3.4 C++. Data structures.

Question 1. How do you declare an array?

Answer: An array can be declared either on stack, or on heap.

```
//created on stack, uninitialized
T identifier[size];

//created on stack, initialized
T identifier[] = initializer_list;

//created on heap, uninitialized
T* identifier = new T[size];
```

Example:

```
int foo[3];
int bar[] = {1,2,3};
int* baz = new int[3];
```

Question 2. How do you get the address of a variable?

Answer: Use the ampersand before the name of the variable, e.g.

```
T var;
T* ptr = &var
```

Example:

```
int foo = 1;
int* foo_ptr = &foo;
```

Question 3. How do you declare an array of pointers?

Answer: The same way as declaring an array, but making the type, `T`, a pointer:

```
T* identifier[size];
T* identifier[] = initializer_list;
T** identifier = new T*[size];
```

Example:

```
int a = 1; int b = 2; int c = 3;
int* foo[3];
int* bar[] = {&a, &b, &c};
int** baz = new int*[3];
```

Question 4. How do you declare a const pointer, a pointer to a const and a const pointer to a const?

Answer:

```
//pointer to a read only variable
const T* identifier;
T const* identifier;

//read only pointer to a variable
T *const identifier = rvalue;

//read only pointer to a read only variable
const T *const identifier = rvalue;
T const *const identifier = rvalue;
```

Example:

```
//read only variables
const int a = 2; const int b = 2;
int c  = 1;

//pointer to a read only b
int const* foo_two;
foo_two = &a; foo_two = &b;

//pointer to read only a
```

```
const int* foo;
foo = &a; foo = &b;

//read only pointer to c
//it needs to be initialized
int *const bar = &c;

//read only pointer to read only a
//it needs to be initialized
const int *const baz = &a;
```

Question 5. How do you declare a dynamic array?

Answer:

```
T* identifier = new T[size];
T* identifier = nullptr;
T* identifier;

delete[] identifier;
```

Example:

```
int *foo = new int[4];
int *bar = nullptr;
bar = new int[4];
```

Question 6. What is the general form for a function signature?

Answer:

```
return_type function_name(parameter_list);
```

Example:

```
int my_sum(int a, int b);
```

Question 7. How do you pass-by-reference?

Answer:

```
return_type function_name(T & identifier);
```

The identifier is now an alias for the argument.

Question 8. How do you pass a read only argument by reference?

Answer:

```
return_type function_name(const T & identifier);
```

Once you define a parameter as `const`, you will not be able to modify it in the function.

Question 9. What are the important differences between using a pointer and a reference?

Answer: Several differences between using a pointer and a reference are:

- A pointer can be re-assigned any number of times, while a reference cannot be reassigned after initialization.
- A pointer can point to NULL (`nullptr` in C++11), while a reference can never be referred to NULL.
- It is not possible to take the address of a reference as it is done with pointers.
- There is no reference arithmetic.

Question 10. How do you set a default value for a parameter?

Answer:

```
return_type function_name(T identifier = rvalue)
```

The parameters with default value must be placed at the end of the parameter list.

Question 11. How do you create a template function?

Answer:

```
template<class T>
return_type function_name(parameter_list);

template<typename T>
return_type function_name(parameter_list);
```

Note that the parameter type can be specified, when calling the function, explicitly or implicitly. Also note that there is no technical difference between using `class` or `typename` besides code readability (`typename` for primitive types and `class` for classes).

Example:

```
template<typename T>
T temp_sum(T a, T b) {return a+b;}

struct Processor{
    int a;
    int apply(int b) {return a+b;}
};

template<class T>
int temp_sum_2(int a, int b) {
    T processor;
    processor.a = a;
    return processor.apply(b);
}

int main(){
    //implicit, foo equals 3
    int foo = temp_sum(1,2);

    //explicit, bar equals 3
    int bar = temp_sum_2<Processor>(1,2);
}
```

Question 12. How do you declare a pointer to a function?

Answer:

```
return_type (*identifier) (list_parameter_types)
```

Example:

```
int my_sum(int a, int b) {return a+b;}
int main(){
    int(*p_func)(int,int);
    p_func = & my_sum;

    // foo equals 3
    int foo = p_func(1,2);
}
```

Question 13. How do you prevent the compiler from doing an implicit conversion with your class?

Answer: Use the keyword `explicit` to define the constructor:

```
explicit Classname(parameter_list)
```

Question 14. Describe all the uses of the keyword `static` in C++.

Answer: Inside a function, using the keyword `static` means that once the variable has been initialized it remains in memory up until the end of the program.

Inside a class definition, either for a variable or for a member function, using the keyword `static` means that the there is only one copy of them per class, and shared between instances.

As a global variable in a file of code, using the keyword `static` means that the variable is private within the scope of the file.

Question 15. Can a static member function be const?

Answer: When the `const` qualifier is applied to a non-static member function it implies that member function can not change the instance class when called (i.e. can not change any non `mutable` members from `*this`). Since static member function are defined at a class level, where there is no notion of `this` the `const` qualifier for member functions does not apply.

Question 16. C++ constructors support the initialization of member data via an initializer list. When is this preferable to initialization inside the body of the constructor?

Answer: The initialization list has to be used for const members, references and with members without default constructors, but for any type of members initialization through the initialization list is still preferable, since it is for efficient. Using the initialization list, the members are initialized calling directly their constructors. If the initialization is done in the body of the constructor for each member being initialized there is an instance of it created and then a copy assignment operation is called to assign that instance to its respective member.

Question 17. What is a copy constructor, and how can the default copy constructor cause problems when you have
pointer data members?

Answer: A copy constructor allows you to create a new object as a copy of an existing instance. The default copy constructor creates the new object by copying the existing object, member by member, and thus when there are member pointer you end up with two objects pointing to the same object.

It is important to note that the copy constructor is

called every time a function receives an object via the pass-by-value mechanism. This means that the copy constructor needs to be implemented using a pass by reference. Otherwise you will be recursively calling the copy constructor. You should always set the parameter for a copy constructor to be const.

```
ClassName( const ClassName& other );
ClassName( ClassName& other );
ClassName( volatile const ClassName& other);
ClassName( volatile ClassName& other );
```

Question 18. What is the output of the following code:

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

class A
{
public:
    int * ptr;
    ~A()
    {
        delete(ptr);
    }
};

void foo(A object_input)
{
    ;
}

int main()
{
    A aa;
    aa.ptr = new int(2);
```

```
    foo(aa);
    cout<<(*aa.ptr)<<endl;
    return 0;
}
```

Answer: The output of the code is an uncertain number, depending on the compiler used; for some compilers it could generate an error. The reason for this is that we do not define our own copy constructor.

When we call the foo function, the compiler will generate a default copy constructor which will shallow copy every data members defined in class A. This will lead to the result that two pointers, one in temporary object and the other in the object aa, will point to the same area in memory. When we get out the foo function, the compiler will automatically call the destructor function of the temporary object in which the pointer will be deleting and the area it points to will be free. In this situation, the pointer in aa will still point to the same area which has been free. When we try to visit the data through the pointer in aa, we will get garbage information.

Question 19. How do you overload an operator?

Answer:

```
type operator symbol (parameter_list);
```

If you define the operator outside of the class, then it will be a global operator function.

Example:

```
struct FooClass{int a;};
int operator + (FooClass lhs, FooClass rhs) {
    return lhs.a + rhs.a;
}
```

Question 20. What are smart pointers?

Answer: A smart pointer is a class built to mimic a pointer (offering dereferencing, indirection, arithmetic) that also offers extra features to simplify the usage, sharing and management of resources.

C++11 comes with three implementations of smart pointers: `shared_ptr`, `unique_ptr`, and `weak_ptr`.

Example:

```
//shared_pointer maintains
//a reference count
//when the count is zero the object
//pointed to is destroyed
std::shared_ptr<int> foo(new int(3));
std::shared_ptr<int> bar = foo;

//memory not released
//bar is still in scope
foo.reset();

//releases the memory,
//since no one is using it
bar.reset();
}
```

Question 21. What is encapsulation?

Answer: Encapsulation is the ability to expose an interface while hiding implementation. This is usually achieved through access modifiers (public, private, protected, etc.).

Question 22. What is a polymorphism?

Answer: Polymorphism is the ability for a set of classes to all be referenced through a common interface.

Question 23. What is inheritance?

Answer: Inheritance is the ability for one class to extend another through sub-classing. This is also referred to as "white-box" (the opposite of "black-box") re–use. A library can provide base classes that may be extended by the application developer.

Question 24. What is a virtual function? What is a pure virtual function and when do you use it?

Answer: Virtual functions are functions that are resolved by the compiler, at runtime, to the most derived version with the same signature. This means that if a function that was defined using a base class `Foo`, with a virtual member function `f`, is called using an instance of a sub class `FooChild`, that function is going to be dynamically binded to the implementation of the sub class (regardless that the actual code only refers to the base class).

A pure virtual function is a virtual function with no implementation in the base class, making the base class abstract (and thus can't be instantiated). Derived classes are forced to override the pure virtual function if they want to be instantiated. You use the same syntax as the virtual function but add `=0` to its declaration within the class.

Question 25. Why are virtual functions used for destructors? Can they be used for constructors?

Answer: Destructors are recommended to be defined as virtual, so the proper destructor (in the class hierarchy) is called at running time.

When calling a constructor, the caller needs to know the exact type of the object to be created, and thus they cannot be virtual.

Question 26. Write a function that computes the factorial of a positive integer.

Answer:

```
//for implementation
int factorial(int n){
    int output =1;
    for (int i =2 ; i <= n ; ++i)
        output *= i;
    return output;
}

//recursive implementation
int factorial(int n){
   if (n == 0) return 1;
   return n*factorial(n-1);
}

//tail recursive implementation
int factorial(int n, int last = 1){
    if (n == 0) return last;
    return factorial(n-1, last * n);
}
```

Question 27. Write a function that takes an array and returns the subarray with the largest sum.

Answer:

```
#include <vector>
#include <algorithm> // std::max

using namespace std;

template <typename T>
T max_sub_array(vector<T> const & numbers){
    T max_ending = 0, max_so_far = 0;
    for(auto & number: numbers){
        max_ending = max(0, max_ending + number);
```

```
            max_so_far = max(max_so_far, max_ending);
        }
        return max_so_far;
}
```

Question 28. Write a function that returns the prime factors of a positive integer.

Answer:

```
#include <vector>
using namepsace std;

vector<int> prime_factors(int n){
    vector<int> factors;
    for (int i = 2; i <= n/i ; ++i)
        while (n % i == 0) {
            factors.push_back(i);
            n /= i;
        }
    if (n > 1)
        factors.push_back(n);
    return factors;
}
```

Question 29. Write a function that takes a 64-bit integer and swaps the bits at indices `i` and `j`.

Answer:

```
long swap_bits (long x, const int &i, const int &j){
    if ( ((x >>i) & 1L) != ((x>>j) & 1L) )
        x ^= (1L <<i ) | (1L <<j);
    return x;
}
```

Question 30. Write a function that reverses a single linked list.

Answer:

```
#include <memory> // shared_ptr
using namespace std;

template<typename T>
struct node_t {
    T data;
    shared_ptr<node_T<T>> next;
};

//recursive implementation
template<typename T> shared_ptr<node_t<T>>
    reverse_linked_list(
        const shared_ptr<node_t <T>> &head){
    if (!head || !head->next) {
        return head;
    }

    shared_ptr<node_t<T>>
      new_head = reverse_linked_list(head->next);
    head->next->next = head;
    head->next = nullptr;
    return new_head;
}

//while implementation
template<typename T> shared_ptr<node_t<T>>
    reverse_linked_list(
        const shared_ptr<node_t <T>> &head){
    shared_ptr<node_t<T>>
        prev = nullptr, curr = head;
     while(curr) {
  shared_ptr<node_t<T>> temp = curr-> next;
  curr->next = prev;
  prev = curr;
  curr = temp;
```

```
    }
    return prev;
}
```

Question 31. Write a function that takes a string and returns true if its parenthesis are balanced.

Answer:

```
#include<string>
#include<stack>
using namespace std;

bool is_par_balanced(const string input)
{
    //"())())"=> false
    //"(a(dd)()(()))"=>true
    stack<char> par_stack;
    for(auto &c: input)
    {
        if(c==')')
        {
            if(par_stack.empty())
            return false;
            else if(par_stack.top()=='(')
            par_stack.pop();
        }
        else if(c=='(')
        par_stack.push(c);
    }
    return par_stack.empty();
}
```

Question 32. Write a function that returns the height of an arbitrary binary tree.

Answer:

```
#include<memory> //std::shared_ptr
#include<algorithm> //std::max
using namespace std;

template <typename T>
struct BinaryTree {
    T data;
    shared_ptr<BinaryTree<T>> left, right;
};


template <typename T>
int height(
    const shared_ptr<BinaryTree<T>> &tree,
    int count = -1){
        if (!tree) return count;
        return max(
           height(tree->left, count + 1),
           height(tree->right, count +1));
}
```