Templates

Prepared
by
Naveen Choudhary

Templates

- Template are used to create generic functions & classes
- Generic function - Same action (code) but different data

```
                    // Function template example.
#include <iostream>
using namespace std;
                  // This is a function template.
template <class X> void swapargs(X &a, X &b)
/* this statement tells the compiler two things that a
template is being created and a generic definition is
beginning */

//<class X> or you can write <typename X> in place of
//<class X>
// the above statement can also be written as under
//    template <class X>                    //1
//    void swapargs(X &a, X &b)             //2
// but nothing else should be there b/w statements 1
and 2
{
  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
// Note :- Because swapargs() is generic function,
//the compiler automatically creates three
//versions of swapargs() one that will exchange
//integer values, one that will exchange floating
//point values, and one that will swap characters.
  cout << "Swapped i, j: " << i << ' ' << j << '\n';
  cout << "Swapped x, y: " << x << ' ' << y << '\n';
  cout << "Swapped a, b: " << a << ' ' << b << '\n';
  return 0;
}
```

- note : the template function for a specific type will be generated when the compiler actually encounters the call
- *note that standard type conversion are not applied to function templates*

Functions with Two Generic type

Template <class type1, class type2>
Void myfunc(type1 x, type2 y)

Calls

1. **myfunc (10, "I like C++")**

2. **myfunc(98.6,19L);**

Explicitly overloading a generic function

1) **Even though a generic function overloads itself as needed, you can explicitly overload one too**

2) **if you overload a generic function, that overloaded function overrides (hides) the generic function relative to the specific version**

3) **generally should be used when functionality (action/code) of the function differ for a particular data type**

Functions with Two Generic type - Example

```
// 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
intigers. Generally should be used when functionality
(action/code) of the function differ for a particular data
type. */

void swapargs(int &a, int &b)
{
 int temp;
 temp = a;
 a = b;
 b = temp;
 cout << "Inside swapargs int specialization.\n";
}
```

```
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;
}
```

New style of writing explicit specialization of template function

```
// Use of new-style specialization syntax.

template<> void swapargs<int>(int &a, int &b) {
  int temp;
  temp = a;
  a = b;
  b = temp;
  cout << "Inside swapargs int specialization.\n";
}
```

Note:- However, as a general rule, if you need to have different versions of a function for different data types, you should use overloaded function rather than templates.

Overloading a function template

```
                        // 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 & Generic Classes

- **We can mix standard parameters with generic type parameters in a template function**

Template <class X> void tabout (x data, int tab) { ----- }

Calls **::**
Tabout ("this is a test", 0)
Tabout(100,1)
Tabout ('x', 2)
Tabout(3.14, 3)

Note : generic functions are similar to overloaded functions except that they are more restrictive. When function are overloaded, you may have different actions performed within the body of each function. But a generic function must perform the same general action for all versions. Only the type of data can differ.

Generic classes

- **Useful when a class uses a logic that can be generalized**
- **Ex.1**

  queue (of integers) → delete front, add to rear {algorithm}
  queue(of char) → ---------- do ----------

- **Ex.2**

  linked list of various type - the operation performed will be same as insert, delete, search an item in the linked list

Note:- Member functions of a generic class are themselves automatically generic. you need not use template to explicitly specify them as such

syntax **:- template <class type> class class-name**
{
}

Generic Class Example

```
        // This function demonstrates a generic stack.
#include <iostream>
using namespace std;
const int SIZE = 10;
// Create a generic stack class
template <class StackType> class stack {
  StackType stck[SIZE];        // holds the stack
  int tos;                     // index of top-of-stack
public:
  stack() { tos = 0; }         // initialize stack
  void push(StackType ob);  // push object on stack
  StackType pop();             // pop object from stack
};
        // Push an object.
template <class StackType> void stack<StackType> ::
push(StackType ob)  {
  if(tos==SIZE) {
       cout << "Stack is full.\n";
       return;
     }
  stck[tos] = ob;
  tos++;
}
       // Pop an object.
template <class StackType> StackType stack<StackType> ::
pop() {
  if(tos==0) {
     cout << "Stack is empty.\n";
    return 0; // return null on empty stack
    }
  tos--;
  return stck[tos];
}
```

```
int main()
{
        // Demonstrate character stacks.

  stack<char> s1, s2;     // create two character stacks
  int i;
   s1.push('a');
  s2.push('x');
  s1.push('b');
  s2.push('y');
  s1.push('c');
  s2.push('z');
  for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "\n";
  for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop() << "\n";
          // demonstrate double stacks
  stack<double> ds1, ds2;  // create two double stacks
  ds1.push(1.1);
  ds2.push(2.2);
  ds1.push(3.3);
  ds2.push(4.4);
  ds1.push(5.5);
  ds2.push(6.6);
  for(i=0; i<3; i++) cout << "Pop ds1: " << ds1.pop() << "\n";
  for(i=0; i<3; i++) cout << "Pop ds2: " << ds2.pop() << "\n";
  return 0;
}
```

Generic Class Example……Contd

➢**when a specific instance of stack is declared, the compiler automatically generates all the functions and variables necessary for handling the actual data.**

```
Stack <char> s1, s2;                  // creates two char stack
Stack <double> ds1, ds2;              // creates two double stack
stack <char *> *s1,*s2;               // creates pointers s1 and s2 to stack type
                                      // of object. This stack object stores
                                      // character pointer

struct addr
{
        char name[40];
        char street[40];
        char city[20];
        char state[20];
        char zip[12];
};

stack <addr> obj;                     //create a stack object obj. of user defined data
                                      // type addr
```

Two Generic Data Types -Example

```
/* This example uses two generic data types in a  class definition.   */
#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', "Templates add power.");
  ob1.show();               // show int, double
  ob2.show();              // show char, char *
  return 0;
}
```

Using non-type arguments with generic classes

- **In the template specification for a generic class, you may specify non-type arguments also**

```
template <class type, int size> class type  {
                                                    .
                                                    .
                                                    .
        }
```

Call:-

```
type <int, 10> intob;                  //integer array of size = 10
type <double, 15> dob;                 // double array of size = 15
```

Note:- The arguments that you pass to a non-type parameter must consist of either an integer constant, (no float can be non type argument) or a pointer or reference to a global function or object. & thus there values inside the class's function can't be changed. Why because →The information contained in non –type arguments must be known at compile time. these non-type parameters should themselves be thought as constants.

Using default arguments with template classes

```
template <class x = int> class myclass{  }
```

Note:- It is also permissible for non-type arguments to take default argument. the default value is used when no explicit value is specified when the class is initiated

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

Calls:-

```
atype <int, 100> intarray;                  // integer array , size 100

atype <double> doublearray;          // double array, default size of 10

atype <> defarray;                          // default to int array of size 10
```

Explicit class specialization

As with template function, you can create an explicit specialization of a generic class. to do so, use the template construct, which works the same as it does for explicit function specialization.

```
// Demonstrate class specialization.
#include <iostream>
using namespace std;
template <class T> class myclass {
  T x;
public:
  myclass(T a) {
    cout << "Inside generic myclass\n";
    x = a;
  }
  T getx() { return x; }
};

// Explicit specialization for int.

template <> class myclass<int> {
// the above statement tells the compiler that an explicit
// integer specialization of myclass is being created
 int x;
public:
  myclass(int a) {
    cout << "Inside myclass<int> specialization\n";
    x = a * a;
  }
  int getx() { return x; }
};
```

```
int main()
{
  myclass<double> d(10.1);
  cout << "double: " << d.getx() << "\n\n";
  myclass<int> i(5);
  cout << "int: " << i.getx() << "\n";
  return 0;
}
```

note : explicit class specialization expands the utility of generic classes because it lets you easily handle one or two special cases while allowing all others to be automatically processes by the compiler

EXPORT KEYWORD

The export keyword can precede a template declaration. It allows other files to use template declared in a different file by specifying only its declaration rather that duplicating its entire definition

Exception Handling

Exception handling allow you to manage run – time errors in an orderly fashion. Using exception handling, your program can automatically invoke an error handling routine when an error occurs.

Exception handling fundamental

```
try {
        //try
}
catch (type1 arg) {
        //catch block
}
catch (type2 arg) {
 //catch block

}
        .
        .
catch(typeN arg) {
        //catch block
 }
```

1. *only code within in the try block (including) function definition block of the function called from within the try block) can be monitored for exception*
2. *catch block to catch & handle Exception*
3. *Multiple catch statement. so which catch to execute depends on type of exception. i.e. the catch with the data type which matches that of exception then that catch is executed*
4. *Any type of data can be caught i.e. build in data type & even user defined class*
5. In general, catch expression are checked in the order in which they occur in a program & only the matching catch is executed and all other catch are ignored

- **You can have a throw exception :: statement with in the try block to throw the exception manually**
- **If there is no corresponding catch block then standard library function terminate () will be invoked. By default terminate () calls abort () to stop your program.**

Example

```
// A simple exception handling example.

#include <iostream>
using namespace std;
int main() {
  cout << "Start\n";
  try {                                  // start a try block
    cout << "Inside try block\n";
    throw 100;                           // throw an error
    cout << "This will not execute";
  }
  catch (int i) {                        // catch an error
    cout << "Caught an exception -- value is: ";
    cout << i << "\n";
  }
  cout << "End";
  return 0;
}
```

```
/* Throwing exception from with in the function
  definition of the function call which is in try block. */

#include <iostream>
using namespace std;
void Xtest(int test) {
  cout << "Inside Xtest, test is: " << test << "\n";
  if(test) throw test;
}
int main() {
  cout << "Start\n";
  try {                                  // start a try block
 cout << "Inside try block\n";
  Xtest(0);
  Xtest(1);
  Xtest(2);
  }
  catch (int i) {                        // catch an error
    cout << "Caught an exception -- value is: ";
    cout << i << "\n";
  }
  cout << "End";
  return 0;
}
```

Local try/catch to a function

```
// Try block can be localized to a function

#include <iostream>
using namespace std;
                        // Localize a try/catch to a function.
void Xhandler(int test) {
  try{
    if(test) throw test;
  }
  catch(int i) {
    cout << "Caught Exception #: " << i << '\n';
  }
}
int main() {
  cout << "Start\n";
  Xhandler(1);
  Xhandler(2);
  Xhandler(0);
  Xhandler(3);
  cout << "End";
  return 0;
}
```

o/p → start
Caught Exception # : 1
Caught Exception # : 2
Caught Exception # : 3
End

```
/*The code associated with a catch statement will be
 executed only if it catches an exception. Otherwise,
      execution simply bypass the catch block. */

#include <iostream>
using namespace std;
int main() {
  cout << "Start\n";
  try {                        // start a try block
    cout << "Inside try block\n";
    cout << "Still inside try block\n";
  }
  catch (int i) {              // catch an error
    cout << "Caught an exception -- value is: ";
  // this catch block will not be executed as there is no
                    //exception in the try block
    cout << i << "\n";
  }
  cout << "End";
  return 0;
}
```

o/p → start
Inside try block
Still inside try block
End

Catching Class Types

```
/*Generally we create a class, which can describe the
exception\error and use this object to throw exception. */

#include <iostream>     // Catching class type exceptions.
#include <cstring>
using namespace std;
class MyException {
public:
  char str_what[80];
  int what;
  MyException() { *str_what = 0; what = 0; }
  MyException(char *s, int e) { strcpy(str_what, s); what = e; }
};
int main() {
  int i;
  try {
    cout << "Enter a positive number: ";
    cin >> i;
    if(i<0)
      throw MyException("Not Positive", i);
// An object of type MyException is created & then thrown
  }
  catch (MyException e) {               // catch an error
    cout << e.str_what << ": ";
    cout << e.what << "\n";
  }
  return 0;
}
```

Using multiple catch statement

```
/ *Catch expression are checked in the order in which
they occur in a program. Only a matching statement
is executed. All other catch blocks are ignored. */

#include <iostream>
using namespace std;
            // Different types of exceptions can be caught.
void Xhandler(int test) {
  try {
    if(test) throw test;
    else throw "Value is zero";
  }
  catch(int i) {
    cout << "Caught Exception #: " << i << '\n';
  }
  catch(const char *str) {
    cout << "Caught a string: ";
    cout << str << '\n';
  }
}
int main() {
  cout << "Start\n";
  Xhandler(1);
  Xhandler(2);
  Xhandler(0);
  Xhandler(3);
  cout << "End";
  return 0;
}
```

o/p → start
Caught Exception # : 1
Caught Exception #: 2
Caught a String: Value is zero
Caught Exception #: 3
End

Handling derived – class Exception

A catch class of a base class will also match any class derived from that base. Thus, if you want to catch exception of both a base class type and a derived class type, put the derived class first in the catch sequence. If you don't do this, the base class catch will also catch all derived classes.

```
// Catching derived classes.
#include <iostream>
using namespace std;
class B {
};
class D: public B {
};
int main() {
  D derived;
  try {
    throw derived;
  }
  catch(B b) {
    cout << "Caught a base class.\n";
  }
  catch(D d) {
    cout << "This won't execute.\n";
  }
  return 0;
}
```

o/p → caught in base class

Exception Handling Options
Catching all Exception

In some circumstances you will want an exception handler to catch all exception, instead of just a certain type. Simply use the form of catch shown below.

```
Catch(…){
// process all exception
}
```

- **one very good use for catch (…) is as the last catch of a cluster of catches. In this capacity it provides a useful default or "Catch all" statement.**

Restricting exception

- You can restrict the type of exception that a function can throw.

return_type func_name (arg_list) throw (type_list)

→ empty list, mean function can not throw any exception.

- The func_name can only throw those data type enclosed in the type_list. Throwing any other type will call the standard library function unexpected () to be called, which by default call abort () resulting in program termination.

- It is important to understand that a function can be restricted only in what type of exceptions it throws back to the try block that called it. That is, a try block within a function may throw any type of exception so long as it is caught with in that function. The restriction applies only when throwing an exception outside of the function.

```
// Restricting function throw types.
#include <iostream>
using namespace std;

// This function can only throw ints, chars, and doubles.

void Xhandler(int test) throw(int, char, double) {
  if(test==0) throw test;              // throw int
  if(test==1) throw 'a';               // throw char
  if(test==2) throw 123.23;            // throw double
}

int main() {
  cout << "start\n";
  try{
    Xhandler(0); // also, try passing 1 and 2 to Xhandler()
  }
  catch(int i) {
    cout << "Caught an integer\n";
  }
  catch(char c) {
    cout << "Caught char\n";
  }
  catch(double d) {
    cout << "Caught double\n";
  }
  cout << "end";
  return 0;
}
```

Rethrowing an Exception

- If you wish to rethrow an expression from within an exception handler, you may do so by calling throw, by itself with no exception (i.e. with no data or data type)

- Why to do it → one catch may do part of exception handling & other catch might be doing other part of exception handling

- An exception can only be rethrown from within a catch block (or from any function called from with in that block) When you rethrow an exception, it will not be recaught by the same catch statement, it will propagate outward to the next catch statement.

```
// Example of "rethrowing" an exception.
#include <iostream>
using namespace std;
void Xhandler() {
  try {
    throw "hello"; // throw a char *
  }
  catch(const char *) { // catch a char *
    cout << "Caught char * inside Xhandler\n";
    throw ; // rethrow char * out of function
  }
}
int main()
{
  cout << "Start\n";
  try{
    Xhandler();
  }
  catch(const char *) {
    cout << "Caught char * inside main\n";
  }
  cout << "End";
  return 0;
}
o/p → start
caught char * inside Xhandler
caught char * inside main
End
```

Understanding terminate () and unexpected ()

void terminate ();
void unexpected ();

- **void terminate()**
  → This function is called whenever the exception handling subsystem fails to find a matching catch statement for an exception

  →It is also called if your program attempts to rethrow an exception when no exception was originally thrown.

- **void unexpected ();**
  →This function is called when a function attempts to throw an exception that is not allowed by its throw list.

Setting the Terminate & Unexpected Handlers By default

Terminate () –calls→ abort ()
Unexpected () –calls→ terminate () –calls→ abort ()

→ However, you can change the function that are called by terminate () & Unexpected(). Doing so allow your program to take full control of the exception handling subsystem.

→ To change the terminate handler use set_terminate () function.
terminate_hamdler set_terminate (terminate_handler *newhandler*) throw ();
//the function return the pointer to the new terminate //handler.

→New handler is a pointer to the new terminate handler. The newterminate handler must be of type terminate_handler which is defined like this

Typedef void (*terminate_handler) (); →pointer to a function returning void

→The only thing that your terminate handler must do is stop program execution. It must not return to program or resume it in any way similarly.

unexpected_handler set_unexpected (unexpected_handler *newhandler*) throw()
typedef void (*unexpected_handler) (); → returns pointer to a function returning void

Example

```
                    // Set a new terminate handler.
#include <iostream>
#include <cstdlib>
#include <exception>
using namespace std;
void my_Thandler() {
  cout << "Inside new terminate handler\n";
  abort();
}
int main()
{
                        // set a new terminate handler
  set_terminate(my_Thandler);
  try {
    cout << "Inside try block\n";
    throw 100;              // throw an error
  }
  catch (double i) {      // won't catch an int exception
    // ...
  }
  return 0;
}
```

- **bool uncaught_exception ();**
  - **the function returns true if an exception has been thrown but not yet caught. Once caught the function returns false.**

- **The exception & bad_exception classes**
  - When a function supplied by the C++ standard library throws an exception, it will be an object derived from the base class exception.

Exception handling (divide by zero)

```
#include <iostream>
using namespace std;
void divide(double a, double b);
int main() {
  double i, j;
  do {
    cout << "Enter numerator (0 to stop): ";
    cin >> i;
    cout << "Enter denominator: ";
    cin >> j;
    divide(i, j);
  } while(i != 0);
  return 0;
}
void divide(double a, double b)
{
  try {
    if(!b) throw b; // check for divide-by-zero
    cout << "Result: " << a/b << endl;
  }
  catch (double b) {
    cout << "Can't divide by zero.\n";
  }
}
```