CPP CHEAT SHEET:

TOC

- Key
- Fundamentals
  - Variables & Data
  - Operators, Conditionals, & Loops
  - Pointers, References, & Memory
  - Arrays & Strings
  - Functions
  - Structures
- Object Oriented Programming (OOP)
  - Classes & Objects
  - Encapsulation, Abstraction, Inheritance, & Polymorphism
- Standard Template Library (STL)
  - Containers
    - Sequential Containers: Arrays, Vectors, Deques, Lists, & Forward Lists
    - Container Adapters: Stacks, Queues, & Priority Queues
    - Associative Containers: Sets/Multisets, Maps/Multimaps
    - Unordered Containers: Unordered Set/Multiset, Unordered Map/Multimap
  - Algorithms
    - Sort, Copy, Max Element, Find (remove?), For Each
    - Range?, distance() (think for iterators)
  - Iterators
  - Functors: equal_to, not_equal_to, greater, less, plus, minus
- Miscellaneous
  - Compiling
  - Concepts
  - Directives
  - Errors
  - Exceptions
  - Files
  - I/O
  - Lambda Functions
  - Namespaces
  - Overloading
  - Templates : Pairs, tuples (include useful template section)
  - Typedefs

Compiler, linker, make-files, paths, shared libraries, using/creating libraries, debugger, timing code

Variables & Data:

```
//TYPES
    // primitive data types
    int             // 4 bytes
    float           // (6 decimal precision, 4 bytes memory, runs slower)
    double          // (15 decimal precision, 8 bytes memory, runs faster)
    char            // 1 byte
    wchar_t         // (2 or 4 byte char instead of 1 byte) (primitive in C++ only)
    void            //
    bool            // 1 byte (primitive in C++ only)

    // derived data types
    function
    array
    pointer
    reference

    // (abstract) user defined data types
    class           // (C++ only)
    struct
    union
    enum
    enum class      // (scoped enums in C++ only)
    typedef (int8_t, char16_t, uint32_t, etc.)

    // special (useful) types
    size_t          // (platform dependent size)
    std::string     // (size varies)
    std::nullptr_t  //

    // type modifiers
    signed/unsigned (int/uint)
    short/long, long long

    // type qualifiers
    const type my_variable;     // value cannot be changed after initialization
    constexpr type my_variable; // lets compiler evaluate value of var/func at compile time
    volatile type my_variable;  // tells compiler value of variable may change at any time...
                                // ...prevents optimizations and forces compiler to read back mem location
    restrict type my_variable;  // (C Only?)

    // storage class specifers
    auto type my_variable;      // let compiler deduce variable type (C++ only)
    register type my_variable;  //
    static type my_variable;    // (locally) declares var which holds value each invocation...
                                // ...(globally) dictates access control only in immediate script
    extern type my_variable;    // used to declare variable in one file but define it in another
    mutable type my_variable;
    thread_local type my_variable;
```

```
//DECLARE/INITIALIZE
    type my_variable;          // declare
    const type MY_VARIABLE;    // declare constant
    type my_variable = value;  // initialize
    type my_variable{value};   // initialize (alternate syntax)

//LITERALS (apply to all standard types)
    int my_int = 0X1A;              // int literals can be written in several number systems (bin, hex, oct, etc.)
    float my_float = 0.0f;          // f to specify float
    double my_double = 0.0ULL;      // ints and floating point nums can have opt type modifiers (ULL = unsigned long long)
    bool my_bool = true/false;      // true/false keyword for boolean literals
    int* my_int_ptr = nullptr/NULL; // nullptr keywords for unintialized ptrs

//CASTING
    my_var2 = (char)my_var1; // explicit conversions (less dangerous than implicit)
    my_var3 = (int)my_var2;  // char to int gives number of character location (ASC-II)

    /*----------------------------------------------*/// (C++ only)
    var = static_cast<type>(var);                    // simple compile-time cast
    ptr = reinterpret_cast<type*>(ptr);              // used to convert ptr of some type to ptr of another type
    ptr = dynamic_cast<BaseClass*>(&DerivedClass);  // run-time cast (Upcast or Downcast)
    ptr = const_cast<type*>(ptr);                    // used to remove const-ness of any object (ptr is of same type)
```

Operators:

```
++my_var/--my_var                             // preincrement/decrement operators
my_var++/my_var--                             // postincrement/decrement operators
+, -, *, /, %                                 // arithmetic operators
==, !=, >, <, >=, <=                          // relational/comparison operators
&&, ||, !                                     // logical/boolean operators
my_var1 = my_var2, my_var1 += my_var2, etc.   // assignment/compound operators (can do w/ any arithmetic operator)
&, |, ^, ~, <<, >>                            // bitwise operators (only for char/int data types)
sizeof(), ?:, &, ., ->, <<, >>                // other operators (here <<, >> are streaming operators (C++only))

Priority        Symbol
1               ++,  --,  ()
2               !, (typecast)
3               *,  /,  %
4               +,  -
5               <, <=, >, >=
6               ==, !=
7               &&
8               ||
9               all assignment operators (=, +=, etc.)
```

See also operator overloading

Conditionals:

```c
if (/*condition*/)      // brackets unrequired for one line statements
    // action;

if (/*condition1*/) {
    // action 1;
} else if (/*condition2*/) {
    // action 2;
} else {
    // action 3;
}

switch(/*comparable value*/) {
    case value1:
        // action 1;
        break;          // need break statement to avoid further actions!!!
    case value2:
        // action 2;
        break;
    default:
        // action 3;
        break;
}

/*condition*/ ? /*true_action*/ : /*false_action*/;                      // ternary operator (?)

int min = my_var1 < my_var2 ? my_var1 : my_var2;                        // ex: int min stores low value
my_var1 < my_var2 ? printf("%d\n", my_var1) : printf("%d\n", my_var2);  // ex: print low value
```

Loops:

```c
while (/*condition*/) {     // while loop
    // actions;
}

do {                        // do-while loop
    // actions;
} while (/*condition*/);

for (/*initialize counter*/, /*condition*/, /*change in counter*/)  // for loop: basic format

for (size_t i = 0; i < total; ++i) {                                // for loop: example
    // actions;
    break;          // leave loop
    continue;       // next iter
    goto /*label*/; // jump to label
}
```

>---EXCLUSIVE_TO_C++--

```cpp
type numbers[] = { ... };
for (auto num : numbers) {   // for-each loop example (auto deduces type from list)
    // actions;
}
```

---EXCLUSIVE_TO_C++--<

Arrays:

```
/*
Arrays are static, contiguous blocks of memory. Can contain any data type but must be
consistent. Must be careful not to access elements outside indices of an array, other
data is being stored there.

Access/Modify -> O(1)
Insert/Delete -> N/A
*/

type my_array[size];                          // create uninitialized array
type my_array[] = {val1, val2, val3, ...};   // create initialized array

my_array[idx];              // access value
my_array[idx] = new_value;  // modify value

type my_matrix[row_size][column_size];                        // create uninitialized matrix
type my_matrix[][column_size] = {{val1, val2}, {val1, val2}, ...};  // create initialized matrix

my_matrix[row_idx][col_idx];                // access value
my_matrix[row_idx][col_idx] = new_value;    // modify value

sizeof(my_array)/sizeof(my_array[0]);       // returns len of array
sizeof(my_matrix)/sizeof(my_matrix[0]);     // returns len of rows
sizeof(matrix[0])/sizeof(my_matrix[0][0]);  // returns len of cols
```

>---EXCLUSIVE_TO_C++---

```
std::size(my_array);        // returns len of array
std::size(my_matrix);       // returns len of rows
std::size(my_matrix[0]);    // returns len of cols
```

---EXCLUSIVE_TO_C++---<

Strings:

```
/* An array of chars */

    char str[] = {'h', 'e', 'l', 'l', 'o', '\0'};   // create str as initialized arr (must be initialized)
                                                    // \0 is null terminating char (i.e. end of arr)

    char str[] = "hello";   // create str as string literal (easier to type)
                            // null terminating char is implicit

    const char *ptr {"hello"};  // const char ptr can be initialzed with str literal
                                // ptr points to address of first char
                                // printing ptr will print str literal directly

    str[idx];               // access value (same as arrays)
    str[idx] = new_value;   // modify value (same as arrays)

// Character Manipulation:
    // C (#include <ctype.h>)
    isalnum(char)/isalpha(char)...etc;
    islower(char)/isupper(char);
    tolower(char)/toupper(char);

    // C++ (#include <cctype>) https://en.cppreference.com/w/cpp/header/cctype
    std::isalnum(char)/std::isalpha(char)...etc;
    std::islower(char)/isupper(char);
    std::tolower(char)/toupper(char);

// String Manipulation:
    // C (#include <string.h>)
    strlen(str);        // gives length of string (excluding null terminator)
    strcat(dst, src);   // concatenate (dst assumes new value of combined str)
    strcpy(dst, src);   // copy string (dst assume value of src) *dst len must be >= src len

    // C++ (#include <cstring>) https://en.cppreference.com/w/cpp/header/cstring
    std::strlen(str);
    std::strcat(dst, src);
    std::strcpy(dst, src);
```

>--EXCLUSIVE_TO_C++--

```
// String Type (#include <string>)
    std::string my_string;                      // create empty string
    std::string my_string {"Hello"};            // create initalized string
    std::string my_string1 {my_string2};        // initalize string with another string
    std::string my_string1 {my_string2, n};     // initalize string with part of another string (n = num of chars)
    std::string my_string1 {my_string2, i, n};  // initalize string with part of another string (i = starting index)
    std::string my_string {n, char};            // initalize string with n copies of a certain char
```

--EXCLUSIVE_TO_C++--<

Pointers:

```
/* A pointer is a special type of variable that stores the memory address of a regular variable.
It can store the address of any variable type (including other pointers). The address stored
is the address of the first byte of memory that stores a variable. Pointers are dangerous, be
cautious not to silently overwrite memory values in your program. */

type* my_ptr;   // pointer declaration
type *my_ptr;   // alternate syntax
type * my_ptr;  // alternate syntax
                // uninitialized pointer declaration is dangerous!
                // current address is NOT a valid location and can cause issues if referenced

type my_var;
type* my_ptr = &my_var;     // creating a pointer and assigning variable address
                            // (& = address-of operator, obtains address)
                            //  ptr stores memory address of variable (as hexadecimal)
                            //  ptr address can change but type must be consistent w/ variable type

type my_var = *my_ptr;      // assign data stored in memory pointed to by my_ptr
                            // (* = dereference operator, accesses data at address)

my_ptr += n;    // incrementing my_ptr to store address n <data type> sizes away
my_ptr -= n;    // decrementing "..."
                // ptrs can only do addition/subtraction with int (i.e. n = integer)
                // must be careful not to increment/decrement ptr beyond the bounds of memory space

type *my_ptr = NULL;        // initialize null pointer (C)
type *my_ptr = nullptr;     // initialize null pointer (C++)
                            // preferred to just declaring ptr which gives indeterminate value
                            // still dangerous, my_ptr points to "nowhere", can still cause issues if referenced

const type* my_ptr = &my_var;       // pointer to const int (my_var doesn't have to be const, but CAN be; can't modify my_var through ptr)
type const* my_ptr = &my_var;       // alternate syntax
const type* const my_ptr = &my_var; // const pointer to const int (my_ptr is const, my_var CAN be const; still can't modify my_var through ptr)
type const* const my_ptr = &my_var; // alternate syntax
```

>--EXCLUSIVE_TO_C++--

```
// Smart Pointers:   (#include <memory.h>)
/* Class that manages pointers through RAII, allows for auto freeing and for pointers to be exception safe */

std::unique_ptr<type>    // does not share ownership, will free resource at end of scope (eos)
std::shared_ptr<type>    // shares ownership, frees resource if no other owners counted at eos
std::weak_ptr<type>      // used with shared ptr, does not add to reference counter
auto ptr = std::make_unique<type>(args);    // function to create unique pointer
auto ptr = std::make_shared<type>(args);    // function to create shared pointer
```

References:

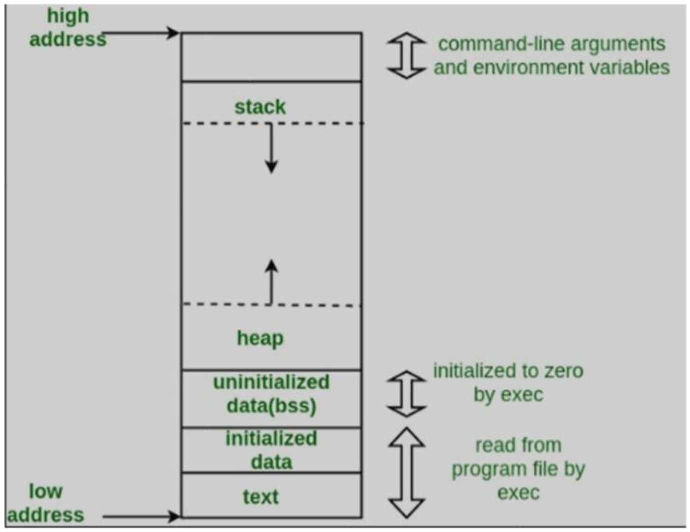
```
// REFERENCES:

    /* References are aliases for other variables that can modify the data of the variable being referred to.
    They will inherit the same address as the variable they refer to. They can store a reference to any variable
    type (including other references). They can be thought of as const pointers (less dangerous than actual pointers) */

    type &my_var1 = my_var2;    // assigning a reference, my_var is an alias for my_var2 and cannot be reassigned
    type& my_var1 = my_var2;    // alternate syntax
    type & my_var1 = my_var2;   // alternate syntax
                                // (& = reference operator, establishes my_var as ref for my_var2)

    type&& my_var = 0;          // rvalue reference

    my_var1 = my_var3;          // modifying a reference (my_var) will modify the target (my_var2) as well

    const &my_var1 = my_var2;               // const ref, can't modify original variable through ref
    type my_function(type const &my_var)    // const ref = computational efficiency w/out modifying data
    {
        return 0;
    }
```

--EXCLUSIVE_TO_C++--<

```
/* Pointers vs References
    pointers:
    1) must go through dereference operator to read/write through pointed to value
    2) can be chnaged to point elsewhere
    3) can be declared unitialized (w/ garbage address)

    references:
    1) doesn't use dereferencing for read/write
    2) can't be changed to reference something else
    3) must be initialized on declaration
/*
```

Memory:

```
/* program memory is generally organized into a few main categories: text, data, bss, heap, and stack
   two of primary importance in C/C++ programming are the stack and heap

     o   stack: highly ordered section of memory stored in a LIFO structure (used as scratch spaced for thread execution)
     o   heap: relatively unordered section of memory used for dynamic allocation

 Stack vs Heap
     stack:
     1) variables created on stack are automatically deallocated when they go out of scope
     2) much faster to allocate memory compared to heap
     3) data created on stack can be used with/without pointers
     4) typically has a max size determined when program starts
     5) each thread gets its own stack (stack frame)
     6) risk of stack overflow causing program to crash (typically caused by heavy recursion or enormous allocations)
     7) best to use stack if you know exactly how much data you need to allocate before compile time (and it is not too large!)

         Stores local data, return addresses, used for parameter passing

     heap:
     1) variables allocated on heap must be destroyed manually and never fall out of scope (otherwise there will be a memory leak)
     2) slower to allocate memory compared to stack
     3) data created on heap will be pointed to by pointers
     4) memory can be added to heap by OS if required
     5) typically only one heap is shared by an application (even multi-threaded apps)
     6) risk of fragmentation with a lot of allocations/deallocations
        risk of heap overflow (typically caused from memory leak or enormous allocations)
     7) best to use heap if you don't know exactly how much data you will need at run time or if you need to allocate a lot of data

 Storage
     Text: stores executable instructions from program to be run
     Data: contains all initialized global and static variables
     BSS: contains all uninitialized global and static variables
     Stack: stores all local (automatic) variables
     Heap: stores all user allocated data
*/
```




```
// Dynamic Memory Allocation/Deallocation

    // C (#include <stdlib.h>)
    type *ptr = (type*) malloc(size);           // reserve as many bytes as desired on heap
    type *ptr = (type*) calloc(n,size);         // reserve contiguous memory for n elements of any data type
    type *ptr = (type*) realloc(ptr,size);      // expand/contract block of reserved memory
    free(ptr);                                  // release previously allocated memory

    // C++
    type *ptr = new type;           // dynamically allocate memory
    type *ptr = new type(value);    // allocate and initialize (NOTE: need constructor for user-defined data)
    delete ptr;                     // delete memory
    ptr = nullptr;                  // good practice to reset memory to nullptr after releasing it
                                    // also good practice to check for valid address (i.e. not null) before using ptr

    type *array_ptr = new type[size]{values};   // create dynamic array (different from standard array! some array operations won't work!)
    delete[] array_ptr;                         // delete dynamic array
    array_ptr = nullptr;                        // reset dynamic array ptr

    type *ptr = new(std::nothrow) type{};       // std::nothrow returns a nullptr instead of throwing exception
```

Functions:

```cpp
// Declaring/Defining
type my_function(type param1, type param2);         // prototype/declaration (typically found in header files)

type my_function(type param1, type param2=val) {    // function definition (default arguments for parameters are optional)
    /*actions*/
    return /*output*/;
}

void my_function(void) {                            // fxn def w/ no input or return (optional to specify void input)
    return;
}

// Input Parameters
type my_function(type param) {/*function def*/}         // pass by value

type my_function(type *param) {/*function def*/}        // pass by pointer

type my_function(type &param) {/*function def*/}        // pass by reference (use const ref to avoid data manipulation)

type my_function(type my_array[]) {/*function def*/}    // passing arrays by value will decay to pointers (my_array[] = *my_array)
                                                        // (can avoid by passing by ref)

// Function Specifiers
inline type my_function() {}        // instructs compiler to isnert fxn body where called; can be beneficial for speed (sometimes)
virtual type my_function() {}       // declares member function as virtual, allowing it to be overriden in derived classes
type my_function() override {}      // specifies that a member function is intended to override a base class function
virtual type my_function() final {} // prevents a virtual function from being overridden in derived classes
explicit my_function() {}           // specifies that a constructor or conversion operator should not be implicitly invoked
static type my_function() {}        // specifies that a member fxn doesn't operate on instance of class & can be called w/out obj
constexpr type my_function() {}     // indicates that the func can produce a constant expression & can be evaluated @ compile time
friend type my_function() {}        // allows a non-member function to access the private and protected members of the class

// Passing Arguments
my_function(arg1, arg2);                        // passing args to function
my_variable = my_function(*arg1, &arg2);        // passing args and storing result in variable
                                                // args passed to func can be literals, variables, pointers, addresses, references, etc.
```

```cpp
// Passing Arguments
/*
    By Value:
        1) copies of arguments passed are made and stored in params
        2) generally runs slower for complex objects but faster for simple objects (ie primitive types) (overhead is copying)
        3) values of args CANNOT be modified

   By Pointer:
        1) no copies of arguments need to be made (memory efficient)
        2) generally runs faster for complex objects but slower for simple objects (overhead is dereferencing)
        3) values of args CAN be modified

   By Reference:
        1) no copies of arguments need to be made (memory efficient)
        2) generally runs faster for complex objects but slower for simple objects (overhead is accessing)
        3) values of args CAN be modified

    Ptr vs Ref
        both are similar with similar computational efficiency
        generally passing by ref is more straightforward and preferred
        however passing by ptr can be useful in situations where it is useful to pass a NULL arg
*/

// Returning Values
/*
    It is okay to return a ref or ptr from a func if neccesary
    If return val could be NULL or will be dynamic memory address, better to use ptr (specifically smart pointer in C++)
    Otherwise it is generally more common to return a ref
    Returning a ref can be useful to avoid copying of more complex objects
    However, generally won't serve much purpose due to Copy Elision
    CAREFUL not to return a reference to a local variable (because it goes out of scope)
*/
```

See also function overloading, lambda functions, functors, and function templates (stdlib functions)

Structures:

```
/* Allow you to collect many different data types into one derived data type */

struct myStruct {       // define struct

    char* str;
    int x;
    struct otherStruct;
    etc...              // variables should be declared, not initialized

};

struct myStruct struct1;                        // declare struct of type myStruct with name struct1

struct myStruct struct1[n];                     // declare struct array of size n

struct myStruct struct1 = {"hello", 1, ...};    // initialize (ordered)

struct myStruct struct1 = {                     // initialize (unordered)

    .x = 0,
    .str = "hello",
    etc...              // don't need to initialize member variables in a struct immediately (can also just do a few)

};                      // myStruct is the "type" of structure and struct1 is the initialized structure


struct1.x = 1;                  // dot operator
struct1.str = "goodbye";        // access/modify member variables


struct myStruct* structPointer = &struct1;  // struct pointer

(*structPointer).x;                         // access/modify member variables through derefenced struct pointer
structPointer->str;                         // alternate syntax
```

>---EXCLUSIVE_TO_C++---

Classes and Objects:

```
// CLASSES & OBJECTS

/* A class is a user-defined data type and a template for objects. An object is an instance of a class.

    Classes hold data members and member functions which can be accessed by objects of the class

    Classes are built in .hpp files (including member function declaration), but member functions
    (including constructor/destructor) should be defined in the related .cpp file, although they
    can be defined inside the class as well
*/
// Defining Classes (in .hpp file)
    class MyClass {

        type my_attribute;                      // class attribute (private scope, class members are private by default)

        public:                                 // access specifier (scope is available to all code)

            MyClass(type param=default_val, ...);   // declare constructor(s)

            ~MyClass();                         // declare destructor

            MyClass(MyClass& oldObj);           // declare copy constructor
            MyClass& operator=(MyClass&);       // declare copy assignment operator

            MyClass(MyClass&& myObj);           // declare move constructor
            MyClass& operator=(MyClass&&);      // declare move assignment operator

            type& operator[](type);             // declare other operator functions (ex.)

            type getValue();                    // declare accessor function
            void setValue(type new_variable);   // declare mutator function

            type my_function(type param);       // declare member function

            type my_variable;                   // declare data member (class attribute)

        protected:                              // access specifier (scope is within class and subclasses)

            // protected code;

        private:                                // access specifier (scope is within class only)
                                                // ...for explicitly defining more private material under public scope
            // private code;

    };
```

```cpp
/*
    All member functions (including constructors/destructors) created inside class are inline by default,
    functions defined outside require inline specifier
*/
// Constructors (in .cpp file)
    /* called by compiler everytime an object of the class is instantiated */

    MyClass::MyClass() {                        // default constructor (provided by compiler implicitly if no constructors defined)
        // constructor code                                         // (best practice to define explicitly)
    }

    MyClass::MyClass(param, ...)                // parameterized constructor (can provide default args to params so it acts as default constructor as well)
        : my_attribute(param)... {}             // member initializer list (must use for const and reference attributes!)...
                                                // ...also best to use for init class members (especially for classes with heavy constructors)

    MyClass::MyClass(param, ...) {              // alternate definition of parameterized constructor
        my_attribute = param;
    }

    MyClass::MyClass(const MyClass& myObj) {    // copy constructor (compiler creates implicit definition if no explicit definition)
        ...                                     // ... typically needed when an object contains pointers or non-shareable refs
        my_variable = myObj.my_variable;
    }

    MyClass::MyClass(MyClass&& myObj) {         // move constructor (compiler creates implicit definition if no explicit definition)
        ...                                     // ... useful for efficiently transferring ownership of resources (less overhead)
        my_variable = myObj.my_variable;
    }
// Destructors (in .cpp file)
    /*
        used to destroy (release memory) objects of a class

        called by compiler if...
        1) object out of scope
        2) object explicitly deleted
        3) program ends

        cannot be overloaded
        cannot be declared static or const
        does not take any inputs or return anything
        called in reverse order of constructor invocations

        generally if constructor allocates memory on the heap or class contains some other dynamically allocated memory,
        destructor should be defined to free that memory
    */

    MyClass::~MyClass() {   // define destructor (compiler creates implicit definition if no explicit definition)
        // destructor code
    }
// Operators (in .cpp file)
    MyClass& MyClass::operator=(MyClass& oldObj) {          // copy assignment operator
        //
    }

    MyClass& MyClass::operator=(MyClass&& myObj) noexcept {  // move assignment operator
        delete my_variable;
        my_variable = myObj.my_variable;
        myObj.my_variable = nullptr;
        return *this;
    }

                                                            // other operators (ex.)
/*
    Rule of 5: if one of the following special functions is created in a class, all should be created in the class
    Destructor, Copy constructor, Copy assignment operator, Move constructor, Move assignment operator
*/
```

```cpp
// Defining Member Functions (in .cpp file)
    type MyClass::getValue() {                  // accessor function
        return my_variable;
    }

    void MyClass::setValue(type new_variable) { // mutator function
        my_variable = new_variable;
    }

    type MyClass::my_function(param) {          // define member function
        return my_attribute;                    // class attributes are in scope for member functions
    }
```

```cpp
// Defining SubClass
    class SubClass: public BaseClass {  // Inheritance (subclass or derived class inherits from superclass or baseclass)
        // class code
    };
```

```cpp
// Create and Manipulate Objects (and Class Attributes)
        MyClass myObject;                         // instantiate object of class (must have valid default constructor!)
        MyClass myObject(args);                   // instantiate object with attributes (implicitly)
        MyClass myObject = MyClass(args);         // instantiate object with attributes (explicitly)

        MyClass newObject(myObject);              // instantiate object from another object (copy)
        MyClass newObject = myObject;             // instantiate object from another object (copy, alternate syntax)
        MyClass newObject(std::move(myObject));   // instantiate object from another object (move)

        MyClass.my_attribute = 0;                 // assign attribute value

        myObject.my_variable = 0;                 // modify data member of object
        myObject.my_function();                   // call function of class on object

        myObject.~MyClass();                      // destroy object
```

```cpp
// OOP Concepts
    /*
       Encapsulation – bundling related data/functions within limited data scope (consolidate)

       Abstraction – hiding complex logic and making code more accessible (simplify)

       Inheritance – classes inheriting attributes and functions of other classes (reuse)

       Polymorphism – same entities can operate differently under different situations (interchange)
    */
```

See also class templates

--EXCLUSIVE_TO_C++--<

Containers:

>---EXCLUSIVE_TO_C++--

Vectors:

```
/*
Vectors are dynamic arrays

Support random access iterators

Access/Modify -> O(1)
Insert/Delete -> O(n)
*/

// #include <vector>

std::vector<type> my_vec;                          // declare vector (type can't be changed after declaration)
std::vector<type> my_vec(size);                    // declare with pre-allocated size
std::vector<type> my_vec(size, val1);              // declare with pre-allocated size & initialize w/ some value
std::vector<type> my_vec({ val1, val2, val3...});  // declare and initialize with multiple values
std::vector<type> my_vec = {val1, val2, val3};     // alternate syntax
std::vector<type> my_vec(other_vec);               // initialize with another vector

my_vec[idx];    // access value at index

// Member Functions
    // Iterators
    my_vec.begin(); // returns an iterator pointing to first element in vec
    my_vec.end();   // returns an iterator pointing to one past last element in vec

    // Capacity
    my_vec.size();          // returns number of elements in vec
    my_vec.capacity();      // returns number of elements allocated to vec
    my_vec.resize(n);       // resizes vec so that it contains n elements
    my_vec.empty();         // returns whether vec is empty
    my_vec.shrink_to_fit(); // reduces vec capacity to fit size (destroys elements beyond capacity)
    my_vec.reserve(n);      // requests vec capacity to be at least enough to contain n elements

    // Access
    my_vec.at(idx); // returns reference to element at idx (safer access method, throws error if idx out of bounds)
    my_vec.front(); // returns reference to first element of vector
    my_vec.back();  // returns reference to last element of vector

    // Modifiers
    my_vec.assign(size,val);    // assigns new value to vector elements
    my_vec.pop_back();          // removes element from end of vector (no return value)
    my_vec.push_back(val);      // adds element to end of vector (temporary obj creation)
    my_vec.emplace_back(val);   // adds element to end of vector (direct obj creation)
    my_vec.emplace(it,val);     // extends vector by placing new element at position (by construction)
    my_vec.insert(it,val);      // inserts new element(s) before specified position (by copy/move)
    my_vec.erase(it);           // removes elements from specified position/range
    my_vec.clear();             // removes all elements of vector
```

Stacks:

“Container that stores elements in LIFO order. Implemented as container adaptor (a class that uses another container class as its underlying container. The underlying container class can be vector, deque, or list. Default is deque.”

std::stack<*type*> stackName; // initiate stack

std::stack<*type*, *containerType*<*type*>> stackName; // initiate stack w/ specified container *type* must be same, deque is default container type

Useful member functions:

.empty() // returns true if no elements

.pop() // removes last item added to top of stack

.push() // adds element to top of stack

.size() // returns number of elements in stack

.top() // returns element on top of stack

Queues:

“container adaptors that store elements in a FIFO order”

queue<*type*> queueName; // create queue

Useful member functions: // must include <queue> library

.empty() // checks if queue is empty

.front() // returns next element within queue

.pop() // renoves element at front of queue

.push() // adds element at back of queue

.size() // returns number of elements in queue

Sets:

"associative containers which store unique elements that can be referenced by an element's value. Values are constant once assigned. Existing values can be removed or new values can be added. Values of set are sorted in ascending order."

std::set<*type*> setName; // initiate a set

std::set<*type*, std::greater<*type*>> // initiate set with values in descending order (changed comparison function)

Useful member functions:

.clear() // removes all values from a set

.erase() // removes a single value from a set

.insert() // inserts a single value into a set

Maps:

"data structure that stores a collection of elements formed by a combination of a key value and mapped value"

std::map<*keydatatype*, *valuedatatype*> mapName; // creates empty map

std::map<*keydatatype*, *valuedatatype*> mapName = {{key1, value1}, key2, value2}, …};

mapName[key] // access elements of map

"comparison function sorts elements by keys in ascending order by default"

std::map<*keydatatype*, *valuedatatype*, std::greater<*keydatatype*>> mapName;

// sort elements in descending order

Useful Member Functions: // include <map> or <unordered_map>

.clear() – removes all elements from map

.erase() – removes an element by key from map

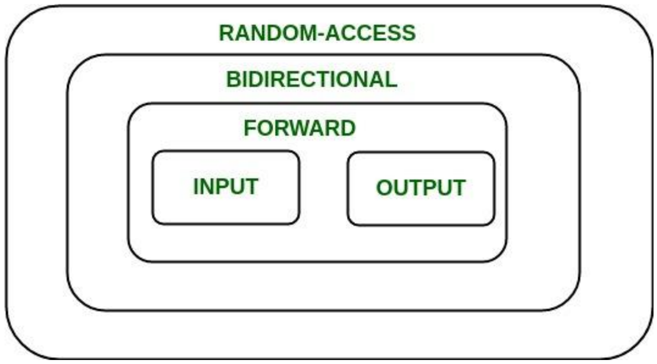
.insert() – inserts a key-value pair into map

---EXCLUSIVE_TO_C++---<

Iterators:

>---EXCLUSIVE_TO_C++--

```
/*
Iterators are a concept, not a concrete or abstract type,
but any type that obeys iterator like rules. They can be
thought of as abstractions of a pointer.

Iterators are primarily used to move through the contents of containers

The major advantages of using iterators are that they
    1) bring you closer to container independence
    2) allow for efficient dynamic processing of containers

Iterator Types:
    Contiguous          ^
    Random Access       |
    Bidirectional       |   powerfulness
    Forward             |
    Output              |
    Input              ---

Iterators point to memory addresses inside a container (similar to a pointer).

DEPENDING ON THE TYPE of iterator, they can potentially be dereferenced, used
with increment/decrement operators, arithmetic operators, or relational operators
for various purposes
*/

// #include <iterator>

// Declaration
std::container_type<type>::iterator it;   // declare iterator called it

// Dereference
*it;        // dereference iterator (same as pointer)
it->member; // accessing member element called "member" (same as pointer)
            // dereferenced iterator may be used as rvalue or lvalue depending on iterator type

// Iterator Operations
my_container.begin();   // returns an iterator pointing to first element in container
my_container.end();     // returns an iterator pointing to one past last element in container

std::advance(it,n);             // increments given iterator by n elements
std::next(it,n);                // returns the nth successor of iterator
std::prev(it,n);                // returns the nth predecessor of iterator
std::inserter(my_container,it); // returns insert_iterator which can be used to insert elements
std::copy(s_it,e_it,d_it);      // copies elements in source range to destination range
```



ITERATORS	PROPERTIES				
	ACCESS	READ	WRITE	ITERATE	COMPARE
Input	->	= *i		++	==, !=
Output			*i=	++	
Forward	->	= *i	*i=	++	==, !=
Bidirectional		= *i	*i=	++, --	==, !=,
Random-Access	->,[ ]	= *i	*i=	++, --, +=, -==, + ,-	==, !=, <,>,<=,>=

CONTAINER	TYPES OF ITERATOR SUPPORTED
Vector	Random-Access
List	Bidirectional
Deque	Random-Access
Map	Bidirectional
Multimap	Bidirectional
Set	Bidirectional
Multiset	Bidirectional
Stack	No iterator Supported
Queue	No iterator Supported
Priority-Queue	No iterator Supported

https://en.cppreference.com/w/cpp/iterator

---EXCLUSIVE_TO_C++---<

Functors:

>--EXCLUSIVE_TO_C++---

"Object or Struct that can be called like a function by overloading function call operator '( )' "

```
class MyClass {
        public:
                type operator( )(args) {            // defining a functor
                        // function body
                }
}

MyClass myClass;          // instantiate object
myclass(args);            // calling a functor
```

--EXCLUSIVE_TO_C++--<

Miscellaneous---

Compiling:

(note: these are all terminal commands)

gcc *script*.c	// compile command (for gcc compiler) in C
gcc *script*.c -o *scriptname*	// compile and name output file in C
g++ *script*.cpp -o *scriptname*	// compile and name output file in C++
./*scriptname*.out	// run script
g++ *script1*.cpp *script2*.cpp	// link & compile multiple files

[Concepts](#):

```cpp
/*
    Concepts are a mechanism to place constraints on template type parameters
*/

// #include <concepts>

    // Standard Concepts (examples)
    std::integral           // Core Language Concept | specifies type is integral type
    std::floating_point     // Core Language Concept | specifies type is floating point type
    std::boolean-testable   // Comparison Concept | specifies type can be used in Boolean contexts
    std::moveable           // Object Concept | specifies object can be moved and swapped
    std::copyable           // Object Concept | specifiess object can be copied, moved, and swapped
                            // Callable Concept |

    // Custom Concepts (creation)
    template <typename T>                                   // syntax
    concept MyConcept = /*requirements*/;

    template <typename T>                                   // example
    concept IsIntegral = std::is_integral_v<T>;

    template <typename T>                                   // syntax
    concept MyConcept = requires(T param, ..) {
        /*statement(s)*/
    };

    template <typename T>                                   // example
    concept Multipliable = requires(T a, T b) {
        a * b;                                              // simple requirement : only checks syntax!
    };

    template <typename T>                                   // example
    concept Multipliable = requires(T a, T b) {
        (a * b) != 0;
        requires (a * b) != 0;                              // nested requirement : checks condition
    };

    template <typename T>                                   // example
    concept Multipliable = requires(T a, T b) {
        {a * b} noexcept -> std::convertible_to<int>;       // compound requirement : checks syntax and result requirement
    };                                                      // (noexcept is optional)
```

```cpp
// Constraints (application)
template <typename T>                                                   // syntax 1
requires /*concept-name*/<T>
type my_function(type param) {}

template </*concept-name*/ T>                                           // syntax 2
type my_function(type param) {}

/*concept-name*/ auto my_function(/*concept-name*/ auto param) {}       // syntax 3

template <typename T>                                                   // syntax 4
type my_function(type param) requires /*concept-name*/<T> {}

template <typename T>                                                   // concept combination
requires /*concept-name*/<T> && /*concept-name*/<T> || /*concept-name*/<T>
type my_function(type param) {}

/*concept-name*/ auto my_variable = my_function(arg);                   // variable constraint
```

(Pre-Processor) Directives:

Macros

```
#define MYMACRO                // define object-like macro (note: no semicolon)
#define ADD(a, b)  (a + b)     // define function-like macro
#undef MYMACRO                 //undefine macro
// Predefined macros
__DATE__      // current date formatted as MMM DD YYYY
__TIME__      // current time formatted as HH:MM:SS
__FILE__      // current filename
__LINE__      // current line number
UINTMAX, etc...
```

Conditionals

```
#if / #ifdef / #ifndef         // pre-processor conditionals
#elif
#else
#endif
```

Other:

```
#import
#include
#line
#pragma
#using
```

Errors:

```
/*
    compile-time errors

        // typically syntax or semantics
        ex. syntax errors, type errors

    link-time errors

        // occur when trying to combine files into exe (link)
        ex. incorrect header files, can’t find function/library, etc.

    run-time errors

        // code has run issues due to requests comp can’t handle
        ex. div by zero

    logic errors

        // incorrect logic in code produces wrong result
*/
```

Exceptions:

>--EXCLUSIVE_TO_C++---

```cpp
// EXCEPTION HANDLING:

    try {
        // code to try running

        if (/*condition*/) {                                // fail condition checks
            throw 0;                                        // throws error of type int
        } else if (/*condition*/) {
            throw std::runtime_error("text to display");    // throws runtime error with text
        } else if (/*condition*/) {
            throw;                                          // throws error and stops program
        }
    }

    catch (int int_error_variable) {
        // code to run if error of type int
    }

    catch (std::exception exception_variable) { // catches exceptions of type std::exception and saves them to error object under exception_variable
        // code to run if error of type exception
        std::cout << exception_variable.what(); // prints what the exception was
    }

    catch (...) {                               // ellipses catches any and all exceptions (catch all)
        // code to run if error of any type
    }
```

--EXCLUSIVE_TO_C++---<

Files:

#include <iostream>

#include <fstream>

classes: ofstream (output file stream), ifstream (input file stream), fstream (combo)

```
std::ofstream MyFile("file.txt");        // create and open text file
MyFile << "hello world";                 // write to file
MyFile.close();                          // close file
std::ifstream MyFile("file.txt");        // read from file
std::string myString;                    // create string
getline(MyFile, myString);               // output line of file to string
```

File types: .cc, .cpp, .tpp, .ipp, .hpp, .c, .h, .anythinguwant

file types are mostly a matter of indication of the contents of the file

only source files (i.e. .cpp) are compiled

certain compilers may need correct file extensions?

I/O:

printf("string to display", [list of optional parameters])

sprintf()

scanf()

symbol	**type**		**symbol**	**effect**
%d or %i	int		\n	newline
%f	double or float		\r	carriage return
%c	char		\t	tab
%s	string			
%p	pointer			

>---EXCLUSIVE_TO_C++---

#include <ios>,

std::cout << “HelloWorld\n”;   // print to terminal

std::cout << variable << endl;   // print variable to terminal

std::cin >> variable;   // request user input and assign to variable

#include <ios>, #include <iomanip>

std::endl

std::flush

etc…

---EXCLUSIVE_TO_C++---<

Lambda Functions:

>--EXCLUSIVE_TO_C++---

--EXCLUSIVE_TO_C++--<

Namespaces:

>---EXCLUSIVE_TO_C++---

```cpp
// NAMESPACES:

/* Namespaces define the space or context in which identifiers (i.e. variable, function, class)
   are defined to avoid naming collisions

    namespace declarations only appear at global scope
    namespace declarations can be nested within another namespace
    namespace definition can be split over several units (important!)
*/

    // Define
    namespace my_namespace              // define namespace
    {
        type my_variable;

        type my_function(type param) {
            /*function def*/
        }

        class MyClass {
            /*class def*/
        };
    }

    namespace my_namespace              // extending definition (NOT new namespace)
    {
        class MyNewClass {
            /*class def*/
        };
    }

    namespace                           // unnamed (anonymous) namespace
    {
        type my_variable;
    }                                   // ... used for creating unique identifiers for a given translation unit
                                        // ... (superior replacement for static variables/functions; also allow for classes)
                                        // ... compiler generates unique name for namespace
// Access
my_namespace::my_variable;          // scope resolution operator to access identifiers in namespace
my_namespace::my_function(arg);
my_namespace::MyClass myObject;

// Using Directive
using namespace my_namespace;       // tells compiler following code is using specified namespace (Dangerous!)
using my_namespace::my_variable;    // using my_variable from specified namespace

// Aliasing
namespace my_ns = my_namespace;     // my_ns now refers to my_namespace

// Inline Namespaces
namespace my_namespace1
{
    inline namespace my_namespace2  // inline allows identifiers of nested namespace to belong to parent/enclosing namespace
    {
        type my_variable;
    }
}
```

---EXCLUSIVE_TO_C++---<

Overloading:

>---EXCLUSIVE_TO_C++---

Operator Overloading

- Operator **overloading** allows for re-defining of built-in operators as user-defined classes.
- Compiler will determine correct definition of overloaded operator to use based on args
- ? (ternary), :: (scope resolution), and . *or* .* (member access) operators CANNOT be redefine
- Can only be overloaded within Classes

class className {

public:

returnType operator *symbol* (*args*) {

// redefine

}

}

Function Overloading

```
/* Function overloading allows for more than one definition of a function in the same scope */

// Example:
int my_same_function(int x, int y) {
    /* actions */
}

float my_same_function(float x, float y) {
    /* actions */
}

/* Overloaded function will share the same name, however parameters must be different types OR
there must be different number of parameters (or both). Return type can be same only if one or
both of the previous conditions are true. */
```

--EXCLUSIVE_TO_C++---<

Templates:

>---EXCLUSIVE_TO_C++---

Function Templates

```
// Functions
/* A function template is a tool that allows the addition of data types as parameters.
   The compiler uses the template as a blueprint to generate instructions for a function (template instances).

    Templates are created entirely in header files
    Templates allow selection of type implementation (both I/O) when called
    Templates will generally slow down compile time but speed up execution time
    Templates serve to eliminate the need to overload a function repetitively, however they can be overloaded as well!
*/

    // Defining
    template <typename T>                        // function template syntax (T = typeidentifier)
    T my_function(T param1, T param2) {          // function with same parameter and return type "T"
        /* function def */
    }

    template <class T>                           // use of 'class' or 'typename' is arbitrary
    template <class T1, class T2>                // multiple type identifiers are permitted

    const T& my_function(const T& param) {}      // can use references as always in function signature

    template <>                                  // template specialization (for specific inputs)
    const char* my_function<const char*> (const char* param) {
        /* function def */
    }

    // Calling
    my_function (arg1, arg2);                    // template type deduction (compiler deduces type based on arg(s))
    my_function <type> (arg1, arg2);             // explicit type arg (allows implicit conversion of args)
    my_function <type1, type2> (arg1, arg2);     // must use multiple type params for templates w/ multiple identifiers
```

Class Templates

```
// Classes
/* Class templates are classes defined along with a generic type that can be applied throughout the class definition

    Templates are typically defined in header files,
    but member functions are implemented in .tpp files (with .hpp files included at the end)
*/

    template <class typeidentifier> class MyClass {     // class template
        // class def
    };

    MyClass <type> myObject(params);                    // instantiation
```

---EXCLUSIVE_TO_C++---<

Typedefs:

C main script template:

```
# include <header.h>                                   // standard libraries
# include “header.h”                                   // project files

# define alias value                                   // macros
# define alias calculation

typedef dataType alias                                 // type aliases
typedef returnType alias(paramType);

type my_function (param1Type, param2Type, etc.);       // prototypes
…
…

varType varName;                                       // global variables
static varType varName;                                // static global variables
…
…

int main(int argc, char** argv) {                      // main function
}

void my_function(void) {                               // other functions
}
```