# Lecture Notes 3

## C++ Smart Pointers

- Memory Locations
  - Data – Place where global variables reside
  - Stack – Place where local variables are automatically allocated/freed
  - Heap – Place where variable data is dynamically allocated/freed
- Memory Leak – Memory is allocated on the heap and not freed before the last reference to it is lost
  - Issue – Performance may suffer because allocating memory that is unused
- Dangling Reference – A reference to memory is invalid (the location has already been freed, possibly repurposed)
  - Issue – The memory may be freed by the system and can cause a program crash, or worse the program could have reallocated the code for another purpose and multiple both are overwriting one another's work without being able to detect it
- Unique Pointer – `unique_ptr`
  - Only has single reference, allows for explicit hand-off of object ownership
  - Automatically frees memory when done
  - `make_unique` (C++14) – Preferred way to create an object that has a `unique_ptr`
  - Example:

```
void foo(){
    std::unique_ptr<int> APtr = std::make_unique<int>(3);
    std::unique_ptr<int> BPtr;

    // The following line prints out 3
    std::cout<<(*APtr)<<std::endl;
    // The following line hands over ownership to BPtr
    BPtr = std::move(APtr);
}
```

- Shared Pointer – `shared_ptr`
  - Allows for multiple references
  - Reference counting, when reference count goes to zero, memory is automatically freed
  - `make_shared` – Preferred way to create an object that has a `shared_ptr`
  - Example:

    ```
    void foo(){
        std::shared_ptr<int> APtr = std::make_shared<int>(3);
        std::shared_ptr<int> BPtr;

        // The following line prints out 3
        std::cout<<(*APtr)<<std::endl;
        // BPtr has a copy of the reference
        BPtr = APtr;
        // At end of foo the int will be freed because ref
        // count goes to zero
    }
    ```

- Weak Pointer – `weak_ptr`
  - Allows for multiple references, but doesn't maintain the reference count breaks cycles
  - `lock()` – Returns a `shared_ptr` to the object, or returns a `nullptr` if already freed
  - Example:

    ```
    std::weak_ptr<int> WPtr;
    void foo(){
        // Has to be copied into a shared_ptr before usage
        if(auto SPt = WPtr.lock()){
            // Access to pointer through SPt
        }
        else{
            // Failed to gain access
        }
    }

    {
        auto SPtr = std::make_shared<int>(42);
        WPtr = SPtr;
        foo();
    }
    foo(); // WPtr will not be able to access through lock
    ```

- Shared Pointer from `this` Pointer – `enable_shared_from_this`
  - Don't create a new `shared_ptr` from the `this` raw pointer, it will create a new reference count, need to use `shared_from_this()`.
  - `enable_shared_from_this` – Template class that will allow creating a `shared_ptr` from `this` raw pointer

- `shared_from_this()` – Returns a `shared_ptr` to the object that is calling the function
- Example:

```
class C1 : public std::enable_shared_from_this<C1> {
    private:
        int Val;
    public:
        int foo();
        int bar(std::shared_ptr<C1> ptr);
};

int C1::foo(){
    return Val * bar(shared_from_this());
}

int C1::bar(std::shared_ptr<C1> ptr){
    return ptr->Val + 3;
}
```

## Function Argument Passing

- Pass by Value – Creates a copy of the variable that is passed in
  - Example
```
int foo(int x){
    int y = x;
    x += 4;
    return x + y;
}

int main(){
    int z = 3;
    int w = foo(z);
    // w is now 10 and z is still 3
    return 0;
}
```
- Pass by Reference – Passes a reference that will modify the argument if modified in the function
  - Example
```
int foo(int &x){
    int y = x;
    x += 4;
    return x + y;
}

int main(){
    int z = 3;
    int w = foo(z);
    // w is now 10 and z is now 7
    return 0;
}
```
- `const &` – Provides a contract that the function will not modify the variable that is passed by reference
  - Example
```
int foo(const int &x){
    int y = x;
    x += 4; // This line won't compile because modifying x
    return x + y;
}
```
- Pointers – An address to an object of a specified type
  - Example:
```
int *x;    // This is a pointer to an int
double *y; // This is a pointer to a double
```
- Address-of Operator `&` - Gets the address of the variable
  - Example:
```
int x;
int *y = &x; // y is an int pointer that is pointing to x
```

- Dereferencing – Accessing the value at the location specified by the pointer
  - Example:

    ```
    int x = 6;
    int *y = &x; // y is an int pointer that is pointing to x
    int z = *y;  // y is dereferenced, so the value returned is
                 // 6 because y is "pointing to" x and x has
                 // the value 6
    ```

- Passing a Pointer – This is passing actually passing by value, it copies the address passed in
  - Example

    ```
    int foo(int *x){
        int y = *x;    // Dereferencing x
        *x += 4;
        x = &y;        // x now points to y
        return *x + y;
    }

    int main(){
        int z = 3;
        int w = foo(&z); // pass in the address of z
        // w is now 6 and z is now 7
        return 0;
    }
    ```

## Conversion, Operator Overloading

- Constructor – The function responsible for initializing the object, can be overloaded for different types
  - Example:

```
class Value{
     public:
         std::string Data;
         Value(); // Default constructor
         Value(const Value &val); // Copy constructor
         Value(const std::string &str); // String
                                        // constructor
         Value(int i); // int constructor
         Value(double d); // double constructor
};

Value::Value(){
}

Value::Value(const Value &val){
    Data = val.Data;
}

Value::Value(const std::string &str){
    Data = str;
}

Value::Value(int i){
    Data = std::to_string(i);
}

Value::Value(double d) {
    Data = std::to_string(d);
}
```

- Assignment – Overloads the = operator allows for customizing the assignment from RHS
  - Example:

```
class Value{
     public:
         std::string Data;
         // Constructors here
         Value &operator=(const Value &val); // assignment
};

Value &Value::operator=(const Value &val){
    if(this != &val){
        Data = val.Data;
    }
    return *this;
}

int main(){
    Value i{3};
    Value j;

    j = i; // same thing as j.operator=(i);
    return 0;
}
```

- Cast Overload Operator – Allows for conversion to another type from the object
  - Example:

```
class Value{
     public:
         std::string Data;
         // Other functions here
         operator int() const; // int cast
};

Value::operator int() const{
    return std::stoi(Data);
}

int main(){
    Value i{3};
    int j = 7;

    j = i; // same thing as j = i.operator int();
    return 0;
}
```

- Extended Example

```
class Value{
     public:
         std::string Data;
         Value(const Value &val); // Copy constructor
         Value(int i); // int constructor
         Value &operator=(const Value &val); // assignment
         operator int() const; // int cast

};

Value::Value(const Value &val){
    Data = val.Data;
}

Value::Value(int i){
    Data = std::to_string(i);
}

Value &Value::operator=(const Value &val){
    if(this != &val){
        Data = val.Data;
    }
    return *this;
}

Value::operator int() const{
    return std::stoi(Data);
}

int main(){
    Value i{3}, k{4};
    int j = 7;
    k = j + i;
    // k.operator=(Value(j + i.operator int());
    return 0;
}
```