

ANSI C++ Standard Library

Jonathan Hoyle

Eastman Kodak

12/21/00

ANSI C++ Standard Library

✓ Purpose:

- To try out *Krispy Kreme Doughnuts*™
- To introduce new C++ library classes

✓ Topics:

- Standard C Library
- Header file usage
- ANSI C++ standard classes
- Q & A



Standard C Libraries

✓ Standard C Library contains **functions**:

- <stdio.h>: printf(), scanf(), fopen()
- <stdlib.h>: malloc(), free(), atoi()
- <string.h>: memcpy(), strcpy(), strcat()
- <math.h>: sin(), log(), floor(), ceil()
- <stddef.h>: size_t, wchar_t, NULL
- Others:
 - <assert.>, <ctype.h>, <errno.h>, <float.h>, <limits.h>, <locale.h>, <setjmp.h>, <signal.h>, <stdarg.h>, <time.h>



C++ Header Files

✓ Two flavors of headers: <xxx> vs. <xxx.h>

- <xxx> requires the std namespace:

```
#include <iostream>
std::cout << "Hello, World!" << endl;
```

- <xxx.h> already includes the namespace:

```
#include <iostream.h>
cout << "Hello, World!" << endl;
```

✓ C Headers:

- <xxx.h> also available as <cxxx>:

```
#include <stdlib.h>
#include <cstdlib>
```



ANSI C++ Classes

✓ Standard C++ Library contains **classes**:

- cout & other iostream classes
- string
- complex<>
- auto_ptr<>
- valarray<>
- STL containers:
 - list<>, vector<>, deque<>, queue<>, map<>,
 - stack<>, set<>



cout & other iostream classes

- ✓ `cout` is not an operator
- ✓ `cout` is a global variable of type ostream
- ✓ “`cout << variable`” streams variable to the standard output, similar to `printf(variable)`
- ✓ “`cin >> variable`” streams variable to the standard input, similar to `scanf(variable)`
- ✓ `cerr`, `clog` outputs errors
- ✓ `wcout`, `wcin`, `wcerr` and `wclog` are widestream equivalents



cout flags

```
cout << hex << 100;           // prints "64"  
cout << dec << 0x100;        // prints "256"  
cout << oct << 100;          // prints "144"  
cout << fixed << 100.0;      // prints "100.00"  
cout << scientific << 100.0;  // prints "1.0e+02"  
cout << boolalpha << (1==2); // prints "false"  
  
const double pi = 3.1415926535897932;  
cout.precision(12);  
cout << pi;                  // prints "3.14159265359"  
cout.precision(5);  
cout << pi;                  // prints "3.1416"
```



Adding cout capabilities to a class

- ✓ Given a class X, containing say 2 int's:

```
ostream &operator <<(ostream &outStream,
                           const X &var)
{
    outStream << var.getInt1() << var.getInt2();
    return outStream;
}

// Now I can print myVariable
X myVariable;
cout << myVariable << endl;
```



ANSI string class

- ✓ Comes for free when #including <string.h>
- ✓ Generic string class with expected methods:
 - constructors from: `char *`, `string`, `default`
 - operators: `+`, `+=`, `[]`
 - `size()`, `length()`, `clear()`, `copy()`
 - `append()`, `insert()`, `replace()`, `find()`
- ✓ `wstring` is the wide string equivalent
- ✓ Related: `basic_string< >`



ANSI complex<> class

- ✓ Templatized to allow degrees of precision:
`complex<float>`, `complex<double>`, etc.
- ✓ `real()` and `imag()` methods
- ✓ All the usual operations: `+`, `-`, `*`, `/`, `==`, `!=`, ...
- ✓ All `<math.h>` functions with complex
prototypes: `sin()`, `cos()`, `log()`, `pow()`, ...
- ✓ Specialized complex functions: `arg()`,
`norm()`, `conj()`, `polar()`, ...



auto_ptr<>

- ✓ `auto_ptr< >` is a “somewhat smart” pointer
- ✓ An `auto_ptr` automatically deletes memory it has ownership over when it destructs
- ✓ Copying one `auto_ptr` to another transfers ownership to the new `auto_ptr`
- ✓ `auto_ptr`'s point to objects created with `new`
- ✓ `auto_ptr`'s do not point to arrays of objects
- ✓ `auto_ptr< >` calls `delete` on destruction



auto_ptr<> Examples

```
class X { ... };

auto_ptr<X> ptr1(new X);           // ptr1 owns this X
auto_ptr<X> ptr2 = ptr1;          // ptr2 now own it

ptr2->xMethod();                // access methods
ptr1->xMethod();                // ptr1 still may access
x2 = *ptr2;                      // dereference

// auto_ptr methods
xPtr = ptr2.get();               // get the original ptr
ptr2.release();                  // releases ownership
delete ptr1.get();               // deletion now OK
ptr2.reset(new X);               // reset to another X
```



auto_ptr<> Misuses

```
auto_ptr<X> ptr1(new X[10]);      // can't use arrays
auto_ptr<X> ptr2 = new X;          // non-explicit ctr

X *dPtr = ptr1.get();             // OK assignment
delete dPtr;                     // deleting owned memory!

ptr1.reset(new X);               // OK, resetting new ptr
if (someCondition)
{
    ptr2 = ptr1;                 // ownership transferred
    DoStuff(ptr1);              // OK to still use ptr1
}
ptr1->xMethod();                // ptr1 not valid!
```



Key points to remember for `auto_ptr`

- ✓ Multiple `auto_ptr`'s can point to an object, but *exactly one* is the designated owner.
- ✓ When the owning `auto_ptr` is destructed, it deallocates what it points to (via `delete`).
- ✓ Copying an `auto_ptr` transfers ownership from the copied ptr to the copying ptr.
- ✓ `auto_ptr`'s must be initialized with a ptr to memory created with `new`, not `new[]`, not `malloc()`, not `GlobalAlloc()`, etc.



valarray<>

- ✓ A **valarray**< > is an array treated as a value
- ✓ It has traditional array access: `myValarray[i]`
- ✓ It is optimized for numeric calculations
- ✓ Manipulate **valarray**'s as wholes:

```
va3 = va1 + va2;           // va3[i] = va1[i]+va2[i];  
va3 = 10 * va1;            // va3[i] = 10 * val[i];  
vab = (va1 < 0);          // vab[i] = (va1[i] < 0);  
va3 = sin(va1);            // va3[i] = sin(va1[i]);  
va3 = va1.apply(f);        // va3[i] = f(va1[i]);  
va3 *= val1;               // va3[i] *= val1[i];  
va3 <<= 8;                 // va3[i] <<= 8;
```



Constructing valarray<>'s

```
valarray<int>      v1;           // default constructor
valarray<int>      v2(100);       // 100 sized, uninit
valarray<int>      v3(0,100);     // 100 sized, init to 0
valarray<string>   v4("Hi",5);    // strings set to "Hi"
valarray<char>     v5(ptr,256);   // init to a raw ptr
valarray<char>     v6 = v5;       // copy constructor
```

// Using other methods

```
size_t aSize = v1.size();          // get the array size
int    aMin  = v1.min();           // min value in array
int    aMax  = v1.max();           // max value in array
int    aSum  = v1.sum();           // array sum
v1.resize(250);                  // resize array
v2 = v1.shift(10);               // shifted array
```



valarray<> slices

- ✓ `slice`'s allow you to take arbitrary subsets
- ✓ Example: suppose you have a pointer p to RGB interleaved data. Let us retrieve the red channel as a contiguous plane.

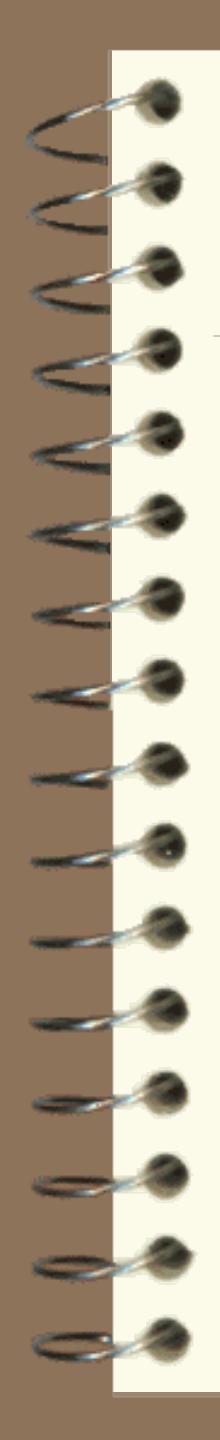
```
valarray<char>    imageArray(p,768); // image array
valarray<char>    redArray(256);      // red array

// start slice at 0, make 256 entries, jump by 3
slice      redSlice(0, 256, 3);

// pass slice into array [ ] operator
redArray = imageArray[redSlice];

redArray = imageArray[slice(0, 256, 3)];
```





STL History

- ✓ STL = “Standard Template Library”
 - ✓ Created by Alexander Stepanov to develop “uncompromisingly generic algorithms”
 - ✓ Implemented in other languages (Ada, etc.) before it was ported to C++ in 1994
 - ✓ Most of STL was accepted by the ANSI/ISO committee to be part of C++
 - ✓ Design favors efficiency over readability
- 

STL Overview

- ✓ STL consists of these components:
 - Containers (*linked list vs. array vs. stack-based, etc.*)
 - Iterators (*generalization of ptrs to iterate through containers*)
 - Algorithms & Functions (*container-independent fcns*)
 - Adaptors (*classes which act like functions, won't discuss*)
- ✓ Since ANSI approval, it is technically no longer “STL”; it is just part of the ANSI Standard Library



STL Philosophy

- ✓ STL classes do not follow an O.O. design; they are designed for performance:
 - No common base class among the containers
 - Separate specialized containers & iterators
 - Each container has its own iterator type
 - No special range or type checking is done
 - Methods specific to class capabilities, e.g.
 - `vector< >` has [] accessor but `list< >` does not
 - Documented high performance



STL Containers

✓ STL containers are specialized “list-like” classes, optimized for specific use:

- **vector**< >: *one dimensional array*
- **list**< >: *doubly-linked list*
- **queue**< >: *first-in/first-out model*
- **stack**< >: *last-in/first-out model*
- **deque**< >: *double-ended queue*
- **map**< >: *mapping between two types*
- **set**< >: *non-indexed collection*
- *Other containers may be user-defined*



STL Container Examples

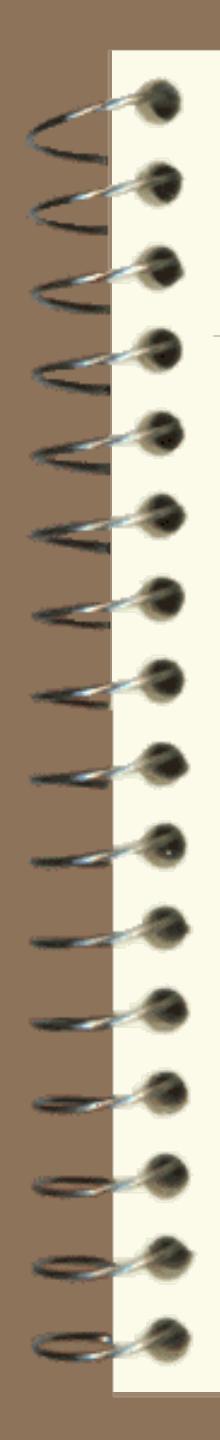
```
vector<int> v(100);           // 100 item vector
deque<int> d(100);           // 100 item deque
list<int> l(100);            // 100 item list

v[50] = 256;                  // OK
d[50] = 256;                  // OK
l[50] = 256;                  // Error: no [ ] on lists

l.push_front(10);             // OK
d.push_front(10);             // OK
v.push_front(10);             // Error: no push_front()

v.pop_back(25);               // OK
d.pop_back(25);               // OK
l.pop_back(25);               // OK
```





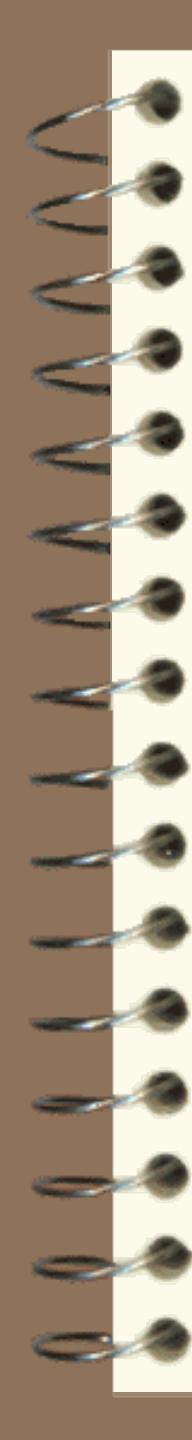
STL Iterators

✓ STL iterators are generalizations of pointers, used to iterate through containers:

- Input Iterators
- Output Iterators
- Forward Iterators
- Bidirectional Iterators
- Random Access Iterators
- *Other iterators may be user-defined*

✓ Not all iterators are available to all containers





STL Algorithms

- ✓ STL Algorithms are designed to be written from a container-independent, data type independent point of view:
 - sort(), search(), equal(), find(), swap(), replace(), remove(), merge(), ...
- ✓ STL Algorithms operate on iterators (pointers into the containers)
- ✓ A given algorithm is container-independent; only the iterator type is important.



STL Function Example

```
// Function to erase 0's from a given container
template <class T>
void EraseZeroes(T &container)
{
    T::iterator p = find(container.begin(), container.end(), 0);
    while (p != container.end())
    {
        container.erase(p);
        p = find(++p, container.end(), 0);
    }
}

// We can call EraseZeroes() on each of these different objects
vector<int>      v;      ...      EraseZeroes(v);
deque<char>        d;      ...      EraseZeroes(d);
list<double>       l;      ...      EraseZeroes(l);
stack<void *>      s;      ...      EraseZeroes(s); //Error!
```



vector<> or valarray<> ?

- ✓ Both are optimized array containers
- ✓ Both allow random access via []
- ✓ A `vector< >` has specialized methods for STL functions, such as `begin()`, etc.
- ✓ A `valarray< >` is usually treated as a single entity rather than as a container
- ✓ A `valarray< >` is usually a numeric type used for computational purposes, while a `vector< >` can be any generalized class



Q & A