1. C++ inherits C pointers and pointer arithmetics. Does C++ still need such unsafe pointers? Explain why (not?). . How does the use of pointer facilities differ in C and in C++? Give general advice how to reliably use C pointers in C++ programs. Show how and explain your solutions.

C++ does not really need such unsafe pointers as they can use references instead. Pointers are generally unsafe and can hold null values whereas references cannot, it is a better practice to use references than pointers and C++ to some extent hides the fact and makes it seem as if we are passing by value. For example,

```
main()
{
   int i = 10;
   doSomething(i);  // passes i by references since doSomethingElse() receives it
                            // by reference, but the syntax makes it appear as if i is passed
                           // by value
}


public void doSomething(int& i)  // receives i as a reference
{
   cout << i << endl;
}
```

Instead of having to use a pointer in which we need to pass the reference explicitly by using &i

```
main()
{
   int i = 10;
   doSomethingElse(&i);
}
public void doSomethingElse(int* i)
{
   cout << *i << endl;
}
```

The use of pointer differs in C and C++ in terms of you need to explicitly cast away const pointer in C++ but not in C.

For example, this would work in C

const char* p = &buffer[0];

char* c = (p + 6);

but you need to explicitly cast away the const in C++.

char* c = (char*) p + 6;

Generally we would use references over pointer but if we want to decouple compilation units to improve compilation time we can use C pointer, the useful property of a C pointer is that we only require a forward declaration of the pointed-to type, which will improve our compilation time. If we want to use pointers dynamically, we can make use of the new operator in C++ in which it will assign an address to the pointer itself, however we should remember to deallocate the memory on our own during the program execution.

unsigned short * pPointer;

pPointer = **new** unsigned short;

delete pPointer;

2. Find out how to create new dynamic objects in C++ (i.e., objects allocated from the heap). How do you create and destroy single objects vs. arrays of objects, respectively? How can we assure that such objects really are appropriately destroyed, i.e., how to prevent memory leaks? Describe and discuss the techniques we can use to customize the memory management of dynamic objects in C++. Can you create a new object in some given buffer or some other reserved (preallocated) memory area? Explain how/why not. (Hint: placement new and destructor.)

Answer: The dynamic objects can be created with the new operator. For a single object we can use

MyClass *obj = new MyClass;, For an array of object we can use MyClass *oArray = new MyClass [50]{}; And to destroy them w can use delete [] oArray and delete obj for the object and array.

We can assure that such objects are appropriately destroyed by ensuring that any dynamically allocated memory within the class is destroyed by calling delete in the destructor of the class. After deallocating the object, you can also assign null to the pointer so that the pointer does not point to the same memory location.

Another technique is to use a smart pointer in which we can define the pointers in the memory header file, they ensure proper destruction of dynamically allocated objects and are automatically deleted.

Yes we can create a new object in some given buffer or other reserved memory area, we can use the operator new to return a pointer to a chunk of memory enough to hold the object, it would obtain a raw memory for the particular object, and we then can deallocate the object by using operator delete.

```
void *operator new(size_t);       // allocate an object

void *operator new[](size_t);     // allocate an array

void *operator delete(void*);     // free an object

void *operator delete[](void*);  // free an array
```

3. Define what is meant by so-called callbacks. What is their purpose and in which languages and systems are they commonly used? Compare callback systems with event/message systems. What are the similarities and differences? In your opinion, which one do you consider to be more fundamental and essential in game/engine programming? Can you apply one of the techniques to implement the other one - which one, and how?

A callback is a function passed as an argument to another function. The purpose is normally for asynchronous function, where one function has to wait for another function to load, callback ensure that a function is not going to run before a task is completed but will only run right after it has completed.It is prevalent in JavaScript and normally used to assign events to UI elements.

Event/Message system is a type of callback in which when it is called whenever an event occurs. Their difference is that callback is needed to complete before further processing can be done but for event/message system it is processed by eventqueue and is asynchronous. I consider event/message system as more fundamental and essential as we wouldn't want too many callback functions each halting the process and making our game slow. We can apply the event/message system as callback in which we pass the event handler into another function, the event handler function would be called when the said function is called.

4. In your own words, explain the Typed Message design pattern. Devise an illustrative example, using the C++ language, and explain it. The example should show how to send a typed message. For that message kind, define two separate receivers which process a message in different ways. Compare Typed Message pattern with Observer design pattern with regard to expressivenes and modularity.

Answer:

Typed Message design pattern is a type-safe solution in which the senders and handlers do not have to know of each other but just the events/messages.

Say we have the template

```
template void DeliverTypedMsg (T const& msg)
{
 TypedMsgHandler ::Deliver (msg); // forwards the msg
}
```

We just need to derive a subclass to receive the message

```
class GameEntity: public TypedMsgHandler  {
public:
        void handle (std::string const& msg) override {
        std::cout << "Game event received: " + msg << std::endl;
       }
};
```

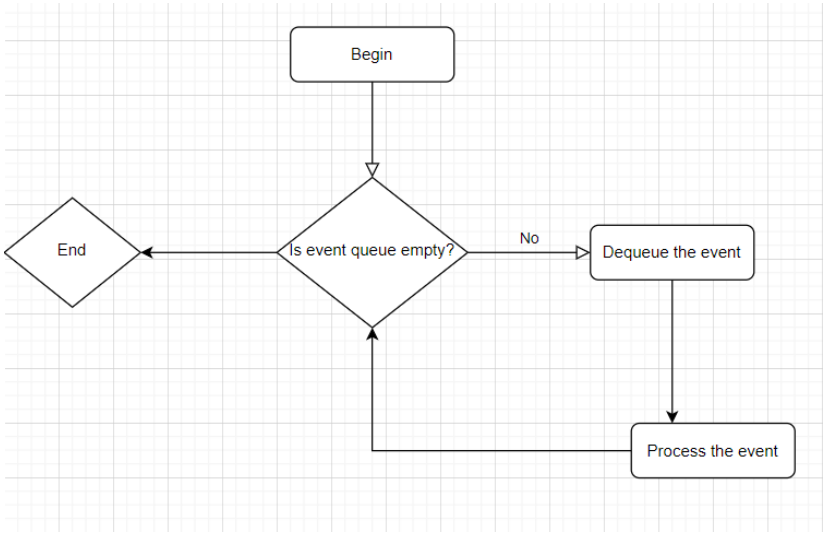
In terms of modularity, the observer is less modular than typed message system as observer pattern is tightly coupled whereas observer pattern can be its standalone class.

In terms of expressiveness, there is a graph of game entities that need to be introduced to each other in the observer pattern, so it is not that expressive, however, for typed message design, you just need to derive a subclass in order to received the message.

5. Explore the Event Queue game programming pattern. Explain its general purpose, structure, and consequences. For what situations or purposes would Event Queue be used in game programs? Sketch a sample design for Event Queue. Implement it in a suitable pseudolanguage and explain verbally what is needed to make it work. Discuss the other (game) programming patterns related to this one. How do they resemble or differ from Event Queue?

Answer:

Event Queue would decouple when a message or event is sent from when it is processed, th queue would store a series of notifications or requests in first-in, first-out order. Sending a notification enqueues the request and returns. The request processor then processes items from the queue at a later time, which decouples the sender from the receiver. They can be used in the announcement system or audio engine in the game e.g. when an enemy is killed “enemy died” event is played, or a sound is played given an identifier.



In our class Audio we would initialise the number of pending requests to 0, have an update function that is called whenever there is a pending event.

```
class Audio
{
public:
 static void init()
 {
  numPending_ = 0;
 }
 static void update()
 {
  for (int i = 0; i < numPending_; i++)
  {
```

```
    ResourceId resource = loadSound(pending_[i].id);
    int channel = findOpenChannel();
    if (channel == -1) return;
    startSound(resource, channel, pending_[i].volume);
   }

   numPending_ = 0;
 }

private:
 static const int MAX_PENDING = 16;
 static PlayMessage pending_[MAX_PENDING];
 static int numPending_;
};
```

We can call the function to slot a new message in the array at the end.

```
void Audio::playSound(SoundId id, int volume)
{
 assert(numPending_ < MAX_PENDING);

 pending_[numPending_].id = id;
 pending_[numPending_].volume = volume;
 numPending_++;
}
```

In which we can call in our Menu class to play a sound

```
Class Menu
{
Public:
        Void onSelect(int index)
        {
                Audio::playSound(SREAM, MAX_VOLUME);
        }
```

}

We could have used Observer and Command pattern if our purposes are just to decouple who receive a message from its sender, the event queue is only needed if we want to decouple something in time. They are different in the sense that the observer pattern is highly coupled as the subject “knows” its “observers” and the observer would need to maintain a ref to the subjects to get its state.