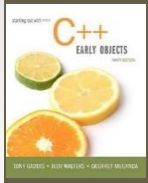
COP2334

Introduction to Object Oriented Programming with C++

D. Singletary

Module 12

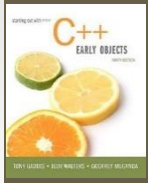
Ch. 16 Exceptions, Templates, and the Standard Template Library



Ch. 16 Exceptions

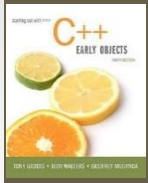
- Exceptions are used to signal errors or unexpected events that occur while a program is running
  - An exception is thrown to indicate an error has occurred:

```
double divide(double numerator, double denominator)
{
    if (denominator == 0)
        throw string("ERROR: divide by 0");
    else
        return numerator / denominator;
}
```



- The thrown argument can be of any type
  - The line containing the throw statement is the throw point
  - Any code following the throw is not executed
  - Throwing an exception transfer program control to an exception handler

```
try {
        quotient = divide(num1, num2);
        cout << "quotient = " << quotient << endl;
} catch (string exceptionString) {
        cout << exceptionString << endl;
}
```

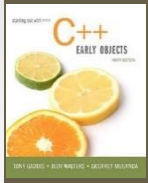


```
// testexception.cpp
// Demonstrates throwing and catching an exception

#include <iostream>
using namespace std;

double divide(double numerator, double denominator);

int main()
{
   double num1 = 10.0;
   double num2 = 3.0;

   double quotient = 0.0;

   try {
      quotient = divide(num1, num2);    // executes normally
      cout << "quotient 1 = " << quotient << endl;
      num2 = 0.0;
      quotient = divide(num1, num2);    // this aborts on the exception
      cout << "quotient 2 = " << quotient << endl;
   } catch (string exceptionString) {
      cout << exceptionString << endl;
   }

   return 0;
}

double divide(double numerator, double denominator)
{
      if (denominator == 0)
            throw string("ERROR: divide by 0");
      else
            return numerator / denominator;
}
```

```
quotient 1 = 3.33333
ERROR: divide by 0
```

Exercise

- Rewrite the previous program so that it throws a custom exception (create a class for this) instead of a string.



C++ Standard Exceptions

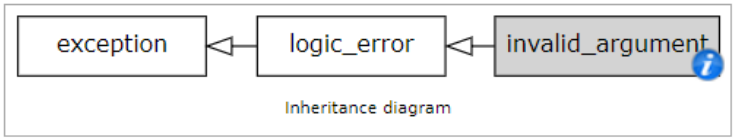
std::invalid_argument

Defined in header <stdexcept>

```
class invalid_argument;
```

Defines a type of object to be thrown as exception. It reports errors that arise because an argument value has not been accepted.

This exception is thrown by `std::bitset::bitset`, and the `std::stoi` and `std::stof` families of functions.



Inheritance diagram

Member functions

(constructor)	constructs the exception object (public member function)

std::invalid_argument::invalid_argument

```
explicit invalid_argument( const std::string& what_arg );    (1)
explicit invalid_argument( const char* what_arg );           (2)   (since C++11)
```

Constructs the exception object with `what_arg` as explanatory string that can be accessed through `what()`.

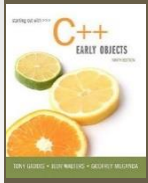
This message is typically stored internally as a separately-allocated reference-counted string, so that copying the exception object does not throw an exception. This is also why there is no constructor taking `std::string&&`: it would have to copy the content anyway.

Parameters

what_arg - explanatory string

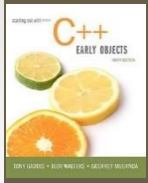
Templates

- C++ templates support generic programming, which refers to writing type-independent code
- Allows functions and classes to operate on many different types without rewriting for each type (code reuse)



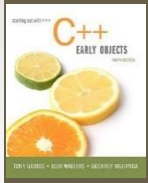
Function Templates

- A function template is a generic function which works with different data types
- Specifications (return type, parameters, implementation statements) are type-independent
- A function template uses type parameters to specify a generic data type
  - Compiler generates code to handle specific types



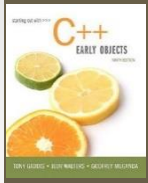
```
template <class T>
T square(T number)
{
      return  number * number;
}
```

- template prefix begins with the word "template"
- generic data type is enclosed in angle brackets, begins with the word "class"
  - multiple types can be used, separated by commas



```
template <class T>
T square(T number)
{
      return  number * number;
}
```

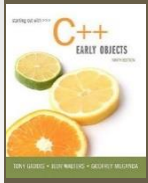
- Function definition is written as usual, but type parameters are substituted for actual type names
- This function returns the square of a number of an unspecified type (determined at compile time)



```
template <class T>
T square(T number)
{
      return  number * number;
}
```

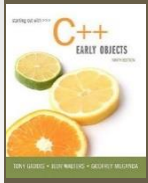
- Call the function normally; the compiler fills in applicable data type

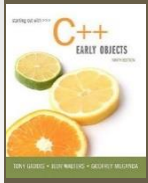
```
int x = 4;
int y = square(4);
cout << x << " squared is " << y << endl;
```



```
// template.cpp
// demonstrates templates

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

template <class T> // define a square function template
T square(T number)
{
      return number * number;
}

int main()
{
      int x = 4;
      int y = square(4);
      cout << x << " squared is " << y << endl;

      double d1 = 10.0;
      double d2 = square(d1);
      cout << fixed << setprecision(2);
      cout << d1 << " squared is " << d2 << endl;

      return 0;
}
```

```
4 squared is 16
10.00 squared is 100.00
```





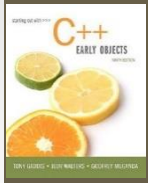
- Templates should be placed at the top of the source file or in a header
  - The compiler must know the template's contents before the function can be called
  - A template is only a specification, no memory is allocated until a call to the function is encountered
- All operators used in a template must be compatible with the type used in the call
- The "square" function template would not work with a string type

error: no match for 'operator*' (operand types are 'std::basic_string<char>' and 'std::basic_string<char>')

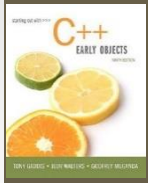
Class Templates

- Templates can be used to create generic classes and abstract data types
- Class templates are declared in a similar fashion to function templates:

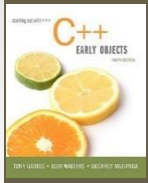
```
template <class T>
class ClassName
{ …
```



```
template <class T> // define a wrapper class for an array
class SimpleArrayClass
{
   private:
      int arraySize;
      T *array;

   public:
      SimpleArrayClass()
      {
         this->arraySize = 0;
         this->array = nullptr;
      }

      SimpleArrayClass(int arraySize)
      {
         this->arraySize = arraySize;
         this->array = new T[arraySize];
      }
```



```
    ~SimpleArrayClass()
    {
       delete[] this->array;
       this->array = nullptr;
       this->arraySize = 0;
    }

    int getArraySize() { return this->arraySize; }

    const T* getArray() { return this->array; }

    void setArray(const T srcArray[])
    {
       for (int i = 0; i < this->arraySize; i++)
          this->array[i] = srcArray[i];
    }

    void printArray()
    {
       for (int i = 0; i < this->arraySize; i++)
          cout << "array[" << i << "] = " << this->array[i] << endl;
    }
};
```

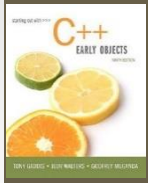


```
int main()
{
   const int SIZE = 5;

   SimpleArrayClass<int> intTable(SIZE);
   int iArray[SIZE] = { 10, 20, 30, 40, 50 };
   intTable.setArray(iArray);
   intTable.printArray();

   SimpleArrayClass<double> dubTable(SIZE);
   double dArray[SIZE] = { 100.0, 200.0, 300.0, 400.0, 500.0 };
   dubTable.setArray(dArray);
   cout << fixed << setprecision(2);
   dubTable.printArray();

   return 0;
}
```

```
array[0] = 10
array[1] = 20
array[2] = 30
array[3] = 40
array[4] = 50
array[0] = 100.00
array[1] = 200.00
array[2] = 300.00
array[3] = 400.00
array[4] = 500.00
```

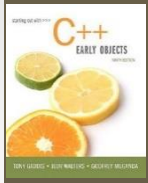
The Standard Template Library

- The STL contains templates for useful algorithms and data structures
- Containers
  - A class that stores and organizes data
  - Sequential and Associative
- Iterators
  - Provide access to items stored in containers
- STL is now part of the C++ Standard Library



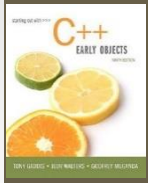
Sequential Containers

- Store items in a sequence
- Items are ordered by their position in the container
- vector
  - stores items as a dynamic (growable) array
- deque
  - sequenced items, most efficient insertions/deletions are from front and back
- list
  - efficient insertion/deletions from any position



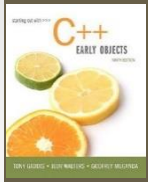
Vector Examples

```
#include <vector>
using namespace std;  // required for STL
...
// declare vectors which store integers
vector<int> numbersA; // empty
vector<int> numbersB(10); // starting size
vector<int> numbersC(10,2); // initial value of 2
vector<int> numbersD { 10, 20, 30, 40 }; // no "="

vector<string> names;
vector<char> letters(25, 'A');
```



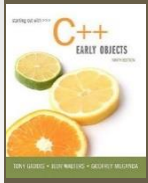
```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   const int SIZE = 5;

   vector<int> n(SIZE);

   for (int i = 0; i < SIZE; i++)
   {
      cout << "enter value " << i + 1 << ": ";
      cin >> n[i];
   }

   for (int i = 0; i < SIZE; i++)
   {
      cout << "n[" << i << "] = " << n[i] << endl;
   }

   return 0;
}
```

```
enter value 1: 10
enter value 2: 20
enter value 3: 30
enter value 4: 40
enter value 5: 50
n[0] = 10
n[1] = 20
n[2] = 30
n[3] = 40
n[4] = 50
```



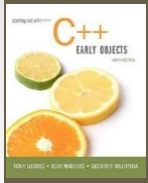
```
const int SIZE = 5;

vector<int> n(SIZE);

for (int i = 0; i < SIZE; i++)
{
   cout << "enter value " << i + 1 << ": ";
   cin >> n[i];
}

cout << "current vector size is " << n.size() << endl;

for (int i = 0; i < SIZE; i++)
   n.push_back(n[i] * 100);

cout << "new vector size is " << n.size() << endl;

for (int i = 0; i < n.size(); i++)
{
   cout << "n[" << i << "] = " << n[i] << endl;
}
```

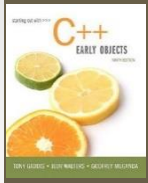
```
enter value 1: 10
enter value 2: 20
enter value 3: 30
enter value 4: 40
enter value 5: 50
current vector size is 5
new vector size is 10
n[0] = 10
n[1] = 20
n[2] = 30
n[3] = 40
n[4] = 50
n[5] = 1000
n[6] = 2000
n[7] = 3000
n[8] = 4000
n[9] = 5000
```



Vector Member Functions

(constructor)	constructs the `vector` (public member function)
(destructor)	destructs the `vector` (public member function)
`operator=`	assigns values to the container (public member function)
`assign`	assigns values to the container (public member function)
`get_allocator`	returns the associated allocator (public member function)
Element access	
`at`	access specified element with bounds checking (public member function)
`operator[]`	access specified element (public member function)
`front`	access the first element (public member function)
`back`	access the last element (public member function)
`data` (C++11)	direct access to the underlying array (public member function)
Iterators	
`begin` `cbegin`	returns an iterator to the beginning (public member function)
`end` `cend`	returns an iterator to the end (public member function)
`rbegin` `crbegin`	returns a reverse iterator to the beginning (public member function)
`rend` `crend`	returns a reverse iterator to the end (public member function)





Capacity

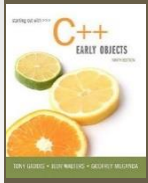
`empty`	checks whether the container is empty (public member function)
`size`	returns the number of elements (public member function)
`max_size`	returns the maximum possible number of elements (public member function)
`reserve`	reserves storage (public member function)
`capacity`	returns the number of elements that can be held in currently allocated storage (public member function)
`shrink_to_fit` (C++11)	reduces memory usage by freeing unused memory (public member function)

Modifiers

`clear`	clears the contents (public member function)
`insert`	inserts elements (public member function)
`emplace` (C++11)	constructs element in-place (public member function)
`erase`	erases elements (public member function)
`push_back`	adds an element to the end (public member function)
`emplace_back` (C++11)	constructs an element in-place at the end (public member function)
`pop_back`	removes the last element (public member function)
`resize`	changes the number of elements stored (public member function)
`swap`	swaps the contents (public member function)

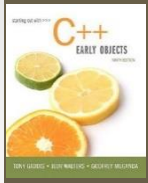
Associative Containers

- Items are stored and accessed using a key
  - e.g. telephone book has names (keys) and associated phone numbers
- set
  - a set of keys (no associated values); no duplicates are allowed
- multiset
  - a set which allows duplicate keys
- map
  - maps keys to data items (duplicated keys not allowed
- multimap
  - a map which allows duplicated items



Iterators

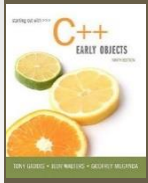
- Objects that act like pointers
- Used to access items stored in containers
- Iterator Types
  - Forward (++ operator)
  - Bidirectional (++ and -- operators)
  - Random Access



Iterator Example

```
const int SIZE = 5;

vector<int> n(SIZE);
int count = 1;

for (int i = 0; i < SIZE; i++)
{
   cout << "enter value " << i + 1 << ": ";
   cin >> n[i];
}

cout << "current vector size is " << n.size() << endl;

for (int i = 0; i < SIZE; i++)
   n.push_back(n[i] * 100);

cout << "new vector size is " << n.size() << endl;

auto iter = n.begin();
while (iter != n.end())
{
   cout << *iter << endl;
   iter++;
}
```

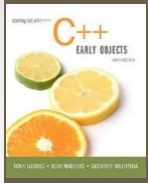
```
enter value 1: 10
enter value 2: 20
enter value 3: 30
enter value 4: 40
enter value 5: 50
current vector size is 5
new vector size is 10
10
20
30
40
50
1000
2000
3000
4000
5000
```



Range-Based For Loops

- C++ 11 gives us the "range-based for loop" (aka the "foreach" loop in Java). This can be used for STL containers a well as standard arrays

```
for (datatype var : container)
        // var takes value of each element

 for (int elem : n)
        cout << elem << endl;
```



Range-Based For Loops

```
const int SIZE = 5;

vector<int> n(SIZE);
int count = 1;

for (int i = 0; i < SIZE; i++)
{
   cout << "enter value " << i + 1 << ": ";
   cin >> n[i];
}

cout << "current vector size is " << n.size() << endl;

for (int i = 0; i < SIZE; i++)
   n.push_back(n[i] * 100);

cout << "new vector size is " << n.size() << endl;

for (int elem : n)
   cout << elem << endl;
```

```
enter value 1: 10
enter value 2: 20
enter value 3: 30
enter value 4: 40
enter value 5: 50
current vector size is 5
new vector size is 10
10
20
30
40
50
1000
2000
3000
4000
5000
```