

C++ TEMPLATES

Problem Solving with Computers-II

C++

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

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

Finding the Maximum of Two Integers

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

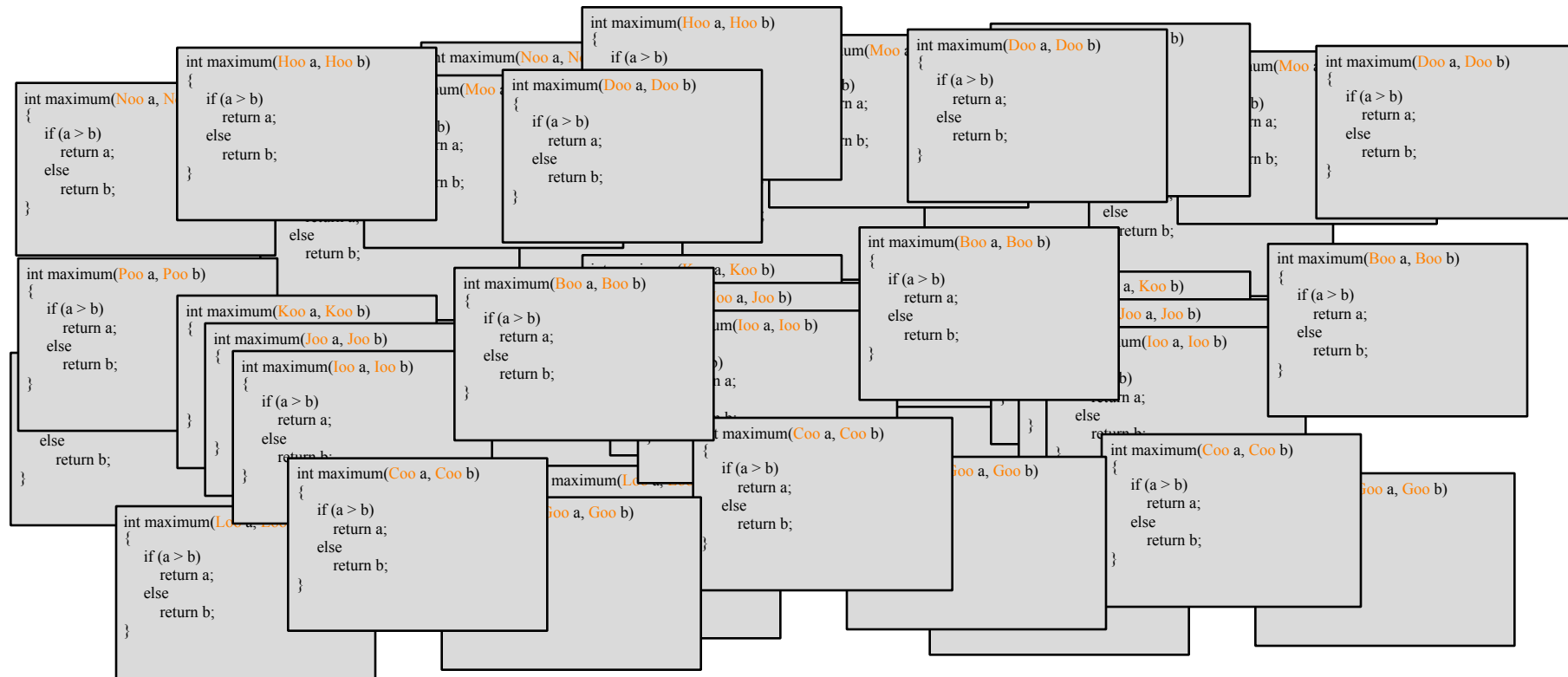
```
int maximum(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

Finding the Maximum of Two Points

```
point maximum(Point a, Point 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...



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

What are the advantages over typedef?

```
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;
}
```

BST, with templates:

```
template<typename 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 = 0;
    }

};
```

BST, with templates:

```
template<typename 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 = 0;
    }

};
```

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<typename 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 = 0;
    }

};
```

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<typename 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 = 0;
    }

};
```

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
```

CHANGING GEARS: C++STL

- The C++ Standard Template Library is a very handy set of three built-in components:
 - Containers: Data structures
 - Iterators: Standard way to search containers
 - Algorithms: These are what we ultimately use to solve problems

C++ STL container classes

```
array  
vector  
forward_list  
list  
stack  
queue  
priority_queue  
set  
multiset (non unique keys)  
deque  
unordered_set  
map  
unordered_map  
multimap  
bitset
```

Working with a BST

```
template<typename Data>
class BST {

private:

    /** Pointer to the root of this BST, or 0 if the BST is
    empty */
    BSTNode<Data>* root;

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

```
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);
}
```

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

- A. Yes. The function correctly inserts the data
- B. No. There is something missing.

Working with a BST: Insert

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

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

}
```


Template classes

Using a Typedef Statement:

```
class bag
{
public:
    typedef int value_type;
    . . .
```

Using a Template Class:

```
template <class Item>
class bag
{
public:
    typedef Item value_type;
    . . .
```

Template classes: Non-member functions

```
bag operator +(const bag& b1, const bag& b2)...
```

```
template <class Item>
```

```
bag<Item> operator +(const bag<Item>& b1, const bag<Item>& b2)...
```

Template classes: Member function prototype

- Rewrite the prototype of the member function “count” using templates

Before (without templates)

```
class bag{  
    public:  
        typedef std::size_t size_type;  
        ....  
        size_type count(const value_type& target) const;  
        .....  
};
```

Template classes: Member function definition

```
bag::size_type bag::count(const value_type& target) const ...
```

The function's return type is specified as `bag::size_type`. But this return type is specified before the compiler realizes that this is a `bag` member function. So we must put the keyword *typename* before `bag<Item>::size_type`. We also use `Item` instead of `value_type`:

```
template <class Item>  
typename bag<Item>::size_type bag<Item>::count  
    (const Item & target) const ...
```

Template classes: Including the implementation

```
#include "bag4.template" // Include the implementation.
```

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.