Iteration 2 Submission : Due THURSDAY at 11:55pm.
PULLING at midnight!

ITEM	SUBMISSION
Priority 1, 2, 3 Implemented Iteration 2 folder!!!	Merge into master branch Tag with v.2.0
Bug Report	*"BugReport.*"* in /docs OR as part of Doxygen along with todo list
Design Document Major design decisions with justification including other options that you considered.	"DesignDocument.*" in /docs
UML to match your code	UML.pdf in /docs
Mainpage.h (do you need to change this?) COMPILE YOUR DOXYGEN. Don't leave stuff around.	Do not submit the html folder. We will compile.
Google Style Compliant : Check Naming Conventions!	Run cpplint – no errors!
Github Usage : branches, issues, regular commits, good messages!!	We look at history and a usage report
Google Tests : tests for sensors.	Visual inspection – not passing.

Bug Report

Identify all known bugs.
Describe the bug.
If you don't have any known bugs, report that.

Bugs are not todo lists – they are for implementation that is not working as required or desired!

ALSO,
if you do have incomplete functionality include it here making it clear what is not implemented and what is buggy.

OR,
If you are doing this in doxygen, use @bug and @todo.

You will be docked points
if we find a bug that you did not report.

Templates and Containers

CS3081 Program Design and Development

Polymorphism

Polymorphism: generally defined as "the ability to create a variable, a function, or an object that has more than one form." The result is that you get different behavior (i.e. different pieces of code are executed) depending on the type of object or objects that are being acted upon.

- **Operator Overloading**: One operator can be applied to different types.
- **Method Overriding (Ad-hoc polymorphism)**: Derived class redefining base class method.
- **Method Overloading (Ad-hoc polymorphism)**: Multiple function definitions with different parameter lists.
- **Subtype Polymorphism**: Upcasting – derived class object can be used in place of base class object.
- **Parametric Polymorphism**: Templates – one function with same behavior across multiple types. (Stack of ints, strings, ClassA, ...)

cite: Wikipedia and http://www.catonmat.net/blog/cpp-polymorphism/

Templates

"Inheritance and composition provide a way to reuse object code. The *template* feature in C++ provides a way to reuse *source* code."

A way for you to write generic, type-less code.

Example:

class IntegerArray and class FloatArray and class CharArray

.....

instead

template<class Type> class Array

Strong Type Checking*

- A type can be defined as
  - a set of permitted values and
  - a set of operations permitted on these values

- Important operations in programming languages related to types:
  - defining a new type
  - declaring variables to be of a certain type
  - checking that no type errors can occur – type checking

* from Gopalan Nadathur via Eric VanWyk

Type Checking

- Types and type checking help us avoid type errors.

- Type error occurs when sequence of bits that represent one kind of data is interpreted as another kind of data. (e.g. …
  - reading sequence of bits that store an int value as a char *
  - reading the representation of an object as a string

- The nature of research in type systems:
  - preserve strong typing while
  - making the type system more liberal.

  That is, continue to disallow any bad programs but allow more good programs.

  A major advance: parametric polymorphism.

* from Gopalan Nadathur via Eric VanWyk

Implementation techniques

To implement a function that has parameterized types a compiler can:

1. Create a single function that works on all types (in machine language)

2. Determine what types are passed as parameters to the functions and generate a special case function for each of these types. This is what C++ does for template types.*

Vector Template

```
std::vector<class ArenaEntity*> entities_;
entities_.push_back(new ArenaEntity(20));
entities_.push_back(new ArenaEntity(25));

for (std::vector<class ArenaEntity*>::iterator ent = entities_.begin();
ent != entities_.end();
++ent ) {
  (*ent)->Print();
}
```

```
// This is the same as above, except with the convenience of auto typing
std::cout << std::endl;
for (auto ent : entities_) {
  ent->Print();
}
```

The Need for Polymorphism

```
void swapI( int& x, int& y ) {
  int temp;
  temp = x;
  x = y;
  y = temp;
}
```

```
void swapF( float& x, float& y) {
  float temp;
  temp = x;
  x = y;
  y = temp;
}
```

```
int main() {

  int x(50), y(10);
  float v(3.2), w(6.4);
```

Wouldn't it be nice to ...

```
int x,y;
float v,w;
swap(x,y);
swap(v,w);
```

```
x:50 y:10 | v:3.20 w:6.40
swap
x:10 y:50 | v:6.40 w:3.20
```

```
printf("\n x:%d y:%d | v:%.2f w:%.2f \n",x,y,v,w);
swapI(x,y);
swapF(v,w);
cout << " swap" << endl;
printf(" x:%d y:%d | v:%.2f w:%.2f \n\n",x,y,v,w);
```

Overloaded Functions (Better)

```
void olSwap( int& x, int& y ) {
  int temp;
  temp = x;
  x = y;
  y = temp;
}
```

```
void olSwap( float& x, float& y ) {
  float temp;
  temp = x;
  x = y;
  y = temp;
}
```

Wouldn't it be nice to write 1 piece of code for all types!

```
int main() {

  int x(50), y(10);
  float v(3.2), w(6.4);
```

```
x:50 y:10 | v:3.20 w:6.40
overloaded swap
x:10 y:50 | v:6.40 w:3.20
```

```
print
swapI
swapF
cout
print
```

```
printf("\n x:%d y:%d | v:%.2f w:%.2f \n",x,y,v,w);
olSwap( x, y );
olSwap( v, w );
cout << " overloaded swap" << endl;
printf(" x:%d y:%d | v:%.2f w:%.2f \n",x,y,v,w);
```

Generic Swap with Templates

```
void swapI( int& x, int& y ) {
  int temp;
  temp = x;
  x = y;
  y = temp;
}
```

```
void swapF( float& x, float& y) {
```

... and all other types we might swap.

```
template<typename T>
void mySwap( T& x, T& y ) {
  T temp;
  temp = x;
  x = y;
  y = temp;
}
```

Wouldn't it be nice to ...

```
int i;
float f;
swap( i, f );
```

```
int main() {

  int x(50), y(10);
  float v(3.2), w(6.4);
```

```
mySwap( int z(3), float m(2.8) );
```

```
print
swapI
swapF
cout
print
```

```
char c1('a'), c2('b');
printf(" x:%d y:%d | v:%.2
mySwap( x, y );
mySwap( v, w );
mySwap( c1, c2 );
cout << " swap" << endl;
printf(" x:%d y:%d | v:%.2
```

```
x:50 y:10 | v:3.20 w:6.40
swap
x:10 y:50 | v:6.40 w:3.20

x:10 y:50 | v:6.40 w:3.20 | c1:a c2:b
swap
x:50 y:10 | v:3.20 w:6.40 | c1:b c2:a
```

Templates and Classes

```
class Position {
private:
   int x;
   int y;
public:

   Position( ) : x(0), y(0) {}
   Position( int inX, int inY ) : x(inX), y(inY) {}
```

```
class PositionFloat {
private:
   float x;
   float y;
public:
   PositionFloat(float inX=0.0, float inY=0.0 ) : x(inX), y(inY) {}
```

Classes of Different Types

```
template<class T>
class Position {
private:
  T x;
  T y;
public:
  Position( T xx=0, T yy=0 ) : x(xx), y(yy) {}

  friend ostream& operator<<(ostream& os, const Position& pf) {
    return os << "[" << pf.x << "," << pf.y << "]";
  }
};
```

Wouldn't it be nice to ...

```
Position<float>  fP( 1.5, 1.2);
Position<int>  iP( 10, 10 );
cout << ( iP + fP ) << endl;
cout << ( fP + iP ) << endl;
```

```
int main() {

  Position<int> iP(10,10);
  Position<float> fP(1.5,1.2);

  cout << iP << " " << fP << " " << endl;
```

```
[10,10] [1.5,1.2]
```

Mixed Types in Swap

```
void swapI( int& x, int& y ) {
  int temp;
  temp = x;
  x = y;
  y = temp;
}
```

```
void swapF( float& x, float& y) {
```

... and all other types we might swap.

```
template<typename T>
void mySwap( T& x, T& y ) {
  T temp;
  temp = x;
  x = y;
  y = temp;
}
```

```
mySwap( int z(3), float m(2.8) );
```

```
template<typename T1, typename T2>
void mixedSwap( T1& x, T2& y ) {
  T1 temp = x;
  x = (T1) y;
  y = (T2) temp;
}
```

Cast a class ?
x + y ??

Wouldn't it be nice to ...

```
Position<float> fP( 1.5, 1.2);
Position<int> iP( 10, 10 );
cout << ( iP + fP ) << endl;
cout << ( fP + iP ) << endl;
```

```
printf("\n x:%d y:%d | v:%.2f w:%.2f \n",x,y,v,w);
mixedSwap( x, v );
mixedSwap( x, y );
cout << " mixed swap" << endl;
printf(" x:%d y:%d | v:%.2f w:%.2f \n",x,y,v,w);
```

```
x:10 y:50 | v:6.40 w:3.20
mixed swap
x:50 y:6 | v:10.00 w:3.20
```

Array

Define a class called Array that holds N objects of type T

```
template<class T, int N>
```

Overload the operator [] for indexing the array.

Be sure to check bounds when indexing

C++ Containers

Sequence containers

Sequence containers implement data structures which can be accessed sequentially.

array (since C++11)	static contiguous array (class template)
vector	dynamic contiguous array (class template)
deque	double-ended queue (class template)
forward_list (since C++11)	singly-linked list (class template)
list	doubly-linked list (class template)

Container adaptors

Container adaptors provide a different interface for sequential containers.

stack	adapts a container to provide stack (LIFO data structure) (class template)
queue	adapts a container to provide queue (FIFO data structure) (class template)
priority_queue	adapts a container to provide priority queue (class template)

- http://en.cppreference.com/w/cpp/container (there are other containers)

Built-In Functionality of Containers

Member functions

(constructor)	constructs the vector (public member function)
(destructor)	destructs the vector (public member function)
operator=	assigns values to the c (public member function)
assign	assigns values to the c (public member function)

Element access

at	access specified elem (public member function)
operator[]	access specified elem (public member function)
front	access the first eleme (public member function)
back	access the last eleme (public member function)

Modifiers

clear	clears the contents (public member function)
insert	inserts elements (public member function)
emplace (C++11)	constructs element in-place (public member function)
erase	erases elements (public member function)
push_back	adds elements to the end (public member function)
emplace_back (C++11)	constructs elements in-place at the (public member function)
pop_back	removes the last element (public member function)
resize	changes the number of elements s (public member function)
swap	swaps the contents (public member function)

Iterators

begin cbegin	returns an iterator to the beginning (public member function)
end cend	returns an iterator to the end (public member function)

Modifying a Container

- Delete Elements
  - vector.pop_back(); // remove last element
  - vector.erase(iterator); // remove element at iterator location
  - vector.clear(); // remove all elements
- Access Elements
  - vector.front(); // first element
  - vector.back(); // last element
  - vector.at(position); // check bounds
  - vector[#];
- Iterator (special “smart” pointers for accessing containers)
  - vector.begin(); // returns iterator to first element
  - vector.end(); // returns iterator to last element

Iterators

```
cout << endl << "MyContainer expect { 20, 21, 22, 23 }:" << endl;
for ( int i = 0; i < myContainer.size(); i++ ) {
  cout << myContainer[i] << " ";
}
```

It helps to have "someone else" keep track of the size.
This is an okay method for looping to initialize, but access with iterators.

```
vector<int>::iterator p;
p = myContainer.begin();
```

How you declare an iterator.

```
myContainer.insert(p,102);
```

Iterators resolve the array access issues (seg faults).

```
for ( p = myContainer.begin(); p != myContainer.end(); p++ ) {
  cout << *p << " ";
}
```

Notice the dereference operator – iterator is a pointer!

Templates and Containers

- Templates
  - Compiler creates a class/function for each needed type.
  - Defining (put it all in header)
  - Methods in template must be defined for class.

- Containers
  - For safety and ease.
  - Use iterators (declared as a specific container type iterator).