Basic Classes

```
class Set {
 public:
       Set();
       Set(string name, int capacity = 2);
       ~Set();
       Set(const Set& src);

       Set& operator=(const Set& src);

       void rename(const string& val) {...}
       bool insert(const string& val);
       bool contains(const string& val) const;
       int size() const;
 private:
       int m_size = 0;
       int m_capacity;
       string m_name;
};
```

Note: default parameters must come after non-defaults in function. When being called, there can't be "gaps", i.e. you must provide the first n parameters, leaving the rest to be defaults.

Structs

Same as classes but members are public by default. Typically used for simple data storage.

Pointers

```
int p;  defining regular variable
&p              gets address
int *q;         defining pointer
*q              gets value at location q

void a(int &ptr) {...}        passes by reference
a(p); a(*q);
void b(int ptr) {...}  uses pointers instead
b(&p); b(q);

int nums[3];
int *numptr = nums;           note: can omit &
nums[i] = *(nums + i)
int fn(int *arr) OR (int arr[]) OR (int arr[6]);

int plus(int a, int b) {return a + b;}
int minus(int a, int b) {return a - b;}
int (*f)(int, int);
f = plus;
f(2, 2);        returns 4
f = minus;
f(2, 2);        returns 0
```

Note: can omit the & during function pointer assignment and can only assign f to a function if params are the same type

Dynamic Allocation

Calling new allocates memory for and initializes a new object/array of objects. Calling delete frees the memory allocated by new. Call new exactly as many times as you call delete.

```
Set *a = new Set;
delete a;

Set *b = new Set[5];
delete [] b;
```

This

A "this pointer" is added to all methods in a class by the compiler and hidden by default. It can be used to refer to the object that calls a certain function/locate a class in memory.

```
int Set::size() const {return m_size}

Under the hood:
int Set::size(Set *this) const {
       return this->m_size;
}
```

Member access:

Objects of the same class can access each others' public and private members. Outside of the same class, only public members are accessible.

```
a.m_size (objects)
a->b (*a).b (pointer to objects)
```

Constructors

A default constructor is implicitly generated only if the user does not define one, which calls the default constructor for all members. This default constructor does not initialize primitives. Constructors are called whenever a new variable of a class is created.

Default constructor

```
Set::Set() {...}
Set a;
Set* b = new Set;
a(), new Set() also work
```

With arguments

```
Set::Set(string name, int capacity) {...}
Set a("albert", 5);
Set b* = new Set("bill", 6);
```

Initializer Lists

Required for reference members, const members, and member objects/parent class without default constructors. They are generally good practice as well and can be used for any variable.

```
Set::Set(string name, int capacity)
   : m_name(name), m_capacity(capacity) {...}
```

Note: placed with the constructor's definition, not prototype.

Rule of Three

If you need a copy constructor, assignment operator, or destructor, you probably need all three. Define them all manually.

Copy Constructors

A copy constructor is automatically generated if the user does not define one, which copies the values of every member variable from a source of the same type at initialization time.

This is not sufficient for instances involving pointers/dynamic allocation as the value of the pointer is copied (a shallow copy) instead of the underlying data (a deep copy), meaning the user must define one.

```
Set::Set(const Set& src) {...}
Set a = b;
```

```
Set* c = new Set(d);
```

Destructors

Destructors are automatically called when variables drop out of scope or when the delete keyword is used. They need to be manually defined when classes contain pointer members/uses dynamic allocation as destructing the pointer variable does not free the memory it points to.

```
Set::~Set() {...}
```

Assignment Operator

Used when variables of a class are assigned to other variables of the same class with the = operator.

```
Set::Set& operator=(const Set& src) {
        if (this != &src) {...}
        return *this;
}
Set a;                  - default constructor called
a = b;                  - assignment operator
Set *c = new Set;       - default constructor called
*c = d;                 - assignment operator
```

Class Composition

Occurs when classes contain members of other classes. C++ calls member variables' default constructors first, in order of definition, running the outer object's constructor last. Destruction runs in the opposite direction.

```
class Bed {};
class Room {
 private:
        Bed kingBed;
}
class House {
 private:
        Room bedroom;
}


Construction order:
    1. kingBed's constructor
    2. bedroom's constructor
    3. house's constructor

Destruction order:
    1. house's destructor
    2. bedroom's destructor
    3. kingBed's destructor
```

Note: if a class does not have a default constructor, any class that contains it as a member must specify arguments for it using initializer lists.

Includes

1. Don't include .cpp files, only include .h files to avoid linker errors
2. Don't put "using namespace"s inside .h files as it forces .cpp files to use a certain namespace
3. Don't assume .h files will include other .h files
4. .h files containing class definitions only need to be included when you define a regular variable or use a variable of the class in the .h
    a. Using a class to define a parameter to a function, act as the return type for a function, or define a pointer/reference only requires a one line class definition

Preprocessor Directives

```
#define X

#ifdef THING
... (runs if THING was defined)
#endif

#ifndef THING
... (runs if THING was not defined)
#endif
```

Separate Compilation

Included .h files are simply copied to the top of the .cpp files that include them. If .h files include each other, we may end up with the same things being defined twice. Include guards fix this problem and need to be added to all .h files.

```
In xyz.h:
#ifndef XYZ_H
#define XYZ_H
...
#endif // XYZ_H
```

Linked Lists

A linked list is a data structure that has nodes that each hold values and a pointer to the next node.

```
struct Node {
        int value;
        Node* next;
        Node* prev;
};

class LinkedList {
 public:
        ...
 private:
        Node *head;
        Node *tail;
        Node dummy;
};
```

Basic Linked Lists: have a head pointer pointing to the first node and maybe a tail pointer pointing to the last node.

Singly Linked: nodes only have a next pointer, allowing for traversal in one direction only. The ending node's pointer points to nullptr.

Doubly Linked: nodes have both next and prev pointers which point to the next and previous nodes in the list respectively. The starting node's prev pointer and the ending node's next pointer point to nullptr.

Dummy Node: removes the head pointer and instead uses a node member variable. Allows each node to have a parent and thus removes special case code dealing with the head pointer.