
4.4. Callable Objects

At different places, the C++ standard library uses the term *callable object*, which means objects that somehow can be used to call some functionality:

- A function, where additional *args* are passed to as arguments
- A pointer to a member function, which is called for the object passed as the first additional argument (must be reference or pointer) and gets the remaining arguments as member function parameters
- A function object (operator `()` for a passed object), where additional *args* are passed as arguments
- A lambda (see Section 3.1.10, page 28), which strictly speaking is a kind of function object For example:

Click here to view code image

```
void func (int x, int y);

auto l = [] (int x, int y) {
               ...
         };

class C {
  public:
    void operator () (int x, int y) const;
    void memfunc (int x, int y) const;
};

int main()
{
    C c;
    std::shared_ptr<C> sp(new C);

    // bind() uses callable objects to bind arguments:
    std::bind(func,77,33)();              // calls: func(77,33)
    std::bind(l,77,33)();                 // calls: l(77,33)
    std::bind(C(),77,33)();               // calls: C::operator()(77,33)
    std::bind(&C::memfunc,c,77,33)();     // calls: c.memfunc(77,33)
    std::bind(&C::memfunc,sp,77,33)();    // calls: sp->memfunc(77,33)

    // async() uses callable objects to start (background) tasks:
    std::async(func,42,77);               // calls: func(42,77)
    std::async(l,42,77);                  // calls: l(42,77)
    std::async(c,42,77);                  // calls: c.operator()(42,77)
    std::async(&C::memfunc,&c,42,77);     // calls: c.memfunc(42,77)
    std::async(&C::memfunc,sp,42,77);     // calls: sp->memfunc(42,77)
}
```

As you can see, even smart pointers (see Section 5.2, page 76) can be used to pass an object a member function is called for. See Section 10.2.2, page 487, for details about `std::bind()` and Section 18.1, page 946, for details about `std::async()` .

To declare *callable objects*, in general class `std::function<>` can be used (see Section 5.4.4, page 133).