Extended CPP Notes

Nathan Warner

Northern Illinois
University

Computer Science
Northern Illinois University
April 17, 2024
United States

Contents

Templates

Templates in C++ are a powerful feature that allows writing generic and reusable code. They enable functions and classes to operate with different data types without being rewritten for each specific type.

1.1 Template Function

A template function defines a family of functions that work with different data types. Here's how to write and use a template function:

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

int main() {
    std::cout << add<int>(3, 4) << std::endl; // Instantiates add<int>
    std::cout << add<double>(2.5, 3.1) << std::endl; // Instantiates add<double>
    return 0;
}
```

1.2 Template Class

```cpp
template <typename T>
class Stack {
    std::vector<T> data;

public:
    void push(T value) {
        data.push_back(value);
    }
    void pop() {
        data.pop_back();
    }
    T top() const {
        return data.back();
    }
    bool empty() const {
        return data.empty();
    }
};

int main() {
    Stack<int> intStack; // Instantiates Stack<int>
    intStack.push(10);
    intStack.push(20);
    std::cout << intStack.top() << std::endl;

    Stack<double> doubleStack; // Instantiates Stack<double>
    doubleStack.push(1.1);
    doubleStack.push(2.2);
    std::cout << doubleStack.top() << std::endl;
    return 0;
}
```

1.3 Class vs typename keyword

The choice between using **class** and **typename** in template declarations in C++ is largely a matter of style and historical context, as both keywords serve the same purpose

1.4 Handle friend functions

1.4.1 Friendship to a Non-Template Function

This is straightforward. You directly declare a non-template function as a friend inside your template class. This grants that specific function access to all instances of the template class, regardless of the type parameter.

```cpp
template <typename T>
class MyClass {
    friend void someFunction(MyClass<T>&);
};
```

1.4.2 Friendship to a Template Function

More commonly, you want a template function to be a friend to a template class. This allows each instantiation of the function template to access the corresponding instantiation of the class template.To achieve this, you need to forward declare the function template and then declare it as a friend inside your class template. The tricky part is that the syntax for declaring a template function as a friend inside a template class can vary based on what you're trying to achieve:

How to forward declare:

```cpp
    template<typename T>
    class myclass;

    template<typename T>
    void foo(myclass<T>&);


    template<typename T>
    class myclass {

    public:
        friend void foo <T>(const myclass<T>& obj);
    };

template <typename T>
void foo(myclass<T>& obj) {
    // Define
}
```

Different types:

- More general form used when you want the friendship to apply to all instantiations of the function template

```
template <typename T>
class MyClass {
    friend void someFunction<>(MyClass<T>&); // Specific instantiation
};
```

- All instantiations of the function template are friends:

```
template <typename T>
class MyClass {
    template <typename U>
    friend void someFunction(MyClass<U>&); // All instantiations
};
```

- This form ties the friendship to the specific template instantiation of both MyClass and someFunction using the same template argument T. The function template that takes the same template parameters:

```
template <typename T>
class MyClass {
    friend void someFunction<T>(MyClass<T>&); // Matched instantiation
};
```

1.5 Function Template Specialization

Concept 1: **Template specialization** allows you to define a different implementation for a particular data type.
□

```
template <typename T>
T min(T x, T y) {
    return (x < y) ? x : y;
}

template <>
const char* min(const char* x, const char* y) {
    return (strcmp(x, y) < 0) ? x : y;
}
```

1.6 Class/Struct Template Specialization

```
template<typename T>
struct foo {
    T x = 20;
};

template<>
struct foo<char> {
    char x = 'z';
};
```

1.7 Template Parameters

Concept 2: Templates can have more than one parameter, including non-type parameters.
□

```
template <typename T, int size>
class FixedArray {
    private:
    T arr[size];
    // implementation
};
```

1.8 Trailing return type

In traditional C++, the return type of a function is declared at the beginning of the function declaration. However, C++11 introduced a new syntax that allows the return type to be specified after the parameter list, using auto at the beginning and -> Type after the parameter list.

1.8.1 Syntax

```
auto functionName(parameters) -> returnType {
    // function body
}
```

1.8.2 Example

```
auto foo(int a, int b) -> int {
    return a + b;
}
```

1.9 decltype

Concept 3: **decltype** is a keyword in C++ introduced in C++11, which stands for "declared type". It is used to query the type of an expression without actually evaluating that expression. This can be particularly useful in template programming and type deduction, where the type of an expression might not be known until compile time.
□

1.9.1 Syntax

```
decltype(expression) variable_name;
```

Here, **variable_name** will have the same type as the type of **expression**. It's important to note that **expression** is not evaluated; **decltype** only deduces its type.

1.9.2 Example

```
int a = 5;
decltype(a) b = 5;

cout << typeid(b).name() << endl; // Output: i
```

1.10 Template functions with mixed types (Trailing return type)

Concept 4: To address the challenge of determining the return type for a template function that accepts two different types, we can utilize a strategy involving **auto** and a **trailing return type** with **decltype**. This approach effectively resolves the ambiguity of the return type in such template functions.
□

```
template<typename T, typename U>
auto add(T t, U u) -> decltype(t + u) {
    return t + u;
}
```

1.11 Template functions with mixed types (Deduced return type)

Alternatively, C++14 introduced the concept **deduced return type**. Which provides a simpler way to handle the situation described above

```
template<typename T, typename U>
decltype(auto) foo(T a, U b)  {
    return a + b;
}
```

When initializer lists are required

Using initialization lists to initialize data members in a constructor can be convenient if you don't need to do any error-checking on the constructor arguments. There are also several instances in C++ where the use of an initializer list to initialize a data member is actually required:

- Data members that are const but not static must be initialized using an initialization list.
- Data members that are references must be initialized using an initialization list.
- An initialization list can be used to explicitly call a constructor that takes arguments for a data member that is an object of another class (see the employee constructor example above).
- In a derived class constructor, an initialization list can be used to explicitly call a base class constructor that takes arguments.

Inheritence and Subtype Polymorphism

3.1 OOP Main Concepts

An **object** is a software bundle of related state (data members or properties) and behavior (member functions or methods). Software objects are often used to model the real-world objects that you find in everyday life.

A **class** is a blueprint or prototype from which objects are created. A class is an abstract definition that is made concrete at run-time when objects based upon the class are created.

Encapsulation, also known as data hiding, is the act of concealing the functionality of a class so that the internal operations are hidden, and irrelevant, to the programmer. With correct encapsulation, the developer does not need to understand how the class actually operates in order to communicate with it via its publicly available member functions and data members, known as its public interface. Encapsulation is essential to creating maintainable object-oriented programs. When the interaction with an object uses only the publicly available interface of member functions and properties, the class of the object becomes a correctly isolated unit. This unit can then be replaced independently to fix bugs, to change internal behavior or to improve functionality or performance. Encapsulation also promotes data integrity by allowing public "set" member functions to validate new values that are to be assigned to private data members.

Message passing, also known as interfacing, describes the communication between objects using their public interfaces. The primary way of passing a message to an object in C++ is to call a member function for that object.

Abstraction is the process of representing simplified versions of real-world objects in your classes and objects. A car class does not describe every possible detail of a car, only the relevant parts for the system being developed. Modeling software around real-world objects can vastly reduce the time required to understand a solution and be able to develop and maintain it.

3.2 Object Relationships

Objects can work together in many ways within a system. In some situations, classes and objects can be tightly coupled together to provide more complex functionality. This "has-a" relationship is known as composition. For example, modeling a car might involve creating individual classes such as wheel, engine, and transmission. The car class could then contain objects of these classes as data members, since a car "has" an engine, wheels, and a transmission. The internal workings of each class are not important due to encapsulation as the communication between the objects is still via passing messages to their public interfaces.

Other types of relationships may be modeled. A class may simply "use" an object of another class (perhaps creating the object as a local variable in one of its member functions). A class may also "know" about an object of another class without owning it (in C++, this association relationship might be modeled using a pointer or reference to the object).

Inheritance is an object-oriented programming concept used to model an "is-a" relationship between two classes. It allows one class (the derived class or subclass) to be based upon another (the base class or superclass) and inherit all of its functionality automatically. Additional code may then be added to create a more specialized version of the base class.

3.3 Ineritance

A **derived class** is more specific than its base class and represents a smaller group of objects.

A **direct base class** is the base class from which a derived class explicitly inherits. An indirect base class is inherited from two or more levels up the class hierarchy.

In the case of single inheritance, a class is derived from one base class. C++ also supports multiple inheritance, in which a derived class inherits from multiple (possibly unrelated) classes. Single inheritance is straightforward. Multiple inheritance can be complex and error prone.

Single-inheritance relationships form tree-like hierarchical structures - a base class exists in a hierarchical relationship with its derived classes.

C++ offers three kinds of inheritance - public, protected, and private. public inheritance in C++ is used to model "is a" relationships. Every object of a derived class is also an object of that derived class's base class. However, base-class objects are not objects of their derived classes. For example, all car objects are also vehicle objects, but not all vehicle objects are car objects.

With public inheritance, a derived class may

- add new data members
- add new member functions
- override member functions defined in the base class

private and protected inheritance do not model "is-a" relationships and are not used as frequently.

3.4 Inheritance and Member Access

Base class modifier	public Inheritance	protected Inheritance	private Inheritance
public	public	protected	Hidden
protected	protected	protected	Hidden
private	Hidden	Hidden	Hidden

A base class's public members are accessible anywhere that the program has a "handle" to an object (an object name or a pointer or reference to an object) of the base class or to an object of one of that base class's derived classes. Derived class member functions can access public base class data members directly.

A base class's private members are "hidden" - they are accessible only within the definition of that base class or from a friend of that class. A derived class cannot access the private members of its base class directly; allowing this would violate the encapsulation of the base class. A derived class can only access private base-class members through non-private member functions defined in the base class and inherited by the derived class.

A base class's protected members have an intermediate level of protection between public and private access. A base class's protected members can be directly accessed by member functions of that base class, by a friend of that base class, by member functions of a class derived from that base class, and by a friend of a class derived from that base class.

The use of protected data members allows for a slight increase in performance, because we avoid incurring the overhead of a call to a "set" or "get" member function. Unfortunately, protected data members often yield two major problems. First, the derived class object does not have to use a "set" member function to change the value of the base class's protected data. A derived class object can easily assign an illegal value to a protected data member. Second, derived class member functions are more likely to depend on base class implementation details. Changes to the base class may require changes to some or all of the derived classes of that base class.

Declaring data members private, while providing non-private member functions to manipulate and perform validation checking on this data, enforces good software engineering. The programmer should be able to change the base class implementation freely, while still providing the same services to the derived class. The performance increases gained by using protected data members are often negligible compared to the optimizations that compilers can perform. It is appropriate to use the protected access modifier when a base class should provide a service (i.e., a member function) only to its derived classes and should not provide the service to other clients.

When a base class member function is inappropriate for a derived class, that member function can be redefined in the derived class with an appropriate implementation. This is called overriding the base class member function.

When a derived class member function overrides a base class member function, the base class member function can still be accessed from the derived class by preceding the base class member function name with the base class name and the scope resolution operator (::).

When an object of a derived class is created, the base class's constructor is called immediately (either explicitly or implicitly) to initialize the base class data members in the derived class object (before the derived class data members are initialized). Explicitly calling a base class constructor requires using the same special "member initialization list syntax" used with composition and const data members.

When a derived class object is destroyed, the destructors are called in the reverse order of the constructors - first the derived class destructor is called, then the base class destructor is called.

3.5 Inheritance Syntax

To declare a derived class:

```
1  // car is a derived class of vehicle.
2  class car : public vehicle
3  {
4      // Car data members and member functions
5  };
```

A constructor initialization list can be used to pass arguments from a derived class constructor to a base class constructor:

```
// Pass the string color to the base class vehicle constructor.
car::car(const string& color, int num_doors) : vehicle(color)
{
    this->num_doors = num_doors;
}
```

A derived class member function that overrides a base class member function can call the base class version of the function to do part of its work:

```
void car::print() const
{
    vehicle::print();   // Call the vehicle version of print()
    ↪ to print the car's color.
    cout << num_doors;
}
```

3.6 Upcasting and Downcasting

Upcasting is converting a derived class pointer (or reference) to a pointer (or reference) of the derived class's base class. In other words, upcasting allows us to treat a derived type as though it were its base type. It is always allowed for public inheritance, without an explicit type cast. This is a result of the "is-a" relationship between the base and derived classes. For example, if car is a class derived from vehicle, the following code is legal:

```
vehicle* vptr = new car();
```

The car object does not actually become a vehicle object as a result of this type cast (in a sense, it already is one). However, the vehicle pointer can only be used to access parts of the car object that are defined in the vehicle class. For example, you can only call member functions that are defined in the vehicle class. The car object is treated like any other vehicle, and its car-specific data members and member functions are unavailable.

```
vehicle* vptr = (vehicle*) new car();
vehicle* vptr = dynamic_cast<vehicle*>(new car()); // Using c++
↪ casting
vehicle* vptr = static_cast<vehicle*>(new car()); // Using c++
↪ casting
```

The opposite process, converting a base class pointer (or reference) to a derived class pointer (or reference) is called **downcasting**. Downcasting is not allowed without an explicit type cast. The reason for this restriction is that the "is-a" relationship is not always symmetric. A car is a vehicle, but a vehicle may or may not be a car. For example:

```
car c1;

vehicle* vptr = &c1;          // Upcast - no type cast required.

car* car_ptr = (car*) vptr;   // Downcast - type cast required.
```

The code shown above works, because the object pointed to by vptr actually is a car object. If it wasn't, the results could lead to an unsafe operation.

```
bus b1;                       // Assume bus is also a derived
↪ class of vehicle.

vehicle* vptr = &b1;          // Upcast - no type cast required.

car* car_ptr = (car*) vptr;   // Downcast - type cast required.
↪ Fails because vptr
                                // points to a bus, not a car.
```

C++ provides a special explicit cast called **dynamic_cast** that allows for safe downcasting. If the type cast fails, it will return nullptr rather than crashing your program:

```
car* carptr = dynamic_cast<car*>(vptr);
if (carptr != nullptr)
{
    // Type cast succeeded, vptr was pointing to a car object
    // Can now safely call car-specific member functions using
↪   carptr
}
```

3.7 More on Downcasting

Downcasting is the process of converting a base class pointer or reference to a derived class pointer or reference

- Downcasting is potentially unsafe, so it requires an explicit cast.
- dynamic_cast should be used for safe downcasting to check if the cast is valid at runtime.
- Downcasting typically requires polymorphic classes with virtual functions to enable dynamic_cast.

In C++, the dynamic_cast operator, used for safe downcasting, requires that the base class has at least one virtual function. This is because dynamic_cast relies on runtime type information (RTTI), which is only available for polymorphic classes.

- **RTTI Availability:**
  - RTTI is used to store type information at runtime, which dynamic_cast uses to determine the exact type of the object being cast.
  - RTTI is only generated by the compiler for polymorphic classes, which are classes that have at least one virtual function.
- **Polymorphic Behavior:** Virtual functions enable polymorphic behavior, allowing derived classes to override base class functions. This is the essence of polymorphism, which dynamic_cast utilizes to ensure safe downcasting.
- **Checking Actual Type:** The type information stored in RTTI allows dynamic_cast to check if the object being cast is indeed of the target derived type. If not, it returns nullptr (for pointers) or throws a std::bad_cast exception (for references).

3.7.1 What Happens Without Virtual Functions

- **No RTTI:** If the base class has no virtual functions, it is not polymorphic, and the compiler doesn't generate RTTI for it. Without RTTI, dynamic_cast cannot validate types at runtime.
- **Compile-Time Error:** Attempting to use dynamic_cast on a class without virtual functions will lead to a compile-time error indicating that the class type is not polymorphic.

3.7.2 Downcasting example

```cpp
#include <iostream>

class Base {
    public:
    virtual ~Base() = default; // Make the class polymorphic
};

class Derived1 : public Base {
    public:
    void func1() {
        std::cout << "Derived1 Function\n";
    }
};

class Derived2 : public Base {
    public:
    void func2() {
        std::cout << "Derived2 Function\n";
    }
};

int main() {
    // Case 1: Valid downcast, will not get nullptr
    Base* basePtr1 = new Derived1(); // Base pointer to Derived1
    Derived1* derived1Ptr = dynamic_cast<Derived1*>(basePtr1);
    if (derived1Ptr) {
        derived1Ptr->func1(); // This will execute correctly
    } else {
        std::cout << "Downcast to Derived1 failed\n";
    }

    // Case 2: Invalid downcast, will get nullptr
    Base* basePtr2 = new Derived2(); // Base pointer to Derived2
    Derived1* derived1PtrInvalid =
    ↪ dynamic_cast<Derived1*>(basePtr2);
    if (derived1PtrInvalid) {
        derived1PtrInvalid->func1(); // This will not execute
    } else {
        std::cout << "Downcast to Derived1 failed\n"; // This
    ↪ will be printed
    }

    // Clean up
    delete basePtr1;
    delete basePtr2;

    return 0;
}
```

Note:-

When a Base* points to a Base object and we attempt to downcast it to a derived type using dynamic_cast, the cast will fail and return nullptr. This is because the actual type of the object being pointed to is Base, not the derived type.

3.7.3 Base class pointer example

```
class base {

public:
    virtual void print() const {
        cout << "Base class" << endl;
    }

};


class derived : public base {
    void print() const override {
        cout << "Child class" << endl;
    }
};


int main(int argc, char* argv[]) {

    base* bptr = new derived();

    bptr->print(); // Child class
    bptr->base::print(); // Base class

    return EXIT_SUCCESS;
}
```

Note:-

Notice we are able to call the private method, this is because the virtual function mechanism directs the call to the most derived method.

3.8 Object Slicing

Object slicing occurs when a derived class object is assigned or copied to a base class object, causing the derived class's specific members to be "sliced off."

```
base b1 = base();
derived d1 = derived();

b1 = d1; // Slicing
d1 = b1; // Does not work
```

3.9 Multiple Inheritance

Multiple inheritance is a feature in C++ that allows a derived class to inherit from more than one base class.

3.9.1 Why Use Multiple Inheritance?

- **Combining Functionality:** When a derived class needs to combine the functionalities of multiple base classes.
- **Mixins:** Allows implementing mixins, which are small base classes that provide specific functionalities to derived classes.
- **Interface Implementation:** Multiple inheritance can also be used to implement interfaces (abstract base classes) in C++.

3.9.2 Example

```cpp
#include <iostream>

class Base1 {
public:
    void func1() {
        std::cout << "Base1 Function" << std::endl;
    }
};

class Base2 {
public:
    void func2() {
        std::cout << "Base2 Function" << std::endl;
    }
};

class Derived : public Base1, public Base2 {
    // Derived class inherits from both Base1 and Base2
public:
    void func3() {
        std::cout << "Derived Function" << std::endl;
    }
};

int main() {
    Derived d;
    d.func1(); // Inherited from Base1
    d.func2(); // Inherited from Base2
    d.func3(); // Own function

    return 0;
}
```

3.9.3 Issues with Multiple Inheritance

1. **Ambiguity:**
    - If two base classes have the same method or attribute name, calling it from the derived class can cause ambiguity.
    - This can be resolved using the scope resolution operator :: to specify which base class method to call.
2. **Diamond Problem:**
    - If two base classes inherit from the same class and a derived class inherits from both base classes, it leads to ambiguity in the derived class about which base class's implementation to use.
    - This can be addressed using virtual inheritance to ensure only one copy of the shared base class exists.

3.10 Virtual inheritance

Virtual inheritance in C++ is a mechanism to prevent multiple "copies" of a base class when using multiple inheritance. It ensures that the derived class has only one shared instance of the base class, thus preventing ambiguity and redundant base class objects.

3.10.1 The Diamond Problem

The diamond problem occurs when two classes inherit from the same base class and another class inherits from those two classes. This results in ambiguity and multiple copies of the shared base class.

```cpp
#include <iostream>

class Base {
    public:
    int data;
    Base() : data(0) {}
};

class Derived1 : public Base {};

class Derived2 : public Base {};

class FinalDerived : public Derived1, public Derived2 {};

int main() {
    FinalDerived obj;

    // Error: Ambiguity, which Base::data to access?
    // obj.data = 10;
    // Explicit resolution:
    obj.Derived1::data = 10;
    obj.Derived2::data = 20;

    std::cout << obj.Derived1::data << ", " <<
    ↪ obj.Derived2::data << std::endl;

    return 0;
}
```

There are two separate instances of the Base class, one inherited through Derived1 and one through Derived2 This leads to ambiguity and multiple instances.

3.10.2 Solution with Virtual Inheritance

Virtual inheritance addresses this problem by ensuring that the shared base class is inherited virtually. This makes the derived class have a single, shared instance of the base class.

```cpp
#include <iostream>
class Base {
public:
    int data;
    Base() : data(0) {}
};

class Derived1 : virtual public Base {};

class Derived2 : virtual public Base {};

class FinalDerived : public Derived1, public Derived2 {};

int main() {
    FinalDerived obj;

    // No ambiguity, only one instance of Base exists
    obj.data = 10;

    std::cout << obj.data << std::endl;

    return 0;
}
```

3.11 Subtype Polymorphism

The term binding means matching a function or member function call to a function or member function definition.

In C++, binding normally takes place when the program is compiled and linked. This is referred to as early binding or static binding.

In object-oriented programming, subtype polymorphism refers to the ability of objects belonging to different types to respond to member function calls of the same name, each one according to an appropriate type-specific behavior. The calling code does not have to know the exact type of the called object; which member function definition is called is determined at run-time (this is called late binding or dynamic binding).

In order for dynamic binding to take place in C++, several conditions must be met:

1. The call must be to a member function, not a standalone function. Function calls in C++ always use static binding.
2. The member function must have been declared using the keyword virtual. Calls to non-virtual member functions always use static binding.

3. The member function must be called through a pointer or reference to an object, not an object name. All calls to member functions (including those to virtual member functions) through object names use static binding.

With dynamic binding, C++ distinguishes between a static type and a dynamic type of a variable. The static type is determined at compile time. It's the type specified in the pointer declaration. For example, the static type of vptr is vehicle*. However, the dynamic type of the pointer is determined by the type of object to which it actually points: car* in this case. When a virtual member function is called using vptr, C++ resolves the dynamic type of vptr and ensures that the appropriate version of the member function is invoked, a process referred to as virtual dispatch.

Dynamic binding exacts a toll. Resolving the dynamic type of an object takes place at runtime and therefore incurs performance overhead. However, this penalty is negligible in most cases.

One of the most common runtime techniques for implementing virtual dispatch is a virtual member function table, or v-table. A v-table is simply an array of pointers to member functions. Each class that contains virtual member functions has a v-table. Each object that is an instance of a class with virtual member functions contains, as a hidden field, a pointer to the class's v-table. The compiler encodes a member function call as an offset into a v-table, and the appropriate v-table is used with that offset at runtime to access the correct member function.

3.12 Declaring Virtual Member Functions

Declaring Virtual Member Functions

```
1  virtual void print() const;
```

Note:-

A member function in a derived class that overrides a virtual member function in a base class is automatically virtual as well.

Destructors may also be virtual. You should make the destructor for your class virtual if it contains any virtual member functions.

In C++, when you overload a virtual function from a base class in a derived class, you do not necessarily need to mark the function in the derived class as virtual again for it to behave as a virtual function. It will still be virtual in any further derived classes. However, explicitly marking it as virtual in the derived class can improve code readability and make the class's design intentions clearer.

3.12.1 The override keyword

Using override: Instead of (or in addition to) marking functions as virtual in derived classes, C++11 introduced the override specifier. This ensures that the function is intended to override a virtual function in a base class. Using override helps catch errors at compile-time where the function signature does not match any virtual function in the base class, thus preventing unintended behavior.

```cpp
class Base {
    public:
    virtual void foo() { /* implementation */ }
};

class Derived : public Base {
    public:
    virtual void foo() override { /* new implementation */ }  // Using 'virtual' and 'override' for clarity
    void foo() override; // Better approach... no need for another virtual keyword
};
void Base::foo() {
    // Base class definition
}
void Derived::foo() {
    // Derived class definition
}
```

3.13 Abstract or Pure virtual Member Functions

An abstract member function is a member function that has a special prototype, but no definition. C++ refers to abstract member functions as pure virtual member functions. The prototype for a pure virtual member function ends with = 0, like this:

```cpp
virtual void earnings() const = 0;
```

Note:-

Since a pure virtual member function has no definition, you can't really call it. However, if a base class contains a pure virtual member function, a derived class is allowed to override the member function and provide a definition.

3.14 Abstract Classes

A class that contains one or more pure virtual member functions is called an abstract class (as opposed to a concrete class that provides definitions for all of its member functions).

You cannot create an object of an abstract class. However, an abstract class can be used as a base class for inheritance purposes. A class derived from an abstract class must provide definitions for any pure virtual member functions that it inherits, or it is also an abstract class.

You can also declare a pointer (or a reference) of an abstract class type. Such a pointer (or reference) would typically be used to point to a derived class object.

3.15 Interface Inheritance

Interface inheritance allows a derived class to inherit a base class's data type (which can be useful for subtype polymorphism) without actually inheriting any of the base class's implementation (member function definitions, etc.).

An interface can be defined in C++ as an abstract class that contains only pure virtual member functions and symbolic constants (public data members that are static and const).

<regex.h> Pattern Matching and String Validation

4.1 regcomp

Concept 5: Compiles a regular expression into a format that the **regexec()** function can use to perform pattern matching.
□

4.1.1 Signature

```
1  int regcomp(regex_t *preg, const char *regex, int cflags)
```

- **preg:** A pointer to a regex_t structure that will store the compiled regular expression.
- **regex:** The regular expression to compile.
- **cflags:** Compilation flags that modify the behavior of the compilation. Common flags include REG_EXTENDED (use extended regular expression syntax), REG_ICASE (ignore case in match), REG_NOSUB (don't report the match), and REG_NEWLINE (newline-sensitive matching).

4.1.2 Return value

Upon successful completion, the regcomp() function shall return 0. Otherwise, it shall return an integer value indicating an error as described in <regex.h>, and the content of preg is undefined.

4.1.3 Return errors

- **REG_NOMATCH**: regexec() failed to match.
- **REG_BADPAT**: Invalid regular expression.
- **REG_ECOLLATE**: Invalid collating element referenced.
- **REG_ECTYPE**: Invalid character class type referenced.
- **REG_EESCAPE**: Trailing '\' in pattern.
- **REG_ESUBREG**: Number in "\digit" invalid or in error.
- **REG_EBRACK**: "[]" imbalance.
- **REG_EPAREN**: "\(\)" or "()" imbalance.
- **REG_EBRACE**: "\{\}" imbalance.

- **REG_BADBR**: Content of "\{\}" invalid: not a number, number too large, more than two numbers, first larger than second.
- **REG_ERANGE**: Invalid endpoint in range expression.
- **REG_ESPACE**: Out of memory.
- **REG_BADRPT**: '?', '*', or '+' not preceded by valid regular expression.

4.1.4 Flags

- **REG_EXTENDED:** Enables the use of Extended Regular Expressions (ERE) rather than Basic Regular Expressions (BRE). EREs allow a broader set of regex features, such as more flexible quantifiers and additional metacharacters without needing to escape them.
- **REG_ICASE:** Makes the pattern matching case-insensitive. This means that characters will match regardless of being in upper or lower case. For example, using REG_ICASE, the pattern "a" would match both 'a' and 'A'.
- **REG_NOSUB:** Disables the reporting of the position of matches. This flag is useful when you only need to know if a match occurred but not where it occurred. It can lead to performance improvements because the regex engine does not need to track and store the match positions.
- **REG_NEWLINE:** Alters the handling of newline characters in the text. Specifically, it:
  - Prevents the match from spanning multiple lines. The ^and metacharacters will match the start and end of the input string but not the start or end of a line within the string.
  - Causes the dot . metacharacter to stop matching at a newline, which it normally would match.
  - Treats newline characters in the input as a boundary that cannot be crossed by the quantifiers *, +, ?, and n unless explicitly included in a character class.
- **REG_NOTBOL and REG_NOTEOL:**
  - **REG_NOTBOL (Not Beginning Of Line):** Tells the regex engine that the beginning of the provided string should not be treated as the beginning of the line. This affects how the $\wedge$ anchor (which normally matches the start of the string) behaves. Use this if the string is a substring that does not start at the beginning of a new line.
  - **REG_NOTEOL (Not End Of Line):** Indicates that the end of the provided string should not be treated as the end of the line. This affects how the $ anchor (which normally matches the end of the string) behaves. Use this if the string is a substring that does not end at the end of a line.

4.2 Regexec

Concept 6: After compiling a regular expression, we can use regexec to match against strings.
□

4.2.1 Signature

```
int regexec(const regex\_t *preg, const char *string, size\_t
↪  nmatch, regmatch\_t pmatch[], int eflags)
```

- **preg:** The compiled regular expression.
- **string:** The string to match against the regular expression.
- **nmatch:** The maximum number of matches and submatches to find.
- **pmatch:** An array of regmatch_t structures that will hold the offsets of matches and submatches.
- **eflags:** Execution flags that modify the behavior of the match. A common flag is REG_NOTBOL which indicates that the beginning of the specified string is not the beginning of a line.

4.2.2 Return value

Upon successful completion, the regexec() function shall return 0. Otherwise, it shall return REG_NOMATCH to indicate no match.

4.3 Regerror

Concept 7: This function translates error codes from regcomp() and regexec() into human-readable messages.
□

4.3.1 Signature

```
size_t regerror(int errcode, const regex\_t *preg, char *errbuf,
↪  size\_t errbuf\_size)
```

- **errcode:** The error code returned by regcomp() or regexec().
- **preg:** The compiled regular expression (if the error is related to regexec()).
- **errbuf:** The buffer where the error message will be stored.
- **errbuf_size:** The size of the buffer.

4.3.2 Return value

Upon successful completion, the regerror() function shall return the number of bytes needed to hold the entire generated string, including the null termination. If the return value is greater than errbuf$_size, the string returned in the buffer pointed to by errbuf has been truncated.$

4.4 Regfree

Concept 8: Frees the memory allocated to the compiled regular expression.
□

4.4.1 Signature

```
1  void regfree(regex\_t *preg)
```

- **preg:** The compiled regular expression to free.

4.5 regmatch_t and pmatch

4.5.1 regmatch_t

regmatch_t is a structure used to describe a single match (or submatch) found by regexec(). It contains at least the following two fields:

- **rm_eo:** This is the end offset of the match, which is one more than the index of the last character of the match. In other words, rm_eo - rm_so gives the length of the match.
- **rm_so:** This is the start offset of the match, relative to the beginning of the string passed to regexec(). If the match is successful, rm_so will be the index of the first character of the match.

4.5.2 pmatch array

When you call regexec(), you can pass it an array of regmatch_t structures as the pmatch argument. This array is where regexec() will store information about the matches (and submatches) it finds. The size of this array (nmatch) determines how many matches regexec() will look for and fill in. The zeroth element of this array corresponds to the entire pattern's match, and the subsequent elements correspond to parenthesized subexpressions (submatches) within the regular expression, in the order they appear.

4.6 Regex Example

```c
    regex_t regex;
    int reti;
    char msgbuf[100];
    regmatch_t pmatch[1]; // Array to store the match positions
    const char* search = "abc";

    // Compile regular expression
    reti = regcomp(&regex, "^a[[:alnum:]]", REG_EXTENDED);
    if (reti) {
        fprintf(stderr, "Could not compile regex\n");
        exit(EXIT_FAILURE);
    }

    // Execute regular expression
    // Note: Changed the third argument to 1 to indicate we want to capture up to 1 match
    // and the fourth argument to pmatch to store the match position.
    reti = regexec(&regex, search, 1, pmatch, 0);
    if (!reti) {
        printf("Match\n");
        // If you want to use the match information, you can do so here.
        // For example, to print the start and end positions of the match:
        printf("Match at position %d to %d\n", (int)pmatch[0].rm_so, (int)pmatch[0].rm_eo - 1);
    }
    else if (reti == REG_NOMATCH) {
        printf("No match\n");
    }
    else {
        regerror(reti, &regex, msgbuf, sizeof(msgbuf));
        fprintf(stderr, "Regex match failed: %s\n", msgbuf);
        exit(EXIT_FAILURE);
    }regex_t regex;
int reti;
char msgbuf[100];
regmatch_t pmatch[1]; // Array to store the match positions

// Compile regular expression
reti = regcomp(&regex, "^a[[:alnum:]]", REG_EXTENDED);
if (reti) {
    fprintf(stderr, "Could not compile regex\n");
    exit(EXIT_FAILURE);
}
```

```
// Execute regular expression
// Note: Changed the third argument to 1 to indicate we want to capture up to 1 match
// and the fourth argument to pmatch to store the match position.
reti = regexec(&regex, "abc", 1, pmatch, 0);
if (!reti) {
    printf("Match\n");
    // If you want to use the match information, you can do so here.
    // For example, to print the start and end positions of the match:
    printf("Match at position %d to %d\n", (int)pmatch[0].rm_so, (int)pmatch[0].rm_eo - 1);
}
else if (reti == REG_NOMATCH) {
    printf("No match\n");
}
else {
    regerror(reti, &regex, msgbuf, sizeof(msgbuf));
    fprintf(stderr, "Regex match failed: %s\n", msgbuf);
    exit(EXIT_FAILURE);
}

// Free the compiled regular expression
regfree(&regex);

    for (int i=(int)pmatch[0].rm_so; i<=(int)pmatch[0].rm_eo; ++i) {
        cout << search[i];
    }
    cout << endl;

    regfree(&regex);
```