Function Parameters

Pointers + Functors
Memory Organization

• Up through now, we’ve talked a lot about pointers at various times, for various reasons.
  – They’re handy for referencing large amounts of data, or for simply pointing to data rather than making replicas.

• We also noted the rough organization of memory long ago...
Memory Organization

(MIPS architecture)
Memory Organization

• Global variables are given special space in a dedicated, fixed area of main memory.
  – “Static data” on the previous slide.

• Objects and data allocated by reference, at construction, are given space on the “heap.”
  – “Dynamic data” on the previous slide.
Memory Organization

• The local variables of a function are stored in the “stack,” on the opposite end of main memory from the heap.
  – When control passes to a function, memory is allocated at the “top” of the stack for that method’s local variables.
  – Value types are stored directly in this area.
  – For reference types, their pointer (the address) is stored here.
Memory Organization

• Note one of the areas that hadn’t been mentioned before: The “text” segment.
  – This is where the program lives in memory.
  – As programs are installed to our hard drives and take up space there, they exist as data… data that is interpreted as a program by the OS, and thus CPU.
Pointers

• We’ve noted previously that pointers effectively serve as addresses of data in memory.
  – If allocated through the `new` keyword, on the “dynamic/heap” segment...
  – If allocated locally, by-value, within a function, on the “stack” segment.
  – If allocated by-value in a global constant, in the “static” segment.
Function Pointers

- Accordingly, it is also possible (and desirable, at times) to have pointers to functions.
  - Note that the program is effectively another form of data stored in memory.
  - That pointer can then be used to call the referenced function, almost as if the variable itself were the function.
Function Pointers

- Function pointer syntax is somewhat complicated, due to all the details that must be specified for a function.
  - Return type
  - The parameters
  - Class name (for methods)
  - More on this in a bit.
Function Pointers

• Suppose we have a function as follows:

```cpp
bool myMethod(int x, int y);
```

• A function pointer to that function could be declared as follows:

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

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

• The “declaration” of the function pointer’s type needs to match the function’s declaration near-precisely.
  – Const on the parameters, & or *, etc.
Function Pointers

• One way to alleviate the complexity of the function pointer’s type specification in code: the keyword `typedef`.
Function Pointers

• Given our example method

```c
bool myMethod(int x, int y);
```

we might declare the following type to make function pointers simpler.

```c
typedef bool (*ExFuncPtr)(int, int);
ExFuncPtr funcPtr = myMethod;
```
Function Pointers

- Given our “type definition” below,

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

the following are equivalent.

```
ExFuncPtr funcPtr = myMethod;
```

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

• The `typedef` keyword is useful for giving alternate names – or “aliases” – to potentially complex types.

• To simplify this even further: `decltype`.
Function Pointers

• Suppose we have a function as follows:

```cpp
bool myMethod(int x, int y);
```

• A typename alias can also be defined as follows:

```cpp
typedef decltype(myMethod) *ExFuncPtr;
```
Function Pointers

• Variables *and parameters* can be declared in either way.
  – Of course, the *typedef* must be set up first to use the custom type name.
  – C++ is rigid about type safety and requires any potentially-assigned functions to match completely.
Function Pointer

- To call the function being referenced by the function pointer, use the variable name as if it were the actual function.

```c
if(funcPtr(2, 3)) { ... }
```

//Alternatively,
//(*funcPtr)(2, 3);

Method Pointers

• Suppose the following:

```cpp
bool MyClass::myMethod(int x, int y);
typedef bool (*ExFuncPtr)(int, int);
ExFuncPtr funcPtr = myMethod;
```

• Will this work?
  – Actually... no.
Method Pointers

• Things get more complex when we add object-orientation (and classes) into the mix.
  – Primary complication: the this pointer.
  – In particular, the method (unless static) requires a class instance in order to work.
    • If myMethod() were static, then the last slide’s example would work.
  – This requires tracking of another reference in addition to the function pointer itself.
Method Pointers

• Things get more complex when we add object-orientation (and classes) into the mix.
  – Primary complication: the `this` pointer.
  – While not seen directly in the method signature, `this` is a hidden parameter of every non-static class method.
  – Thus, there’s an extra parameter for any class method.
Method Pointers

• This gives the general insight and overview of why C++’s implementation is what it is.
  – There are even more complexities than this behind the scene.
  – Worse case: what if the function being referenced is virtual?
Method Pointers

• C++’s fix for the problem:

```cpp
bool MyClass::myMethod(int x, int y);

typedef bool (MyClass::*ExFuncPtr)(int, int);
ExFuncPtr funcPtr = myMethod;
```

– The class must be specified as part of the function pointer’s type spec.
Functors

• Suppose instead that we have a situation where we wish to have a function... that takes on a permanent, set value for one or more parameters.

  – A very simple example: suppose a function `int add(int x, int y)` where `y = 5` sometimes, but `y = 3` others?
Functors

• In essence, we want a “function” that permanently remembers values, like a class with fields.

• The general solution?
  – Function objects! (also called *functors*)
Functors

• Note: not all languages allow you to directly work with function pointers.
  – Java, for example, does not.

• However, even in many languages without function pointers, the function object is very possible.
  – If nothing else, define a common base class for desired function objects – polymorphism helps here.
Functors

- C++ has a special way of solving the problem – operator overloading.
  - In particular, `operator()`, the function call operator.
  - Overloading this operator gives the following syntax when using the operator:

```c++
addFunctor(2); // Looks like a // method!
```
class AddFunctor
{
public:
    AddFunctor(int c)
    : term(c) {}

    int operator()(int x);

private:
    int term;
};
Functors

AddFunctor addFunctor(3); // Initialization:
    // will add 3.

    // ...

int x = addFunctor(2); // x = 5 (2 + 3)
Functors

• Alternatively, there may be times we may want functors, but without parameters.
  – Consider a template class that may want to take on one of myriad possible functions via “pointer” for its operation.
  – Working out a common signature for a template class’s function parameter could be problematic or raise issues.
Functors

• Template programming creates a special, compiled version of templated methods/objects for each oncoming type.

  – As a result, the compiler can adapt a version of templated functions/objects to make use of incoming functors and ensure type safety and signature compatibility.
Functors

• A possible example for functor use:
  – Performing some operation on each element of a for-loop.
  – That operation could take a parameter and then be applied to each element of an iterable structure.
  – It would be possible to write a special for-loop that takes in functors and does this!
Functors

• Other uses:
  – `std::sort` can take functors for ordering elements.
  – `std::map` operates similarly.
  – To each the use of each, there exists (in `<functional>`) pre-built functors like `less`, `greater`, etc.
  – There also exist `plus`, `minus`, etc.
Functors

- In essence, functors are objects designed to look like \textit{and operate} like functions.
  - In C++, the syntax is such that you can “call” a functor, just like calling a function.