Generic Programming

CPE 212 -- Lecture 12

UAHuntsville

Function Overloading

Overloaded Functions - 1

- One function identifier (name) may refer to multiple functions so long as the parameter lists are distinct
- Example We Have Already Encountered:
    - Default Constructor vs Parameterized Constructor

      Stack() vs Stack(int maxsize)

    - Two different functions, sharing the same name but differing by their parameter lists

Overloaded Functions - 2

- The compiler identifies overload functions by a combination of the function name AND the data types and order in which the parameters appear
  - Return types are ***NOT*** sufficient to distinguish overloaded functions with identical names and parameter lists

Function Overloading

```
// Prototypes
void Print(int);
void Print(float);
void Print(int, int);
…
void Print(int someInt)
{
   cout << someInt << endl;
}
void Print(float someFloat)
{
   cout << setw(4) << setprecision(2)
        << someFloat << endl;
}
void Print(int someInt1, int someInt2)
{
   cout << "(" << someInt1 << ", "
         << someInt2 << ")" << endl;
}
```

Operator Overloading

Operator Overloading

- An operator's capabilities may be extended so that it can work with new objects defined by the programmer

- Overloading can clarify the code even though function calls may accomplish the same task
  - MatrixA = MatrixB + MatrixC*MatrixD;
  - MatrixA = MatrixAdd(MatrixB,MatrixMultiply(MatrixC,MatrixD));

- The functionality of an overloaded operator should mimic that of the built-in operator to avoid confusion

Operator Overloading Restrictions

- Operators which cannot be overloaded

  .   .*   ::   ?:

- Associativity of an operator cannot be changed by overloading

- Operators must be explicitly overloaded
  - Overloading **+** and **=** does not mean that **+=** has been overloaded
    - Object2 = Object2 + Object1;
    - Object2 += Object1; // No!!

- One argument of an operator must be an **object**

- Only existing operators may be overloaded

- See page 367 of your textbook for additional details

Operator Overloading Restrictions

- Trying to alter the use of an operator with built-in types causes a syntax error
- The keyword **operator** must be used immediately before the operator to be overloaded
  - Example: operator+
- Overloading does not change precedence, operator symbol, or number of operands

Operator Overloading - 1

```
// Example of a member function used to test the values of two Time objects for equality

bool Time::Equal( Time otherTime ) const
{
    return (hrs == otherTime.hrs && mins == otherTime.mins && secs == otherTime.secs);
} // End Time::Equal(…)
```

```
Sample Use of Above:
if ( Time1.Equal(Time2) )
  cout << “The times are equal .Equal” << endl;
```

Operator Overloading - 2

```
// Example of overloaded operator as a member function of the Time class

bool operator==( const Time otherTime ) const;    // In time.h

bool Time::operator==( const Time otherTime ) const           // In time.cpp
{
    return (hrs == otherTime.hrs && mins == otherTime.mins && secs == otherTime.secs);
} // End Time::operator==(…)


Sample Use of Above:
if ( Time1.operator==(Time2) )
  cout << "The times are equal .operator==" << endl;
```

Operator Overloading - 3

```
// Example of overloaded operator as a non-member function of the Time class

bool operator==( const Time& t1, const Time& t2 ) const
{
    return (t1.Equal(t2));
} // End operator==(…)


Sample Use of Above:
if ( Time1 == Time2 ) )
  cout << "The times are equal .operator==" << endl;
```

Operator Overloading - 4

```
// Example of overloaded operator definition as a FRIEND function of Time class

friend bool operator==( const Time& leftTime, const Time& rightTime ); // In time.h

bool operator==( const Time& leftTime, const Time& rightTime ) // In time.cpp
{
    return (leftTime.hrs == rightTime.hrs
          && leftTime.mins == rightTime.mins
          && leftTime.secs == rightTime.secs);
} // End operator==(…)


Sample Use of Above:
if ( Time1 == Time2 )
  cout << "The times are equal ==" << endl;
```

friend functions

- friend functions are declared within a class definition but they are not member functions of the class
- A friend function is able to directly access private members of the class

Function Templates

Function Templates

- ***Function Template***
  - A C++ construct that allows the compiler to generate multiple versions of a ***function*** by allowing parameterized types
  - At compile time, template arguments are substituted in place of the corresponding parameters creating multiple instances of the function

Function Template - 1

class

```
template <typename  ItemType>
ItemType  Square(ItemType  value)
{
    return  value * value;
}

cout << Square<int>(3) << endl;
cout << Square<float>(3.14) << endl;
```

Function Template - 2

```
template <typename  ItemType>
void  Swap(ItemType&  value1, ItemType& value2)
{
   ItemType  temp;
   temp = value1;
   value1 = value2;
   value2 = temp;
}
```

Usage:

```
Swap<int>(someInt, anotherInt);
Swap<StudentRec>(someStudent, anotherStudent);
```

Class Templates

Class Templates

- ***Generic Data Type***
  - A type for which the operations are defined but the data types of the items being manipulated are not

- ***Class Template***
  - A C++ construct that allows the compiler to generate multiple versions of a ***class type*** by allowing parameterized types

Generic Stacks Using Templates and Linked Lists

Complete Program

```
//*******  GStack.h Standard Header Information Here **********
#ifndef GSTACK_H
#define GSTACK_H

template <typename ItemType>                // or template <class ItemType>
struct NodeType;                            // Forward declaration

template <typename ItemType>
class GStack                                // Node-based Stack class
{
 private:
  NodeType<ItemType>* topPtr;               // Top of stack pointer

 public:
  GStack();                                 // Default constructor
                                            // Postcondition: Empty stack created

  ~GStack();                                // Destructor

  bool IsEmpty() const;                     // Checks to see if stack is empty
                                            // Postcondition: Returns TRUE if empty, FALSE otherwise

  bool IsFull() const;                      // Checks to see if stack is full
                                            // Postcondition: Returns TRUE if full, FALSE otherwise

  void Push(ItemType item);                 // Adds item to top of stack

  void Pop();                               // Removes top item from stack

  ItemType Top() const;                     // Returns a copy of top item on stack
                                            // Postcondition: item still on stack, copy returned

  void MakeEmpty();                         // Removes all items from stack
};

#endif
```

```cpp
//*******  GStack.cpp Standard Header Information Here **********
#include "gstack.h"
#include <cstddef>
#include <new>                                  // for bad_alloc

using namespace std;

template <typename ItemType>
struct NodeType
{
  ItemType info;
  NodeType<ItemType>* next;
};

//***************************************************************
template <typename ItemType>
GStack<ItemType>::GStack()                      // Default constructor
{                                               // Postcondition: Empty stack created
  topPtr = NULL;
}

//***************************************************************

template <typename ItemType>
GStack<ItemType>::~GStack()                     // Destructor
{
  NodeType<ItemType>* tempPtr;

  while ( topPtr != NULL )                      // Deallocate any nodes on the stack
  {
    tempPtr = topPtr;
    topPtr = topPtr->next;
    delete tempPtr;
  }
}

//***************************************************************
```

UAHuntsville

```
//*******  GStack.cpp continued above **********
template <typename ItemType>
bool GStack<ItemType>::IsEmpty() const  // Checks to see if stack is empty
{                                        // Postcondition: Returns TRUE if empty, FALSE otherwise
  return (topPtr == NULL);
}

//**************************************************************

template <typename ItemType>
void GStack<ItemType>::Push(ItemType item)        // Adds item to top of stack
{                                        // Precondition: stack is not full
  NodeType<ItemType>* tempPtr = new NodeType<ItemType>;
  tempPtr->info = item;
  tempPtr->next = topPtr;
  topPtr = tempPtr;
}

//**************************************************************

template <typename ItemType>
void GStack<ItemType>::Pop()             // Removes top item from stack
{                                        // Precondition: stack is not empty
  NodeType<ItemType>* tempPtr;
  tempPtr = topPtr;
  topPtr = topPtr->next;
  delete tempPtr;
}

//**************************************************************

template <typename ItemType>
ItemType GStack<ItemType>::Top() const  // Returns a copy of top item on stack
{                                        // Precondition: stack is not empty
  return topPtr->info;                   // Postcondition: item still on stack, copy returned
}
```

```cpp
//*******  GStack.cpp continued above **********

//*************************************************************

template <typename ItemType>
void GStack<ItemType>::MakeEmpty()       // Removes all items from stack
{
  NodeType<ItemType>* tempPtr;

  while ( topPtr != NULL )
  {
    tempPtr = topPtr;
    topPtr = topPtr->next;
    delete tempPtr;
  }
}

//*************************************************************

template <typename ItemType>
bool GStack<ItemType>::IsFull() const   // Returns true if there is no room for another ItemType
{                                       //  on the free store; false otherwise.
  NodeType<ItemType>* location;
  try
  {
    location = new NodeType<ItemType>;  // new raises an exception if no memory is available
    delete location;
    return false;
  }
  catch(std::bad_alloc)                 // This catch block processes the bad_alloc exception
  {                                     // should it occur
    return true;
  }
}
```

```cpp
//*******  GStackClient.cpp Standard Header Information Here **********
#include <iostream>
#include <fstream>
#include "gstack.h"


using namespace std;

int main()                    // Note:  Implementation changed but no change in client program!!
{
  // ***** Now create and use a stack of integers
  GStack<int> temps;
  ifstream datafile;
  int someTemp;

  datafile.open("June05Temps");

  cout << "Raw Data" << endl;
  datafile >> someTemp;
  while (datafile)
  {
    cout << someTemp << endl;
    if ( !temps.IsFull() )
    {
      temps.Push(someTemp);
    }
    datafile >> someTemp;
  }

  cout << "Stack Values" << endl;
  while ( !temps.IsEmpty() )
  {
    cout << temps.Top() << endl;
    temps.Pop();
  }
  datafile.close();
```

UAHuntsville

```
//*******  GStackClient.cpp continued above **********

  // ***** Now create and use a stack of characters
  GStack<char> text;
  char someChar;
  datafile.open("mytext.txt");

  cout << "Raw Data" << endl;
  datafile >> someChar;
  while (datafile)
  {
    cout << someChar << endl;
    if ( !text.IsFull() )
    {
      text.Push(someChar);
    }
    datafile >> someChar;
  }

  cout << "Stack Values" << endl;
  while ( !text.IsEmpty() )
  {
    cout << text.Top() << endl;
    text.Pop();
  }
  datafile.close();

  return 0;
}
```