Week 5

Functions & Error Handling

Agenda

- Week 5-1 – Functions
  - Function Syntax
  - Linkage/Recursion
  - Function Pointers, Function Objects, Lambda Expressions
- Week 5-2 – Error Handling
  - Exceptions
  - Standard Exceptions
  - Exits

Week 5-1

Function Syntax

- Generally speaking a function takes on the form of:

```
return_type identifier(parameter-list) [qualifers]
```

```
ostream& Playdoh::display(ostream& os) const { … }
```

- In C++11 they added ability to have a type inferred return type via the auto keyword and a trailing return type declaration:

```
auto identifier(parameter-type-list) -> return-type;
```

- This variant of the function syntax is useful in conjunction with enums and templated functions

Linkage with Functions

- Similar to the notion of linkage addressed before with variables and entities, it can also be applied to functions
- By default all functions are externally linked meaning they can be seen outside of their own translation unit. Labeling a function as extern is then a bit redundant
- We can make a function have internal linkage through the same static keyword

Recursion

- Functions can be recursive in that they call themselves within their own body.
- Recursive functions need to have a exit condition to determine when the recursion should end (else it would keep going on and on and on)
- When that exit condition is met each recursive call returns back through the call stack back to the initial call of the function
- Recursive functions take up a good of stack space/memory to keep track of each recursion

Recursion (isPrime)

- Let's consider making a function that determines whether or not a given number is a prime number (divisible only by itself and 1 evenly)
- What would be its exit condition? How do we know when to stop recurring?
- What do we do when we keep recurring? How do we call ourselves (the function)?

Recursion (isPrime)

```
bool isPrime(int num, int recur) {
  bool ret = false;
  if (num == 1 || recur == 1)
    ret = true;
  else if ((num % recur) == 0)
    ret = false;
  else {
    ret = isPrime(num, recur - 1);
  }
  return ret;
}
```

Iterative Version (isPrime)

```
bool isPrimeIter(int num) {
  bool ret = true;
  if (num != 1) {
    int i = num - 1;
    bool done = false;
    while (i > 1 && !done) {
      if (num % i == 0) {
        done = true;
        ret = false;
      }
      i--;
    }
  }
  return ret;
}
```

Function Pointer

- As functions also reside in memory they have an address
- This then means we can store these addresses in pointers which leads us to function pointers – pointers that hold an address of a function type
- The syntax of a pointer to a function is:

```
return-type (*identifier) (parameter-list) [= fn];
```

Function Pointer

Note the *

return-type (*identifier) (parameter-list) [= fn];

Return type of the function

Name of the pointer to function

Param list

Assignment to an initial address

Function Pointer

```
void (*ptrToFuncA) () = funcA;
```

```
void funcA() {
  cout << "I am Function A" << endl;
}
```

```
void (*ptrToFuncB) () = funcB;
```

```
void funcB() {
  cout << "I am Function B" << endl;
}
```

Array of Pointers to Functions

- Much like pointers of non function types we can have an array pointers to functions
- This can be useful to store a bunch of functions in a more compact way
- The syntax for an array of function pointers can look like this:

```
return-type (*identifier[arraysize]) (parameter-list) [= fn];
```

Note the array size here

Function Objects

- Along with using pointers with functions, we can also represent functions using objects
- Compared with function pointers we can also store state in a function object
- These function objects are also called functors
- In C++ terms a functor can be defined as the class from which function objects are instantiated
- Effectively we'll be creating classes that overload the function call operator

Function Objects

- Along with using pointers with functions, we can also represent functions using objects
- Compared with function pointers we can also store state in a function object
- These function objects are also called functors
- In C++ terms a functor can be defined as the class from which function objects are instantiated
- Effectively we'll be creating classes that overload the function call operator

Lambda Expressions

- Lambda expressions are a method of creating a localized function (that only exists within a function)
- These expressions don't require a name (anonymous functions) and are a short hand for creating ad-hoc function objects
- If the expression is used more than once it can be given a name to be referred to in that function's scope
- A lambda expression can capture variables from within the host function (meaning it can make use of variables from its surrounding scope)

Lambda Expressions

- A lambda expression's syntax is:

[capture-list](parameter-declaration-clause) [mutable] -> optional-return-type {

    // function body

}

```
[](char c) {
  c = toupper(c);
  return c != 'A' || c != 'E' || c != 'I' || c != 'O' || c != 'U';
};
```

Lambda Expressions

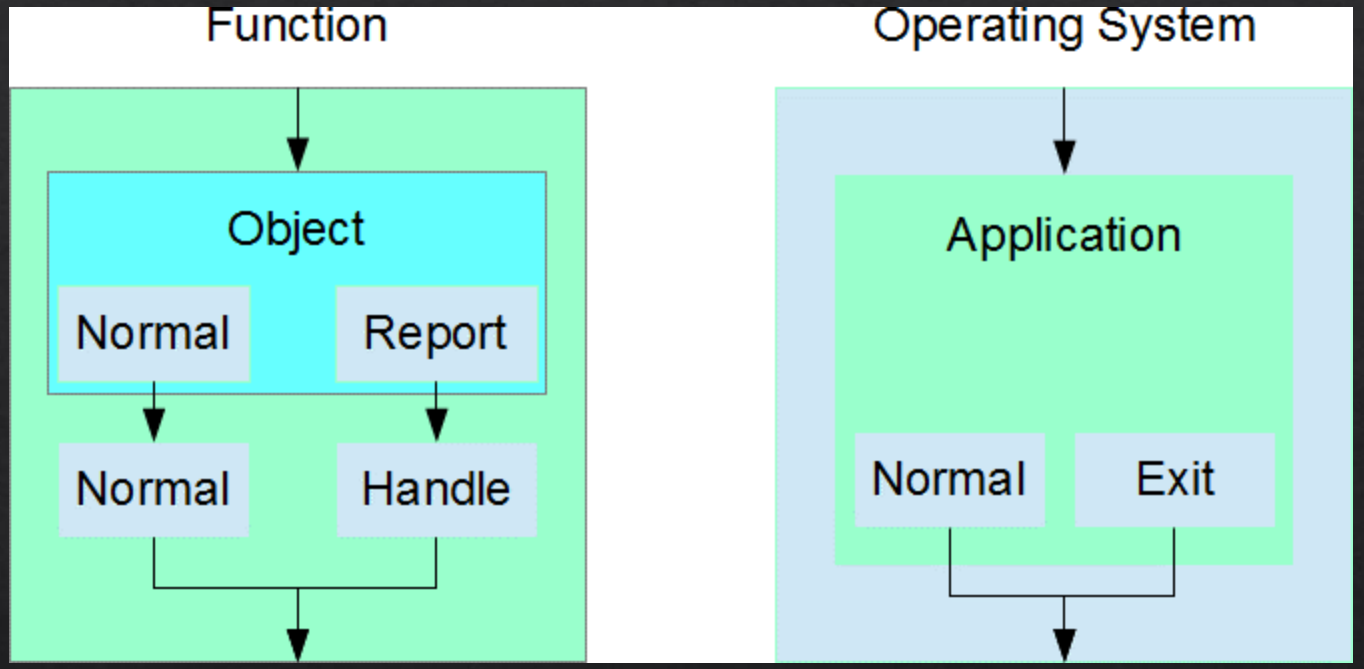
- The capture list from the expression is a mechanism for passing non local variables to the body of the lambda expressions
- The capture list can take a few forms:
  - [ ] – An empty capture list will denote no capture thus no reliance on variables in the same scope as the function
  - [=] – An equals sign denotes capture by value where we can use all variables from the same scope as the function in a const like manner
  - [&] – An ampersand sign denotes capture by reference where we can use all variables from the same scope as the function and be able to modify those variables

Lambda Expressions

- The capture list also be selective about what variables they capture and whether to do it with by value or by reference:
  - [=](...) - captures all non-local variables by value
  - [&](...) - captures all non-local variables by reference
  - [=,&x,&y](...) - captures x and y by reference, all else by value
  - [&,x,y](...) - captures x and y by value, all else by reference
  - [x,&y](...) - captures x by value and y by reference
  - [this](...) - captures this by value

Week 5-2

Exceptions


Function
Object
Normal
Report
Normal
Handle
Operating System
Application
Normal
Exit

Exceptions

- Exceptions much like they sound are things that occur outside of the normal
- It could be caused by some error in calculation, or an operation that went off the rails
- What is considered an exception can be a part of decision in design
- When dealing with exceptions it has two distinct parts:
  - Reporting the exception (throw)
  - Handling the exception (catch)

Exceptions (Reporting)

- In order to report an exception having occurred there is the throw keyword and its usage takes the form of:

throw [expression]

Where the expression is an expression of a previously defined type (eg. a standard exception type)

Exceptions (Handling)

- In order to handle a reported exception it needs to be ‘caught’
- The language offers two keywords try and catch to perform the handling process
- First we have the try block which is block of code that may generate an exception
- It’s then followed by a number of catch blocks which will handle said exceptions if they occurred from our try attempt

Exceptions (Handling)

```
try {
     // code that might generate exceptions

} catch (Type identifier) {
     // handler code for a specific type of exception
} catch (Type identifier) {
     // handler code for another specific type of exception
} catch (...) {
     // handler code for all other types of exception
}
```

The Type identifier portion refers to what we're trying to catch, if that something is thrown it'll get caught by the relevant catch block

Standard Exceptions

- The standard C++ libraries include a library of exception classes.
- The base class of this hierarchy is called exception and is in the <exception> header file
- Some of the classes derived from this base include:
  - out_of_range – used to indicate going beyond the range of an array for example
  - length_error – used to indicate attempts to exceed the implementation defined lengths of some object
  - overflow_error – used to indicate that arithmetic overflows (ie when a result of a computation is too large for the destination type)

noexcept

- C++11 introduced a new keyword called noexcept to identify a function as one that doesn't throw an exception
- This allows for the compiler to perform some optimizations that wouldn't be possible if exceptions passed through the function
- If a function marked as noexcept has an uncaught exception then the program would terminate immediately

Exits

- The main function of a program is its entry point when it exits it returns an integer to the operating system to convey an exit code
- In C++, that integer is based on the return statement in the main (defaults to return 0)
- In addition to this return statement there are library functions that provide some different ways to exit a program:
  - Standard Exits
  - Abnormal Exits

Standard Exits

- Normal exits involve calling the destructors that would be called normally at the end of the life time of those objects and then return the status integer to the operating system
- To simulate this kind of exit, there are two functions we could use:
  - int exit(int) – initiates a termination process that destroy objects in scope and performs other clean up then returns the exit code
  - int atexit(void (*) (void)) – registers a function to be called when the above exit function called, each registered function has to be of the type void (*identifier) (void)). Can support up to 32 functions

Abnormal Exits

- Two functions that simulate an abnormal termination (when programs don't exit as expected):
  - void std::terminate() – The function terminates a program execution as a result of an error related to exception handling.
  - void std::abort() – The function terminates a program via a SIGABRT signal, the function like terminate() note execute destructors for objects or any other clean up or call any functions registered by the atexit() function