

Why Containers?

Containers are virtually unavoidable in any semi serious program. You will, for example, need containers for most group projects, as:

Email on the internet

Architectural Walk through

Traffic Simulation

Kitchen Designer

Web based PhD Tutor's Assistant

... ..

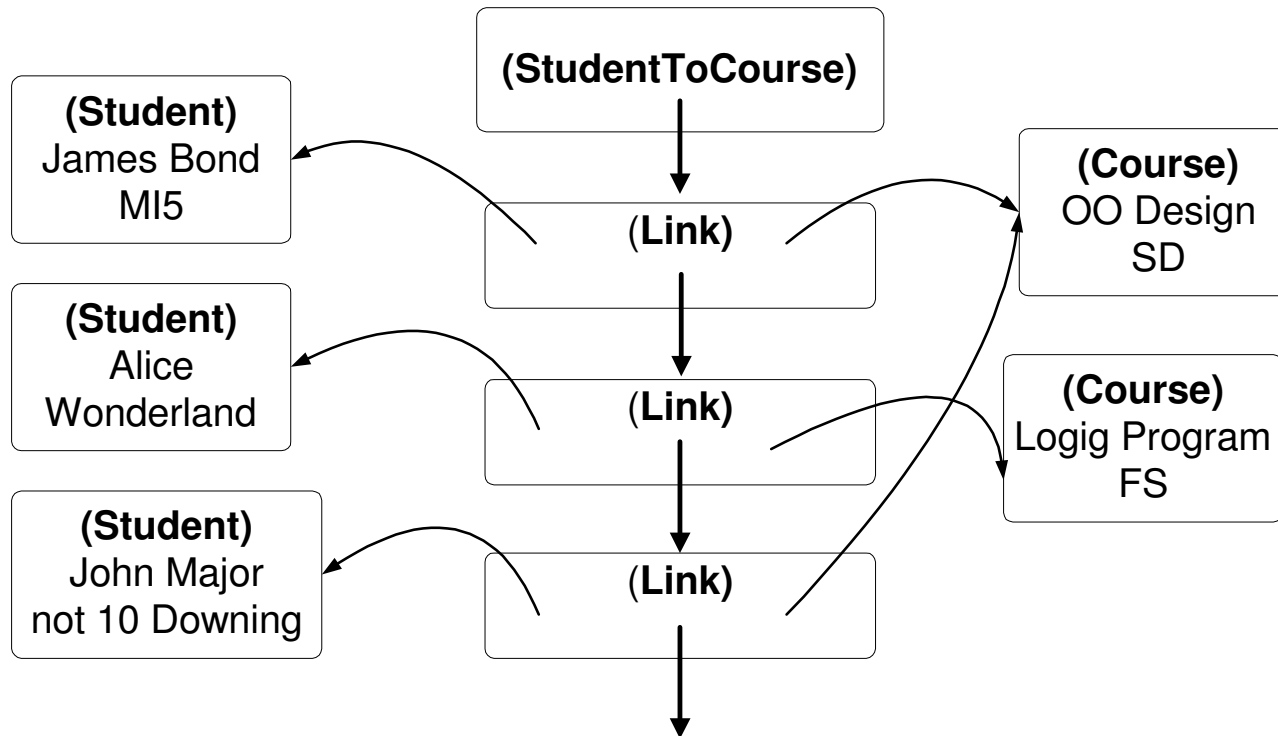
Templates

WS, ch 13

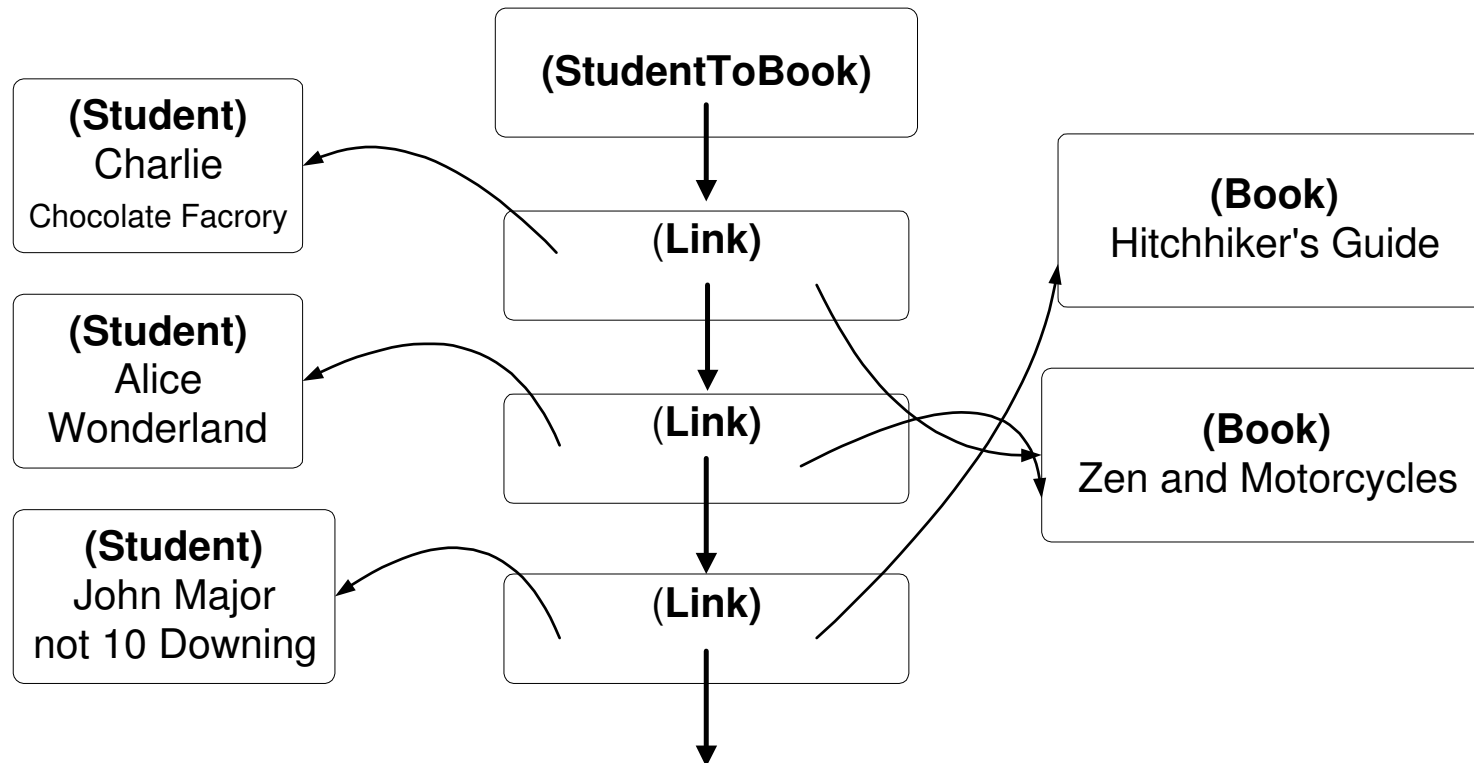
- support *parameterized* ADTs
- usually express generic containers.
- avoid code repetition.
- are a kind of macro (textual substitution) mechanism.
- approach is similar to that of polymorphism in functional programming (Hope, Miranda, Haskell etc.), or Ada generics.
- are only first order.
- are quite wonderful, but not perfect!

Motivation for Templates

Assume classes `Student` (name and address) and `Course` (title and lecturer). Consider class `StudentToCourse`, giving the favourite course of each student, implemented as a linked list of pairs:



Now, consider a class `StudentToBook`, which gives the favourite book of each student. It too can be implemented as a linked list of pairs:



We compare code for `StudentToCourse`, with code for `StudentToBook`.

The classes Student and Course

```
class Student {
public:
    Student(char* aName, char* addr);
    friend ostream& operator<<(ostream&, const Student&);
    static void testMe();    // tests class Student
private:
    char* name;
    char* address;    };

class Course {
public:
    Course(char* aTitle, char* teacher) ;
    friend ostream& operator<<(ostream&, const Course&);
    static void testMe();    // tests class Course
private:
    char* title;
    char* taughtBy;    };
```

The implementations (bodies) are straightforward.

Class StudentToCourse.

```
// a lookup table for each student's favourite course
```

```
class StudentToCourse {  
public:  
    StudentToCourse(){ theList = NULL; };  
    void add(Student* s, Course* c);  
    Course* lookUp(const Student* s);  
    static void testMe();  
private:  
    class Link{  
        Student* theStudent;  
        Course* theCourse;  
        Link* next;  
        Link(Student* s, Course* c)  
            :theStudent(s), theCourse(c) { next = NULL; };  
        friend class StudentToCourse;  
    } ;  
    Link* theList;  
};
```

Note: nested class Link (purely a scoping mechanism).

Question: Why did I not define `lookUp` as

```
Course lookUp(const Student* s);
```

or, as

```
Course* lookUp(const Student s);
```

Design Alternatives for `StudentToCourse` tables

- Can the table be empty? *yes, ~~no~~*
- What if I search a student's preference in empty table?
error, null, ~~default-course~~
- What if I search a student's preference who is not in table?
error, null, ~~default-course~~
- What if I search a student's preference who is in table?
his preference
- What if I enter a preference for a student already in table?
ignore, overwrite

We now continue with the bodies of the functions

```
StudentToCourse::StudentToCourse() { theList = NULL; };
```

```
void add StudentToCourse::(Student* s, Course* c)
{ if ( theList == NULL )
  { theList = new Link(s,c);   return; };
  // find the link with Student s, if any
  // and if found, overwrite the contents
  Link* aList = theList;
  while ( aList->next != NULL )
    { if (s == aList->theStudent ) // found
      {aList->theCourse = c;   return; };
    aList = aList-> next;
  };
  // we now have reached the last link
  if (s == aList->theStudent ) //last link has s
    {aList->theCourse = c; return; }
  else // the last link does not have student s
    { aList->next = new Link(s,c) ;
  return; };
```

```
Course* StudentToCourse::lookUp(const Student* s)
```

```

{ Link *aList = theList;
  while ( aList != NULL )
    { if ( s == aList->theStudent )
      {return aList->theCourse; };
      aList=aList->next; };
return NULL;    };

```

```

void StudentToCourse::testMe()

```

```

{ cout << "*** StudentsCourses::testMe() ***\n";
  Course* LP = new Course("Logic Pondering", "FS" );
  Course* OODP = new Course("OO Cookingg", "SD" );

  Student* Charlie = new Student
    ("Charlie", "The Chocolate Factory");
  Student* Jack = new Student("Jack", "Up the hill");
  Student* Jill = new Student("Jill", "Up the hill");

  StudentToCourse favouriteCourseOf;
  // when list is empty , print the pointer value
  cout << " expect 0, get " <<
    favouriteCourseOf.lookup(Charlie) << "\n";

  // looking up something nonexistant
  favouriteCourseOf.add(Jack, OODP);

```

```

cout << " expect 0, get " <<
    favouriteCourseOf.lookup(Jill) << "\n";

// contains unique entry which looked up
cout << " expect OODP, get " <<
    *favouriteCourseOf.lookup(Jack) << "\n";

// when the linked list contains several entries
favouriteCourseOf.add(Charlie,LP);
favouriteCourseOf.add(Jill,OODP);
cout << " expect LP, get " <<
    *favouriteCourseOf.lookup(Charlie) << "\n";

// when several entries for same key, we expect the
// most recent one
favouriteCourseOf.add(Charlie,OODP);
cout << " expect OODP, get " <<
    *favouriteCourseOf.lookup(Charlie) << "\n";
cout << "* StudentsCourses::testMe() End *\n "; };

```

Class StudentToBook.

The implementation of StudentToBook can be obtained by copying class

StudentToCourse and then replacing Course by Book:

Remember class Book:

```
class Book{
private:
    char* title;
    static int BookCounter;
    int serialNumber;
public:
    Book(char*);
    friend ostream& operator<<(ostream&, const Book&);
};
```

Now for the student to book lookup table.

```

// a lookup table mapping students to Books
class StudentToBook {
public:
    StudentToBook(){ theList = NULL; };
    void add(Student* s, Book* c);
    Book* lookUp(const Student* s);
    static void testMe();
private:
    class Link{
        Student* theStudent;
        Book* theBook;
        Link* next;
        Link(Student* s, Book* c)
        { theStudent=s; theBook= c; next = NULL;};
        friend class StudentToBook;
    };
    Link* theList;
};

StudentToBook::StudentToBook(){ theList = NULL; };

void StudentToBook::add(Student* s, Book* c)
    { if (s==NULL || c ==NULL) return;

```

```

    if ( theList == NULL )
    { theList = new Link(s,c); return; };
    // find the link with Student s, if any
    // if found, overwrite the contents
    Link* aList = theList;
    while ( aList->next != NULL )
    { if (s == aList->theStudent ) // found
      {aList->theBook = c; return; };
      aList = aList-> next; };
    // we now have reached the last link
    if (s == aList->theStudent) // last link has s
      {aList->theBook = c; return; }
    else // last link does not have s
      {aList->next = new Link(s,c) ; return; };
    };

```

```

Book* StudentToBook::lookUp(const Student* s)
{ Link *aList = theList;
  /* etc .. etc .. etc */ };

```

```

void StudentToBook::testMe()
{ cout << "*** StudentsBooks::testMe() ***\n";
  Book* Zen = new Book

```

```

        ("Zen and the art of motorcycle maintenance" );
Book* Hitch = new Book
        ("Hitchhiker's Guide of the Universe" );

Student *Charlie, *Jack, *Jill; // etc ...Jill = new ...
StudentToBook favouriteBookOf;
// when linked list is empty print the pointer value
cout << " expect 0, get " <<
        favouriteBookOf.lookup(Charlie) << "\n";

// when looking up something that has not been entered
favouriteBookOf.add(Jack, Hitch);
cout << " expect 0, get " <<
        favouriteBookOf.lookup(Jill) << "\n";

// etc . . .      etc . . .      etc . . .
};

```

Copying, pasting and modifying is unsatisfactory because we did not express the similarities/differences between StudentToCourse, and StudentToBook.

We want to use templates instead!

- A *class template* specifies how several individual classes may be constructed out of a blueprint. It has the form

```
template< templFormalParam* > class ...
```

The template is the blueprint; it contains the similarities of several classes, the template parameters are the parts where these classes differ.

- Templates may take types, values or functions as formal parameters.
- Formal parameters may be used inside the body of the template.
- An actual class is created through instantiation of a class template. A template class instantiation may appear wherever a type is expected.
- Template instantiation is textual substitution.

Simple Templates for Look Up Tables

We define a template look up table class. It takes two parameters: the type of the contents, and the type of the key

```
// a dictionary of KeyType and ContType
#include <iostream.h>
template <class KeyType, class ContType>
class TableTempl {
    // a lookup up table template
    // stores entities of ContType under keys KeyType
public:
    TableTempl(){ theList = NULL; };
    void add(const KeyType k, const ContType c);
    // enters c under key k into the list
    // overwrites any previous entry under k
    ContType* lookUp(const KeyType k);
    // returns entry under key k if any,
    // returns null otherwise
private:
    /* we see the rest on slide 20 */ };
```

Template Instantiation

When we instantiate the template look up table, we have to pass two type parameters:

```
// template instantiation
typedef TableTempl<int, char> CharTable

/* Therefore, the above is equivalent to
class CharTable {
public:
    CharTable () { ... };
    void add(const int, const char);
    char * lookUp(const int);
private:
    /* ... */ };
*/
```

CharTable is a “normal” class, and can be used as such, eg:

```
CharTable aTbl, AnotherTable;
AnotherTable = aTbl;
```

Obviously, objects of class CharTable have all member functions of class template TableTempl, where ContType is replaced by char

and KeyType is replaced by int.

```
// illegal : aTbl.lookup(OODP)

// when the LinkedList is empty
cout << " expect 0, get " <<
    ( aTbl.lookup(7) ? *aTbl.lookup(7) : '0') << "\n";
// when looking up something that has not been entered
aTbl.add(18, 's');
cout << " expect 0, get " <<
    (aTbl.lookup(7) ? *aTbl.lookup(7) : '0') << "\n";

// linked list contains one entry which is looked up
cout << " expect 's', get " <<
    (aTbl.lookup(18) ? *aTbl.lookup(18) : '0') << "\n";
```

etc, etc...

We may instantiate the template look up table again. Again, we have to pass two type parameters:

```
typedef TableTempl<int, char> anotherCharTable;
```

Another instantiation of the class template:

```
TableTempl<Student*, Course*> favourite;
```

```

/* The above is equivalent to
class AnonClass {
public:
    AnonClass () { ... };
    void add(const Student*, const Course*);
    Course** lookUp(const Student*);
private:
    /* ... */ };
AnonClass favourite;

Course *LP, *OODP; ... // initialize LP, OOP
Student *Charlie, *Jack; ...// initialize Charlie

// looking up something that has not been entered
favourite.add(Jack, OODP);
cout << "expect 0, get " << favourite.lookUp(Jill);
// looking up something that has been entered
cout << "expect OODP, get " << ** (favourite.lookUp(Jack));

```

We need double dereferencing in ` (favourite.lookUp(Jack))`**

because

Note however, that the following is forbidden:

```
TableTempl anotherTable;
```

Template Bodies

Template parameters may be mentioned inside the class template

```
template <class KeyType, class ContType>
class TableTempl {
public:
    TableTempl(){ theList = NULL; };
    void add(const KeyType k, const ContType c);
    ContType* lookUp(const KeyType k);
    static void testMe();
private:
    class Link{
        KeyType theKey;
        ContType theContents;
        Link* next;
        Link(const KeyType k, const ContType c)
        { theKey=k; theContents= c; next = NULL;};
        friend class TableTempl<KeyType, ContType>;
    };
    Link* theList; };
```

The bodies of the functions declared in the class template:

```
template <class KeyType, class ContType>
void TableTempl<KeyType, ContType>
    ::add(const KeyType k, const ContType c)
{ // if the list is empty
  if ( theList == NULL )
  { theList = new Link(k,c); return; };
  // find the link with key k, if any
  // and if found, overwrite the contents
  Link* aList = theList;
  while ( aList->next != NULL )
  { if (k == aList->theKey ) //found link with k
    {aList->theContents = c; return; };
    aList = aList-> next;
  };
  // we now have reached the last link
  if (k == aList->theKey )// last link has k
    {aList->theContents = c; return; }
  else // the last link does not have key k
    { aList->next = new Link(k,c); return; };
```

```

template <class KeyType, class ContType>
ContType* TableTempl<KeyType, ContType>::lookUp(const KeyType k)
{
    Link *aList = theList;
    while ( aList != NULL )
    { if ( k == aList->theKey )
        { return & aList->theContents; };
      aList=aList->next; };
    return NULL;
};

```

```

template <class KeyType, class ContType>
void TableTempl<KeyType, ContType>::testMe()
{
    // template instantiation
    TableTempl<int, char> t1;

    // when the LinkedList is empty
    cout << " expect 0, get " <<
        ( t1.lookUp(7) ? *t1.lookUp(7) : '0') << "\n";

    // when looking up something that has not been entered
    t1.add(18, 's');
    cout << " expect 0, get " <<
        (t1.lookUp(7) ? *t1.lookUp(7) : '0') << "\n";

    // linked list contains one entry which is looked up

```

```

cout << " expect s, get " <<
      (t1.lookup(18) ? *t1.lookup(18) : '0') << "\n";

// linked list contains several entries
t1.add(36, 'd'); t1.add(5, 'e');
char c1 = ( t1.lookup(36) ? *t1.lookup(36) : '0' ) ;
cout << " expect 'd', get " << c1 << "\n";
c1 = ( t1.lookup(18) ? *t1.lookup(18) : '0' ) ;
cout << " expect 's', get " << c1 << "\n";

//several entries for same key, expect the most recent
t1.add(5, 'f'); t1.add(5, 'g');
c1 = ( t1.lookup(5) ? *t1.lookup(5) : '0' ) ;
cout << " expect 'g', get " << c1 << "\n";

};

. . .

```

Note the static member functions, eg

```

// in the main program
TableTempl<Student, Course>::testMe();
// illegal: TableTempl::testMe();

```

Templates – the Rules

- Templates may be class or function templates. They describe a family of types or functions.

- Templates have the form:

```
template< templFormalParam* > declaration
```

- Template arguments may have the form

```
class T           a type expected
```

```
typ_1 v           a constant expression of typ_1 expected
```

```
typ_2 (*f) (argTypes)
```

a pointer to a function with appropriate argument and return type expected

- Instantiation is textual substitution (copy and replace the formal template parameters by the actual template parameters).

- Class templates are instantiated explicitly by

```
templClassName< templActParams >
```

- Member functions of a class template are function templates.
- Function Templates are instantiated implicitly through calls
`templFuncName(funcActParams)`

Only the actual function parameters are passed. The actual template parameters are inferred from the actual function parameters (more later).

- A class template may be the base class of another class template.
- Templates are blueprints: class templates may not be used as types before instantiation.
- Two instantiated template class types are equivalent if they consist of the instantiation of the same template with the same template arguments.

- Some errors detected at template instantiation, or even at call of instantiated template function

```
typedef TableTempl<Student, Course> Trouble;  
Trouble::testMe();  
Trouble table;  
  
Student Ch ("Charlie", "The Chocolate Factory");  
... table.lookup(Ch)
```

causes the error message

```
... ' class Student' does not define binary '==' ...
```

to appear under the template function.

- The error is caused by
- The position of error message is inconvenient, because
- The error message appears at this point because
- In order to avoid this error we could
 - modify the template class, i.e.
 - modify the template argument class, i.e.

Functions and Values as Template Parameters

We augment the previous template by two parameters:.

- v value to return when looking up non-existing key
- eq pointer to a function comparing KeyType entities

```
// project Templates, file LinkListOf2.h
#include <iostream.h>
template <class KeyType, class ContType,
         ContType v, int (*eq)(KeyType&, KeyType&) >
class TableTempl {
public:
    TableTempl(){ theList = NULL; };
    void add(const KeyType k, const ContType c);
    const ContType lookUp(const KeyType k);
    static void testMe();
private:
    class Link{
        KeyType theKey;
        ContType theContents;
        Link* next;
        Link(const KeyType k, const ContType c)
```

```

        theKey(k):theContents(c) { next = NULL; };
    friend class
        TableTempl<KeyType, ContType, v, eq>;
} ;
Link* theList;
};

```

```

template <class KeyType, class ContType,
        ContType v, int (*eq)(KeyType&, KeyType&) >
void TableTempl<KeyType, ContType, v, eq>
::add(const KeyType k, const ContType c)
{ // if the list is empty
    if ( theList == NULL )
        { theList = new Link(k, c); return; };
    // find the link with key k, if any
    // and if found, overwrite the contents
    Link* aList = theList;
    while ( aList->next != NULL )
        { if ((*eq)(k, aList->theKey)) //found
            { aList->theContents = c;
              return; };
        /* etc . . etc . . . etc */ };
}

```

```

template <class KeyType, class ContType,
    ContType v, int (*eq)(KeyType&,KeyType&) >
const ContType
    TableTempl<KeyType, ContType, v, eq>
        ::lookUp(const KeyType k)
    { /* etc . . . etc . . . etc */ };

int eqInts(int i1, int i2){return i1==i2;};

template <class KeyType, class ContType,
    ContType v, int (*eq)(KeyType&,KeyType&) >
void TableTempl<KeyType, ContType,v,eq>::testMe()
{ cout << "* TableTempl::testMe() Start Testing ***\n ";
    TableTempl<int, char, '?', &eqInts> aLL;
    // when the LinkedList is empty
    cout << " expect ?, get " << aLL.lookUp(7) << "\n";
    /* etc . . . etc . . . */ };

// the main program
int eqStudents(const Student* s1, const Student* s2)
    { return strcmp(s1->name,s2->name); };
Course* Trouble = new Course("Trouble", "@@" );
int main()

```

```

{ cout << "*** start testing templates **\n";
  Student Ch = Student("Charlie", "Chocolate Factory");

  typedef TableTempl<Student*, Course*,
                    Trouble, &eqStudents> FavouriteCourses;
  FavouriteCourses::testMe();
  FavouriteCourses favourite;
  Course* LP = new Course("Logic Pondering", "FS" );
  Student* Charlie = new Student("Charlie", "Choco Factory");
  // when the linked list is empty
  cout << " expect Trouble, get " <<
        *favourite.lookup(Charlie) << "\n";
  // looking up something that has not been entered
  favourite.add(Jack, OODP);
  cout << " expect Trouble, get " <<
        *favourite.lookup(Jill) << "\n";
  // etc . . . etc . . . etc

  // several entries for same key, expect the latest
  favourite.add(Charlie, LP); favourite.add(Charlie, OODP);
  cout << " expect OODP, get " <<
        *favourite.lookup(Charlie) << "\n";
  cout << "*** End testing *****\n";   return 0; };

```

Function Templates

- Blueprint for a family of functions.
- Template arguments not explicitly specified when calling a function template.
- No associated class template necessary

```
#include <iostream.h>
template <class T>
T max(T a, T b) { return ( a>b ? a : b ); }
```

```
int i1 = 7; int i2 = 77;
char c1 = 'Z'; char c2 = 'A';
```

```
int main()
{ cout << max (i1,i2);
  // out 77
  cout << max (c1,c2);
  // out Z
  return 0; }
```

Putting it together: Vectors

Arrays are “unsafe” in C++, i.e. array index access not checked. Using templates we can define vectors which provide safe access.

```
template <class T, T errVal>
class Vector{
    T* values;
    int itsSize;
    T theErrorValue;
public:
    Vector(int n)
    { itsSize = n; theErrorValue = errVal;
      if (n<=0)
          { cout << "error\n"; values=NULL; }
        else
          { itsSize = n; values = new T[n];
            int i=0;
            while (i<itsSize) { values[i]=errVal; i++;}; };
    };
```

```

T& operator[](int n)
{ if ( (n<0) || (n>itsSize))
    { cout << "index out of bound\n";
      // best to throw an exception!
      return theErrorValue; }
  else
    { return values[n]; };
};

};

int main()
{ typedef Vector<char, '?'> charVectorType;
  charVectorType aCharVector(14);
  cout << "expect ?, get: " << aCharVector[3] << "\n";
    // out          expect ?, get: ?
  aCharVector[5]='d';
  cout << "expect d, get: " << aCharVector[5] << "\n";
    // out          expect d, get: d
  cout << "expect ?, get: " << aCharVector[43] << "\n";
    // out          index out of bound
    // out          expect ?, get: ?
  return 0;};

```

Templates - Summary

- are macro mechanisms
- useful in avoiding repetition
- good for programming in the large
- very powerful (esp. when combined with inheritance, and function parameters)
- STL : Standard Template Library
- limitations:
 - abysmal error messages
 - first order only

but these limitations have been addressed in more modern oo languages

Parameterized ADTs will soon be part of Java, and are almost part of C#