Function Parameters, pt. 2

Event-Based Coding + Practicality
Function Pointers

- Last time, we took our first “deep” look at function pointers.
  - We then based an example off of the following method and its signature.

\[
\text{bool myMethod(int x, int y);}\]
Function Pointers

bool (*funcPtr)(int, int);
Function Pointers

• Given the “type definition” below,

```c
typedef bool (*ExFuncPtr)(int, int);
```

the following are equivalent.

```c
ExFuncPtr funcPtr = myMethod;
```

```c
bool (*funcPtr)(int, int);
funcPtr = myMethod; // &myMethod
```
Function Pointers

• Our example typename alias can also be defined as follows:

typedef decltype(myMethod) *ExFuncPtr;
Further Complications

• We also examined “method” pointers and how the class name is a required element.

• Lastly, we began examining *functors*, objects designed and coded explicitly to model functions.
Applications

- We’ve previously mentioned that function pointers and functors can help to allow an algorithm’s functionality to be customized.
  - A comparison function can be provided to a sorting algorithm, for example.
Applications

• An alternate usage for functors and function pointers – they can be used deliberately to pass information and control within a program.
One design structure common in some programming language is that of the event.

- An object’s events are designed to notify other objects of interesting changes in its state.
Event-Based Programming

• Colloquially speaking, events are like a single user’s Twitter account.
  – Other users (objects) can subscribe to the event.
  – Whenever the event is raised, it’s like the user posting a tweet – all subscribers are notified of the new tweet.
Event-Based Programming

• In many languages, events are supported by structures called delegates.
  – Delegates take in functors or function pointers from their “subscribers” – typically, other objects.
  – At a later time, when the object wishes to raise the delegate’s event, the delegate is invoked, calling each functor/function pointer.
Delegates

In effect, delegates are lists of functors/function pointers that are stored for future notification.

- Whenever the event is raised, each stored functor/function pointer is called in its turn.
Event-Based Programming

• When each stored functor or function pointer is called, control naturally passes to the “subscribed” object.
  – That object may then react to the event however necessary, performing operations.
  – When finished, the next event “subscriber” is notified.
Event-Based Programming

- This paradigm is often used when creating GUI programs.
  - Each operating system has its own unique code for creating windows.
  - Programs use this pre-existing code in various arrangements to create the GUIs we are used to working with.
Event-Based Programming

• To allow individual programs to add functionality to windows, operating systems have events in their provided window code.
  – Programmers provide these events with functions designed to handle them appropriately, which serve as the bridge between program and OS.
Practicality

- As we noted last time, “function” pointers are incompatible with “method” pointers of similar signature.

```c
bool myMethod(int x, int y);
bool (*funcPtr)(int, int);
// vs.
bool MyClass::myMethod(int x, int y);
bool (MyClass::*ExFuncPtr)(int, int);
```
Practicality

• In fact, we cannot use a single variable definition to handle methods from multiple classes.
  – The exception: if the alternate classes are derived from the definition’s class and are calling methods known to exist from the base class.
  – Note: those methods could be overridden, being virtual in the base class.
Practicality

• One solution: if the function pointer is only needed within one other method, as a parameter... use of template programming can work.
  – The compiler will generate appropriate versions for each function pointer source.
  – Note: we still can’t mix function pointers with method pointers.
template <typename T>
void calledMethod

    (T obj, bool (T::*funcPtr)(int, int))
{
}

• This method can fit any appropriate method pointer.
Practicality

• But what if we want to store function pointers or method pointers within a single instance of our class?

  – Having various possible class types involved would easily complicate our efforts.

  – We can’t foreseeably template on all the possible class types at once for a single instance.
Polymorphism

- A reasonable solution would be to use polymorphism with functors to ensure a common type.
  - It may take more work, but it keeps the code somewhat clean.
Polymorphism

class TwoIntFunctor
{
public:
    virtual bool operator()(int a, int b) = 0;
}

// This class serves as a good polymorphic base
// class for our method functor to come, as well as
// a function pointer functor.
Polymorphism

class TwoIntFunctionPointer: public TwoIntFunctor
{
public:

TwoIntFunctionPointer(bool (*fnc)(int, int));
bool operator()(int a, int b);

private:

bool (*funcPtr)(int, int);

};

// This allows us to wrap a standard function pointer and have it fit methods with functor arguments.
template <typename T>
class TwoIntMethodFunctor<T>: public TwoIntFunctor
{
public:

    TwoIntMethodFunctor(T* obj, 
    bool (T::*fnc)(int, int));

    bool operator()(int a, int b);

private:

    T* obj;

    bool (T::*funcPtr)(int, int);
};
Polymorphism

// Note that while the class template will generate numerous versions, they are all polymorphic to the base class TwoIntFunctor.

template <typename T>
TwoIntMethodFunctor<T>::TwoIntMethodFunctor(
    T* obj, bool (T::*fnc)(int, int))
{
    this->obj = obj;
    funcPtr = fnc;
}
Polymorphism

template <typename T>
bool TwoIntMethodFunctor<T>::operator()(int a, int b);
{
    return obj->*funcPtr(a, b);
}

// "->*" is a special operator signifying (here)
// that funcPtr is a pointer to a member of obj,
// and that we need to dereference it for the call.
Polymorphism

- The “method functor” we created requires both a method pointer and an instance upon which it is to be called.
  - This is similar to what “delegates” store within the .NET language family.
  - More efficient implementations exist for C++, but they are far more complicated.