# Epilogue

# WHAT IS STD::BIND?

- The stl::bind is a function adapter.
- You can adapt the signature, parameters, and attributes of a function.

# WHAT CAN YOU DO?

Adapt an existing function:

- Set constant values
- Provide input values
- Change the number of parameters
- Update the type of parameters
- Change the position of parameters

# STL::BIND DETAILS

The method stl::bind returns an object that references an existing function and some function parameters, values for missing parameters, and a call operator to invoke the referenced function through the stl::bind function pointer.

Several adapters are deprecated with stl::bind:

- ptr_fun
- mem_fun
- bind1st
- bind2nd

# STL::BIND IMPLEMENTATIONS

- include *functional* header file for std::bind
- Use namespace *std::placeholders* for placeholders, such as _1, _2, and _3.
- stl::bind is a function template.  Here is the signature.

```
bind(function  pointer, parameters…)
```

# USING STL::BIND

```
#include <iostream>
#include <functional>
using namespace std::placeholders;
using namespace std;

long doincrement(int start,int numof, int increment) {
     auto result=start + (numof*increment);
     return result;
}
int main()
{
    auto inc = bind(doincrement, _1, _2, 5);
    cout << inc(1, 5) << endl;
}
```

# PARAMETER PLACEHOLDERS

- You can bind to function parameters using placeholders: _1, _2, and so on
- Binds function parameters to placeholders
- Reorder placeholders changes the sequence of parameters
- Placeholder range is [ _1, . . . , _N ]. _N is implementation specific.

# ALGORITHM / COLLECTIONS

- You can use bind within an algorithm as the function object.
- With the for_each, each element of the collection is bound as a parameter to the function object.

```
void doincrement(int start,
         int numof, int increment){

     auto result=start +
          (numof*increment);

     cout << result << endl;

}

int main(){

     std::list<int> mylist =
          { 3, 2, 1 };

     for_each(mylist.begin(),
          mylist.end(),
          bind(doincrement,
          _1, 5, 1));

     return 0;

}
```

# FUNCTION TRY BLOCK

- Catch exceptions from initialization lists, destructors, and so on.
- Provides an opportunity to handle an exception in an initialization list in an orderly manner.

```
class Foo {
private:
  Bar *obj;
public:
  Foo() try : obj(new Bar()) { }
       catch(...) { }
};
```

# PREDEFINED MACROS

These predefined macros are not new but nonetheless useful for diagnostics, debugging, and logging.

- __COUNTER__
- __DATE__
- __TIME__
- __FILE__
- __LINE__
- __func__

```
void example() {
    printf("%d\n", __COUNTER__);
    printf("%s\n", __func__);
    printf("%s\n", __FILE__);
    printf("%s\n", __DATE__);
    printf("%d\n", __LINE__);
    printf("%d\n", __COUNTER__);
}

int main(){
    example();
    printf("\n%s\n", __func__);
    printf("%d\n", __COUNTER__);
    return 0;
}
```

# REF QUALIFIERS

Overload a method based on lvalue or rvalue object type.

```
class ClassA {
    public:
    void FuncA() & {
        int a = 5;
        ++a;
    }
    void FuncA() && {
        int a = 5, b = 10;
        a /= b;
    }
};
```

# COMMA OPERATOR

Some surprising operators, such as the comma operator, can be overloaded.  This can lead to some imaginative solutions.

Other available operators:

- operator &
- operator &&
- operator ||
- operator->

# COMMA OPERATOR EXAMPLE CODE

```
class IntWrapper {
public:
IntWrapper(int value) : _value(value) {}
IntWrapper operator ,(const Int & rhs) {
   return Int(_value + rhs.value);
}
IntWrapper get() const{
    return _int;
}
private:
    int _value;
};
```

```
int main() {
    IntWrapper obj1(10),
               obj2(20),
               obj3(30);
    obj1 = (obj1, obj2, obj3);
    cout << obj1.get();
    return 0;
}
```