

C++ TEMPLATES

Problem Solving with Computers-II

C++

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

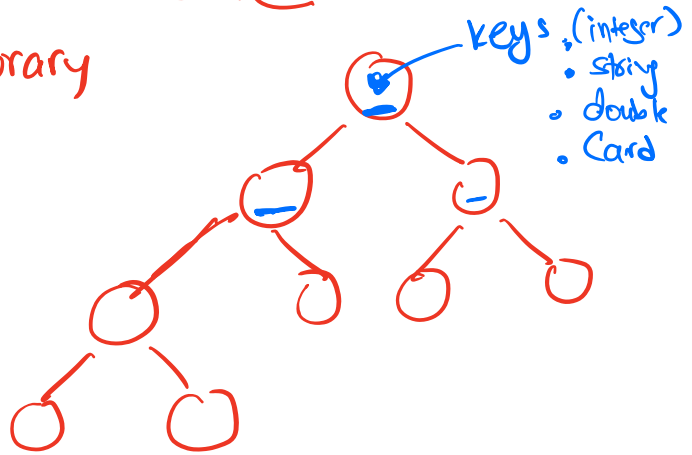
int main(){
    cout<<"Hola Facebook!";
    return 0;
}
```

set: Balanced BST ←

height $\rightarrow H = O(\log n)$ ← number of keys

Standard Template Library

(STL)
Vector
list
set

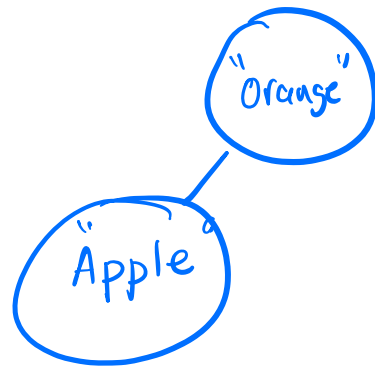


set class can store keys of
any type (generic)

C++ templates

$s_1 < s_2$

"Apple" < "Orange"

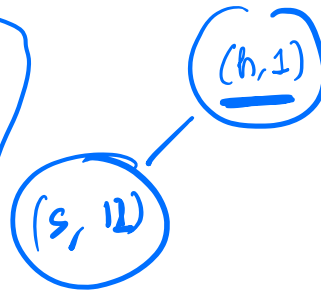


key < x

True

Suppose we want to store keys that are custom type (say Card)

then... we need to overload the relational operators for Card type (<, ==)



Finding the Maximum of Two Integers

- Here's a small function that you might write to find the maximum of two integers.

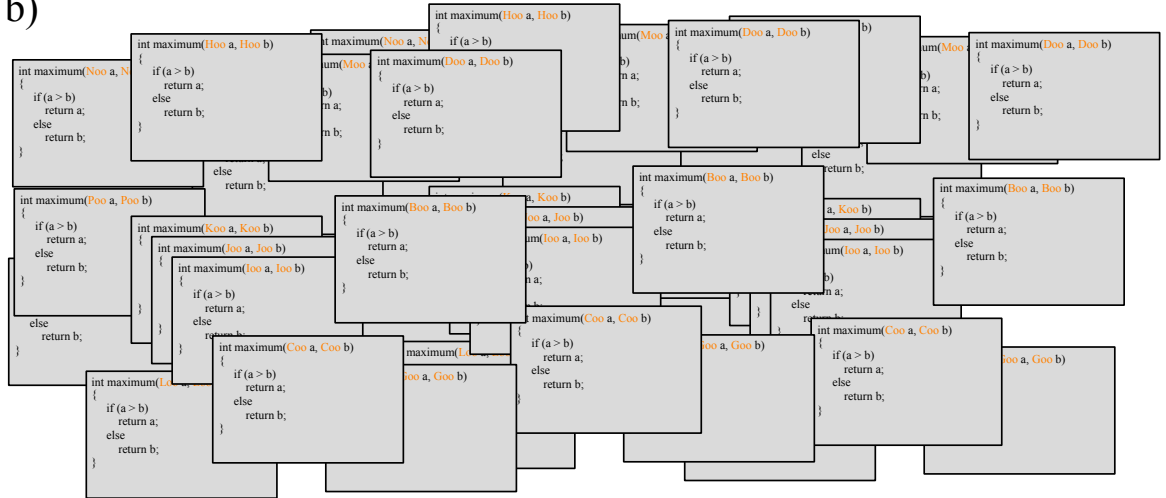
```
int maximum(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

```
string maximum (string a, string b) {
    if (a > b)
        return a;
    else
        return b;
}
```

One Hundred Million Functions...

Suppose your program uses 100,000,000 different data types, and you need a maximum function for each...

```
int maximum(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}
```



A Template Function for Maximum

When you write a template function, you choose a data type for the function to depend upon...

```
template <class Item>
```

type generic Blueprint of the maximum function

```
Item maximum(Item a, Item b)
```

```
{  
    if (a > b)  
        return a;  
    else  
        return b;  
}
```

```
int a = 10, b = 20;  
maximum(a, b);  
string x = "apple", y = "banana";  
maximum(x, y);
```

The compiler creates actual instances of maximum depending on the parameters passed to the function when it's called

BST, without templates:

```
class BSTNode {
public:
    BSTNode* left;
    BSTNode* right;
    BSTNode* parent;
    int const data;

    BSTNode( const int& d ) :
        data(d) {
        left = right
        = parent = nullptr;
    }
};
```

*We want to
make these*

*types
generic*

BST, with templates:

Generic Type

```
template<class Data>
class BSTNode {
public:
    BSTNode<Data>* left;
    BSTNode<Data>* right;
    BSTNode<Data>* parent;
    Data const data;

    BSTNode( const Data & d ) :
        data(d) {
        left = right
        = parent = nullptr;
    }
};
```

Now that BSTNode uses templates, we need to specify the template parameter when declared objects or pointers of type BSTNode

e.g. previously (without templates) to create a static object of type BSTNode, we would write the following declaration

```
BSTNode n;
```

This is no longer a valid declaration. Instead we need to specify the template parameter as follows

```
BSTNode <int> n;
```

Similarly the declaration to create a pointer to a BSTNode is:

```
BSTNode <int> * p;
```

of type BSTNode

If we are creating an object, within another function or class that uses templates we may use the template parameter of that function/class

e.g.

```
template <class T>
void foo (T t) {
    BSTNode <T> b;
}
```

In this case the template parameter T is determined when we call the function foo

```
int x=5;
foo(x);
```

BST, with templates:

```
template<class Data>
class BSTNode {
public:
    BSTNode<Data>* left;
    BSTNode<Data>* right;
    BSTNode<Data>* parent;
    Data const data;
```

```
    BSTNode( const Data & d ) :
        data(d) {
        left = right = parent = nullptr ;
    }
};
```

How would you create a **BSTNode** object on the runtime stack?

- A. `BSTNode n(10);`
- B. `BSTNode<int> n;`
- C. `BSTNode<int> n(10);`
- D. `BSTNode<int> n = new BSTNode<int>(10);`
- E. More than one of these will work

{ } syntax OK too

BST, with templates:

```
template<class Data>
class BSTNode {
public:
    BSTNode<Data>* left;
    BSTNode<Data>* right;
    BSTNode<Data>* parent;
    Data const data;

    BSTNode( const Data & d ) :
        data(d) {
        left = right = parent = nullptr ;
    }

};
```

How would you create a **pointer** to BSTNode with integer data?

- A. BSTNode* nodePtr;
- B. BSTNode<int> nodePtr;
- C. BSTNode<int>* nodePtr;

BST, with templates:

```

template<class Data>
class BSTNode {
public:
    BSTNode<Data>* left;
    BSTNode<Data>* right;
    BSTNode<Data>* parent;
    Data const data;

    BSTNode( const Data & d ) :
        data(d) {
        left = right = parent = nullptr ;
    }

};

```

Complete the line of code to create a new BSTNode object with int data on the heap and assign nodePtr to point to it.

```

BSTNode<int>* nodePtr ;

```

```

nodePtr = new BSTNode<int>(10);

```

Working with a BST

```
template<typename Data>
```

```
class BST {
```

```
private:
```

```
    BSTNode<Data>* root; //Pointer to the root of this BS
```

```
public:
```

```
    /** Default constructor. Initialize an empty BST. */
```

```
    BST() : root(nullptr){ }
```

```
    void insertAsLeftChild(BSTNode<Data>* parent, const Data& item){
```

```
        // Your code here
```

```
}
```

Working with a BST: Insert

```
//Assume this is inside the definition of the class
void insertAsLeftChild(BSTNode<Data>* parent, const Data& item)
{
    // Your code here
}
```

Which line of code correctly inserts the data item into the BST as the left child of the parent parameter.

A. `parent.left = item;`

B. `parent->left = item;`

C. `parent->left = BSTNode(item);`

D. `parent->left = new BSTNode<Data>(item);`

E. `parent->left = new Data(item);`

Working with a BST: Insert

```
void insertAsLeftChild(BSTNode<Data>* parent, const Data& item) {  
    parent->left = new BSTNode<Data>(item);  
    parent → left → parent = parent;  
}
```

Is this function complete? (i.e. does it do everything it needs to correctly insert the node?)

A. Yes. The function correctly inserts the data

B. No. There is something missing.

*Need to update the parent pointer
as shown*

What is difference between templates and typedefs?

```
template <class Item>
Item maximum(Item a, Item b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

```
typedef int item;
item maximum(item a, item b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

Template classes: Non-member functions

```
BST operator+(const BST& b1, const BST&b2);
```

```
template <class T>
```

```
BST<T> operator+(const BST<T>& b1, const BST<T>&b2);
```

Template classes: Member function definition

For the compiler a name used in a template declaration or definition and that is dependent on a template-parameter is assumed not to name a type *unless* its preceded by a typename

```
template<class T>
class BST{
    //Other code
    Node* getNodeFor(T value, Node* n) const;
};
```

The correct way of defining this function outside the class definition is

```
template < class T >
typename BST<T>:: Node * getNodeFor ( T value, Node * n ) const
{ }
```


Template classes: Including the implementation

```
//In bst.h  
class BST{  
//code  
};
```

```
#include "bst.cpp"
```

The include statement comes at the bottom because the class definition should precede the definition of the functions defined in bst.cpp

How to Convert a Container Class to a Template

1. The template prefix precedes each function prototype or implementation.
2. Outside the class definition, place the word `<Item>` with the class name, such as `bag<Item>`.
3. Use the name `Item` instead of `value_type`.
4. Outside of member functions and the class definition itself, add the keyword *typename* before any use of one of the class's type names. For example:

```
typename bag<Item>::size_type
```
5. The implementation file name now ends with `.template` (instead of `.cxx`), and it is included in the header by an include directive.
6. Eliminate any using directives in the implementation file. Therefore, we must then write `std::` in front of any Standard Library function such as `std::copy`.
7. Some compilers require any default argument to be in both the prototype and the function implementation.

Review and demo an example