Function Templates

①
```
template <typename T>
T maximum(T value1, T value2, T value3)
{
  T max = value1;
  if (value2 > max)
    { max = value2; }
  if (value3 > max)
    { max = value3; }
  return (max);
}
```

②
```
template <typename T1, typename T2)
T1 multipleTs(T2 num)
  { return (T2 * T2); }
```

↳ to call in main...
```
functionName<int>(int variable)
OR functionName(int variable)
```

Pointers

↳ declaration
```
int * intPtr;
```
type pointed to; pointer name

↳ initialization
```
int * aPtr;        can also have
aPtr = &a;         aPtr = NULL;
```

↳ dereferencing
```
int a = Ptr; → return address
int b = *Ptr; → return value
```

↳ arithmetic and arrays
```
int a[5] = {1,2,3,4,5}; → 1 2 3 4 5
int *aptr = &a[0];      → at a[0]
a[2] = a[2]+1;          → 1 2 4 4 5
a[3] = a[3]+a[4];       → 1 2 4 9 5
aptr++;                 → at a[1]
(*aptr)++;              → 1 3 4 9 5
(*(++aptr))--;          → 1 3 3 9 5
```

↳ functions: passing a pointer to a function is like passing by reference

↳ strings
```
const char *str = "hello";
```
str → | 'h' | 'e' | 'l' | 'l' | 'o' | '\0' |

↳ member access operator (->)
a->b is equivalent to (*a).b

References vs Pointers

↳ can't change references once they have been declared

↳ references and variables name a location, pointers point to location

↳ different function calls can change the same variable

const keyword

↳ declare variable: const int a=5;

↳ within function: void printArray(const int A[])
↑ compiler won't let you change input array

↳ mark getters as const
int getHour() const;

Text File Manipulation

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
  { ofstream myfile;
    myfile.open("name.txt", ios::out);
    myfile << "write this" << endl;
    myfile.close();
  }
```
or...
```
ifstream myfile("name.txt", ios::in);
myfile >> x0 >> y0 >> x1 >> y1;
myfile.close();
```

String objects vs pointer-base strings

↳ string mystr = "hi"; → mystr is an object #include<string>

↳ char *str = "Hello"; → str points to char array

↳ member functions and operators

+	appends rhs to lhs (concatenates)
==	compares length & letters (case matters!)
a > b	true if a goes after b alphabetically
a < b	true if a comes before b alphabetically
size()	returns length (as an unsigned int)
find()	finds first occurrence of pattern → returns position find(str findme, int startsearchhere) ↳ or char or char* ↳ default is 0
rfind()	finds last occurrence of pattern
find_first_of()	searches for first of any char provided ex) find_first_of("aeiou"); → returns position
find_last_of()	searches for last of any char provided
find_first_not_of()	searches for first char not provided
c_str()	get a c-string equivalent - returns a pointer to a char array (last char is null)
substr()	generate new string that is a copy of a substring
erase()	deletes characters in a string → returns string& erase(int initalpos, int lengthtoerase);
insert()	adds chars to the middle of a string right before char indicated with pos insert(int pos, string& addme);

↳ iterators

begin	returns iterator to beginning
end	returns iterator to end

↳ element access

back	access last character (returns reference)
front	access first character (returns reference)

↳ Modifiers

push_back()	append char to a string str.push_back(char);
pop_back()	erases last char of string → void return str.pop_back();

Arguments to main()

```
int main(int argc, char* argv[])
```
argc: argument count

argv[]: or char** argv, argument vector, argv[0] → a.out

Pass an entire 1D array to a function

↳ prototype: void func1(int[], int);

↳ call: func1(array, size);

Classes

*Constructors
↳ initalize objects

```
class Point
{
public:
  Point(); //default
  Point(int a, int b) //parametrized
  ~Point() //destructor
private:
  int x, int y;
};
Point::Point(): x(0), y(0) {}
Point::Point(int a, int b)
{
  x=a;
  y=b;
}
Point::~Point() {} //deletes object
```

*Public vs Private
↳ data members are private
↳ member functions/constructors are public
↳ friend designation lets a class or function access private or protected data-breaks encapsulation
put "friend [function prototype or class]" before public or private designations, implement outside of class

*This pointer
↳ The "this" keyword inside a member function stores a value of the pointer to the object

* Operator overloading
↳ global: specify lhs and rhs, must use "get" or friend

```
complex operator*(complex &lhs, complex &rhs)
{
  double r = (lhs.getReal()*rhs.getReal()) -
      (lhs.getImag()*rhs.getImag());
  double i = (lhs.getImag()*rhs.getReal()) +
      (lhs.getReal()*rhs.getImag());
  return (complex(r, i));
}
```

↳ member function

```
complex complex::operator+(complex &rhs)
{ return complex(real+rhs.getReal(),
      imag+rhs.getImag()); }
```

↳ pre-fix increment

```
Time& Time::operator++()
{
  hour++;
  return (*this);
}
```

return by reference

↳ postfix increment (tells compiler this is post-fix)

```
Time Time::operator++(int)
{
  Time temp(*this);
  hour++; // hour from orig object
  return temp;
}
```

must return by value or its a dangling reference

cout << object++ << endl; → returns old copy
cout << ++object << endl; → return updated object

↳ output stream << (global!)

```
ostream& operator<<(ostream& output, const Time& TimeObject)
{                          //LHS              //RHS
  output << timeObject.getHour() << ":" << time object.getMin()
    << ":" << time object.getSec();
  return output;
}
```

*Class Templates (or typename)

```
↳ template <class T>          //general template
  class myPair
  {
  public:
    myPair(T first, T second)
      { values[0]= first;
    private: values[1]= second;}
      T values[2];
  };
```

↳ member function template (use if member function is implemented outside of class definition)

```
template <typename T>
type className <T>::memFunc(T variable)
{
  // implementation
}
```

↳ to call in main

```
className <int> myObject;
```

*Standard Template Library
↳ vector #include vector
→ like a DArray, contiguous memory, extra space is automatically added to the end when created

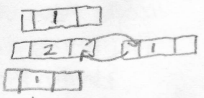
```
Vector <double> values; → empty
Vector <int> values2(10, 2) → size, initialization
```

→ Member functions

```
values.push_back(3.14)   //add 3.14 to back
cout << values[0]        //print 1st value
values.size()            //current # of values
values.resize()          //change size.. could cause
values.front()           //return 1st   performance issues
values.back()            //return last
values.capacity()        //amount allocated
```

↳ Lists #include <list>
→ a double-linked list, can move backward or forward, faster than big memory reallocation
→ member functions/implementation

```
list <int> values
values.push_front(1);
values.push_front(2);
values.pop_front();
values.merge(list2);     //add list2 to end
values.reverse();        //flip order
```



↳ containers: the data structure itself (vectors, lists...)
↳ iterators: special pointers for containers, refer to elements
*dereferencing; +, −, ++ pointer arithmetic

```
Vector <int>::iterator, itexample;
```

container type | iterator type | name of iterator

↳ algorithms: perform operations on containers using iterators

```
sort(vectorValues.begin(), vectorValues.end());
```

Dynamic Memory

*New and delete operators (must be an int!)
double *beginPtr;
beginPtr = new double[size]; → avoid memory leaks!
delete [] beginPtr;
(new returns a pointer)

*Rule of Three
① destructors: same name as class with a ~ before, never any arguments, never explicitly called, c++ has default destructors for you
```
DArray::~DArray()
{ delete [] beginPtr;}
```
↳ order matters! Delete most recently created first
↳ destructor automatically called when leaving scope

② overloading assignment operator: want two copies of data, not 2 pointers to the same place
↳ function prototype: DArray& operator=(const DArray&);
↳ return by reference
avoid dangling references... you can only return by reference if reference points to an object/variable that existed before call.
↳ only overload = if class has dynamic data pointer
```
DArray& DArray::operator=(const DArray &RHS)
{ //check self-assignment
  if(this != &RHS)
  { //check if size is the same
    if(size != RHS.size)
    { delete [] beginPtr;
      size = RHS.size;
      beginPtr = new double [size];
    }
    //fill in values
    for(int i=0; i<size; i++)
      { beginPtr[i] = RHS.beginPtr[i];}
  }
}
```

③ overload copy constructor: called when passing an object to a function by value or returning an object by value, or called directly. c++ has a default
```
//class interface   (must be by reference, or stuck in ∞ recursion!)
DArray(const DArray&);
// implementation
DArray::DArray(const DArray& copyMe)
{ size = copyMe.size;
  //allocate memory
  beginPtr = new double[size];
  //copy new values
  for(int i=0;i<size;i++)
    {beginPtr[i] = copyMe.beginPtr[i];}
}
```

Inheritance

*Base classes and derived classes
↳ derived class contains behaviors inherited from base class, and can have add'l behaviors
```
Class Militime: public Time
{ public:
    MiliTime(int=0, int=0, int=0, int=0);
    void setMilisec(int);
    int getMilisec();
    void printTime();   //new print!
  private:
    int milisec;
```
↳ derived classes CANNOT access base class private data directly: making "private" data "protected" will fix this, or use getters
↳ derived class constructors
```
MiliTime::MiliTime(int hr, int min, int sec, int milisec):
  Time(hr, min, sec), milisec(0)
{ setMilisec(milli);
}
```
↳ Direct base class: 1st degree relation, indirect base class 2+
↳ member functions
```
class Base
{ public:
    void F1();
    void F2();
    void F3();
    void F4();
};
class A: public Base
{ public:
    void F2();
    void F4();
};
class B: publicA
{ public:
    void F1()
    void F4()
};
class C: public Base
{ public:
    void F1();
    void F4();
};
```
call F1 from each object
BaseObject.F1(); → Hi Base F1
AObject.F1(); → Hi Base F1
BObject.F1(); → Hi B F1
CObject.F1(); → Hi C F1
call F2 from each object
Base → Hi Base F2
A → Hi A F2
B → Hi A F2
C → Hi Base F2

Polymorphism

* Handles to access objects: name, reference, pointer
↳ you can point a base class pointer to a derived object
basePtr = &derivedObject;
↳ you can't point a base pointer to derived member function
basePtr → derivedMemberFunction(); → will call base function unless base fn is virtual

*virtual functions: allow access to derived member function through base ptr
virtual void Function1(); (in base)

*static vs dynamic binding
↳ static: baseObject.memberFunction(); //resolved at compile time
↳ dynamic: basePtr→virtualFunction(); //resolved at run time

*Abstract vs Concrete classes
↳ abstract classes: never intend to implement
↳ concrete: plan to implement; must have or inherit an implementation of a pure virtual function
↳ pure virtual function: no implementation, belongs to an abstract class
virtual returnType funcName()=0; (=0 tells compiler pure virtual)
↳ make code more general, easier to enhance

ECE 2036 – Test 2

C++ Basics

* control structures
  - for loops
    ```
    for(int i=1; i<=10; i++)
      { ;}        no;
    ```
  - do...while loops
    ```
    do
    {
    }while (condition);
    ```
  - while loops
    ```
    while (condition)
      {
      }
    ```
  - switch statements
    ```
    switch (variable)
    {
      case value1:
        :
        break;
      case value2:
        :
        break;
      default:
    }
    ```
  - conditionals
    ```
    if (condition1)
      {;}
    else if (condition2)
      { ;}
    else
      { }
    ```

* preprocessor directives
  - #include <string>
  - #include "matrix.h"
  - inclusion guards in header file...
    ```
    #ifndef MATRIX_H
    #define MATRIX_H
      //normal header code
    #endif
    ```

* I/O stream manipulators #include<iomanip>
  - setw (not sticky)
    `cout<< a<< setw(10)<< b<<endl;`
  - setfill (sticky)
    `cout<< a<<setfill('*')<<setw(10);`
  - hex/oct/dec (sticky)
    `cout << hex << num << endl;`
  - fixed/scientific (sticky)
    `cout << fixed << num<<endl;`
  - setprecision (sticky)
    `cout<<setprecision(sigfigs);`

* Arrays
  - initializing (starts at index 0!)
    `int nameArray[size]= {a, b, c...};`
  - pass to a function: copy is not made, array is passed by pointer!
    `void initArray(int [], int)` → prototype

* variable scope rules
  - local variables exist only in their block of code/function
  - global variables are defined outside of any function
  - inside a function, global < local if names are the same
    use ::name to access global variable

* storage class
  - auto: default for all local variables — int a or auto int a
  - static: variable is stored in the compiler for the life of the program – the value is maintained between calls — static int b

* standard c libraries: cmath #include<cmath>
  - rounding: ceil, floor, fabs
  - math: exp, log, cos, sin, sqrt, tan, rand
  - random number
    ```
    value = rand%101   //returns random btwn 0-100
    value2= rand()%25+1 // returns random btwn 1-25
    ```

* Functions
  - function prototypes include name, input/output type
  - argument coercion does not change value of variable in main, promotes mixed types to match highest type (ex: int→float)
  - function overloading: functions can have the same name but different input types (ex: print())
  - default arguments: allow a function to be called without providing one (or more) trailing arguments
    `void point(int x=0, int y=0)` → if y (or x) is not specified, they are given value zero

* Pass by value vs pass by reference
  - pass by value: default pass. copy is used. slower
  - pass by reference (&): manipulates directly. riskier
    argument coercion will not work!

* Recursion
  - Fibonacci
    ```
    int fibonacci(int n)
    {
      if (n<=0)
        { return 0;}
      else if (n==1)
        { return (1);}
      else
        { return (fibonacci(n-1)+fibonacci(n-2));}
    }
    ```
  - Factorial
    ```
    int factorial(int n)
    {
      if (n<=1)
        { return 1;}
      else
        { return (n*factorial(n-1));}
    }
    ```
  - everything that requires recursion can be implemented iteratively – makes program much faster/more efficient!