Function Pointers

Before we begin

- Any questions?

Recall: Program Memory

- The memory used by a program is generally separated into the following sections:
  - Code – Where the executable code is kept
  - Global – Where storage for global variables is kept
  - Stack – Runtime stack (where local variables are kept)
  - Heap – Free store for dynamically allocated variables.
  - Exception – special place for things thrown

Function pointers

- Provides access to executable code section.
- Function Pointers are pointers
  - variables, which point to the address of a function.
  - Contains a memory address
- Examples from
  - http://www.function-pointer.org/
  - Yes, function pointers have their own web site

Function pointers: but why?

```
// the four arithmetic operations
// one of these functions is selected at runtime
// with a switch or a function pointer
float Plus (float a, float b) { return a+b; }
float Minus (float a, float b) { return a-b; }
float Multiply (float a, float b) { return a*b; }
float Divide (float a, float b) { return a/b; }
```

Function pointers: but why?

```
// solution with a switch-statement –
// <opCode> specifies which operation to execute

void Switch(float a, float b, char opCode)
{
  float result; // execute operation
  switch(opCode) {
      case '+' : result = Plus (a, b); break;
      case '-' : result = Minus (a, b); break;
      case '*' : result = Multiply (a, b); break;
      case '/' : result = Divide (a, b); break;
   }
   cout << "switch: 2+5=" << result << endl;
}
```

Using function pointers

```
// solution with a function pointer
// <pt2Func> is a function pointer and points to
// a function which takes two floats and returns a
// float. Thefunction pointer
// "specifies" which operation shall be executed.

void Switch_With_Function_Pointer(float a, float b,
   float (*pt2Func)(float, float))
{
  // call using function pointer
  float result = pt2Func(a, b);
  cout << result << endl;
}
```

Using function pointers

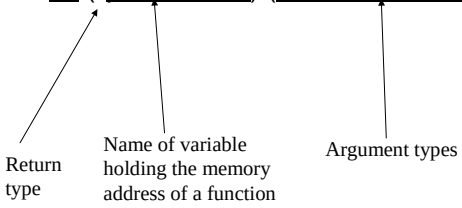
```
// execute example code

void Replace_A_Switch()
{
  // '+' specifies function 'Plus' to be executed
  Switch(2, 5, '+');

  //pointer to function 'Minus
  Switch_With_Function_Pointer(2, 5, &Minus);
}
```

Using function pointers

- **Important note:**
  - A function pointer always points to a function with a specific signature!
  - all functions, you want to use with the same function pointer, must have the **same parameters and return-type!**
- Questions so far?

Function pointer syntax

- int (*pt2Function) (float, char, char) ;



Function pointer syntax

- int (*pt2Function) (float, char, char) ;
  - Defines a pointer variable pt2Function
  - The function that this pointer is pointing to takes a float and 2 chars as arguments
  - The function that this pointer is pointing to will return an int.

Function pointer syntax

- Note:
  - int (*pt2Function) (float, char, char) ;
    - Is not the same as
  - int *pt2Function (float, char, char) ;

Function pointer syntax

- Note:
  - int (*pt2Function) (float, char, char) ;
    - Defines a function pointer variable
  - int *pt2Function (float, char, char) ;
    - Defines a function that returns a pointer to an int.

Function pointer syntax

- Assigning to a function pointer:

```
int DoIt (float a, char b, char c)
{ printf("DoIt\n"); return a+b+c; }

int DoMore(float a, char b, char c)
{ printf("DoMore\n"); return a-b+c; }

   int (*pt2Function) (float, char, char) ;
   pt2Function = DoMore; // assignment
   pt2Function = &DoIt; // alternative
```

Must have same arguments and return type!

Function pointer syntax

- **Calling a Function using a Function Pointer**
  - Can call directly or dereference

```
int result1 = pt2Function (12, 'a', 'b');
int result2 = (*pt2Function) (12, 'a', 'b');
```

Function pointer syntax

- Can also assign to member functions.

```
class TMyClass {
public:
  int DoIt (float a, char b, char c){ return a+b+c; };
  int DoMore(float a, char b, char c){return a-b+c; }; /*
  more of TMyClass */
};

int (TMyClass:: *pt2Function) (float, char, char) ;
pt2Function = TMyClass::DoMore; // assignment
pt2Function = &TMyClass::DoIt; // alternative
```

Function pointer syntax

- Can also assign to member functions

```
int (TMyClass:: *pt2Function) (float, char, char) ;
pt2Function = TMyClass::DoMore; // assignment

TMyClass A = new TMyClass();
A->pt2Function (7.7, 'a','b'); // direct call
A->(* pt2Function)(7.7, 'a','b'); // dereference
```

Function pointer syntax

- Once again, return type and args must match:

```
void (*pf)(string);
void f1 (string);
int f2 (string);
void f3 (int *);

void f()
{
   pf = &f1;  // okay
   pf = &f2;  // bad return type
   pf = &f3;  // bad arg type
   pf ("Foo");  // okay
   pf (1);      // bad arg type
   int i = pf ("Zero");  // bad return type;
}
```

Function pointer syntax

- Passing function pointer to a function

```
// <pt2Func> is a pointer to a function which returns an int
   and takes a float and two char
void PassPtr(int (*pt2Func)(float, char, char))
{
   // call using function pointer
   float result = pt2Func(12, 'a', 'b');
}

// execute example code - 'DoIt' is a suitable function
void Pass_A_Function_Pointer()
{
    PassPtr(&DoIt);
}
```

Function pointer syntax

- Returning a function pointer

```
// function takes a char and returns a pointer to
   a
// function which is taking two
// floats and returns a float.
// <opCode> specifies which function to return

float (*GetPtr1(const char opCode))(float, float)
{
   if(opCode == '+') return &Plus;
   if(opCode == '-') return &Minus;

}
```

Function pointer syntax

- `float (*GetPtr1(const char opCode))(float, float)`

Actual function name

Indicates pointer to a function is returned

Args of function pointed to

Type returned by function pointed to

Function pointer syntax

- Returning a function pointer
- Use typedef to avoid that crazy syntax

```
typedef float(*pt2Func)(float, float);

pt2Func GetPtr2(const char opCode)
{
    if(opCode == '+') return &Plus;
    if(opCode == '-') return &Minus;

}
```

Arrays of function pointers

- Since function pointers are just pointers, you can easily have arrays of them

```
typedef int (*pt2Function)(float, char, char); /

void Array_Of_Function_Pointers()
{
   pt2Function funcArr[10];

   funcArr[0] = &DoIt;
   funcArr[1] = &DoMore;

   printf("%d\n", funcArr[1](12, 'a', 'b'));
   printf("%d\n", funcArr[0](12, 'a', 'b'));
}
```

Arrays of function pointers

- But why?
  - Let's assume we have a menu system for a GUI.
  - Each menu item will correspond to an action.
  - Can use array of function pointers rather than a large switch or if/then statements.

Arrays of function pointers

```
typedef void (*MenuF)();

MenuF edit_ops[] = { &cut, &copy, &paste, &find };
MenuF file_ops[] = { &open, &new, &close, &save };


MenuF *button2 = edit_ops;
MenuF *button3 = file_ops;

// When selection is made
Button2[2]();
```

Questions?

Callbacks

- Function Pointers provide the concept of callback functions.
- Example

```
typedef void (*terminate_handler)();
terminate_handler set_terminate(terminate_handler);
```

Callbacks

- Consider qsort:

```
void qsort(
    void* field,
    size_t nElements,
    size_t sizeOfAnElement,
    int(_USERENTRY *cmpFunc)(const void*, const void*)
);
```

Callback that defines compare function

Callbacks

```
void qsort( ... , int(_USERENTRY *cmpFunc)(const void*,
   const void*))
{
   /* sort algorithm - note: item1 and item2 are void-
   pointers */
   int bigger=cmpFunc(item1, item2); // make callback

  /* use the result */

}
```

Callbacks

- Of course, if we want to do generic programming, why not use STL?

The function object

- Or functor
  - Object that mimics a pointer to a function.
  - Overrides the call operator (operator()).

```
class cmpFunct
{
public:
     cmpFunct() {}
     int operator() (int a, int b) { return a  < b;
  }
}

cmpFunct f;
int result = f (7, 10);
```

The function object

- The signature of the function object is dependent on the args/return type of the operator() that is overridden

```
class cmpFunct
{
public:
     cmpFunct() {}
     int operator() (int a, int b) { return a  < b; }
}
```

The function object

- Interesting means to have multiple definitions.

```
class cmpFunct
{
public:
     cmpFunct() {}
     int operator() (int a, int b) { return a  < b; }
     int operator() (double a, double b)
            { return a  < b; }
}

cmpFunct f;
int result = f (7, 10); // okay
int result2 = f (5.6, 8.9); // also okay
```

The function object

- Must still follow the rule that signatures cannot differ by return type alone.

```
class cmpFunct
{
public:
     cmpFunct() {}
     int operator() (int a, int b) { return a  < b; }
     double operator() (int a, int b) { // } // not ok
     int operator() (double a, double b)
            { return a  < b; }
}
```

Generic programming

- We all know that the “correct” way of doing generic programming in C++ is to use STL algorithms:

```
template <class RandomAccessIterator, class Compare>
void sort (RandomAccessIterator first,
  RandomAccessIterator last, Compare comp);

vector<int> v1;
cmpFunct f;
sort (v1.begin(), v1.end(), f)
```

Templated functors

- Of course, function objects can be templated

```
template <class T>
class cmpFunct
{
public:
     cmpFunct() {}
     int operator() (T a, T b) { return a  < b; }
}

cmpFunct<int> f;
int result = f (7, 10); // okay
```

Templated functors

- Can use on classes as well!
  - As long as operator< is defined for Foo

```
template <class T>
class cmpFunct
{
public:
     cmpFunct() {}
     int operator() (T a, T b) { return a  < b; }
}

cmpFunct<Foo> f;
int result = f (Foo(7), Foo(10)); // okay
```

Functors and state

- Because functors are objects, they can contain state and additional methods

```
template <class T> class Sum
{
private:
   T res;
public:
   Sum (T i=0) : res (i) {}
   void operator() (T x) { res += x; }
   T result const { return res; }
};
```

Fine as long as T can be initialized by 0

Fine as long as += Is defined for T

Functors and state

- What you can do with this:

```
void f (list<double> ld)
{
     Sum<double> s;
     for_each (ld.begin(), ld.end(), s);
     cout << "The sum is " << s.result() << endl;
}
```

Functors and algorithms

- Generally, algorithms do not care if a "function argument" is a
  - Function
  - Pointer to a function
  - Functor

Functors and algorithms

```
int compareFunction (int a, int b)
{ return a < b; }

vector<int> v1;
cmpFunct f;
sort (v1.begin(), v1.end(), f);  // is okay
sort (v1.begin(), v1.end(), &compareFunction);  // is
   okay
sort (v1.begin(), v1.end(), compareFunction);  // is
   okay
```

Questions?

Functors

- Questions?