано при создании массива объектов, то деструкторы будут вызваны только для успешно созданных элементов. Если память выделяется динамически с помощью операции new и в конструкторе возникает исключение, память из-под объекта корректно освобождается.

# Стандартные исключения

- `bad_alloc`
- `bad_cast`
- `bad_typeid`
- `bad_exception`

## Файловые потоки:

- ifstream — входной файловый поток
- ofstream — выходной файловый поток
- fstream — двунаправленный

---

```
char buf[100];
ifstream fff("file.txt", ios::in | ios::nocreate);
if(!fff)
cout << "err";
while(!fff.eof())
fff.getline(buf, 100);
```

---

## Строковые потоки:

- `istringstream` — входные строковые потоки;
- `ostringstream` — выходные строковые потоки;
- `stringstream` — двунаправленные строковые потоки.

---

```
#include <sstream>
ostringstream os;
time_t t;
time(&t);
os << "time:" << ctime(&t);
```

---

# Потоки и типы определенные пользователем

---

```
friend ostream& operator << (ostream& out, MyClass&
    C)
{ return out << "x=" << C.x << "y=" << C.y; }

friend istream& operator >> (istream& in, MyClass&
    C)
{ cout << "Enter x:"; in >> C.x;
  cout << "Enter y:"; in >> C.y;
  return in;
}
```

---

Класс `string` входит в стандартную библиотеку C++.

---

```
string s1("Text");
string s2;
```

```
char *str = "Txt";
string s3(str);
```

---

## Функции:

---

```
s.at(1);  
s1.append(s2); // +  
s1.append(s2, 3, 5);  
s1.insert(1, str2, 3, 5);  
s1.replace(1, 2, str2, 5);  
s1.swap(s2);  
s1.c_str(); //возвращает const char*  
s1.find(s2); //возвращает позицию  
s1.compare(1, 2, s2, 1, 2); //аналог strstr
```

---

## Последовательные контейнеры:

- `vector` — структура, эффективно реализующая произвольный доступ к элементам, а также добавление в конец и удаление из конца;
- `dequeue` — структура, эффективно реализующая произвольный доступ к элементам, а также добавление в оба конца и удаление из обоих концов;
- `list` — список, эффективно реализует вставку и удаление элементов в произвольное место, но не имеет произвольного доступа к своим элементам.

---

```
vector<int> v;
v.push_back(5);
v.push_back(6);
v.push_back(7);
for( vector<int>::iterator i = v.begin(); i != v.
    end(); i++)
{ cout << *i << " "; }
```

---

## Функции:

---

```
insert() //вставка в произвольное место  
erase() //удаление из произвольного места  
at, [] //произвольный доступ к элементу
```

---

Ассоциативные контейнеры:

- map — словарь
- multimap — словарь с дубликатами

# std::map

---

```
map<int ,int> maps;
maps[100] = 1;
maps[200] = 3;
```

---

# Технология ООП

Санкт-Петербургский государственный политехнический университет

25 октября 2011

Qt — кросс-платформенный инструментарий разработки ПО на языке программирования C++.

- Qt Commercial
- GNU GPL
- GNU LGPL

# Преимущества перед другими фреймворками

- Объектно-ориентированная архитектура библиотеки.
- Открытость.
- Кроссплатформенность.
- Отличная документация.
- Обилие примеров.
- Множество готовых классов и методов.

- QtCore
- QtGui
- QtNetwork
- QtOpenGL
- QtSql

<http://qt.nokia.com/downloads> :

- Qt SDK (1.2Gb):
  - Qt libraries
  - Qt Creator IDE
  - Qt development tools
- Qt libraries for Windows:
  - MinGW 4.4
  - VS 2008
- Qt Creator for Windows
- Qt Visual Studio Add-in

- download qt-everywhere-opensource-src-4.7.4.zip
- Start > Programs > Microsoft Visual Studio 2010 > Visual Studio Tools > Visual Studio Command Prompt.
- set QTDIR=C:\Qt\4.7.4
- set QMAKESPEC=win32-msvc2010
- cd /d C:\Qt\4.7.4
- configure -debug-and-release -opensource -platform win32-msvc2010
- nmake
- Qt > Qt Options > Qt Versions > Add

- Download jom <ftp://ftp.qt.nokia.com/jom>
- Extract to C:\jom
- Вместо nmake используйте C:\jom\jom.exe -j 4 , где 4 — число ядер процессора.

- Qt Assistant
- частично-переведенная документация  
<http://doc.crossplatform.ru/qt/>
- examples && demos
- <http://www.developer.nokia.com/>

# Пример

Сборка приложений:

- qmake -project
- qmake
- nmake

Или для Visual Studio:

- qmake -project
- qmake -tp vc
- Открыть получившийся .vcproj в VS.

# Пример

---

```
#include <QApplication>
#include <QLabel>

int main(int argc, char **argv)
{
    QApplication app(argc, argv);
    QLabel *label = new QLabel("HelloWorld!");
    label->show();
    return app.exec();
}
```

---

# Форматирование HTML

```
QLabel *label = new QLabel("<h2><i>Hello</i><font color=red>World!</font></h2>");
```

---

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char **argv)
{
    QApplication app(argc, argv);
    QPushButton *button = new QPushButton("Quit");
    QObject::connect(button, SIGNAL(clicked()), &app, SLOT(quit()));
    button->show();
    return app.exec();
}
```

---

Сигналы и слоты — это фундаментальный механизм Qt, позволяющий связывать объекты друг с другом. Связанным объектам нет необходимости что-либо «знать» друг о друге. Сигналы и слоты гораздо удобнее механизма функций обратного вызова (callbacks) и четко вписываются в концепцию ООП.

Для использования этого механизма объявление класса должно содержать специальный макрос Q\_OBJECT на следующей строке после ключевого слова class:

---

```
class MyClass {  
Q_OBJECT  
  
public:  
...  
};
```

---

После макроса Q\_OBJECT не нужно ставить точку с запятой. Перед выполнением компиляции, Meta Object Compiler (MOC) анализирует такие классы и автоматически внедряет в них всю необходимую информацию.

Сигнал может быть определен следующим образом:

---

```
class MyClass: public QObject {  
Q_OBJECT  
  
public:  
...  
  
signals:  
void mySignal();  
  
};
```

---

Для того чтобы инициировать сигнал (выслать сигнал) нужно использовать ключевое слово `emit`.

Сигналы могут использовать параметры для передачи дополнительной информации.

---

```
void MyClass :: sendMySignal()
{
    emit mySignal();
}
```

---

# Слоты

Слоты практически идентичны обычным членам-методам C++, при их объявлении можно использовать стандартные спецификаторы доступа public, protected или private.

---

```
class MyClass: public QObject {  
Q_OBJECT  
  
public slots:  
    void mySlot()  
    {  
        ...  
    }  
};
```

---

# Соединение сигналов и слотов

Для соединения сигналов и слотов можно использовать статический метод `connect`, определенный в классе `QObject`. Один сигнал может быть соединен со многими слотами, а также множество сигналов могут быть соединены с единственным слотом.

В общем виде соединение выглядит следующим образом:

---

```
connect(sender, SIGNAL(signal), receiver, SLOT(slot))
```

---

`sender` и `receiver` — это указатели на `QObject`. `signal` и `slot` — сигнатуры сигнала и слота.

Пример соединения:

---

```
QObject :: connect( spinBox , SIGNAL( valueChanged( int ) )
    , slider , SLOT( setValue( int ) ) );
```

---

sender и receiver — это указатели на QObject. signal и slot — сигнатуры сигнала и слота.

Метод `disconnect` можно использовать для того, чтобы удалить соединение между сигналом и слотом.

---

```
disconnect( sender0 , SIGNAL( overflow() ) , receiver1 ,
            SLOT( handleMathError() ) );
```

---

На практике прямой вызов `disconnect` используется редко, так как Qt автоматически удаляет все соединения при удалении объектов.

```
#include <QApplication>
#include <QHBoxLayout>
#include <QSlider>
#include <QSpinBox>

int main(int argc, char **argv)
{
    QApplication app(argc, argv);
    QWidget *window = new QWidget;
    window->setWindowTitle("Age:");
    QSpinBox *spinBox = new QSpinBox;
    QSlider *slider = new QSlider(Qt::Horizontal);
    spinBox->setRange(0, 130);
    slider->setRange(0, 130);
    QObject::connect(spinBox, SIGNAL(valueChanged(int))
                     ,
                     slider, SLOT(setValue(int))));
```

```
QObject::connect( slider , SIGNAL(valueChanged(int)) ,  
spinBox , SLOT(setValue(int))) ;  
spinBox->setValue(35) ;  
QHBoxLayout *layout = new QHBoxLayout ;  
layout->addWidget( spinBox ) ;  
layout->addWidget( slider ) ;  
window->setLayout( layout ) ;  
window->show() ;  
    return app.exec() ;  
}
```

# Базовый класс QObject

Фундамент Qt - это объектно-ориентированная модель.

Абсолютное большинство классов - это наследники QObject.

Данный базовый класс содержит поддержку основополагающих механизмов, которые активно используются потомками:

- сигналы и слоты
- метаинформация
- иерархии объектов
- события, фильтры событий
- свойства
- приведение типов
- таймеры

QObject в качестве предка при организации множественного наследования предполагает два ограничения:

- В перечне предков QObject (или его потомок) должен быть объявлен первым.
- Только один из предков может быть наследником QObject.

Объекты класса QObject (или его потомков) нельзя непосредственно копировать или передавать как значения так как это противоречит многим ключевым особенностям, таким как иерархические структуры, привязки сигналов/слотов.

Конструктор копирования и оператор присваивания QObject объявлены пустыми с помощью макроса `Q_DISABLE_COPY`

Метаинформация содержит:

- данные о классе
- информацию о наследовании
- информацию о слотах/сигналах

Для хранения метаинформации используется класс QMetaObject, объектом которого владеет QObject.

Получить указатель на QMetaObject можно с помощью метода metaObject():

---

```
// obj – объект класса потомка QObject
if( obj->metaObject()->className() == "FirstClass"
    )
{
    // obj является объектом класса FirstClass
    ...
}
```

---

Для проверки информации о наследовании можно воспользоваться методом `QObject::inherits()`:

```
// obj – объект класса потомка QObject
if( obj->inherits("QAbstractButton") )
{
    QAbstractButton* button = static_cast<
        QAbstractButton*>(obj);
    ...
}
```

Тоже самое можно сделать воспользовавшись  
qobject\_cast<T>:

---

```
QAbstractButton* button = qobject_cast<
    QAbstractButton*>(obj);
if( button )
{
    ...
}
```

---

qobject\_cast — является аналогом dynamic\_cast, но не требует включения RTTI и для наследников QObject класса работает намного быстрее.

# Иерархии объектов в Qt

Для построения иерархий необходимо использовать классы, рожденные от `QObject`. Объекты классов должны создаваться динамически.

Класс `QObject` содержит реализацию всех необходимых методов для организации иерархий объектов. Конструктор `QObject` выглядит следующим образом:

---

```
QObject( QObject *parent=0);
```

---

Одно из основных достоинств иерархий объектов — это автоматическое удаление всех дочерних объектов при удалении корневого объекта. Разработчику достаточно удалить только те объекты, которые являются вершинами.

Методы QObject для работы с иерархиями:

- `setParent()` — позволяет задать нового родителя.
- `parent()` — возвращает указатель на текущего родителя.
- `children()` — возвращает указатель на список дочерних объектов.
- `findChildren()` — позволяет искать дочерние объекты по имени (поддерживаются регулярные выражения).
- `dumpObjectInfo()` — отображает отладочную информацию об иерархии.
- `setObjectName()/objectName()` — задать/получить имя объекта.

# Обработка событий в классе QObject

События используются для оповещения объектов о возникновении/изменении каких-либо ситуаций. Основными источниками событий являются элементы пользовательского интерфейса, кроме того существуют события, генерируемые системными объектами (например, таймерами). В общем случае, сразу после возникновения, событие помещается в очередь для дальнейшей обработки.

# Таймеры класса QObject

Класс QObject содержит встроенные таймеры, которые позволяют организовать периодическое повторение определенных действий. Для использования таймеров QObject содержит следующие методы:

- int startTimer( int interval )
- virtual void timerEvent( QTimerEvent\* event )
- void killTimer( int id )

## Таймеры класса QObject

Объект QObject позволяет создавать множество таймеров. При реализации потомка таймеры могут быть запущены следующим образом:

---

```
void MyObject :: beginPolling()
{
    timer1 = startTimer( 1000 );    // каждую секунду
    timer2 = startTimer( 60000 );   // каждую минуту
}
```

---

## Таймеры класса QObject

При этом виртуальный метод timerEvent() должен действовать в зависимости от идентификатора вызывающего таймера:

```
void MyObject :: timerEvent( QTimerEvent* event )
{
    switch( event->timerId() )
    {
        case timer1:
            // ежесекундное действие
            break;

        case timer2:
            // ежеминутное действие
            break;

        default:
            break;
    }
}
```

# Технология ООП

Санкт-Петербургский государственный политехнический университет

1 ноября 2011

# Иерархии объектов в Qt

Для построения иерархий необходимо использовать классы, рожденные от `QObject`. Объекты классов должны создаваться динамически.

Класс `QObject` содержит реализацию всех необходимых методов для организации иерархий объектов. Конструктор `QObject` выглядит следующим образом:

---

```
QObject( QObject *parent=0);
```

---

Одно из основных достоинств иерархий объектов — это автоматическое удаление всех дочерних объектов при удалении корневого объекта. Разработчику достаточно удалить только те объекты, которые являются вершинами.

Методы QObject для работы с иерархиями:

- `setParent()` — позволяет задать нового родителя.
- `parent()` — возвращает указатель на текущего родителя.
- `children()` — возвращает указатель на список дочерних объектов.
- `findChildren()` — позволяет искать дочерние объекты по имени (поддерживаются регулярные выражения).
- `dumpObjectInfo()` — отображает отладочную информацию об иерархии.
- `setObjectName()/objectName()` — задать/получить имя объекта.

# Обработка событий в классе QObject

События используются для оповещения объектов о возникновении/изменении каких-либо ситуаций. Основными источниками событий являются элементы пользовательского интерфейса, кроме того существуют события, генерируемые системными объектами (например, таймерами). В общем случае, сразу после возникновения, событие помещается в очередь для дальнейшей обработки.

# Таймеры класса QObject

Класс QObject содержит встроенные таймеры, которые позволяют организовать периодическое повторение определенных действий. Для использования таймеров QObject содержит следующие методы:

- `int startTimer( int interval )`
- `virtual void timerEvent( QTimerEvent* event )`
- `void killTimer( int id )`

## Таймеры класса QObject

Объект QObject позволяет создавать множество таймеров. При реализации потомка таймеры могут быть запущены следующим образом:

---

```
void MyObject :: beginPolling()
{
    timer1 = startTimer( 1000 );    // каждую секунду
    timer2 = startTimer( 60000 );   // каждую минуту
}
```

---

## Таймеры класса QObject

При этом виртуальный метод timerEvent() должен действовать в зависимости от идентификатора вызывающего таймера:

```
void MyObject :: timerEvent( QTimerEvent* event )
{
    switch( event->timerId() )
    {
        case timer1:
            // ежесекундное действие
            break;

        case timer2:
            // ежеминутное действие
            break;

        default:
            break;
    }
}
```

# qmake

qmake создает файл сборки, основываясь на информации в файле проекта.

qmake содержит дополнительные свойства для поддержки разработки с Qt, включая автоматическое создание правил для мос и uiс.

qmake имеет два режима работы, в первом он генерирует проектные файлы (.pro), а во втором на основе проектных файлов генерирует правила сборки. Опции:

- -recursive
- -nodepend
- -nomoc
- -Wnone
- -Wall

## файлы проекта qmake. Переменные

В файле проекта, переменные используются для хранения списков строк. В простых проектах эти переменные информируют qmake о параметрах настройки, именах файлов и каталогах, которые используются в процессе сборки.

HEADERS = mywidget.h mylist.h \\  
test.h

Оператор **\$\$** используется для извлечения содержимого переменной и может быть использован для передачи значений между переменными или передачи их в функции:

EVERYTHING = \$\$SOURCES \$\$HEADERS

Специальный оператор `$$[...]` может быть использован для получения доступа к различным опциям конфигурирования, которые были установлены при сборке Qt:

```
message(Qt version: $$[QT_VERSION])
```

Для получения содержимого значения окружения при запуске qmake используйте оператор `$$(...)`:

```
DESTDIR = $$($PWD) #выполняется на этапе обработки  
проекта
```

```
DESTDIR = $(PWD) #выполняется на этапе обработки  
makefile'a
```

## Операторы:

- variable = value
- variable += value
- variable -= value
- variable \*= value
- variable /= value

DEFINES += QT\_DLL

CONFIG += gui

# Комментарии

Для добавления комментария в файл проекта используется символ `#`. Если необходимо вставить символ `#` как часть переменной, то нужно использовать переменную `LITERAL_HASH`.

```
urlPieces = http://qt.nokia.com/doc/4.0/qtextdocument.html  
pageCount  
message($$join(urlPieces, $$LITERAL_HASH))
```

# Области видимости

Области видимости аналогичны операторам if в процедурных языках программирования. Если некоторое условие истинно, то объявления внутри области видимости обрабатываются.

---

```
<условие> {
    <команда или определение>
    ...
}
```

---

Открывающая скобка должна находиться в той же строке, что и условие.

---

```
win32 {
    SOURCES += paintwidget_win.cpp
}
```

---

# Области видимости

Области видимости допускают вложенность, а также сокращенную запись используя оператор «`:`». Значения, сохраненные в переменной CONFIG, специально обрабатываются для qmake. Например:

`CONFIG += opengl`

---

```
opengl {  
    TARGET = application-gl  
} else {  
    TARGET = application
```

---

Простые циклы создаются с помощью перебора списка значений, используя встроенную функцию `for`. Следующий код добавляет каталоги в переменную `SUBDIRS`, но только в том случае, если они существуют:

---

```
EXTRAS = handlers tests docs
for(dir , EXTRAS) {
    exists($$dir) {
        SUBDIRS += $$dir
    }
}
```

---

# Встроенные функции

Встроенные функции позволяют обрабатывать содержимое переменных. Эти функции обрабатывают переданные им аргументы и возвращают в качестве результата значение или список значений.

---

```
HEADERS = model.h
HEADERS += $$OTHER_HEADERS
HEADERS = $$unique(HEADERS)
win32:INCLUDEPATH += $$quote(C:/mylibs/extra
    headers)
```

---

Выполняются только при выполнении условия, например,  
isEmpty(variablename):

---

```
isEmpty( CONFIG ) {  
CONFIG += qt warn_on debug  
}
```

---

# Пользовательские функции

---

```
defineReplace(headersAndSources) {
    variable = $$1
    names = $$eval($$variable)
    headers =
    sources =

    for(name, names) {
        header = $$${name}.h
        exists($$header) {
            headers += $$header
        }
        source = $$${name}.cpp
        exists($$source) {
            sources += $$source
        }
    }
    return($$headers $$sources)
}
```

Перечень переменных которые распознаются Qt:

- CONFIG
- DESTDIR
- FORMS
- HEADERS
- QT
- RESOURCES
- SOURCES
- TEMPLATE

# Шаблоны проекта

Переменная TEMPLATE используется для определения типа проекта, который будет собран. Доступные типы проектов:

- app
- lib
- subdirs
- vcapp
- vclib
- vcsubdirs

# Общие настройки

Переменная CONFIG определяет параметры и возможности, которые должен использовать компилятор, и библиотеки, с которыми будет идти компоновка.

- release
- debug
- debug\_and\_release
- ordered
- warn\_on
- warn\_off
- qt
- thread
- x11

# Объявление библиотек Qt

Если переменная CONFIG содержит значение qt, то qmake поддерживает приложения Qt.

- core
- gui
- network
- opengl
- sql
- svg
- xml
- qt3support

---

```
CONFIG += qt
QT += network xml
```

---

# Объявление библиотек Qt

Если в проекте используются не только библиотеки Qt, то дополнительные библиотеки для линковки, а также пути поиска заголовочных файлов можно добавить:

---

```
LIBS += -Ld:/stl/lib -lmath  
INCLUDEPATH = c:/msdev/include d:/stl/include
```

---

Мета-объектный компилятор(MOC) — программа, которая обрабатывает расширения C++ от Qt. Каждый класс использующий возможности мета-объектной системы должен быть обработан мос.

Инструмент мос читает заголовочный файл C++. Если он находит одно или более объявлений классов, которые содержат макрос Q\_OBJECT, то он порождает файл исходного кода C++, содержащий мета-объектный код для этих классов.

# Использование MOC

Обычно moc используется с входным файлом, содержащим такое объявление класса:

---

```
class MyClass : public QObject
{
    Q_OBJECT
public:
    MyClass( QObject *parent = 0 );
    ~MyClass();
};
```

---

После чего будет сгенерирован файл .cpp на основании файла, например, если файл с объявлением был `myclass.h`, то на выходе получится `moc_myclass.cpp`.

Мос не обрабатывает весь C++ из-за этого классы-шаблоны не могут содержать сигналы и слоты.

Вызов препроцессора mos происходит до вызова препроцессора языка, а значит и до выполнения подстановки директив компилятора, таких как define

# Механизм свойств

Классы, унаследованные от `QObject`, могут использовать механизм свойств, поддерживаемый библиотекой Qt.

Свойства — это поля класса, который определяются специальным образом и благодаря которым к атрибутам объекта класса можно получить доступ извне (например, из `Qt Script`).

Для определения свойств используются директивы препроцессора:

---

```
Q_PROPERTY  
(  
    type name  
    READ getFunction  
    [WRITE setFunction]  
    [RESET resetFunction]  
    [DESIGNABLE bool]  
    [SCRIPTABLE bool]  
    [STORED bool]  
)
```

- WRITE — запись значения
- RESET — сброс значения
- DESIGNABLE — отображение в инспекторе свойств
- SCRIPTABLE — доступность в Qt Script
- STORED — поддержка сериализации

## Пример задания свойства

---

```
class MyExample : public QObject {
Q_OBJECT
Q_PROPERTY(int firstProperty READ getFirstPropValue
           WRITE setFirstPropValue)
private:
int m_firstProp;
public:
int getFirstPropValue() { return m_firstProp; }
void setFirstPropValue(int value) { m_firstProp =
    value; }
};
```

---

## Пример задания свойства

---

```
myExample->setProperty ("firstProperty" , 10);
int propValue = myExample->property ("firstProperty"
).toInt();
```

---

# Строковые значения

Класс QString представляет собой строку символов Unicode. QString хранит строку 16-битных QChar, где каждый QChar хранит символ Unicode 4.0.

---

```
const QChar * constData () const
bool isEmpty () const
boolisNull () const
int length () const
QChar at ( int i ) const
QByteArray toAscii () const
...
```

---

## Строковые значения

---

```
QString str = "Hello";
QString str1;
if (str == "test1" || str == "Hello" )
{ ... }

static const QChar data[4] = { 0x0055 , 0x006e , 0
    x10e3 , 0x03a3 };
QString str(data , 4);

QString str = "and";
str.prepend("rock" );           // str == "rock and"
str.append("u roll");           // str == "rock and
    roll"
str.replace(5 , 3 , "&");      // str == "rock & roll"
```

---

# Строковые значения

Qt также предоставляет класс `QByteArray` для хранения произвольных байтов и 8-битных оканчивающихся на '\0' строк.

---

```
const char * constData () const
bool isEmpty () const
boolisNull () const
int length () const
char at ( int i ) const
int indexOf ( char ch, int from = 0 ) const
int lastIndexOf ( const QByteArray & ba, int from =
-1 ) const
bool contains ( const QByteArray & ba ) const
```

---

# QStringList

Список строк, аналог конструкции `QList<QString>`.

---

```
QStringList fonts;
fonts << "Arial" << "Helvetica" << "Times" <<
    "Courier";

for (int i = 0; i < fonts.size(); ++i)
    cout << fonts.at(i).toLocal8Bit().
        constData() << endl;

QStringList::const_iterator constIterator;
for (constIterator = fonts.constBegin();
    constIterator != fonts.constEnd();
    ++constIterator)
    cout << (*constIterator).toLocal8Bit().
        constData() << endl;

QString str = fonts.join(",");
QStringList list;
list = str.split(",");
```

# Технология ООП

Санкт-Петербургский государственный политехнический университет

8 ноября 2011

- отладчик
- QObject::dumpObjectInfo()
- Q\_ASSERT() — выводит предупреждение, если не равно true
- Q\_CHECK\_PTR() — проверяет указатель на NULL
- qDebug(), qWarning(), qFatal()

В процессе отладки рекомендуется присваивать объектам имена при помощи функции `QObject::objectName()`

---

```
qDebug() << "Test" << QString("String") << QChar('x') << QRect(0, 10, 50, 40);
```

---

```
qDebug("%d", i);
```

# Глобальные определения

Содержатся в заголовочном файле QtGlobal

- qMax, qMin
- qAbs, qRound

---

```
int max = qMax<int>(3, 5);  
int abs = qAbs(-5);  
int rnd = qRound(-5.2);
```

---

## Типы Qt

- qint8, qunit8
- ...
- qint64, quint64
- qlonglong
- qulonglong

Последовательные и ассоциативные.

Последовательные:

- QVector<T>
- QList<T>
- QLinkedList<T>
- QStack
- QQueue

Ассоциативные:

- QSet<T>
- QMap<K,T>
- QMultiMap<K,T>
- QHash<K,T>
- QMultiHash<K,T>

Во всех контейнерных классах переопределены операции:

- == !=
- = , кроме QSet
- begin() end()
- clear()
- insert()
- remove()
- size() count()
- value() , кроме QSet
- empty isEmpty()

Java Style:

---

```
QList<QString> list;  
QListIterator<QString> it(list);  
while(it.hasNext()) { qDebug() << it.next(); }
```

---

Если необходимо изменять значения списка то для этого надо воспользоваться QMutableListIterator'ом.

Java Style:

---

```
QList<QString> list;
QMutableListIterator<QString> it( list );
while( it.hasNext() ) { if( it.next() == "test" ) { it.
    setValue("new"); } }
```

---

STL Style:

---

```
QList<QString> list;  
QList<QString>::iterator it = list.begin();  
for(; it != vec.end(); it++)  
{ qDebug() << "Element" << *it }
```

---

# Итераторы

Если не планируется изменять значения элементов, то эффективнее использовать const\_iterator;

---

```
QList<QString> list;
QList<QString>::const_iterator it = list.constBegin();
for(; it != vec.constEnd(); it++)
{ qDebug() << "Element is" << *it }
```

---

foreach — макрос который позволяет перебрать элементы

---

```
QList<QString> list;  
foreach (QString str, list) { qDebug() << str; }
```

---

# Методы последовательных контейнеров

- +
- += «
- at()
- back() last()
- front() first()
- contains()
- indexOf()
- lastIndexOf()
- push\_back() append()
- push\_front() prepend()

---

```
QVector<QString> vec;  
vec.append("string");
```

---

Представляет собой динамический массив. Вставка в начало и в середину происходят за время  $O(n)$ , в конец массива может выполняться за время  $O(1)$ .

- `data()`
- `resize()`
- `reserve()`

---

```
QVector<QString> vec;
vec.push_back("string");
vec.push_back("string2");
vec.push_back("string3");
```

---

# QList

Представляет собой упорядоченный набор связанных друг с другом элементов. Поиск выполняется за время  $O(1)$ , вставка в середину за  $O(n)$ , вставка в начало или конец списка как правило за  $O(1)$ , эти времена достигаются за счет того что внутри QList реализован на базе массива.

- swap()
- move()
- takeAt()

---

```
QList<QString> lst;
lst << "str1" << "str2";
QList<QString>::iterator it;
for( it = lst.begin(); it != lst.end(); ++it )
{ qDebug() << *it; }
```

---

# QLinkedList

Двусвязный список. Вставка в любое место списка выполняется за время  $O(1)$ , поиск за  $O(n)$ .

---

```
QLinkedList<QString> lst;
lst << "str1" << "str2";
QLinkedList<QString>::iterator it;
for( it = lst.begin(); it != lst.end(); ++it )
{ qDebug() << *it; }
```

---

Стек — реализует структуру данных, работающую по принципу LIFO. Наследуется от QVector'a.

- push()
- pop()
- top()

---

```
QStack<int> stack ;
stack . push(1) ;
stack . push(2) ;
stack . push(3) ;
while (!stack . isEmpty())
    cout << stack . pop() << endl ;
```

---

Стек — реализует структуру данных, работающую по принципу FIFO. Наследуется от QList'a.

- enqueue()
- dequeue()
- head()

---

```
QQueue<int> queue;
queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);
while (!queue.isEmpty())
    cout << queue.dequeue() << endl;
```

---

Для всех контейнеров этого типа доступны методы:

- contains()
- erase()
- find()
- insert() отсутствует в QSet
- insertMulti() отсутствует в QSet
- key() отсутствует в QSet
- keys() отсутствует в QSet
- keys() отсутствует в QSet
- values()

Словари хранят элементы одного и того же типа, индексируемые ключевыми значениями, в QMap — ключи должны быть уникальны в отличии от QMultiMap (элементы словаря отсортированы по ключу). Вставка и поиск элементов осуществляются за время  $O(\log n)$ .

## QMap<K,T> QMultiMap<K,T>

```
QMap<QString , int> map
map["one"] = 1;
map["three"] = 3;
map["seven"] = 7
map.insert("twelve", 12);
int num1 = map["thirteen"];
int num2 = map.value("thirteen");
QMap<QString , int>::iterator it = map.begin();
for( ; it != map.end(); it++)
{
qDebug << it.key() << it.value();
}
if(map.contains("one")) { qDebug << "one"; }
```

## QMap<K,T> QMultiMap<K,T>

Если необходимо связать с одним ключем несколько значений, например, в адресной книге, то необходимо использовать структуру QMultiMap.

---

```
QMultiMap<QString , int> map
map.insert("twelve", 12);
map.insert("twelve", 13);
QMap<QString , int >::iterator it = map.find("twelve");
for( ; it != map.end() && it.key() == "twelve"; it++)
{
    qDebug << it.key() << it.value();
}
if(map.contains("one")) { qDebug << "one"; }
```

---

## QHash<K,T>

В отличии от словарей не используют сортировку по ключу, а использует хэш-таблицу, что позволяет работать с этой коллекцией намного быстрее. Время доступа к элементам  $O(1)$ , в худшем случае  $O(n)$ , время вставки элемента  $O(1)$ , в худшем случае  $O(n)$ .

При использовании оператора `[]`, как для `QHash` так и для `QMap` следует учитывать, что если при сравнении элемент не обнаружен он будет создан.

---

```
QHash<QString , int> hash;
hash.insert("twelve", 12);
hash.insert("twelve", 13);
QHash<QString , int>::iterator it = hash.find("twelve");
for( ; it != hash.end() && it.key() == "twelve";
    it++)
{
    qDebug << it.key() << it.value();
}
```

Для создания коллекции из собственных классов, необходимо переопределить оператор `==` и специализированную функцию `qHash`, которая должна возвращать уникальное число для каждого находящегося в хеше элемента.

Является частным случаем таблицы QHash, хранит внутри себя только ключи без значений. Позволяет выполнять операции над множествами, такие как объединение, пересечение, разность.

- intersect()
- subtract()
- toList()
- unite()

---

```
QSet<QString> set1;
QSet<QString> set2;
set1 << "str1";
set2 << "str2";
set1.unite(set2);
set1.intersect(set2);
```

---

Входят в заголовочный файл `QtAlgorithms` и предоставляют операции применяемые к контейнерам.

- `qBinaryFind()`
- `qCopy()`
- `qCopyBackward()`
- `qCount()`
- `qDeleteAll()`
- `qEqual()`
- `qFill()`
- `qFind()`
- `qLowerBound()`
- `qUpperBound()`
- `qSwap()`

# qSort

Для функции qSort необходимо чтобы были переопределены операторы сравнения строк.

---

```
QList<int> list;  
list << 33 << 12 << 68 << 6 << 12;  
qSort(list.begin(), list.end());  
// list: [ 6, 12, 12, 33, 68
```

---

Отвечает за поиск элементов в коллекции, возвращает итератор установленный на первый найденный элемент.

---

```
QStringList list;
list << "one" << "two" << "three";

QStringList::iterator i1 = qFind( list.begin() ,
    list.end(), "two");
// i1 == list.begin() + 1

QStringList::iterator i2 = qFind( list.begin() ,
    list.end(), "seventy");
// i2 == list.end()
```

---

Позволяет сравнить две коллекции различных типов, например QList и QVector. В качестве первого и второго параметра передаются итераторы указывающие на начало и конец первой последовательности, а в качестве третьего итератор на начало второй последовательности.

## qEqual

---

```
QStringList list;
list << "one" << "two" << "three";

QVector<QString> vect(3);
vect[0] = "one";
vect[1] = "two";
vect[2] = "three";

bool ret1 = qEqual(list.begin(), list.end(), vect.
    begin());
// ret1 == true

vect[2] = "seven";
bool ret2 = qEqual(list.begin(), list.end(), vect.
    begin());
// ret2 == false
```

---

Заполнить коллекцию какими-либо заданными значениями.

---

```
QStringList list;
list << "one" << "two" << "three";

qFill(list.begin(), list.end(), "eleven");
// list: [ "eleven", "eleven", "eleven" ]

qFill(list.begin() + 1, list.end(), "six");
// list: [ "eleven", "six", "six" ]
```

---

# Регулярные выражения

Описываются классом QRegExp, представляют из себя шаблон который предназначен для поиска текста в строке.

---

```
. //a.b
$ //Abc$
[] // [abc]
- // [0-9A-Za-z]
^ // [^def]
* // A*b
+ // A+b
? // A?b
{ n } // A{3}b
{ n, } // a{3,}b
{ ,n } // a{,3}b
{ n,m } // a{2,3}b
| // ac/bc
\b // a\b
\B // a\Bd
```

---

# Регулярные выражения

---

```
( ) // (ab | ac)ad  
\d // q\d // q2  
\D // a \D // ad  
\s //  
\S // a  
\w // c  
\W // 2  
\A // \A qwert
```

```
QRegExp reg ("  
[0-9]{1,3}\\. [0-9]{1,3}\\. [0-9]{1,3}\\. [0-9]{1,3}  
");  
QString str ("my ip address is 192.168.0.1");  
qDebug() << str.contains (reg) : 1 ? 0;  
QRegExp rx ("^\\d\\d?$");  
rx . indexIn ("123");  
rx . indexIn ("-6");  
rx . indexIn ("6");
```

# Регулярные выражения

```
QRegExp rx("*.html");
rx.setPatternSyntax(QRegExp::Wildcard);
rx.exactMatch("index.html");
rx.exactMatch("default.htm");
rx.exactMatch("readme.txt");
```

# Объект QVariant

Позволяет хранить объекты произвольных типов.

---

```
QVariant v(123);
int x = v.toInt();
v = QVariant("hello");
int y = v.toInt();
QString s = v.toString();
```

---

# Технология ООП

Санкт-Петербургский государственный политехнический университет

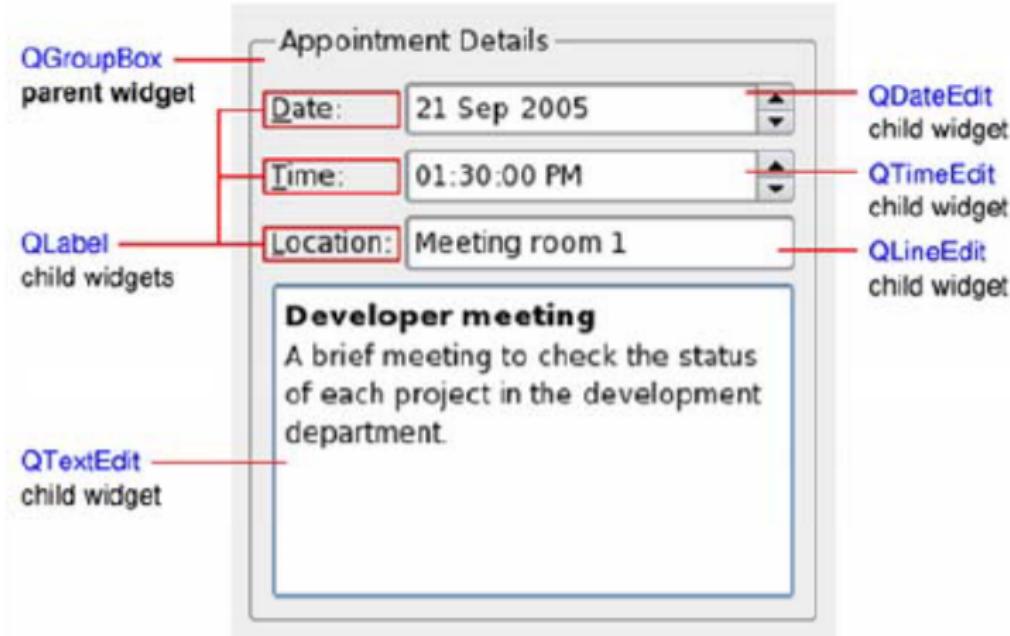
15 ноября 2011

Класс QWidget является фундаментальным и содержит в себе огромное количество методов и свойств, для таких каких вещей как изменение местоположения, обработки событий и т.п. QWidget унаследован от QObject что позволяет использовать механизмы сигналов/слотов.

Любой QWidget может являться контейнером для другого виджета. Виджеты могут иметь потомков которые будут отображаться внутри предка — это позволяет наследовать свойства виджетов расположенных на вершине иерархии, т.е. скрыв один виджет верхнего уровня автоматически скрываются все его потомки.

Виджет без потомка называется виджетом верхнего уровня, и представляет из себя отдельное окно.

# QWidget



# QWidget

---

```
QWidget ( QWidget * parent = 0, Qt::WindowFlags f = 0 )
```

---

```
bool isVisible () const
bool isVisibleTo ( QWidget * ancestor ) const
virtual void setVisible ( bool visible )
void show () [slot]
bool isHidden () const
void hide () [slot]
```

---

---

```
const QFont & font () const
void setFont ( const QFont & )
QCursor cursor () const
void setCursor ( const QCursor & )
void unsetCursor ()
const QPalette & palette () const
void setPalette ( const QPalette & )
```

---

---

```
QString windowTitle () const  
void setWindowTitle ( const QString & )  
QIcon windowIcon () const  
void setWindowIcon ( const QIcon & icon )
```

---

---

```
Qt::WindowStates windowState () const
void showNormal () [slot]
bool isMinimized () const
void showMinimized () [slot]
bool isMaximized () const
void showMaximized () [slot]
bool isFullScreen () const
void showFullScreen () [slot]
bool isWindowModified () const
void setWindowModified ( bool )
```

---

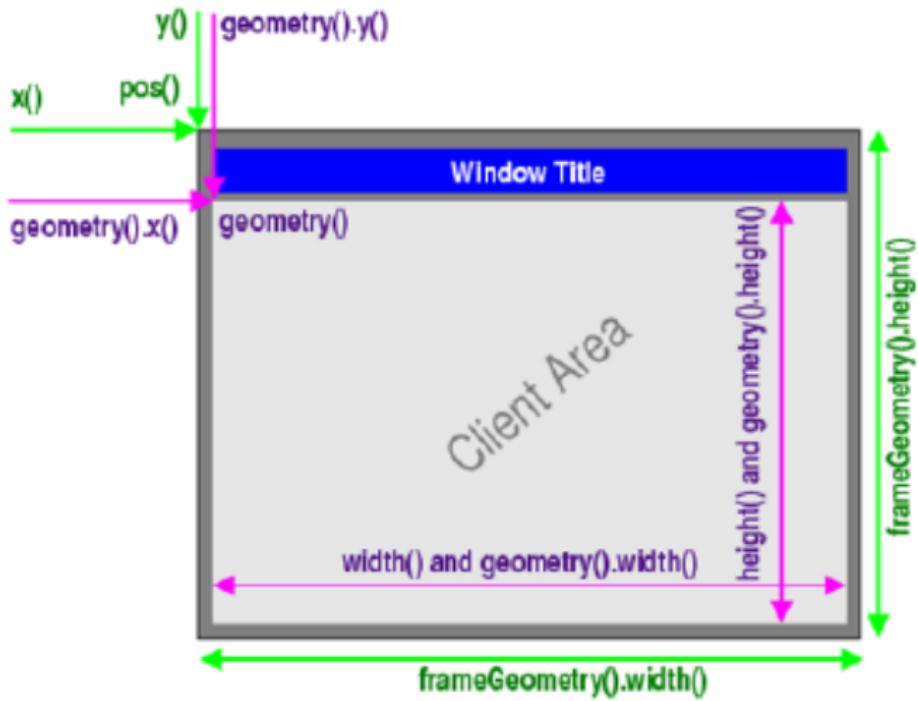
# Свойства виджетов

---

```
QWidget * window () const  
bool isWindow () const  
QWidget * parentWidget () const  
void setParent ( QWidget * parent )
```

---

# Размеры виджетов



---

```
QPoint pos () const
 QRect frameGeometry () const
 void move ( const QPoint & )
 const QRect & geometry () const
 QRect rect () const
 void setGeometry ( const QRect & )
 QSize size () const
 void resize ( const QSize & )
```

---

---

```
virtual QSize sizeHint () const
virtual QSize minimumSizeHint () const
QSize minimumSize () const
void setMinimumSize ( const QSize & )
void setMinimumSize ( int minw, int minh )
QSize sizeIncrement () const
void setSizeIncrement ( const QSize & )
void adjustSize ()
void setFixedSize ( const QSize & s )
```

---

---

```
QRect childrenRect () const
QRegion childrenRegion () const
QPoint mapFromParent ( const QPoint & pos ) const
QPoint mapToParent ( const QPoint & pos ) const
QPoint mapFromGlobal ( const QPoint & pos ) const
QPoint mapToGlobal ( const QPoint & pos ) const
QRegion visibleRegion () const
Qt::WindowModality windowModality () const
```

---

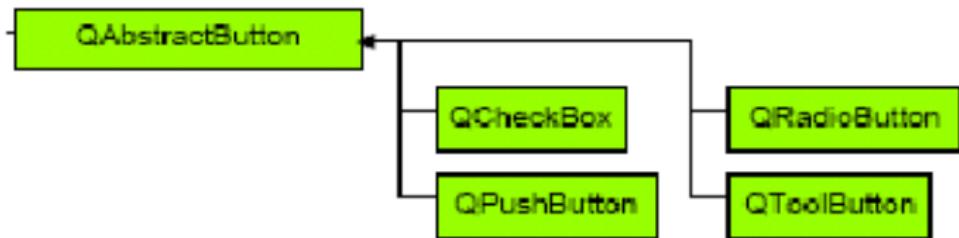
# Взаимоотношения виджетов

---

```
bool isEnabled () const
void setEnabled ( bool )
bool hasFocus () const
void setFocus ( ) [slot]
void clearFocus ()
Qt::FocusPolicy focusPolicy () const
void setFocusPolicy ( Qt::FocusPolicy policy )
bool focusNextChild () [protected]
bool focusPreviousChild () [protected]
```

---

# QAbstractButton



```
bool isCheckable () const
void setCheckable ( bool )
bool isChecked () const
void setChecked ( bool )
bool isDown () const
void setDown ( bool )
bool autoExclusive () const
void setAutoExclusive ( bool )
bool autoRepeat () const
void setAutoRepeat ( bool )
int autoRepeatDelay () const
void setAutoRepeatDelay ( int )
int autoRepeatInterval () const
void setAutoRepeatInterval ( int )
```

---

---

```
QIcon icon () const
void setIcon ( const QIcon & icon )
QSize iconSize () const
void setIconSize ( const QSize & size )
QKeySequence shortcut () const
void setShortcut ( const QKeySequence & key )
QString text () const
void setText ( const QString & text )
```

---

Слоты:

---

```
void animateClick ( int msec = 100 )
void click ()
void setChecked ( bool )
void setIconSize ( const QSize & size )
void toggle ()
```

---

## Сигналы:

---

```
void clicked ( bool checked = false )  
void pressed ()  
void released ()  
void toggled ( bool checked )
```

---

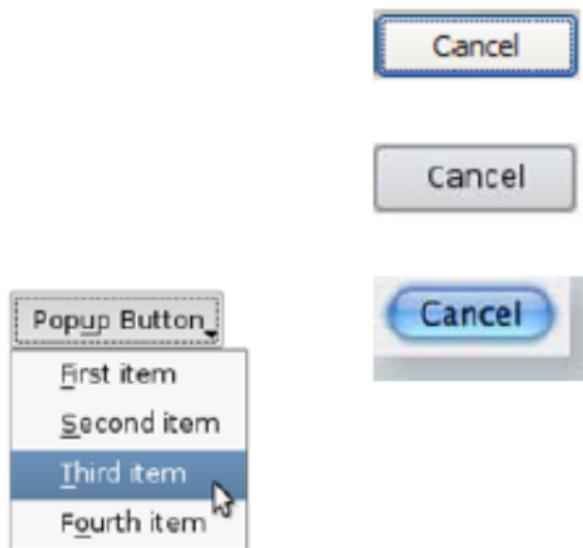
# QPushButton

---

```
QPushButton ( const QString & text , QWidget *  
            parent = 0 )  
QPushButton ( const QIcon & icon , const QString &  
            text , QWidget * parent = 0 )  
bool isDefault () const  
void setDefault ( bool )  
bool autoDefault () const  
void setAutoDefault ( bool )  
bool isFlat () const  
void setFlat ( bool )  
QMenu * menu () const  
void setMenu ( QMenu * menu )
```

---

# QPushButton



# QCheckBox

---

```
QCheckBox ( const QString & text , QWidget * parent  
          = 0 )  
Qt::CheckState checkState () const  
void setCheckState ( Qt::CheckState state )  
bool isTristate () const  
void setTristate ( bool y = true )  
  
enum Qt::CheckState  
Qt::Unchecked  
Qt::PartiallyChecked  
Qt::Checked
```

---

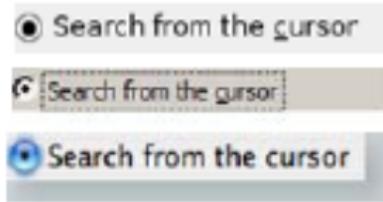
Сигналы:

---

```
void stateChanged ( int state )
```

---

# QCheckBox & QRadioButton



# QRadioButton

Свойство autoExclusive устанавливается в true.

---

```
QRadioButton ( const QString & text , QWidget *  
parent = 0 )
```

---

# QToolButton



---

```
bool autoRaise () const
void setAutoRaise ( bool enable )
QMenu * menu () const
void setMenu ( QMenu * menu )
QAction * defaultAction () const
setDefaultAction ( QAction * action )
Qt::ToolButtonStyle toolButtonStyle () const
void setToolButtonStyle ( Qt::ToolButtonStyle style
)
```

---

Сигналы:

---

```
void triggered ( QAction * action )
```

---

Слоты:

---

```
void setDefaultAction ( QAction * action )
void setToolButtonStyle ( Qt::ToolButtonStyle style )
```

---

# QButtonGroup

---

```
bool exclusive () const
void setExclusive ( bool )
QList<QAbstractButton *> buttons () const
void addButton ( QAbstractButton * button )
QAbstractButton * checkedButton () const
void removeButton ( QAbstractButton * button )
void addButton ( QAbstractButton * button, int id )
QAbstractButton * button ( int id ) const
int checkedId () const
int id ( QAbstractButton * button ) const
void setId ( QAbstractButton * button, int id )
```

---

Сигналы:

---

```
void buttonClicked ( QAbstractButton * button )
void buttonClicked ( int id )
void buttonPressed ( QAbstractButton * button )
void buttonPressed ( int id )
void buttonReleased ( QAbstractButton * button )
void buttonReleased ( int id )
```

# QGroupBox

---

```
QGroupBox ( const QString & title , QWidget * parent  
          = 0 )  
QString title () const  
void setTitle ( const QString & title )  
Qt::Alignment alignment () const  
void setAlignment ( int alignment )  
bool isCheckable () const  
void setCheckable ( bool checkable )  
bool isChecked () const  
bool isFlat () const  
void setFlat ( bool flat )
```

---

Сигналы:

---

```
void clicked ( bool checked = false )  
void toggled ( bool on )
```

---

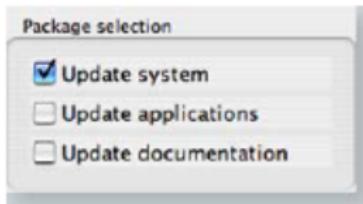
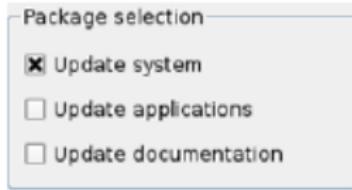
Слоты:

---

```
void setChecked ( bool checked )
```

---

# QGroupBox



# QSpinBox



Слоты:

---

```
void setValue ( int val )
```

---

Сигналы:

---

```
void valueChanged ( int i )
void valueChanged ( const QString & text )
```

---

# QSpinBox

---

```
int minimum () const
void setMinimum ( int min )
int maximum () const
void setMaximum ( int max )
int singleStep () const
void setSingleStep ( int val )
void setRange ( int minimum, int maximum )
QString prefix () const
void setPrefix ( const QString & prefix )
QString suffix () const
void setSuffix ( const QString & suffix )
int value () const
QString cleanText () const
```

---

# QSlider

---

```
QSlider ( Qt::Orientation orientation , QWidget *  
          parent = 0 )  
int tickInterval () const  
void setTickInterval ( int ti )  
TickPosition tickPosition () const  
void setTickPosition ( TickPosition position )
```

---



---

```
int notchSize () const
qreal notchTarget () const
bool notchesVisible () const
void setNotchTarget ( double target )
bool wrapping () const
```

---

# QDial



```
bool hasFrame () const
void setFrame ( bool )
Qt::Alignment alignment () const
void setAlignment ( Qt::Alignment flag )
int maxLength () const
void setMaxLength ( int )
QString text () const
void setText ( const QString & )
int cursorPosition () const
void setCursorPosition ( int )
EchoMode echoMode () const
void setEchoMode ( EchoMode )
bool hasAcceptableInput () const
bool hasSelectedText () const
QString displayText () const
QString selectedText () const
bool isUndoAvailable () const
bool isRedoAvailable () const
```

```
const QValidator * validator () const
void setValidator ( const QValidator * v )
void cursorBackward ( bool mark, int steps = 1 )
void cursorForward ( bool mark, int steps = 1 )
int cursorPosition () const
int cursorPositionAt ( const QPoint & pos )
void cursorWordBackward ( bool mark )
void cursorWordForward ( bool mark )
void backspace ()
void del ()
void deselect ()
void end ( bool mark )
void home ( bool mark )
void insert ( const QString & newText )
```

---

# QLineEdit

## Слоты:

---

```
void clear ()  
void copy () const  
void cut ()  
void paste ()  
void redo ()  
void selectAll ()  
void setText ( const QString & )  
void undo ()
```

---

## Сигналы:

---

```
void cursorPositionChanged ( int old, int new )  
void editingFinished ()  
void returnPressed ()  
void selectionChanged ()  
void textChanged ( const QString & text )  
void textEdited ( const QString & text )
```

---

# Технология ООП

Санкт-Петербургский государственный политехнический университет

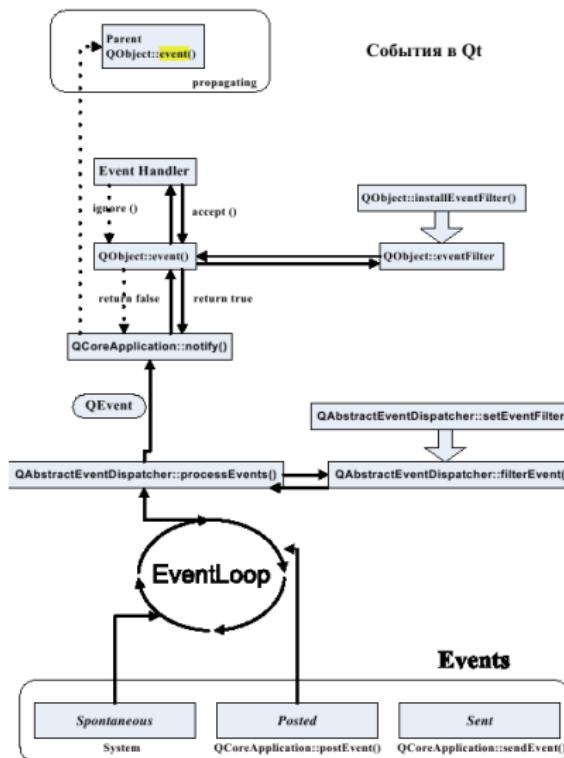
22 ноября 2011

# События в Qt

В Qt событие — это экземпляр класса QEvent. В Qt редко приходится использовать события напрямую, т.к. большинство виджетов сами генерируют сигналы в ответ на любое существенное событие.

# События в Qt

54



# События в Qt



# События в Qt

События клавиатуры обрабатываются путем переопределения функций keyPressEvent() и keyReleaseEvent().

---

```
void MyClass :: keyPressEvent(QKeyEvent *event)
{
    switch( event->key() )
    {
        case Qt::Key_Space:
            qDebug( "Space pressed" );
            break;
        ...
    default:
        QWidget::keyPressEvent( event );
    }
}
```

---

---

```
void accept ()
void ignore ()
bool isAccepted () const
void setAccepted ( bool accepted )
bool spontaneous () const
Type type () const
int registerEventType ( int hint = -1 ) [static]
```

---

---

```
void postEvent ( QObject * receiver , QEvent * event )
void processEvents ( QEventLoop::ProcessEventsFlags
                     flags = QEventLoop::AllEvents )
void removePostedEvents ( QObject * receiver )
bool sendEvent ( QObject * receiver , QEvent * event )
void sendPostedEvents ( QObject * receiver , int
                        event_type )
void sendPostedEvents ()
```

---

QObject:

---

```
virtual bool event ( QEvent * e )
virtual bool eventFilter ( QObject * watched ,
    QEvent * event )
void installEventFilter ( QObject * filterObj )
```

---

# Обработчики событий

---

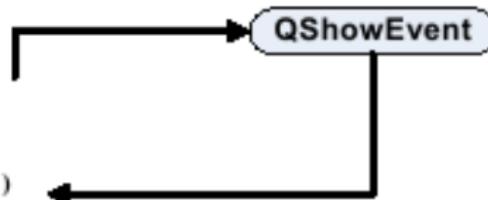
```
bool QObject::event(QEvent *event)
{
    switch (e->type()) {
        case QEvent::Timer:
            timerEvent((QTimerEvent*)e);
            break;
        case QEvent::ChildAdded:
        case QEvent::ChildInserted:
        case QEvent::ChildRemoved:
            childEvent((QChildEvent*)e);
            break;
        case QEvent::DeferredDelete:
            delete this;
            break;
    }
    return true;
}
```

# Обработчики событий

```
bool QWidget::event(QEvent *event)
{
    switch (e->type()) {
        case QEvent::KeyPress:
            keyPressEvent((QKeyEvent *)event);
            if (!((QKeyEvent *)event)->isAccepted())
                return false;
            break;
        case QEvent::KeyRelease:
            keyReleaseEvent((QKeyEvent *)event);
            if (!((QKeyEvent *)event)->isAccepted())
                return false;
            break;
        ...
    }
    return true;
}
```

# События в Qt

- `virtual void setVisible ( bool visible )`
- `void show () [slot]`



- `void showEvent ( QShowEvent * event ) [virtual protected]`

- `void hide () [slot]`
- `void :hideEvent ( QHideEvent * event ) [virtual protected]`

# События в Qt

- `bool close () [slot]` → **QCloseEvent**
- `void closeEvent ( QCloseEvent * event ) [virtual protected]` ← **QCloseEvent**

# Установка фильтров событий

```
void MyLineEdit::keyPressEvent(QKeyEvent *event)
{
    if (event->key() == Qt::Key_Space) { focusNextChild
        (); }
    else { QLineEdit::keyPressEvent(event); }
}
```

Настройка фильтров событий состоит из следующих этапов:

- Регистрация объекта-перехватчика с целевым объектом посредством вызова функции `installEventFilter()`.
- Обработка события в функции `eventFilter()` перехватчика

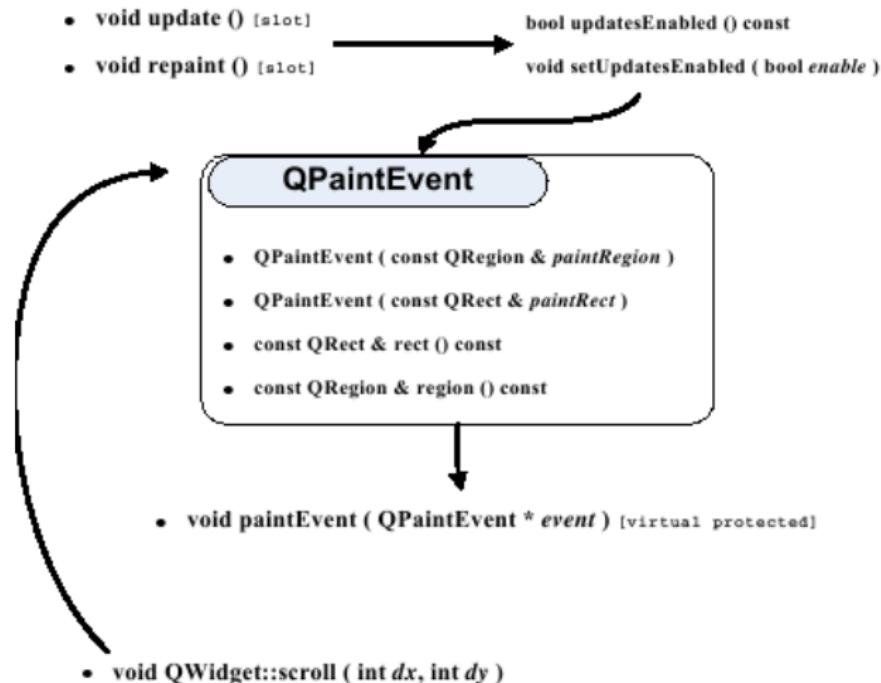
## Установка фильтров событий

---

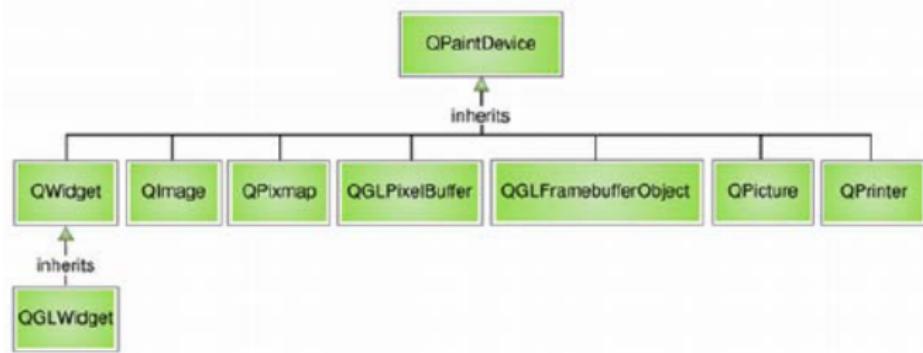
```
MyClass :: MyClass(QWidget *parent)
{
    firstNameEdit->installEventFilter(this);
    secondNameEdit->installEventFilter(this);
    cityEdit->installEventFilter(this);
}

bool MyClass::eventFilter(QObject *target, QEvent *
    event)
{
    if(target == firstNameEdit || target ==
        secondNameEdit || ... )
    {
        QKeyEvent *keyEvent = static_cast<QKeyEvent*>(event
            );
        if(keyEvent->key() == Qt::Key_Space) {
            focusNextChild(); return true; }
    }
    return QWidget::eventFilter(target, event);
}
```

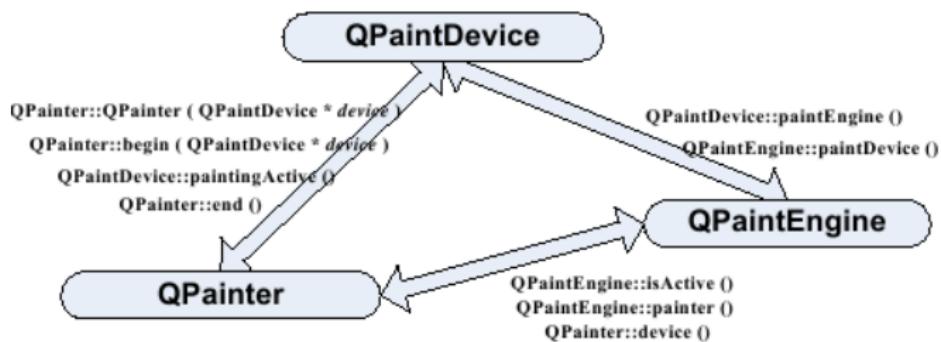
# События в Qt



# Вывод графики



# Вывод графики



## Вывод графики

QPainter может использоваться для вычерчивания на различных устройствах, таких как QWidget, QPixmap, QImage, QPainter и т.п., что позволяет использовать один программный код для отображения на экран, получения изображения и печати отчетов.

---

```
enum QPaintEngine::Type
    QPaintEngine::X11
    QPaintEngine::Windows
    QPaintEngine::MacPrinter
    QPaintEngine::CoreGraphics
    QPaintEngine::QuickDraw
    QPaintEngine::QWindowSystem
    QPaintEngine::PostScript
    QPaintEngine::OpenGL
    QPaintEngine::Picture
    QPaintEngine::SVG
    QPaintEngine::Raster
    QPaintEngine::Direct3D
    QPaintEngine::Pdf
    QPaintEngine::User
```

---

---

```
virtual void drawEllipse ( const QRect & rect )
virtual void drawLines ( const QLine * lines , int
    lineCount )
virtual void drawPath ( const QPainterPath & path )
virtual void drawImage ( const QRectF & rectangle ,
    const QImage & image , const QRectF & sr , Qt::
    ImageConversionFlags flags = Qt::AutoColor )
virtual void drawPoints ( const QPoint * points ,
    int pointCount )
virtual void drawPolygon ( const QPoint * points ,
    int pointCount , PolygonDrawMode mode )
```

---

---

```
int manhattanLength () const
int (qreal) & rx ()
(int | qreal) & ry ()
void setX ((int | qreal)x )
void setY ( int y )
int x () const
int y () const
```

---

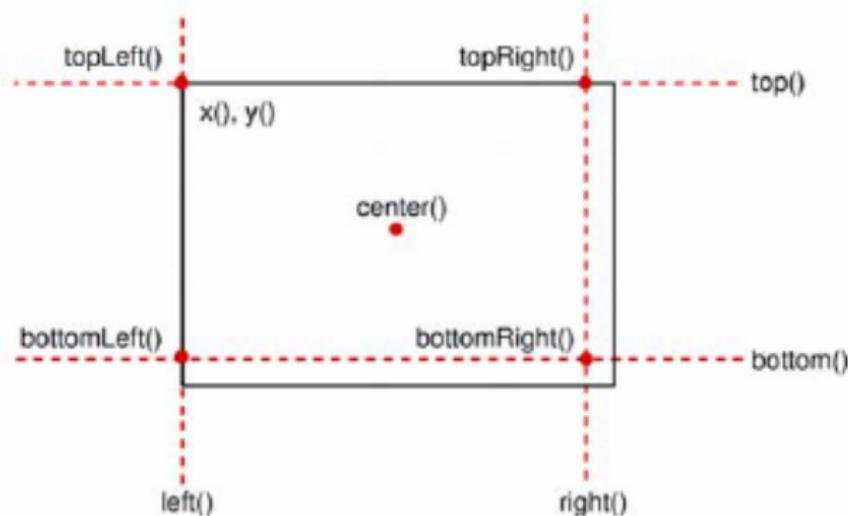
# QLine(F)

---

```
void setP1 ( const QPoint & p1 )
void setP2 ( const QPoint & p2 )
void setLine ( int x1, int y1, int x2, int y2 )
void setPoints ( const QPoint & p1, const QPoint &
    p2)
void translate ( const QPoint & offset )
QLine translated ( const QPoint & offset ) const
qreal angle () const (F)
qreal angleTo ( const QLineF & line ) const (F)
qreal length () const (F)
QLine toLine () const (F)
```

---

# QRect(F)

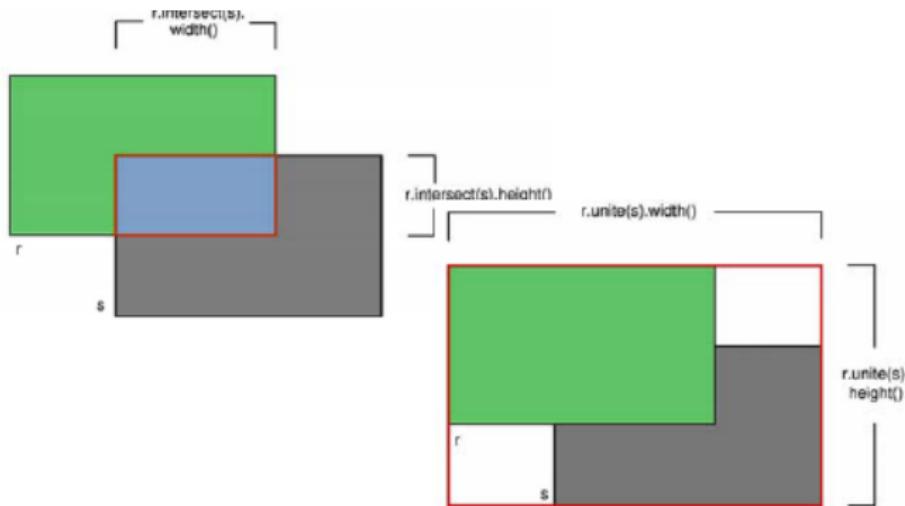


---

```
void adjust ( int dx1, int dy1, int dx2, int dy2 )
QRect adjusted ( int dx1, int dy1, int dx2, int dy2
    ) const
bool contains ( const QPoint & point, bool proper =
    false ) const
QRect intersected ( const QRect & rectangle ) const
bool intersects ( const QRect & rectangle ) const
QRect united ( const QRect & rectangle ) const
```

---

# QRect(F)



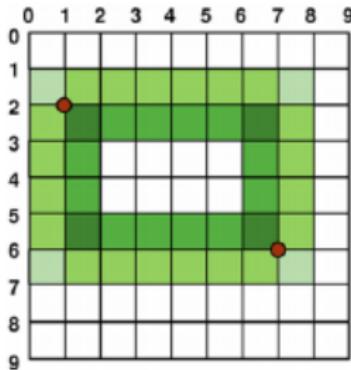
# QPolygon(F)

---

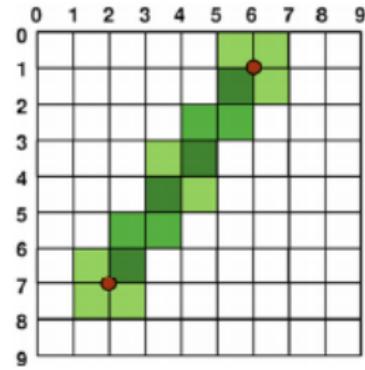
```
QRect united ( const QRect & rectangle ) const
QRectF boundingRect () const
bool containsPoint ( const QPointF & point , Qt::
    FillRule fillRule ) const
QPolygonF intersected ( const QPolygonF & r ) const
bool isClosed () const
QPolygonF subtracted ( const QPolygonF & r ) const
QPolygon toPolygon () const (F)
void translate ( const QPointF & offset )
void translate ( qreal dx, qreal dy )
QPolygonF united ( const QPolygonF & r ) const
```

---

# Рендеринг



```
QPainter painter(this);
painter.setRenderHint( QPainter::Antialiasing);
painter.setPen(Qt::darkGreen);
painter.drawRect(1, 2, 6, 4);
```



```
QPainter painter(this);
painter.setRenderHint(QPainter::Antialiasing);
painter.setPen(Qt::darkGreen);
painter.drawLine(2, 7, 6, 1);
```

# QPainter::RenderHint

---

```
enum QPainter::RenderHint
flags QPainter::RenderHints
    QPainter::Antialiasing
    QPainter::TextAntialiasing
    QPainter::SmoothPixmapTransform
    QPainter::HighQualityAntialiasing

RenderHints renderHints () const
void setRenderHint ( RenderHint hint , bool on =
    true )
void setRenderHints ( RenderHints hints , bool on =
    true )
```

---

# QPen



Qt::SolidLine



Qt::DashLine



Qt::DotLine



Qt::DashDotLine



Qt::DashDotDotLine



Qt::CustomDashLine



Qt::SquareCap



Qt::FlatCap



Qt::RoundCap



Qt::BevelJoin



Qt::MiterJoin



Qt::RoundJoin

---

```
void setColor ( const QColor & color )
void setStyle ( Qt::PenStyle style )
void setWidth ( int width )
void setWidthF ( qreal width )
Qt::PenStyle style () const
int width () const
qreal widthF () const
void setBrush ( const QBrush & brush )
void setCapStyle ( Qt::PenCapStyle style )
void setJoinStyle ( Qt::PenJoinStyle style )
```

---

---

```
QColor ( int r, int g, int b, int a = 255 )
QColor ( QRgb color )
QColor ( const QString & name )
QColor ( const char * name )
QColor ( const QColor & color )
QColor ( Qt::GlobalColor color )
void setCmyk ( int c, int m, int y, int k, int a =
    255 )
void setHsv ( int h, int s, int v, int a = 255 )
void setRgb ( int r, int g, int b, int a = 255 )
```

---

---

**enum**

```
Qt::GlobalColor Qt::white  
Qt::black Qt::red  
Qt::darkRed Qt::green  
Qt::darkGreen Qt::blue  
Qt::darkBlue Qt::cyan  
Qt::darkCyan Qt::magenta  
Qt::darkMagenta Qt::yellow  
Qt::darkYellow Qt::gray  
Qt::darkGray Qt::lightGray
```

---

# QBrush

---

```
QBrush ( Qt::GlobalColor color, Qt::BrushStyle
          style = Qt::SolidPattern )
const QColor & color () const
const QGradient * gradient () const
void setColor ( const QColor & color )
void setColor ( Qt::GlobalColor color )
void setStyle ( Qt::BrushStyle style )
void setTexture ( const QPixmap & pixmap )
void setTextureImage ( const QImage & image )
Qt::BrushStyle style () const
QPixmap texture () const
QImage textureImage () const
```

---

---

```
QPainter ( QPaintDevice * device )
bool begin ( QPaintDevice * device )
bool end ()
QPaintEngine * paintEngine () const
QPaintDevice * device () const
const QBrush & background () const
Qt::BGMode backgroundMode () const
const QBrush & brush () const
const QPen & pen () const
void setBackground ( const QBrush & brush )
void setBackgroundMode ( Qt::BGMode mode )
void setBrush ( const QBrush & brush )
void setBrush ( Qt::BrushStyle style )
void setPen ( const QPen & pen )
void setPen ( const QColor & color )
void setPen ( Qt::PenStyle style )
```

---

# QPainter

---

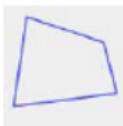
```
void drawPoint ( const QPointF & position )
void drawLine ( const QLineF & line )
void drawLines ( const QLineF * lines , int
    lineCount )
void drawRect ( const QRectF & rectangle )
void drawRects ( const QRectF * rectangles , int
    rectCount )
void fillRect ( const QRectF & rectangle , const
    QBrush & brush )
void eraseRect ( const QRectF & rectangle )
```

---

# QPainter

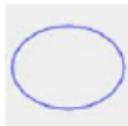


```
void drawRoundedRect ( const QRectF & rect, qreal xRadius,  
qreal yRadius, Qt::SizeMode mode = Qt::AbsoluteSize )
```



```
void drawPolygon ( const QPolygonF & points, Qt::FillRule fillRule = Qt::OddEvenFill  
  
void QPainter::drawPolyline ( const QPointF * points, int pointCount )
```

# QPainter



```
void drawEllipse ( const QRectF & rectangle )
```



```
void drawPie ( const QRectF & rectangle, int startAngle, int spanAngle )
```

# QPainter



```
void drawArc ( const QRectF & rectangle, int startAngle, int spanAngle )
```



```
void drawChord ( const QRectF & rectangle, int startAngle, int spanAngle )
```

# QPainter

---

```
void MyWidget::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing, true)
        ;
    painter.setPen(QPen(Qt::black, 12, Qt::DashDotLine,
        Qt::RoundCap));
    painter.setBrush(QBrush(Qt::green, Qt::SolidPattern
        ));
    painter.drawEllipse(80, 80, 400, 240);
    QPointF points[4] = { QPointF(10.0, 80.0), QPointF
        (20.0, 10.0), QPointF(80.0, 30.0), QPointF
        (90.0, 70.0) };
    painter.drawPolygon(points, 4);
    QRectF rectangle(10.0, 20.0, 80.0, 60.0);
    painter.drawRect(rectangle);
    painter.drawText(100, 100, "Hello Qt");
    painter.drawImage(300, 100, QImage(":/image.png"));
    painter.drawImage(80, 80, 400, 240, 60 * 16, 270 * 16)
```

# Технология ООП

Санкт-Петербургский государственный политехнический университет

29 ноября 2011

Qt обеспечивает интерфейс для общения с базами данных независящий от платформы и конкретной реализации базы данных.

Для использования баз данных, необходимо подключить модуль QSql.

Добавить в проектный файл: QT += sql

Для использования классов этого модуля:

---

```
#include <QtSql>
```

---

Классы модуля QSql разделены на 3 уровня:

- уровень драйверов
- программный уровень
- уровень пользовательского интерфейса

К уровню драйверов относятся классы для получения данных на физическом уровне: QSqlDriver, QSqlDriverPlugin, QSqlResult.

Связь с базой данных обеспечивается объектом QSqlDatabase.  
Qt включает в себя следующие драйверы:

- QDB2 — DB2
- QIBASE — Interbase
- QMYSQL — MySQL
- QOCI — Oracle
- QODBC — ODBC драйвер для MS SQL Server, IBM DB2 и т.д.
- QPSQL — PostgreSQL
- QSQLITE — PostgreSQL
- QTDS — Sybase

# Уровень драйверов

Для выполнения запросов SQL необходимо сначала установить соединение с базой данных.

Обычно настройка выполняется отдельной функцией вызываемой при запуске приложения.

---

```
bool createConnection()
{
    QSqlDatabase *db = QSqlDatabase::addDatabase("QSQLITE");
    db->setDatabaseName("base.dat");
    if (!db->open()) { qDebug("Error open database");
        return false;
    }
}
```

---

При использовании серверов баз данных, необходимо также задать setHostName, setUsername, setPassword.

# Уровень драйверов

Задав имя соединения, при подключении к базе данных, можно вернуть на него потом указатель.

---

```
QSqlDatabase *db = QSqlDatabase::addDatabase("  
    QSQLITE", "MYDATABASE");  
  
...  
QSqlDatabase db1 = QSqlDatabase::database("  
    MYDATABASE");
```

---

# Программный уровень

Программный интерфейс для обращения к базе данных:  
QSqlDatabase, QSqlQuery, QSqlIndex, QSqlRecord.

# Программный уровень

После установки соединения можно применять класс QSqlQuery для выполнения любых запросов SQL поддерживаемых используемой базой данных.

---

```
QSqlQuery query;  
query.exec( "SELECT country , year FROM artist" )
```

---

После выполнения запроса, можно посмотреть результат:

---

```
while ( query . next () ) {  
    QString country = query . value ( 0 ) . toString ();  
    int year = query . value ( 1 ) ..toInt ();  
}
```

---

Функция next передвигает указатель на следующую возвращенную запись, когда записей не осталось, либо возвратилась ошибка, то возвращается false.

Функция value возвращает QVariant на ячейку.

Указание полей в шаблоне запроса при операциях вставки  
(используя имена):

---

```
QSqlQuery query;
query.prepare("INSERT INTO person(id , forename ,  
surname)  
VALUES (:id , :forename , :surname)");
query.bindValue(":id", 1001);
query.bindValue(":forename", "Bart");
query.bindValue(":surname", "Simpson");
query.exec();
```

---

Указание полей в шаблоне запроса при операциях вставки  
(используя позиции):

---

```
QSqlQuery query;
query.prepare("INSERT INTO person(id , forename ,  
surname)  
VALUES (:id , :forename , :surname)");
query.bindValue(0, 1001);
query.bindValue(1, "Bart");
query.bindValue(2, "Simpson");
query.exec();
```

---

Указание полей в шаблоне запроса при операциях вставки  
(используя позиции, версия 2):

---

```
QSqlQuery query;
query.prepare("INSERT INTO person (id , forename ,  
surname) " +  
            "VALUES (?, ?, ?)");  
query.bindValue(0, 1001);  
query.bindValue(1, "Bart");  
query.bindValue(2, "Simpson");  
query.exec();
```

---

Указание полей в шаблоне запроса при операциях вставки  
(используя позиции, версия 3):

---

```
QSqlQuery query;
query.prepare("INSERT INTO person (id , forename ,  
surname) " +  
            "VALUES (?, ?, ?)");  
query.addBindValue(1001);  
query.addBindValue("Bart");  
query.addBindValue("Simpson");  
query.exec();
```

---

Указание полей в шаблоне запроса при операциях вставки  
(средствами подстановки Qt):

---

```
QSqlQuery query;
QString str ("INSERT INTO person (id , forename ,  

    surname) VALUES (%1, %2, %3)" );
str .arg (1001);
str .arg ("Bart");
str .arg ("Simpson");
query .exec (str);
```

---

Передача значений в хранимые процедуры в базе данных:

---

```
QSqlQuery query;
query.prepare("CALL AsciiToInt(?,?)");
query.bindValue(0, "A");
query.bindValue(1, 0, QSql::Out);
query.exec();
int i = query.boundValue(1).toInt();
```

---

# Программный уровень

Qt поддерживает транзакции в тех базах где они предусмотрены, для активации транзакции используется функция `transaction()` класса `QSqlDatabase`.

---

```
QSqlDatabase :: database () . transaction () ;
QSqlQuery query (" select id from artist where name =
    'Gluecifer ' ) ;
if ( query . next () )
{
    int artistId = query . value ( 0 ) . toInt () ;
    query . exec (" insert into cd ( id , artistid , title ,
        year ) VALUES ( 201 , " + QString :: number ( artistId
        ) + " , 'Riding the Tiger' , 1997 ) " ) ;
}
QSqlDatabase :: database () . commit () ;
```

---

Закрепление транзакции осуществляется функцией `commit()`, а отмена — `rollback()`.

Поддерживает ли база данных транзакции можно проверить функцией `hasFeature`

Представляет собой модели для отображения результата запросов: QSqlQueryModel, QSqlTableModel, QSqlRelationTableModel.

# Концепция модель-представление

Концепция модель-представление позволяет предоставлять данные многократно без дублирования.

Возможность быстро адаптировать изменения под новую модель хранения данных

Удобство тестирования, позволяющие использовать разные тесты для любой модели реализующей заданный интерфейс.

# Концепция модель-представление

Модель — отвечает за управление данными и предоставляет интерфейс для их чтения и записи.

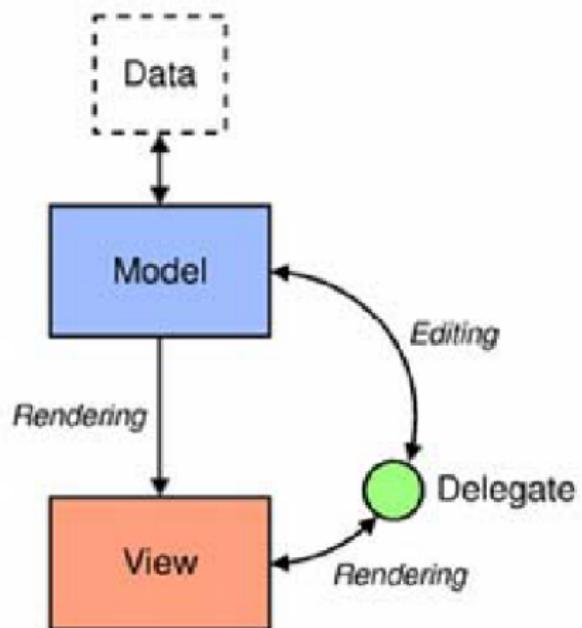
Представление — представление данных пользователю и их расположение.

Делегат — отвечает за рисование и редактирование каждого элемента.

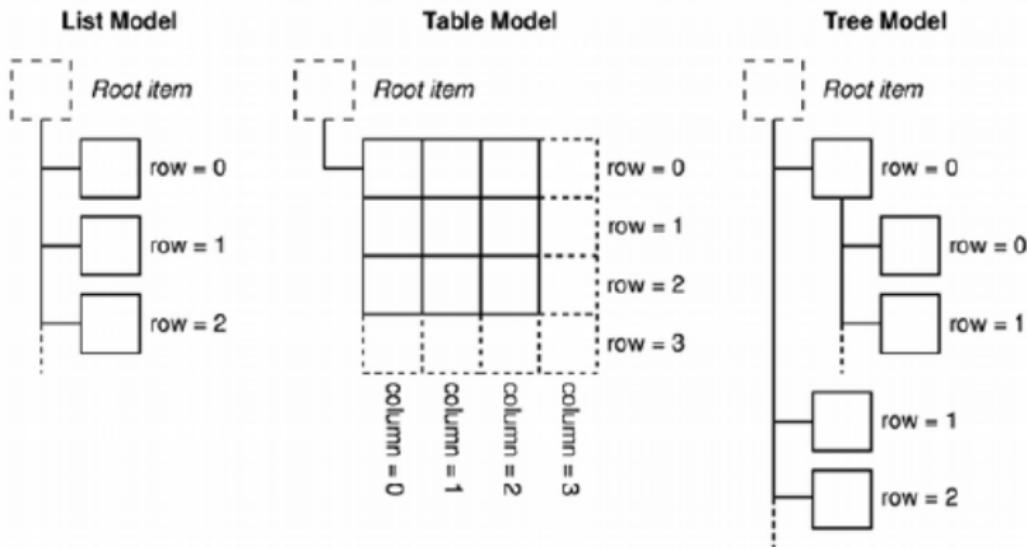
Возможность быстро адаптировать изменения под новую модель хранения данных

Удобство тестирования, позволяющие использовать разные тесты для любой модели реализующей заданный интерфейс.

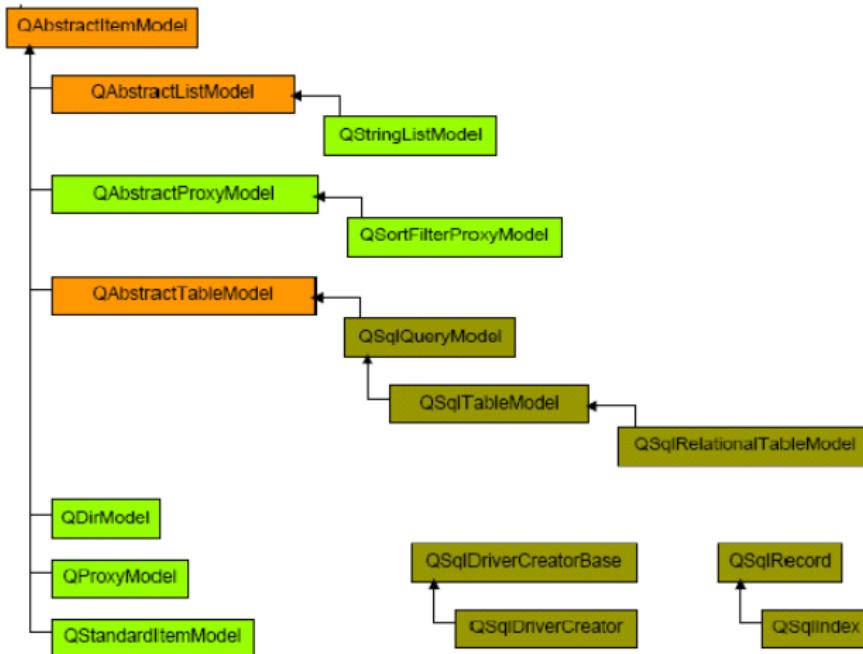
# Архитектура модель-представление



# Архитектура модель-представление



# Архитектура модель-представление



Пример использования модели для осуществления выборки данных:

---

```
QSqlTableModel model;  
model.setTable("cd");  
model.setFilter("year >= 1998");  
model.select()
```

---

Аналогично запросу:

---

```
select * from cd where year >= 1998;
```

---

Для просмотра результатов результирующей выборки используется функция record().

```
for( int i = 0; i < model.rowCount(); i++)
{
    QSqlRecord record = model.record(i);
    QString title = record.value("title").toString();
    int year = record.value("year").toInt();
    qDebug << title << " " << year;
}
```

QSqlRecord::value() может принимать имя поля, либо его индекс.

## Вставка данных в модель:

---

```
QSqlTableModel model;
model.setTable("cd");
model.insertRows(0, 1);
model.setData(model.index(0, 0), 113);
model.setData(model.index(0, 1), "Test");
model.setData(model.index(0, 2), 224);
model.setData(model.index(0, 3), 2004);
model.submitAll();
```

---

Для вставки используется функция `insertRow()`, для вставки пустой строки и функция `setData()` — для задания значений столбцов, принимает в качестве параметров указатель на ячейку (`QModelIndex`).

После вызова `submitAll()` добавленная запись может быть перемещена в другую позицию, в зависимости от упорядоченности таблицы. А также для записи изменений в базу данных.

## Обновление данных:

---

```
QSqlTableModel model;
model.setTable("cd");
model.setFilter("id = 12");
model.select();
if (model.rowCount() == 1)
{
    QSqlRecord record = model.record(0);
    record.setValue("title", "Test");
    record.setValue("year", record.value("year").toInt()
        () + 1);
    model.setRecord(0, record);
    model.submitAll();
}
```

---

## Удаление данных:

---

```
QSqlTableModel model;
model.setTable("cd");
model.setFilter("id = 12");
model.select();
if (model.rowCount() == 1)
{
    model.removeRows(0, 1);
    model.submitAll();
}
```

---

В `removeRows()` указывается номер строки с которой произвести удаление и число удаляемых строк.

Просмотр модели и ее связь с отображением для пользователя.  
Для связи модели с отображением используются представления  
QTableView и функция setModel.

---

```
model->setHeaderData(0, Qt::Horizontal, "Name");  
QTableView *view = new QTableView;  
view->setModel(model);  
view->setColumnHidden(0, true);  
view->resizeColumnsToContents();  
view->setEditTriggers(AbstractItemView::  
    AllEditTriggers)
```

---

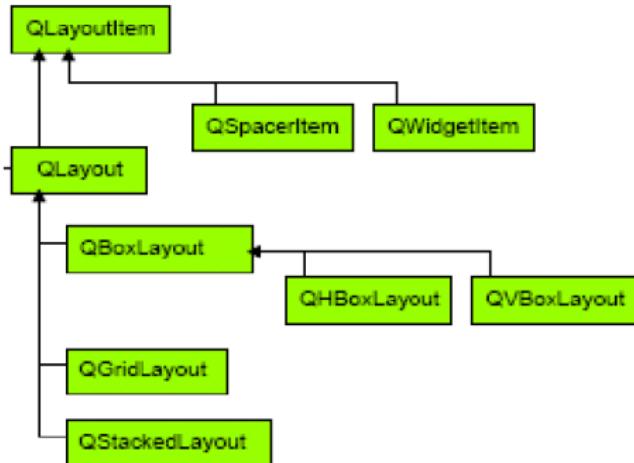
# Технология ООП

Санкт-Петербургский государственный политехнический университет

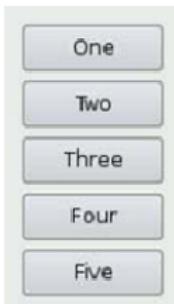
13 декабря 2011

Qt содержит несколько классов обеспечивающих компоновку виджетов на форме: QBoxLayout, QVBoxLayout, QHBoxLayout. Менеджеры компоновки позволяют виджетам автоматически адаптироваться к изменению размеров, формы, шрифтов.

# Менеджеры компоновки



# QHBoxLayout/QVBoxLayout



---

```
void addWidget ( QWidget * widget , int stretch = 0 ,
    Qt::Alignment alignment = 0 );
void addSpacing ( int size );
void addStretch ( int stretch = 0 );
void addLayout ( QLayout * layout , int stretch = 0
);
```

# QGridLayout



---

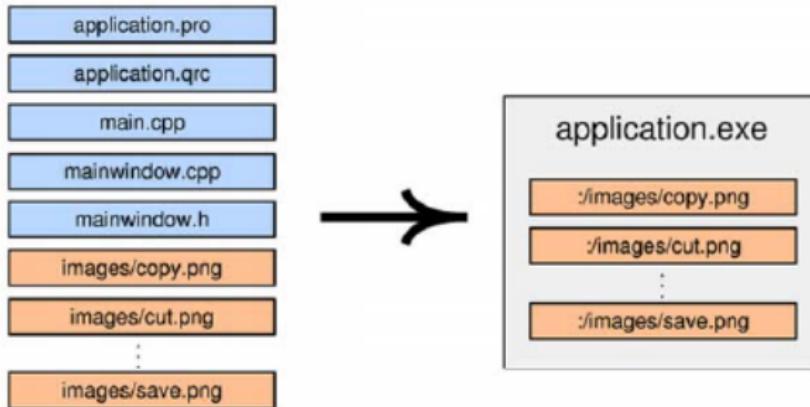
```
void addLayout ( QLayout * layout , int row , int
    column , Qt::Alignment alignment = 0 );
void addWidget ( QWidget * widget , int row , int
    column , Qt::Alignment alignment = 0 );
void setHorizontalSpacing ( int spacing );
void setVerticalSpacing ( int spacing );
```

---

# QGridLayout

Font	Font style	Size
Times	Roman	10
Times	Roman	10
Helvetica	Italic	12
Courier	Oblique	14
Palatino		16
Gill Sans		18

# Ресурсы



## \*.qrc files:

---

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
    <file>images/copy .png</file>
    <file>images/cut .png</file>
    <file>images/new .png</file>
    <file>images/open .png</file>
    <file>images/paste .png</file>
    <file>images/save .png</file>
</qresource>
</RCC>
```

---

\*.qrc files:

---

```
<qresource>
    <file>cut.jpg</file>
</qresource>
<qresource lang="fr">
    <file alias="cut.jpg">cut_fr.jpg</file>
</qresource>
```

---

Подключение ресурсов, в .pro файл добавить:

---

```
RESOURCES = application.qrc
```

---

Использование:

---

```
cutAct = new QAction(QIcon(":/images/cut.png"), tr("Cu&t"), this);
```

---

Qt обеспечивает широкую поддержку различных языков благодаря использованию Unicode в программных интерфейсах, это позволяет без проблем вводить текст на любых языках. Для перевода интерфейса приложения на другой язык используется система Qt Linguist.

## Source text

```
QObject::tr( text )
```

```
QCoreApplication::translate ( text )
```

```
QT_TR_NOOP ( sourceText )
```



```
lupdate [options] [project-file]
```

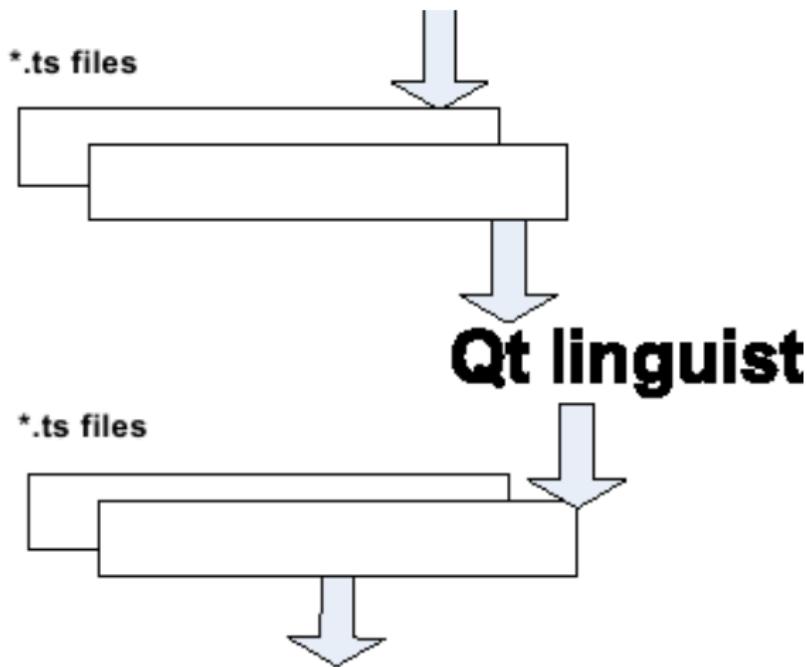
```
lupdate [options] [source-file|path]... -ts ts-files
```

\*.ts файлы необходимо прописать в проектный файл:

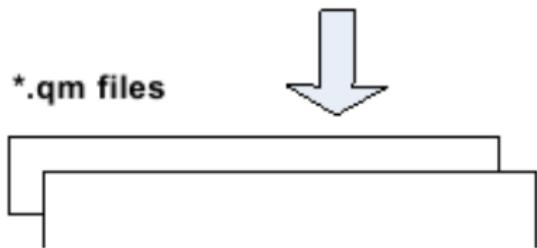
---

```
TRANSLATIONS = myapp_fi.ts myapp_ru.ts
```

---



```
lrelease [options] project-file  
lrelease [options] ts-files [-qm qm-file]
```



Для использования полученной интернационализации необходимо подключить транслятор в самом начале запуска приложения, где в метод load передается имя скомпилированного файла с передов, без расширения qm.

---

```
QTranslator myappTranslator;
myappTranslator.load("myapp_" + QLocale::system().name());
app.installTranslator(&myappTranslator);
```

---

Представляет собой программный интерфейс опирающийся на концепцию Model-View.

Содержит в себе класс `QGraphicsScene` являющийся моделью для графических элементов, которые наследуются от `QGraphicsItem`.

`QGraphicsView` — представление, использующееся для показа элементов модели.

Каждый элемент (`QGraphicsItem`) включает в себя поддержку событий мыши, клавиатуры и позволяет: перетаскивать, группировать и определять столкновения объектов.