Templates & Container Classes (STL)

A template describes a "configurable" class

- Templates were introduced to support containers (collections)
  - Templates replaced multiple inheritance (also introduced to support containers) in this regard
- A container is an object that can hold other objects
  - Built-in type (int, char, float, etc.)
  - User-defined type (i.e., struct)
  - Object instantiated from a C++ class
    - E.g., Lists, Trees, Stacks, etc.
- The Smalltalk class library was duplicated in C++
  - OOPS library by Keith Gorlen at NIH — called the NIHCL (pronounced "nickel") library
- The Standard Template Library (STL) is the current standard

Templates

A generic data type

- Templates may be used with classes, member functions, and non-member functions
- A non-member function Example

```cpp
template <class T>
void swap(T& x, T& y)
{
    T temp = x;
    x = y;
    y = temp;
}
```

Stack Example

Template class

```cpp
template <class T>
class Stack
{
    private:
        T* stackmem;
        int stackptr;
        int stacksize;

    public:
        Stack(int s) : stackptr(0), stacksize(s) {}
        ~Stack();
        void Push(T data);
        T Pop();
};
```

Constructor / Destructor

```cpp
template <class T>
Stack<T>::Stack(int s)
{
    stackmem = new T[s];
}

template <class T>
Stack<T>::~Stack()
{
    delete[] stackmem;
}
```
Stack Example

Member functions

```cpp
template <class T>
void Stack<T>::Push(T data) {
    if (stackptr < stacksize)
        stackmem[stackptr++] = data;
    // assumes operator= is defined for T
}
```

```cpp
template <class T>
T Stack<T>::Pop() {
    if (stackptr > 0)
        return stackmem[--stackptr];
}
```

Usage

```cpp
#include <iostream>
#include "stack.h"
using namespace std;

void main(void) {
    Stack<int> S(10);
    S.Push(10);
    S.Push(20);
    S.Push(30);
    cout << S.Pop() << endl;
    cout << S.Pop() << endl;
    cout << S.Pop() << endl;
}
```

Template Arguments

Types and values

- May be class types
  - Multiple arguments are okay
  - May be built-in types
    - Become compile-time constants
    - Default values can be specified

```cpp
template <class T, int size = 100>
class Stack {
private:
    int stackptr;
    T stackmem[size];
public:
    Stack() : stackptr(0) { }
    void Push(T data);
    T Pop();
};
```

```cpp
void main() {
    Stack<int, 10> S;
    Stack<double> D;
    S.Push(10);
    S.Push(20);
    D.Push(2.7);
    D.Push(3.141495);
    cout << S.Pop() << endl;
    cout << D.Pop() << endl;
}
```

Multiple Class Arguments

No syntactical limit

```cpp
template <class IndexType, class BaseData>
class Array {
private:
    BaseData* arrayData;
    IndexType lIndex;
    IndexType hIndex;
public:
    Array(IndexType lo, IndexType hi);
    ~Array();
    BaseData& operator[](IndexType index);
    friend ostream& operator<<<ostream& io, Array<IndexType, BaseData>&>
};
```

Templates and Inheritance

Templates work with other C++ features

```cpp
template <class IndexType, class BaseData>
class SortArray : public Array<IndexType, BaseData> {
protected:
    int (*compare)(const void* x, const void* y); // pointer to a function
public:
    SortArray(IndexType lo, IndexType hi);
    ~SortArray();
    void Sort();
};
```

```cpp
void main() {
    SortArray<int, int> S(10);
    S.Push(10);
    S.Push(20);
    S.Push(30);
    cout << S.Pop() << endl;
    cout << S.Pop() << endl;
    cout << S.Pop() << endl;
}
```
Historical Error Handling

Not really object-oriented

- Deal with the error at the point detected
  - assert
  - if (expression) { statement }
- Return error information
  - Special return value (e.g., NULL)
  - Set a global error flag (e.g., errno)
- Use signal() and raise()
  - Limited user-definable signals
- Use setjmp() and longjmp()
- Constructed objects are not cleaned up

Logical Object Layering

A more object-oriented approach

- Object-oriented programs are a set of cooperating objects
- “Knowledge” of program is high at top, low at bottom
  - Lowest classes may be library code
- What action should a low-level object take when an error is detected?
  - Errors in lower classes are often passed up to higher classes for an appropriate response

Exception Handling

An overview

- C++ advanced error handling mechanism
- Exceptions are objects
  - Built-in types
  - Enumerated types
  - User created types (structs)
  - Class types
- Exceptions can not be ignored
  - Exceptions are thrown when an error is detected and caught higher in the stack where that error is dealt with

Declaring Exceptions

Programmer-created exceptions

class E_underflow
  { private:
    string error;
  public:
    E_underflow(string message) : error(message) {  }
    void show() { cerr << "STACK UNDERFLOW: " << error << endl; }
  };

class E_overflow
  { private:
    string error;
  public:
    E_overflow(string message) :  error(message) { }
    void show() { cerr << "STACK OVERFLOW: " << error << endl; }
  };

Throwing Exceptions

Announcing an error

template<class T>
void Stack<T>::Push(T data)
{  
  if (stackptr < stacksize)
    stackmem[stackptr++] = data;
  else
  {  
    string error = "Stack Overflow";
    throw E_overflow(error);
  }
}
template<class T>
T Stack<T>::Pop(void)
{  
  if (stackptr > 0)
    return stackmem[--stackptr];
  else
  {  
    string error = "Stack Underflow";
    throw E_underflow(error);
  }
}

The throw Keyword

Launching an exception

- Creates and constructs a new object
- The new object is "returned" to the calling scope
- Until caught, the object is passed to ever more global scope
- When the exception is caught, execution resumes in an exception handler
- If the exception is not caught, the program terminates
  - "Abnormal program termination"
Catching Exceptions

“The buck stops here”

- try block
  - Exactly one try-block
  - Encloses one or more statements that may cause an exception
  - try {...}
- catch blocks (exception-handlers)
  - One or more catch-blocks
  - Each block matches one exception
  - Encloses statements that deal with a specific exception
  - Each catch-block searched top to bottom until a match is found
  - catch (exception formal) {...}

try/catch Example

Anticipating potential problems

```cpp
try {
  S.Push(10);
  S.Push(20);
  S.Push(30);
}
catch(E_overflow OF) {
  OF.show();
  // do whatever is appropriate
}
```

```cpp
try {
  cout << S.Pop() << endl;
  cout << S.Pop() << endl;
  cout << S.Pop() << endl;
}
catch(E_underflow UF) {
  UF.show();
  // do whatever is appropriate
}
```

Exception Passing

Following the flow of control

- try (push())
  - catch (E_overflow E)
- push()
  - throw E_overflow

Termination vs. Resumption

The two theoretical models

- Exception handling theory has two basic models
  - Termination (the C++/Java model)
  - Resumption (following remedial action, execution resumes at the point where the exception was thrown)

```cpp
bool done = false;
while (!done) {
  try {
    /*might throw an exception*/; done = true;
  } catch (Bad B) {
    /* remedial action */
  } catch (Fatal F) {
    /* clean up/error message */; exit(1);
  }
}
```

Exception Specification

“Passing the buck”

- Explicit declaration of what exceptions a function may throw
- Part of the function declaration and appears after the argument list
- void f(int) throw (Up, Fit);
- f may throw either exception
- void f(int);
- f may throw any type of exception
- For backward compatibility
- void f(int) throw();
- f does not throw any exceptions

The `unexpected()` Function

Dealing with uncaught exceptions

- A pointer to a function that is called if an unadvertised exception is thrown
- Can be reset with `set_unexpected()`
  - Must `#include <except.h>`
  - New function must have no argument and a `void` return type
  - `set_unexpected()` returns previous function
- Set to `terminate()` by default
  - `terminate()` is also a pointer to a function (`abort()` by default)
  - It is called if an exception is uncaught
  - Changed with `set_terminate()` function
Catching Any Exception

Intervening

* Any exception can be caught with:
  * `catch(...) {}`
* Catches any exception, so put it at the end of the list of catch handlers
* The exception type is unknown
* A caught exception can be rethrown with `throw` without an argument:

```
catch (...) // ... are a part of the syntax
{       // handle the exception
  cerr << "Unknown exception\n";
  throw;
}
```

Exception Matching

Match an exception with the right handler

* "Nearest" handlers are tried first
* Handler searching stops with the first match
* Matches need not be perfect
  * An object or reference to a derived class will match a handler for a base class (objects are "sliced" however)
  * If a pointer is thrown, standard pointer conversions are used to match the exception
  * Automatic type conversions are not used for matching

Standard C++ Exceptions

Main classes

* Exceptions thrown by functions in the standard C++ library

```
exception  // Base class for all exceptions thrown in the library.
logic_error  // Derived from exception. Errors presumably detectable before program execution.
runtime_error  // Derived from exception. Errors presumably detectable only during program execution.
```

Derived from logic_error

Specialized exceptions

```
domain_error  // Violation of a precondition.
invalid_argument  // Invalid argument to throwing function.
length_error  // Attempt to produce an object whose length is greater than NPOS (largest value of type size_t).
out_of_range  // Function argument out of range.
bad_cast  // Invalid dynamic_cast (see "Run Time Type Identification" — RTTI).
bad_typeid  // Null pointer p in typeid(*p) (see RTTI).
```

Derived from runtime_error

Specialized exceptions

```
range_error  // Violation of a postcondition.
overflow_error  // Arithmetic overflow.
bad_alloc  // Failure allocating storage.
```

Exception Rules-of-Thumb

Basic guidelines

* Avoid exceptions for:
  * Asynchronous Events
  * Errors that can be handled where detected
  * Errors handled by hardware or the operating system (e.g., divide-by-zero)
  * Flow-of-Control
  * For very simple programs
* Always use exception specifications
* Start with standard exceptions
* Nest your own exceptions in the class they deal with
* Use exception hierarchies
* Use multiple inheritance