Question 2

0 / 1 pts

Which of the following statements is true?

Correct answer

- All instances of a specific subroutine are the same size on the call stack.
- A new subroutine can not be added onto the call stack until the currently active one has finished.
- A subroutine can regain program control even if the subroutine on top of it on the call stack is not finished.

You Answered

- An active subroutine on the call stack can change size while on the stack.

Question 3

1 / 1 pts

What does this operation do?

some_integer << 5

- Return a value equal to some_integer with the bits shifted 5 spaces to the left, right 5 bits filled with 1
- Shift the bits within some_integer 5 spaces to the left, filling the blanks with 1
- Shift the bits within some_integer 5 spaces to the left, filling the blanks with 0
- **Correct!** Return a value equal to some_integer with the bits shifted 5 spaces to the left, right 5 bits filled with 0

Question 4

0 / 1 pts

The short integer 3672 (HEX: 0x0E58 - BIN: 0000 1110 0101 1000) is packed into a character string in little endian format.
What will the binary values of the characters in the array (indexes [0, 1]) be?

- **Correct answer** [0101 1000, 0000 1110]
- **You Answered** [0000 1110, 0101 1000]
- [0001 1010, 0111 0000]
- [0111 0000, 0001 1010]

Question 5

0 / 1 pts

What will the following code do:

my_file_stream.seekp(0, ios::end)

Correct answer

- Make sure that the next write operation will append to the end of the file.
- Make sure the next read operation will read the file from the start.
- Make sure the next write operation will write to the beginning of the file.

You Answered

- Return a boolean value stating whether the reading position has reached the end of a file.

Question 1

1 / 1 pts

The first line of a program is:

```
int some_integer = 7;
```

Either one of the following statements is true and the other three false, or one of them is false while the other three are true. Choose the one that is different.
(*choose* $X \in \{T, F\}$ *where the other three are* $\neg X$)

- The syntax **int *p = &some_integer;** is allowed because the **&** operator returns the memory location of the variable **some_integer**.

Correct!

- The syntax **int *p = &some_integer;** is allowed because the **&** operator reserves new memory on the heap with the value from **some_integer** and returns that new location.

- The syntax **int *p = new int[some_integer];** is allowed because the **new** operator with the [ ] operator reserves new memory on the heap for consecutive (samhliða) integer values, as many as the value of **some_integer** determines, and returns the location to the beginning of that memory.

- The syntax **int *p = new int;** is allowed because the **new** operator reserves new memory on the heap for a single integer and returns that new location.

Question 2

1 / 1 pts

The first line of a program is:

```
int *some_pointer;
```

Later in the program **some_pointer** is assigned a memory location with an integer value.

Either one of the following statements is true and the other three false, or one of them is false while the other three are true. Choose the one that is different. (*choose* $X \in \{T, F\}$ *where the other three are* $\neg X$)

- The syntax **int some_integer = *some_pointer;** is allowed because the * operator changes the value of **some_pointer** to the memory location of **some_integer**.
- The syntax ***some_pointer = some_integer;** is allowed because the * operator changes the value of **some_pointer** to the memory location of **some_integer**.

Correct!

- The syntax **int some_integer = *some_pointer;** is allowed because the * operator returns the value of the memory location that **some_pointer** points to, thus copying the underlying integer value into **some_integer**.
- The syntax **int some_integer = &some_pointer;** is allowed because the & operator returns the value of the memory location that **some_pointer** points to, thus copying the underlying integer value into **some_integer**.

Question 3

0 / 1 pts

The following program is compiled and run:

```
The following program is compiled and run:

void operation(){
    int array_1[10];
    int *array_2 = new int[10];

    for(int i = 0; i < 10; i++){
        cout << array_1[i] << " : " << array_2[i] << endl;
    }
}

int main(){
    operation();
                                    <-------- WE ARE HERE IN THE PROGRAM
    do_some_other_stuff();
}
```

When the program is at the point marked with "WE ARE HERE..." what is the status of the memory allocated for the arrays in operation()?

Correct answer

- array_2 still has memory reserved that can not be reused but there is no way to reference that memory ever again in the program.
- Both arrays still have memory allocated but no way to access that memory.

You Answered

- The memory for both arrays was cleaned up when operation() was taken off the stack.
- array_2 still has memory reserved but we can access that memory again when operation() is called again.

Question 4

1 / 1 pts

The following program is compiled and run:

```
int* operation(){
    int array_1[10];
    int *array_2 = new int[10];

    for(int i = 0; i < 10; i++){
        cout << array_1[i] << " : " << array_2[i] << endl;
    }

    return array_2;
}

int main(){
    int *ip = operation();
                        <-------- WE ARE HERE IN THE PROGRAM
    for(int i = 0; i < 10; i++){
        cout << ip[i] << ", ";
    }

    do_some_other_stuff();
}
```

When the program is at the point marked with "WE ARE HERE..." what is the status of the memory allocated for the arrays in operation()?

Correct!

- (●) array_1 has been cleaned up but array_2 is still accessible through the variable ip.
- ( ) array_1 still has memory reserved but we can access that memory again when operation() is called again.
- ( ) array_1 has been cleaned up and the variable ip is a pointer to a memory address which has nothing allocated to it.
- ( ) array_1 still has memory reserved that can not be reused but there is no way to reference that memory ever again in the program.

Question 5

0 / 1 pts

The following program is compiled and run (if possible):

```
int* operation(){
    int array_1[9];

    return array_1;
}

int main(){
    int *ip = operation();

    for(int i = 0; i < 10; i++){
        cout << ip[i] << ", ";
    }
                        <-------- WE ARE HERE IN THE PROGRAM
    do_some_other_stuff();
}
```

When the program is at the point marked with "WE ARE HERE..." what has happened?

Correct answer

- The program never compiled because it is trying to return statically allocated memory which will not exist anymore once operation() is taken off the stack.

- 10 values from array_1 have been printed to the screen.

You Answered

- The program never compiled because it is not possible to assign a statically allocated int array to a variable declared as int*.

- The program crashed because the for loop reads further than the allocated memory (too many iterations).

Question 1

1 / 1 pts

Consider the following code which will be compiled with: **g++ *.cpp -std=c++11**

```
#include <iostream>
#include <string>
#include <random>

class DataClass{
public:
    virtual ~DataClass(){
        delete[] numvec;
    }
    std::string str;
    int *numvec = nullptr;
};

int main(){
    DataClass *dc = new DataClass();
    dc->str = "Your Name";
    dc->numvec = new int[7];
    for(int i = 0; i < 7; i++){
        dc->numvec[i] = (rand() % 90) + 10;
    }
    for(int i = 0; i < 7; i++){
        std::cout << dc->numvec[i] << " ";
    }
    delete dc;
    std::cout << std::endl;

    //NOW WE DO MORE THINGS...
}
```

What is wrong with this program and could cause it to either not compile, crash later on or leak memory down the line?

Correct!

- (●) There is nothing wrong with this program
- ( ) "delete dc" should be "delete[] dc"
- ( ) DataClass must explicitly define at least one constructor if we want to construct it with "new DataClass()"
- ( ) The memory will not be correctly deleted because we "new" it in main() but try to "delete" it in ~DataClass()

Question 3

1 / 1 pts

Consider the following code which will be compiled with: **g++ *.cpp -std=c++11**

It is intended to write the number array to a file and read the same numbers back out, so it will print the same thing before the write as after the read.

```
#include <iostream>
#include <random>
#include <fstream>

int main(){

    int count = 14;
    int *ivec = new int[count];
    for(int i = 0; i < count; i++){
        ivec[i] = (rand() % 90) + 10;
    }

    std::cout << "BEFORE WRITE" << std::endl;
    std::cout << "count: " << count << std::endl;
    for(int i = 0; i < count; i++){
        std::cout << ivec[i] << " ";
    }
    std::cout << std::endl;

    std::ofstream fout("file.bin", std::ios::binary);
    fout.write((char *)&count, sizeof(count));  //LINE 1
    fout.write((char *)ivec, sizeof(ivec));     //LINE 2
    fout.close();

    for(int i = 0; i < count; i++){
        ivec[i] = 0;
    }
    delete[] ivec;

    std::ifstream fin("file.bin", std::ios::binary);
    fin.read((char *)&count, sizeof(count));  //LINE 3
    ivec = new int[count];
    fin.read((char *)ivec, sizeof(ivec));     //LINE 4
    fin.close();


    std::cout << "AFTER READ" << std::endl;
    std::cout << "count: " << count << std::endl;
    for(int i = 0; i < count; i++){
        std::cout << ivec[i] << " ";
    }
    std::cout << std::endl;
}
```

Why will this not work as intended?

- Lines 2 and 4 are only writing and reading the **memory location** of *ivec*, not the actual integer array data.
- *ivec* should be **referenced** in lines 2 and 4 like *count* is in lines 1 and 3. It should be **(char *)&ivec** to reference the actual data

Correct!

- Lines 2 and 4 are only writing and reading the **first** value from *ivec*, not the whole array. We should use **count * sizeof(int)** rather than **sizeof(ivec)**.
- C++ will not let us type cast an **int*** into a **char***

Question 4

1 / 1 pts

Consider the following code which will be compiled with: **g++ *.cpp -std=c++11**
There are implementations of the functions further down in the program code.

WorkingClass is a **class**.
wc is an **instance**.

```
class WorkingClass{
public:
    static void do_the_static_work();
    void do_the_non_static_work();
};

int main(){
    WorkingClass wc;
    wc.do_the_static_work();                //LINE 1
    wc.do_the_non_static_work();            //LINE 2
    WorkingClass::do_the_static_work();     //LINE 3
    WorkingClass::do_the_non_static_work(); //LINE 4
}
```

Which lines work and why?

Correct!

- (selected) LINE 4 does not work because you can not call a non-static function directly through the **class**. Other lines work.
- All the lines work.
- Neither line 1 nor line 4 work for the reasons given in other choices.
- LINE 1 does not work because you can not call a static function through an **instance** of a class. Other lines work.

Question 1

0 / 1 pts

Consider the following code:

```
class Car{
public:
    void who_am_i(){
        cout << "I am a car" << endl;
    }
};

class Toyota : public Car{
public:
    void who_am_i(){
        Car::who_am_i();
        cout << "My name is Toyota";
    }
};

class Corolla : public Toyota{
public:
    void who_am_i(){
        Toyota::who_am_i();
        cout << " Corolla" << endl;
    }
};

int main(){
    Corolla corolla;
    corolla.who_am_i();
}
```

What will its output be and why?

- "I am a car." because without the virtual keyword C++ will call whichever class is at the top of the inheritance hierarchy.

You Answered

- "I am a car. My name is Toyota Corolla" because C++ will always find the version of the function at the bottom (subclass) of the inheritance hierarchy.

- "My name is Toyota Corolla" because we are calling Corolla directly and that version only calls its parent, Toyota, not the grandparent.

Correct answer

- "I am a car. My name is Toyota Corolla" because we are calling the class Corolla directly so it will always use that version.

Question 5

1 / 1 pts

Consider the following code which will be compiled with: **g++ *.cpp -std=c++11**
It uses the colon, : , as syntax to delegate constructors.

```
class RulingClass{
public:
    RulingClass();
    RulingClass(int number);
    RulingClass(int number, char letter);
};

int main(){
    RulingClass rc(4, 'c');
}

RulingClass::RulingClass(){
    std::cout << "No parameters" << std::endl;
}
RulingClass::RulingClass(int number) : RulingClass(){
    std::cout << "One parameters" << std::endl;
}
RulingClass::RulingClass(int number, char letter) : RulingClass(number){
    std::cout << "Two parameters" << std::endl;
}
```

What will the program print?
Before copying the code into VSC and thoughtlessly stealing the answer, please take a little time to look at the code and come to your own conclusion about what the answer will be and see if you agree with the actual output.

Correct!

- No parameters
  One parameters
- (•) Two parameters

- No parameters
- ( ) Two parameters

- ( ) Two parameters

- Two parameters
- ( ) No parameters

Question 2

0 / 1 pts

Consider the following code:

```
class Car{
public:
    void who_am_i(){
        cout << "I am a car" << endl;
    }
};

class Toyota : public Car{
public:
    void who_am_i(){
        Car::who_am_i();
        cout << "My name is Toyota";
    }
};

class Corolla : public Toyota{
public:
    void who_am_i(){
        Toyota::who_am_i();
        cout << " Corolla" << endl;
    }
};

int main(){
    Car *car = new Corolla();
    car->who_am_i();
}
```

What will its output be and why?

- "I am a car. My name is Toyota Corolla" because we are calling the class Corolla directly so it will always use that version.

You Answered

- "I am a car." because without the virtual keyword C++ will call whichever class is at the top of the inheritance hierarchy.

- "I am a car. My name is Toyota Corolla" because C++ will always find the version of the function at the bottom (subclass) of the inheritance hierarchy.

Correct answer

- "I am a car." because we are calling the operation as an instance of Car, so it will use the version defined in Car.

Question 3

1 / 1 pts

Consider the following code:

```
class Car{
public:
    virtual void who_am_i(){
        cout << "I am a car" << endl;
    }
};

class Toyota : public Car{
public:
    virtual void who_am_i(){
        Car::who_am_i();
        cout << "My name is Toyota";
    }
};

class Corolla : public Toyota{
public:
    virtual void who_am_i(){
        Toyota::who_am_i();
        cout << " Corolla" << endl;
    }
};

int main(){
    Car *car = new Corolla();
    car->who_am_i();
}
```

What will its output be and why?

Correct!

- (selected) "I am a car. My name is Toyota Corolla" because with the virtual keyword C++ will call whichever class is at the bottom (subclass) of the inheritance hierarchy.
- "I am a car." because without the virtual keyword C++ will call whichever class is at the top of the inheritance hierarchy.
- "I am a car." because we are calling the operation as an instance of Car, so it will use the version defined in Car.
- "I am a car. My name is Toyota Corolla" because we are calling the class Corolla directly so it will always use that version.

Question 4

1 / 1 pts

Consider the following code:

```
class Car{
public:
    Car(){
        my_string = "I am a car. ";
    }
    void who_am_i(){ cout << my_string; }
private:
    string my_string;
};

class Toyota : public Car{
public:
    Toyota() : Car(){
        my_string += "My name is Toyota";
    }
};

class Corolla : public Toyota{
public:
    Corolla() : Toyota(){
        my_string += " Corolla";
    }
};

int main(){
    Car *car = new Corolla();
    car->who_am_i();
}
```

What will its output be and why?

Correct!

- (selected) It will not compile because the subclasses are using the variable my_string which is not defined in their scope.
- "I am a car." because we are calling the operation as an instance of Car, so it will use the version defined in Car.
- It will not compile because the operation who_am_i() is not defined on the class Corolla.
- "I am a car. My name is Toyota Corolla" because the the string is built in the constructors, so the virtual keyword doesn't matter here.

Question 5

1 / 1 pts

Consider the following code:

```
class Car{
public:
    Car(){
        my_string = "I am a car. ";
    }
    void who_am_i(){ cout << my_string; }
protected:
    string my_string;
};

class Toyota : public Car{
public:
    Toyota() : Car(){
        my_string += "My name is Toyota";
    }
};

class Corolla : public Toyota{
public:
    Corolla() : Toyota(){
        my_string += " Corolla";
    }
};

int main(){
    Car *car = new Corolla();
    car->who_am_i();
}
```

What will its output be and why?

- It will not compile because the subclasses are using the variable my_string which is not defined in their scope.
- It will not compile because the operation who_am_i() is not defined on the class Corolla.
- **Correct!** "I am a car. My name is Toyota Corolla" because the the string is built in the constructors, so the virtual keyword doesn't matter here.
- "I am a car." because we are calling the operation as an instance of Car, so it will use the version defined in Car.

Question 4

0 / 1 pts

Which statement about network packets, ports and sockets in networking is correct?

Correct answer

A network packet is a block of binary data, one or more bytes, including information for its destination. A port is the value of a 16 bit unsigned integer (u_short) which is attached to each network packet. A socket is also an integer value, used as an ID when reading and writing to the network hardware/system. The network hardware/system uses the port value to decide which socket, if any, should receive a particular network packet. The operating system uses the socket to deliver the packet contents to the correct program.

A network packet is a small piece of transistor which is sent over a wire between computers. A port is a physical diode on the network hardware, each network card having between 1.024 and 65.536 ports, depending on sophistication. A socket is a small tube in the hardware that delivers the packet along the system hub to the computer's CPU, where it is delivered to the program currently running.

You Answered

A network packet is a block of binary data, one or more bytes. A port is the value of a 16 bit unsigned integer (u_short) which is stored in the stack or heap memory of a running process. A socket is also an integer value, specifically a pointer. The network polls the currently running program for the port value to decide if it is the recipient of an incoming packet. The socket points to the memory location where the network packet will be stored for reading.

A network packet is an electrical current running at different voltages depending on its destination. A port is the value of a 16 bit unsigned integer (u_short) containing the voltage that its socket is reading at. A socket is a file descriptor, set to sense current at that specific voltage, so only data delivered at that current is stored in the program which defined that socket.

Question 5

1 / 1 pts

Consider the following program which should wait for a remote client connection, otherwise print "invalid socket" to the terminal:

```
int main(){
    check_and_start_if_windows();
    SOCKET_TYPE my_socket = socket(AF_INET, SOCK_STREAM, 0);

    sockaddr socket_address;
    make_socket_address(&socket_address, TCP_PORT);
    ADDRESS_SIZE addr_size = sizeof(socket_address);
    SOCKET_TYPE new_socket = accept(my_socket, (sockaddr *)&socket_address, &addr_size);
    if(new_socket == -1 || new_socket == ~0){
        cout << "invalid socket" << endl;
        return false;
    }
}
```

Which of the following fixes will result in an error free execution of the program, where it waits for the connection and accepts an incoming connection request on the TCP_PORT?

The program will work unchanged and wait for the client connection not returning from the call to ***accept()*** until a connection has been established.

The socket needs to be set to listening in order to accept a client connection. Therefore the program must include a call to ***listen()*** in order for it to wait for the connection:

```
    listen(my_socket, 1);
```

The socket must be bound to a specific port and address in order for it to accept a client connection. Therefore the program must include a call to bind():

```
    bind(my_socket, &socket_address, sizeof(socket_address));
```

Correct!

The socket needs to be set to listening in order to accept a client connection. However listening for connection requests will fail unless it is bound to a port and an address (even if the address is INADDR_ANY, meaning its not specified). There for it is necessary to first bind the socket and then set it to listening in order for the call to ***accept()*** to wait for a connection:

```
    bind(my_socket, &socket_address, sizeof(socket_address));
    listen(my_socket, 1);
```