CSCI 3132 – Object Orientation and Generic Programming

Week 5
– Structs and Enums
– Function and Operator Overloading

Structures in C++

Data Structures in C++

- Arrays are used to store several items of the same data type
- Is it possible to store several items of different data types?
  - Structures can store several values of several different types
  - Useful when a lot of data needs to be grouped together
  - Example:
    - Store records for a database table
    - Store information for a single contact (e.g. name, addr, phone, email)
- Syntax:

struct *struct_name* {
    member variables;
};

Struct Example

```
struct AddrBook {
      string name;
      string addr;
      int phone;
      string email;
};
int main(){
      AddrBook contact1;
      contact1.name = "Khurram";
      contact1.addr = "6050 University Ave";
      contact1.phone = 9021234567;
      contact1.email = "mymail@mymail.com";
      cout<<contact1;
      return 0;
}
//identify the problem in this code!
```

Using Pointers with Struct

```
struct AddrBook_t {
      string name;
      string addr;
      long long phone;
};
int main(){
      AddrBook_t * contact1 = new AddrBook_t;
      contact1->name = "Khurram";
      (*contact1).addr = "6050 University Ave";
      contact1->phone = 9021234567;

      cout << contact1->name<<"\t"<<contact1-
      >addr<<"\t"<< (*contact1).phone<<endl;
      return 0;
}
```

Arrays with Struct Type

- Is it possible to have arrays of structs?
  - Yes
- Where could they be used?
  - Arrays of structs could be used to create tables or databases
  - Consider the AddrBook structure shown in previous slide
    - An array of AddrBook_t structure can be used to hold several addresses
    - Similar to a database table holding several addresses

Arrays with Struct Type

```
struct AddrBook_t {
     string name;
     string addr;
     long long phone;
} contact[3];

void PrintContact(AddrBook_t *cont) {
     cout << cont->name << " " << cont-
          >addr << " " << cont->phone << endl;
}
```

Arrays with Struct Type – cont'd.

```
int main(){
    for (int i = 0; i < 3; i++) {
        cout << "\nEnter name: ";
        getline(cin, contact[i].name);
        cout << "\nEnter address: ";
        getline(cin, contact[i].addr);
        cout << "\nEnter phone: ";
        cin>>contact[i].phone;
    }
    for (int i = 0; i < 3; i++) {
        AddrBook_t * addrb = &contact[i];
        PrintContact(addrb);
    }
    return 0;
}
```

Nesting Structures

- An element of a structure can be a structure itself
  - Example:

```
struct Contacts_t {
    int phone;
     string email;
};
struct AddrBook_t {
     string name;
     string addr;
     Contact_t cont;
}addr1, addr2;
```

  - Valid expressions:

```
addr1.name
addr2.cont.phone;
addr1.cont.email;
```

Struct versus Class

- Struct in C++ is essentially a class with public data members by default
- Should struct ever be used in object-oriented programming?
  - Consider the get() and set() methods and what they do
  - Consider code visibility versus the advantages of encapsulation
  - Consider future extension of your program a.k.a. code reusability

Enumeration

Variables with a Defined Set of Values

- Consider the difference between the following two variable definitions

```
int my_var = 10;
const int my_var = 10;
```

- What if a variable needs to be used for a specific purpose
  - Only allowed to have a small subset of values
  - E.g. months of the year, or directions of the compass

Variables with a Defined Set of Values

- A possible way to do this is by using #define
  - E.g. In order to define a valid direction

```
#define NORTH 0
#define SOUTH 1
#define EAST 2
#define WEST 3

int direction = SOUTH;
```

- Problem → Doesn't prevent assignment of non-sensical values, e.g. int direction = 125;
  - Compiler won't complain, but program logic would be wrong

Enumeration Types

- Enumerated types allow creation of new data types that can take on a restricted set of values
  - Example:
    ```
    enum directions_t {NORTH, SOUTH, EAST, WEST};
    directions_t direc = SOUTH;       //valid
    directions_t direc = SOMETHINGELSE; //invalid
    ```

Enumeration Types

Q. What values do the constants take on?

A. By default {0, 1, 2, 3}

- Defaults values can be changed
  - Example:
    enum directions_t {NORTH=4, SOUTH=3, EAST=2, WEST=1};

Q. Why would you need to give explicit values to enum?

A. No reason if values of the constant never used

- Might be required in some cases, for example:
  - Storing text colors and their numerical values
  - One value having multiple names (1 for JAN or JANUARY)

Enumeration Types

```
enum directions_t {
NORTH,
SOUTH,
EAST,
WEST
};

int main(){
      directions_t direc = NORTH;
      int i_direc = 33;
      i_direc = EAST;
      cout<<direc<<endl;        //Output = 0
      cout<<i_direc<<endl;      //Output = 2
      return 0;
}
```

Enumeration Examples

Q. `enum Suit { Diamonds=1, Hearts, Clubs, Spades };`
What is the value of Clubs?

A. 3

Q. `enum Suit { Diamonds=5, Hearts, Clubs=4, Spades };`
What is the value of Spades?

A. 5

Q.
```
enum Suit { Diamonds, Hearts, Clubs, Spades };
    Suit cur_suit = Diamonds;
    cur_suit++;        //what is the value of cur_suit?
```

A. Compiler error. Need to type cast cur_suit;
```
cur_suit = Suit(cur_suit+1);
```

Memory Allocation – Stack versus Heap

Program Memory

- Code Segment
  - Also known as a text segment
  - Typically a read-only segment containing the program's executable instructions
- BSS Segment
  - Also known as uninitialized data
  - Contains all uninitialized (or initialized to 0) global and static variables
  - Example: static int i;
- Data Segment
  - Usually adjacent to BSS
  - Contains initialized static variables (global and static local variables)
  - Size determined by size of data types defined in the program's source code and does not change at run time
  - Example: int i = 3;

Program Memory

- Heap Segment
  - Also called the free store
  - Stores dynamically allocated variables
- Call Stack
  - Simply called stack
  - Stack is a LIFO structure
  - Stack pointer register tracks the top of stack, adjusted each time a value is pushed onto or popped from the stack
  - Stores function parameters, local variables and other function-related information

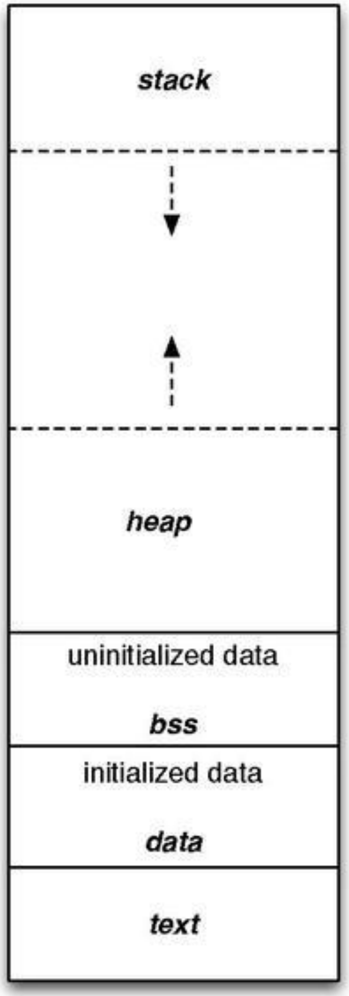


Image Ref: Dougct, Wikimedia Commons,

Memory Allocation in C++

- Mainly three types of memory allocation in C++
  - Static memory allocation
    - For static and global variables
    - Memory allocated once when the program is run and persists throughout the life of the program
  - Automatic memory allocation
    - For local variables and function parameters
    - Memory allocated when the relevant code block (e.g. function) is entered and freed when the block is exited
  - Dynamic memory allocation
    - For requesting memory when needed
    - Memory requested when needed and must be freed when not needed any more

Memory Allocation in C++

- Static and automatic allocation have two things in common
  - The size of the variable/array must be known at compile time
  - Memory allocation and deallocation is automatic
- Static variables are not allocated on the stack
  - Allocated in BSS (if uninitialized) or Data Segment (if initialized)
- Automatic variables are allocated on the stack
  - Variables only persist during a single call to a function

The Call Stack

- Call stack keeps track of all active functions
  - Handles allocation of all function parameters and local variables
  - Implemented as a container data structure holding multiple variables
    - Allows peek() or top(), pop() and push()
    - Last-in first-out (LIFO) structure
    - Data associated to one function call kept in a stack frame
    - A stack pointer keeps track of the top of the call stack

The Call Stack

- Working of the call stack
  1. Program encounters a function call
     a) A stack frame is constructed and pushed on the stack. It contains
        – The return address
        – Function arguments
        – Local variables
        – Register values that might need to be restored after the function call
     b) CPU execution jumps to function's start point and function begins execution
  2. Function terminates
     a) Registers are restored from the stack
     b) Stack frame is removed from the stack, freeing memory located to local variables and arguments
     c) CPU resumes execution at the return address

The Call Stack

- Advantages
  - Stack operations are fast
  - Memory known at compile time and can be accessed directly through a variable
  - Memory allocation and deallocation handled automatically
  - Memory is contiguous and not fragmented
- Disadvantages
  - Size is relatively small
    - Typically 1 MB on Windows, up to 8 MB on Linux
  - Some of the allocated memory may be unused and thus wasted (e.g. char description[50];)
  - Variables cannot be resized
  - LIFO access only

The Heap

- Heap is a larger pool of memory managed by OS
- Heap memory is dynamically allocated to variables as they require it
  - Memory can be returned to the OS once the program is done using it
- The program itself is responsible for allocation and deallocation
  - New allocations are made by creating a block of suitable size from the available memory blocks
  - Request to make a large block may fail if none of the free blocks are of a large enough size (heap fragmentation)

Example 1

- Identify the kind of variable and where it is allocated
- Would this piece of code work?

```
void func()
{
    if(true) {
        int temp = 0;
    }
    temp = 1;
}
```

Example 2

```
class TestClass;
TestClass * ptr_test( )
{
    int i;                                    //automatic, local, stack
    TestClass objTC;                          //automatic, local, stack
    TestClass *ptrA = &objTC;                 //automatic, local, stack
    TestClass *ptrB = new TestClass();        //ptr local, on stack,
                                              //object dynamic, on heap
    return ptrB; //is this safe?              //safe, ptr's object is on heap
    return ptrA; //is this safe?              //unsafe, ptr's object is local,
                                              //on stack
}
```

Example 3

Identify the kinds of variable, where they are allocated and what is their scope

```
int g_var1 = 5;                //static, global, DS
static int s_var2;             //static, global, BSS
void func(int arg1){           //automatic, local, stack
    static int l_var3=1;       //static, local, DS
    int l_var4;                //automatic, local, stack
    int * p_var5;              //automatic, local, stack
    p_var5 = new int;          //dynamic, local,heap
    delete p_var5;             //deallocates space
}
```

Function Overloading

Overloading

- Specification of more than one function of the same name in the same scope.
- Two different functions can have the same name if their parameters are different
  - They have a different number of parameters, or
  - They have a different type of parameters
- Function return types are not considered for uniqueness. These would be invalid:

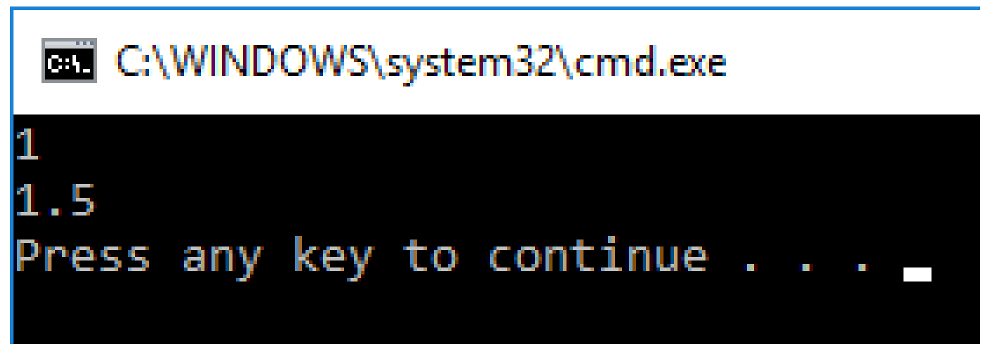
```
int add(double a, double b);
double add(double a, double b);
```

Example 1 – Different Number of Parameters

```
int add(int a, int b) {
        return (a + b);
}

int add(int a, int b, int c) {
        return (a + b +c);
}

int main() {
        int a = 3, b = 2;
        cout << add(a, b) << '\n';       //outputs 5
        int c = 1;
        cout << add(a, b, c) << '\n';  //outputs 6
        return 0;
}
```

Example 2 – Different Type of Parameters

```
int divide(int a, int b){
        return (a/b);
}

double divide(double a, double b){
        return (a/b);
}

int main(){
        int a = 3, b = 2;
        cout << divide(a, b) << '\n';
        double c = 3.0, d = 2.0;
        cout << divide(c, d) << '\n';
        return 0;
}
```



Example 3 – Different Type of Parameters

Will this work?

```
double multiply(int a, double b) {
      return (a * b);
}

double multiply(double a, int b) {
      return (a * b);
}

int main() {
      int a = 3;
      double b = 2.0;
      cout << multiply(a, b) << '\n';
      cout << multiply(b, a) << '\n';
      return 0;
}
```

Overloading Considerations

Function Declaration Element	**Used for Overloading?**
Function return type	No
Number of arguments	Yes
Type of arguments	Yes
Presence or absence of ellipsis	Yes
Use of **typedef** names	No
Unspecified array bounds	No
const or **volatile**	Yes (in arguments list)

Resolving Function Calls – Argument Matching

- Call to an overloaded function can have the following outcomes:
  - Match is found
    - arguments can be matched to a particular overloaded function
  - Match is not found
    - arguments cannot be matched to any overloaded function
  - Ambiguous match is found
    - Arguments matched more than one overloaded function

Resolving Function Calls – Argument Matching

- Argument matching – a match is found
  - An exact match was found.
  - A trivial conversion was performed.
  - An integral promotion was performed.
  - A standard conversion to the desired argument type exists.
  - A user-defined conversion (either conversion operator or constructor) to the desired argument type exists.
  - Arguments represented by an ellipsis were found.

Resolving Function Calls – Argument Matching

- Exact match

```
void Func(char *value);
void Func (int value);
Func(0); // exact match with Func(int)
```

- Match through promotion

```
void Func(char *value);
void Func(int value);
Func('a'); // promoted to match Func(int)
```

- Match through standard conversion

```
void Func(float value);
void Func(Shape value); //Shape may be a struct
Func('a'); // converted to match Func(float)
```

Evaluating Arithmetic Expressions

- When evaluating expressions, each expression is broken down into individual subexpression using:
  - Numeric promotion
    - If operand is narrower than an int, it is converted to an int
  - Numeric conversion
    - If operands still do not match, the compiler finds the highest priority operand and implicitly converts the other operand to match
    - Converting from larger to smaller operand can lead to unexpected results (and a compiler warning in most cases)

Numeric Promotion

- Numeric promotion is converting a value from one type to a value of a similar larger type
- Can be of two types:
  - Integral promotion
    - Conversion of smaller integral types (bool, char, short) into int
  - Floating point promotion
    - Conversion of float to double
- Promotions are always safe
  - No data loss can occur

Numeric Conversion

- Occurs when converting between different types
  - Includes converting from larger to a similar smaller type
    - Works as long as value fits in the smaller type
    - Example:

```
long d = 4.0;        //converts from double to long
short s = d ;        //converts from long to short
```

  - May result in data loss
    - No loss in example above as value fits, however, decimal value is discarded

Priority of Operands

- How does compiler decide what type to convert to?
  - It finds the highest priority operand for implicit conversion
- Priority of operands is as follows:
  - long double (highest priority)
  - double
  - float
  - unsigned long long
  - long long
  - unsigned long
  - long
  - unsigned int
  - int

Example

```
#include <iostream>
#include <typeinfo> // for typeid()

int main()
{
      long l(3);
      double d(4.0);
      short s(2);
      std::cout << typeid(l+d+s).name()
            << " " << l+d+s << std::endl;
      return 0;
}
```

Output is: double 9.0

Examples

`short a(10);`

`short b(20);`

Type and output of `a+b = Integer 30`

`char c('A');`

`short a(2);`

Type and output of `c+a = Integer 67`

`char a('A');`

`float b(20.9);`

Type and output of `a+b = Float 85.9`

How about std::cout 5u – 10?

4294967291

Ambiguous Matches

- Every overloaded function must have unique parameters
  - How can a call result in more than one match?
    - All standard conversions are considered equal
    - Ambiguous match results if a function call matches multiple candidates via standard conversion
  - Example:

```
void Func(long var);
void Func(float var);
Func('A');        //results in ambiguous match
//compiler returns an error
```

Ambiguous Matches

What happens with the argument for each function call, and which of the functions is called?

```
void Func(int value) { cout << value; }
void Func(float value) { cout << value; }

int main()
{
     Func('a');          //promotion, calls Func(int)
     Func(0);            //exact match, calls Func(int)
     Func(3.14159);      //ambiguous, results in error
     return 0;
}
```

Function Templates

- Overloaded functions can have same function body
  - The last two examples had the same function body but different arguments
- Function templates can be defined in C++ with generic types
  - Function template is defined like a regular function but using the keyword *template*
  - Syntax:
    template <template-parameters> function-declaration

How to Create Function Templates

- Consider the multiply function in the previous example:

```
int multiply(int a, int b) {
     int result = a * b;
     return result;
}
```

- For creating a template, replace all occurrences of ‘int’ with ‘Type’

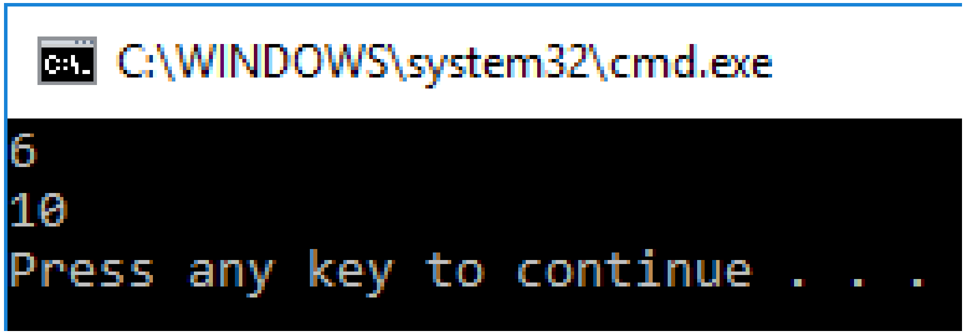
```
Type multiply(Type a, Type b) {
     Type result = a * b;
     return result;
}
```

- Now we just need to tell the compiler what ‘Type’ is:

```
template <typename Type>
```

Example 4 – Function Templates

```
template <class T>
T multiply(T a, T b) {
	T result = a * b;
	return result;
}
int main() {
	int a = 3, b=2;
	double c = 2.0, d = 5.0;
	cout << multiply<int>(a, b) << '\n';
	cout << multiply<double>(c, d) << '\n';
	return 0;
}
```



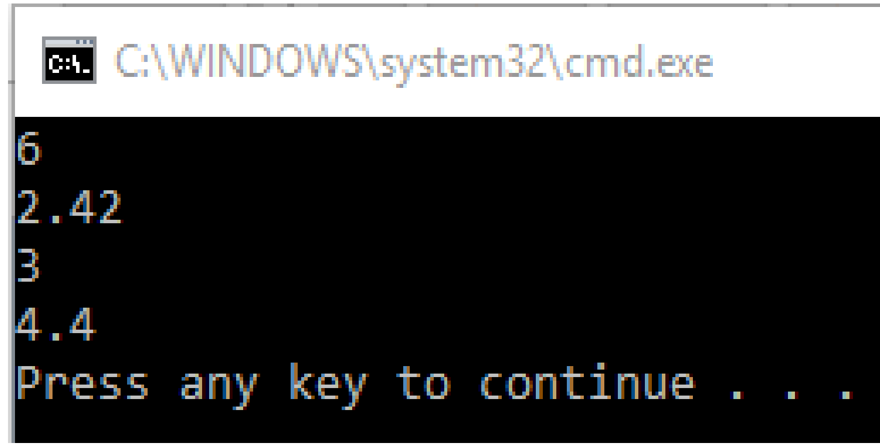
Exercise

- Rewrite Example 3 (below) using a Function Template

```
double multiply(int a, double b) {
      return (a * b);
}

double multiply(double a, int b) {
      return (a * b);
}

int main() {
      int a = 3;
      double b = 2.0;
      cout << multiply(a, b) << '\n';
      cout << multiply(b, a) << '\n';
      return 0;
}
```

Exercise – Solution

```
template <class T, class U>
      T multiply(T a, U b) {
      return a*b;
}
int main() {
      int a = 3, b=2;
      double c = 1.1, d = 2.2;
      cout << multiply(a, b) << '\n';
      cout << multiply(c, d) << '\n';
      cout << multiply(a, c) << '\n';
      cout << multiply(d, b) << '\
      return 0;
}
```



What could be a potential problem?

Operator Overloading

Operator Overloading

- Redefine the function of most built-in operators globally, or on a class-by-class basis
  - Overloaded operators are implemented as functions
- Operator overloading means changing the meaning of operators
  - One of the operands has to be a C++ user-defined type
    - User-defined types include class, struct, enum etc.
- Syntax:
  - *type* operator *operator-symbol* (*parameter-list*)
  - Example: Point operator< (Point &);

Which Operators can be Overloaded

- Arithmetic operators
  - +, -, *, /, %, +=, -=, *=, /=, %=, ++, --
- Boolean
  - ==, !=, <, >, <=, >=, ||, &&, !
- Bit manipulation
  - &, |, ^, <<, >>, &=, |=, ^=, <<=, >>=, ~
- Memory management
  - new, delete, new[], delete[]
- Miscellaneous
  - =, [ ], ->, ->*, *, &, ( )

Which Operators cannot be Overloaded

- Although most of the operators can be overloaded, following cannot:
  - . ,   :: ,   .* ,   #,   ##,   ?:

Rules of Operator Overloading

- Rule 1: Don't do it!
  - Unless the meaning of an operator is obviously clear in the application domain
  - Generally, it is better to provide a function with a clear and well-chosen name
  - Examples – when to use it:
    - Adding and subtracting complex numbers
    - Operating on point coordinates
- Rule 2: Stick to the operator's well known semantics
  - Use + to add, ++ to increment by one etc.
- Rule 3: Overload all related operators
  - a+b, a+=b; a++, ++a; a>b, a<b, a==b, a!=b

Member versus Non-member Functions

- Overloaded operators are functions with special names
- Can be implemented as:
  - Member functions of their left operand's type
  - Non-member functions

Member versus Non-member Functions

- Member functions
  - Binary operators (= [ ] ->) must always be implemented as member functions
- Non-member functions
  - If the left operand cannot be modified
  - Example: the input and output operators (<<, >>) whose left-operands are stream classes from the standard library

Member versus Non-member Functions

- Member or non-member
  - Implement unary operators as member functions
    - Example: ++ -- - !
  - Implement binary operator as member function of its left operand type if it has to access the operand's private members
  - Implement binary operator as non-member if it treats both operands equally (i.e. it does not change them)

General Rules of Implementation

- Unary operators
  - Take no arguments if declared as member functions
  - Take one argument if declared as non-member (global) functions
- Binary operators
  - Take one argument if declared as member function
  - Take two arguments if declared as non-member
- Operators that can be used either as unary or as binary (e.g. &, *, + , -) can be overloaded separately for each use
- Overloaded operators cannot have default arguments
- All overloaded operators except assignment (=) operator are inherited by the derived class
- The first argument for member-function overloaded operator is always of the class type of the object for which the operator is invoked

Ref: https://docs.microsoft.com/en-us/cpp/cpp/general-rules-for-operator-overloading

General Rules of Implementation

- Consider the following statements that are equivalent for built-in types (e.g. int i):
  - → i = i+1;
  - → i += 1;
  - → i++;
  - → ++i;
- For overloading, this functionality has to be explicitly provided for each operator
- Some requirements implicit to the use of these operators for basic types are relaxed for overloaded operators
  - Example: left operand of += must be an l-value for basic type, however, for overloaded operator, this is not a requirement

Example

- Overloading the + operator for a complex number
  - Example illustrates the member function implementation
  - For addition of complex numbers, the real parts of the two numbers and the imaginary parts are added separately
    - Example: (3+4i) + (2+5i) = 5+9i

Example

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

struct Complex {
   Complex (double r, double i) : real(r), imag(i) {}
   Complex operator+ (Complex &num);
   void Display() {cout<<real<<", "<<imag<<endl;}
private:
   double real, imag;
};

// Operator overloaded using a member function
Complex Complex::operator+ (Complex &num) {
   return Complex(real + num.real, imag + num.imag);
}
```

Example

```
int main() {
   Complex a = Complex( 3.0, 4.0 );
   Complex b = Complex( 2.0, 5.0 );
   Complex c = Complex( 0.0, 0.0 );

   c = a + b;  //outputs 5.0, 9.0 i.e. 5+9i
   c.Display();
}
```

Exercise

- Overload the * and / operators to multiply and divide the (x, y) coordinates of a point with a scalar
  - E.g. Point p(5,10);
  - Point rslt = p * 2   //returns (10,20)
  - Point rslt = p / 5   //returns (1,2)

Exercise – A Possible Solution

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

class Point {
public:
   Point (double x, double y) : ptX(x), ptY(y) {}
   Point operator* (int i);
   Point operator/ (int i);
   void Display() {cout<<ptX<<", "<<ptY<<endl;}
private:
   double ptX, ptY;
};

Point Point::operator* (int i) {
   ptX *= i;
   ptY *= i;
   return *this;
}
```

Exercise – A Possible Solution

```
Point Point::operator/ (int i) {
   ptX /= i;
   ptY /= i;
   return *this;
}

int main() {
   Point p1 = Point(5.0, 10.0 );
   Point p2 = Point(0.0, 0.0 );
   p2 = p1 * 2;   //outputs 10.0, 20.0
   p2.Display();
   p2 = p1 / 5; //outputs 2.0, 4.0
   p2.Display();
}
```