Lecture Eleven

Template and Exception Handling

Ref: Herbert Schildt, Teach Yourself C++, Third Ed^{n} (Chapter 11)

© **Dr. M. Mahfuzul Islam**
Professor, Dept. of CSE, BUET

Generic Functions

- A general function defines a general set of operations that will be applied to various types of data.
- A general function is created using the keyword template.
- The general form of a template function definition is:

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

Ttype is a placeholder name for a data type used by the function.

- The keyword typename can be used instead of class.

```
template <typename Ttype> ret-type function-name(parameter list){
}
```

- The template portion of a generic function definition does not have to be on the same line as the function's name.

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

Generic Functions

- **Note that no other statement can occur between the template statement and the start of generic function definition.**

```
template <class Ttype>
int n;                  //Error
ret-type function-name(parameter list){
}
```

- **More than one generic type can be defined with the template statement, using a comma-separated list.**

```
template < class Ttype1, class Ttype2>
ret-type function-name( parameter list){
}
```

- **When you create a generic function, the compiler generate as many different versions of that function as necessary.**

Generic Functions

- If you overload a generic function, the overloaded function overrides the generic function.

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

template <class X>
void swapargs( X &a, X &b){
        X temp;
        temp = a;
        a = b;
        b = temp;
}

void swapargs(int a, int b){
         cout << “Overloaded Generic
                                    functions\n”;
}
```

```
int main(){
      int i = 10, j = 20;
      float x = 10.1, y = 23.3;

      cout << i << “  “ << j <<’\n’;
      cout << x << “  “ << y <<’\n’;
      swapargs(i, j);
      swapargs(x, y);
      cout << i << “  “ << j <<’\n’;
      cout << x << “  “ << y<<’\n’;

      return 0;
}
```

Generic Classes

- A ***generic class*** **defines all algorithms used by that class, but the actual type of the data being manipulated specified as a parameter.**
- **Generic classes are useful when a class contains generalizable logic. For example, the same algorithm that maintains a queue of integers will also work for a queue of characters.**
- **The general form of a generic class declaration is:**

```
template <class Ttype> class class-name{
};
```

- **A specific instance of a *generic class* is as follows:**

```
class-name <type> ob;
```

- **The Standard Template Library (STL) is built upon template classes.**
- **A template class can have more than one generic data type, comma separated.**

```
template <class Ttype1, class Ttype2> class class-name{
};
```

Generic Classes

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

```
template <class data_t> class list{
    data_t data;
    list *next;
public:
    list (data_t d);
    void add(list *node){
        node->next = this;
        next = 0;
    }
    list *getnext(){ return next; }
   data_t getdata(){ return data; }
};
```

```
template <class data_t>
list<data_t>:: list(data_t){
     data = d;
     next = 0;
}
```

```
int main(){
    list<char> start(‘a’);
    list<char> *p, *last;
    int i;

    last = &start;
    for(i =1; i < 26; i++){
        p = new list<char> (‘a’ + i);
        p->add(last);
        last = p;
     }

     p = &start;
     while(p){
        cout << p->getdata();
        p = p->getnext();
     }

     return 0;
}
```

Generic Classes

- Instead of char we can use int or struct as like:

```
 list<int> istart(1);
or
list<addr> ob(myaddr);
```

where

```
struct addr {
       char name[40];
       char street[40];
       char city[30];
       char state[3];
       char zip[12];
};

addr myaddr;
```

Generic Classes

- **Same program using stack of character and stack of integer:**

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

#define SIZE 10
```

```
template <class sType> class stack {
      sType stck[SIZE];
      int tos;
public:
     stack() { tos = 0; }
      void push( sType x);
      sType pop();
};
```

```
template <class sType>
void stack<sType>::push(sType ob){
     if (tos == SIZE){
          cout <<”stack is full\n”;
          return;
     }
     stck[tos] = ob;
     tos++
}
```

```
template <class sType>
sType stack<sType>::pop(){
       if (tos == 0){
             cout <<”stack is empty\n”;
             return 0;
       }
       tos--;
       return stck[tos];
}
```

```
int main(){
      stack<char> s1;
      stack<double> s2;
      stack<int> s3;

      s1.push(‘a’);
      s2.push(1.1);
      s3.push(5);

      cout << s1.pop()<<‘ ‘<<s2.pop() <<‘ ‘;
      cout << s3.pop();

      return 0;
}
```

Generic Classes

- A **template class** can have **more than one** generic data type, comma separated.

```
#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 text”);

       ob1.show();
       ob2.show();

       return 0;
}
```

Exception Handling

- **Exception handling is used to manage and respond to run-time errors.**
- **There are three keywords for exception handling: try, throw and catch.**
- **Program statements that to be monitored are contained in a try block.**
- **The general form of try is as follows:**

```
try {
}
```

- **If an exception (i.e., an error) occurs within the try block, it is thrown using throw. The general form of throw is as follows:**

```
throw exception;
```

- **The exception is caught using catch. The general form of catch:**

```
catch(type arg) {
}
```

- **If you throw an exception and there is no applicable catch statement for it, an abnormal program termination is occurred. The standard library function terminate() is invoked which call abort() to stop the program.**

Exception Handling

- After the **catch** statement **executes**, program control continues with the statements following the **catch**.
- The **type of exception** must match the type specified in a **catch** statement.
- An **exception** can be **thrown** from a statement that is **outside the try block** as long as the statement is within **a function** that is called within **try block**.

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

```
void Xtest(int test) {
      cout << “Inside Xtest: test ” <<test<<’\n’;
      if (test) throw test;
}
```

```
OUTPUT:
Start
Inside try block
Inside Xtest: test 0
Inside Xtest: test 1
Number 1
end
```

```
int main(){
      cout << “Start\n”;
      try{
           cout <<”Inside try block\n”;
           Xtest(0);
           Xtest(1);
      }
      catch( int i){
           cout << “Number” << i<<’\n’;
      }
     cout <<”end”;
     return 0;
}
```

Exception Handling

- **An Example:**

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

```
void Xtest(int test) {
       try{
            if (test) throw test;
            else throw “Zero”;
       }
      catch( int i){
            cout << “Number: “ << i <<’\n\;
      }
      catch( char *str){
            cout << str << ‘\n’;
      }
}
```

```
int main(){
      Xtest(0);
      Xtest(1);

       return 0;
}
```

OUTPUT:

```
Zero
Number: 1
```

More About Exception Handling

- ➢ **The following form can handle all exceptions instead of just a certain type:**

```
catch(...){
}
```

catch handling specific type of exception overrides the above form.

- ➢ **A function can throw exception out of it using the general form:**

```
ret-type func-name(arg-list) throw(type list) {
}
```

✓**Comma separated *type-list* may be thrown by the function.**

✓ **Throwing any other type of exception causes abnormal program termination. Standard library function unexpected() is called which causes terminate() function to be called.**

✓ **If no exception is thrown, the list is empty.**

More About Exception Handling

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

```
void Xtest(int test) throw(int, char, double) {
        if (test==0) throw test;
        if (test==1) throw ‘a’;
        if (test==2) throw 12.2;
}
```

```
int main(){
     try{
          Xtest(0);
          Xtest(1);
          Xtest(2);
     }

     catch( int i){
          cout << “Number: ” << i<<’\n’;
      }

      catch( … ){
          cout << “all exception\n ”;
      }

      return 0;
}
```

- **Empty type-list prevents throwing any exception:**

```
void Xtest(int test) throw() {
     if (test==0) throw test;
     if (test==1) throw ‘a’;
     if (test==2) throw 12.2;
}
```

More About Exception Handling

➢ **An exception can be rethrown from within a catch block. When an exception is rethrown, it will not be recaught by the same catch statement.**

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

void Xtest(){
      try{
          throw “hello”;
      }

      catch(char *){
          cout << “caught in Xtest\n”;
          throw ;
      }
}
```

```
int main(){
     try{
          Xtest();
     }

     catch(char *){
           cout << “caught in main\n’;
      }

      return 0;
}
```

Handling Exception Thrown by new

- **In modern C++, the new operator throw an bad_alloc exception if an allocation request is fails. To have access to this exception, <new> header must be included in the program.**
- **Previously, the new operator returns NULL.**
- **In modern C++, the following form returns NULL instead of throwing an exception.**

p_var = new(nothrow) type;

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

int main(){
     double *p;

     do{
          try{
               p = new double(1000000);
            // p = new (nothrow) double(1000000);
          }
```

```
          catch(bad_alloc xa){
               cout << “Allocation Failure\n’;
               return 1;
          }

          Cout << “Allocation is ok\n”;
     } while(p);

     return 0;
}
```