

# C++

## Templates, Exceptions, Namespaces

### Templates

---

#### Review

- We learned a new relationship between classes: “inheritance”
  - public inheritance shows “is a kind of” relationship, Triangle is a kind of Shape
  - “derived class” “inherits” data members and member functions from “base class”
    - may add more data members and re-define (“override”) inherited member functions
  - virtual functions are bound dynamically - to match dynamic type
    - `Shape * sp = new Triangle(); sp->draw(); // calls Triangle::draw() if draw() is virtual`
  - dynamic binding works only with pointers and references
  - polymorphic functions can be written by calling virtual functions
  - useful for building frameworks, like GUI, Java Applets in Browser
- 

#### Templates

- C++ is strongly typed
- sometimes it is useful to have type-independent definitions
  - generic containers (lists, stacks, dictionaries)
  - generic algorithms (sort, min, max, reverse, size, copy)
- C++ templates allow such genericity

- EG

```
int min( int a, int b )
{
    return a < b ? a : b;
}
double min( double a, double b )
{
    return a < b ? a : b;
}
```

- ---

**C solution**

```
# define min(a,b) ((a) < (b) ? (a) : (b))
```

- but may have problems with side effects

```
int i = min( j++, k++ );
```

- and you must be careful to fully parenthesize to avoid precedence problems
  - templates avoid these problems and allow programmers to specify algorithms once
- 

#### Template Functions

- template function definition

```
template
< typename Type >
Type min( Type a, Type b )
{
    return a < b ? a : b;
}
```

- Type stands for one (consistent) type for the template definition
- template function use

```
int main()
{
```

```
int x = min( 10, 20 ); // OK: Type is int
double y = min( 10.5, 20.8 ); // OK: Type is double
}
```

---

## General Template Form

- syntax

**template**

```
< templateParameterList >
functionOrMethodOrClassDefinition
```

- `templateParameterList` is comma separated list of

```
typename identifier // useful for containers of elements
class identifier // same as typename, but deprecated
int value // useful for array sizes or defaults
```

---

## Specializing a Template Function

- templates are typically general for any type
- sometimes this general case is inappropriate (or inefficient) for a special case
- e.g., `min` for two character strings
- you can specialize by defining the specialized function

```
char * min( char * a, char * b )
{
    return strcmp( a, b ) < 0 ? a : b;
}
```

- this function will be used for arguments of type `char *`
  - template will be used for any other type
- 

## Revised Function Resolution Algorithm

- get all non-template functions matching the call
  - if more than one, ambiguity error
  - if there is exactly one, select it
  - if none, examine all template instances
  - if more than one, ambiguity error
  - if exactly one match, select it (if it is not instantiated, first instantiate it)
  - if none, re-examine all non-templates to see if there is a match via type conversion
- 

## Parameter Matching

- non-const formal cannot match to a const actual,  
e.g., `strcat("Hello", "World"); // ERROR`
  - const formal can match to const or non-const actual,  
e.g., `int len = strlen("Hello"); char s[]="Hello"; int len = strlen(s); // BOTH OK`
  - const & formal can match to either non-const or const  
(it creates anonymous object for literals (e.g., 10 or "hello"))
  - & (reference) formal can only match to valid l-value,  
e.g., `int x=10, y=20; swap(x, y);` but not `swap(10, 20)`  
`const char const * p = "hello";`
- 

## XXXTemplate Classes

- useful for generic containers (and more)
- usually best to start by writing a specialized class first
- then, after testing, convert it into a template

E.g.,  
`class Stack`

```

{
    int len;
    int top;
}
•
    int * buf;
public:
    Stack( int capacity = 100 )
        : len( capacity ), top( 0 ),
          buf( new int [capacity] )
    {
    }
    ~Stack()
    {
        delete[] buf;
    }
    void push( int x )
    {
        buf[top++] = x;
    }
    int pop()
    {
        return buf[--top];
    }
    int size()
    {
        return top;
    }
};
•

```

---

**now make it into a template**

```

template
    < typename Type >
class Stack
{
    int len;
    int top;
    Type * buf;
public:
    Stack( int capacity = 100 )
        : len( capacity ), top( 0 ),
          buf( new Type [capacity] )
    {
    }
    ~Stack()
    {
        delete[] buf;
    }
    void push( Type x )
    {
        buf[top++] = x;
    }
    Type pop()
    {
        return buf[--top];
    }
}

```

```

    int size()
    {
        return top;
    }
};

```

---

## Defining Template Methods

- defining methods outside class can be painful
  - we want to move method definitions to a separate .cpp file
  - must prefix each method with template parameter specification
  - must qualify each name with Stack<Type>::
- 

### Stack.h file

```

template
    < typename Type >
class Stack
{
    int max;
    int len;
    Type * buf;
public:
    Stack( int capacity = 100 );
    ~Stack();
    void push( Type x );
    Type pop();
    int size();
};

```

---

### Stack.cpp file

```

template
    < typename Type >
Stack<Type>::Stack( int capacity = 100 )
    : max( capacity ), len( 0 ),
      buf( new Type [capacity] )
{
}
template
    < typename Type >
Stack<Type>::~~Stack()
{
    delete[] buf;
}
template
    < typename Type >
void Stack<Type>::push( Type x )
{
    buf[len++] = x;
}
template
    < typename Type >
Type Stack<Type>::pop()
{
    return buf[--len];
}
template

```

```

< typename Type >
int Stack<Type>::size()
{
    return len;
}

```

---

## Uses of the Template

- **auxiliary definitions**

```

#include <iostream>
#include "Stack.h"
#define ArrayLength(a) (sizeof(a) / sizeof(*a))
using namespace std;
typedef Stack<int> intStack;
typedef Stack<char> charStack;
typedef Stack<double> doubleStack;
template
< typename T >
void fill( Stack<T> & stk, T * a, int len )
{
    for ( int i=0; i < len; ++i )
        stk.push( a[i] );
}
template
< typename T >
void empty( Stack<T> & stk )
{
    while ( stk.size() > 0 )
        cout << stk.pop() << ' ';
    cout << endl;
}

```

---

- **The main**

```

int main()
{
    doubleStack dstack;
    charStack cstack;
    intStack istack;
    static double dlist [] = { 1.5, 2.5, 3.5, 4.5, 6.5, 9.98 };
    static char clist [] = "Hello";
    static int  ilist [] = { 0, 1, 2, 3, 4, 6, 7, 8, 9, 10 };
    fill( dstack, dlist, ArrayLength( dlist ) );
    fill( istack, ilist, ArrayLength( ilist ) );
    fill( cstack, clist, ArrayLength( clist ) );
    empty( dstack );
    empty( istack );
    empty( cstack );
}

```

---

- **the output**

```

9.98 6.5 4.5 3.5 2.5 1.5
10 9 8 7 6 4 3 2 1 0
o l l e H

```

---

## Template Parameter Defaults

- similar to default function parameters

- hypothetical example

```
template
< typename charType = char, int capacity = 100 >
class basic_string
{
    // can have a string of wchar_t too,
    // but default is char String
};
```

---

## Member Templates

- member functions (of non-template classes) may be templates
- often used for writing type conversion operators
- **wouldn't it be cool if they had defined the following:**

```
class ostream
{
    ostream & operator << ( const int & item )// for primitive types
    {
        // ...
    }
    // ... rest of specializations for fundamental (built-in) types
    template
    < class T >
    ostream & operator << ( const T & item )
    {
        item.print( *this );
        return *this;
    }
};
```

- then we could just write method print on our new class and get operator << for free

---

## Non-type Template Parameters

- template  
    < size\_t Bits > // Bits is an unsigned integer  
class bitset  
{  
    // ...  
};

- use

```
enum Day { Mon, Tue, Wed, Thu, Fri, Sat, Sun, numDays };
bitset < numDays > workDays;
workDays.set( Mon );
workDays.set( Wed );
workDays.set( Fri );
for ( int d = (int)Mon; d < (int)numDays; ++d )
    if ( workDays[d] )
        cout << "Have to work on << d << "th day of the week\n";
```

---

## Specializing a class Template

- specializations can be defined, usually for improved efficiency
- EG

```
template
< typename indexType, typename valueType >
class map
```

```

{
    // implemented as a balanced search tree for O(lg N) performance
};
template
    < int, typename valueType >
class map
{
    // implemented as an array for O(1) performance
};

```

---

## Partial Template Instantiation

- you can specialize part or all of the template

```

template
    < typename T, typename U >
class The_Class { .... };

```

- partial template specialization

```

template
    < typename U >
class The_Class<int,U> { ... };

```

- full template specialization

```

template
    <>
class The_Class<int, short> { ... };

```

- pay attention to the empty formal argument list!
- partial template instantiation is used in the STL, EG

```

template
    < typename T, typename Allocator = allocator<T> >
class vector
{
    // lots of stuff here, dynamic array of Ts
};

```

- partial template specialization for bool
- the idea is to use one bit per entry, but it is tricky...

```

template
    < typename Allocator = allocator<T> >
class vector<bool, Allocator>
{
    // lots of stuff here, bits packed in words
};

```

---

## Explicit Template Instantiation

- you can control where a specific template instantiation is created
- classes can be explicitly instantiated

```

template
    < typename T >
class List { ... };
...
template class List<int>; // creates List<int>

```

- template functions can be explicitly instantiated

```

template

```

```
< typename T >
T max( T const & x, T const & y ) { ... }
...
template int max( int const &, int const & );
    // creates max(int,int)
```

---

## Exceptions

- they are designed to support run-time error handling
- they handle only synchronous exceptions such as array range checks
- they don't handle asynchronous events like GUI events, interrupts, or signals
- they are also a non-local alternative to the return statement, but should be used with care
- Examples

```
Stack stk;   stk.pop();    // where stk is empty()
String s;    s[i] = s[j];    // where index is out of bounds
```

---

## Exception Specifications

- used to declare which exceptions a function or method may throw

```
void f()
    throw(std::bad_alloc, std::bad_cast)
{
    if (blah)
        throw std::bad_alloc;
    if (blahBla)
        throw std::bad_cast;
    do_something_else();
}
```

- default is that function may throw any exception

```
void g()
    // implied throw(...)
{
}
```

- to declare that a method throws no exceptions

```
void h()
    throw()
{
}
```

- violation of exception specification calls `unexpected()`
- `unexpected()` usually calls `terminate()`, unless `std::bad_exception` is in the spec, then it re-throws `std::bad_exception`
- a function never throws an exception not listed in its specification
- you can define `unexpected()` as a handler

---

## Example definition

- ```
class BaseException {};
class RangeException : public BaseException {};
class SizeException : public BaseException {};
class Vector (yet again!)
class Vector
{
    int * buf;
    int size;
protected:
```



```

bool inbounds(int i)
{
    return 0 <= i && i < size;
}
public:
static const max = 100;
Vector( int newSize )
    throw( SizeException )
{
    if ( newSize < 0 || newSize > max )
        throw SizeException();
    size = newSize;
    buf = new int[newSize];
    for (int i=0; i < newSize; ++i )
        buf[i] = 2 * i;
}
int & operator [] ( int i )
    throw( RangeException )
{
    if ( !inbounds(i) )
        throw RangeException();
    return buf[i];
}
};

```

---

### Use of Vector

```

int getInt( char * prompt )
    // implies throw( ... )
{
    cout << prompt << ": ";
    int i;
    cin >> i;
    if ( i < 0 )
        throw "Hello"; // throws a const char *
    return i;
}

```

---

### Use of Vector (cont.)

```

void testExceptions()
    throw() // says we throw no exceptions
{
    for ( ; ; )
        try {
            int size = getInt( "Enter a size" );
            Vector v1(size);
            int index = getInt( "Enter an index" );
            cout << v1[index];
        }
        catch ( const RangeException & e )
        {
            cout << "Index out of bounds\n";
        }
        catch ( const SizeException & e )
        {
            cout << "Size out of range 0..max\n";
        }
}

```

```

    }
    catch ( ... )
    {
        cout << "Got some unknown exception\n";
        throw; // rethrows the current exception
    }
    // calls unexpected() which calls terminate()
}

```

- **Use of Vector (cont.)**

```

int main()
{
    while (true)
        try {
            testExceptions();
        }
        catch ( const BaseException & e )
        {
            cout << "Got some kind of BaseException\n";
        }
}

```

- **Another Exception Example (with various type exceptions)**

```

#include <iostream>
using namespace std;
void f(int i)
    throw(int, string, char)
{
    cout << "Entering f, i = " << i << endl;
    switch (i)
    {
        case 0:
            break;
        case 1:
            throw 10;
        case 2:
            throw string("hello");
        default:
            throw 'A';
    }
    cout << "Leaving f, i = " << i << endl;
}

```

```

int main()
{
    for (int i=0; i<4; i++)
        try {
            cout << "before call to f i = " << i << endl;

```

```

    f(i);
    cout << "after call to f i = " << i << endl;
}
catch (int i)
{
    cout << "caught int exception i = " << i << endl;
}
catch (char c)
{
    cout << "caught char exception c = " << c << endl;
}
catch (string s)
{
    cout << "caught string exception s = " << s << endl;
}
cout << "Leaving main"<< endl;
}

```

/\* output of this program:

```

before call to f i = 0
Entering f, i = 0
Leaving f, i = 0
after call to f i = 0
before call to f i = 1
Entering f, i = 1
caught int exception i = 10
before call to f i = 2
Entering f, i = 2
caught string exception s = hello
before call to f i = 3
Entering f, i = 3
caught char exception c = A
Leaving main
*/

```

[Visualize](#)

## Namespaces

- allows organization of the global name space (scope)
- necessary support for development of [many standard libraries](#)
- EG

```

namespace mySpace
{
    class Stack;
    ostream & operator << ( ostream & out, const Stack & stk );
    void myStackFunction( Stack s );
}

```

- a namespace may be split across files - they are cumulative  
**namespace** mySpace  
{  
    void myTopLevelFunction(); // adds to mySpace  
}
  - all C++ standard library identifiers are in namespace **std**
- 

## Using Namespaces

- qualified use of entities in a namespace  
`mySpace::Stack stk;`  
`mySpace::myTopLevelFunction();`  
`myStackFunction( stk ); // OK: argument is a mySpace::stk`  
`std::cout << stk; // OK: stk is of type mySpace::stk`
- un-qualified use of entities in a namespace  
**using** mySpace::Stack;  
**using** std::cout;  
`Stack stk;`  
`myStackFunction( stk ); // OK: argument is a mySpace::stk`  
`cout << stk; // OK: both arguments are directly visible`
- open up a namespace (everything is directly visible)  
**using** namespace std;  
**using** namespace mySpace;  
`Stack stk;`  
`myTopLevelFunction();`  
`myStackFunction( stk );`  
`cout << stk;`
- never use “using” when the context isn't clear
  - in a .h file
  - or in another namespace (makes them part of that namespace)