Class libraries

Container classes

- A class with the main purpose of holding objects
- The type `void *`
  - Useful to build arrays of pointers that point to an element of unknown type
  - Any pointer can be placed in a `void *` without a type mismatch warning
  - Does not keep information about the size of the element being pointed to
    - Cannot get the elements by dereferencing (the `*` operator)
    - Cannot go to the offset by adding a number to it (how many bytes is `p+1` away from `p`)
  - You have to use a typecast to go from a `void *` to an `X *`
  - Only three operations possible on a `void *` – copy, compare, cast
  - Interface for a generic stack class is

```
class stack
{
    // ...
    public:
        stack( void );
        void push ( void * );
        void * pop ( void );
        void * top ( void ) const;
        bool is_empty ( void ) const;
};
```

  - Using the class

```
employee staff[100];
stack estack;

estack.push ( staff+n );

employee e = *(employee *) estack.pop(); // cast is necessary to get the *
```

    - The code above is only storing the addresses of the structures and not the elements themselves

Using inheritance

- Inheritance can be used to provide a type-safe interface to stored `void *` pointers
- Let us assume that we want to stack matrices using the above stack class
- We'll call the derived class `matrix_stack`

```
class matrix_stack : public stack
{
    // no new data
    public:
        void push ( matrix * p ) { stack::push( p ); }
        matrix * pop ( void ) { return ( matrix * ) ( stack::pop() ); }
        matrix * top ( void ) const { return ( matrix * ) ( stack::top() ); }
};
```

- We have performed an inline replacement of functions in the derived class
- Notice that the method `is_empty` is inherited without change from the base class

- In the above example, we have created a derived class that merely provides a [type-safe] interface to the base class without actually performing any functions itself
- If you use inheritance, a virtual destructor must be placed in the derived class to perform the necessary cleanup, such as recycling a block in the free store

Libraries and Makefile

- `Makefile` to handle libraries of classes

```
CC = g++
DEBUGFLAG = -g

LIBS = libvec_math.a
CLASSES = vector.o matrix.o scalar_matrix.o
OBJS = main.o interface.o

vec_math:  ${OBJS} ${LIBS}
        g++ -o $@ ${OBJS} -L. -lvec_math

libvec_math.a: ${CLASSES}
        ar rc $@ ${CLASSES}
        ranlib $@

.SUFFIXES: .C .o

.C.o:
        ${CC} ${DEBUGFLAG} -c $<

clean:
        /bin/rm -f *.o vec_math
```

  - `ar` maintains groups of files combined into a single archive file
    - The archive file contains a set of member files and header information for each file
    - The option `r` replaces the named file in the archive
    - The option `c` creates the archive and also suppresses the default messages upon creation
  - `ranlib` converts each archive to a form that can be linked more rapidly by adding a table of contents to the beginning of the archive

Templates

- To use the generic containers for a specific data type, we need to provide handcrafted code such as the code for stack of matrices
- Templates provide a way to automate the above derivation of classes
- A template is used to specify a family of classes that differ only in a type
  - All type-safe stacks derived from the `stack` class stacking `void *` pointers are alike, except for the type of pointer they hold

- The equivalent implementation using templates is

```
template<class T>
class ptr_stack : public stack
{
    public:
        void push ( T * p ) { stack::push( p ); }
        T * pop ( void ) { return ( T * ) ( stack::pop() ); }
        T * top ( void ) const { return ( T * ) ( stack::top() ); }
}
```

- This template defines a family of `ptr_stack` classes
- For any type `T`, a specific pointer stack can be obtained by *instantiating* the template
- The class `ptr_stack<matrix>` stores `matrix *` pointers
- The class `ptr_stack<vec3d>` stores `vec3d *` pointers
- You can use the template as

```
ptr_stack<matrix> s;        // Declare s to be a stack of matrices
matrix *m;                  // Pointer to a matrix
s.push( m );                // Put the pointer to matrix on stack
```

- Templates can also have compile time constants
  - If you want to make a stack of actual objects (not pointers), you can do the following:

```
template<class T, int SIZE>
class stack
{
    T s[SIZE];              // Stack of type T using array-based implementation
    int sp;                 // Stack pointer

    public:
        stack ( void ) : sp( 0 ) {}   // Constructor function
        void push ( T );
        T pop ( void );
        T top ( void ) const;
        bool is_empty ( void ) const;
        bool is_full ( void ) const;
};
```

  - `int SIZE` provides compile time constant as a parameter to the template
  - Make a stack of integers using the above template

```
stack<int, 20> s;          // Stack of 20 integers
s.push( -10 );
```

  - We can define the member functions using the prototypes as

```
template<class T, int SIZE>
void stack<T, SIZE>::push ( T t )
{
    if ( sp < SIZE )
        s[sp++] = t;
}

template<class T, int SIZE>
T stack<T, SIZE>::pop ( void )
```

```
{
    if ( sp )
        return ( s[--sp] );

    static T def_val;        // Default value
    return ( def_val );
}

template<class T, int SIZE>
inline bool stack<T, SIZE>::is_empty ( void ) const
{
    return ( ! sp );
}
```

- In the `pop()` function, we cannot return 0 if the stack is empty because there is no way to convert 0 to a specified object type
- We return an object of type `T` constructed with the default constructor for the class
- By making the return value a `static` type, we do return a 0 for integer or double

• Class and member function templates must be placed in header files and not in source files since their code must be accessible to every module that instantiates them

Exceptions

• With the addition of every new class, you add new kinds of errors that need to be handled

• Upon error, you can abort the program or return to the caller and report the problem
  - Aborting the program is good only during testing
  - If you return to immediate caller, it may be powerless to do anything about it, and may have to return to its caller and report the error, and so on until a part of the code is reached that can handle the error competently
  - Consider the following code segment:

```
matrix matrix::operator+ ( const matrix& a, const matrix& b )
{
    matrix tmp( a.row, a.col );
    // ...
}
```

    * What if the memory is exhausted in the constructor of the `matrix` object?
    * Who should test that the constructor method succeeded?
      · Remember that the constructors have no return value and hence, cannot return an error code
      · May be we could tweak the constructor in such a way that it sets a global error variable or a special error data attribute in the `matrix` object
      · In either case, the method `operator+` will have to check for the error and return immediately to caller, setting another flag to be checked by the caller, and so on
      · This leads to a large section of the code to concern itself with error diagnostics that may never happen

• Exception handling in C++
  - Provides an elegant alternative to the above error diagnostic code
  - Separates the portion of the code that handles errors resulting in many benefits
    * The code does not get cluttered with error handlers
    * The code itself is more readable and to the point

    * You do not have to add code for error checking in every function
  - Designed to handle synchronous errors (divide by zero) but cannot handle asynchronous errors (disk I/O completion, mouse clicks, network message arrival)
    * Asynchronous errors are best handled by interrupt processing mechanism
  - Used in situations in which the system can recover from the error causing the exception
  - When to use exception handling
    * To process exceptions for program components that are not geared to handle those exceptions directly
    * To process exceptions from software components such as functions, libraries, and classes that are likely to be widely used, and where it does not make sense for those components to handle their own exceptions
    * For errors that must be processed in a different scope from where they occur
    * On large projects to handle error processing in a uniform manner project wide
    * Never use it for purposes other than error handling (leads to reduction in program performance)

- Throwing an exception
  - When an error is encountered that cannot be handled where it is detected, an exception is raised with the command

```
throw e;
```

  - `throw` terminates the regular program control flow, and the function throwing the exception does not return to its caller
  - An exception can throw any value, such as an integer or string

```
throw "Stack underflow";
```

  - It is common to declare special exception classes and to throw objects of those classes:

```
throw stack_error ( "Stack underflow" );
```

or

```
stack_error e ( "Stack underflow" );
throw e;
```

  - Error objects can carry an arbitrary amount of information from the locus of detection to the error handler
  - Fixing the problem of return value in case of stack underflow

```
template<class T, int SIZE>
T stack<T, SIZE>::pop ( void )
{
    if ( sp )
        return ( s[--sp] );
    else
        throw stack_error ( "Stack underflow" );
}
```

    * If stack is empty, the function throws an exception
    * It does not return a value to its caller and is not responsible for creating a return value
    * The statement following the exception is not executed
  - Another example to illustrate exception with out of memory error

```
string::string ( const string& s )
    : sp ( new char[ strlen ( s.sp ) + 1 ] )
{
    if ( ! sp )    // sp is NULL
        throw memory_error();
    strcpy ( sp, s.sp );
}
```

  - The point at which an exception is thrown is called the `throw` point
- Gaining control once an exception is detected – the `try/catch` clause
  - The block of code that may generate an error is enclosed in a `try` block
    * An exception is thrown in a `try` block in a function, or from a function called directly or indirectly from within the `try` block
  - The `try` block is immediately followed by one or more `catch` block(s).
    * Each `catch` block specifies the type of exception it can catch and handle
    * Each `catch` block contains an exception handler
    * If the exception matches the type of parameter in one of the `catch` blocks, the code in that `catch` block is executed; otherwise, the function `terminate` is called which by default calls `abort`
  - When an exception is raised by `throw`, program control leaves the `try` block and searches the `catch` blocks in order for an appropriate handler
  - If no exceptions are `thrown` in the `try` block, the exception handlers for that block are skipped and the program resumes execution after the last `catch` block
  - Example

```
// A simple exception handling example

#include <iostream.h>

class div_by_zero_err
{
    const char *msg;

    public:
        div_by_zero_err ( void )
            : msg ( "Divide by zero" )
        {}
        void print_msg ( void )
        {
            cout << msg << endl;
        }
};

float quotient ( int n1, int n2 )
{
    if ( n2 == 0 )
        throw div_by_zero_err();
    return ( float(n1) / float(n2) );
}

main()
{
    cout << "Enter two integers to get their quotient: ";

    int n1, n2;
    cin >> n1 >> n2;

    try
    {
        float r = quotient ( n1, n2 );
```

```
        cout << "The quotient is: " << r << endl;
    }

    catch ( div_by_zero_err error )
    {
        cout << "Error: ";
        error.print_msg();

        return ( 1 );
    }

    return ( 0 );
}
```

* The above program is to be compiled with the exception handling feature of the compiler enabled using the command

```
g++ -fhandle-exceptions -g -o foo foo.C
```