TEMPLATES AND EXCEPTIONS

ERRORS

Concept a program can handle errors in several ways

Example

```
double divide(double a, double b) {
    if( b == 0 ) { exit(1); }                        // terminate with error value
    return a/b;
}

double divide(double a, double b) {
    assert(b!=0);                                    // terminate and print error
    return a/b;
}

int linearSearch(int *a, int size, int v) {
    for(int i=0; i<size; ++i) { if(a[i] == v) return i; }
    return -1;                                       // -1 indicates element not found
}
```

EXCEPTIONS

Concept a method of handling errors to improve and simplify the flow of code

Throw a signal that an exception (an error) has occurred

Try a block of code which has the potential for throwing an exception due to errors

Catch a block of code which handles thrown exceptions from the try block

Example

```
try {                                                   // code block to try
      if( b == 0 ) { throw “Cannot divide by 0”; }      // error to handle with message
      cout << a/b;                                      // code to run if b is not 0
}
catch( const char* e ) {                                // code to catch the exception
      cerr << “Error: “ << e << “\n”;                   // print the error message
}
```

ex1_exceptions.cpp

EXCEPTIONS AND FUNCTIONS

Concept a thrown exception terminates the function to reach the catch block

Example

```
double divide(double a, double b) {
    if( b == 0 ) { throw “Cannot divide by 0”; }     // error to handle with message
    return a/b;                                       // code to run if b is not 0
}

try {                                                 // code block to try
    cout << divide(a/b);                              // error to handle with message
}
catch( const char* e ) {                              // code to catch the exception
    cerr << “Error: “ << e << “\n”;                   // print the error message
}
```

ex2_exceptions.cpp

MULTIPLE EXCEPTIONS

Concept one or more exceptions can be handled by the same catch block

Example

```
double dividePositive(double a, double b) {
    if( b == 0 ) { throw “Cannot divide by 0”; }       // error to handle with message
    if( a<0 || b<0) { throw “Negative values”; }       // another error to handle
    return a/b;                                        // code to run if b is not 0
}

try {                                                  // code block to try
    cout << divide(a/b);                               // error to handle with message
}
catch( const char* e ) {                               // code to catch the exception
    cerr << “Error: “ << e << “\n”;                    // print the error message
}
```

ex3_muiltiple_exceptions.cpp

EXCEPTION CLASSES

Concept   a class for objects that are designed to be thrown as exceptions

Example

```
class DivByZero {                                          // exception class
private:
    const char* msg;                                       // stores error message
public:
    DivByZero(const char* msg): message(msg) { }           // construct an error object
    const char* getMsg() const { return msg; }             // return the error message
}

double divide(double a, double b) {
    if( b==0 ) { throw DivByZero(“Cannot divide by 0”); }  // throw a DivByZero
    return a/b;
}

try { cout << divide(a/b); }
catch( const DivByZero &e ) { cerr << e.getMsg(); }        // catch a DivByZero
```

ex4_exception_classes.cpp

THE NEED FOR TEMPLATES

Problem functions must be overloaded to support the same operation upon different types

Example

```
void swap(int &a, int &b) {          // function to swap integers
    int temp = a;
    a = b;
    b = temp;
}

void swap(char &a, char &b) {        // overloaded function to swap chars
     char temp = a;
    a = b;
    b = temp;
}
```

ex1_templates.cpp

TEMPLATE FUNCTIONS

Concept

functions which can be applied to many different types
type is decided by the function call
compiler converts the template function into a typed function before run time

Example

```
template<typename T>         // specify this is a template function with type T
void swap(T &a, T &b) {      // function to swap values of type T
    T temp;                  // create a variable of type T
    temp = a;
    a = b;
    b = temp;
}

int a=1, b=2;
swap(a, b);                  // call function to swap integer variables

string s=“Hi”, r=“Bye”;
swap(s, r);                  // call function to swap string variables
```

ex2a_templates.cpp

TEMPLATES CASTING

Concept any type T can only be one type, casting is not permitted

Example

```
template<typename T>          // specify this is a template function with type T
void swap(T &a, T &b) {       // function to swap values of type T
    T temp;                   // create a variable of type T
    temp = a;
    a = b;
    b = temp;
}

int a=1;
double b=2;
swap(a, b);                   // compiler error:
                              // T is int because of a
                              // T cannot be casted to double for b (already set as int)
```

ex2b_template_functions_type_casting.cpp

TEMPLATES MULTIPLE TYPES

Concept

template functions with support for multiple types in the same function
this supports multiple types as well as same types for flexibility

Example

```
template<typename T, typename U, typename V>       // specify three types T , U, V
V sum(T a, U b) {
      return a+b;
}

sum(5, 10.3);                                      // T int, U double, V double
                                                   // sum int double to return double

sum(“Hi”, “ There”);                               // T,U,V are all strings
                                                   // concatenate strings
```

ex2c_template_functions_multiple_types.cpp

TEMPLATES CLASSES

Concept a class which is not type specific due to the use of templates

Example

```
template <typename T>                 // identify class as a template class of type T
class Node {
private:
    T element;                        // store element of type T
public:
    Node(T e);                        // construct a node with element of type T
    T getElement() const;             // return the element of type T
    void setElement(T e);             // set the element of type T
}
template <typename T>                 // external definition requires template
void Node<T>::setElement(T e) {       // type is Node<T>, parameter is T
    element = e;
}

Node<int> n1(i);                      // type is Node<int> to store an integer
```

ex3_template_classes.cpp

TEMPLATES POLYMORPHISM

Example

```
template <typename T>                                     // declare type T
class ID {
    T idVal;                                              // store type T
public:
    ID(T v): idVal(v) {}                                  // constructor with type T
    virtual void output() const { cout << idVal; }        // virtual function
}
template <typename T, typename U>                         // declare type T and U
class StudentID: public ID<T> {                           // type ID<T>
    U name;                                               // store type U
public:
    StudentID(T i, U n): ID<T>(i), name(n) { }            // construct both objects
    void output() const { cout << idVal << name; }        // redefined
}
void print(const ID<T> &id) { id.output(); }              // non-member function

ID<int> *id = new StudentID<int, string>(1323, “John”);   // note the types
print(*id);                                               // pass by reference
```

ex4_template_polymorphism.cpp

TEMPLATE CONTAINERS

Concept

a container is an object which stores a collection or data structure of other objects
a container provides member functions to interact with data
a container is responsible for encapsulating memory management

Example

```
template <typename T>                                     // declare type T
class MyArray {
    T *p;                                                 // array pointer of type T
    int size;
public:
    MyArray(int s);
}

template<typename T>                                      // declare type T
MyArray<T>::MyArray(int s) {                              // type MyArray<T>
    if(s<=0) throw “Array size must be greater than 0”;   // throws an exception
    p = new T[size];
}
```

ex5_template_containers.cpp

TEMPLATE VALUES

Concept

pass by value recommended, pass by reference for specific applications
the class user decides to store objects or pointers to objects
support for polymorphism will require pass by reference

Example

```
template <typename T>
class MyArray {
    T *p;
    int size;
public:
    MyArray(int s);
}

MyArray<int> numbers(10);     // create an array of integers stored by value
                              // T is an integer, so we store integer variables

MyArray<obj*> objects(10);    // create an array of obj pointers
                              // T is a pointer, so we store pointers to obj objects
```