Templates

Templates

- Using templates, it is possible to create generic functions and classes.
- In a generic function or class, the type of data upon which the function or class operates is specified as aparameter. Thus, you can use one function or class with several different types of data,without having to explicitly recode specific versions for each data type.

Templates

- Type-independent patterns that can work with multiple data types.
  - Generic programming
  - Code reusable
- Function Templates
  - These define logic behind the algorithms that work for multiple data types.
  - By creating a generic function, you can define the nature of the algorithm, independent of any data.
  - The compiler will automatically generate the correct code for the type of data that is actually used when you execute the function.
- Class Templates
  - These define generic class patterns into which specific data types can be plugged in to produce new classes.

Function and function templates

- C++ routines work on specific types. We often need to write different routines to perform the same operation on different data types.

```
int maximum(int a, int b, int c)
 {
      int max = a;
      if (b > max) max = b;
      if (c > max) max = c;
      return max;
}
```

Function and function templates

```
float maximum(float a, float b, float c)
 {
      float max = a;
      if (b > max) max = b;
      if (c > max) max = c;
      return max;
}
```

Function and function templates

```
double maximum(double a, double b, double c)
 {
     double max = a;
     if (b > max) max = b;
     if (c > max) max = c;
     return max;
}
```

The logic is exactly the same, but the data type is different.

Function templates allow the logic to be written once and used for all data types – generic function.

Function and function templates

Function templates allow the logic to be written once and used for all data types – generic function.
A generic function is created by using the keyword **template**.

The general form of a template function definition is shown here:

```
template <class Ttype> ret-type func-name(parameter list)
{
                // body of function
}
```

Ttype is a placeholder name for a data type used by the function. This name can be used within the function definition. However, it is only a placeholder that the compiler will automatically replace with an actual data type when it creates a specific version of the function.

Function Templates

- Generic function to find a maximum value (see maximum example).

```
Template <class T>
T maximum(T a, T b, T c)
 {
     T max = a;
     if (b > max) max = b;
     if (c > max) max = c;
     return max;
}
```

- Template function itself is incomplete because the compiler will need to know the actual type to generate code. So template program are often placed in .h or .hpp files to be included in program that uses the function.
- C++ compiler will then generate the real function based on the use of the function template.

Function Templates Usage

- After a function template is included (or defined), the function can be used by passing parameters of real types.

```
Template <class T>
T maximum(T a, T b, T c)
…
int i1, i2, i3;
…
Int m = maximum(i1, i2, i3);
```

- maximum(i1, i2, i3) will invoke the template function with T==int. The function returns a value of int type.

Function Templates Usage

- Each call to `maximum()` on a different data type forces the compiler to generate a different function using the template. See the maximum example.
  - One copy of code for many types.

```
int i1, i2, i3;
                    // invoke int version of maximum
cout << "The maximum integer value is: "
  << maximum( i1, i2, i3 );
                    // demonstrate maximum with double values
double d1, d2, d3;
                    // invoke double version of maximum
cout << "The maximum double value is: "
  << maximum( d1, d2, d3 );
```

Another example

```
template< class T >
void printArray( const T *array, const int count )
{
   for ( int i = 0; i < count; i++ )
      cout << array[ i ] << " "; cout << endl;
}
```

Usage

```
template< class T >
void printArray( const T *array, const int count );

char  cc[100];
int      ii[100];
double dd[100];
……
printArray(cc, 100);
printArray(ii, 100);
printArray(dd, 100);
```

Usage

```
template< class T >
void printArray( const T *array, const int count );

char  cc[100];
int      ii[100];
double dd[100];
myclass xx[100];  <- user defined type can also be used.
……
printArray(cc, 100);
printArray(ii, 100);
printArray(dd, 100);
printArray(xx, 100);
```

Use of template function

- Can any user defined type be used with a template function?
  - Not always, only the ones that support all operations used in the function.
  - E.g. if myclass does not have overloaded << operator, the printarray template function will not work.

Example

- A generic function that swaps the values of the two variables with which it is called.

```
// Function template example.
#include <iostream>
using namespace std;
// This is a function template.
template <class X>
void swapargs(X &a, X &b)
{
X temp;
temp = a;
a = b;
b = temp;
}
```

```
int main()
{
int i=10, j=20;
double x=10.1, y=23.3;
char a='x', b='z';
cout << "Original i, j: " << i << ' '
<< j << '\n';
cout << "Original x, y: " << x << ' '
<< y << '\n';
cout << "Original a, b: " << a << ' '
<< b << '\n';
swapargs(i, j); // swap integers
swapargs(x, y); // swap floats
swapargs(a, b); // swap chars
cout << "Swapped i, j: " << i << ' '
<< j << '\n';
cout << "Swapped x, y: " << x << ' '
<< y << '\n';
cout << "Swapped a, b: " << a << ' '
<< b << '\n';
return 0;
```

The output is shown here.

```
Original i, j: 10 20
Original x, y: 10.1 23.3
Original a, b: x z
Swapped i, j: 20 10
Swapped x, y: 23.3 10.1
Swapped a, b: z x
```

A Function with Two Generic Types

- You can define more than one generic data type in the **template** statement by using a comma-separated list.

```
#include <iostream>
using namespace std;
template <class type1, class type2>
void myfunc(type1 x, type2 y)
{
cout << x << ' ' << y << '\n';
}
int main()
{
myfunc(10, "hi");
myfunc(0.23, 10L);
return 0;
}
```

When you create a template function, you are, in essence, allowing the compiler to generate as many different versions of that function as are necessary for handling the various ways that your program calls the function.

Overloading a Function Template

Create another version of the template that differs from any others in its parameter list.

```
// Overload a function template declaration.
#include <iostream>
using namespace std;
         // First version of f() template.
template <class X> void f(X a)
{
cout << "Inside f(X a)\n";
}
         // Second version of f() template.
template <class X, class Y> void f(X a, Y b)
{
cout << "Inside f(X a, Y b)\n";
}
int main()
{
f(10); // calls f(X)
f(10, 20); // calls f(X, Y)
return 0;
}
```

Using Standard Parameters with Template Functions

Non-generic parameters work just like they do with any other function.

```
#include <iostream>
using namespace std;
// Display data specified number of
times.
template<class X> void repeat(X data, int
times)
{
do {
cout << data << "\n";
times--;
} while(times);
}
int main()
{
repeat("This is a test", 3);
repeat(100, 5);
repeat(99.0/2, 4);
return 0;
}
```

Here is the output produced by this program:

```
This is a test
This is a test
This is a test
100
100
100
100
100
49.5
49.5
49.5
49.5
```

Class template

- So far the classes that we define use fix data types.
- Sometime is useful to allow storage in a class for different data types.
- The actual type of the data being manipulated will be specified as a parameter when objects of that class are created.
  - Function templates allow writing generic functions that work on many types.
  - Same idea applies to defining generic classes that work with many types -- extract the type to be a template to make a generic classes.

Class template

- The compiler will automatically generate the correct type of object, based upon the type you specify when the object is created.

- The general form of a generic class declaration is shown here:

  template <class *Ttype*> class *class-name* {
  }

- You can define more than one generic data type by using a comma-separated list.
- You create a specific instance of that class by using the following general form:

  class-name <*type*> *ob*;
- Here, *type* is the type name of the data that the class will be operating upon. Member functions of a generic class are, themselves, automatically generic. You don't need to use **template** to explicitly specify them as such.

Class template

- To make a class into a template, prefix the class definition with the syntax:

  template< class T >

  - Here T is just a type parameter. Like a function parameter, it is a place holder.
  - When the class is instantiated, T is replaced by a real type.
- To access a member function, use the following syntax:
  - className< T >:: memberName.
  - SimpleList < T > :: SimpleList()
- Using the class template:
  - ClassName<real type> variable;
  - SimpleList < int > list1;

Class Template - Example

```
// Demonstrate a generic queue class.
#include <iostream>
using namespace std;
const int SIZE=100;
// This creates the generic class queue.
template <class QType> class queue {
QType q[SIZE];
int sloc, rloc;
public:
queue() { sloc = rloc = 0; }
void qput(QType i);
QType qget();
};
// Put an object into the queue.
template <class QType>
void queue<QType>::qput(QType i)
{
if(sloc==SIZE) {
cout << "Queue is full.\n";
return;
}
sloc++;
q[sloc] = i;
}
// Get an object from the queue.
template <class QType> QType queue<QType>::qget()
{
if(rloc == sloc) {
cout << "Queue Underflow.\n";
return 0;
}
rloc++;
return q[rloc];
}
```

```
int main()
{
queue<int> a, b; // create two integer queues
a.qput(10);
b.qput(19);
a.qput(20);
b.qput(1);
cout << a.qget() << " ";
cout << a.qget() << " ";
cout << b.qget() << " ";
cout << b.qget() << "\n";
queue<double> c, d; // create two double queues
c.qput(10.12);
d.qput(19.99);
c.qput(-20.0);
d.qput(0.986);
cout << c.qget() << " ";
cout << c.qget() << " ";
cout << d.qget() << " ";
cout << d.qget() << "\n";
return 0;
}
```

You can create another queue that stores character pointers:
queue<char *> chrptrQ;

The output is.

```
10 20 19 1
10.12 -20 19.99 0.986
```

An Example with Two Generic Data Types

```
#include <iostream>
using namespace std;
template <class Type1, class Type2> class
myclass
{
Type1 i;
Type2 j;
public:
myclass(Type1 a, Type2 b) { i = a; j = b; }
void show() { cout << i << ' ' << j << '\n'; }
};
int main()
{
myclass<int, double> ob1(10, 0.23);
myclass<char, char *> ob2('X', "This is a
test");
ob1.show(); // show int, double
ob2.show(); // show char, char *
return 0;
}
```

This program produces the following output:

```
10 0.23
X This is a test
```

Creating a Generic Array Class

By combining operator overloading with a generic class, it is possible to create a generic safe-array type that can be used for creating safe arrays of any data type.

```
// A generic safe array example.
#include <iostream>
#include <cstdlib>
using namespace std;
const int SIZE = 10;
template <class AType> class atype {
AType a[SIZE];
public:
atype() {
register int i;
for(i=0; i<SIZE; i++) a[i] = i;
}
AType &operator[](int i);
};
// Provide range checking for atype.
template <class AType> AType
&atype<AType>::operator[](int i)
{
if(i<0 || i> SIZE-1) {
cout << "\nIndex value of ";
cout << i << " is out-of-bounds.\n";
exit(1);
}
return a[i];
}
```

```
int main()
{
atype<int> intob; // integer array
atype<double> doubleob; // double array
int i;
cout << "Integer array: ";
for(i=0; i<SIZE; i++) intob[i] = i;
for(i=0; i<SIZE; i++) cout << intob[i] << " ";
cout << '\n';
cout << "Double array: ";
for(i=0; i<SIZE; i++) doubleob[i] = (double)
i/3;
for(i=0; i<SIZE; i++) cout << doubleob[i] << "
";
cout << '\n';
Int ob[12] = 100; // generates runtime error
return 0;
}
```

Using Non-Type Arguments with Generic Classes

- In a template specification, you can specify what you would normally think of as a standard argument, such as an integer or a pointer.
- The syntax to accomplish this is essentially the same as for normal function parameters: Simply include the type and name of the argument.

Creating a Generic Array Class

```
// Demonstrate non-type template arguments.
#include <iostream>
#include <cstdlib>
using namespace std;
// Here, int size is a non-type argument.
template <class AType, int size>
class atype {
AType a[size];
// length of array is passed in size
public:
atype() {
register int i;
for(i=0; i<size; i++) a[i] = i;
}
AType &operator[](int i);
};
// Provide range checking for atype.
template <class AType, int size>
AType &atype<AType, size>::operator[](int i)
{
if(i<0 || i> size-1) {
cout << "\nIndex value of ";
cout << i << " is out-of-bounds.\n";
exit(1);
}
return a[i];
}
```

```
int main()
{
atype<int, 10> intob; // integer array of size
10
atype<double, 15> doubleob; // double array of
size 15
int i;
cout << "Integer array: ";
for(i=0; i<10; i++) intob[i] = i;
for(i=0; i<10; i++) cout << intob[i] << " ";
cout << '\n';
cout << "Double array: ";
for(i=0; i<15; i++) doubleob[i] = (double) i/3;
for(i=0; i<15; i++) cout << doubleob[i] << " ";
cout << '\n';
intob[12] = 100; // generates runtime error
return 0;
}
```

- Non-type parameters are restricted to integers, pointers, or references. Other types, such as **float**, are not allowed.
- Thus, non-type parameters should, themselves, be thought of as constants, since their values cannot be changed.
- For example, inside **operator[ ]( )**, the following statement is not allowed:

```
        size = 10; // Error
```

Using Default Arguments with Template Classes

A template class can have a default argument associated with a generic type. For example:

```
template <class X=int> class myclass { //...
```

It is also permissible for non-type arguments to take default arguments.

```
// Demonstrate default template arguments.
#include <iostream>
#include <cstdlib>
using namespace std;
// Here, AType defaults to int and size defaults
to 10.
template <class AType=int, int size=10> class
atype {
AType a[size]; // size of array is passed in size
public:
atype() {
register int i;
for(i=0; i<size; i++) a[i] = i;
}
AType &operator[](int i);
};
// Provide range checking for atype.
template <class AType, int size>
AType &atype<AType, size>::operator[](int i)
{
if(i<0 || i> size-1) {
cout << "\nIndex value of ";
cout << i << " is out-of-bounds.\n";
exit(1);
}
return a[i];
}
```

```
int main()
{
atype<int, 100> intarray;
// integer array, size 100
atype<double> doublearray;
// double array, default size
atype<> defarray;
// default to int array of size 10
int i;
cout << "int array: ";
for(i=0; i<100; i++) intarray[i] = i;
for(i=0; i<100; i++) cout << intarray[i]
<< " ";
cout << '\n';
cout << "double array: ";
for(i=0; i<10; i++) doublearray[i] =
(double) i/3;
for(i=0; i<10; i++) cout <<
doublearray[i] << " ";
cout << '\n';
cout << "defarray array: ";
for(i=0; i<10; i++) defarray[i] = i;
for(i=0; i<10; i++) cout << defarray[i]
<< " ";
cout << '\n';
return 0;
}
```

Creating a Generic Array Class

❑ Pay close attention to this line:

```
template <class AType=int, int size=10> class atype {
```

❑ Here, **`AType`** defaults to type **int**, and **size** defaults to 10.

❑ As the program illustrates, **`atype`** objects can be created three ways:

- By explicitly specifying both the type and size of the array
- By explicitly specifying the type, but letting the size default to 10
- By letting the type default to **int** and the size default to 10