# 1 C++ Templates

10. Using metaprogramming, write a templated class `prime`, which evaluates whether a literal integer constant is prime or not at compile time.

```
template<int p, int i> struct factor{
    enum{factorless = (i <= 1) || (p % i) && factor<p, i-1>::factorless};
};

template<int p> struct factor<p, 1>{
    enum{factorless = true};
};
template<int p> struct prime{
    enum {is_prime = !factor<p, p-1>::factorless};
};
```

11. How can you be sure that your implementation of class `prime` has been evaluated at compile time?

    Values held in enums must be known at compile time. Since the value we wish to compute is held in an enum, we know it must therefore have been evaluated it compile time.

    Secondly when I checked; the assembly generated for this program (with a main function that initialised a prime and output the value) was bit-for-bit identical to programs which output `true` or `false` – therefore the condition must have been evaluated at compile time.

# 2 Diamond Problem

What is the diamond problem? How does C++ deal with it? Illustrate your answer with code.

The diamond inheritance problem is a byproduct of multiple inheritance. It arises when a class inherits from different classes which themselves inherit from the same superclass. To prevent superclasses breaking each others invariants, C++ has multiple copies of the attributes from this doubly-inherited superclass. However if we request for an attribute which the subclass has multiple copies of then it is ambiguous which copy we meant. C++ requires the programmer to explicitly disambiguate using `::` (the scope resolution operator).

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

struct X{
    int x;
    X(): x(0){}
};

struct Y: X{
    int y;
    Y(): y(0){}
};

struct Z: X{
    int z;
    Z(): z(0){}
};
```

```
struct C: Y, Z{
    C()= default;
    void setyx(int param){
    // set the x attribute inherited through Y
        Y::x = param;
    }
    int getyx(){
    // return the x attribute inherited through Y
        return Y::x;
    }
    void setzx(int param){
    // set the x attribute inherited through Z
        Z::x = param;
    }
    int getzx(){
    // return the x attribute inherited through Z
        return Z::x;
    }
};

int main(){
    C c;
    c.setyx(10);
    c.setzx(5);
    cout<<c.getyx()<<endl; // outputs 10
    cout<<c.getzx()<<endl; // outputs 5
    return 0;
}
```

# 3 `virtual` keyword

1. Give an example where failure to make a destructor `virtual` causes a memory leak.

   If a destructor is not virtual then which destructor is called will be based on the static type of the object which is being deleted. If the static type is different from the runtime type – and the runtime type has additional attributes on the heap, then they will not be deallocated; causing a memory leak. Declaring a destructor as virtual means the destructor is determined at runtime and is therefore guaranteed to be the runtime type rather than the static type.

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

   struct Base{};

   struct Derived: public Base{
       set<int> *set_x;
       Derived(){
       // allocate a set<int> on the heap
           set_x = new set<int>();
       }
       ~Derived(){
       // deallocate the set<int> to prevent a memory leak
           delete set_x;
   ```

```
    }
};

int main(){
        // create a Base * that points to a Derived allocated on the heap
    Base *b = new Derived();

    /* this calls the Base destructor -- not Derived destructor
     * so set_x is never deallocated from the heap -- a memory leak! */
    delete b;
    return 0;
}
```

2. Why should destructors in an abstract class always be declared virtual?

   An Abstract class in C++ is a class with at least one pure virtual function. Abstract classes allow polymorphism by providing an interface specifying what functions should be supported without specifying implementation details. Functions can take abstract classes as arguments or return abstract classes.

   The point of an abstract class is that functionality has not been fully implemented – and that subclasses should be able to implement this functionality in whichever way they wish. Subclasses should therefore be able to have other attributes. However, if an object has the static type of the abstract class and is deleted then the destructor run will be that of the abstract class. This would leak any of the attributes that the subclass added. This can be solved by resolving the destructor functions by dynamic dispatch and calling the subclasses destructor. In C++ we do this by declaring the destructor virtual.

   Failure to declare the destructor virtual would prevent subclasses from adding new attributes; seriously restricting the way the subclass are implemented and making the abstract class almost unusable.

# 4 C++ Concepts

Explain what the following C++ concepts do and when one should use them:

(a) `volatile` keyword

The volatile keyword is used on variables which may be changed by multiple or processes. It ensures the compiler does not overly-optimise resulting in incorrect performance.

We should use the volatile keyword when writing multithreaded code with shared memory.

(b) `friend` keyword

A class can declare another class or a function as "friend" and allow the it to access its private methods and attributes.

This allows us to get the same effect as nested classes in Java. For example we can make complicated data structures in C++ by allowing nodes to be accessed by the structure; whilst also retaining private fields so that other methods cannot break invariants.

(c) pure virtual function

A pure virtual function is a function where the class provides no implementation details: the function is declared using the following syntax `A f(...) = 0;`. Any class

with any pure virtual functions is known as an abstract class and cannot be directly instantiated.

We should use pure virtual functions to define functionality that a type of class should support in order to enable polymorphism.

# 5 Overloading

C++ allows function overloading. Give an example of it. How and when is the executed function chosen?

Overloading means declaring multiple functions with the same name and return types but different argument types or lengths. There are then multiple definitions which the compiler will decide between.

Since C++ is a statically typed language, overload resolution (choosing which function to execute) can be performed at compile time. The compiler will choose the best match for the type signature. If there is a perfect match then this function will be called. Otherwise if the function arguments can be cast to *exactly one* function then this function will be called. Otherwise which function we wish to call is ambiguous and a compiler error will be thrown.

The function `f` is overloaded:

```
int f(int x){
        return x;
}

int f(char x){
        return x + 1;
}

int g(int x){
        return x;
}

int g(int x[]){
        return 1;
}

int main(){
        f(3); // 3
        f((char) 3); // 4
        f(3LL); // error!
        /* long long is not a perfect match but
         * can be cast to multiple declarations */

        g((char) 3); // 3
        /* which declaration of g we want is unambiguous
         * as char cannot be cast to int[] */
        return 0;
}
```

# 6 Pointers and References

Give 2 similarities and 2 differences between pointers and references. Support it with code if needed.

Similarities:

- Both pointers and references allow objects to be passed without copying.
- Both pointers and references can be cast to different types.

Differences:

- Pointers can point to uninitialised objects; while references can only point to initialised objects.
- We can perform pointer arithmetic on pointers – accessing data at offsets; while we cannot do comparable operations with references.