Тема 12. Адаптери (Adaptors)

12.1 Адаптери на контейнери (Container Adaptors)

- Стек
- Опашка
- Приоритетна опашка
- Битова редица

12.2. Адаптери на функции

Адаптери (Adaptors)

Дефиниция:

Адаптерите са темплейтни класове, които предоставят интерфейсно съответствие. Те са базирани на други класове за имплементиране на нова функционалност. Член функциите могат да се добавят или скриват или да се комбинират за постигане на нова финкционалност.

12.1 Адаптери на контейнери (Container Adaptors)

Дефиниция: Контейнерните адаптери са контейнери с ограничена функционалност. Най-съществоното ограничение е липсата на итератори.

Видове:

Има 4 вида адаптери на контейнери:

- Стек;
- Опашка;
- Приоритетна опашка;
- Битова редица.

12.1 Адаптери на контейнери
Стек

- **Стек**.

  Стекът може да се инстанциира чрез vector, list или deque. Той имплементира известната абстрактна структура-последен записан се извлича първи (LIFO). Основните член функции, които са достъпни за потребителя са следните: empty, size, top, push и pop. Дефинирани за операторите operator== и operator< за ставняване на два стека.

Пример:

```
stack<vector<int> > s1;       //създаване чрез вектор
stack<list<int> >    s2;  //създаване чрез списък
stack<deque<int> > s3;       //създаване чрез опашка
s1.push(1); s1.push(5);   // запис на 1 5
cout << s1.top() << endl; // извличане на елемента на върха
s1.pop();           // Премахване на същия
cout << s1.size() << endl; // текуща големина на стека
s1.empty()? cout << "празен стек" : cout << " пълен стек ";
```

Изход: *5*

1

пълен стек

12.1 Адаптери на контейнери
Опашка

Опашка (Queue). Опашката може да се инстанциира с list или deque

Примери:

queue<list<int> > q1;

queue<deque<int> > q2;

Публичните член функции, които са дефинирани са empty, size, front, back, push и pop. Функцията front връща първият елемент от опашката, а pop премахва този елемент. back връща последният елемент, вмъкнат с push в опашката. Както и при stack, две опашки могат да се сравняват с операторите operator== и operator<.

12.1 Адаптери на контейнери Приоритетна опашка

- **Приоритетна опашка** (Priority queue). Приоритетната опашка може да се инстанциира с vector или deque. Приоритетната опашка съхранява елементите, добавяни чрез push, сортирани с обекта функция comp от тип Compare.

```
// използва се less като обект сравнител
priority_queue<vector<int>, less<int> > pq1;
// използва се greater като обект функция сравнител
priority_queue<deque<int>, greater<int> > pq2;
```

Пример:
Създаване на приоритетна опашка priority_queue чрез vector, с less като функция обект сравнител

```
vector v(3, 1);
priority_queue<deque<int>, less<int> > pq3 (v.begin(), v.end() );
```

12.1 Адаптери на контейнери
Приоритетна опашка

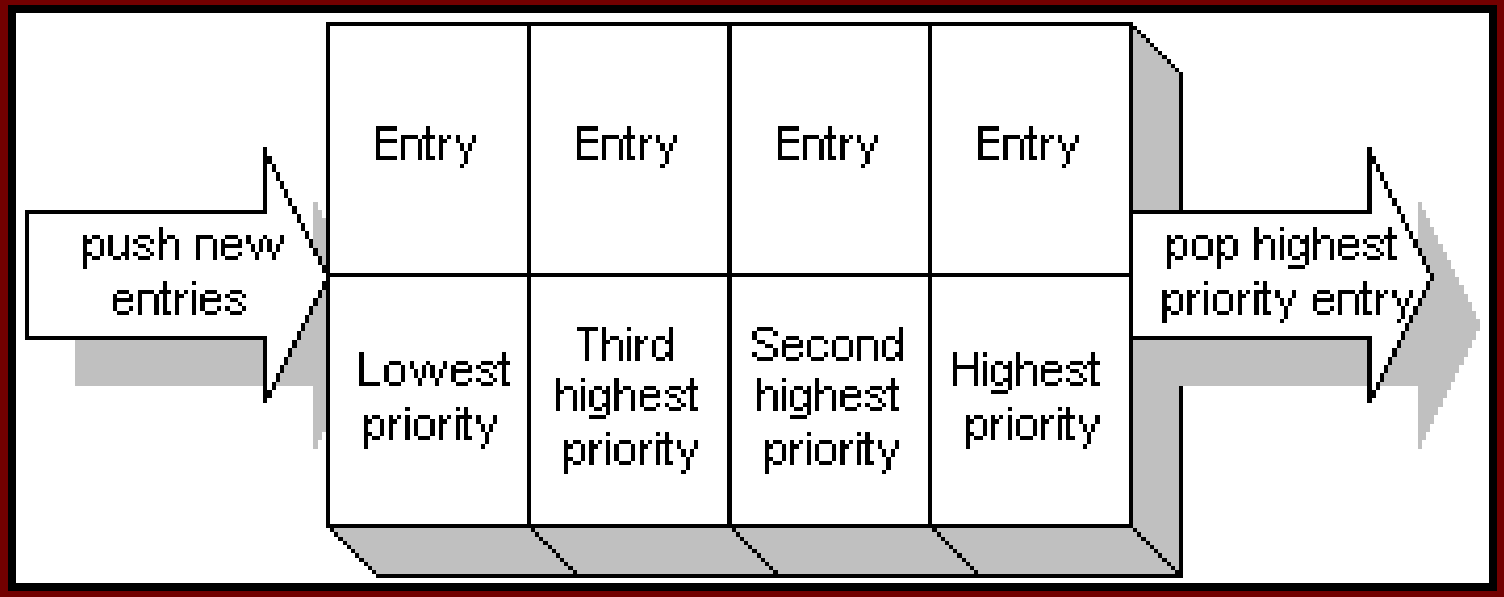
Приоритетна опашка (Priority queue).

- Вътрешно представяне
- Приоритетната опашка може да се инстанциира с vector или deque.
- Основни характеристики:
- Приоритетната опашка съхранява елементите, добавяни чрез push, сортирани с обекта функция comp от тип Compare.

12.1 Адаптери на контейнери
Приоритетна опашка

Приоритетна опашка (Priority queue).

- Вътрешно представяне



12.1 Адаптери на контейнери
Приоритетна опашка

Определяне на приоритета и организацията

- Дефиниране на сравнител less като обект сравнител

```
priority_queue<vector<int>, less<int> > pq1;
```

- greater като обект функция сравнител

```
priority_queue<deque<int>, greater<int> > pq2;
```

Пример:

Създаване на приоритетна опашка priority_queue чрез vector,

с less като функция обект сравнител

```
vector v(3, 1);
priority_queue<deque<int>, less<int> > pq3 (v.begin(), v.end() );
```

12.1 Адаптери на контейнери
Приоритетна опашка

Пример Network:

Подготвителна част образува вектор от двойки цели числа:

```
// Четене на двойки стойности от входен поток в масив:
static int a[MAX];  // времена за чакане на пристигащ пакет
int n = 0;

for (char ch; fscanf(in, "%d%c", &a[n++], &ch) == 2 && ch == ' ';);
vector < pair <int, int> > v;
//зареждане на вектора със стойности-двойки <време,индекс>
for (int i = 0; i < n; i++){
        if (a[i] != 0) v.push_back(make_pair(i + a[i] + 1, i + 1));
}
// сортиране на стойностите по време, индекс
sort(v.begin(), v.end());
```

12.1 Адаптери на контейнери
Приоритетна опашка

Пример:

Създаване на приоритетна опашка priority_queue от цели числа:

```
priority_queue <int> q;
 for (int i = 0, t = 1; i < (int)v.size() || !q.empty(); t++) {
    while (i < (int)v.size() && v[i].first <= t)
        q.push(-v[i++].second);
    if (!q.empty()) {
        fprintf(out, "%d", -q.top()); q.pop();
        fprintf(out, "%c", q.empty() && i >= (int)v.size() ? '\n' : ' ');
    }
    else fprintf(out, "0 ");
 }
```

12.1 Адаптери на контейнери
Битова редица

- Битова редица

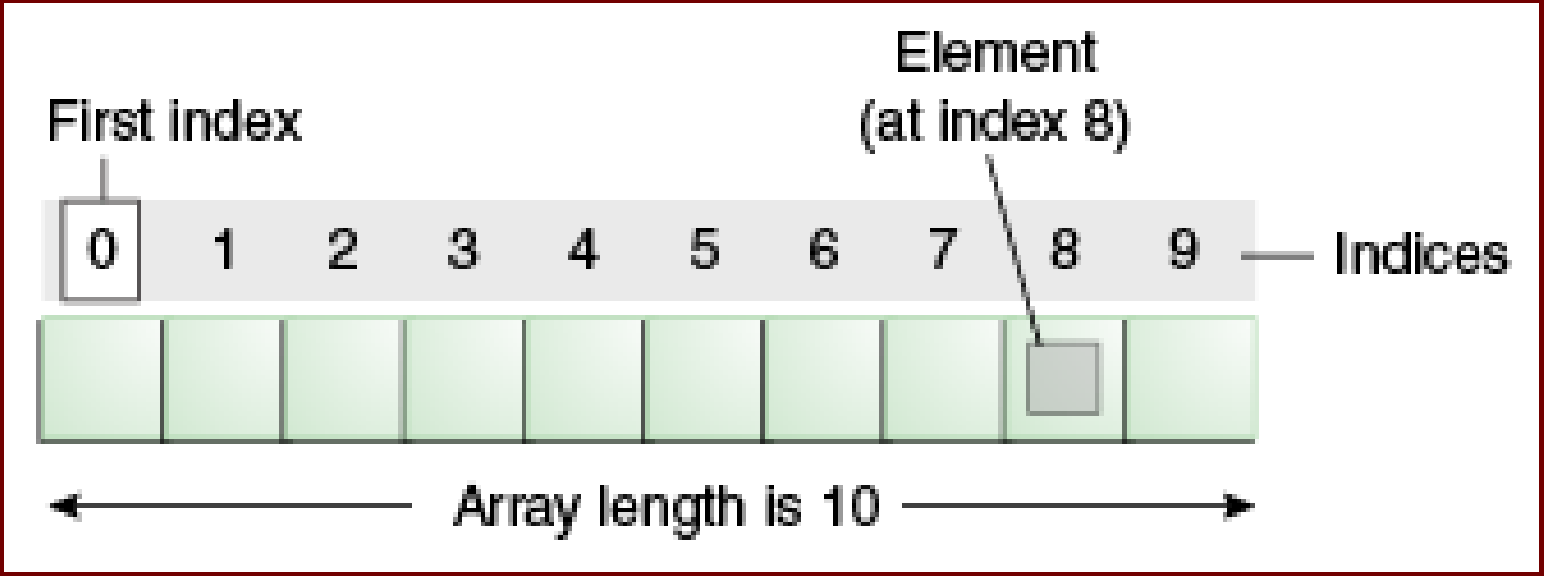
Дефиниция:

Битова редица е фиксирана по размер последователност от битове или логически стойности. Битовите редици (C++) съхраняват **постоянен брой n** бита в една или повече последователни машинни думи в зависимост от размерността **n**.

12.1 Адаптери на контейнери
Битова редица

- Битова редица

Схема на представяне:



12.1 Адаптери на контейнери
Битова редица

Характеристики:

Те се характеризират с начален адрес на разположение на първата дума и размер. Броят на машинните думи, които заема поредицата се определя от размера на редицата/дължината на машинната дума.

Дефиниция:

```
#include <bitset>
template <size_t Bits>
class bitset;
```

12.1 Адаптери на контейнери
Битова редица

Примери:

```
// Инициализация : 0000...0001010101
bitset<50> Bitset50(string("1010101"));

// Инициализация : 0000...0000000110
bitset<50> Bitset50(string("11110000"),2,3);
```

12.1 Адаптери на контейнери
Битова редица

Операции, непроменящи последователността:

size_t **bitset<***bits***>::size** () const //Брой на битовете-*bits*

size_t **bitset<***bits***>::count** () const// Установени битове=1

bool **bitset<***bits***>::any** () const //Има ли установен бит?

bool **bitset<***bits***>::none** () const //Няма ли установен бит?

//Установен ли е бит *idx* в 1:
bool **bitset<***bits***>::test** (size_t *idx*) const //! За грешки

//Проверка дали всички битове на *this и *bits* са еквивалентни:
bool **bitset<***bits***>::operator ==** (const bitset<*bits*>& *bits*) const

//Проверка дали всички битове на *this и *bits* са различни:
bool **bitset<***bits***>::operator !=** (const bitset<*bits*>& *bits*) const

12.1 Адаптери на контейнери
Битова редица

Операции, променящи последователността. Всички връщат променената редица -> генерират изключение out_of_range ако *idx* > size():

bitset<*bits*>& bitset<*bits*>::set() // установяване всички битове в 1

bitset<*bits*>& bitset<*bits*>::set (size_t *idx*) // установяване бит *idx* в 1

// установяване бит *idx* във *value(0 или 1):*
bitset<*bits*>& bitset<*bits*>::set (size_t *idx,* int *value*)

bitset<*bits*>& bitset<*bits*>::reset()// установяване всички битове в 0

// установяване бит *idx* в *0:*
bitset<*bits*>& bitset<*bits*>::reset (size_t *idx*)

// установяване всеки бит в обратната му стойност *0 в 1 и обратно:*
bitset<*bits*>& bitset<*bits*>::flip ()

// установяване бит *idx* в обратната му стойност *0 в 1 и обратно:*
bitset<*bits*>& bitset<*bits*>::flip (size_t *idx*)

12.1 Адаптери на контейнери
Битова редица

Операции, променящи последователността (продължение)

```
// побитово изключващо или с bits
bitset<bits>& bitset<bits>::operator ^= (const bitset<bits>& bits)

// побитово или с bits
bitset<bits>& bitset<bits>::operator | = (const bitset<bits>& bits)

// побитово и с bits
bitset<bits>& bitset<bits>::operator &= (const bitset<bits>& bits)

// побитово изместване наляво с num бита
bitset<bits>& bitset<bits>::operator <<= (size_t num)

// побитово изместване надясно с num бита
bitset<bits>& bitset<bits>::operator >>= (size_t num)
```

12.1 Адаптери на контейнери
Битова редица

Оператор за индексиране. Връщат бита на позиция *idx*:

bitset<*bits*>::reference **bitset<*bits*>::operator [ ]** (size_t *idx*)

bool **bitset<*bits*>::operator [ ]** (size_t *idx*) const

Функции за промяна:

reference& **operator=** (bool) //Установяване на бит по стойността

reference& **operator=** (const reference&)// Установяване чрез копиране

reference& **flip** () // инвертиране на бит

operator bool () const // преобразуване на стойността в логическа

bool **operator~** () const // връща инвертираната стойност на бит

12.1 Адаптери на контейнери
Битова редица

- Оператори, създаващи нови редици. Създават и връщат новосъздадената редица:

//Връща нова битова редица с инвертирани на текущата битове:

bitset<*bits*> **bitset<*bits*>::operator ~** () const

//Всички битове се изместват с *num* позиции наляво:

bitset<*bits*> **bitset<*bits*>::operator <<** (size_t *num*) const

//Всички битове се изместват с *num* позиции надясно:

bitset<*bits*> **bitset<*bits*>::operator >>** (size_t *num*) const

12.1 Адаптери на контейнери
Битова редица

- Оператори, създаващи нови редици. Създават и връщат новосъздадената редица:

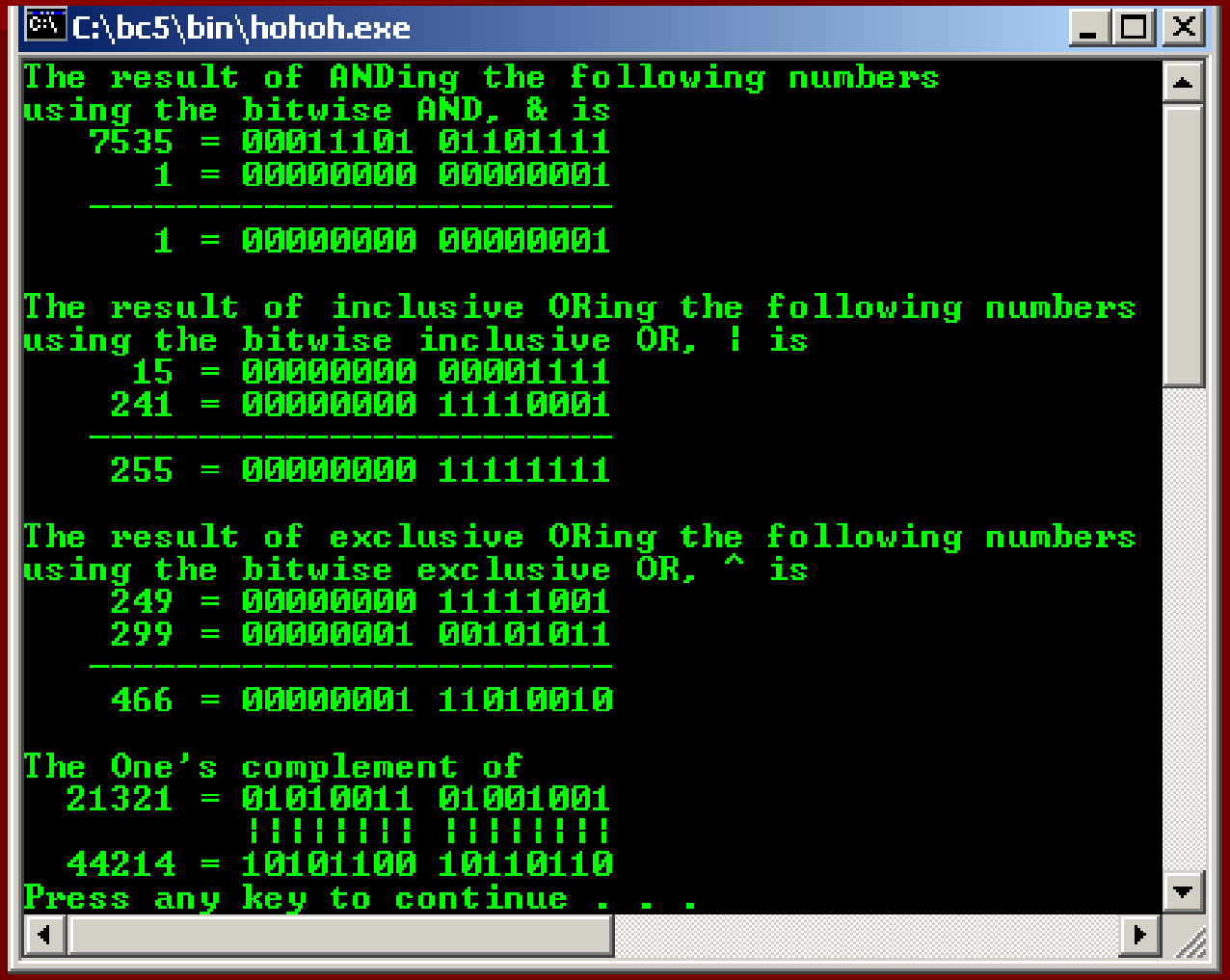
//Резултатът се получава с побитово и между редици *bits1* и *bits2*
bitset<*bits*> **operator &** (const bitset<*bits*>& *bits1,* const bitset<*bits*>& *bits2*)

//Резултатът се получава с побитово или между редици *bits1* и //*bits2*
bitset<*bits*> **operator |** (const bitset<*bits*>& *bits1,* const bitset<*bits*>& *bits2*)

//Резултатът се получава с побитово изключващо или между редици //*bits1* и *bits2 (1 в bits1 и 0 в bits2 или обратно) :*
bitset<*bits*> **operator^** (const bitset<*bits*>& *bits1,* const bitset<*bits*>& *bits2*)

12.1 Адаптери на контейнери
Битова редица

Илюстрация:



12.1 Адаптери на контейнери
Битова редица

- Входно-изходни операции върху битови редици.

//Четене на битова редица от входен поток. Входът трябва
//да съдържа последователности от символи '0' и '1'.
// Завършва при различен символ или символ за край на файла:
istream& **operator>>** (*istream*& *strm,* bitset<*bits*>& *bits*)
//Запис на битова редица като низ от символи, съдържащи
// символи '0' и '1':

ostream& **operator <<** (*ostream*& *strm,* const bitset<*bits*>& *bits*)

12.1 Адаптери на контейнери
Битова редица

Пример:

```
enum WeekDays { Mon,Tue,Wed,Thu,Fri,Sat,Sun,WeekDaysNum
    };
bitset<WeekDaysNum> workingDays, weekendDays;
```

Начално установяване на редиците в 0:

```
workingDays.reset(); weekendDays.reset();
```

Установяване на работните дни от седмицата в 1:

```
  workingDays.set(Mon); // понеделник
  workingDays.set(Tue); // вторник
  workingDays.set(Wed); // сряда
  workingDays.set(Thu); // четвъртък
  workingDays.set(Fri); // петък
```

Установяване на почивните дни в 1:

```
  weekendDays.set(Sat);
  weekendDays.set(Sun);
```

12.1 Адаптери на контейнери
Битова редица

Извеждане на битовите редици в изхода:

```
cout << "Работни дни са : " << workingDays << endl;
cout << "Почивни дни са : " << weekendDays << endl;
```

Работни дни са : 0011111

Почивни дни са : 1100000

Операции с множествата, представени с битови редици.

- Операция допълнение:

```
cout << "Почивни дни са неработните: " << ~workingDays <<
    endl;
```

Изход:

Почивни дни са неработните : 1100000

- Операция сечение:

```
cout << "Почивни и работни: " <<
(workingDays & weekendDays) << endl;
```

Изход:

Почивни и неработни : 0000000

- Операция обединение:

```
cout << "Почивни или работни : " <<
(workingDays | weekendDays) << endl;
```

Изход:

Почивни или неработни : 1111111

12.1 Адаптери на контейнери
Битова редица

Пример за приложение – регионална 2012 г. Румъния:

Определяне на прости числа в голям диапазон:

В много на брой серии от граници. Обща размерност:

1 до 10000000

- Трябва да се съхранят всички възможни отговори;
- При искане на произволна серия да се дава отговор от холдера;
- Ползването на холдер от типа на данните int дава лош резултат;
- Изход????

12.2. Адаптери на функции

- Инвертори:

Инверторите not1 и not2 са функции, които използват унарна или бинарна предикатни функции респективно и връщат техните комплиментарни (логически обратни резултати). Спецификацията на унарният и бинарен обратен предикат може да се представи по следния начин:

```
template <class Predicate>
unary_negate<Predicate> not1(const Predicate& pred) {
    return unary_negate<Predicate>(pred);
}
template <class Predicate>
binary_negate<Predicate> not2(const Predicate& pred) {
    return binary_negate<Predicate>(pred);
}
```

12.2. Адаптери на функции

- Класовете unary_negate и binary_negate работят само с обекти функции, които имат дефинирани аргументни и резултантен типове. Това означава, че при инстанциирането на класовете инвертор могат да се използват за унарните обекти и бинарните обекти функции аргументният и резултантен тип съответно:

Predicate::argument_type и
Predicate::result_type
Predicate::first_argument_type, Predicate::second_argument_type и Predicate::result_type

```
vector<int> v;
// запълване на v с 1 2 3 4
sort (v.begin(), v.end(), not2 (less_equal<int>()) );
```

Изход: *4 3 2 1*

12.2. Адаптери на функции

- Ограничители

Ограничителитете bind1st и bind2nd получават обект функция f с два аргумента и стойност x и връщат обект функция с един аргумент, конструиран от f с първи или втори аргумент, в съответствие със стойността x.

Пример: Даден е контейнер със стойности цели числа. Да се заместят всички елементи по-малки от определена граница (5) с тази граница.

```
#include <iostream> //cout, ostream_iterator
#include <vector>
#include <algorithm> //replace_if
#include <functional> //bind2nd, less
using namespace std;
void main() {
vector<int> v;
v.push_back(4); v.push_back(6); v.push_back(10); v.push_back(3); v.push_back(13);
    v.push_back(2);
int bound = 5;
replace_if (v.begin(), v.end(), bind2nd (less<int>(), bound), bound);
  copy (v.begin(), v.end(), ostream_iterator<int> (cout, " ") );
```

Изход: 5 6 10 5 13 5

Въпроси ?