C++
Function Pointers & Templates

CO650 Advanced Programming

Topics

- Pointers to Functions
- Pointers to Members
- Function Templates
- Class Templates

Functions as Pointers

- A function is a constant pointer.
- We can display the functions address by simply omitting the call operator ().

```
cout << FunctionName << endl;
```

- To invoke the function we precede its name with **()** the call operator. At this point the function is dereference and executes.
- It is possible to define a pointer to a function.
- The pointer can only be assigned functions that match the type used in the pointer declaration.
- Through this pointer we can invoke the function.

Function's Type

- A functions type is determined by its return type and number and type of its parameters.
- To declare a pointer to a function

```
type (*PointerName)(parameter list);
```

- Note the brackets around the pointer name
- . If these were omitted we would be declaring a function that returns a pointer.
- For a function pointer ptr that has no return type and no parameters the declaration is

```
void (*ptr)();
```

- For a ptr to a function that returns an int type and takes two int parameters

```
int (*ptr)(int, int);
```

Function Pointers

- The function pointer can be assigned any function that conforms to the pointer type (function type).

```
PointerName = FunctionName;
```

- At this stage we are assigning an address to the function, Not invoking it (brackets are omitted and no arguments are passed).

```
ptr = AddNumbers;
```

- To invoke the function whose address is held in the pointer we include its name, brackets and arguments.

```
int answer = ptr(5,7);
```

Function Pointer Example

```
int Add(int a,int b){
        return a + b;
}

int main(){
        int answer;
        int (*ptr1)(int, int);
        ptr1 = Add;
        answer = ptr1(6,3);
        cout << "Answer is : " << answer <<endl;

        // or using typedef
        typedef int (*TP)(int, int);
        TP ptr2;
        ptr2 = Add;
        answer = ptr2(7,9);
        cout << "Answer is : " << answer <<endl;
        return 0;
}
```

typeDef

- The function type can become unwieldy
- Using typeDef allows us to create an alias for the type

```
typedef  aliasName;
```

- We can use the alias any where we would use the original type.
- A typedef for a function type

```
typedef  type (*aliasName)(Parameter/s);
```

- Here we create an alias named TP and use it within the declaration of a pointer ptr.

```
typedef int (*TP)(int, int);
TP ptr;
```

Pointers To Member Functions

- A pointer to a function can’t point to a member function, even if the function types match.
- A function type for a member function has
  1. The same return type
  2. Same number and type of parameters
  3. **The class to which the function belongs**

```
type (ClassName::*PointerName)(parameter/s);
```

Member Function Pointer Example

```
class MyClass{
public:
        int Add(int a,int b){
                return a + b;
        }
};


int main(){
        MyClass obj;
        int (MyClass::*ptr)(int,int);
        ptr = &MyClass::Add;
        int answer = (obj.*ptr)(3,9);
        cout << answer << endl;
        return 0;
}
```

Invoking Pointers To Member Functions

- The pointer is invoked through an object of the class type
- For static objects and references

```
(object.*PointerName)(argument/s);
```

- For pointers to objects

```
(object->*PointerName)(argument/s);
```

- Parentheses required as the call operator () has precedence over the pointer to member operator.
- Assign the member function address to the pointer before invoking it.

```
PointerName = &ClassName::MemberFunctionName;
```

Function Templates

- Functions that can operate with generic types
- Avoids duplicating code
- Implemented through template parameters
- Parameters that accept a type as an argument

```
template <class identifier> functionDeclaration;
```

or

```
template <typename identifier> functionDeclaration;
```

- Both methods are acceptable

Function Templates

- To use the template function

```
functionName <type>(parameter/s);
```

- The compiler uses the template to generates a function.
- It replaces instances of the type identifier with the actual template parameter.

Function Templates Example

- Return the greater of two numbers

```
 template <typename T>
T GetMax(T a, T b){
        T result;
        result = (a>b)? a : b;
        return result;
}


int main(){
        cout << GetMax<int>(14,8) << endl;
        cout << GetMax<float>(1.5,2.75) << endl;
        cout << GetMax<char>('M','C') << endl;
        return 0;
}
```

Function Templates

- If the compiler can determine the types of the parameter being passed there is no need to explicitly specify it when invoking the template function.
- In this example a & b are both declared as integers.

```
int a = 6, b = 9;
cout << GetMax(a,b) << endl;
```

- In fact the literals will be interpreted as integers too

```
cout << GetMax(14,8) << endl;
```

- Mixing the parameter types will cause a compiler error as only one type has been included within the function.

Function Templates

- More than one type can be specified within the template function.

```
template <class identifier1, class identifier2> functionDeclaration;
```

- And passed to the function when invoked.

```
functionName <type1,type2>(parameter/s);
```

- Warning: ensure the return type is assigned a valid value and is not out of range.

Function Templates Example

- Return the smallest of two numbers

```
template <typename T, typename U>T GetMin(T a, U b){
        T result;
        result = (a<b)? a : b;
        return result;
}

int main(){
    int i = 10;
    long l = 99;
    cout << GetMin<int,long>(i,l) << endl;
    return 0;
}
```

Class Templates

- Classes can use the same template mechanism.
- Both member variables and functions can take advantage.

```
template <class T>
class className{


};
```

Class Templates Example

```
template<class T>
class Vector2D{
private:
    T coordinate[2];
public:
    Vector2D(T x, T y){
            coordinate[0] = x;
            coordinate[1] = y;
    }
    void Display(){
            cout << "x : " << coordinate[0] << " y: " << coordinate[1] << endl;
    }
};

int main(){
    Vector2D<int> v(3,4);
    v.Display();
}
```

Class Templates

- If the member function is not inline we must precede the definition with template<..>

```
template <class T>
T className<T>::functionName(){
}
```

- The T after the namespace indicates the function's template parameter is also the class template parameter.
- All member function must include the prefix and <T> after the namespace , even if they don't utilise the type.
- In the next example the constructor definition has been moved out of the class declaration. Note: the constructor does not have a return type.

Class Templates Example

```
template<class T>
class Vector2D{
private:
    T coordinate[2];
public:
    Vector2D(T x, T y);
    void Display(){
            cout << "x : " << coordinate[0] << " y: " << coordinate[1] << endl;
    }
};

template<class T>
Vector2D<T>::Vector2D(T x, T y){
                        coordinate[0] = x;
                        coordinate[1] = y;
}
```

Class Templates

IMPORTANT :

- When creating your template class the class definition, all function definitions must be placed within the header file (.h). If you place the function definitions in the cpp file Visual Studio will generate a linker error.

Summary

- Template provide a flexible mechanism for writing code that can handle multiple types. Thus avoiding the inherent problems associated with duplication of code.
- Pointers to member functions require an object through which they are invoked and are therefore less useful than the Delegates mechanism found in .NET