Обобщенное программирование

templates

Обобщенное программирование

Уровни абстракции:

- Функции
- Классы
- Шаблоны

Замечание: универсальные указатели

- void*
- char*

Уровни абстракции. Функции

Выполняют одни и те же (низкоуровневые) действия над разными наборами данных.

Специфика:

- Программист задает типы параметров и возвращаемого значения
- Компилятор проверяет соответствие вызова и прототипа

Низкоуровневый код один и тот же!

Уровни абстракции. Классы

Специфика:

- Программист задает типы переменных и прототипы методов
- Компилятор для любого объекта резервирует одинаковый объем памяти
- **Низкоуровневый код методов один и тот же!**

Идеи обобщенного программирования

Хотелось бы иметь возможность:

- **Текст** на языке высокого уровня **один и тот же**
- А **низкоуровневый код** **разный** в зависимости от типов параметров функций и типов переменных класса

Параметризация

Шаблоны используются для

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

Главное достоинство – **type safe**, так как используемые типы становятся известными на этапе компиляции, поэтому **компилятор проверяет соответствие**

Зачем нужны шаблоны функций

Функция выполняет одни и те же действия с параметрами разного типа:

- текст на языке высокого одинаковый
- но низкоуровневый код будет зависеть от типов параметров.

```
int min(int x, int y){return (x<y) ? x : y;}
double min(double x, double y){return (x<y)?x:y;}
…
```

Зачем нужны шаблоны классов

Данные класса отличаются только типом, а реализация методов на языке высоко уровня выглядит одинаково

```
class Point{
        int x,y;
};
class Point{
        double x,y;
};
```

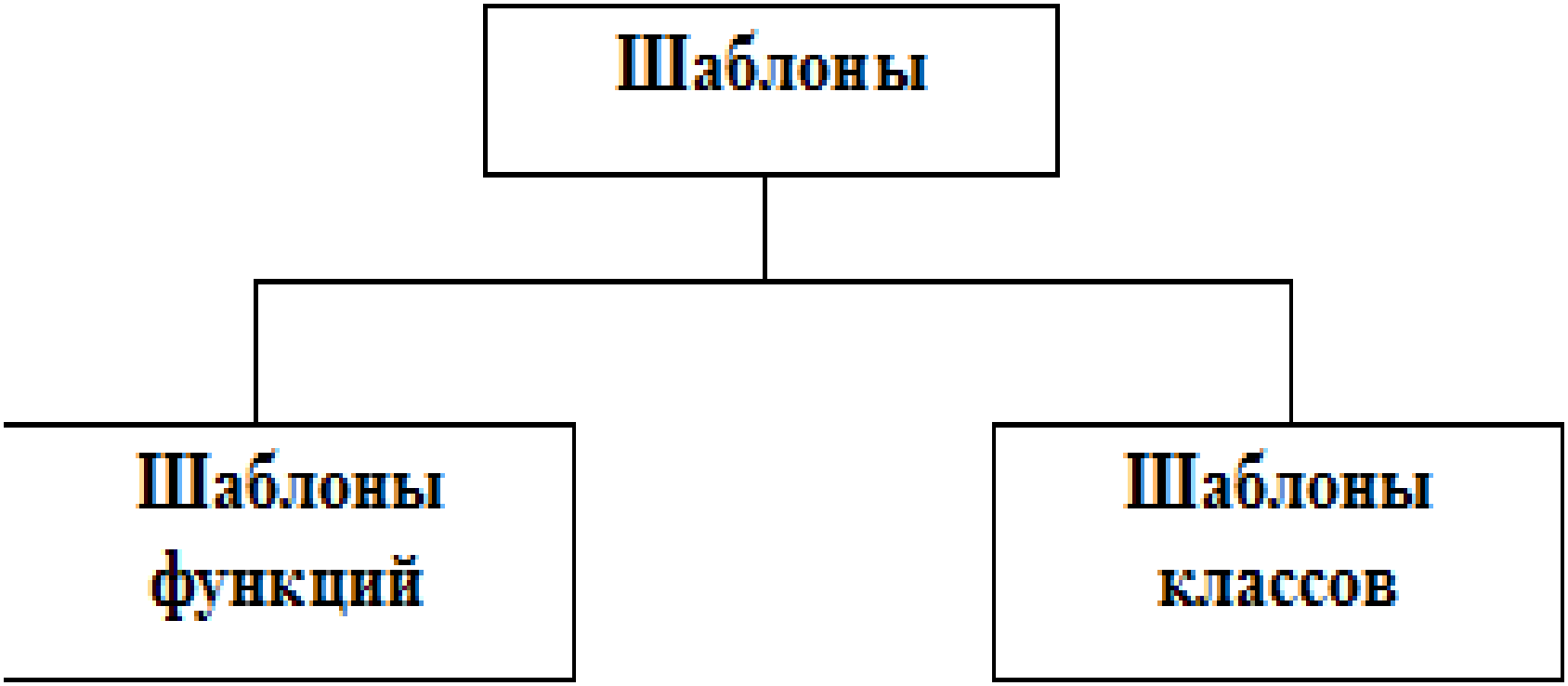
Зачем нужны шаблоны

```
T min(T x, T y){return (x<y) ? x : y;}


class Point{
        T x,y;
};
```

С помощью ключевого слова **template** на C++ можно задать компилятору **образец** кода для обобщенного типа данных

Виды шаблонов

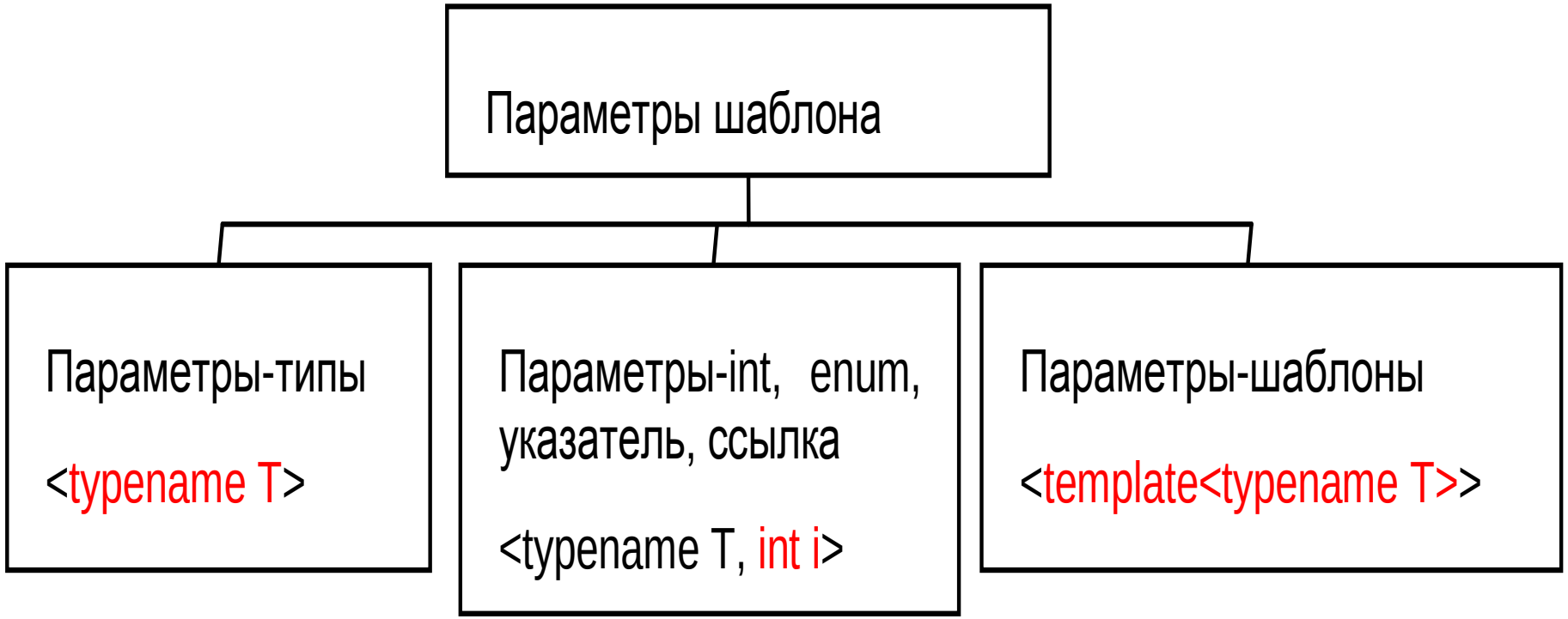


Объявление шаблона

template < список_параметров_шаблона **>**
объявление_функции_или_класса

Параметры шаблона указывают компилятору для каких понятий нужно генерировать низкоуровневый код по-разному

Параметры шаблона



Специфика параметров шаблона

- видя объявление шаблона, компилятор **не создает никакого кода** (без явного указания). Только встретив обращение к данному шаблону (вызов функции или создание экземпляра класса и вызов его методов) в тексте программы, компилятор сгенерирует соответствующий код.
- для обозначения обобщенного параметра рекомендуется использовать ключевое слово **typename** вместо class
- в качестве обобщенного типа можно задать как имя пользовательского типа данных, так и базового
- можно задать значения параметров шаблона по умолчанию:
  template< **typename A=int**>
  template< typename A, **typename B=A**>

Термины, связанные с шаблонами

- **Инстанцирование** - подстановка реальных типов в качестве параметров шаблона (это генерация кода функции или класса по шаблону для конкретных параметров). Различают:
  - неявное инстанцирование, которое происходит при вызове функции или создании объекта класса,
  - и явное инстанцирование с помощью резервированного слова template.

  Важно! инстанцирование можно делать только в точке программы, где доступна реализация шаблона функции или методов шаблонного класса
- **Специализация** - версия шаблона для конкретного набора параметров

Шаблоны функций

Способы обобщения функций, выполняющих одинаковые действия, но оперирующих данными разных типов???

Перегрузка имен функций

```
int min( int a, int b ) {return ( a < b ) ? a : b;}
double min( double a, double b )
 {return ( a < b ) ? a : b;}

int main(){
        int iX=1, iY=-1;
        int iResult = min(iX, iY);
        double dX=1.0, dY = 3;
        double dResult = min(dX, dY);
}
```

Перегрузка имен функций

Недостатки:

Программист должен явно реализовать несколько функций, которые отличаются только типами параметров.

Макросы с параметрами

```
#define min(a,b) ( a < b ) ? a : b
int main()
{
        int x=5, y=2;
        int r1=min(x, y) * 2; //???
        int r2= min(x++, y++); //???
}
```

Шаблон функции

- позволяет уменьшить количество «дублируемого» текста, определив только один шаблон, оперирующий с некоторым обобщенным типом данных
- лишен недостатков макроподстановок, так как тело функции по шаблону реализуется компилятором

Объявление шаблона функции

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

Специфика:

- как и обычную функцию, шаблон такой короткой функции Вы можете объявить встраиваемым:

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

- параметры шаблона (также как и параметры функции) могут иметь значения по умолчанию: template < typename T> void func(const **T& a); //???**
- тело шаблонной функции должно быть «видно» компилятору => обычно в **заголовочном файле**.
- функция-шаблон может наряду с параметрами обобщенного типа принимать параметры любого типа
- функция-шаблон может быть перегружена

Перегрузка шаблонной функции

- Функция «ищет» минимальный элемент массива (тип элемента массива - любой)

Инстанцирование и вызов

```
{
        int iX=1000, iY=500;
        int iResult = min(iX, iY);       //
        double dX=1.0, dY = 3;
        double dResult = min(dX, dY);     //
         int iResult2 = min(5, 10);
```

} => программист один раз пишет заготовку для обобщенного типа, а компилятор реализует тело для каждого использованного в программе типа

Инстанцирование и вызов

```
iResult = min(iX, dX);   //ошибка!
iResult = min<int>(iX, dX);
iResult = min<double>(iX, dX);
```

Шаблоны функций и объекты пользовательского типа

template <typename T> T min(T a, T b)

{return (a < b) ? a : b;}

Модификация шаблона:

- требуется обеспечить эффективную работу с пользовательскими типами
- сохранить возможность работы с базовыми типами

Шаблоны функций и объекты пользовательского типа

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

Шаблоны функций и объекты пользовательского типа

Модификация пользовательского типа данных:

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

Шаблоны функций и объекты пользовательского типа

Для пользовательского типа должны быть

перегружены все операторы,

используемые в теле функции

Шаблоны функций и объекты пользовательского типа

```
int main()
{
        Rect r1(1,1,3,3), r2(0,0,10,10);
        Rect res = min(r1,r2);

        MyString s1(“abc”), s2(“qwerty”);
        MyString resStr = min(s1,s2);
}
```

Специализация шаблонной функции и перегрузка имен

```
{
        int iX=1000, iY=500;
        int iResult = min(iX, iY); //OK
        Rect r1(1,2,3,4), r2(5,6,7,8);
        Rect rectResult = min(r1,r2);   //OK

        const char* strRes = min(“abc”, “qwerty”);//?

}
```

Перегрузка имен

1.h

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

const char* min(const char* str1, const char* str2);
```

Перегрузка имен

1.cpp

```
#include “1.h”
const char* min(const char* str1, const char* str2)
{
        return (strcmp(str1, str2)>0)? str2:str1;
}
```

Специализация шаблонной функции

1.h

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

Специализация шаблонной функции

1.cpp

```
#include ”1.h”
template <>  const char*&
 min<const char*>(const char*& a, const char*& b )
{
        return (strcmp(a, b)>0)? b:a;
}
```

Шаблоны классов

Шаблон класса предоставляет компилятору информацию:

- Какой объем памяти выделять под объект
- Как генерировать код методов класса (шаблоны методов)

Специфика шаблонных классов

- Могут участвовать в наследовании и перемежаться с нешаблонными классами
- Могут иметь виртуальные методы, но! Эти методы не могут быть в свою очередь шаблонами, базирующимися на других параметрах
- Могут иметь «друзей» (friend-функции и friend-классы)
- Могут иметь статические члены класса

Пример шаблонного класса

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

move???

Пример шаблонного класса

```
template <typename T, int size>  class  MyArray
{
        T ar[size];
 public:
        MyArray(){for(int i=0;i<size;i++) ar[i]=T();}
        T& operator[](int i){return ar[i];}
       const T& operator[](int i)const {return ar[i];}
};
```

Объявление и «реализация» метода

```
template <typename T, int size>  class  MyArray
{
        T ar[size];
 public:
        …
        T& operator[](int i);
};
template < typename T, int size> T& MyArray< T, size >
::      operator[](int i)
{
   if( i >= 0  && i<size) {return ar[i];}
        throw out_of_range(“Out of range”);
}
```

Создание объектов шаблонного класса и генерация методов

```
int main()
{
        MyArray< int, 5 > a1; //
        int n;
        std::cin>>n;
    try{
        ar1[n] = 1;                          //
        std::cout<<ar1[n];
       }catch(std::out_of_range& er){…}

        MyArray< Rect, 10 > a2; //
        MyArray< MyString, 20 > a3; //
}
```

Замечания

- Целочисленные параметры шаблона должны быть заданы константами

```
int N = 5;
// MyArray< int, N > ar;        //ошибка
```

Замечания

- методы класса являются шаблонами функций => должны быть «реализованы» тоже в заголовочном файле

Параметры шаблона могут иметь значения по умолчанию

```
template<typename T, int size=10> class MyArray
                {…};


int main()
{
    MyArray<int> ar5; //
}
```

Параметры шаблона могут иметь значения по умолчанию

```
template<typename T=int, int size=10>
                class MyArray{…};


int main()
{
    MyArray<> ar6; //
}
```

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(const 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] = 5.3;  //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 <typename T, int size>  class  MyArray
{
   T m_ar[size];
public:
    …
    template<typename Other> void copy(const Other* p,
                                                    size_t num)
    {
        int n = (num<size) ? num : size;
        for(int i = 0; i<n; i++) { m_ar[i] = p[i];}
    }
};
```

Реализация шаблонного метода вне класса

```
template <typename T, int size>  class  MyArray
{
        …
        template<typename Other>
                void copy(const Other* p, size_t num);
};
template <typename T, int size>
template <typename Other> void
MyArray<T, size>::copy(const Other* p, size_t num)
{…}
```

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

friend функция шаблонного класса

```
MyArray<int,10> a;
std::cout<<a; //???
```

friend функция шаблонного класса

```
template <typename T, int size>  class  MyArray
{
   T ar[size];
public:
    template<typename T, int s> friend std::ostream& operator<<
                    ( std::ostream& os, const MyArray<T,s>& a)
    {
          for (int i=0; i<s; i++) {os<<a.ar[i]<<'  ';}
          return os;
    }
};
```

friend класс шаблонного класса

```
template<typename K, typename V> class Pair{
        K key; V val;
        …
        template<typename K, typename V>
        friend class Arr;
};
template<typename K, typename V> class Arr{
        Pair<K,V>* p;
         size_t m_n, m_cap;
        …
};
```

static

A.h

```
template<typename T> class A{
        T m_t;
        static size_t count;
public:
        A(const T& t):m_t(t){}//как задать значение по умолчанию???
        ???
        static size_t GetCount(){return count;}
};
template<typename T> size_t A<T>::count;
```

static

```
#include “A.h”
int main()
{
        std::cout<< A<double>:: count; //???
        size_t num = A<double>::GetCount(); //???
        A<double> a1(1.1);
        num =a1. GetCount(); //???
        A<int> a2(33);
        num =a2. GetCount(); //???
}
```

typename

```
//.h
class A{
public:
    class B{
    …
    };
};
template<typename T> class C{
typename T::B* p;
};

//.cpp
C<A> ccc;
```

Специализация шаблонного класса

- **template<>** class MyArray<int,10>;
- **template** class MyArray<int,10>;

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

C++11 – внешние шаблоны

(уменьшение затрат компилятора и компоновщика)

Две единицы компиляции требуют инстанцировать один и тот же шаблон с одним и тем же типом параметра?

- в стандартном C++ компилятор должен инстанцировать шаблон для каждой единицы компиляции!!!
- C++11 вводит возможность **запретить** компилятору инстанцировать шаблон **в данной единице компиляции**:

```
extern template class MyArray<int,10>;
```

Пример

1.cpp

```
#include "MyArray.h"
 // предотвращаем неявное инстанцирование - MyArray<int,10> будет явно инстанцирован где-то в другом месте
extern template class MyArray<int,10>;
void f(MyArray<int,10>& v) { // использование }
```

2.cpp

```
#include "MyArray.h"
//делаем MyArray доступным клиентам – просим реализовать все методы
template class MyArray<int,10>;
```

vector, list, deque

КОНТЕЙНЕРЫ

Полубенцева М.

Понятие контейнера

Определение согласно стандарту:

Контейнер – это объект, который **хранит** другие объекты и **контролирует размещение** (allocation и deallocation) этих объектов посредством конструкторов, деструкторов и методов вставки/удаления (insert()/erase())

Основное назначение контейнеров:

- реализуют посредством шаблонов классов основные структуры хранения данных
- => позволяют «не изобретать велосипед», а пользоваться разработанными профессионалами и отлаженными классами для хранения как базовых, так и любых пользовательских типов

Специфика контейнеров:

- Внутренняя организация контейнеров **разная**!
- А хочется:
  - **отделить** программиста от внутренней реализации
  - предоставить программисту **универсальный** интерфейс для манипулирования любым контейнером (смысл и результат операций одинаков, реализация разная)
- Следствие – легкая **взаимозаменяемость** контейнеров

VECTOR

Эмуляция контейнера vector

Проблемы работы с массивами:

- Тип элемента
- Компилятор не контролирует выход индекса за пределы массива
- Изменение размера:
  - Для встроенных невозможно
  - Для динамических требует явного участия программиста
- При необходимости замены массива на другую структуру данных необходимо переписывать весь текст

«Нулевое приближение»

```
template<typename T> class MyVector{
        T* m_p;
        size_t m_n;
        size_t m_cap;
        …
};
```

default - конструктор

```
//MyVector.h
template <typename T> class MyVector{
        …
        MyVector()
        {
                ???
        }
…
};

//клиент.cpp
#include “MyVector.h”
int main()
{
        MyVector<int> v;
}
```

Конструктор с параметрами

```
//MyVector.h
template <typename T> class MyVector{
        …
        MyVector(size_t n)
        {
                ???
        }
};

//клиент.cpp
#include “MyVector.h”
int main()
{
        MyVector<Rect> v1(10);
        MyVector<double> v2(10);
}
```

Реализация

```
MyVector(size_t n)
{
        m_cap= m_n = n;
        m_p = static_cast<T*>(malloc(n*sizeof(T));
        for(size_t i=0; i<n; i++)
        {
                new(&m_p[i]) T();
        }
}
```

Конструктор с параметрами

```
//MyVector.h
template <typename T> class MyVector{
        …
        MyVector(size_t n, ???)
        {
                ???
        }
};

//клиент.cpp
#include “MyVector.h”
int main()
{
        MyVector<Rect> v1(10, Rect(1, 1, 2, 2));
        MyVector<double> v2(10, 33.22);
}
```

Реализация

```
MyVector(size_t n, const T& t)
{
        m_cap= m_n = n;
        m_p = static_cast<T*>(malloc(n*sizeof(T));
        for(size_t i=0; i<n; i++)
        {
                new(&m_p[i]) T(t);
        }

}
```

Конструктор копирования

```
//MyVector.h
template <typename T> class MyVector{
        …
        MyVector(const MyVector& r)      //memcpy???
        {
                ???
        }
};

//клиент.cpp
#include “MyVector.h”
int main()
{
        MyVector<Rect> v1(10, Rect(1, 1, 2, 2));
        MyVector<Rect> v2=v1;
        MyVector<double> v3 = v1; //???
}
```

Деструктор

```
//MyVector.h
template <typename T> class MyVector{
        …
        ~MyVector()
        {
                ???
        }
};
```

Реализация деструктора

```
~MyVector()
{
        for(size_t i=0; i<m_n; i++)
        {
                //delete(&m_p[i], &m_p[i]);
                m_p[i].~T();
        }
        free(m_p);
}
```

Очистка вектора

```
//MyVector.h
template <typename T> class MyVector{
        …
        void clear()
        {
                ???
        }
};
```

Размер

```
//MyVector.h
template <typename T> class MyVector{
        …
        size_t size()
        {
                ???
        }
};
```

Доступ к «крайним элементам»

```
//клиент.cpp
#include “MyVector.h”
int main()
{
        MyVector<int> v(10,1);
        std::cout<<v.front();  //реализация???
        v.front() = 22;
        std::cout<<v.back();  //реализация???
        v.back() = 33;
}
```

Доступ к «крайним элементам»

```
template <typename T> class MyVector{
        …
        T& front(){
                if(m_n) {return *m_p;}
                throw std::length_error(“Error”); //в стандартной библиотеке – в этом случае unpredicted behaviour }

        T& back(){
                if(m_n) return *(m_p+m_n-1);
                throw std::length_error();}
        }
};
```

operator[], at()

```
//клиент.cpp
#include “MyVector.h”
int main()
{
        MyVector<int> v(10);
        for(int i=0; i<v.size(); i++)
        { v[i] = i;}
        int n;
        std::cin>>n;
        try{
        v.at(n) = 33;
        }catch(std::out_of_range& er){…}
}
```

operator[], at()

```
//MyVector.h
template <typename T> class MyVector{
        …
        T& operator[](int i){return m_p[i];}
        const T& operator[](int i)const{return m_p[i];}

        T& at(int i){   if(i>=0 && i<m_n)return m_p[i];
                                throw out_of_range("…");}
        const T& at(int i) const {
                if(i>=0 && i<m_n)return m_p[i];
                                throw out_of_range("…");}
};
```

Добавить элемент в конец массива

```
//клиент.cpp
#include “MyVector.h”
int main()
{
        MyVector<int> v;
        v.push_back(1); // реализация???

}
```

push_back() неэффективный

```
//MyVector.h
template <typename T> class MyVector{
  void push_back(const T& el)
  {
        if(m_n==m_cap)
        {
              T* p= static_cast<T*>(malloc(m_cap*sizeof(T));
              for(int i=0; i<m_n; i++) {p[i] = std::move(m_p[i]);}
              free(m_p);
              m_p = p;
        }
        m_p[m_n] = el; //добавили новый элемент
        m_n++;
}
```

push_back() эффективный

```
//MyVector.h
template <typename T> class MyVector{
 template<typename A> void push_back(A&& t)
  {
            if(m_n==m_cap)
            {
                        ++m_cap;
                        T* p = new T[m_cap];
                        for(int i=0; i<m_n; i++) {p[i] = std::move(m_p[i]);}//memcpy???
                        free(m_p);
                        m_p = p;
            }
            m_p[m_n] = forward<A>(el); //добавили новый элемент
            m_n++;
}
```

Удалить последний элемент

???

```
//клиент.cpp
#include “MyVector.h”
int main()
{
        MyVector<int> v(10,1);
        v.pop_back();
}
```

pop_back()

```
//MyVector.h
template <typename T> class MyVector{
  void pop_back()
  {
                if(m_n)
                {
                        m_n--;
                        m_p[m_n].~T();
                }
}
```

Обмен двух векторов

???

```
//клиент.cpp
#include “MyVector.h”
int main()
{
        MyVector<int> v1(10,1);
        MyVector<int> v2(20,2);

        v1.swap(v2);
}
```

swap()

```
//MyVector.h
template <typename T> class MyVector{
        …
         void swap(MyVector& v)
        {
                int n = m_n;
                m_n = v.m_n;
                v.m_n = n;

                //m_cap – по аналогии

                T* p = m_p;
                m_p = v.m_p;
                v.m_p = p;
        }
};
```

Что еще необходимо определить?

???

Вводим понятие итератора

Для любой последовательности нужно обеспечить основные действия, которые независимо от вида последовательности должны выглядеть одинаково:

- получать значение элемента последовательности
- «перемещаться» от одного элемента последовательности к другому

Какие операторы подходят для реализации указанных действий**???**

Аналогия с указателем

Исходя из выполняемых итератором задач он «похож» на указатель =>

Какие **операторы** используются применительно к указателю?

Основные действия

Такие действия мог бы обеспечить вспомогательный класс, в котором перегружены операторы:

- разыменования *
- инкремент ++
- == !=
- ->

Реализация итератора

```
template<typename T> class MyVector{
        …
public:
        class iterator{ //встроенный класс генерируется по тому же параметру шаблона
                T* p;//на что в самом деле нужно указывать
        public:
                iterator(){p=0;}
                iterator(T* pT){p=pT;}
                T& operator*() {return *p;}
                iterator& operator++(){++p; return *this;}
                iterator operator++( int unused){return iterator(p++);}
                bool operator==(const iterator& r) const {return p==r.p;}
                bool operator!=(const iterator& r) const {return p!=r.p;}
                T* operator->(){return p;}
                };
        …
};
```

Методы для получения итераторов

```
template<typename T> class MyVector{

        …
public:
        class iterator{
        …
        };
//Методы для получения итераторов
        iterator begin(){???} //итератор на первый
        iterator end(){???} //итератор на конец последовательности
};
```

Методы для получения итераторов

```
template<typename T> class MyVector{

        …
public:
        class iterator{
        …
        };
//Методы для получения итераторов
        iterator begin(){return iterator(m_p);}
        iterator end(){return iterator(m_p+m_n);}
};
```

Пример: распечатать элементы вектора с помощью [] at()???

???

Пример: распечатать элементы вектора с помощью итератора

???

Шаблонный конструктор

- Инициализация вектора любой другой последовательностью

Шаблонный конструктор

```
template<typename T> class MyVector{
                        …
public:
   template<typename IT> MyVector(IT first, IT last)
   {
            m_n = 0;
            IT tmp = first;
            while(tmp!=last){ m_n++; ++tmp;}
            m_p = new T[m_n];
            for(int  i= 0; i<m_n; i++){ m_p[i] = *first; ++first;}
            m_cap = m_n;
   }
};
```

Стандартные имена типов

```
template<typename C> void f(C& c)
{
        //Как определить/получить тип
                элемента контейнера?

}
```

Стандартные имена типов

```
template<typename T> class MyVector {
 public:
        typedef  T value_type; //T – снаружи не видно, а псевдоним виден!
        …
  };

int main()
{
        int k1;
        //MyVector<int>::T
        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

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

STL

vector

Полубенцева М.

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

Пример отделения захвата памяти от инициализации

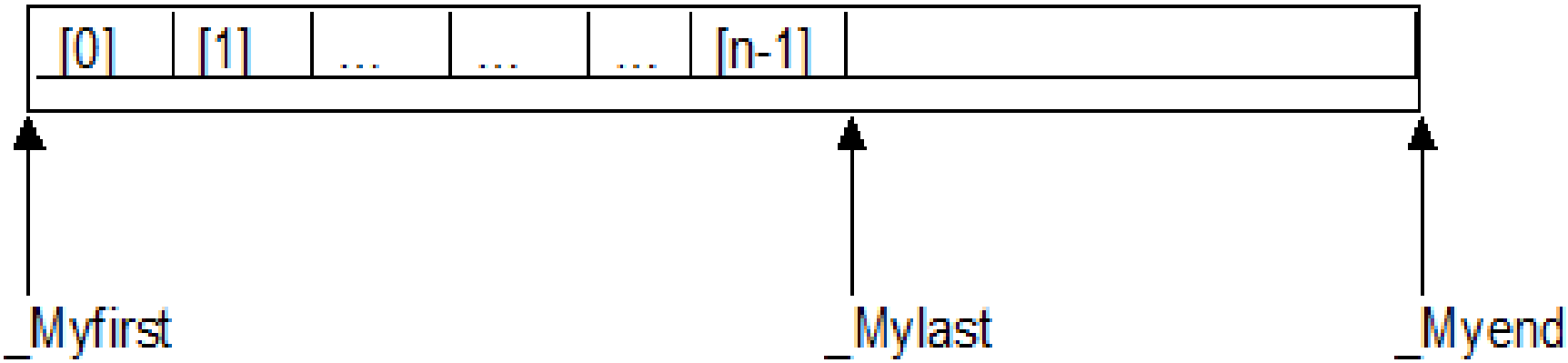
```
#include <new>

{
        int n=…
        A* p = static_cast<A*>(malloc (n*sizeof(A));
        …
        for(int i=0; i<n; i++)
        {
                new (&ar[i]) A(i);
        }
}
```

Класс _Vector_val

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

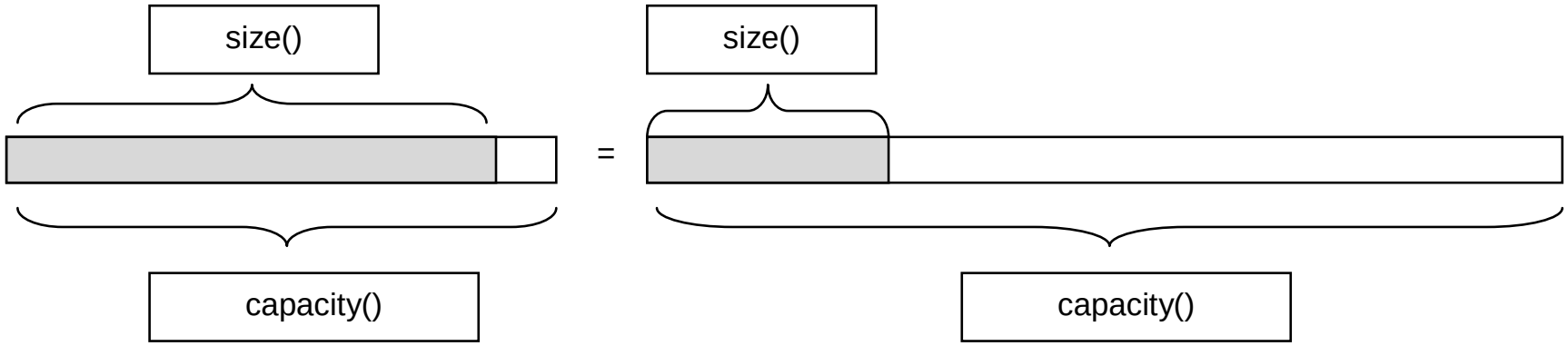
Внутреннее устройство vector



Размеры vector

- **size()** //???
- **capasity()** //???

Оптимизации операций



vector typedefs

difference_type	A type that provides the difference between the addresses of two elements in a vector.
iterator	A type that provides a random-access iterator that can read or modify any element in a vector.
pointer	A type that provides a pointer to an element in a vector.
reference	A type that provides a reference to an element stored in a vector.
reverse_iterator	A type that provides a random-access iterator that can read or modify any element in a reversed vector.
size_type	A type that counts the number of elements in a vector.
value_type	A type that represents the data type stored in a vector.

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 );
vector(std::initializer_list<int> list);
```

C++11 Универсальные списки инициализации

```
int ar[] = {1,2,3,4};
vector<int> v = {1,2,3,4};
```

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, const 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)
  throw out_of_range;
- 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(); //???
```

emplace_back() – C++11

конструирует объект «по месту» без создания копий:

```
class A{
        int a, b;
public:
        A(int, int);
};

int main() {
  vector<A> v;
  v.push_back(A(1,2));
  v. emplace_back(1,2);
}
```

Нет операций с началом последовательности

Нет

push_front(), pop_front()

Использование итераторов

- iterator begin();
- const_iterator begin() const;//cbegin()- C++11
- iterator end();
- const_iterator end() const;//cend() - C++11
- reverse_iterator rbegin();
- const_reverse_iterator rcbegin() const;
- reverse_iterator rend();
- const_reverse_iterator rcend() 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

Пример insert()

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

Пример 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()

Все корректно???

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

Пример erase()

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

Обмен двух векторов

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

Уменьшение емкости вектора

```
void shrink_to_fit( );
```

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

std::vector::emplace_back()

```
int ar[10][3] = { { 1, 2, 3 }, { 4, 5, 6 }};
vector< vector<int>> vv;
vv.reserve(10);
for (int i = 0; i<10; i++)
{
        vv.emplace_back(ar[i], ar[i + 1]);
}
```

vector::emplace()
сравнение insert() и emplace()

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

std::vector<A> v;
v.emplace(<someIterator>, 1);
v.insert(<someIterator>, A(1));
```

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???
        …
}//???
```

Для массива
C++11!!!

```
int ar[] = {1,2,3…};
int* b = begin(ar);
int * e = end(ar)
```