Templates and Generics

Templates in C++ (Chapter 13)

Templates (also known as generics) allow code to be written in such a way that the same program can be used for different data types.

Consider for example a C linked-list program that is written to hold a list of integers.

If you wanted to store a list of, say, floating point types, you’d need to create a new program for that.

And if you wanted to store a list of strings, you’d need to create yet another version of the same program.

All of these different versions of the linked-list program would have a great deal in common, since, after all, all linked lists possess the same fundamental structure: you have a list of linked nodes, and, at the least, each node must hold one value and, at the same time, point to the next node.
Ordinarily, this problem also exists with C++ programs.

If you were to create, say, a linked-list program in C++ for storing integers, you would not be able to use the same program for storing strings.

To get around these limitations, we have templates in C++ and generics in Java.

A templatized program (or a generic program) can be a class or a method that is parameterized by a variable that can be instantiated to different types.
For example, an ordinary linked-list program in C++ may look like

```cpp
class LinkedList {

    struct Node {
        Node* next;
        int val;
    };

    Node* head;

public:
    // public interface of the class
    ....
    ....
};
```

The individual nodes of the linked list will be instances of the inner class, in this case a `struct` named `Node`.

Note that since the data member `val` of the struct `Node` is defined to be of type `int`, this linked list will only be able to hold integers.
Templatized version of the program:

```cpp
template<class T> class LinkedList {

    struct Node {
        Node* next;
        T val;
    };

    Node* head;

public:
    // public interface of the class
    ....
    ....
};
```

Note how the header of the class has been parameterized by incorporation of the variable `T` and how the type of the data member `val` inside the inner class `Node` has been declared to be of type `T`.

With this new definition, `LinkedList<int>` would do exactly the same as our previous `LinkedList` class.

But now we can use the same program to create a `LinkedList<double>` class for storing a list of doubles, a `LinkedList<string>` class for storing a list of strings, etc.
Templatized Classes and Methods in C++

We'll look at how one might structure a C++ program for a linked list of ints.

We will then "templatize" the program so that the same program can be used for different data types.
We evidently need some kind of a data structure to serve as a node in a linked list. We could use the following structure:

```cpp
struct Node {
    Node* pre;
    Node* next;
    int item;
    // ....
};
```

In this node, the data member `item` can be used to store one item in the list of integers and the data members `pre` and `next` tell us how to get to the next node and the previous node in the linked list.

To be able to create a node, we also need a constructor for a `Node`:

```cpp
struct Node {
    Node* pre;
    Node* next;
    int item;

    Node( Node* p, Node* s, const int& i)
        : pre(p), next(s), item(i) {}
};
```
We can now create a new Node by

```cpp
int i = 10;
Node* newNode = new Node(0, 0, i);
```

and then link it to the head of the linked list of nodes already in place by

```cpp
newNode->suc = head;
head->pre = newNode;
head = newNode;
```

The entire chain of nodes created in this manner can be encapsulated in an object of type LinkedList, defined below, so as to hide the details of the implementation from a user of the linked list:
class LinkedList {

    struct Node {
        Node* pre;
        Node* next;
        int item;
        Node( Node* p, Node* n, const int& i)
            : pre(p), next(n), item(i) {} 
    };

    Node* head;

public:

    LinkedList() : head(0) {} 

    LinkedList( const int& x ) : head( new Node(0, 0, x) ) {} 

    void print_all() {
        for ( Node* p = head; p; p = p->next )
            cout << p->item << ' ';
    }

    void addToList( int );

    void removeFromList( int );

};
Note how the struct **Node**, together with its constructor, belongs to the private section of **LinkedList**.

That’s because the users of a **LinkedList** do not need to know how exactly the members of a **LinkedList** are stored.

Also private to the class **LinkedList** is the data member **head**, which will serve as a pointer to the first node of a linked list.

In the public section, we define a no-arg constructor for **LinkedList** by

```cpp
LinkedList() : head( 0 ) {}
```

which effectively sets the data member **head** to the null pointer, an appropriate thing to do for an empty list.
We then define what it means for a **LinkedList** to contain a single integer item:

```cpp
List( const int& x ) : head( new Node(0, 0, x) ) {} 
```

For this case, we have a **LinkedList** of just one node, pointed to by the data member `head`, whose `pre` and `next` pointers are both 0 and for which the `item` data member is set equal to the `int` to be stored.
That brings us to the `print_all()` function:

```cpp
void print_all() {
    for ( Node* p = head; p; p = p->next )
        cout << p->val << ' ';
}
```

Note that the loop variable `p` will be non-zero for all the nodes in a `LinkedList`.

When this variable acquires the value 0, we have reached the end of the `LinkedList`. 
```cpp
//LinkedList.cc

#include <iostream>

class LinkedList {

    struct Node {
        Node* pre;
        Node* next;
        int item;
        Node( Node* p, Node* n, const int& i)
            : pre(p), next(n), item(i) {}
    }

    Node* head;

public:

    LinkedList() : head(0) {}
    LinkedList( const int x ) : head( new Node(0, 0, x) ) {}
    void print_all() {
        for ( Node* p = head; p; p = p->next )
            cout << p->item << ' ';
        cout << endl;
    }
    void addToList( int m ) {
        Node* p = head;
        while (p->next)
            p = p->next;
        p->next = new Node(0, 0, m);
        p->next->pre = p;
    }
    void removeFromList( int );
    void removeFromList( int );
};
```
void LinkedList::removeFromList( int m ) {
    Node* p = head;
    if (p == 0) return;
    while (p->item != m) {
        if (p->next == 0) return;
        p = p->next;
    }
    if (p == head) {
        head = head->next;
        head->pre = 0;
        delete p;
        return;
    }
    p->pre->next = p->next;
    if (p->next != 0) // item to be deleted is at the end of list
        p->next->pre = p->pre;
    delete p;
}

int main() {
    LinkedList alist(3); // 3
    alist.addToList(5); // 3 5
    alist.addToList(7); // 3 5 7
    alist.addToList(9); // 3 5 7 9
    alist.addToList(11); // 3 5 7 9 11
    alist.print_all(); // 3 5 7 9 11
    alist.removeFromList(7); // 3 5 9 11
    alist.print_all();
}
alist.removeFromList(3); // 5 9 11
alist.print_all();

alist.removeFromList(11); // 5 9
alist.print_all();
}
A Templatized Linked-List Program

The program we showed can only be used for storing a list of \texttt{ints}, or for storing other data types that the compiler is allowed to convert to \texttt{ints}.

We could write a similar program for storing a list of \texttt{chars}, \texttt{strings}, etc.

All of these programs would have a great deal in common, as was mentioned before, but would not be interchangeable functionally.
template<class T> class LinkedList {

  struct Node {
    Node* pre;
    Node* next;
    T item;
    Node( Node* p, Node* n, const T& t)
      : pre(p), next(n), item(t) {} 
  };

  Node* head;

public:

  LinkedList() : head( 0 ) {} 

  LinkedList( const T& t ) : head( new Node(0, 0, t) ) {} 

  void print_all() {
    for ( Node* p = head; p; p = p->next )
      cout << p->val << ' '; 
  } 

  void addToList( T );

  void removeFromList( T );
};
Note the prefix

\[ \text{template<class } T\text{> } \]

that specifies that a template is being declared and that type parameter \( T \) will be used in the declaration.

The scope of \( T \) extends to the end of the declaration prefixed by \texttt{template<class } T\text{>}.\]

In this case, what we have created is a class template – as opposed to, say, a function template that we will talk about later – because the prefix \texttt{template<class } T\text{> } is followed by the keyword \texttt{class}.

The name of this class template is \texttt{LinkedList} since that’s the identifier that comes next.
A templatized class can be used like any other class in C++ after a previously defined type is substituted for the parameter.

For example, the templatized class shown could be used as the following classes:

\[
\begin{align*}
\text{LinkedList<int>} \\
\text{LinkedList<double>} \\
\text{LinedList<float>} \\
\text{LinkedList<string>}
\end{align*}
\]

or, even,

\[
\text{LinkedList<LinkedList>}
\]
Function Templates in C++

In addition to class templates, one can also have function templates.

In fact, in general, a class template will contain functions that would depend on the parameter $T$ appearing in the prefix `template<class T>` of the templatized class definition.

The precise syntax for defining these functions depends on whether or not they are defined within a class.
Consider, for example, the function `addToList(T item)` whose header appears in the `LinkedList` class definition.

To provide a definition for this function outside of the class definition, its syntax would be something like this:

```cpp
template <class T> void LinkedList<T>::addToList( T item ) {
    Node* newNode = new Node(0, head, item);
    head = newNode;
    newNode->next->pre = head;
}
```
Note how the implementation of the function carries the prefix `template<class T>` in its header:

```cpp
    template <class T> void LinkedList<T>::add_to_list( T item )
```

Because it is a function template, in the sense that the function uses a type parameter `T` in its definition, its implementation must begin with the prefix `template<class T>`.

Then comes the return type of the function, in this case `void`.

This is followed by the name of the class template for which the function is being defined, the name here being `LinkedList<T>`.

Next comes the scope operator `::`, followed by actual name of the function, `add_to_list`.

The parameter list that comes next also contains the type parameter `T` in the example here.
//LinkedListGeneric.cc

#include <iostream>
#include <string>

template <class T> class LinkedList {

  struct Node {
    Node* pre;
    Node* next;
    T item;
    Node( Node* p, Node* n, const T& t)
    : pre(p), next(n), item(t) {}
  };

  Node* head;

public:

  LinkedList() : head() {}
  LinkedList( const T& t ) : head( new Node(0, 0, t)) {}  
  void print_all() {
    for (Node* p = head; p; p = p->next )
      cout << p->item << ' ';  
  }

  template <class T> void addToList(T item ) {
    Node* newNode = new Node(0, head, item );
    head = newNode;
    newNode->next->pre = head;
  }

  void removeFromList( T );

};
template<class T> void LinkedList<T>::removeFromList(T item) {
    Node* p = head;
    for (; p->item != item; p = p->next)
        if (p->next == 0) return;  // item not in list
    if (p == head) {  // item to be deleted in the first node
        head = head->next;
        head->pre = 0;
        delete(p);
        return;
    }
    p->pre->next = p->next;
    if (p->next != 0)  // item to be deleted is at the end of list
        p->next->pre = p->pre;
    delete(p);
}

int main() {

    // use the template for a list of ints

    int i = 100;
    LinkedList<int>* numlist = new LinkedList<int>(i);
    numlist->print_all();
    cout << "\nadding 500 to the list" << '\n';
    numlist->addToList(500);
    numlist->print_all();
    cout << "\nadding 600 to the list" << '\n';
    numlist->addToList(600);
    numlist->print_all();
    cout << "\nremoving 600 from the list" << '\n';
    numlist->removeFromList(600);
}
numlist->print_all();
cout << endl;

// use the template for a list of chars

cchar x = 'a';
LinkedList<char>* charlist = new LinkedList<char>(x);
charlist->print_all();
cout << "\nadding char b to the list" << 'n';
charlist->addToList( 'b' );
charlist->print_all();
cout << "\nadding char c to the list" << 'n';
charlist->addToList( 'c' );
charlist->print_all();
cout << "\nremoving b from the list" << 'n';
charlist->removeFromList( 'b' );
charlist->print_all();
cout << endl;

// use the template for a list of strings

string str = "high";
LinkedList<string>* stringList = new LinkedList<string>( str );
stringList->print_all();
cout << "\nadding the string sierras to the list" << endl;
stringList->addToList( "sierras" );
stringList->print_all();
cout << "\nadding the string green to the list" << endl;
stringList->addToList( "green" );
stringList->print_all();
cout << "\nadding the string tiaras to the list" << endl;
stringList->addToList( "tiaras" );
stringList->print_all();
cout << "\nremoving 'high' from the list" << 'n';
stringList->removeFromList( "high" );
stringList->print_all();
cout << endl;
}
Template Specialization

It is not always possible to write a C++ template class that would work universally for every data type.

In such cases, it could become necessary to provide alternative definitions for the same template and let the compiler choose the most applicable one.

For example, the previously defined class template for a linked list would not work for those types for which the overload definitions of the operators ‘==’ and ‘!=’ (needed in the `removeFromList()` function) are not provided.
Let’s say we wanted to use the previously defined class template (and the associated function templates) for C-style strings, i.e., for the type \texttt{char*}.

In other words, we wish to create objects of type

\begin{verbatim}
LinkedList<char*> 
\end{verbatim}

But, unfortunately, that may not work because C-style strings are traditionally compared using the \texttt{strcmp()} function from the \texttt{string.h} library and not by using the ‘==’ and ‘!=’ operators.

So, in such cases, we may wish to provide an alternative definition of the template.

The next slide shows a specialization of the \texttt{LinkedList<T>} class for the case of \texttt{char*}: 
```cpp
#include<iostream>
#include "LinkedListGeneric.cc" //without the main in LinkedListGeneric.cc

class LinkedList<char*> { 

    struct Node {
        Node* pre;
        Node* next;
        char* item;
        Node( Node* p, Node* n, char* c )
        : pre(p), next(n), item(c) {}  
    };

    Node* head;

public:
    LinkedList() : head( 0 ) {}  
    LinkedList( char* t ) : head( new Node(0, 0, t)) {}  
    void print_all() {
        for (Node* p = head; p; p = p->next )
            cout << p->item << ' '; 
    }
    void addToList( char* );
    void removeFromList( char* );
};

void LinkedList<char*>::addToList( char* item ) { 
    Node* newNode = new Node(0, head, item);
    head = newNode;

```
newNode->next->pre = head;
}

void LinkedList<char*>::removeFromList( char* item ) {
    Node* p = head;
    for (; 0 != strcmp(p->item, item); p = p->next)
        if (p->next == 0) return;
    if (p == head) { //item to be deleted is at the head of list
        head = head->next;
        head->pre = 0;
        return;
    }
    p->pre->next = p->next;
    if (p->next != 0) //item to be deleted is at the end of list
        p->next->pre = p->pre;
}

int main() {

    // use the template for a list of ints
    int i = 100;
    LinkedList<int>* numlist = new LinkedList<int>(i);
    numlist->print_all();
    cout << "\nadding 500 to the list" << 'n';
    numlist->addToList( 500 );
    numlist->print_all();
    cout << "\nadding 600 to the list" << 'n';
    numlist->addToList( 600 );
    numlist->print_all();
    cout << "\nremoving 600 from the list" << 'n';
    numlist->removeFromList( 600 );
    numlist->print_all();
    cout << endl;
}
// use the template for a list of chars
char x = 'a';
LinkedList<char>* charlist = new LinkedList<char>(x);
charlist->print_all();
cout << "adding char b to the list" << 'n';
charlist->addToList( 'b' );
charlist->print_all();
cout << "adding char c to the list" << 'n';
charlist->addToList( 'c' );
charlist->print_all();
cout << "removing b from the list" << 'n';
charlist->removeFromList( 'b' );
charlist->print_all();
cout << endl;

// use the template for a list of strings
string str = "high";
LinkedList<string>* stringList = new LinkedList<string>( str );
stringList->print_all();
cout << "adding the string sierras to the list" << endl;
stringList->addToList( "sierras" );
stringList->print_all();
cout << "adding the string green to the list" << endl;
stringList->addToList( "green" );
stringList->print_all();
cout << "adding the string tiaras to the list" << endl;
stringList->addToList( "tiaras" );
stringList->print_all();
cout << "removing 'high' from the list" << 'n';
stringList->removeFromList( "high" );
stringList->print_all();
cout << endl;

// use the template for a list of C-style strings
char* cstr = "high";
LinkedList<char*>* cstringList = new LinkedList<char*>(cstr);
cstringList->print_all();
cout << "\nadding the string sierras to the list" << endl;
cstringList->addToList("sierras");
cstringList->print_all();
cout << \nadding the string green to the list" << endl;
cstringList->addToList("green");
cstringList->print_all();
cout << \nadding the string tiaras to the list" << endl;
cstringList->addToList("tiaras");
cstringList->print_all();
cout << \nremoving 'high' from the list" " \n';
cstringList->removeFromList("high");
cstringList->print_all();
cout << endl;
Note that the prefix `template <class T>` is now missing from the header of this specialization to the templatized class `LinkedList`.

However, if we so wanted, we could used the prefix `template <>` and defined the specialization as

```cpp
template <> class LinkedList<char*> {

    // same as before

};
```

Although we do not need the prefix any more, the compiler knows that this specialization is an alternative to the `LinkedList` template defined previously.

So when we try to make objects of type `LinkedList<char*>`, it will automatically choose the specialization rather than the original implementation.

For the same reason we did not need the prefix `template<class T>` for the class template, we do not need it for the associated function templates shown above.
General Syntax of a Template Declaration

```cpp
template< ---- template parameter list ------ > class nameOfClass {

    // implementation

};
```

What follows the keyword `template` inside the tokens `< and >` is called the `template parameter list`.

In this list, a parameter can be either a `type parameter` or a `nontype parameter` representing a constant expression.

A type parameter consists of the keyword `class` or the keyword `typename` followed by an identifier.

A nontype parameter consists of an ordinary parameter declaration.
Let’s first consider the case in which the template parameter list consists of only type parameters:

```cpp
template < class T1, class T2, class T3 > class className {
    // ...
};
```

Here the parameters \( T_1 \), \( T_2 \), and \( T_3 \) can be any built-in or user-defined types.
Now let’s consider the case in which a template parameter list has both a type parameter and a nontype parameter:

```cpp
template< class T, int i > class Buffer {
    T v[ i ];
    int sz;
public:
    Buffer() : sz( i ) {}
    //
};
```

Here \( i \) is a nontype parameter in the parameter list of the template.

So nontype parameters in a template parameter list are ordinary parameter declarations.
The parameters in a template parameter list are allowed to have default values.

These work just like the default values for functions.

As was the case with functions, the parameters are default initialized from the right of the parameter list, meaning that a default-initialized parameter cannot be to the left of an uninitialized parameter.
For example, the above definition for the template class `Buffer` could have been written in the following manner:

```cpp
template< class T, int i = 256 > class Buffer {
    T v[ i ];
    int sz;
public:
    Buffer() : sz( i ) {}  //
};
```

or, in the following manner:

```cpp
template< class T = string, int i = 128 > class Buffer {
    T v[ i ];
    int sz;
public:
    Buffer() : sz( i ) {}  //
};
```