Notes for Classes and Functions

All examples with code and comments highlighted in the directory `./lessons`

Link to code Examples for Lab Sessions

Link to code Examples for Exercises

Link to code Examples for Classes

Link to code Examples for Functions

Variables

> local and global variables. local variables when initialized, get assigned some random values which unlike the case in global ones are assigned ZEROS.
> so it is better to assign null or ZEROS to your integral initialized or objects. `local` and `function arguments` (if not referenced) are stored on the `stack` while `global` and `static variables` are stored on the `heap` even if the static variable is declared in a local scope.

Scope:

> Variables possess a characteristic called the storage class - `storage class is how long a variable persists in memory or it's lifetime`. The most common storage class is automatic. Local variables have the automatic storage class: they exist only while the function in which they are defined is executing. They are also visible only within that function.

> Global variables have static storage class: they exist for the life of a program. They are also visible throughout an entire file. This is why it is preferred for you to make your global variables `const`. Static local variables exist for the life of a program but are visible only in their own function.

Functions

> Functions provide a way to help organize programs, and to reduce program size, by giving a block of code a name and allowing it to be executed from other parts of the program. Function declarations (prototypes) specify what the function looks like, function calls transfer control to the function, and function definitions contain the statements that make up the function. The function declarator is the first line of the definition.

Functions with Arguments:

> Arguments can be sent to functions either by value, where the function works with a copy of the argument, or by reference, where the function works with the original argument in the call- ing program.

- by value

```
double magnitude_from_origin(Distance ref) {
    return (ref.x * ref.x) + (ref.y * ref.y);
}
```

- by reference (take note of the argument)

```
double magnitude_from_origin(Distance &ref) {
    return (ref.x * ref.x) + (ref.y * ref.y);
}
```

Default Arguments:

> If a function uses default arguments, calls to it need not include all the arguments shown in the declaration. Default values supplied by the function are used for the missing arguments.

```
void replchar(const char chin='/', int ntimes=3) {
  for (int i = 0; i < ntimes; ++i) std::cout << chin;
}
```

Typed functions:

> Functions can return only one value. Functions ordinarily return by value, but they can also return by reference, which allows the function call to be used on the left side of an assignment statement. Arguments and return values can be either simple data types or structures.

```
Distance NullDistance;  // Reference objects are made global so that you can perform function pointers with them
Distance &reset_distance() {
  return NullDistance;  // references whatever values NullDistance has
}
```

Overloading Functions:

> An overloaded function is actually a group of functions with the same name. Which of them is executed when the function is called depends on the type and number of arguments supplied in the call.

```
double sum_two_numbers(int A, int B) {
  return A + B;
}

double sum_two_numbers(float A, float B) {
  return A + B;
}
// sum_two_numbers has been overloaded and differs depending on the type of arguments supplied. This is possible because of function signatures.
```

The inline Keyword:

> An inline function looks like a normal function in the source file but inserts the function's code directly into the calling program. Inline functions execute faster but may require more memory than normal functions unless they are very small. when you call an inline function, the body of the function or usually the return part of the inline function get copied and pasted whereever it is used.

```
inline double area_of_circle(double radius) {
  return (3.14159 * (radius * radius));
}
```

The const Keyword in function arguments:

> A function cannot modify any of its arguments that are given the const modifier. A variable already defined as const in the calling program must be passed as a const argument.

Classes and Structs

Class definition:

> A class is a specification or blueprint for a number of objects. Objects consist of both data and functions that operate on that data. In a class definition, the members—whether data or functions— can be private, meaning they can be accessed only by member functions of that class, or public, meaning they can be accessed by any function in the program.

```
class Human {
 public:
  std::string name = "";
  unsigned int age;

  Human(std::string name, unsigned int age) : name(name), age(age) {}
  Human() : name("CPP-1234"), age(0) {}

  void howOldAmI() { std::cout << age; }
  void whatIsMyName() { std::cout << name; }
};
```

Member functions or Methods:

> A member function is a function that is a member of a class. Member functions have access to an object's private data, while non-member functions do not.

```
  void howOldAmI() { std::cout << age; }
  void whatIsMyName() { std::cout << name; }
```

Constructors & constructor overloading:

> A constructor is a member function, with the same name as its class, that is executed every time an object of the class is created. A constructor has no return type but can take arguments. It is often used to give initial values to object data members. Constructors can be overloaded, so an object can be initialized in different ways. **we cannot use default arguments to make constructors**

```
  Human(std::string name, unsigned int age) : name(name), age(age) {}
  Human() : name("CPP-1234"), age(0) {}
```

Destructor:

> A destructor is a member function with the same name as its class but preceded by a tilde (~). It is called when an object is destroyed. A destructor takes no arguments and has no return value.

```
class Pixels {
  int bitmap[256];

  public:
    ~Pixels() {
      free(bitmap); // frees the memory for the values for the pixels
    }
};
```

Shared attributes of a class with Static:

> In the computer's memory there is a separate copy of the data members for each object that is created from a class, but there is only one copy of a class's member functions. You can restrict a data item to a single instance for all objects of a class by making it static.

Modelling the world with OOP:

> One reason to use OOP is the close correspondence between real-world objects and OOP classes. Deciding what objects and classes to use in a program can be complicated. For small programs, trial and error may be sufficient. For large programs, a more systematic approach is usually needed.

Pointers

> We've learned that everything in the computer's memory has an address, and that addresses are pointer constants. We can find the addresses of variables using the address-of operator `&`.

```
  int var = 8;
  int a = 12;
  int b = 12;

  std::cout << "var: " << var << "\n";
  std::cout << "a: " << a << "\n";
  std::cout << "b: " << b << "\n";

  // void pointers
  // void pointers can be NULL like all other pointers
  // before you can deference a void pointer, you have to type cast to the
  // object type it is pointing to first.
  void *var_ptr = nullptr;
  var_ptr = &var;
```

What are Pointers Really?

> Pointers are variables that hold address values. Pointers are defined using an asterisk `*` to mean pointer to. A data type is always included in pointer definitions (except `void*`), since the compiler must know what is being pointed to, so that it can perform arithmetic correctly on the pointer.

Accessing Pointers

> We access the thing pointed to using the asterisk in a different way, as the `dereference operator`, meaning contents of the variable pointed to by.
> The special type void* means a pointer to any type. It's used in certain difficult situations where the same pointer must hold addresses of different types.

```
void increment(int *value) {
  // dereference to manipulate data before increment
  // in C++, we can automatically not make the arguments pointers
  // but instead, normal functions and reference the data
  // this make it possible to do things like ++value without with deference operator
*
  ++*value;
}

int main() {
  int var = 8;
  int a = 12;
  int b = 12;

  std::cout << "var: " << var << "\n";
  std::cout << "a: " << a << "\n";
  std::cout << "b: " << b << "\n";

  // void pointers
  // void pointers can be NULL like all other pointers
  // before you can deference a void pointer, you have to type cast to the
  // object type it is pointing to first.
  void *var_ptr = nullptr;
  var_ptr = &var;

  // we can assign types to the pointers
  // and deference them if the type of the pointer is the same to the data of the
address
  // that is the address of the pointee.
  int *ptr = &a;
  *ptr = 20;

  // referencing the var b to the function increment
  increment(&b);

  // c++ referencing with increment_ref function
  int x = 14;
  increment_ref(x);
  std::cout << "x: " << x << "\n";

  // print the value of b after using the increment function on it
  std::cout << "b after: " << b << "\n";

  // print the value of a after deferencing it with the it's pointer ptr
```

```
  std::cout << "a after: " << a << "\n";

  // printing the address of the value the pointer is pointing to
  std::cout << "Pointee address for var: " << var_ptr << "\n";

  // This instead will print the address of the pointer.
  std::cout << "Address of the Pointer itself: " << &var_ptr << "\n";

  // As said early, we have to type cast void pointers before we can deference them
  // to this part, we access the value of the pointee using the type casting.
  std::cout << *(static_cast<int *>(var_ptr)) << "\n";

}
```

Arrays as Pointers

> Array elements can be accessed using array notation with brackets or pointer notation with an asterisk. Like other addresses, the address of an array is a constant, but it can be assigned to a variable, which can be incremented and changed in other ways.

```
#include <iostream>

int main() {
    // we declare an array
    int arr[3] = {1, 2, 3};

    // since array represents a block of related items,
    // the array name always points to the top of the block
    // in this case, the memory block for arr which contains 1, 2, 3


    // if we can access members from an array using pointers
    int someVar = *arr; // first item
    int anotherVar = *arr + 1; // second item

    std::cout << someVar << " - " << anotherVar;
}
```

```
Program returned: 0
Program stdout
1 - 2
```

> When the address of a variable is passed to a function, the function can work with the original variable. (This is not true when arguments are passed by value.) In this respect passing by pointer offers the same benefits as passing by reference, although pointer arguments must be dereferenced or accessed using the dereference operator.
> Remember this from Functions?

- by value

```
double magnitude_from_origin(Distance ref) {
    return (ref.x * ref.x) + (ref.y * ref.y);
}
```

- by reference (take note of the argument)

```
double magnitude_from_origin(Distance &ref) {
    return (ref.x * ref.x) + (ref.y * ref.y);
}
```

Avoiding Segfaults

> However, pointers offer more flexibility in some cases. A string constant can be defined as an array or as a pointer. The pointer approach may be more
> flexible, but there is a danger that the pointer value will be corrupted. Strings, being arrays of type char, are commonly passed to functions and accessed using pointers.