Short Questions

2017

1. Clearly mark any lines causing a compile error below [1]

```
const int ci = 2;
int j = ci;
ci = j;
int i = 5;
const int cj = i;
std::cin >> ci;
```

2. Call the function `getAddIncrementCount` of `struct A` and print the return value to console, what will be printed? [1]

```
struct A {
  static int count;
  static int getAddIncrementCount() {
    return ++count;
  }
};
int A::count{0};

int main() {
  // Insert your code here






}
```

3. Convert the string "7" to the integer `i` using streams

```
string s{7};
int i;
```

4. Open the file "text.txt" for reading and print "File could not be opened" on failure.

5. Rewrite or mark up the function prototypes in the class `LetterGrade` using const and references as much as possible but not more. [1]

```
class LetterGrade {
  string d_mark{"INC"};
public:
  LetterGrade() = default;
  LetterGrade( string m )
    : d_mark(m) {}
  string get()
    { return d_mark; }
  void set(string m)
    {d_mark = m;}
  bool pass()
    { if ( d_mark < "D" || d_mark == "D+") return true; }
};
```

Consider the following definitions:

```
class Parent {
  int p{1};
public:
  virtual int getA() { return p; }
  virtual int getB()=0;
  int getC() { return p; }
};

class Child : public Parent {
  int p{2};
public:
  int getA() { return p; }
  int getB() override { return p; }
  int getC() { return p; }
};
```

6. What is printed by the following? [1]

```
Child c;
cout << c.getA() << " " << c.getB() << " " << c.getC() << endl;
```

7. What is printed by the following? [1]

```
Child c;
Parent& p = c;
cout << p.getA() << " " << p.getB() << " " << p.getC() << endl;
```

8. Modify the program to print "Downcast failed!" on failure of the dynamic cast. [1]

```
Child c;
Parent& p = c;


auto u = dynamic_cast<Child&>(p);


 std::cerr << "Downcast failed!" << endl;
```

2016

1. Change the following class to make it abstract

```
class A {
   int d_A;
 public:
   virtual void set( int a);
}

void A::set( int a ) {
  d_A = a;
}
```

2. What is printed by the following?

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

class Base {
public:
  virtual ~Base() {};
};

class D1 : public Base {
public:
  D1() = default;
};

class D2 : public Base {
};


int main() {
  D1 dA;
  Base* bA = &dA;
  Base& bB = dA;

  try {
        D2* d = dynamic_cast<D2*>(bA);
  } catch(...) {
        cout << "Error: bA" << endl;
  }
  try {
        D2& d = dynamic_cast<D2&>(bB);
  } catch(...) {
        cout << "Error: bB" << endl;
  }
}
```

3. The following class definition does not compile. Correct the error(s).

```
class Toto {
    const int d_data;
public:
    Toto() { d_data = 20; }
};
```

2015

1. What is printed by the following program?

```
int a{2};
bool b{false};
if (a&&b) {
  int c = a&b;
  cout << c << endl;
} else {
  int c = a|b;
  cout << c << endl;
}
```

2. Given the following definitions

```
class A {
public:
   virtual void print();
};
class B : public A { // omitted };
class C : public A { // omitted };
```

consider the code snippet:

```
A* a;
if ( someVariable > 0 ) {
  a = new B();
} else {
  a = new C();
}
B* b = a; // wrong! Replace this line by downcasting the object *a to *b but you also have to
          // check if the downcast succeeds
```

3. Consider the following two functions:

```
void makePair( int a, int b, int* p) {
            p[0] = a;
            p[1] = b;
}
void makePair( double a, double b, double* p) {
            p[0] = a;
            p[1] = b;
}
```

Define a function template replacing the functions above which can be instantiated correspondingly. For example:

```
int a, b, p[2];
makePairTemp( a, b, p );
double x, y, z[2];
makePairTemp( x, y, z );
```

4. Create a local array (on the stack) of two pointers to integer and then for each of the pointers allocate dynamically an array of 10 integers (on the heap).

5. Given the function declarations below

```
void func(const float a, const float& b, float *c ) {
  cout << "void func( " << a << ", " << b << ", " << *c  << " )" << endl;
}
```

Call the above function such that it prints: `void func(1, 2, 3);`

```
float a{1.},b{2.},c{3.};
```

6. There is one compile error in the program below, identify it.

```
#include <iostream>

struct A {
  void func() {
    std::cout << "A.func()" << std::endl;
  }
};

class B : protected A {
   void func() { A::func();}
};

class C : private B {
public:
   void func() { A::func();}
};

int main() {
  A a; B b; C c;
  a.func();
  b.func();
  c.func();
  return 0;
}
```

Short Programs

2016

1. What is printed by the following program? [3]

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

class Base {
   int d_b = 1;
public:
   Base() = default;
   Base( int b ) : d_b{b} {}
   int get() { return d_b; }
   virtual void set( int b) { d_b = b; }
   virtual void print() { cout << d_b << " "; }
};

class Derived : public Base {
   int d_d = 2;
public:
   Derived() = default;
   Derived( int d ) : d_d{d} {}
   virtual int get() { return d_d; }
   void set( int d) override { d_d = d; }
   virtual void print() { cout << d_d << " ";  }
};

int main() {
  Derived da(4), db, dc(3);
  da.print(); db.print(); dc.print(); cout << endl;
  Base* bPtr = &da;
  Base& bRef = db;
  Base bVal = dc;
  bPtr->print(); bRef.print(); bVal.print(); cout << endl;
  bPtr->set(5); bRef.set(6); bVal.set(7);
  cout << bPtr->get() << " " << bRef.get() << " " <<
          bVal.get() << endl;
  return 0;
}
```

2. Implement a deep assignment operator for the class `DArray`. [3]

```
class DArray {
  double* d_array;
  int d_size;
public:
  DArray(int sz) : d_size{sz} {
    d_array = new double[d_size];
  }
  ~DArray() {
    delete[] d_array;
  }
};
```

3. What is printed by the following program? [3]

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

class Point {
  int d_x=1, d_y=0;
public:
  Point() = default;

  Point(int abs, int ord=0) : d_x{abs}, d_y{ord} {
    cout << "ctor: " << d_x << " " << d_y << "\n"; }

  Point(const Point &);

  Point& add( const Point& oP ) {
    d_x += oP.d_x; d_y += oP.d_y;
    return *this; }

  ~Point();
};

Point::Point(const Point& oP) : d_x{oP.d_x}, d_y{oP.d_y} {
  cout << "copy-ctor: " << d_x << " " << d_y << "\n"; }

Point::~Point () {
  cout << "dtor : " << d_x << " " << d_y << "\n"; }

void fct (Point d, Point * add) {
  cout << "start (fct)\n";
  delete add;
  cout << "end (fct)\n" ;
}

main () {
  cout << "start (main)\n" ;
  Point a, b = 2;
  Point c = a;
  Point* adr = new Point(3,3);
  fct (a, adr);
  cout << "end (main)\n";
}
```

2015

2. Consider the following definitions of the class Time using a 24 hr clock.

```
class Time {
  unsigned char d_hour = 0;
  unsigned char d_minutes = 0;
public:
  inline Time();
  inline Time(unsigned char _hour, unsigned char _minutes);
  inline void get(unsigned char& _hour,
                  unsigned char& _minutes) const;
  inline void set(unsigned char _hour, unsigned char _minutes);
  inline void print() const {
   cout << static_cast<int>(d_hour) << ":" <<
           static_cast<int>(d_minutes);
  }
};
```

Leading to the following WorkWeek definition which uses growable, dynamically allocated arrays of start and end times of work shifts.

```
class WorkWeek {
  Time *d_start;
  Time *d_end;
  int d_size;
  int d_currNum = 0; // next available index to add a shift
public:
  WorkWeek(int _nShifts); // Part 2.a
  WorkWeek(const WorkWeek& _w); // Part 2.b
  ~WorkWeek(); // Part 2.c
  void print() const; // Part 2.d
  Time getTotalHours() const; // Part 2.e
  void addShift(unsigned char _hour,
                unsigned char _minutes, unsigned int _duration);
  // Part 2.f
};
```

a. Define a constructor that dynamically allocates the Time arrays d_start and d_end. [2]

```
WorkWeek::WorkWeek(int _nShifts)
```

b. Define a copy constructor implementing a deep copy strategy [2]:

```
WorkWeek::WorkWeek(const WorkWeek& _w)
```

c. Define the destructor for WorkWeek [2].

```
WorkWeek::~WorkWeek()
```

d. Define the print function for WorkWeek which should print all shifts currently stored in WorkWeek by calling the print function for Time [2]. For example, a shift starting at 8:30 and ending at 12:50 should print in a single line as: `8:30 to 12:50`

```
void WorkWeek::print() const
```

e. Define the function getTotalHours that counts up the total time for all shifts stored in WorkWeek [3].

```
Time WorkWeek::getTotalHours() const
```

f. Define the function addShift for WorkWeek which adds a new shift at the next available slot. If the arrays in WorkWeek are too small, create new arrays twice the size of the old array (growable array strategy) [4].

```
void WorkWeek::addShift(unsigned char _hour,
     unsigned char _minutes, unsigned int _durationMinutes )
```

3. Consider the following program and specify what is printed to the console [5]

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

class Food
{
  protected:
    double calories;
    std::string name;
  public:
    Food(){ cout << "Food constructor" << endl;}
    ~Food() { cout << "Food destructor" << endl; }
    std::string getName() const { cout << "Food name" << endl;
  return name;}
    virtual double getCalories() const { cout << "Food
  calories" << endl; return calories;}
};

class Cake : public Food
{
  private:
    double weight;

  public:
    Cake() { cout << "Cake constructor" << endl; }
   ~Cake(){ cout << "Cake destructor" << endl; }
    std::string getName() const {
      cout << "Cake name" << endl; return name;
    }
    double getCalories() const {
      cout << "Cake calories" << endl; return calories;
    }
};
```

```
void fct(const Food &d) {
    Cake c;
    return;
}
```

```
int main()
{
    Food *food;
    Cake *cake;

    std::cout << "[1]" << std::endl;
    food= new Cake();

    std::cout << "[2]" << std::endl;
    Cake cakes[3];


     std::cout << "[3]" << std::endl;
     cake= dynamic_cast<Cake*>(food);

     std::cout << "[4]" << std::endl;
     food->getCalories();
     food->getName();
     std::cout << "[5]" << std::endl;
     cake->getCalories();
     cake->getName();

     std::cout << "[6]" << std::endl;
     cakes[0]= *cake;

     std::cout << "[7]" << std::endl;
     fct(cake[1]);

     std::cout << "[8]" << std::endl;
     delete food ;
     std::cout << "[9]" << std::endl;
 }
```

2017

Consider the class definition of `StringStore` that holds strings in a growable array in the class variable `d_store`. The class is to implement internal aggregation for the growable array. Similar to the standard library, the current size of the array at `d_store` is stored in the class variable `d_capacity` while the number of strings stored is held in the class variable `d_size`.

```
class StringStore;
ostream& operator<<( ostream& os, const StringStore& stS);

class StringStore {
  std::string* d_store{0}; // pointer to array
  size_t d_size{0}; // no. of elements in array
  size_t d_capacity{0}; // size of array
public:
  StringStore() = default;
  StringStore(const StringStore& oStS );
  StringStore& operator=(const StringStore& oStS);
  ~StringStore();
  StringStore(std::vector<std::string>& sVec );
  void add(const string& str);
  friend std::ostream& operator<<( std::ostream& os,
                                     const StringStore& stS);
};
```

1. Implement the constructor `StringStore(std::vector<std::string>& sVec );` The constructor is to create an array of strings to store all the strings passed in by the sVec. Set the size and capacity of the StringStore to the number of strings in sVec. [2]

2. Implement internal aggregation for the copy constructor of `StringStore`. [2]

3. Implement internal aggregation for the assignment operator of `StringStore`. [3]

4. Implement internal aggregation for the destructor of `StringStore`. [1]

5. Implement the insertion operator to insert all strings to `ostream`, one string per line.[2]

6. Implement the function `void add(const string& str);` The function is to add the passed `str` to the `StringStore`. If the array `d_store` is full, it's capacity is to double to make room for the passed `str`. (growable array). [4]

Programming

2017

Consider the class definition of `Route` that is derived from StringStore. It stores a start and destination as a string and the street names for the route in the `StringStore`.

```
class Route;
ostream& operator<<( ostream& os, const Route& rt);

class Route : protected StringStore {
  string d_destination;
  string d_start;
public:
  Route() = delete;
  Route(std::vector<string>& streetNames, string start, string
  destination);
  void add(const string& streetName );
  friend ostream& operator<<( ostream& os, const Route& rt);
};

int main() {
  std::vector<string> streetsA{"Wellington","Bronson","Laurier"};
  Route rtA{streetsA,"Home","School"};
  rtA.add( "King Edward");
  cout << rtA;
  return 0;
}
```

Program Output:

```
From: Home to School
Wellington
Bronson
Laurier
King Edward
```

7. Is it necessary to implement the copy constructor, assignment operator and destructor for the class Route? Explain in one sentence why. [1]

8. Implement the function `void add(const string& streetName );` [1.5]

9. Implement the insertion operator for `Route`.

```
ostream& operator<<( ostream& os, const Route& rt); [1.5]
```

2016

The class `LinkedList` holds a singly linked list of integers. Each integer is stored in an object of type Node with a field containing a number and a field containing a pointer to the following node. The `LinkedList` class is to use **internal aggregation** and hence it overloads the copy constructor, assignment operator and destructor. Consider the following definitions of the class `LinkedList` with its helper structure `Node`.

```
struct Node {
  int d_value ; // value of an element
  Node *d_next ; // pointer to the next node in the list
};

class LinkedList {
  Node *d_start; // pointer to the beginning of the list or null
  int d_nbElem; // the current number of elements - convenience
public:
  LinkedList(); // constructor creating an empty LinkedList
  LinkedList(const LinkedList&); // copy constructor
  ~LinkedList(); // destructor
  LinkedList& operator=(const LinkedList&);// assignment operator
  void add(int); // add an element to the list
  bool contains(int) const; // check if an element is in the list
  int nbElem() const;  // return number of elements in the list
};
```

1. Implement the default constructor `LinkedList()` to simply initialize a new `LinkedList` which is empty. [1]

```
LinkedList::LinkedList()
```

2. Implement the accessor `nbElem()` to simply return the current number of elements in the list.[1]

```
int LinkedList::nbElem() const
```

3. Implement `contains(int)` to return true if the integer `value` is in the list, false otherwise [2].

```
bool LinkedList::contains(int value) const
```

4. Implement `add(int)` to create a new Node and add an element to the linked list. [3]

```
bool LinkedList::contains(int value) const
```

5. Implement the destructor `~LinkedList()` You can assume that all `Node` objects have been dynamically allocated on the heap. [3]

```
LinkedList::~LinkedList() {
```

6. Implement the copy constructor `LinkedList(const LinkedList&)` You must use internal aggregation. [4]

```
LinkedList::LinkedList(const LinkedList& oL)
```