cis15
advanced programming techniques, using c++
fall 2007
lecture # VII.1

topics:

- generic programming
- templates
- STL (standard template library)

on-line reference:

- `http://www.cppreference.com/index.html`

generic programming

- methodology for enhancing code reuse
- three techniques in C++:
  - generic (`void *`) pointers
  - templates
  - inheritance

- compare the following:

```
int transfer1( int from[], int to[], int size ) {
  for ( int i=0; i<size; i++ ) {
    to[i] = from[i];
  }
  return( size );
}
```

versus:

```
int transfer2( void* from, void* to, int size, int elementSize ) {
  int numBytes = size * elementSize;
  for ( int i=0; i<numBytes; i++ ) {
    static_cast<char *>(to)[i] = static_cast<char *>(from)[i];
  }
  return( size );
}
```

- if you have:

```
int a[10], b[10];
double c[10], d[10];
```


you can only call `transfer1()` with the `int` arrays:

```
transfer1( a, b, 10 );
//  transfer1( c, d, 10 ); WON'T COMPILE!
```

- but you can call `transfer2()` with

```
transfer2( a, b, 10, sizeof( int ));
transfer2( c, d, 10, sizeof( double ));
```

- hence, `transfer2()` is the *generic* version of the function because you can call it with arrays of any simple data type

- another way to write a generic function (like `transfer2()`) is using a C++ feature called a *template*

```
template<class T>
int transfer3( T* from, T* to, int size ) {
  for ( int i=0; i<size; i++ ) {
    to[i] = from[i];
  }
  return( size );
}
```

- `template` is a C++ keyword that implements something called *parametric polymorphism*
- which basically means that you can replace the template class type, in this case `T`, to any data type
- you could call `transfer3()` with either `int` or `double` arrays

stack example, using a template.

- here is an example of a generic stack, using a template and a version of the stack class we defined earlier this term:

```
template <class TYPE>
class stack {
  public:
    explicit stack( int size=100 ) : max_len(size), top(EMPTY),
                                     s( new TYPE[size] )
                                     { assert( s != 0 ); }
    ~stack() { delete []s; }
    void reset() { top = EMPTY; }
    void push( TYPE c ) { s[++top] = c; }
    TYPE pop() { return s[top--]; }
    TYPE top_of() const { return s[top]; }
    bool empty() const { return( top == EMPTY ); }
    bool full() const { return( top == max_len - 1 ); }
  private:
    enum { EMPTY = -1 };
```



```
    TYPE *s;
    int max_len;
    int top;
};
```

- the identifer TYPE is the generic template argument and is replaced when a variable of this type is declared, e.g.:

```
stack<char>   stk_ch;
stack<char *> stk_str(200);
stack<point>  stk_point(10);
```

- the template saves writing essentially the same code to operate on data of different types
- code snippet using stack template to reverse an array of strings:

```
void reverse( char *str[], int n ) {
  stack<char *> stk(n);
  int i;
  for ( i=0; i<n; ++i ) {
    stk.push( str[i] );
  }
```



```
  for ( i=0; i<n; ++i ) {
    str[i] = stk.pop();
  }
}
```

here's a `main()` to go with it:

```
int main( int argc, char *argv[] ) {
  int i;
  cout << "before:\n";
  for ( i=0; i<argc; i++ ) {
    cout << argv[i] << endl;
  }
  reverse( argv, argc );
  cout << "\nafter:\n";
  for ( i=0; i<argc; i++ ) {
    cout << argv[i] << endl;
  }
} // end of main()
```

- if you run the above example, you should enter command-line parameters; the program will print them out in the order they were entered, then run `reverse()` to invert the order of the parameters and print them again, using the new order

- for example:

```
unix-prompt$ ./a.out abc def 123
before:
./a.out
abc
def
123

after:
123
def
abc
./a.out
```


- you can either declare functions in-line or externally; the latter can get awkward but still works
- in-line examples:

```
TYPE top_of() const { return s[top]; }
void push( TYPE c ) { s[++top] = c; }
bool empty() const { return( top==EMPTY ); }
```

- external examples for the same function definitions:

```
template<class TYPE> TYPE stack<TYPE>::top_of() const {
  return s[top];
}

template<class TYPE> void stack<TYPE>::push( TYPE c ) {
  s[++top] = c;
}

template<class TYPE> bool stack<TYPE>::empty() const {
  return( top==EMPTY );
}
```


function templates

- function templates are safer than macros (`#define`)
- in fact, macros are out of fashion nowadays
- but here is one just in case you've never seen one:

```
#define CUBE(X) ((X)*(X)*(X))
```

- which would become:

```
template<class TYPE>
TYPE cube( TYPE n ) {
  return n * n * n;
}
```

- versus class templates, like the earlier stack example where
`template <class TYPE>`
goes before the class declaration as opposed to preceding the function defintion

Standard Template Library

- the STL or standard template library is a collection of useful templates that are part of the C++ standard namespace
- in order to use each template in the STL, you need to include the appropriate header file
- for example, in order to use the `vector` template, you need to do:

```
#include <vector>
using namespace std;
```

- the STL supports a variety of *data structures* and numerical algorithms that are beyond the scope of this class to discuss in detail
- the next few slides provide an overview to what is available
- for more detail, read chapters 6 and 7 in the Pohl textbook
- a very handy online reference is here:
`http://www.cppreference.com/cppstl.html`

containers

- containers are classes that store groups of like elements
- kind of like fancy, more capable arrays
- there are two types of containers:
  - *sequence* containers
    which are: `vector`, `list`, `deque`
  - *associative* containers
    which are: `set`, `multiset`, `map` `multimap` and `bitset`
- all containers have a shared *interface* (i.e., the public functions); these are:
  - constructor and destructor
  - functions to access, insert and delete elements
  - iterators (explained ahead)

sequence containers: vector

- a `vector` is like an array
- but it can also handle dynamic expansion
- which means that it won't overflow
- it can be navigated using either an index (like an array) or an iterator (more ahead on iterators)
- example:

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<int> V(10);
  for ( int i=0; i<10; i++ ) {
    V[i] = i * 10;
  }
  vector<int>::iterator p;
  for ( p = V.begin(); p != V.end(); p++ ) {
    cout << *p << '\t';
  }
  cout << endl;
}
```


sequence containers: list

- the `list` container is similar to a `vector` but it also includes a sorting function
- and you cannot use indexing to access elements—you have to use list functions or an iterator
- example:

```
#include <iostream>
#include <list>
using namespace std;

int main() {
  list<int> L;
  for ( int i=0; i<10; i++ ) {
    L.push_front( i * 10 );
  }
  list<int>::iterator p;
  for ( p = L.begin(); p != L.end(); p++ ) {
    cout << *p << '\t';
  }
  cout << endl;
}
```

- note also in this example that you don't specify the size of the list when you instantiate it; instead, the size is updated dynamically as you add elements to the list (using `push_front()`)

sequence containers: deque

- a `deque` is a double-ended queue
- you can add to / remove from both the back and front of it
- example:

```
#include <iostream>
#include <deque>
using namespace std;

int main() {
  deque<int> DQ;
  for ( int i=0; i<10; i++ ) {
    DQ.push_front( i * 10 );
  }
  for ( int i=0; i<10; i++ ) {
    DQ.push_back( i + 10 );
  }
  DQ.pop_front(); // remove first element
  DQ.pop_back(); // remove last element
  deque<int>::iterator p;
  for ( p = DQ.begin(); p != DQ.end(); p++ ) {
    cout << *p << '\t';
  }
  cout << endl;
}
```

associative containers: set and multiset

- a `set` stores a group of unique values according to some ordering relationship
- it's kind of like `enum`, except you don't have to specify the values of each of the elements in the data structure
- a `multiset` is like a `set` with duplicates (i.e., non-unique elements)
- example:

```
#include <iostream>
#include <set>
using namespace std;

int main() {
  set<int> S;
  for ( int i=0; i<10; i++ ) {
    S.insert( i * 10 );
  }
  set<int>::iterator p;
  for ( p = S.begin(); p != S.end(); p++ ) {
    cout << *p << '\t';
  }
  cout << endl;
}
```


associative containers: map and multimap

- a `map` stores elements in “key-value” pairs
- instead of using numeric indexes, like arrays or vectors, to access elements, the “key” is used as a symbolic index
- with a `map`, each *key* and *value* pair is unique
- with a `multimap`, a single *key* may correspond to multiple values
- example:

```
#include <iostream>
#include <map>
using namespace std;

struct strCmp {
  bool operator()( const char* s1, const char* s2 ) const {
    return( strcmp( s1, s2 ) < 0 );
  }
};

int main() {
  map<const char *, int, strCmp> M;
  M["suz"] = 19;
  M["alex"] = 12;
  M["jen"] = 15;
```



```
  map<const char *,int, strCmp>::iterator p;
  for ( p = M.begin(); p != M.end(); p++ ) {
    cout << "(" << p->first << "," << p->second << ")\t";
  }
  cout << endl;
}
```

and the output is:

```
(alex,12)    (jen,15)    (suz,19)
```

- note that elements are listed in alphabetical order based on the key value
- this is because of the `strCmp` comparison operator that is part of the map definition
- if we reversed the operator, e.g., changed
  `return( strcmp( s1, s2 ) < 0 );`
  to
  `return( strcmp( s2, s1 ) < 0 );`
  then the output would be reversed:

```
(suz,19)    (jen,15)    (alex,12)
```


iterators

- an iterator is like a pointer
- but instead of always advancing by either incrementing or decrementing using memory addresses, iterators move around (forward or backward one element or jumping directly to a particular element) according to the rules of each type of iterator (as well as the type of class they are iterating through)
- for example, compare:

```
int i;
for ( i=0; i<N; ++i ) {
  ...
}
```

with

```
vector<int>::iterator p;
for ( p=v.begin(); p != v.end(); ++p ) {
  ...
}
```

- there are different kinds of iterators:
  - `input_iterator`
    reads values with forward movement can be incremented, compared, and dereferenced
  - `output_iterator`
    writes values with forward movement can be incremented and dereferenced
  - `forward_iterator`
    reads or writes values with forward movement combine the functionality of input and output iterators with the ability to store the iterators value
  - `bidirectional_iterator`
    reads and writes values with forward and backward movement like forward iterators, but can also be incremented and decremented
  - `random_iterator`
    reads and writes values with random access
  - `reverse_iterator`
    either a random iterator or a bidirectional iterator that moves in reverse direction

container adaptors

- container adaptors (`stack`, `queue` and `priority_queue`) are containers that are adapted from sequence containers (`vector`, `list` and `deque`)
- they define how elements are added and removed
- `stack`
  - a stack is a "LIFO" data structure: "last in, first out"
  - which means that items are added to the front of the stack and also removed from the front of the stack
  - we have talked about stacks in the past this semester and used the analogy of a stack of plates in a cafeteria: new plates are added to the top; plates are also removed from the top
  - the STL stack has the following members:

```
constructor
empty()
pop()
push()
```



```
size()
top()
```

- `queue`
  - a queue is a "FIFO" data structure: "first in, first out"
  - which means that items are added to the back of the queue and are removed from the front of the queue
  - a queue is just like a conventional line (of humans) (also called a "queue" if you live in the UK)
  - has the following members:

```
constructor
back()
empty()
front()
pop()
push()
size()
```


- `priority-queue`
  - like a queue, except that the items are ordered according to a comparison operator that is specified when a priority queue object is instantiated
  - has the following members:

```
constructor
empty()
pop()
push()
size()
top()
```