

# Dzisiejszy wykład

- # Wzorce funkcji
- # Wzorce klas
- # Tablica asocjacyjna

# Wzorce

- Często pojawia się konieczność pisania podobnych funkcji lub klas operujących na argumentach różnych typów

```
int minimum(int x, int y)
{
    if (x < y)
        return x;
    else
        return y;
}
```

```
double minimum(double x, double y)
{
    if (x < y)
        return x;
    else
        return y;
}
```

- Wzorce umożliwiają jednokrotne napisanie kodu, który będzie wykorzystywany dla wielu typów danych

```
TYPE minimum(TYPE x, TYPE y)
{
    if (x < y)
        return x;
    else
        return y;
}
```

# Wzorce

```
template <class aType> aType minimum(aType x, aType y)
{
    if(x<y)
        return x;
    else
        return y;
}
```

Słowo kluczowe

Nazwa parametru określającego typ

Nazwa wzorca funkcji

Wyraz *aType* jest nazwą podaną przez użytkownika,  
nie słowem kluczowym  
*aType* może być prawie dowolnym typem  
Operacja *<* musi być zdefiniowana dla *aType*

# Wzorce

```
#include <iostream>
#include <string>
using namespace std;
template < class T > T minimum (T x, T y)
{
    if (x < y)
        return x;      Definicja wzorca funkcji
    else
        return y;
}
int main ()
{
    int x = 50, y = 30;
    string a = "hello", b = "goodbye";
    cout << "minimum for ints " << minimum(x, y) << endl;
    cout << "minimum for strings " << minimum (a, b) << endl;
}
```

Konkretyzacja wzorca dla danego typu/typów argumentów

# Wzorce

# Składnia deklaracji wzorca:

# Wzorzec klasy

```
template <class aType> className {  
    // class definition  
};
```

# Wzorzec funkcji

```
template <class aType> ReturnType  
functionName(arguments)  
{  
    // function definition  
}
```

# Funkcja zamieniająca obiekty

```
#include <iostream>
#include <string>
using namespace std;
template<class T> void swap_value(T& var1, T& var2) {
    T temp;
    temp = var1;
    var1 = var2;
    var2 = temp;
}
int main() {
    int int1 = 1, int2 = 2;
    cout << "original " << int1 << " " << int2 << endl;
    swap_value(int1, int2);
    cout << "after " << int1 << " " << int2 << endl;
    string s1 = "one", s2 = "two";
    cout << "original " << s1 << " " << s2 << endl;
    swap_value(s1, s2);
    cout << "after " << s1 << " " << s2 << endl;
}
```

Działa dla wszystkich klas, dla których istnieje *operator =*

# Funkcja zamieniająca obiekty

```
// using swap_value template as before
class Value {
    private:
        int x;
    public:
        Value(int i = 0) : x(i) {}
        friend ostream& operator<<(ostream& out, Value v);
};
ostream& operator<<(ostream& out, Value v) {
    out << "value(" << v.x << ")";
    return out;
}
int main() {
    Value v1(5), v2(10);
    cout << "original " << v1 << " " << v2 << endl;
    swap_value(v1, v2);
    cout << "after " << v1 << " " << v2 << endl;
}
```

# Sortowanie bąbelkowe

```
// using swap_value template as before
template < class T >
void bubblesort (T * A, unsigned int size)
{
    for (unsigned int i = 0; i < size-1; i++)
        for (unsigned int j = size - 1; j > i; j--)
            if (A[j] < A[j-1])
                swap_value (A[j], A[j-1]);
}
int main ()
{
    string S[5]={"ula","ala","ola","genowefa","stefania"};
    int I[3]={4,1234,-7};
    bubblesort(S,5);
    bubblesort(I,3);
    for(int i=0;i<5;i++)
        cout<<S[i]<<endl;
    for(int i=0;i<3;i++)
        cout<<I[i]<<endl;
}
```

Działa dla wszystkich klas, dla których istnieje operator < i operator =

# Wzorce z punktu widzenia kompilatora

- # Podczas konkretyzacji wzorca kompilator wykonuje podobne czynności jak przy rozwijaniu makrodefinicji
  - # Programista pisze: `minimum(2,3)`
  - # Kompilator emittuje kod dla nowej kopii funkcji nazywając ją np. `minimum_int` i zastępuje `T` przez `int` w całym kodzie wzorca
  - # Kompilator musi mieć dostęp do pełnego kodu wzorca w momencie konkretyzacji
  - # Zazwyczaj cały kod wzorca umieszczany jest w pliku nagłówkowym

# Wzorzec klasy *vector*

```
template<class C> class vector
{
    C *dane;
    unsigned int size;
public:
    class index_out_of_range{};
    explicit vector (int s);
    ~vector ();
    C& operator[] (unsigned int pos);
    C operator[] (unsigned int pos) const;
    vector (const vector<C> & s);
    void swap(vector<C>& s);
    vector<C> & operator= (const vector<C> & s);
    friend ostream &
        operator<< (ostream & o, const vector<C> & v);
};
```

Parametr wzorca

Nazwa wzorca

Słowa kluczowe

# Użycie parametru wzorca

# Do czego można użyć parametru wzorca:

# Do określenia typu danych używanego w klasie

C \*dane;

# Do określenia typu parametru metody

void swap(vector<C>& s);

# Do określenia typu danej zwracanej przez metodę

C& operator[] (unsigned int pos);

# Konkretyzacja wzorca

- Dana jest deklaracja wzorca

```
template<class C> class vector {...};
```

- *vector* jest wzorcem klasy, aby uzyskać nazwę klasy należy dokonać konkretyzacji wzorca

*vector* - nazwa wzorca klasy

*vector<int>* - nazwa klasy, wektora liczb całkowitych

- Obiekty tworzy się przy użyciu skonkretyzowanego wzorca klasy

```
vector<string> a(10);
```

```
typedef vector<int> intVector;
```

```
intVector b(5);
```

- Konkretny typ (*string* lub *int*) zostaje podstawiony pod formalny parametr wzorca (C) w definicji wzorca

# Użycie wzorców klas

- # Po konkretyzacji wzorca używa się tak samo jak zwykłej klasy

```
vector<string> a(5), b(10);  
a.swap(b);
```

- # Parametr dla funkcji zamiany został podany jako *vector<C>&* w definicji wzorca
- # Został odwzorowany jako *vector<string>&* przy deklaracji obiektu *a*
- # Stąd, przy wywołaniu swap musimy jako argument podać *vector<string>*

# Implementacja metod wzorca klas

*template i parametry formalne*

```
template<class C> vector<C>::vector (int s)
{
    //member function body goes here
};
```

operator zasięgu i nazwa funkcji

nazwa klasy i parametry formalne

```
template<class C> void vector<C>::swap(vector<C>& s)
{
    //member function body goes here
};
```

typ wartości zwracanej

```
template<class C> class vector
{
//...
    void swap(vector<C>& s)
    {
        //member function body goes here
    }
//...
};
```

Definicja od razu w  
deklaracji wygląda  
prościej

# Inicjalizacja pamięci i konstruktor domyślny

- # Poniżej przedstawiono reguły określające inicjalizację pamięci zaalokowanej dynamicznie
- # *int* reprezentuje tutaj dowolny typ POD (Plain Old Data), czyli taki, którego definicja wyglądałaby dokładnie tak samo w C

```
class Object /* ... */;

Object* p1 = new Object;    /* Object::Object() used */
Object* p2 = new Object(); /* Object::Object() used */
int*    p3 = new int;      /* not initialized! */
int*    p4 = new int();    /* initialized to zero */
Object* p5 = new Object[7];/* Object::Object() used 7 times */
int*    p6 = new int[7];   /* not initialized! */
int*    p7 = new int[7](); /* initialized to zero! */
```

# Konstruktor kopiący i wyjątki

- # W przypadku wystąpienia wyjątku w konstruktorze klasy *vector<C>*, destruktor klasy *vector<C>* nie zostanie wywołany
- # Dlatego należy zadbać o zwolnienie pamięci w przypadku wystąpienia wyjątku w konstruktorze

```
template<class C> class vector
{
//...
    vector (const vector<C> & s)
    {
        dane = new C[s.size];
        size = s.size;
        try {
            for (unsigned i = 0; i < size; i++)
                dane[i] = s.dane[i];
        }
        catch(...)
        {
            delete [] dane;
            throw;
        }
    }
};
```

# Operator przypisania i wyjątki

■ W celu zagwarantowania poprawnego działania operatora przypisania w przypadku zgłoszenia wyjątku przez operator przypisania dla typu C, operator przypisania implementujemy przy pomocy konstruktora kopiącego i funkcji *swap*

```
template<class C> class vector
{
//...
void swap(vector<C>& s)
{
    C* t1=s.dane;
    unsigned int t2=s.size;
    s.dane=dane;
    s.size=size;
    dane=t1;
    size=t2;
}
vector<C> & operator= (const vector<C> & s)
{
    if (this == &s)
        return *this;
    vector<C> n(s);
    swap(n);
    return *this;
}
};
```

Funkcja *swap* nigdy nie zgłasza wyjątku

W przypadku zgłoszenia wyjątku przez konstruktor kopiący, obiekt pozostaje w niezmienionym stanie

# Założenia co do parametru C wzorca *vector*

- # Posiada konstruktor bezparametryowy
  - # użyty w konstruktorach do stworzenia tablicy obiektów typu C
- # Posiada konstruktor kopiący
  - # użyty przy przekazywaniu parametru w operatorze indeksowania
- # Posiada operator przypisania
  - # użyty w operatorze przypisania wzorca *vector<C>*
- # Posiada operator <<
  - # użyty w operatorze << wzorca *vector<C>*

# Wzorce klas z punktu widzenia kompilatora

- # Programista pisze: `vector<int> a;`
- # Kompilator emuluje nową kopię klasy nazywając ją np. `vector_int` i zastępuje `C` przez `int` w całym kodzie wzorca klasy
- # Kompilator musi mieć dostęp do pełnego kodu wzorca w momencie konkretyzacji
- # Zazwyczaj cały kod wzorca umieszczany jest w pliku nagłówkowym

# Wzorce klas i kontrola typów

- # Ciała metod używają tych samych algorytmów dla wektora liczb całkowitych, liczb rzeczywistych czy łańcuchów tekstowych
  - # Kompilator wykonuje jednak kontrolę typów
- # Jeżeli napiszemy

```
vector<int> a;
vector<double> b;
vector<string> c;
```

kompilator wygeneruje trzy różne klasy z normalnymi regułami kontroli typów

```
a[7] = "ala"; //compile-time error
```

# Tablica asocjacyjna

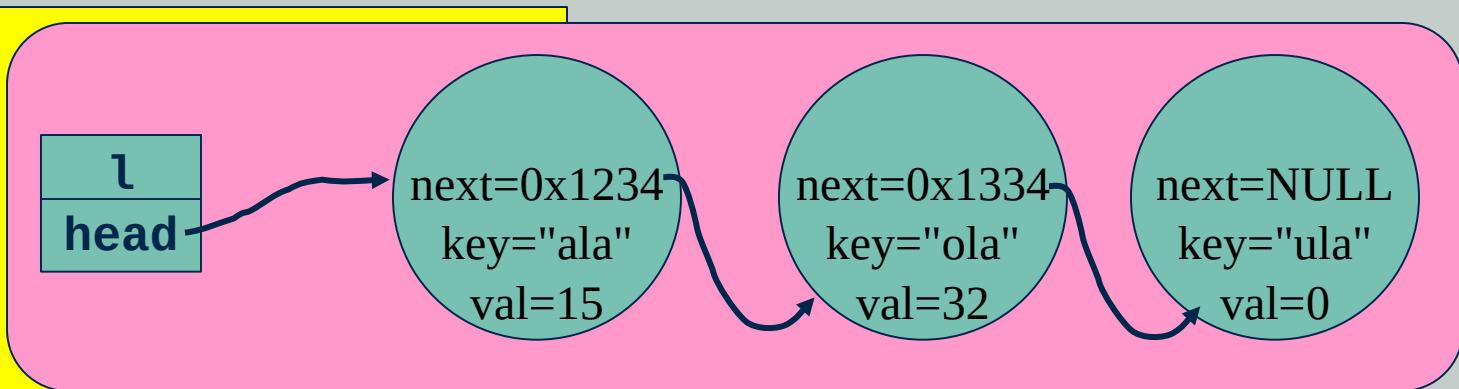
- # Tablica asocjacyjna to struktura danych, zawierająca zbiór par <klucz, wartość>
- # Może być interpretowana jako funkcja przypisująca kluczowi odpowiednią wartość
- # Może to być na przykład tablica liczb całkowitych indeksowana łańcuchami tekstowymi

```
assocTab at;  
at["ala"]=15;  
at["ola"]=32;  
cout << at["ala"]<<at["ola"]<<at["ula"]<<endl;
```

# Tablica asocjacyjna

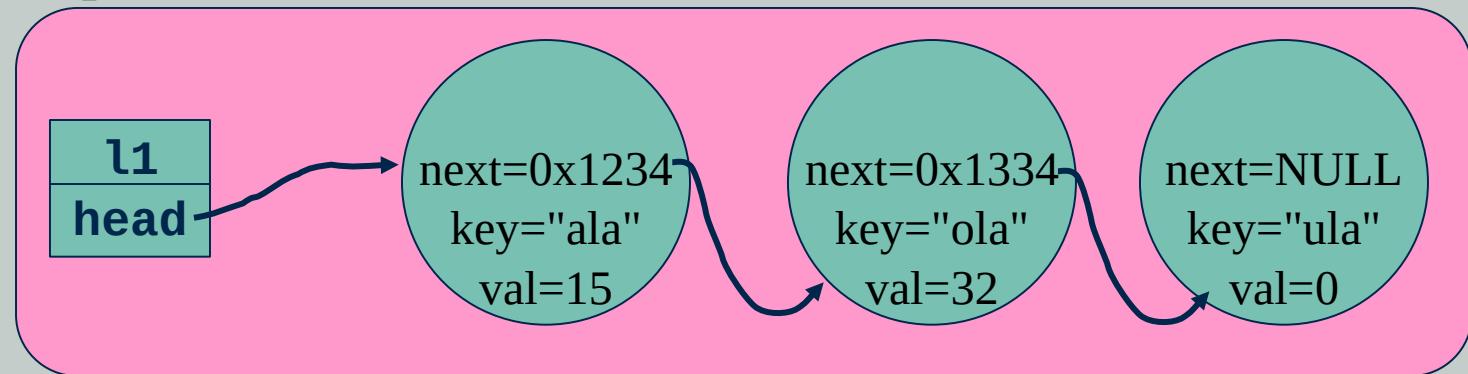
# Tablicę asocjacyjną można zbudować w oparciu o strukturę listy

```
class assocTab
{
private:
    struct node
    {
        node *next;
        char* key;
        int val;
    };
    node * head;
    void insert (const char *key, int value);
    void clear ();
    node *find (const char *key) const;
    void swap (assocTab & l);
public:
    assocTab ();
    assocTab (const assocTab & l);
    assocTab & operator= (const assocTab & l);
    ~assocTab ();
    int &operator[] (const char *);
};
```



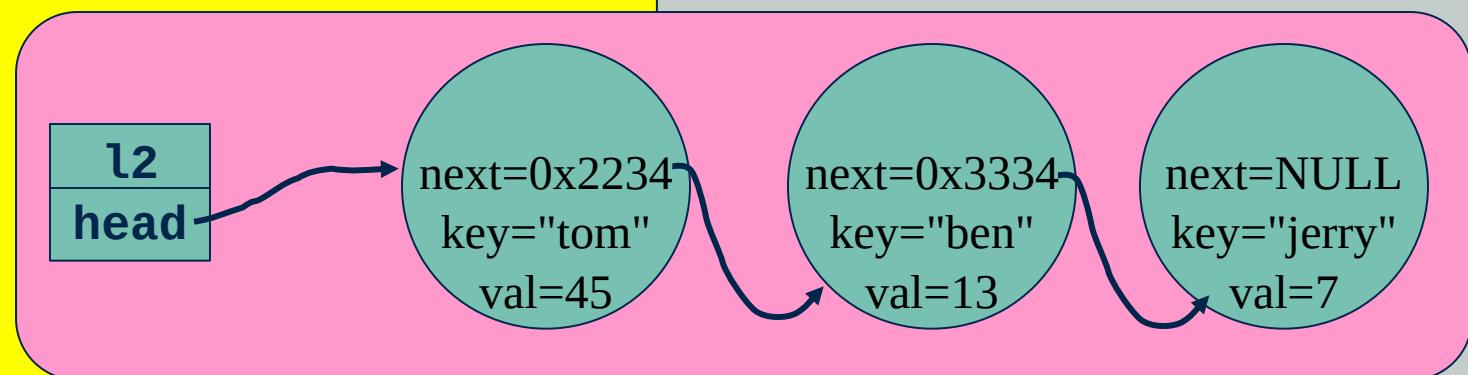
# Tablica asocjacyjna - funkcja swap

- W celu zapewnienia właściwego działania operatora przypisania, zaimplementowano go przy pomocy konstruktora kopiącego i funkcji *swap*



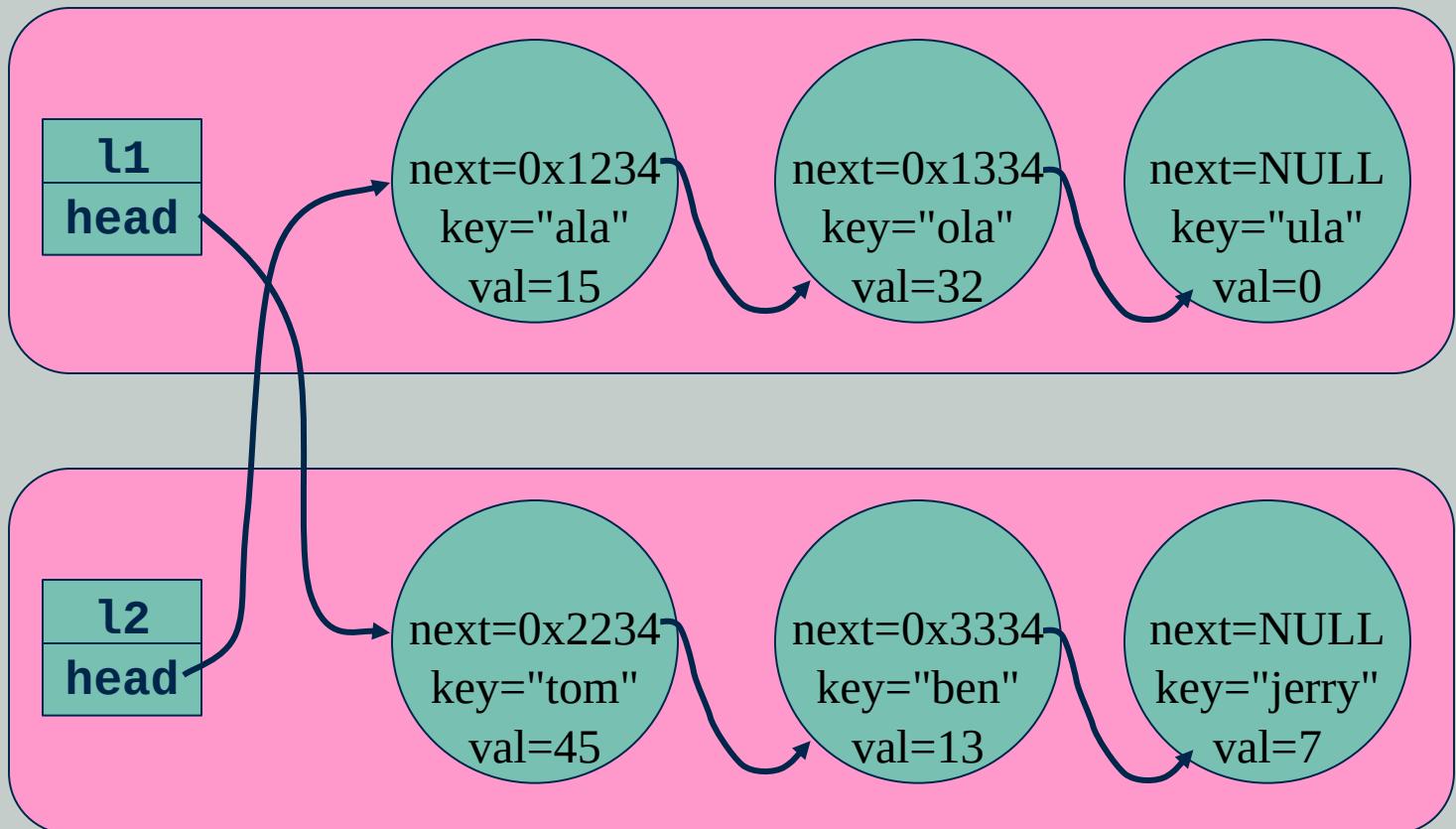
```
assocTab & assocTab::operator= (const assocTab & l)
```

```
{  
    if (&l == this)  
        return *this;  
    assocTab t (l);  
    swap (t);  
    return *this;  
}
```



# Tablica asocjacyjna - funkcja swap

# Funkcja *swap* zamienia jedynie wskaźniki, nie może więc spowodować wyjątku



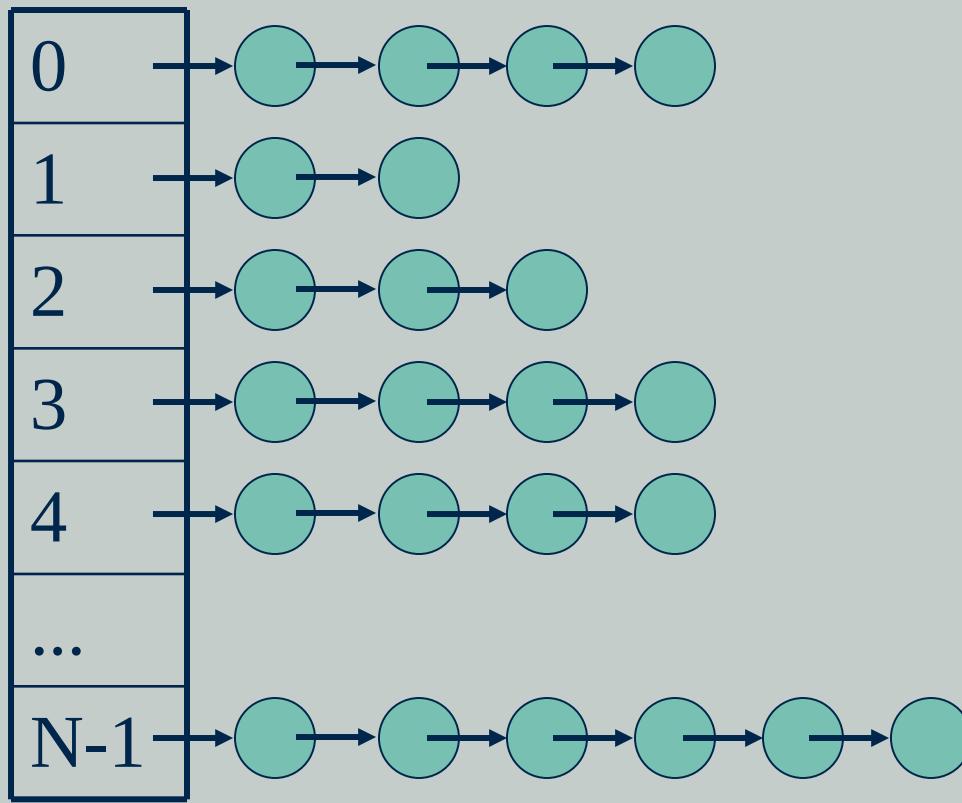
# Tablica asocjacyjna - ulepszenie

- # Czas wyszukiwania w tablicy asocjacyjnej zaimplementowanej jako lista jest proporcjonalny do liczby elementów w tablicy
- # Często stosowaną metodą skrócenia czasu wyszukiwania jest zastosowanie funkcji mieszającej
- # Funkcja mieszająca jest to funkcja, która dla każdego łańcucha tekstowego zwraca liczbę z przedziału  $<0...N-1>$ . Rozkład wartości powinien być równomierny dla wszystkich łańcuchów umieszczanych na liście
- # Dobór odpowiedniej funkcji mieszającej dla danej dziedziny zastosowań jest sztuką

```
unsigned int hash(const char*);
```

# Tablica asocjacyjna z funkcją mieszającą

- # Zamiast jednej listy, tworzymy N list
- # Na każdej liście znajdują się węzły o tej samej wartości funkcji mieszającej dla klucza



W idealnym przypadku, długości wszystkich list są jednakowe i czas wyszukiwania skrócony n-krotnie

# Tablica asocjacyjna z funkcją mieszającą

# W definicji klasy skorzystamy z uprzednio zdefiniowanego wzorca klasy *vector*

```
class hashAssocTab
{
    vector<assocTab> v;
    int chains;
public:
    hashAssocTab(unsigned int c): v(c),chains(c){};
    unsigned int hash(const char* s)
    {
        unsigned sum=0;
        while(*s)
        {
            sum+=(unsigned char)(*s);
            s++;
        }
        return sum % chains;
    }
    int& operator[] (const char *s)
    {
        return (v[hash(s)])[s];
    }
    // default constructor, destructor and assignment operator not necessary
};
```