cs3157: c++ lecture #3 (mon-18-apr-2005)

- today:
  - composition and derivation
  - dynamic memory allocation (new/delete vs malloc/free)
  - container classes
  - iterator classes
  - templates

composition and derivation

- composition:
  - creating objects with other objects as members
  - example: `array4.cpp`
- derivation:
  - defining classes by expanding other classes
  - like "extends" in java
  - example:

```
class SortIntArray : public IntArray {
    public:
      void sort();
    private:
        int *sortBuf;
    }; // end of class SortIntArray
```

  - "base class" (`IntArray`) and "derived class" (`SortIntArray`)
  - derived class can only access public members of base class

  - `public` vs `private` derivation:
    - `public` derivation means that users of the derived class can access the public portions of the base class
    - `private` derivation means that all of the base class is inaccessible to anything outside the derived class
    - `private` is the default
- friendship
  - allows two or more classes to share private members
  - e.g., container and iterator classes
  - friendship is not transitive
- complete example: `array5.cpp`

derivation, continued.

- encapsulation
  - derivation maintains encapsulation
  - i.e., it is better to expand `IntArray` and add `sort()` than to modify your own version of `IntArray`
- friendship
  - not the same as derivation!!
  - example:
    - $b2$ is a friend of $b1$
    - $d1$ is derived from $b1$
    - $d2$ is derived from $b2$
    - $b2$ has special access to private members of $b1$, as a friend
    - but $d2$ does not inherit this special access
    - nor does $b2$ get special access to $d1$ (derived from friend $b1$)

derivation and pointer conversion

- derived-class instance is treated like a base-class instance
- but you can't go the other way
- example:

```
main() {
  IntArray      ia, *pia;      // base-class object and pointer
  StatsIntArray sia, *psia;    // derived-class object and pointer
  pia = &sia;                  // okay: base pointer -> derived object
  psia = pia;                  // no: derived pointer = base pointer
  psia = (StatsIntArray *)pia; // sort of okay now since:
                               // 1. there's a cast
                               // 2. pia is really pointing to sia,
                               // but if it were pointing to ia, then
                               // this wouldn't work (as below)
  psia = (StatsIntArray *)&ia; // no: because ia isn't a StatsIntArray
}
```

- danger:
  - don't point a base class pointer to an array of derived objects!
  - they aren't the same size!

use of new and delete

- used for dynamic objects

```
IntArray *pia = new IntArray; // object creation
delete pia;                   // object destruction
```

  - this makes sure that ctor and dtor are called, unlike malloc() and free()
  - new/delete *must* be called with classes
  - built-in types (e.g., int) can use either new/delete or malloc/free (which is what we've done in example code so far)
- dynamic arrays

```
IntArray pia[] = new IntArray[NUM]; // calls default constructor on
                                    // each element of pia[]
delete[] pia;  // calls default destructor on each element of pia[]
```


- dynamic initialization of multi-dimensional arrays *not* allowed in C++
- example:

```
IntArray ***ia3 = new IntArray[2][3][4]; // NO!!!
IntArray ***ia3 = new IntArray**[2]; // OKAY
         // ^^^ and then use two nested loops to create inner arrays...
IntArray (*ia3)[3][4] = new IntArray[2][3][4]; // OKAY
         // ^^^ dynamic array of static arrays...
```

- no realloc() in C++ on objects
  - do it manually:
    1. call new[] on a temporary object of the new size
    2. copy values from current object to temporary object
    3. destroy current object
    4. rename temporary objec to current object
  - when out of memory, new returns null or program terminates

example, dynamic memory allocation.

```
class mystack {
  public:
    explicit mystack( int size ) : max_len(size), top(EMPTY)
      { assert( size > 0 ); s=new char[size]; assert( s != 0 ); }
    void reset() { top = EMPTY; }
    void push( char c ) { s[++top] = c; }
    char pop() { return s[top--]; }
    char top_of() const { return s[top]; }
    bool empty() const { return( top == EMPTY ); }
    bool full() const { return( top == FULL ); }
  private:
    enum { EMPTY = -1 };
    char *s;
    int  max_len;
    int  top;
}
```

container classess.

- two types:
  - sequence containers
    - vectors: stores a sequence of elements
    - lists: convenient and efficient way to do internal insertion/deletion
    - deques: doubly-ended queue (add at both front and back)
  - associative containers
    - sets: stores a value according to an ordered relationship; contains only unique values
    - multisets: like a set but allows multiple copies of the same item to be stored
    - maps: basic associative array and requires that a comparison operation on the stored elements be defined
    - multimaps: generalization of a map to allow non-unique keys
- the two types share similar interfaces

iterator classes.

- provide navigation over containers
- sort of an enhanced pointer type
- five types: input, output, forward, bidirectional, random-access
- example:

```
int main() {
  int primes[4] = { 2, 3, 5, 7 };
  set<int, greater<int> > s;
  set<int, greater<int> > :: const_iterator c_it;
  while ( ptr != primes + 4 )
    s.insert( *ptr++ );

  cout << "the primes below 10 are: " << endl;
  for ( c_it = s.begin(); c_it != s.end(); ++c_it )
    c_out << *c_it << '\t';
  c_out << endl;
}
```


templates.

- `template` used to allow the same code to be used with respect to various data types (called "parametric polymorphism")
- it's a form of *generic programming* which is meant to support code re-use
- example is used like this:

```
mystack<char>  a;      // declares a stack of 100 chars
mystack<int>   b(200); // declares a stack of 200 ints
```

- there is a *Standard Template Library*, called STL, which contains templates for many things including `string`, `vector` and `complex`
- notes:
  - `char top_of() const { return s[top]; }` means that the `this` parameter will be constant during the method call
  - `assert()` is a function that says if the argument expression is not true, then the program execution should abort

mystack example, using a template.

```
template <class TYPE>
class mystack {
  public:
    explicit mystack( int size=100 ) : max_len(size), top(EMPTY)
      { assert( size > 0 ); s=new TYPE[size]; assert( s != 0 ); }
    ~mystack() { 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 == FULL ); }
  private:
    enum { EMPTY = -1 };
    TYPE *s;
    int  max_len, top;
}
```