Class Templates

Not in Java

a char -stack:

```
class stack {
  public:
    explicit stack(int size = 100)
      : max_len(size), top(EMPTY),s(new char[size]){ assert(s != 0); }
    ~stack() { delete []s; }
    void  reset() { top = EMPTY; }
    void  push(char c) { s[++top] = c; }
    char  pop() { return s[top--]; }
    char  top_of()const { return s[top]; }
    bool  empty()const { return top == EMPTY;}
    bool  full()const { return top == max_len - 1;}
  private:
    enum   { EMPTY = -1 };
    char*  s;
    int     max_len;
    int     top;
};
```

---

a double -stack:

```
class stack {
  public:
    explicit stack(int size = 100)
      : max_len(size), top(EMPTY),s(new double[size]){ assert(s != 0); }
    ~stack() { delete []s; }
    void  reset() { top = EMPTY; }
    void  push(double c) { s[++top] = c; }
    double  pop() { return s[top--]; }
    double  top_of()const { return s[top]; }
    bool  empty()const { return top == EMPTY;}
    bool  full()const { return top == max_len - 1;}
  private:
    enum   { EMPTY = -1 };
    double*  s;
    int     max_len;
    int     top;
};
```


Class Templates

a **template** stack implementation:

Template Header
TYPE: template formal parameter.
a type parameter.

```
#include <iostream>
#include <cassert>

template <class TYPE>
class stack {
  public:
    explicit stack(int size = 100)
      : max_len(size), top(EMPTY),s(new TYPE [size]){assert(s != 0);}
    ~stack() { delete []s; }
    void  reset() { top = EMPTY; }
    void  push(TYPE c) { s[++top] = c; }
    TYPE pop() { return s[top--]; }
    TYPE top_of()const { return s[top]; }
    bool  empty()const { return top == EMPTY;}
    bool  full()const { return top == max_len - 1;}
  private:
    enum   { EMPTY = -1 };
    TYPE * s;
    int    max_len;
    int    top;
};
```

The compiler generates a new specific stack class for each different instantiation of the stack class template.

Instantiate stack Class-Template

Declaration:

```
stack<char>   stck_ch;      // 100 char stack
```

Class-name/type

```
stack<double>  stck_d(200);     // a 200 double stack
stack<int> * istckp= new stack<int>(50); // a dyn allocated int stack
```

Class Templates

Separation of class declaration and implementation:

```
template <class TYPE>
class stack {
  public:
    explicit stack(int size = 100)
      : max_len(size), top(EMPTY),s(new TYPE [size]){assert(s != 0);}
    ~stack() { delete []s; }
    void  reset() { top = EMPTY; }

    void  push(TYPE c);
    TYPE pop();

    TYPE top_of()const;
    bool  empty()const;
    bool  full()const;

  private:
    enum   { EMPTY = -1 };
    TYPE *  s;
    int    max_len;
    int    top;
};
```

```
template <class TYPE>
  void stack<TYPE>::push(TYPE c){ s[++top]= c;}

template <class TYPE>
  TYPE stack<TYPE>::pop(){ return s[top--];}

template <class TYPE>
  TYPE stack<TYPE>::top_of() const { return s[top];}

template <class TYPE>
  bool stack<TYPE>::empty() const { return top==EMPTY;}

template <class TYPE>
  bool stack<TYPE>::full() const {return top== max_len - 1;}
```


Function Templates

```
template<class TYPE>
  void copy(TYPE a[], TYPE b[], int n)
  {  for (int i = 0; i < n; ++i)    a[i] = b[i];
  }

template<class TYPE>
  void print(TYPE a[], int n)
  {  cout << "\nNEW PRINT =";
     for (int i = 0; i < n; ++i)    cout << a[i] << " ";
     cout << endl;
  }
```

```
int main()
{  double    f1[50],  f2[50];
   char      c1[25], c2[50];
   int       i1[75],  i2[75];
   char* ptr1, *ptr2;
```

	implicit function template instantiation
print(f1, 20);	print<double>
print(f2, 20);	
print(i1, 20);	print<int>
print(i2, 20);	
print(c1, 20);	print<char>
print(c2, 20);	
copy(f1, f2, 50);	copy<double>
copy(c1, c2, 10);	copy<char>
copy(i1, i2, 40);	copy<int>
copy(ptr1, ptr2, 100);	copy<char*>
copy(i1, f2, 50);	illegal, compiler cannot instantiate a matching funct
copy(ptr1, f2, 50);	illegal

Templates, Multiple and Default Arguments

```
template<class T1, class T2, int maxsize=100>
class Map
{  public:
      Map(T2 val0):top(0){table[0].val= val0;}
      void insert(T1,T2);
      T2 lookup(T1) const;

   private:
      struct Entry{T1 key; T2 val;};
      Entry table[maxsize];
      int top;
};
```

```
template<class T1, class T2, int maxsize>
void Map<T1,T2,maxsize>::insert(T1 key, T2 val)
{  Entry& e= table[++top];
   e.key=key; e.val=val;
}
```

```
template<class T1, class T2, int maxsize>
T2 Map<T1,T2,maxsize>::lookup(T1 key)
{  for(int i=top; i>0;i--)
   if(table[i].key==key){return table[i].val ;}

   return table[0].val;
}
```

= = as defined for T1

```
int main()
{  Map<int,double,10> idmap(0.0);
   idmap.insert(5, 5.5); idmap.insert(23, 100.45);
   cout << idmap.lookup(23) << ", "
         << idmap.lookup(5) <<  ", "<< idmap.lookup(10) <<  ", " << endl;

   Map<double,char*> icmap("no-val");
   icmap.insert(5,"jensen");
   cout  << icmap.lookup(33)<< ", "
         << icmap.lookup(5)<< endl;
}
```

output:
100.45, 5.5, 0,
no-val, jensen

Parameterising the Class Vector

```
#include <iostream>
#include <cassert>
using namespace std;

template <class T>
class vector {
  public:
    explicit vector(int n = 100);       //default constructor !!
    vector(const vector<T>& v);   //copy constructor
    vector(const T a[], int n);          //copy an array
    ~vector() { delete []p; }

    typedef T* iterator;
    iterator begin(){ return p;}
    iterator end(){ return p + size;}

    T& operator[](int i);                //range checked indexing
    vector<T>& operator=(const vector<T>& v);  //deep assignment

  private:
    T* p;                 //base pointer
    int size;             //number of elements
};

int main()
{ vector<double> v(5);
  vector<double>::iterator p ; // CLASS vector<double>'s iterator type
  int i = 0;

  for (p = v.begin() ; p != v.end(); ++p) *p = 1.5 + i++;
  do { --p;
        cout << *p << " , ";
     } while (p != v.begin());
  cout << endl;
}
```

Parameterising the Class Vector

```cpp
template <class T>
vector<T>::vector(int n = 100): size(n)
{  assert(n > 0);
   p = new T[size];
   assert(p != 0);
}
```

default constructor,
default because argument is optional

```cpp
template <class T>
vector<T>::vector(const vector<T>& v)
{  size = v.size;
   p = new T[size];
   assert(p != 0);
   for (int i = 0; i < size; ++i) p[i] = v.p[i];
}
```

copy constructor

deep copying

```cpp
template <class T>
vector<T>::vector(const T a[], int n)
{  assert(n > 0);
   size = n;
   p = new T[size];
   assert(p != 0);
   for (int i = 0; i < size; ++i)
   p[i] = a[i];
}
```

constructor:
initialise new vector from array elements

```cpp
template <class T>
T& vector<T>::operator[](int i)
{  assert (i >= 0 && i < size );
   return p[i];
}
```

subscript with range check

```cpp
template <class T>
vector<T>& vector<T>::operator=(const vector<T>& v)
{  assert(v.size == size);
   for (int i = 0; i < size; ++i) p[i] = v.p[i];
   return *this;
}
```

redefine =

deep assignment !

Class Vector Template, File Organisation

File: vector.h

```cpp
include <iostream>
#include <cassert>
using namespace std;

template <class T>
class vector {
   public:
      explicit vector(int n = 100);
      vector(const vector<T>& v);
      typedef T* iterator;
      iterator begin(){ return p;}
      …

   private:
      T* p;
      int size;
};

template <class T>
vector<T>::vector(int n = 100): size(n)
{   assert(n > 0); p = new T[size];
    assert(p != 0);
}

...

template <class T>
vector<T>& vector<T>::operator=(const
vector<T>& v)
{   assert(v.size == size);
    for (int i = 0; i < size; ++i) p[i] = v.p[i];
    return *this;
}
```

The Vector template declaration

AND

the template implementation

must all be available to the compiler when template instantiations are constructed:

File: vector.cpp

```cpp
#include <iostream>
#include "vector.h"

int main()
{   vector<double> v(5);
    vector<double>::iterator p ;
    int i = 0;

    for (p = v.begin() ; p != v.end(); ++p)
    *p = 1.5 + i++;
    do   {  --p;
            cout << *p << " , ";
         } while (p != v.begin());
    cout << endl;
    int look; cin >> look;
}
```