A1:
1.a: dynamic_cast afely converts pointers and references to classes up, down, and sideways along the inheritance hierarchy.

Base* b2 = new Derived;
Derived* d = dynamic_cast<Derived*>(b2);

1.b

i. 3
ii. 3
iii. 3.66667
iv. 4

1.c

class ClassName {
public:
constructor declarations
member function declarations
private:
data fields
}

Example Class :

```
class Point {
public:
      Point();// default constructor
      Point(int x, int y) : x(x), y(y) {}
      virtual ~Point(); // destructor

      // copy constructor
      Point(const Point& other);

      void print() const;
      int getX() const { return x; } // accessor
      int getY() const { return y; } // accessor
private:
      int x; // private variables
      int y;
};
```

2.a

By default, arguments in C++ programs are passed ***by value***. When arguments are passed by value, the system makes a copy of the variable to be passed to the function. Sometimes it is useful for the function to modify one of its arguments. To do so, we can explicitly define a formal argument to be a ***reference type***. When we do this, any modifications made to an argument in the function modifies the corresponding actual argument. This is called passing the argument ***by reference***.

```
void f (int value, int& ref ) { // one by value one by ref
value++;
ref ++;
cout << value << endl; cout << ref << endl;
}
int main() {
int cat = 1;
int dog = 5;
```

```
f(cat, dog);
cout << cat << endl; cout << dog << endl;
return EXIT SUCCESS;
}
```

2.b

i. 12
ii. 4
iii. 12
iv. 10

3.a

```
int k = 10;
while(k>0) {
cout << k << endl;
k -= 4;
}
```

3.b

there is a mistake with the condition and the loop never finishes

12
6
3
2
1
0
0
…

4.a

```
template <typename T> T func(T a, …) {
        // do something
        return …
}
```

4.b

```
template<typename T> max(T a, T b) {
        return (a>b) ? a : b;
}
```

5.

public: means that they are accessible from outside the class,
protected: accessible in the class that defines them and in other classes which inherit from that class.
private: means that they are accessible only from within the class.

Private is the default.

A.

1.

A program has to contain a global function named main, which is the designated start of the program. The main function cannot be used anywhere in the program and it cannot be called recursively.
The body of the main function does not need to contain the return statement: if control reaches the end of main without encountering a return statement, the effect is that of executing return 0;.

The include statement pulls in the statndard library definitions for streams, needed for writing to the console.
The main function has one statement which prints “goodbye!” to the console.

2.
0 Adam
1 Paul
2 Joe
3 Mary

(the reverse is pass-by-value, doesn't change arguments.

3.
```
class Complex {
public:
        Complex(const double& a, const double& b): a(a), b(b) {}
        double get_a() const { return a; }
        double get_b() const { return b; }
        Complex& operator*=(const Complex& other) {
                this->a = (a*other.get_a() - b*other.get_b());
                this->b = (a*other.get_b() + b*other.get_a())
                return *this;
        }
private:
        double a, b;
};
```

4.a
A function might be virtual if a further derived class needs to override it.

4.b
i. O(n^3)
ii. O(n^2)

5.a
abstraction: The notion of ***abstraction*** is to distill a complicated system down to its most fundamental parts and describe these parts in a simple, precise language. Typically, describing the parts of a system involves naming them and explaining their functionality. Applying the abstraction paradigm to the design of data structures gives rise to ***abstract data types*** (ADTs)
encapsulation : different components of a software system should not reveal the internal details of their respective implementations. One of the main advantages of encapsulation is that it gives the programmer freedom in implementing the de- tails of a system.
modularity: Modern software systems typically consist of sev- eral different components that must interact correctly in order for the entire system to work properly. Keeping these interactions straight requires that these different components be well organized. In object-oriented design, this code structuring approach centers around the concept of ***modularity***. Modularity refers to an organizing principle for code in which different components of a software system are divided into separate functional units.

inheritance: allows the design of generic classes that can be specialized to more particular classes, with the specialized classes reusing the code from the generic class.
polymorphism : means "many forms." In the context of object-oriented design, it refers to the ability of a variable to take different types. Polymorphism is typically applied in C++ using pointer variables. In particular, a variable *p* declared to be a pointer to some class S implies that *p* can point to any object belonging to any derived class T of S.

B1.

1.a

from lowest to highest:

v: 2^8

vii: 8n + 10

i

vi

iv

iii

ii

viii

1.b.

recursion: occurs when a function refers to itself in its own definition.

2.
Queue:
Another fundamental data structure is the ***queue***, which is a close relative of the stack. A queue is a container of elements that are inserted and removed according to the ***first-in first-out*** (***FIFO***) principle. Elements can be inserted in a queue at any time, but only the element that has been in the queue the longest can be removed at any time. We usually say that elements enter the queue at the ***rear*** and are removed from the ***front***.

The ***queue*** abstract data type (ADT) supports the following operations:

enqueue(*e*): Insert element *e* at the rear of the queue.

dequeue(): Remove element at the front of the queue; an error occurs if the queue is empty.

front(): Return, but do not remove, a reference to the front element in the queue; an error occurs if the queue is empty.

The queue ADT also includes the following supporting member functions:

size(): Return the number of elements in the queue.

empty(): Return true if the queue is empty and false otherwise.

Deque: queue-like data structure that supports insertion and deletion at both the front and the rear of the queue.

The functions of the deque ADT are as follows, where D denotes the deque:

insertFront(e): Insert a new element e at the beginning of the deque.

insertBack(e): Insert a new element e at the end of the deque.

eraseFront(): Remove the first element of the deque; an error occurs if the deque is empty.

eraseBack(): Remove the last element of the deque; an error occurs if the deque is empty.

Additionally, the deque includes the following support functions:

front(): Return the first element of the deque; an error occurs if the deque is empty.

back(): Return the last element of the deque; an error occurs if the deque is empty.

size(): Return the number of elements of the deque.

empty(): Return true if the deque is empty and false otherwise.

2.
We characterise an algorithm's running time as a function of the input size.

Hierarchy of functions:
constant
log
linear
n-log n
quadratic
cubic & other polynomials
exponential

3.
Seven primitive operations:
• Assigning a value to a variable
• Calling a function
• Performing an arithmetic operation (for example, adding two numbers) • Comparing two numbers
• Indexing into an array
• Following an object reference
• Returning from a function

4.
Algorithm1 is O(n^2). The inner loop over k=1 to j in combination with the outer loop leads to O(n^2).

Algorithm 1

```
1  Algorithm1(A, n)
2  Input: an integer array A of size n
3  X ← an integer array of size n
4  for j ← 0, n − 1 do
5      a ← 0
6      for k ← 1, j do
7          a ← a + A[k]
8      end for
9      X[j] ← a/(j + 1)
10 end for
11 return X
```

Algorithm2 has constant time O(1) operations in the for loop, so it is O(n).

Algorithm 2

```
1 Algorithm2(A, n)
2 Input: an integer array A of size n
3 X ← an integer array of size n
4 s ← 0
5 for i ← 0, n − 1 do
6       s ← s + A[i]
7       X[j] ← s/(i + 1)
8 end for
9 return X
```

Algorithm2 will be prefereable to Algorithm1 since O(n) is better than O(n^2).

B2.

1. Algortihm3 is using binary recursion since it calls itself twice in the return statement.

T(0) = 0
T(1) = 0
T(n) = T(n-1) + T(n-2) + c for n > 1

this leads to T(n) ~ O(2^n). Binary recursive fibonacci is exponential.

2. Algorithm4
The division by 2 on line.6 leads to O(log n).
It computes log_2 of n

3.

(n+4)^4 + (n+28)^3 + 6 n log n + 6(n/2) + 2 < (4 + 28 + 6 + 2)n^4
< n^4

when n0 == 2

The function is O(n^4)

4.
Stack:
A ***stack*** is a container of objects that are inserted and removed according to the ***last- in first-out*** (***LIFO***) principle. Objects can be inserted into a stack at any time, but only the most recently inserted (that is, "last") object can be removed at any time.

push(e): Insert element e at the top of the stack.

pop(): Remove the top element from the stack; an error occurs if the stack is empty.

top(): Return a reference to the top element on the stack, without removing it; an error occurs if the stack is empty.

Additionally, let us also define the following supporting functions:

size(): Return the number of elements in the stack.

empty(): Return true if the stack is empty and false otherwise.

Priority Queue:
A ***priority queue*** is an abstract data type for storing a collection of prioritized ele- ments that supports arbitrary element insertion but supports removal of elements in order of priority, that is, the element with first priority can be removed at any time.

size():	Return the number of elements in P.
empty():	Return true if P is empty and false otherwise.
insert(e):	Insert a new element e into P.
min():	Return a reference to an element of P with the smallest associated key value (but do not remove it); an error condition occurs if the priority queue is empty.
removeMin():	Remove from P the element referenced by min(); an error condition occurs if the priority queue is empty.

4.

```
Algorithm foo(L, n)
% n'th node from end of list L

% first get the size of the list
size = 0
cur = L.head
while cur != NULL:
        size += 1
        cur = cur->next

% now skip through the list until we reach element at (len - n + 1)
index = 0
cur = L.head
while index < (len - n + 1):
        cur = cur->next
        index += 1

return cur
```