Lecture Material

- Function templates
- Class templates
- Associative table
- Static members

Templates

- Frequently we need to write similar function or classes operating on arguments of different types

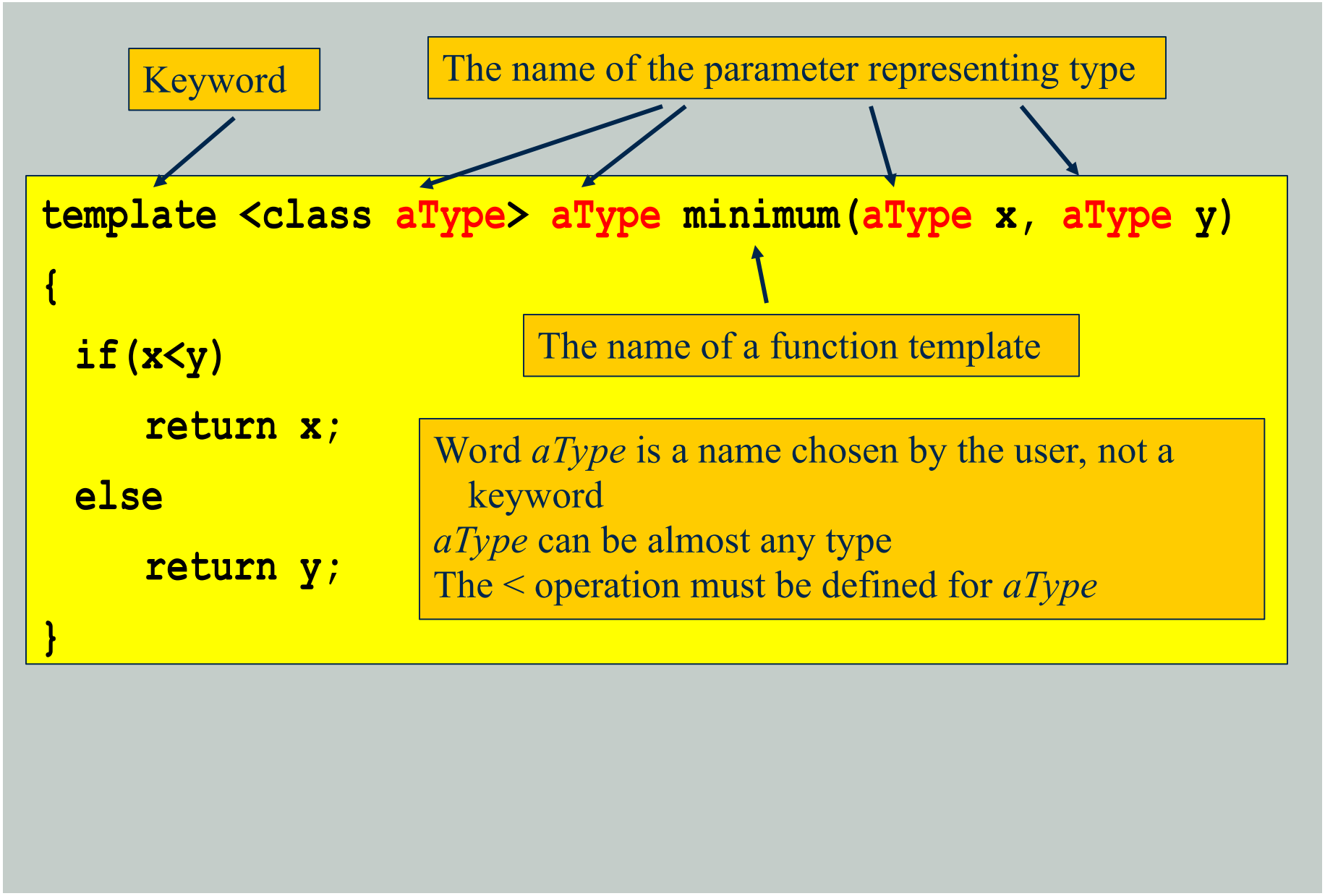
```
int minimum(int x, int y)
{
   if (x < y)
        return x;
   else
        return y;
}
```

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

- The templates enable us to write the code used for different types just once

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

Templates


Keyword
The name of the parameter representing type
template <class aType> aType minimum(aType x, aType y)
{
if(x<y)
return x;
else
return y;
}
The name of a function template
Word aType is a name chosen by the user, not a keyword
aType can be almost any type
The < operation must be defined for aType

Templates

```
#include <iostream>
#include <string>
using namespace std;
template < class T > T minimum (T x, T y)
{
  if (x < y)
    return x;
  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;
}
```

Function template definition

Template instantiation for given argument type(s)

Templates

- Template declaration syntax:
  - Class template

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

  - Function template

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

Object Swap Function

```
#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;
}
```

Works for all classes having operator =

Object Swap Function

```
// 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;
}
```

Bubblesort

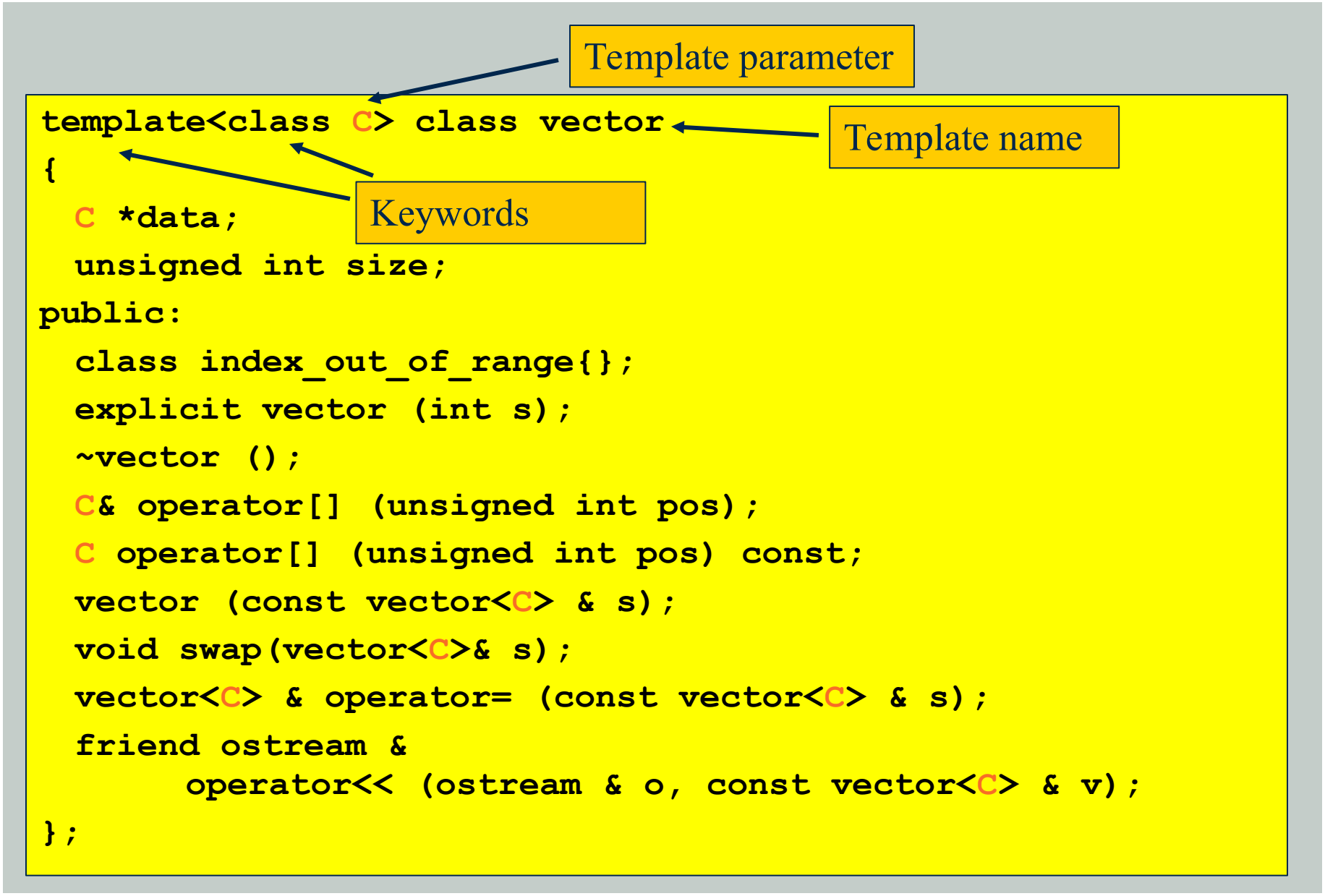
```
// using swap_value template as before
template < class T >
void bubblesort (T * A, unsigned int size)
{
  for (unsigned int i = 0; i < size-1; i++)
    for (unsigned int j = size - 1; j > i; j--)
      if (A[j] < A[j-1])
        swap_value (A[j], A[j-1]);
}
int main ()
{
  string S[5]={"ula","ala","ola","genowefa","stefania"};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

Works for all clasees having operator < and operator =

Templates from the Compiler's Point of View

- During template instantiation the compiler performs similar operaions, as macro evaluation
  - The programmer writes: minimum(2,3)
  - The compiler emits the code for a new copy of a function calling it e.g. minimum_int and replaces T by int in the entire template code
  - The compiler needs access to the full template code at the moment of instantiation
  - Usually the entire template code is located in a header file

The *vector* Class Template




```
template<class C> class vector
{
  C *data;
  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);
};
```

Using Template Parameters

- We can use the template parameter to:
  - Define the data type used in class

```
C *data;
```

  - Define the type of member function parameter

```
void swap(vector<C>& s);
```

  - Define the data type returned by member function

```
C& operator[] (unsigned int pos);
```

Template Instantiation

- We have the template definition

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

- *vector* is a class template, to obtain the class name we have to instantiate the template

  `vector` - name of the class template

  `vector<int>` - name of the class, vector of integers

- The object are created using the instantiated class template

```
vector<string> a(10);
typedef vector<int> intVector;
intVector b(5);
```

- The specific type (*string* or *int*) is substituted as a formal template parameter (*C*) in template definition

Using Class Templates

- After the template instantiation it is used in the same way, as an ordinary class

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

- The parameter to the swap funtion was given as *vector<C>&* in the template definition
- Has been mapped to *vector<string>&* in the definition of object *a*
- Therefore, when calling *a.swap*, we need to give *vector<string>* as an argument

Implementation of Class Template Members

template and formal parameters

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

scope operator and function name

class name and formal parameters

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

type of return value

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

Definition at the point of declaration looks much simpler

Memory Initialization and Default Constructor

- The rules regarding initialization of memory allocated are presented below
- *int* represents here any POD (Plain Old Data) type, i.e. a type, whose definition would look exactly the same in 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 */
```

Copy Constructor and Exceptions

- If an exception is thrown in the constructor of class *vector<C>*, destructor of class *vector<C>* will not be invoked
- Therefore we need to take care of freeing the memory in case of an exception in constructor

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

The Assignment Operator and Exceptions

- To guarantee the proper operation of an assignment operator when an exception is thrown by an assignment operator of type *C*, we implement an assignment operator in terms of copy constructor an the *swap* function

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

The *swap* function never throws an exception

When the exception is thrown in the assignment operator, the state of the object is not changed

Assumptions Regarding the Parameter C

- Has the default constructor
  - used in constructors to create an array of objects of type *C*
- Has the copy constructor
  - used in parameter passing in indexing operator
- Has the assignment operator
  - used in assignment opeator of template *vector<C>*
- Has the << operator
  - used in << operator of template *vector<C>*

Class Templates from the Compiler's Point of View

- The programmer writes: vector<int> a;
- Compiler emits a new copy of a class definition naming it e.g. vector_int and replaces C by int in the entire template code
- The compiler needs the access to the full template code at the moment of instatiation
- Usually entire code of the template is placed in a header file

Class Templates and Type Checking

- The bodies of member functions use the same algorithms for a vector of integers, reals or character strings
  - However, the compiler performs typechecking
- If we write

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

the compiler will generate three different classes with the normal typechecking rules

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

Associative Table

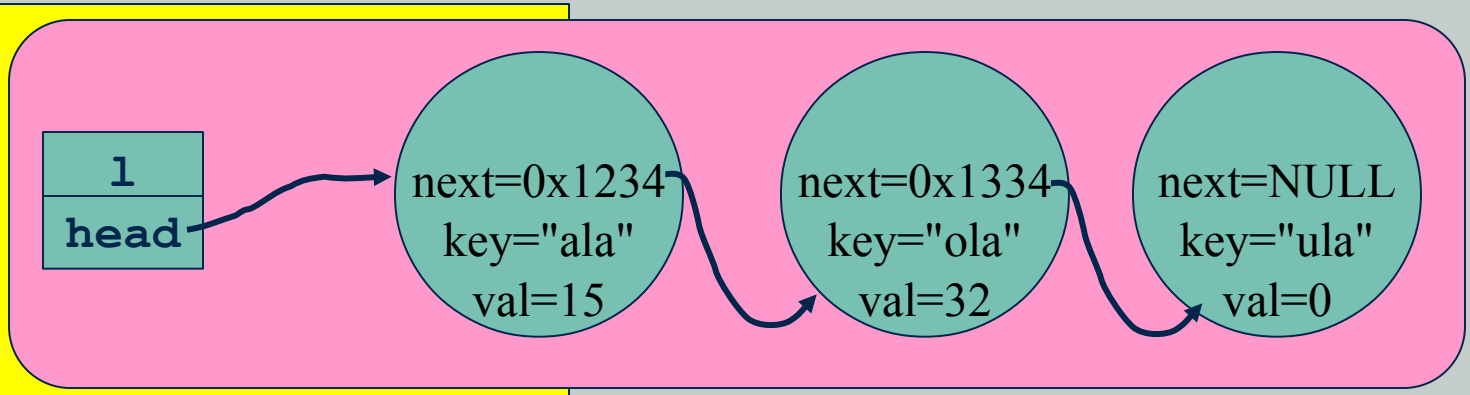
- An associative table is a data structure containing the set of pairs <key,value>
- Can be interpreted as a functions mapping the appropriate value to the key
- It can be e.g. an array of integers indexed by character strings

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

Associative Table

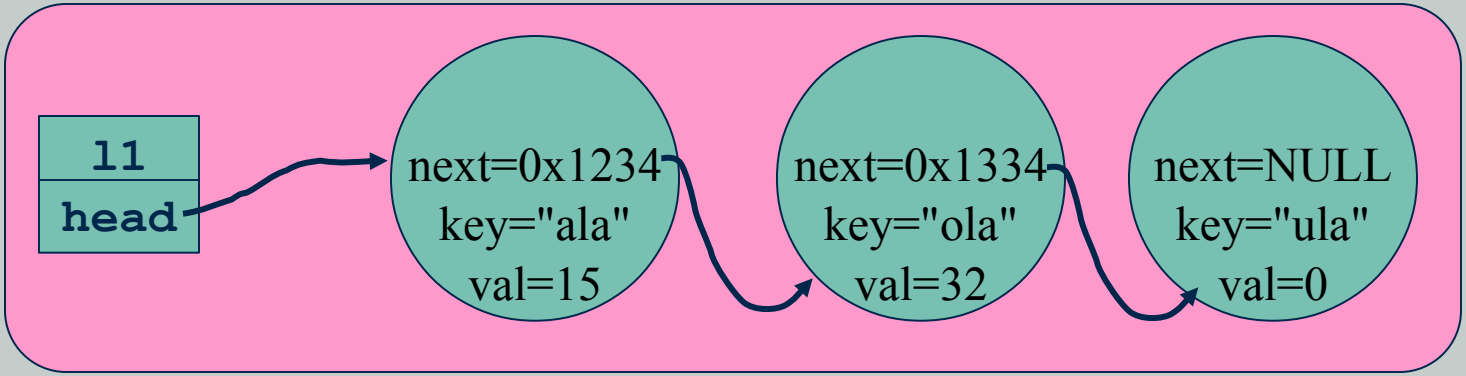
- The associative table can be built based on a list structure

```
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 *);
};
```

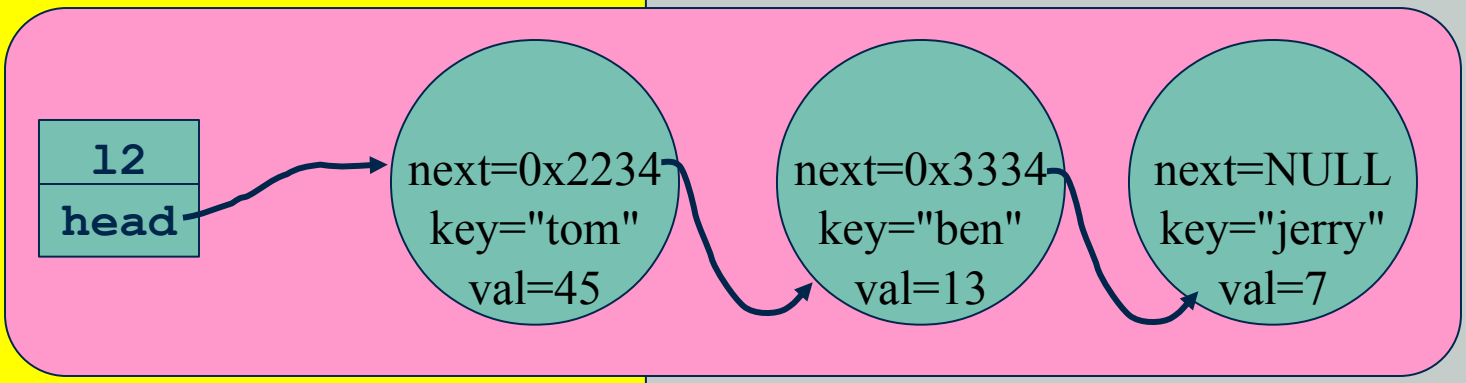


Associative Table - Function *swap*

- To ensure proper operation of an assignment operator, it has been implemented in terms of the copy constructor and the *swap* function

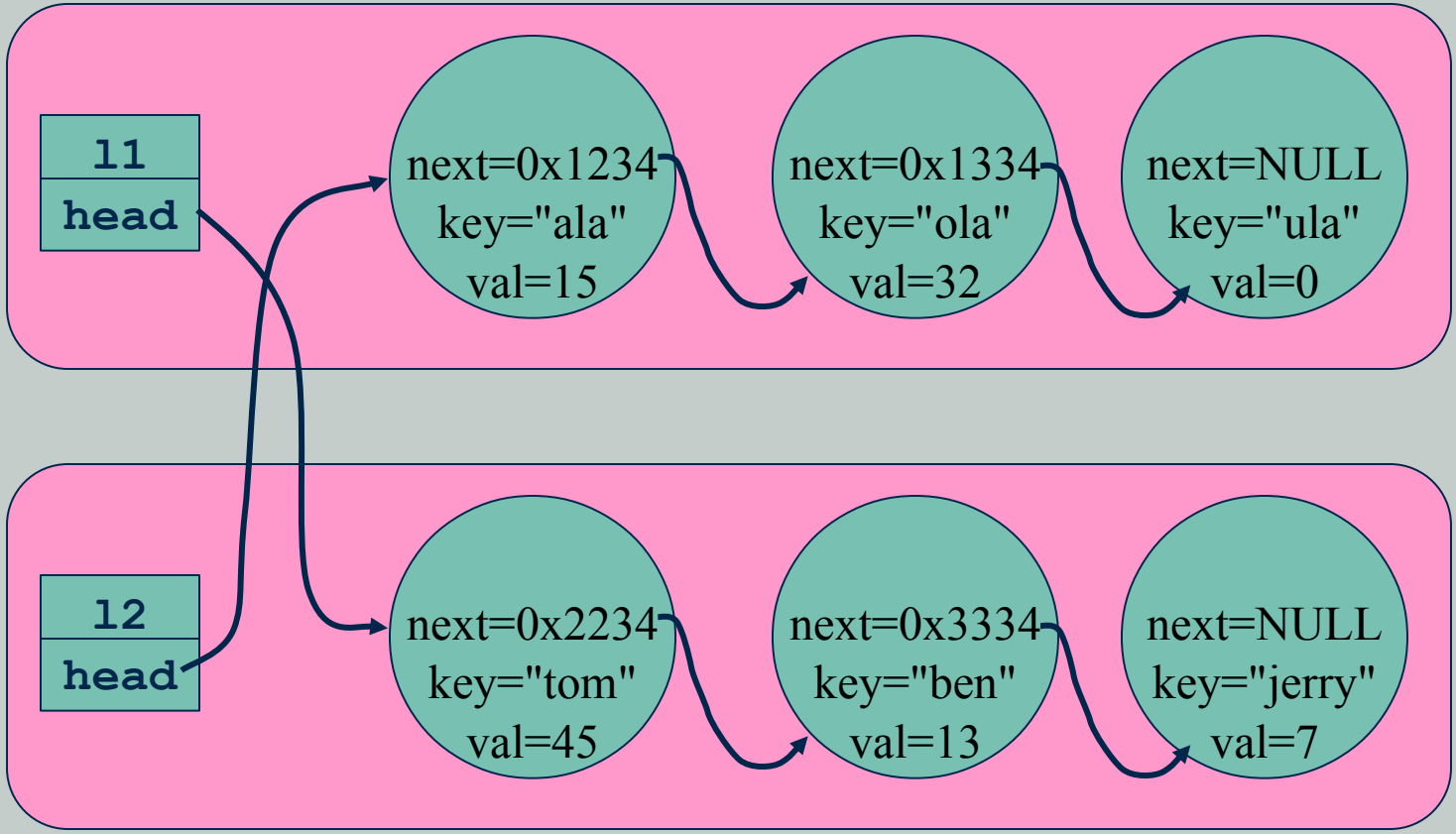



```
assocTab & assocTab::operator= (const assocTab & l)
{
  if (&l == this)
    return *this;
  assocTab t (l);
  swap (t);
  return *this;
}
```



Associative Table - Function *swap*

- The *swap* function swaps only the pointers, so it cannot raise an exception



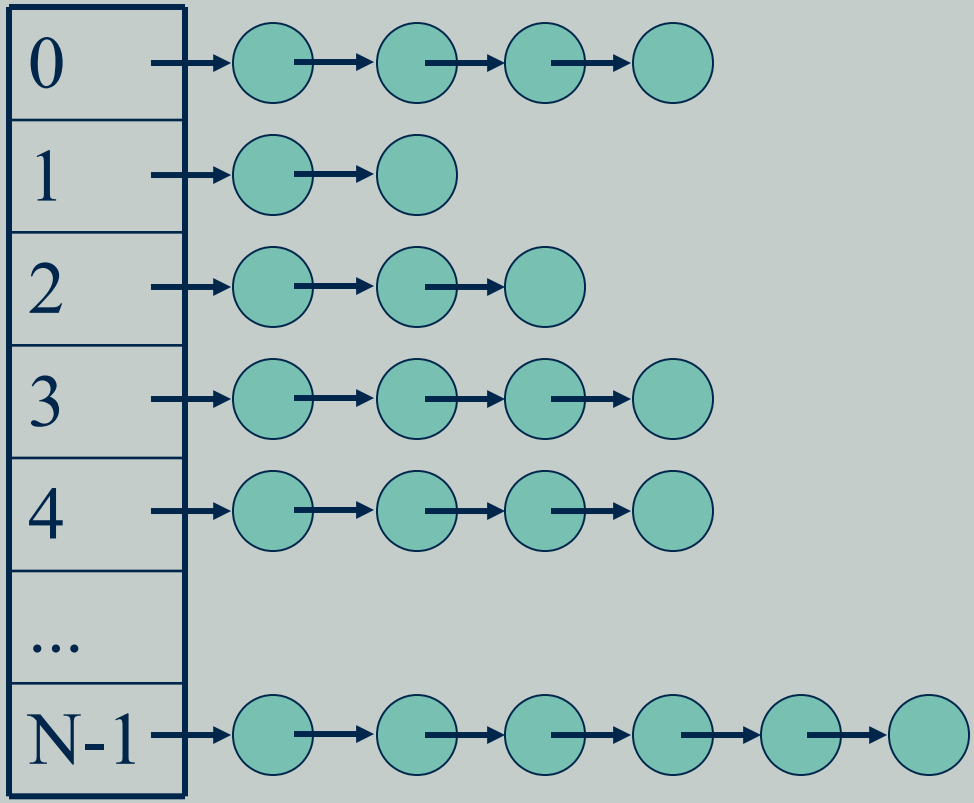
Associative Table - Improved

- The time of search in the associative table implemented as a list is proportional to the number of elements in a table
- Frequently used method to reduce the search time is to use a *hash function*
- Hash function is a function, which maps a character string to the number in range <0...N-1>. The distribution of values should be uniform for all the strings placed in an array
- The choice of appropriate hash function for the given application domain is an art

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

Associative Table with a Hash Function

- Instead of one list, we create N lists
- Every list contains the nodes having the same value of hash function for a key



Ideally, the lenghts of all lists are equal and the search time is reduced n times

Associative Table with a Hash Function

- Defining the class, we will use the previously defined *vector* template

```
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
};
```

Static Class Members

- The static class members are common for all the objects of a given class
- Static member functions do not receive the *this* pointer
- Static member functions can use only the static members of the class
- You can reference the static members either by using any object (using the dot) or by class name (using the scope resolution operator ::)
- Static member variables have to be defined exactly once in a program (not only declared)

```
#include <iostream>
using namespace std;
class example
{
        static int a;
        int v;
public:
        static int getStatic() { return a;};
        static void setStatic(int p) {a=p;};
        example(int p): v(p){};
        int fun(){return v+a;};
};
```

```
int example::a=42;
int main() {
        example e(27);
        example f(47);
        cout << example::getStatic()<<endl;
        f.setStatic(34);
        cout << e.fun()<<endl;
};
```