What Every Professional C++ Programmer Needs to Know—Pared to Its Essentials So It Can Be Efficiently and Accurately Absorbed

C++ COMMON KNOWLEDGE

ESSENTIAL INTERMEDIATE PROGRAMMING

STEPHEN C. DEWHURST

C++ Common Knowledge

(continued inside the back cover)

Item 17 Dealing with Function and Array Declarators

The main confusion with pointer to function and pointer to array declarations arises because the function and array modifiers have higher precedence than the pointer modifier, so parentheses are often required.

```
int *f1(); // function that returns int *
int (*fp1)(); // ptr to function that returns int
```

The same problem obtains with the high-precedence array modifier:

```
const int N = 12;
int *a1[N]; // array of N int *
int (*ap1)[N]; // ptr to array of N ints
```

Of course, once one can have a pointer to a function or to an array, one can have a pointer to such a pointer:

```
int (**ap2)[N]; // ptr to ptr to array of N ints
int *(*ap3)[N]; // ptr to array of N int *
int (**const fp2)() = 0; // const ptr to ptr to func
int *(*fp3)(); // ptr to func that returns int *
```

Note that both the argument and return types contribute to the type of a function or function pointer.

```
char *(*fp4)(int,int);
char *(*fp5)(short,short) = 0;
fp4 = fp5; // error! type mismatch
```

Things can become immeasurably more complex when function and array modifiers appear in the same declaration. Consider the following common, incorrect attempt to declare an array of function pointers:

```
int (*)()afp1[N]; // syntax error!
```

In the (erroneous) declaration above, the appearance of the function modifier () signaled the end of the declaration, and the appended name afp1 signaled the start of a syntax error. It's analogous to writing an array declaration

```
int[N] a2; // syntax error!
```

that works just fine in Java but is not legal C++. The correct declaration of an array of function pointers puts the name being declared in the same location that it appears in a simple pointer to function. Then we say we want an array of those things:

```
int (*afp2[N])(); // array of N ptr to func that returns int
```

Things are starting to get unwieldy here, so it's time to reach for typedef.

```
typedef int (*FP)(); // ptr to func that returns int
FP afp3[N]; // array of N FP, same type as afp2
```

The use of typedef to simplify the syntax of complex declarations is a sign of caring for those poor maintainers who come after you. Using typedef, even the declaration of the standard set_new_handler function becomes simple:

```
typedef void (*new_handler)();
new_handler set_new_handler( new_handler );
```

So, a new_handler (see *Function Pointers* [14, 49]) is a pointer to a function that takes no argument and returns void, and set_new_handler is a function that takes a new_handler as an argument and returns a new_handler as a result. Simple. If you try it without typedef, your popularity with those who maintain your code will plummet:

```
void (*set_new_handler(void (*)()))(); // correct, but evil
```

It's also possible to declare a reference to a function.

```
int aFunc( double ); // func
int (&rFunc)(double) = aFunc; // ref to func
```

References to functions are rarely used and fill pretty much the same niche as constant pointers to functions:

```
int (*const pFunc)(double) = aFunc; // const ptr to func
```

References to arrays do provide some additional capability not provided by pointers to arrays and are discussed in *Array Formal Arguments* [6, 17].

Item 18 | Function Objects

Often you'll need something that behaves like a function pointer, but function pointers tend to be unwieldy, dangerous, and (let's admit it) passé. Often the best approach is to use a function object instead of a function pointer.

A function object, like a smart pointer (see *Smart Pointers* [42, 145]) is an ordinary class object. Whereas a smart pointer type overloads the `->` and `*` (and possibly `->*`) operators to mimic a "pointer on steroids," a function object type overloads the function call operator, `()`, to create a "function pointer on steroids." Consider the following function object that computes the next element in the well-known Fibonacci series (1, 1, 2, 3, 5, 8, 13, …) with each call:

```
class Fib {
  public:
    Fib() : a0_(1), a1_(1) {}
    int operator ();
  private:
    int a0_, a1_;
};
int Fib::operator () {
    int temp = a0_;
    a0_ = a1_;
    a1_ = temp + a0_;
    return temp;
}
```

A function object is just a regular class object, but you can call its `operator ()` member (or members, if there is more than one) with standard function call syntax.

```
Fib fib;
//...
cout << "next two in series: " << fib()
     << ' ' << fib() << endl;
```

The syntax `fib()` is recognized by the compiler as a member function call to the `operator ()` member of `fib`, identical in meaning to `fib.operator()` but presumably easier on the eye. The advantage in this case of using a function object in preference to a function or a pointer to a function is that the state required to compute the next number in the Fibonacci series is stored within the `Fib` object itself. A function implementation would have to resort to global or local static variables or some other base trickery to retain state between invocations of the function, or the information would have to be passed to the function explicitly. Also note that unlike a function that uses static data, we can have multiple, simultaneous `Fib` objects whose calculations do not interfere with each other.

```
int fibonacci () {
    static int a0 = 0, a1 = 1; // problematic...
    int temp = a0;
    a0 = a1;
    a1 = temp + a0;
    return temp;
}
```

It's also possible and common to create the effect of a virtual function pointer by creating a function object hierarchy with a virtual `operator ()`. Consider a numeric integration facility that calculates an approximation of the area under a curve, as shown in Figure 5.

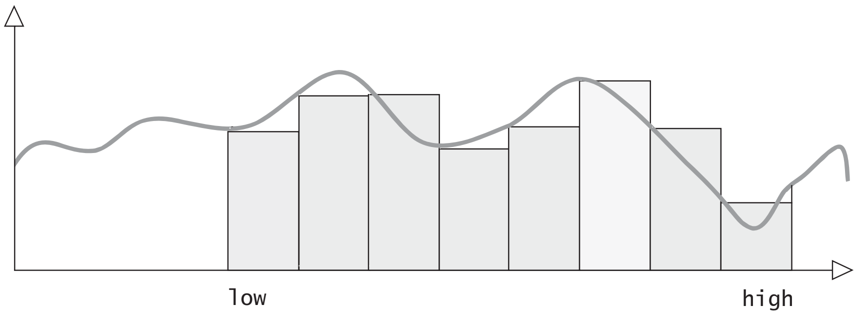

Figure 5 | Numeric integration by summing areas of rectangles (simplified)

Our integration function will iteratively call a function for values between `low` and `high` in order to approximate the area under the curve as a sum of the areas of rectangles (or some similar mechanism).

```
typedef double (*F)( double );
double integrate( F f, double low, double high ) {
    const int numsteps = 8;
    double step = (high-low)/numSteps;
    double area = 0.0;
    while( low < high ) {
        area += f( low ) * step;
        low += step;
    }
    return area;
}
```

In this version, we pass a pointer to the function over which we want to integrate.

```
double aFunc( double x ) { ... }
//...
double area = integrate( aFunc, 0.0, 2.71828 );
```

This works, but it's inflexible because it uses a function pointer to indicate the function to be integrated; it can't handle functions that require state or pointers to member functions. An alternative is to create a function object hierarchy. The base of the hierarchy is a simple interface class that declares a pure virtual `operator ()`.

```
class Func {
  public:
    virtual ~Func();
    virtual double operator ()( double ) = 0;
};
double integrate( Func &f, double low, double high );
```

Now `integrate` is capable of integrating any type of function object that is-a `Func` (see *Polymorphism* [2, 3]). It's also interesting to note that the body of `integrate` does not have to change at all (though it does require recompilation), because we use the same syntax to call a function object as we do for a pointer to function. For example, we can derive a type of Func that can handle non-member functions:

```
class NMFunc : public Func {
  public:
    NMFunc( double (*f)( double ) ) : f_(f) {}
    double operator ()( double d ) { return f_( d ); }
  private:
    double (*f_)( double );
};
```

This allows us to integrate all the functions of our original version:

```
double aFunc( double x ) { ... }
//...
NMFunc g( aFunc );
double area = integrate( g, 0.0, 2.71828 );
```

We can also integrate member functions by wrapping an appropriate interface around a pointer to member function and a class object (see *Pointers to Member Functions Are Not Pointers* [16, 57]):

```
template <class C>
class MFunc : public Func {
  public:
    MFunc( C &obj, double (C::*f)(double) )
        : obj_(obj), f_(f) {}
    double operator ()( double d )
        { return (obj_.*f_)( d ); }
  private:
    C &obj_;
    double (C::*f_)( double );
};
//...
AClass anObj;
MFunc<AClass> f( anObj, &AClass::aFunc );
double area = integrate( f, 0.0, 2.71828 );
```