

Object Oriented Programming

Templates

July 25, 2009 Dr. Mordo Shalom

Templates

- The last step towards MACRO avoidance.
- #define A 5 → const int a=5;
- #define max(a,b) → inline int max(int,int) {...}
- Is the above solution enough?
- No: We have still to write:
 - max(double, double)
 - max(float, float)
 - max(float,int)
- Q: How many of them?
 - A: $36 = \{int, short, long, float, double, char\}^2$
- Q: May be more ?

Templates

- In C++ we can write type independent code.
- This is done by letting the type be a "parameter" in a special way.
- We can write **template functions**, that specify the type as a "parameter".
- We can also write entire classes as **template classes**.
- These templates are used by the compiler to generate the actual code whenever needed.

Further examples

- A generic linked list class that may contain elements of the same type where the latter might be any type.
- Stack, Queue, Tree, Hash Table, ...
- A generic quick sort function.
- bubble sort, ...
- A generic swap function.

Templates - syntax

- template** <class T>
- template** <typename T>
- T can be any identifier.
- T is called the **template parameter**.

Template function example: swap

```

template <typename T>
void swap(T& a, T& b)
{
    T temp = a;
    a = b;
    b = temp;
}

```

The function is good for every type T having
A copy c'tor and an assignment operator.

```

int x = 5, y = 10;
swap(x,y);

String s1, s2;
...
swap(s1,s2);

```

Template errors

- What happens if the actual type does not satisfy the conditions?
- A compile error in the client program.
- A deviation from modularity:
 - The client programmer has to understand the reason by inspecting the code of the template.

The java approach

- templates are called **generics**.
- the underlying mechanism is called **erasure**.
- the template is compiled as if T was void *.
- The client program does not have the source.
- It has only the "include file".
- The include file may specify that, for instance T has to implement some set of interfaces.
- If this does not happen the compiler detects the error, w/o having access to the implementation of the generic code.

*

Example – basic array operations

```
template <class T>
void CopyArray(T a[], T b[], size_t size)
{
    for (int i=0; i < size; ++i)
        a[i] = b[i];
}
```

operator =

```
template <class T>
void PrintArray(T *a, size_t n)
{
    for (int i = 0; i < n; ++i)
        cout << a[i] << endl;
    cout << "\n";
}
```

operator <<

Example – Quick Sort

```
template <class Comparable>
void quicksort(Comparable a[], int left, int right){
    if (left >= right) // array contains less than 2 elements
        return;

    int last = left;
    for (int i = left+1; i <= right; ++i)
        if (a[i] < a[left]) // < needs to be defined for T
            Swap(a[++last], a[i]);
    Swap(a[left], a[last]);
    quicksort(a, left, last-1);
    quicksort(a, last+1, right);
}
```

Just a visual clue, no significance to the compiler

Example – client program

```
int main( )
{
    string names[array_size];
    int numbers[array_size];
    cout << "Enter " << array_size << " names: " << endl;
    for (int i=0; i < array_size; ++i)
        cin >> names[i];
    quicksort(names, 0, array_size-1);
    PrintArray(names, array_size);

    // now with numbers
    cout << "Enter " << array_size << " integers: " << endl;
    for (int i=0; i < array_size; ++i)
        cin >> numbers[i];
    quicksort(numbers, 0, array_size-1);
    PrintArray(numbers, array_size);
    return 0;
}
```

Multiple parameters

```
template <class T1, class T2>
```

```
template <class key, class value>
void SortByKey(key a[], value b[], int size);
```

Modularization

- We already said that the compiler has to access the source of the template when compiling client program.
- Therefore the source must be "#include" d in the client program. Alternatives:
 - prototype + implementation in same .h file
 - prototype in .h file that #include s implementation. (Best practice)

Template class example

```
template<class T>
class Vect
{
public:
    //constructors and destructor
    Vect(); //create a size 10 array
    Vect(size_t n); //create a size n array
    Vect(const T a[], size_t n); //initialization by array
    Vect(const Vect<T>& v); //copy constructor
    ~Vect() { delete []p; }
    int Size() const {return size;}
    Vect<T>& operator=(const Vect<T>& v);
    T& operator[](int i) const; //obtain range checked element
private:
    enum{DEFAULTSIZE = 10};
    T *p; //base pointer
    size_t size; //number of elements
};
#include "Vect_imp.h"
```

Vect example – c'tors

```
// file: vect_imp.h
// Implementation of a safe array template Vect
template <class T>
Vect<T>::Vect()//default c'tor
:size(DEFAULTSIZE) {
    p = new T[size];
    if(p==NULL) {
        cout<<"Unable to allocate memory"<<endl;
        exit(1);
    }
}

template <class T>
Vect<T>::Vect(size_t n)
:size(n) {
    p = new T[size];
    if(p==NULL) {
        cout<<"Unable to allocate memory"<<endl;
        exit(1);
    }
}
```

Vect example – more c'tors

```
template <class T>
Vect<T>::Vect(const T a[], size_t n) //initialization by array
:size(n) {
    p = new T[size];
    if(p==NULL)
    {
        cout<<"Unable to allocate memory"<<endl;
        exit(1);
    }
    for(int i=0; i<size; ++i)
        p[i] = a[i];
}

template <class T>
Vect<T>::Vect(const Vect<T>& v) { //copy constructor
    p = NULL; // so it can be deleted
    *this = v; // uses overloaded operator=
}
```

Vect example – operator=

```
template <class T>
Vect<T>& Vect<T>::operator=(const Vect<T>& v) {
    if(&v==this)
        return *this;
    delete []p;
    size = v.size;
    p = new T[size];
    if(p==NULL) {
        cout<<"Unable to allocate memory"<<endl;
        exit(1);
    }
    for(int i=0; i<size; ++i)
        p[i] = v.p[i];
    return *this;
}
```

Vect example – operator[]

```
template <class T>
inline T& Vect<T>::operator[](int i) const
{
    if(i<0 || i>=size)
    {
        cerr<<"illegal Vect index: "<<i<<"\n";
        exit(1);
    }
    return p[i];
}
```

Vect example – I/O operators

```

template <class T>
ostream& operator<<(ostream& out, const Vect<T>& v) {
    for(int i=0; i<v.Size(); ++i) {
        out<< v[i]<<endl;
    }
    return out;
}

template <class T>
istream& operator>>(istream& in, Vect<T>& v) {
    for(int i=0; i<v.Size(); ++i) {
        in>>v[i];
    }
    return in;
}
    
```

Vect example - quicksort

```

template <class T>
void Swap(T& a, T& b);

template <class T>
void quicksort(Vect<T>& a, int left, int right) {
    if (left >= right) // array contains less than 2 elements
        return;
    Swap(a[left], a[(left + right)/2]);
    int last = left;
    for (int i = left+1; i <= right; ++i)
        if (a[i] < a[left])
            Swap(a[++last], a[i]);
    Swap(a[left], a[last]);
    quicksort(a, left, last-1);
    quicksort(a, last+1, right);
}
    
```

Vect example – client program

```

int main()
{
    size_t size;
    cout<<"Input array size: ";
    cin>>size;
    Vect<Student> v(size);
    cin>>v;

    quicksort(v,0,size-1);

    cout<<endl;
    cout<<v;
    cout<<endl;
    return 0;
}
    
```

STL

- The Standard Template Library (STL) contains implementations of very common containers. (A subject of another lecture.)
- For instance STL contains a vector class that is similar to our Vect example.
- The following slide shows a sample client code for it.

Sample client for vector

```

#include <vector>
#include <algorithm> // for sort
#include <iostream>
using namespace std;
#include "student.h"

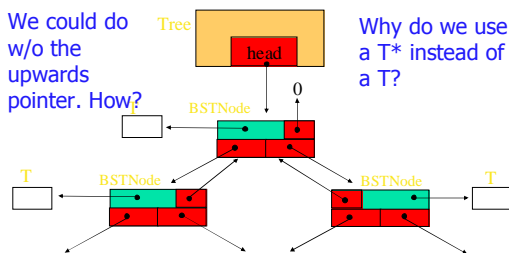
typedef vector<Student> StuVect;

int main() {
    size_t size;
    cout<<"Input array size: ";
    cin>>size;
    StuVect v(size);
    for(int i=0;i<size;++i) cin>>v[i];
    StuVect::iterator startv = v.begin(); // first element of v
    StuVect::iterator endv = v.end(); // last element of v
    sort(startv, endv);
    cout<<endl;
    for (int i=0;i<size;++i) cout<<v[i];
    cout<<endl;
    return 0;
}
    
```

Binary Search Tree

We could do w/o the upwards pointer. How?

Why do we use a T* instead of a T?



```

template<class T>
class BstNode
{
    BstNode<T> *left, *right, *parent;
    T* pData;
    int item_id;
    BstNode(const BstNode<T>&);
    BstNode<T>& operator=(const BstNode<T>&);
public:
    BstNode():left(0),right(0),parent(0),
    item_id(0),pData(new T){}
    BstNode(T* pd, int id):left(0),right(0),parent(0),item_id(id)
    {pData = new T; *pData = *pd;}
    ~BstNode(){delete left; delete right; delete pData;}
}

```

C'tor is private

```

    BstNode<T>* getLeft(){return left;}
    BstNode<T>* getRight(){return right;}
    BstNode<T>* getParent(){return parent;}
    T* getData(){return pData;}
    int getID(){return item_id;}
    void setID(int id){item_id=id;}
    void setLeft(BstNode<T>* l){left=l;}
    void setRight(BstNode<T>* r){right=r;}
    void setParent(BstNode<T>* p){parent=p;}
    void setData(const T& d){*pData = d;}
    void insert(BstNode<T>* newNode);
    BstNode<T>* Find(const T& d);
};

```

Externally implemented functions

```

template <class T>
void BstNode<T>::insert(BstNode<T>* newNode){
    if>(*newNode->pData > *pData){
        if(right){
            right->insert(newNode);
            return;
        }
        else{
            right = newNode;
            right->setParent(this);
        }
    }
    else{
        if(left){
            left->insert(newNode);
            return;
        }
        else{
            left = newNode;
            left->setParent(this);
        }
    }
}

```

Recursive call

```

template <class T>
BstNode<T>* BstNode<T>::Find(const T& d)
{
    if(d == *pData)
        return this;
    else if(d > *pData)
        return right?right->Find(d):NULL;
    else
        return left?left->Find(d):NULL;
}

```

Recursion again

```

template<class T>
class Tree
{
    BstNode<T>* head;
    int counter; // counts the nodes in the tree
    int id_counter; // counts nodes, including those deleted
    Tree(const Tree<T>& t);
    Tree<T> operator=(const Tree<T>& t);
public:
    Tree():head(0),counter(0),id_counter(0){}
    ~Tree(){delete head;}
    int GetID_counter(){return id_counter;}
    int GetCounter(){return counter;}
    BstNode<T>* GetHead(){return head;}
    void insert(T* pData);
    bool Delete(const T& d); // delete a node
    void Delete(); // delete the whole tree
};

```

Externally implemented

```

template<class T>
void Tree<T>::insert(T* pData)
{
    counter++;
    id_counter++;

    BstNode<T>* newNode = new BstNode<T>(pData, id_counter);
    if(head)
        head->insert(newNode);
    else
        head = newNode;
}

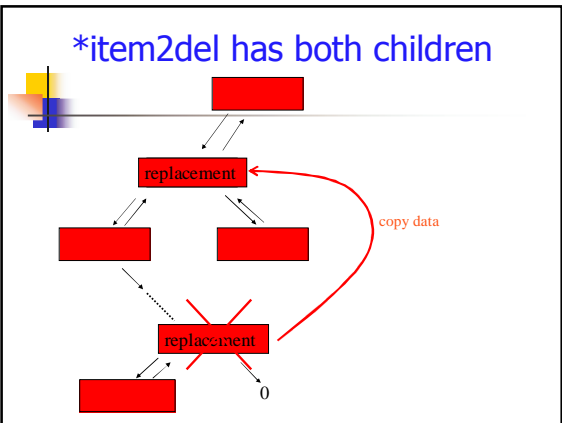
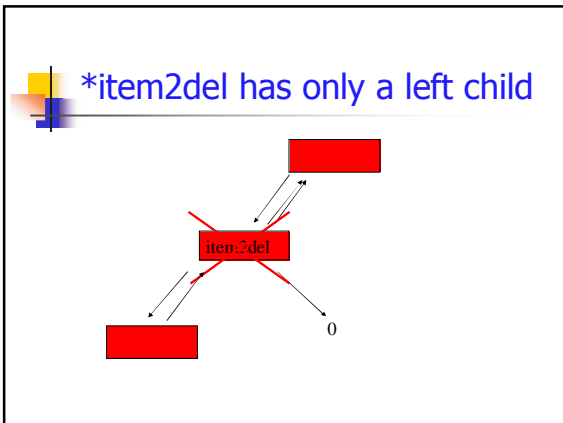
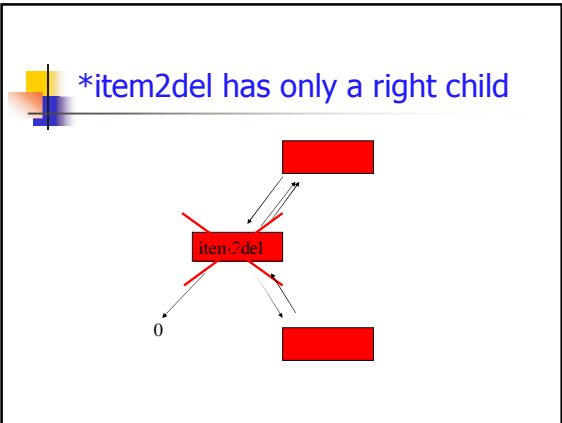
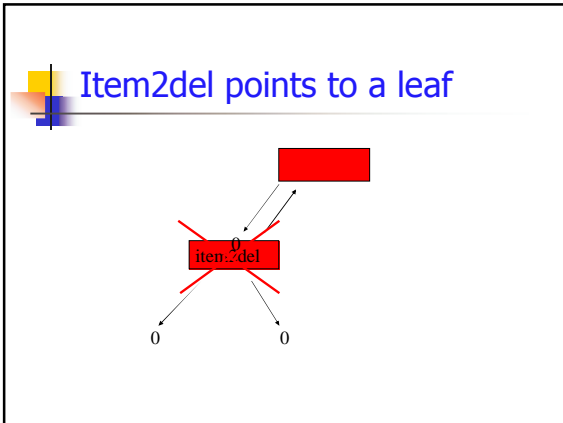
```

```

// delete the whole tree
template<class T>
void Tree<T>::Delete()
{
    delete head; // initiates recursion
    head = 0;
    counter = id_counter = 0;
}

```

- ### Tree::Delete(const T&)
- If the tree is empty, nothing to delete → return false.
 - Invoke find to check whether the node exists. If not → return false.
 - Otherwise item2del contains the pointer to the node to be deleted.
 - There are 4 cases.



Tree::Delete(const T&)

```
// Delete a node in the tree
template<class T>
bool Tree<T>::Delete(const T& d) {
    if(!head)
        return false;
    BstNode<T>* item2del = head->Find(d);
    if(item2del == NULL)
        return false;
    counter--;

    if(item2del->getLeft() == 0) { // left son is empty
        if(item2del->getRight() == 0) { // a leaf
            if(item2del->getParent() != 0) { // not head
                if (item2del->getParent()->getRight() == item2del)
                    item2del->getParent()->setRight(0);
                else
                    item2del->getParent()->setLeft(0);
            }
            else // delete head
                head = 0;
        }
    }
}
```

```
else // not a leaf
{
    if(item2del->getParent() != 0) // not head
    {
        if (item2del->getParent()->getRight() == item2del)
            item2del->getParent()->setRight(item2del->getRight());
        else
            item2del->getParent()->setLeft(item2del->getRight());
        item2del->getRight()->setParent(item2del->getParent());
    }
    else // delete head
    {
        head = item2del->getRight();
        item2del->getRight()->setParent(0);
    }
}
```

```
else if(item2del->getRight() == 0) // right child empty but left son not
{
    if(item2del->getParent() != 0) // not head
    {
        if (item2del->getParent()->getRight() == item2del)
            item2del->getParent()->setRight(item2del->getLeft());
        else
            item2del->getParent()->setLeft(item2del->getLeft());
        if(item2del->getLeft() != 0)
            item2del->getLeft()->setParent(item2del->getParent());
    }
    else // delete head
    {
        head = item2del->getLeft();
        if(item2del->getLeft() != 0)
            item2del->getLeft()->setParent(0);
    }
}
```

```
else // both sons are not empty
{
    BstNode<T>* replacement = item2del->getLeft();
    BstNode<T>* checkReplacement = replacement->getRight();

    while(checkReplacement)
    {
        replacement = checkReplacement;
        checkReplacement = checkReplacement->getRight();
    }
    // replacement is the node to replace item2del
    // taking care of right son
    item2del->setData(*replacement->getData());
    item2del->setID(replacement->getID());
    // replacement will be deleted
    item2del=replacement;
    // proceed as in the case where the right son is empty
    if (item2del->getParent()->getRight() == item2del)
        item2del->getParent()->setRight(item2del->getLeft());
    else
        item2del->getParent()->setLeft(item2del->getLeft());
    if(item2del->getLeft() != 0)
        item2del->getLeft()->setParent(item2del->getParent());
}
```

```
item2del->setLeft(0); // so it can be deleted
item2del->setRight(0); // without ruining its sons
delete item2del;
return true;
}
```

operator<<

```
template <class T>
ostream& operator<<(ostream& out, Tree<T>& t)
{
    if(t.GetCounter() == 0)
        out<<"There are no elements in the tree\n"<<endl;
    else
    {
        if(t.GetCounter() == 1)
            out<<"There is 1 element ";
        else
            out<<"There are "<<t.GetCounter()<<" elements ";
        out<<"in the tree:"<<endl;
        out<<t.GetHead()<<endl;
    }
    return out;
}
```

operator<<

```
template <class T>
ostream& operator<<(ostream& out, BstNode<T>& n)
{
    if(n.getLeft())
        out<<"n.getLeft()";
    out<<"\nId = "<<n.getID()<<"\n";
    out<<"\nData = "<<n.getData();
    if(n.getRight())
        out<<"n.getRight()";
    return out;
}
```

Template class specialization

- Sometimes we want a template class to be implemented differently for specific types.
- For instance, we would like that comparisons between char * types use strcmp() instead of the normal template that uses <.
- In this case we have write a specialized template class as follows:

```
template<>
class BstNode<char*>
{
    //class definition here
};
```

Template class specialization

- There is no inheritance relationship between the original and the specialized class (code duplication).
- It is actually a brand new class. So, why not write it as a normal class?
 - The compiler will not allow a template class and a non-template class to have the same name.
 - Transparent to the client programs.

Method templates

- A method of a class may have parameters that are not part of the template parameters.
- Problem:

```
Vect<int> v1(size);
cin >> v1;
Vect<double> v2 = v1; // error
```

Method Templates – cont'd

```
template<class T>
class Vect {
public:
    Vect(); //create a size 10 array
    Vect(size_t n); //create a size n array
    Vect(const T a[], size_t n); //initialization by array
    Vect(const Vect<T>& v); //copy constructor

    template <class E>
    Vect(const Vect<E>& v); //template copy const; Additional template

    ~Vect() { delete []p; }
    int Size() const {return size;}
    Vect<T>& operator=(const Vect<T>& v);

    template <class E>
    Vect<T>& operator=(const Vect<E>& v);

    T& operator[](int i) const; //obtain range checked element
private:
    enum{DEFAULTSIZE = 10};
    T *p; //base pointer
    size_t size; //number of elements
};
```

Method Templates – cont'd

```
template <class T>
template <class E>
Vect<T>::Vect(const Vect<E>& v) //copy constructor
{
    p = NULL; // so it can be deleted
    *this = v; // uses overloaded operator=
}

template <class T>
template <class E>
Vect<T>& Vect<T>::operator=(const Vect<E>& v)
{
    delete []p;
    size = v.Size();
    p = new T[size];
    // should check if allocation succeeded
    for(int i=0; i<size; ++i)
        p[i] = v[i];
    return *this;
}
```

Template Parameters

There are three types of Template parameters:

- Type parameters (already covered)
- Template template parameters (we will not cover)
- Non-type parameters

Default template parameters

```
#include <vector>
using std::vector;
template <typename T, typename Container = vector<T> >
```

The space is compulsory

Default parameter

```
Grid<int> myIntGrid2;
```

*

Non-type template parameters

- A template may have a parameter which is not a type, but a value.
- The value has to have either an integral type (i.e. int, short, long, bool, char, enum) or a pointer or a reference.

*

Example

- We can define a grid type receiving the width and the height as (non-type) template parameters.

```
template <typename T, int WIDTH, int HEIGHT>
class Grid
{
public:
    void setElementAt(int x, int y, const T& inElem);
    T& getElementAt(int x, int y);
    const T& getElementAt(int x, int y) const;
    int getHeight() const { return HEIGHT; }
    int getWidth() const { return WIDTH; }
protected:
    T mCells[WIDTH][HEIGHT];
};
```

nontype parameters

An automatic array

*

Non-type Template Parameters

```
template <typename T, int WIDTH, int HEIGHT>
void Grid<T, WIDTH, HEIGHT>::setElementAt(int x, int y,
const T& inElem)
{
    mCells[x][y] = inElem;
}
template <typename T, int WIDTH, int HEIGHT>
T& Grid<T, WIDTH, HEIGHT>::getElementAt(int x, int y)
{
    return (mCells[x][y]);
}
template <typename T, int WIDTH, int HEIGHT>
const T& Grid<T, WIDTH, HEIGHT>::getElementAt(int x, int
y) const
{
    return (mCells[x][y]);
}
```

*

Non-type Template Parameters

```
int main()
{
    Grid<int, 7, 5> myGrid;
    Grid<int, 7, 5> anotherGrid;
    myGrid.setElementAt(2, 3, 45);
    anotherGrid = myGrid;
    cout << anotherGrid.getElementAt(2, 3);
    return (0);
}
```

nontype parameters

It will work only if sizes are same. Why?

Q: How we can define copying of Grids of different sizes ?

A: Method templates ?

*

Non-type Template Parameters

method templates

```
template <typename T, int WIDTH = 10, int HEIGHT = 10>
class Grid
{
public:
    Grid() {}

    template <typename E, int WIDTH2, int HEIGHT2>
    Grid(const Grid<E, WIDTH2, HEIGHT2>& src);

    template <typename E, int WIDTH2, int HEIGHT2>
    Grid<T, WIDTH, HEIGHT> operator=(const Grid<E, WIDTH2, HEIGHT2>& rhs);

    void setElementAt(int x, int y, const T& inElem);
    T& getElementAt(int x, int y);
    const T& getElementAt(int x, int y) const;
    int getHeight() const { return HEIGHT; }
    int getWidth() const { return WIDTH; }

protected:
    template <typename E, int WIDTH2, int HEIGHT2>
    void copyFrom(const Grid<E, WIDTH2, HEIGHT2>& src);

    T mCells[WIDTH][HEIGHT];
};
```

Annotations: "Default parameters" points to the default values 10 for WIDTH and HEIGHT. "Method parameters" points to the parameters in the method signatures.

Non-type Template Parameters

method templates

```
template <typename T, int WIDTH, int HEIGHT>
template <typename E, int WIDTH2, int HEIGHT2>
Grid<T, WIDTH, HEIGHT>::Grid(const Grid<E, WIDTH2, HEIGHT2>& src)
{
    copyFrom(src);
}

template <typename T, int WIDTH, int HEIGHT>
template <typename E, int WIDTH2, int HEIGHT2>
Grid<T, WIDTH, HEIGHT> operator=(const Grid<E, WIDTH2, HEIGHT2>& rhs)
{
    // no need to check for self-assignment because this version of
    // assignment is never called when T and E are the same
    // No need to free any memory first
    copyFrom(rhs);
    return (*this);
}
```

Non-type Template Parameters

method templates

```
template <typename T, int WIDTH, int HEIGHT>
template <typename E, int WIDTH2, int HEIGHT2>
void Grid<T, WIDTH, HEIGHT>::copyFrom(const Grid<E, WIDTH2, HEIGHT2>& src)
{
    int i, j;
    for (i = 0; i < WIDTH; i++) {
        for (j = 0; j < HEIGHT; j++) {
            if (i < WIDTH2 && j < HEIGHT2) {
                mCells[i][j] = src.getElementAt(i, j);
            } else {
                mCells[i][j] = T();
            }
        }
    }
}
```

Annotation: "T()" points to the default constructor call for non-template parameters.

Non-type template parameters

- They can be also:
 - of generic type
 - of pointer or reference type
- We do not cover them.