TATA

TATA ELXSI

OBJECT ORIENTED PROGRAMMING USING C++
Module 10

Learning & Development Team

Templates

Templates

- C++ has a mechanism called templates to reduce code duplication when supporting numerous data types.
- A C++ function or C++ class with functions which operates on integers, float and double data types can be unified with a single template function or class.
- This mechanism in C++ is called the "Template".
- C++ templates fall under the category of "meta-programming" and auto code generation although one never sees the code generated

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.
- 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;
}
```

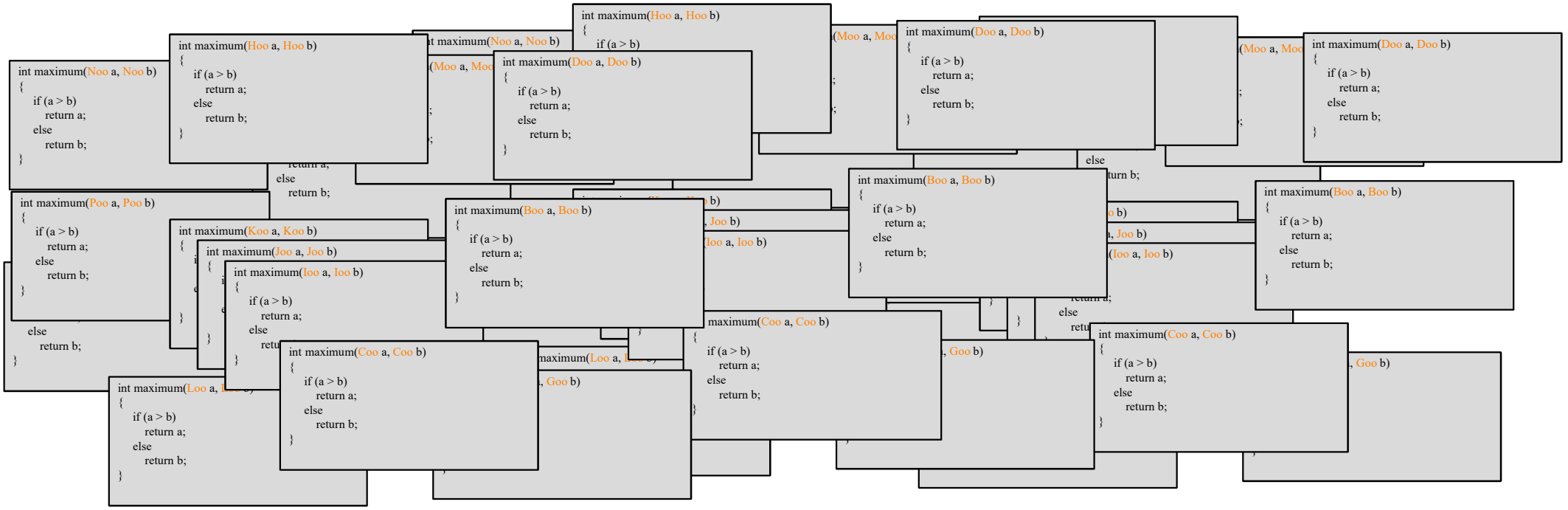
Function and function templates

```
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.

One Hundred Million Functions...

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



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 );
```

C++ template functions Example - 2

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

template <class T>
inline T square(T x)
{
   T result;
   result = x * x;
   return result;
};

main()
{
   int    i, ii;
   float  x, xx;
   double y, yy;

   i = 2;
   x = 2.2;
   y = 2.2222;

   ii = square(i);
   cout << i << ": " << ii << endl;

   xx = square(x);
   cout << x << ": " << xx << endl;

   // Explicit use of template
   yy = square<double>(y);
   cout << y << ": " << yy << endl;

   // Implicit use of template both are same
   yy = square(y);
   cout << y << ": " << yy << endl;
}
```

C++ template functions Example - 3

```
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 print array template function will not work.

Multiple Generic Types

```
template<class T, class U, class V>
void tempfun(T a, U b, V c)
{
    cout<<a<<endl<<b<<endl<<c;
}
void main()
{
    int i=10;
    float j=3.14f;
    char k=‘ T ’;
    tempfun(i, j, k);
}
```

Explicitly Overloading a Generic Function

- Even though a generic function overloads itself as needed, you can explicitly overload one, too. This is formally called **explicit specialization**.

- If you overload a generic function, that overloaded function overrides (or hides) the generic function relative to that specific version.

- Consider the following example:

Explicitly Overloading a Generic Function

```
// Overriding a template function
#include <iostream>
using namespace std;
template <class X> void swapargs (X &a, X &b)
  {X temp;
    temp = a;
    a = b;
    b = temp;
    cout << “Inside template swapargs \n”; }

//This overrides the generic version of swapargs ( ) for ints.
void swapargs( int &a, int &b)
  {int temp;
    temp = a;
    a = b;
    b = temp;
    cout << “Inside swapargs int specialization\n”;
  }
```

Explicitly Overloading a Generic Function

```
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); // calls explicitly overloaded swapargs
    swapargs( x, y); // calls generic swapargs
    swapargs( a, b); // calls generic swapargs
    cout << "Swapped i, j:  " << i << " " << j << '\n';
    cout << "Swapped x, y:  " << x << " " << y << '\n';
    cout << "Swapped a, b:  " << a << " " << b << '\n';
    return 0;
  }
```

Overloading a Template Function

- **template specification itself can be overloaded**

To do so, simply create another version of the template that differs from any others in its parameter list. For example,

```
// 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”;
 }
```

Overloading a Template Function

```
//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;
  }
```

Template Class

- Generic class independent of data type.
- Can be instantiated using type-specific versions.
- Can create an entire range of related overloaded classes called template classes.
- Usually used for data storage (container) classes like stacks etc.
- Class templates cannot be nested.
- Template classes can be inherited

Template Class Example Program

```
#include <iostream>
using namespace std;
template <class T>
class mypair {
    T a, b;
  public:
    mypair (T first, T second)
      {a=first; b=second;}
    T getmax ();
};

template <class T>
T mypair<T>::getmax (){
    T retval;
    retval = a>b? a : b;
    return retval;
}
```

```
int main () {
  mypair <int> myobject (100,75);
    mypair <char> myobject2 ('j','r');

  cout << myobject.getmax();
  cout << myobject2.getmax();

  return 0;
}
```

End of Module 10

Disclaimer

- Some examples and concepts have been sourced from the below links and are open source material
  - **http://cppreference.com**
  - **www.cplusplus.com**

- References:
  - *C++: The Complete Reference* - 4th Edition by Herbert Schildt, Tata McGraw-Hill publications.
  - *The C++ Programming Language*- by Bjarne Stroustrup.
  - *Practical C++ Programming*- by Steve Oualline, O'Reilly publications.

Learning & Development Team

ITPB Road Whitefield
Bangalore 560 048 India
Tel +91 80 2297 9123
Fax +91 80 2841 1474
e-mail info@tataelxsi.com

www.tataelxsi.com