Udacity- C++ Nanodegree Notes

September 21, 2021 - N/A

Courtney Downs

Installation

1. MinGW
    a. https://sourceforge.net/projects/mingw/files/
    b. Click green button **Download latest version mingw-get-setup.exe**

2. Visual Studio Code extensions
    a. C/C++

Running

3. Compile the program
    a. `g++ hello.cpp`
4. Run the program
    a. `./a.exe` (or `a.out`)

Debugging in VS Code

5. Open project folder in VS Code
6. Set a breakpoint in code
7. Select Run and Debug icon on left
8. Select C++ (GDB/LLDB)
9. Select g++.exe Build and debug active file
10. Open terminal in that project dir and type *g++ -g hello.cpp -o hello.exe*
11. Click the green run icon on the left

Workspaces

12. Many projects and select quizzes will be accessed via in-browser environments called *Workspaces*. Workspaces streamline environment setup, simplify project submission, and can enable GPU support. All Workspaces are Linux-based. In the next classroom concepts, you'll see examples of the following three types of Workspaces:

    a. **Notebooks**: For explanations of small pieces of code or for smaller programs that you will write, you will use a Jupyter Notebook with an embedded terminal to compile and run. A notebook is a web application that allows for code, text, and visualizations to be combined and shared.
    b. **Terminals**: For larger programs, you will access other Workspaces directly from an in-browser terminal running a bash shell along with an in-browser code editor. Terminal Workspaces operate in the same manner as Linux terminals.
    c. **Desktops**: Finally, some terminal Workspaces are attached to a virtual desktop which will allow you to display visual output from your program.

        i. **WARNING:** Toggling GPU support may switch the physical server your session connects to, which can cause data loss unless you click the save button before toggling GPU support.
        ii. **WARNING**: In the hamburger menu the "Reset Data" or "Get New Content" button discards all of your changes and restores a clean copy of the Workspace.

Core Guidelines of C++

13. The C++ Core Guidelines are a set of best practices, recommendations, and rules for coding in C++ which have been developed by Bjarne Stroustrup and hundreds of other experts in the field. These guidelines are an important part of the language, as they help users to write the best, most up-to-date C++ possible.
    a. https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md
14. Compiled Languages v. Scripted Languages
    a. Compiled languages
        i. Allows you to build very stable and error free software.
        ii. **Preprocessor directives** *(#include statements)* cause the preprocessor to change your code in some way (by usually adding some library or another C++ file).

iii. Next, the compiler works through the preprocessed code line by line translating each line into the appropriate **machine language instruction**. This will also uncover any syntax errors that are present in your source code and will throw an error to the command line.
iv. Finally, if no errors are present, the compiler creates an object file with the machine language binary necessary to run on your machine. While the object file that the compiler just created is likely enough to do something on your computer, it still isn't a working executable of your C++ program. There is a final important step to reach an executable program.
v. C++ contains a vast library to aid in performing difficult tasks like I/O and hardware manipulation. You can include these libraries with preprocessor directives, but the preprocessor doesn't automatically add them to your code.
vi. In order for you to have a final executable program, another utility known as the **linker** must combine your object files with the library functions necessary to run the code.

b. Scripted languages *(Dynamically checked)*
    i. Runs quickly. No need to compile before running the code.

C++ Basics

15. #include
    a. The ***#include <iostream>*** is a *preprocessor command* (don't end with semicolon) which is executed before the code is compiled. It searches for the iostream *header file* and pastes its contents into the program.
16. Namespaces
    a. Namespaces are a way in C++ to group identifiers(names) together. They provide context for identifiers to avoid naming collisions. The **std** namespace is the namespace used for the standard library.
    b. The **using** command adds a namespace to the global scope of the program.
        i. For example, adding this line of code just below your include statements ***using std::cout;*** will allow you to use cout in your code instead of having to write std::cout.

17. C++ types

a. C++ is a strongly typed programming language, which means when a variable is "declared", you must specify the type in the declaration.
b. Primitives
    i. Int
    ii. Char
    iii. Bool
    iv. Float
    v. etc..

c. Vectors (arrays)
    i.
```
#include <vector>
using std::vector;

vector<int> vec = {10, 20, 30 };
```
d. Auto
    i. It is possible for C++ to do automatic type inference, using the ***auto*** keyword.
    ii. When NOT to use auto?
        1. It is helpful to manually declare the type of a variable if you want to be clear and explicit about the number precision being used. *(Ex. char16_t, char32_t, unsigned int, etc..)*
e. Functions
    i. The syntax of a function is the same as Java. However, by default all arguments passed to a function are by default pass by copy instead of pass by reference. This will slow down your code because of the time that it takes to make copies of things *(for example, passing a very long array to a function would require a copy of each item in the array to be made)*. An alternative is to make your arguments pass by reference by using the **&** symbol.
    ii. Example:

```
//const will  ensure that you don't modify original values
of this reference.
int AdditionFunction(const vector<int> &myVector){
      int sum = 0;
      for(const int &value : myVector){
            sum += value;
      }
```

```
        return sum;
    }
```

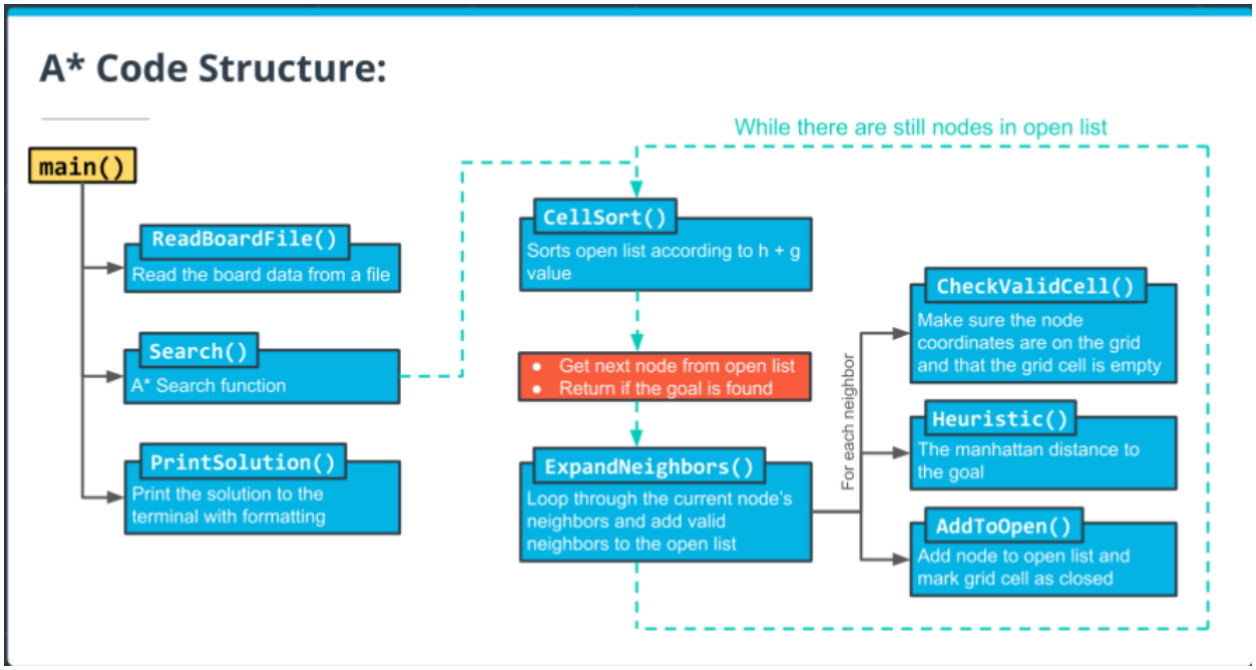
f. File input streams
    i. In C++, you can use the ***std::ifstream*** object to handle input file streams. To do this, you will need to include the header file that provides the file streaming classes: ***<fstream>***
        1. #include <fstream>
        2. Create a std::ifstream object using the path to your file.
        3. Evaluate the std::ifstream object as a bool to ensure that the stream creation did not fail.
        4. Use a while loop with getline to write file lines to a string.

    ii. Example Code
        1. **UdacityC++ > ProjectsUdacityCPP > ReadingBoardFromFile**

g. Constants
    i. ***const*** - I promise not to modify a variable, even though the variable can only be evaluated at run time.
    ii. ***constexpr*** - To guarantee that a variable can be evaluated at compile time.
    iii. Their differences are subtle. When in doubt, use const.

A* Search

18. The A* algorithm finds a path from the start node to the end node by checking for open neighbors of the current node, computing a heuristic for each of the neighbors, and adding those neighbors to the list of open nodes to explore next. The next node to explore is the one with the lowest total cost + heuristic (g + h). This process is repeated until the end is found, as long as there are still open nodes to explore



Pseudocode Algorithm

Search(*grid*, *initial_point*, *goal_point*) :

A. Initialize an empty list of open nodes.
B. Initialize a starting node with the following:
    a. x and y values given by *initial_point*.
    b. g = 0, where g is the cost for each move.
    c. h given by the heuristic function (a function of the current coordinates and the goal).
C. Add the new node to the list of open nodes.
D. **while** the list of open nodes is nonempty:
    a. Sort the open list by f-value
    b. Pop the optimal cell (called the *current* cell).
    c. Mark the cell's coordinates in the grid as part of the path.
    d. **if** the *current* cell is the goal cell:
        i. return the *grid*.
    e. **else**, expand the search to the *current* node's neighbors. This includes the following steps:
        i. Check each neighbor cell in the *grid* to ensure that the cell is empty: it hasn't been closed and is not an obstacle.
        ii. If the cell is empty, compute the cost (g value) and the heuristic, and add to the list of open nodes.
        iii. Mark the cell as closed.
E. If you exit the while loop because the list of open nodes is empty, you have run out of new nodes to explore and haven't found a path.

Writing Multi-file Programs

19. Header files
    a. Header files, or **.h** files, allow related function, method, and class ***declarations*** to be collected in one place. The corresponding ***definitions*** can then be placed in **.cpp** files.
    b. The compiler considers a header declaration a "promise" that the definition will be found later in the code, so if the compiler reaches a function that has not been defined yet, it can continue on compiling until the definition is found. *This allows functions to be defined (and declared) in arbitrary order.*
    c. **Put simply:** Without a header file, the order of the functions that you declare matters. One has to be created before it is referenced.

20. Compiling multiple files from terminal
    a. `g++ -std=c++17 ./code/main.cpp ./code/increment_and_sum.cpp ./code/vect_add_one.cpp`
    b. When you compile a project with **g++**, it actually performs several distinct tasks:
        i. The preprocessor runs and executes any statement beginning with a hash symbol: #, such as #include statements. This ensures all code is in the correct location and ready to compile.

        ii. Each file in the source code is compiled into an **"object file"** (a.o file). Object files are platform-specific machine code that will be used to create an executable.

```
1. // Creates main.o object file for main.cpp
   g++ -c main.cpp
   or
2. // Creates object files for all .cpp files in dir
   g++ -c *.cpp
```

        iii. The object files are **"linked"** together to make a single executable. In the examples you have seen so far, this executable is a.out or a.exe, but you can specify whatever name you want.

```
1. // Converts main.o to an executable.
   g++ main.o
   or
```

```
2. // Converts all object files in dir to executable
   g++ *.o
3. // Run the executable
   ./a.exe
```

c. But what if you make changes to your code and you need to re-compile? In that case, you can compile only the file that you changed, and you can use the existing object files from the unchanged source files for linking.
   i. Example:
```
1. g++ -c main.cpp //only file with changes to
   re-compile.
2. g++ *.o
3. ./a.exe
```
d. Compiling a project like this with g++ works fine if you have a small project with only a handful of files. But if you have a larger project with hundreds or thousands of files then you'll want to use a **build system** to manage all files during the build process.
   i. A build system allows for large projects to be compiled with a few commands, and is able to do this in an efficient way by only recompiling files that have been changed.

21. CMake and Make
   a. CMake is an open-source, platform-independent build system. CMake uses text documents, denoted as CMakeLists.txt files, to manage build environments, like make. https://en.wikipedia.org/wiki/Make_(software)

      i. CMakeLists.txt
         1. CMakeList.txt files are simple text configuration files that tell CMake how to build your project. There can be multiple CMakeLists.txt files in a project. In fact, one CMakeList.txt file can be included in each directory of the project, indicating how the files in that directory should be built.
         2. These files can be used to specify the locations of necessary packages, set build flags and environment variables, specify build target names and locations, and other actions.

      ii. Example CMakeLists.txt File:
```
1. cmake_minimum_required(VERSION 3.5.1)//min cmake ver
   set(CMAKE_CXX_STANDARD 14)        //min C++ ver
```

```
project("<any project name>")
add_executable(myapp.exe src/main.cpp
src/vect_add_one.cpp)
```

2. In the terminal at the same level as CMakeLists.txt run these commands:
   a. `mkdir build`
   b. `cd build`
   c. `// configures project and creates Makefile`
      `cmake ..`
   d. `// finds Makefile to build the project`
      `make`
   e. `// run the executable`
      `./myapp.exe`

3. Now, when you modify multiple files, you'll only need to run the *make* command from the build folder, and only the modified files will be compiled again.

b. **Note:** If you re-run CMake, or if you are having problems with your build, *it can be helpful to delete your build directory and start from scratch. Otherwise, some environment variables may not be reset correctly.*

Pointers

22. A C++ pointer is just a variable that stores the memory address of an object in your program. Once a memory address is accessed, you can store it using a pointer. A pointer can be declared by using the `*` operator in the declaration.

    a. Pointer Example:

```
#include <iostream>
using std::cout;
int main()
{
    int i = 5;
    // A pointer pointer_to_i is declared and initialized
      to the address of i.
    int* pointer_to_i = &i;

    // Prints the same memory addresses
    cout << "The address of i is:          " << &i << "\n";
    cout << "The variable pointer_to_i is: " << pointer_to_i
```

```
        << "\n";


    // Prints 5
    cout << "The value of the variable pointed to by
            pointer_to_i is: "
         <<  *pointer_to_i << "\n";

     // Dereference to change actual value
    *pointer_to_i = 10;

     // Prints the same value of 10
     cout << "The value of the variable pointed to
              by pointer_to_i is: "
          <<  *pointer_to_i << "\n";
     cout << “The value of i is: ” << i << “\n”;
}
```

b. Reference Example:

```
#include <iostream>
using std::cout;

int main()
{
    int i = 1;

    // Declare a reference to i.
    int &j = i;
    cout << "The value of j is: " << j <<"\n";
    // output j=1

     // Change the value of i.
    i = 5;
    cout << "The value of i is changed to: " << i << "\n";
    // output i = 5
    cout << "The value of j is now: " << j << "\n";
    // output j = 5

    // Change the value of the reference.
    // Since reference is just another name for the
       variable,
    j = 7;
    cout << "The value of j is now: " << j << "\n";
    // output j = 7
    cout << "The value of i is changed to: " << i << "\n";
    // output i = 7
}
```

23. * symbol
    a. Once you have a pointer, you may want to retrieve the object it is pointing to. In this case, the * symbols can be used again. This time, however, it will appear on the right hand side of an equation or in front of an already-defined variable, so the meaning is different. In this case, it is called the **"dereferencing operator"**, and it returns the object being pointed to.

    b. ***NOTE**: You can dereference to access the actual value or dereference to change the actual value*
    c. ***NOTE**: When dereferencing it's a good idea to always use **( )** around your dereferenced pointer. This will allow you to treat it just like the original variable and call functions on it. (Example: (*pointerToAnArray)[0])*

```
 i.   int i = 5;
ii.   int *myPointer = &i; //myPointer has value of address to i
iii.  int myObject = *myPointer;   // myObject is 5
```

24. & symbol
    a. If it appears on the left side of an equation (e.g. when declaring a variable), it means that the variable is declared as a reference. If the & appears on the right side of an equation, or before a previously defined variable, it is used to return a memory address, as in the example above.

```
 i.   int i = 5;
ii.   int &myReference = i;//cout << myReference; would display 5
iii.  int myAddress = &i;  // myAddress &myReference. They both
      hold the address of i
```

25. Differences between pointers and references
    a. References are generally easier and safer than pointers. As a decent rule of thumb, references should be used in place of pointers when possible.

    b. However, there are times when it is not possible to use references. One example is object initialization. You might like one object to store a reference to another object. However, if the other object is not yet available when the first object is created, then the first object will need to use a pointer, not a reference, since a

reference cannot be null. The reference could only be initialized once the other object is created.

References	Pointers
References must be initialized when they are declared. This means that a reference will always point to data that was intentionally assigned to it.	Pointers can be declared without being initialized, which is dangerous. If this happens mistakenly, the pointer could be pointing to an arbitrary address in memory, and the data associated with that address could be meaningless, leading to undefined behavior and difficult-to-resolve bugs.
References can not be null. This means that a reference should point to meaningful data in the program.	Pointers can be null. In fact, if a pointer is not initialized immediately, it is often best practice to initialize to `nullptr`, a special type which indicates that the pointer is null.
When used in a function for pass-by-reference, the reference can be used just as a variable of the same type would be.	When used in a function for pass-by-reference, a pointer must be dereferenced in order to access the underlying object.

***NOTE**: When using pointers with functions, some care should be taken. If a pointer is passed to a function and then assigned to a variable in the function that goes out of scope after the function finishes executing, then the pointer will have undefined behavior at that point - the memory it is pointing to might be overwritten by other parts of the program.*

Maps

26. A map (alternatively hash table, hash map, or dictionary) is a data structure that uses key/value pairs to store data, and provides efficient lookup and insertion of the data. The name "dictionary" should provide an excellent idea of how these work, since a dictionary is a real life example of a map.

Classes

27. A class is a group of related variables and functions that can be initialized as a single or multiple objects.

    a. **.h file**

```
#ifndef CAR_H
#define CAR_H

#include <string>
using std::string;
using std::cout;

class Car {
        public:
              void PrintCarData();
              void IncrementDistance();
              // Using a constructor list in the
            constructor:
              Car(string c, int n) : color(c), number(n) {}
        private:
              // The variables do not need to be accessed
            outside of
              // functions from this class, so we can set
            them to private.
              string color;
              int distance = 0;
              int number;
              // NOTE: it's convention to add _ at the end
            of all private variables. (color_   distance_
            number_ ) but I didn't do it above.
};

#endif
```

a. The constructor now uses an **initializer list**
https://en.cppreference.com/w/cpp/language/constructor

Car(string c, int n) : color(c), number(n) {}

Here, the class members are initialized before the body of the constructor *(which is now empty)*. Initializer lists are a quick way to initialize many class attributes in the constructor. Additionally, the compiler treats attributes initialized in the list slightly differently than if they are initialized in the constructor body. For reasons beyond the scope of this course, if a class attribute is a reference, it must be initialized using an initializer list.

28. **.cpp file**

```
// The Car class
class Car {
  public:
    // Method to print data.
    void PrintCarData() {
        cout << "The distance that the " << this->color <<
                " car " << this->number << " has traveled
            is: "
               << this->distance << "\n";
    }

    // Method to increment the distance travelled.
    void IncrementDistance() {
        this->distance++;
    }

    // Class/object attributes
    string color;
    int distance = 0;
    int number;

};
```

29. Object Pointers

```
// Simultaneously dereference the pointer and
// access IncrementDistance().
cp->IncrementDistance();

// Dereference the pointer using *, then
// access IncrementDistance() with traditional
// dot notation.

(*cp).IncrementDistance();
```

Object-Oriented Programming

30. Structures
    a. Allow developers to create their own types ("user-defined" types) to aggregate data relevant to their needs.
    b. Example.

```
// Define outside of the main method.
struct Date {
  int day{1};
  int month{1};
  int year{2000};// Default values {}
};

// Create an instance of the Date structure
Date date;
// Initialize the attributes of Date
date.day = 1;
date.month = 10;
date.year = 2019;
```

    c. Structs can also have accessors and mutators. This will prevent the user from being able to directly set the value of a variable in a struct to some value that might be invalid. For example, in the above code you can just set the **date.day = -7** which wouldn't make any sense for a day of the month.

```
struct Date {
       public:
        int Day() { return day; }
        void Day(int day) { this->day = day; }
        int Month() { return month; }
        void Month(int month) { this->month = month; }
        int Year() { return year; }
        void Year(int year) { this->year = year; }

       private:
        int day{1};
        int month{1};
        int year{0};

};
```

d. Why use a Struct over a Class ?
    i. Typically, one would use a struct instead of a class when you just want to create an object with publicly available data where all of the data members can vary independently of each other and don't really have any limitations.

31. Classes
    a. When you have variables that you want to make private or you want to put limits on them and maybe even have them interdependent, then that's when it's best to use a class instead.

```
class Date {
       public:
        int Day() { return day; }
        void Day(int d) {
          if (d >= 1 && d <= 31) day_ = d; // Invariant
        }

       private:
        int day_{1};
        int month_{1};
        int year_{0};

};
```

    b. Constructors
        i. An operation that initializes an object. Typically a constructor establishes an invariant and often acquires resources needed for an object to be used (for example, a Player object's constructor might accept a 2DCollision object). These are then released by a destructor.

```
class Date {
       public:
              Date(int d, int m, int y) {  // Constructor
                Day(d);
              }
              int Day() { return day; }
              void Day(int d) {
```

```
        if (d >= 1 && d <= 31) day = d;
    }
    int Month() { return month; }
    void Month(int m) {
        // Invariant
        if (m >= 1 && m <= 12) month = m;
    }
    int Year() { return year_; }
    void Year(int y) { year = y; }

private:
 int day{1};
 int month{1};
 int year{0};

};
```

ii. The compiler will define a default constructor, which accepts no arguments, for any class or structure that does not contain an explicitly-defined constructor.

32. Scope Resolution

a. C++ allows different identifiers (variable and function names) to have the same name, as long as they have different scope. For example, two different functions can each declare the variable int i, because each variable only exists within the scope of its parent function.

   In some cases, scopes can overlap, in which case the compiler may need assistance in determining which identifier the programmer means to use. The process of determining which identifier to use is called *"scope resolution"*.

b. `::` is the scope resolution operator. We can use this operator to specify which namespace or class to search in order to resolve an identifier.

   1. `Person::move(); \\ Call the move function that's in Person class.`

c. This becomes particularly useful if we want to separate class *declaration* from class *definition*. (.cpp and .h files)

33. Namespaces

a. Namespaces allow programmers to group logically related variables and functions together. Namespaces also help to avoid conflicts between two variables that have the same name in different parts of a program.

```
namespace English {
void Hello() { std::cout << "Hello, World!\n"; }
}  // namespace English

namespace Spanish {
void Hello() { std::cout << "Hola, Mundo!\n"; }
}  // namespace Spanish

int main() {
  English::Hello();
  Spanish::Hello();
}
```

b. In this example, we have two different void Hello() functions. If we put both of these functions in the same namespace, they would conflict and the program would not compile. However, by declaring each of these functions in a separate namespace, they are able to coexist. Furthermore, we can specify which function to call by prefixing Hello() with the appropriate namespace, followed by the :: operator.

34. Initializer lists

a. Initialize member variables to specific values, just before the class constructor runs. This initialization ensures that class members are automatically initialized when an instance of the class is created. The code below shows what the Date class above (on previous page) looks like with the year set in the initializer list.

```
Date::Date(int d, int m, int y) : year(y) {
    Day(d);
    Month(m);
}
```

b. In this example, the member value year is initialized through the initializer list, while day and month are assigned from within the constructor. Assigning day and month allows us to apply the invariants set in the mutator.

c. Why use an Initializer list ?

i. *Initializer lists exist for a number of reasons. First, the compiler can optimize initialization faster from an initialization list than from within the constructor.*

ii. *A second reason is a bit of a technical paradox. If you have a const class attribute, you can only initialize it using an initialization list. Otherwise, you would violate the const keyword simply by initializing the member in the constructor!*

iii. *The third reason is that attributes defined as references must use initialization lists.*

35. Encapsulation
    a. A grouping together of data and logic into a single unit. Classes encapsulate data and functions that operate on that data.
    b. When designing a class, you want to group together relevant data and functions, but also limit member functions to only those functions that need direct access to the representation of a class. (i.e keep helper functions private)
36. Abstraction
    a. The separation of a class's interface from the details of its implementation. The interface provides a way to interact with an object, while hiding the details and implementation of how the class works.
37. Static
    a. Class members
        i. can be declared **static**, which means that the member belongs to the entire class, instead of a specific instance of the class. More specifically, a static member is created once and then shared by all instances of the class.

```
class Foo {
       public:
              static int count;
              Foo() { Foo::count += 1; }
};
```

        ii. Static members are declared within their class (often in a header file) but in most cases they must be defined within the **global scope**. That's because memory is allocated for static variables immediately when the program begins, at the same time any global variables are initialized.
        iii. An exception to the global definition of static members is if such members can be marked as **constexpr**. In that case, the static member variable can be both declared and defined within the class definition:

```
struct Kilometer {
    static constexpr int meters{1000};
};
```

b. Class functions
   i. In addition to static member variables, C++ supports static member functions (or "methods"). Just like static member variables, static member functions are instance-independent: they belong to the class, not to any particular instance of the class.

   ii. One corollary to this is that we can invoke a static member function without ever creating an instance of the class.
      a. r = Math.random(); // Java

Advanced OOP

38. Inheritance
   a. A way of defining class hierarchies in object-oriented programming to allow for classes to share functionality with child classes.
   b. You can also have the ability to inherit functionality from multiple classes by using a comma. (*class Dog: public Animal, public Pet {};*)
      i. Guidelines on when to use multiple inheritance
         1. http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c135-use-multiple-inheritance-to-represent-multiple-distinct-interfaces

   c. Example:

```
class Vehicle {
public:
        int wheels = 0;
        string color = "blue";

        void Print() const
        {
          std::cout << "This " << color << " vehicle has "
                    << wheels << " wheels!\n";
        }
};


class Car : public Vehicle {
public:
```

```
        bool sunroof = false;
};


class Bicycle : public Vehicle {
public:
        bool kickstand = true;

};
```

i. Notice the **public** access specifier next to the : public Vehicle{ } header. This tells the program that the child class Bicycle can have it's instances have direct access to the member functions of the parent class Vehicle.
    1. I.e you can do this:
        a.
```
Bicycle myBike;
myBike.wheels = 2;
```

ii. If the header used a private access specifier like this:
class Bicycle : **private** Vehicle {
Then this would restrict the member functions of the parent Vehicle class to only be accessible within the Bicycle class and not publicly with Bicycle instances. So you would have to do something like this in the Bicycle class to update wheels.

1.
```
class Bicycle : public Vehicle {
public:
        bool kickstand = true;
        void setBikeWheels(int wheels){
             this.wheels = wheels;
        }

};
```

iii. Inherited Access Specifiers
    1. **Public inheritance:** the public and protected members of the base class listed after the specifier keep their member access in the derived class
    2. **Protected inheritance:** the public and protected members of the base class listed after the specifier are protected members of the derived class

3. **Private inheritance:** the public and protected members of the base class listed after the specifier are private members of the derived class.

39. Composition

a. Composition is a closely related alternative to inheritance. Composition involves constructing ("composing") classes from other classes, instead of inheriting traits from a parent class.
b. A common way to distinguish "composition" from "inheritance" is to think about what an object can do, rather than what it is. This is often expressed as **"has a"** versus **"is a"**.
c. There is no hard and fast rule about when to prefer composition over inheritance. In general, if a class needs only extend a small amount of functionality beyond what is already offered by another class, it makes sense to inherit from that other class. However, if a class needs to contain functionality from a variety of otherwise unrelated classes, it makes sense to compose the class from those other classes.

40. Friend Classes

a. Friend classes provide an alternative inheritance mechanism to derived classes.
    i. The main difference between classical inheritance and friend inheritance is that a friend class can **access private members of the base class**, which isn't the case for classical inheritance.
    ii. In classical inheritance, a derived class can only access public and protected members of the base class.

    iii. Example Code
        1. **UdacityC++ > ProjectsUdacityCPP > Friends**

41. Polymorphism

a. A paradigm in which a function may behave differently depending on how it is called. In particular, the function will perform differently based on its inputs.
b. This is achieved in two ways:
    i. Overloading
    ii. Overriding

c. Overloading

i. You can write two (or more) versions of a function with the same name. Overloading requires that we leave the function name the same, but we modify the function signature.

```
#include <ctime>

class Date {
public:
    Date(int day, int month, int year) : day_(day)
                  , month_(month), year_(year) {}
    Date(int day, int month) : day_(day),
month_(month)
    {
        // automatically sets the Date to the current
      year
        time_t t = time(NULL);
        tm* timePtr = localtime(&t);
        year_ = timePtr->tm_year;
    }

private:
    int day_;
    int month_;
    int year_;

};
```

ii. Operator overloading

1. You can choose any operator from the ASCII table and give it your own set of rules.
2. Operator overloading can be useful for many things. Consider the + operator. *We can use it to add ints, doubles, floats, or even std::strings.*
3. In order to overload an operator, use the operator keyword in the function signature:
4. Example Code
    a. **UdacityC++ > ProjectsUdacityCPP > Operator Overloading**

d. Virtual Functions

i. These functions are declared (and possibly defined) in a base class, and can be overridden by derived classes. This approach declares an interface at the base level, but delegates the implementation of the interface to the derived classes.

ii. In this exercise, class **Shape** is the base class. Geometrical shapes possess both an area and a perimeter. Area() and Perimeter() should be virtual functions of the base class interface. *Append = 0 to each of these functions in order to declare them to be "pure" virtual functions.*

iii. A **pure virtual function** is a virtual function that the base class declares but does not define. (Has a header without implementation details)

iv. A pure virtual function has the side effect of making its class abstract. This means that the class cannot be instantiated. Instead, only classes that derive from the abstract class and override the pure virtual function can be instantiated.

1.
```
class Shape {
    public:
        Shape() {}
        virtual double Area() const = 0;
        virtual double Perimeter() const = 0;
};
```

v. Example Code

1. **UdacityC++ > ProjectsUdacityCPP > VirtualFunctions**

e. Overriding

i. Overriding a function occurs when a derived class defines the implementation of a virtual function that it inherits from a base class.

ii. It is possible, but not required, to specify a function declaration as an override. *(it is good practice to use the override keyword)*

```
class Shape {
public:
  virtual double Area() const = 0;
  virtual double Perimeter() const = 0;
};

class Circle : public Shape {
public:
    Circle(double radius) : radius_(radius) {}
    double Area() const override {
```

```
        return pow(radius_, 2) * PI;
    } // specified as an override function
    double Perimeter() const override {
        return 2 * radius_ * PI;
    }// specified as an override function

private:
  double radius_;
};
```

iii. This specification tells both the compiler and the human programmer that the purpose of this function is to override a virtual function. The compiler will verify that a function specified as override does indeed override some other virtual function, or otherwise the compiler will generate an error.

Generic Programming

42. What is generic programming ?

a. Generic programming is an example of polymorphism that generalizes the parameters of classes and functions. Generic programming basically allows you to implement polymorphism without the need for inheritance and interfaces. You can say this function will work for anything of type T instead of this function will work for objects that are children of Animal.

43. Templates

a. Templates enable generic programming by generalizing a function to apply to any class. Specifically, templates use types as parameters so that the same implementation can operate on different data types.

b. For example, you might need a function to accept many different data types. The function acts on those arguments, perhaps dividing them or sorting them or something else. Rather than writing and maintaining the multiple function declarations, each accepting slightly different arguments, you can write one function and pass the argument types as parameters. At compile time, the compiler then expands the code using the types that are passed as parameters.

i. Use the keyword template to create a generic function as shown below:

```
template <typename Type> Type Sum(Type a, Type b) {
	return a + b;
}

int main() {
	std::cout << Sum<double>(20.0, 13.7) << "\n";
}
```

c. Template deduction
    i. Deduction occurs when you instantiate an object without explicitly identifying the types. Instead, the compiler "deduces" the types. This can be helpful for writing code that is generic and can handle a variety of inputs.
        1. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-deduce

// **TODO**: After project 2 is complete, create a **Linux HTOP notes document** on google drive with all info from Processor Data, Process Data, and System Data pages in project description.

Memory

44. Understanding the number system used by computers to store and process data is essential for effective memory management, the sections below cover binary and hexadecimal number systems and the structure of memory addresses.

45. Why is Binary useful?
    a. Early attempts to invent an electronic computing device met with disappointing results as long as engineers and computer scientists tried to use the decimal system. **John Atanasoff**, proposed a numbering system called Base 2 or binary and it is represented by the digits 0 and 1 (called 'bit') instead of 0-9 as with the

decimal system. Differentiating between only two symbols, especially at high frequencies, was much easier and more robust than with 10 digits.

46. What is ASCII ?
    a. Over the years, many coding schemes and techniques were invented to manipulate these 0s and 1s effectively. One of the most widely used schemes is called ASCII (*American Standard Code for Information Interchange*), which lists the binary code for a set of 127 characters. The idea was to represent each letter with a sequence of binary numbers so that storing texts in computer memory and on hard (or floppy) disks would be possible.

47. Hexadecimal
    a. There are several reasons why it is preferable to use hex numbers instead of binary numbers (which computers store at the lowest level), three of which are given below:

        i. **Readability**: It is significantly easier for a human to understand hex numbers as they resemble the decimal numbers we are used to. It is simply not intuitive to look at binary numbers and decide how big they are and how they relate to another binary number.

        ii. **Information density**: A hex number with two digits can express any number from 0 to 255 (because 16^2 is 256). To do the same in the binary system, we would require 8 digits. This difference is even more pronounced as numbers get larger and thus harder to deal with.

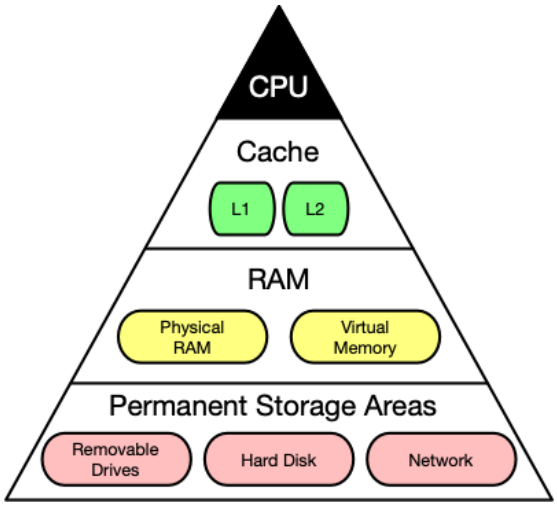
        iii. **Conversion into bytes**: Bytes are units of information consisting of 8 bits. Almost all computers are byte-addressed, meaning all memory is referenced by byte, instead of by bit. Therefore, using a counting system that can easily convert into bytes is an important requirement. We will shortly see why grouping bits into a byte plays a central role in understanding how computer memory works.

48. GNU Debugger (GDB) - Cheat Sheet .
    a. https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf

49. Memory Types

a. When the CPU of a computer needs to access memory, it wants to do this with minimal latency. Also, as large amounts of information need to be processed, the available memory should be sufficiently large with regard to the tasks we want to accomplish.

b. Regrettably though, low latency and large memory are not compatible with each other (at least not at a reasonable price). In practice, the decision for low latency usually results in a reduction of the available storage capacity (and vice versa). This is the reason why a computer has multiple memory types that are arranged hierarchically. The following pyramid illustrates the principle:



c. Let us take a look at a typical computer usage scenario to see how the different types of memory are used:

1. After switching on the computer, it loads data from its read-only memory (ROM) and performs a power-on self-test (POST) to ensure that all major components are working properly. **Additionally, the computer memory controller checks all of the memory addresses with a simple read/write operation to ensure that memory is functioning correctly.**

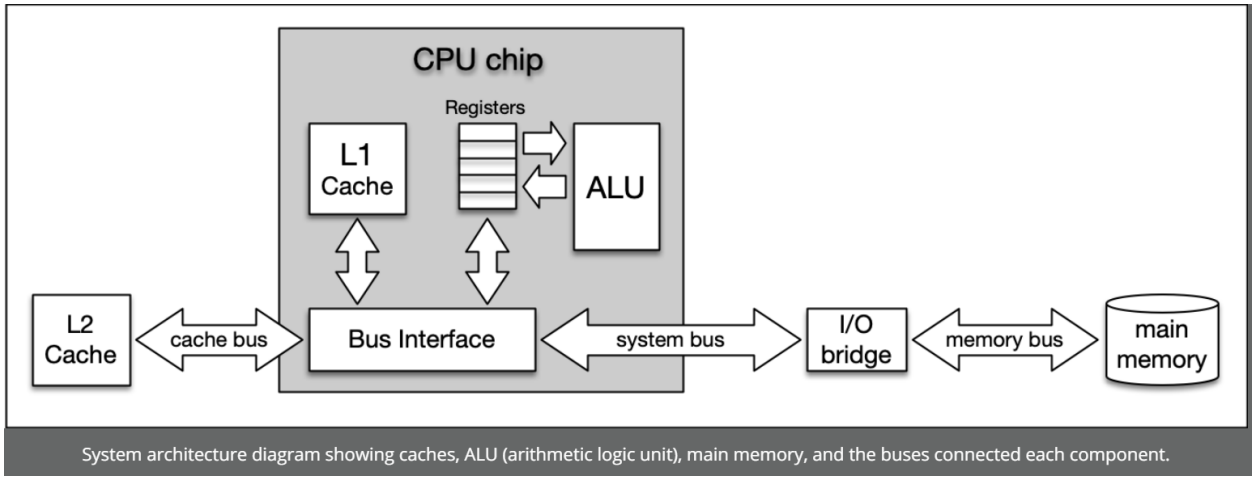
2. After performing the self-test, the computer loads the basic input/output system (BIOS) from ROM. **The major task of the BIOS is to make the computer functional by providing basic information about such things as storage devices, boot sequence, security or auto device recognition capability.**

3. The process of activating a more complex system on a simple system is called "bootstrapping": It is a solution for the chicken-egg-problem of starting a software-driven system by itself using software. During bootstrapping, the computer loads the operating system (OS) from the hard drive into random access memory (RAM). **RAM is considered "random access" because any memory cell can be accessed directly by intersecting the respective row and column in the matrix-like memory layout.** For performance reasons, many parts of the OS are kept in RAM as long as the computer is powered on.

4. When an application is started, it is loaded into RAM. However, several application components are only loaded into RAM on demand to preserve memory. Files that are opened during runtime are also loaded into RAM. When a file is saved, it is written to the specified storage device. After closing the application, it is deleted from RAM.

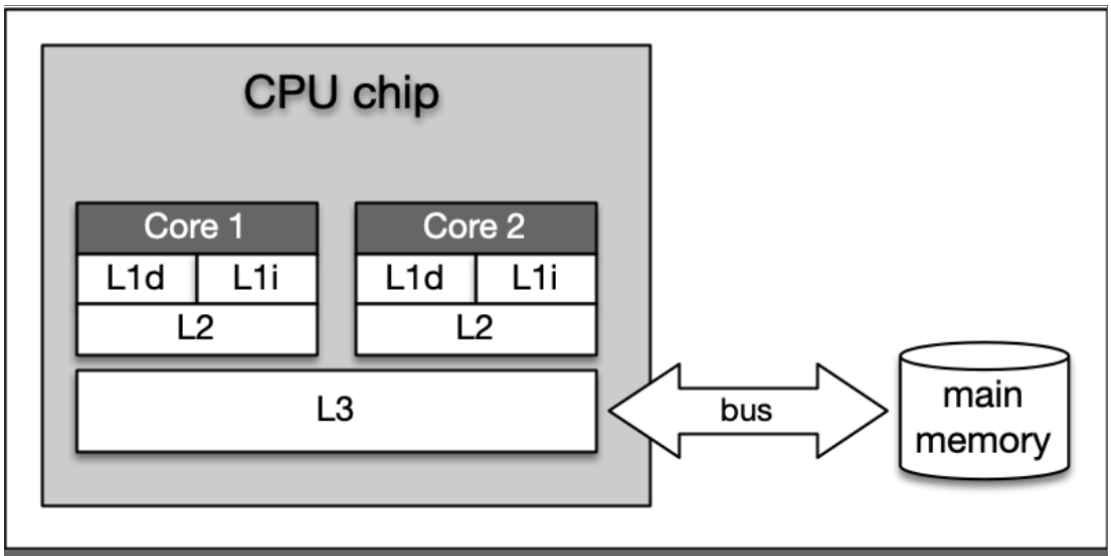
d. Terms
    i. **Bit size**
        1. How many bytes of data it can access in RAM memory at the same time. A 16-bit CPU can access 2 bytes (with each byte consisting of 8 bit) while a 64-bit CPU can access 8 bytes at a time.
    ii. **Processing speed**
        1. Measured in Gigahertz or Megahertz and denotes the number of operations it can perform in one second.

50. Cache Levels
    a. Cache memory is much faster but also significantly smaller than standard RAM. It holds the data that will (or might) be used by the CPU more often. In the memory hierarchy we have seen in the last section, the cache plays an intermediary role between fast CPU and slow RAM and hard disk. The figure below gives a rough overview of a typical system architecture:



System architecture diagram showing caches, ALU (arithmetic logic unit), main memory, and the buses connected each component.

b. The central CPU chip is connected to the outside world by a number of buses. There is a cache bus, which leads to a block denoted as L2 cache, and there is a system bus as well as a memory bus that leads to the computer main memory. The latter holds the comparatively large RAM while the L2 cache as well as the L1 cache are very small with the latter also being a part of the CPU itself.



L1, L2, and L3 cache

c. Cache Levels
   i. **Level 1 (L1) Cache:** Fastest and smallest memory (typically 16 to 64kBytes) type in the cache hierarchy. The importance of the L1 cache grows with increasing speed of the CPU. In the L1 cache, the most frequently required instructions and data are buffered so that as few accesses as possible to the slow main memory are required.

ii. **Level 2 (L2) Cache:** (typically 2 megabytes) Located close to the CPU and has a direct connection to it or located within the CPU itself. The choice between a processor with more clock speed or a larger L2 cache can be answered as follows: With a higher clock speed, individual programs run faster, especially those with high computing requirements. As soon as several programs run simultaneously, a larger cache is advantageous.

iii. **Level 3 (L3) Cache:** Shared among all cores of a multicore processor. The L3 cache has less the function of a cache, but is intended to simplify and accelerate the cache coherence protocol and the data exchange between the cores.

d. NOTE: Cache coherence protocol:
   i. https://en.wikipedia.org/wiki/Cache_coherence

51. Temporal and Spatial Locality
    a. The following table presents a rough overview of the latency of various memory access operations. While L1 access operations are close to the speed of a photon traveling at light speed for a distance of 1 foot, the latency of L2 access is roughly one order of magnitude slower already while access to main memory is two orders of magnitude slower.

```
          0.5 ns - CPU L1 dCACHE reference
          1   ns - speed-of-light (a photon) travel a 1 ft (30.5cm) distance
          5   ns - CPU L1 iCACHE Branch mispredict
          7   ns - CPU L2  CACHE reference
         71   ns - CPU cross-QPI/NUMA best  case on XEON E5-46*
        100   ns - MUTEX lock/unlock
        100   ns - own DDR MEMORY reference
        135   ns - CPU cross-QPI/NUMA best  case on XEON E7-*
        202   ns - CPU cross-QPI/NUMA worst case on XEON E7-*
        325   ns - CPU cross-QPI/NUMA worst case on XEON E5-46*
     10,000   ns - Compress 1K bytes with Zippy PROCESS
     20,000   ns - Send 2K bytes over 1 Gbps NETWORK
    250,000   ns - Read 1 MB sequentially from MEMORY
    500,000   ns - Round trip within a same DataCenter
 10,000,000   ns - DISK seek
 10,000,000   ns - Read 1 MB sequentially from NETWORK
 30,000,000   ns - Read 1 MB sequentially from DISK
150,000,000   ns - Send a NETWORK packet CA -> Netherlands
|   |   |   |
|   |   | ns|
|   | us|
| ms|
```

52. Physical Memory
    a. There are several memory-related problems, that programmers need to know about:
        i. **Holes in address space**: If several programs are started one after the other and then shortly afterwards some of these are terminated again, it must be ensured that the freed-up space in between the remaining programs does not remain unused. If memory becomes too fragmented, it might not be possible to allocate a large block of memory due to a large-enough free contiguous block not being available any more.

        ii. **Programs writing over each other**: If several programs are allowed to access the same memory address, they will overwrite each others' data at this location. In some cases, this might even lead to one program reading sensitive information (e.g. bank account info) that was written by another program. This problem is of particular concern when writing concurrent programs which run several threads at the same time.

53. Virtual Memory

    a. Virtual memory is a valuable concept in computer architecture that **allows you to run large, sophisticated programs on a computer even if it has a relatively small amount of RAM**. A computer with virtual memory artfully juggles the conflicting demands of multiple programs within a fixed amount of physical memory.
        i. This can be very useful for tasks that require intense RAM usage like video editing.

    b. Memory-related problems:
        i. **Holes in address space**
            1. It must be ensured that the freed-up space in between the remaining programs does not remain unused after multiple programs are run one after the other and then shortly afterwards some of them are terminated.
        ii. **Programs writing over each other**
            1. If several programs are allowed to access the same memory address, they will overwrite each others' data at this location.This might even lead to one program reading sensitive information (e.g.

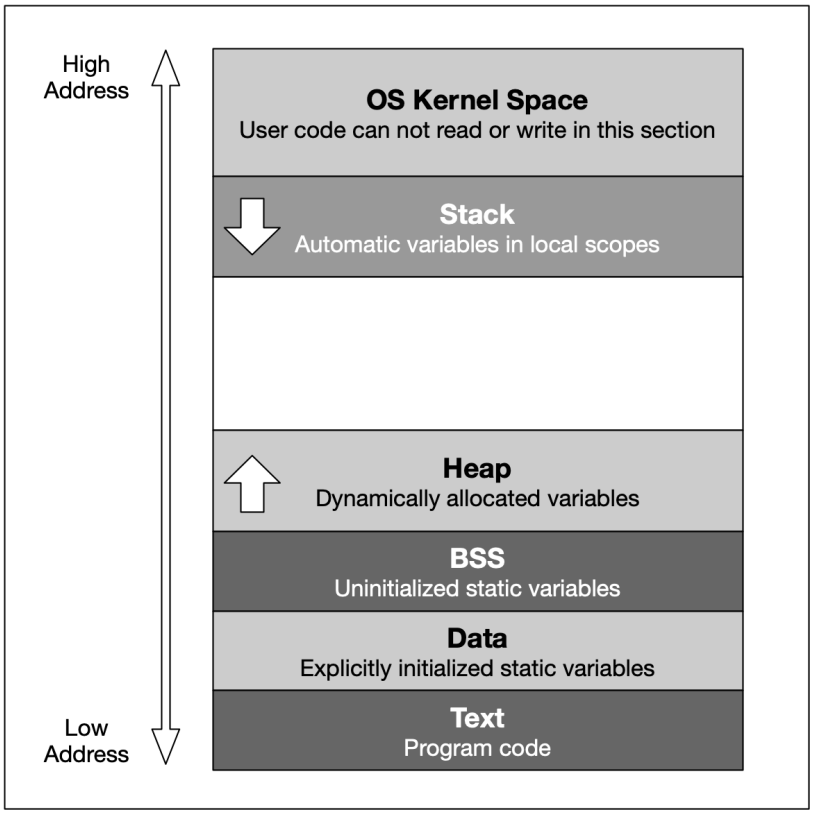
bank account info) that was written by another program. This problem is of particular concern when writing concurrent programs which run several threads at the same time.

c. The basic idea of virtual memory is to separate the addresses a program may use from the addresses in physical computer memory. By using a mapping function, an access to (virtual) program memory can be redirected to a real address which is guaranteed to be protected from other programs.

d. In a nutshell, virtual memory guarantees us a fixed-size address space which is largely independent of the system configuration. Also, the OS guarantees that the virtual address spaces of different programs do not interfere with each other.

54. Process Memory Model

a. A program is assigned its own virtual memory by the OS. This address space is arranged in a linear fashion with one block of data being stored at each address. It is also divided into several distinct areas as illustrated below.



b. **OS Kernel Space**
    i. Only the most trusted code is executed here - it is fully maintained by the OS and serves as an interface between the user code and the system kernel.
c. **Text**
    i. This section of memory is also reserved for the OS. It is where the program code generated by the compiler and linker are.

d. **Stack (aka “free space”)**
    i. The stack is a contiguous memory block with a fixed maximum size. It is used for storing automatically allocated variables such as local variables or function parameters. If there are multiple threads in a program, then each thread has its own stack memory. The stack is managed “automatically” by the compiler.

e. **Heap (aka “free store”, “dynamic memory”, or “storage classes” in C++)**

i. The heap is where data with dynamic storage lives. It is shared among multiple threads in a program, which means that memory management for the heap needs to take concurrency into account. Managing memory on the heap is more computationally expensive for the OS, which makes it slower than stack memory. If memory is allocated on the heap, it is the programmer's responsibility to free it again when it is no longer needed.

f. **BSS (Block Started by Symbol)**
   i. This segment is used in many compilers and linkers for a segment that contains global and static variables that are initialized with zero values. This memory area is suitable, for example, for arrays that are not initialized with predefined values.
g. **Data**
   i. This segment serves the same purpose as the BSS with the major difference being that variables in the Data segment have been initialized with a value other than zero.

55. Memory Allocation in C++

a. **Allocate**
   i. The process of assigning an area of memory to a variable to store its value.
b. **Deallocate**
   i. When the system reclaims the memory from the variable, it no longer has an area to store its value.

Generally, three basic types of memory allocation are supported:

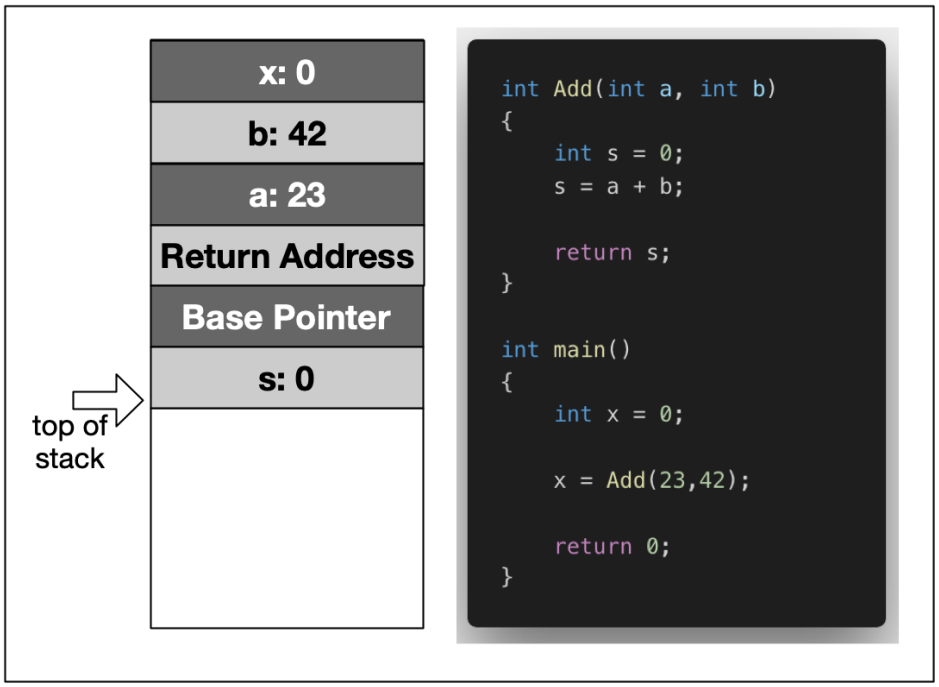
- **Static memory allocation** is performed for static and global variables, which are stored in the BSS and Data segment. Memory for these types of variables is allocated once when your program is run and persists throughout the life of your program.

- **Automatic memory allocation** is performed for function parameters as well as local variables, which are stored on the stack. Memory for these types of variables is allocated when the path of execution enters a scope and freed again once the scope is left.

- **Dynamic memory allocation** is a possibility for programs to request memory from the operating system at runtime when needed. This is the major difference between automatic and static allocation, where the size of the variable must be known at compile time. Dynamic memory allocation

is not performed on the limited stack but on the heap and is thus (almost) only limited by the size of the address space.

c. Properties of Stack Memory
   i. The stack is a **contiguous block of memory**. It will not become fragmented (as opposed to the heap) and it has a fixed maximum size.
   ii. When the **maximum size of the stack** memory is exceeded, a program will crash.
   iii. Allocating and deallocating **memory is fast** on the stack. It only involves moving the stack pointer to a new position.



d. Properties of Heap Memory
   i. The programmer can request the allocation of memory by issuing a command such as **malloc** or **new**. This block of memory will remain allocated until the programmer explicitly issues a command such as **free** or **delete.**
   ii. With dynamically allocated heap memory, variables are allocated at **run-time** as opposed to **compile-time** like that of local variables on the

stack. This means that dynamically allocated variables can have their size tailored.

iii. Heap memory is only constrained by the size of the address space and by the available memory. However, if the programmer forgets to deallocate a block of heap memory, it will remain unused until the program is terminated. This is called a **"memory leak"**.

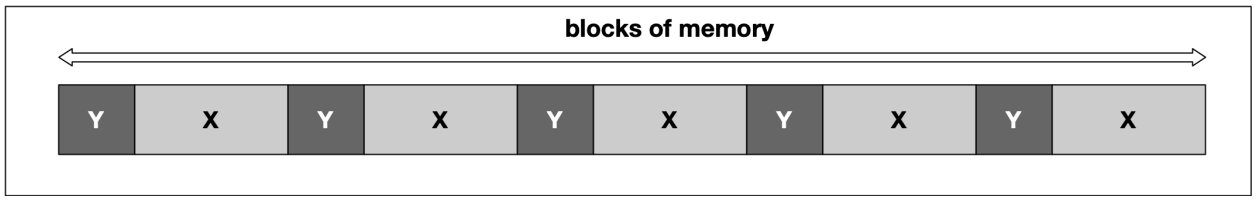
*(**Memory leaks** are mostly problematic for programs that run for a long time and/or use large data structures. In such a case, memory leaks can gradually fill the heap until allocation requests can no longer be properly met and the program stops responding or crashes completely.)*

iv. Unlike the stack, the heap is shared among multiple threads, which means that memory management for the heap needs to take concurrency into account as several threads might compete for the same memory resource.

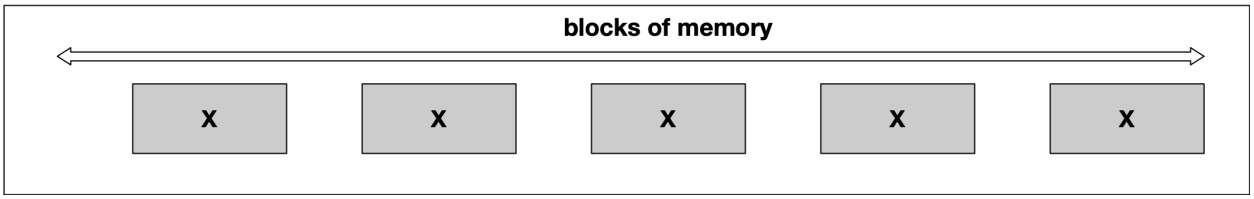
v. When memory is allocated or deallocated on the stack, the stack pointer is simply shifted upwards or downwards. Due to the sequential structure of stack memory management, stack memory can be managed (by the operating system) easily and securely. With heap memory, allocation and deallocation can occur arbitrarily, depending on the lifetime of the variables. This can result in **fragmented memory** over time, which is much more difficult and expensive to manage.

e. Memory Fragmentation

i. Let us construct a theoretic example of how memory on the heap can become fragmented: Suppose we are interleaving the allocation of two data types X and Y in the following fashion: First, we allocate a block of memory for a variable of type X, then another block for Y and so on in a repeated manner until some upper bound is reached. At the end of this operation, the heap might look like the following:



ii. At some point, we might then decide to deallocate all variables of type Y, leading to empty spaces in between the remaining variables of type X. In between two blocks of type "X", no memory for an additional "X" could now be squeezed in this example.



iii. A classic symptom of memory fragmentation is that you try to allocate a large block and you can't, even though you appear to have enough memory free. On systems with virtual memory however, this is less of a problem, because large allocations only need to be contiguous in virtual address space, not in physical address space.

f. Allocating Dynamic Memory

i. To reserve memory on the heap, you can use one of the two following functions (the header file **stdlib.h** or **malloc.h** must be included to use theser):

1. **malloc** (stands for memory allocation)
    a. Used to dynamically allocate a single large block of memory with a specified size. It returns a pointer of type void which should be cast into a pointer of a desired type.

2. **calloc** (stands for cleared memory allocation)
    a. Used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value '0'.

For both functions if the space for the allocation is insufficient, a NULL pointer is returned.

```
pointer_name = (cast-type*) malloc(size);
pointer_name = (cast-type*) calloc(num_elems, size_elem);
```

ii. The **sizeof** command is a convenient way of specifying the amount of memory (in bytes) needed to store a certain data type. For an **int**, sizeof returns 4. To retrieve the entire block of memory that has been reserved, we need to know the data type and the way to achieve this with malloc is by casting the return pointer.

```
int *p = (int*)malloc(sizeof(int));
printf("address=%p, value=%d\n", p, *p);
```

iii. At compile time, only the space for the pointer is reserved (on the stack). When the pointer is initialized, a block of memory of sizeof(int) bytes is allocated (on the heap) at program runtime. The pointer on the stack then points to this memory location on the heap.

iv. Memory for Arrays and Structs

1. Since arrays and pointers are displayed and processed identically internally, individual blocks of data can be accessed using array syntax:

```
// Allocates 3 int variables on heap
int *p = (int*)malloc(3 * sizeof(int));
p[0] = 1; p[1] = 2; p[2] = 3;

// Resize memory to hold hour integers
p = (int*)realloc(p, 4 * sizeof(int));

// Display data stored in first integer
printf("address=%p, second value=%d\n", p, p[1]);
```

2. After defining the **struct** which contains a number of data primitives, a block of memory four times the size of **MyStruct** is created using the calloc command. This allocates four MyStruct variables.

```
struct MyStruct {
    int i;
    double d;
```

```
    char a[5];
};

MyStruct *p = (MyStruct*)calloc(4,sizeof(MyStruct));

// Initializes fields of the first MyStruct struct.
p[0].i = 1; p[0].d = 3.14159; p[0].a[0] = 'a';
```

56. Freeing up Memory
    a. If memory has been reserved, it should also be released as soon as it is no longer needed. If you don't do this then the RAM memory will get fully used up eventually. When that happens the data is swapped out to the hard disk, which will significantly slow down your computer.
    b. Use the **free** function to release the reserved memory area.

```
 i.   void *p = malloc(100); // Allocate 100 bytes to p
ii.   free(p)                // Deallocate those 100 bytes
```

    c. **free** can only free memory that was reserved with malloc or calloc.
    d. **free** can only release memory that has not been released before. If you try to free and freed block of memory then you'll see this error message.

        i. `free(41143,0x1000a55c0) malloc: *** error for object 0x1003001f0: pointer being freed was not allocated.`

    e. Dangling Pointer
        i. When a pointer still holds the address to the memory location which has been freed, but may not access it anymore.
            1. For example:
                a. void *p = **malloc**(100)
                free(p)
                void *p2 = p `// p2 is invalid at this point.`
                free(p2) `// p2 is a dangling pointer.`

57. Object-oriented memory management

    a. When an object is created, its constructor needs to be called to allow for member initialization. Also, on object deletion, the destructor is called to free resources and to allow for programmer-defined clean-up tasks. For this reason, C++

introduces the operators **new/delete**. Whenever you call new you must also call delete eventually, to prevent memory leaks.

```
#include <stdlib.h>
#include <iostream>

class MyClass
{
private:
    int *_number;

public:
    MyClass()  // constructor
    {
        std::cout << "Allocate memory\n";
        _number = (int *)malloc(sizeof(int));
    }
    ~MyClass() // destructor
    {
        std::cout << "Delete memory\n";
        free(_number);
    }
    void setNumber(int number)
    {
        *_number = number;
        std::cout << "Number: " << _number << "\n";
    }
};


int main()
{
    // allocate memory using malloc
    // comment these lines out to run the example below
    MyClass *myClass = (MyClass *)malloc(sizeof(MyClass));
    myClass->setNumber(42); // ERROR EXC_BAD_ACCESS
    free(myClass);

    // allocate memory using new
    MyClass *myClass = new MyClass();
    myClass->setNumber(42); // works as expected
    delete myClass;

    return 0;
}
```

b. Differences between **malloc/free** and **new/delete:**
   i. **Constructors / Destructors**: Unlike `malloc( sizeof(MyClass ) )`, the call `new MyClass()` calls the constructor. Similarly, `delete` calls the destructor.

   ii. **Type safety:** malloc returns a void pointer, which needs to be cast into the appropriate data type it points to. This is not type safe, as you can freely vary the pointer type without any warnings or errors from the compiler as in the following small example:

```
MyObject *p = (MyObject*)malloc(sizeof(MyObject));
```

c. Optimizing Performance with "**placement new**"

   i. In some cases, it makes sense to separate memory allocation from object construction. Consider a case where we need to reconstruct an object several times. If we were to use the standard new/delete construct, memory would be allocated and freed unnecessarily as only the content of the memory block changes but not its size. By separating allocation from construction, we can get a significant performance increase.
   ii. C++ allows us to do this by using a construct called *placement new*: With placement new, we can pass a preallocated memory and construct an object at that memory location. Consider the following code:

```
void *memory = malloc(sizeof(MyClass));
MyClass *object = new (memory) MyClass;
```

   iii. The syntax new (memory) is denoted as *placement new*. The difference to the "conventional" new we have been using so far is that no memory is allocated. The call constructs an object and places it in the assigned memory location. There is however, no delete equivalent to placement new, so we have to call the destructor explicitly in this case instead of using delete as we would have done with a regular call to new:

```
object->~MyClass();
free(memory);
```

   Important: Note that this should never be done outside of placement new.

d. Overloading **new** and **delete**

i. One of the major advantages of new/delete over free/malloc is the possibility of overloading. While both malloc and free are function calls and thus can not be changed easily, new and delete are operators and can thus be overloaded to integrate customized functionality, if needed.

   The syntax for overloading the new operator looks as follows:

```
void* operator new(size_t size);
```

ii. The operator receives a parameter size of type size_t, which specifies the number of bytes of memory to be allocated. The return type of the overloaded new is a void pointer, which references the beginning of the block of allocated memory.

   The syntax for overloading the delete operator looks as follows:

```
void operator delete(void*);
```

iii. The operator takes a pointer to the object which is to be deleted. As opposed to new, the operator delete does not have a return value.

iv. Why overload these in the first place?

1. **Add additional parameters**. A class can have multiple overloaded new operator functions.*(I don't understand why you don't just overload your constructor???)*
2. **Integrate a mechanism similar to garage collection.**
3. **Exception handling capabilities** can be added into new and delete. (*I don't understand why you don't just overload your constructor???)*

4. Helpful External Reading: https://www.geeksforgeeks.org/overloading-new-delete-operator-c/

58. Debugging memory leaks with Valgrind

a. Valgrand is a free software for linux and mac that is able to automatically detect memory leaks. Windows programmers can use the Visual Studio debugger and C Run-time (CRT) to detect and identify memory leaks.

// Come back to this section in the course and fill this out if you need to use Valgrind for Project 3.

Resource Copying Policies

Code Examples location:
\ProjectsUdacityCPP\CopyingPolicies

59. Copy Semantics

a. In C++, a common way of safely accessing resources (such as multi-threaded locks, files, network or database connections, and memory) is by wrapping a manager class around the handle, which is initialized when the resource is acquired (in the class constructor) and released when it is deleted (in the class destructor). This concept is often referred to as **Resource Acquisition is Initialization (RAII)**.
   i. A problem with this approach though is that copying the manager object will also copy the handle of the resource. This allows two objects access to the same resource.

b. The default behavior of both copy constructor and assignment operator is to perform a **shallow copy**. This does not consider the "special" needs of a class which allocates and deallocates a shared resource on the heap. This could lead to segmentation faults and program crashes.

60. Copy-ownership Policy

a. Creating a tailored copy constructor as well as a copy assignment operator. This copying process must be closely linked to the respective resource release mechanism *(i.e. Destructors)*

61. No copying Policy
    a. Forbids copying and assigning class instances all together. This can be achieved by declaring, but not defining a private copy constructor and assignment operator *(or by making both public and assigning the **delete** operator)*

62. Exclusive Ownership Policy
    a. Whenever a resource management object is copied, the resource handle is transferred from the source pointer to the destination pointer. The source pointer is set to **nullptr** to make ownership exclusive. At any time, the resource handle belongs only to a single object, which is responsible for its deletion when it is no longer needed.

63. Deep Copying Policy
    a. The idea is to allocate proprietary memory in the destination object and then to copy the content to which the source object handle is pointing into the newly allocated block of memory. This way, the content is preserved during copy or assignment.

64. Shared ownership policy
    a. The idea is to perform a copy or assignment similar to the default behavior, i.e. copying the handle instead of the content (as with a shallow copy) while at the same time keeping track of the number of instances that also point to the same resource. Each time an instance goes out of scope, the counter is decremented. Once the last object is about to be deleted, it can safely deallocate the memory resource.
        i. This is the central idea of **unique_ptr**, which is a representative of the group of **smart pointers.**

65. The Rule of Three

Code Examples location:
\ProjectsUdacityCPP\RuleOfThree

a. If a class needs to have an overloaded **copy constructor**, **copy assignment** operator, ~or~ **destructor**, then it must also implement the other two as well to ensure that memory is managed consistently.

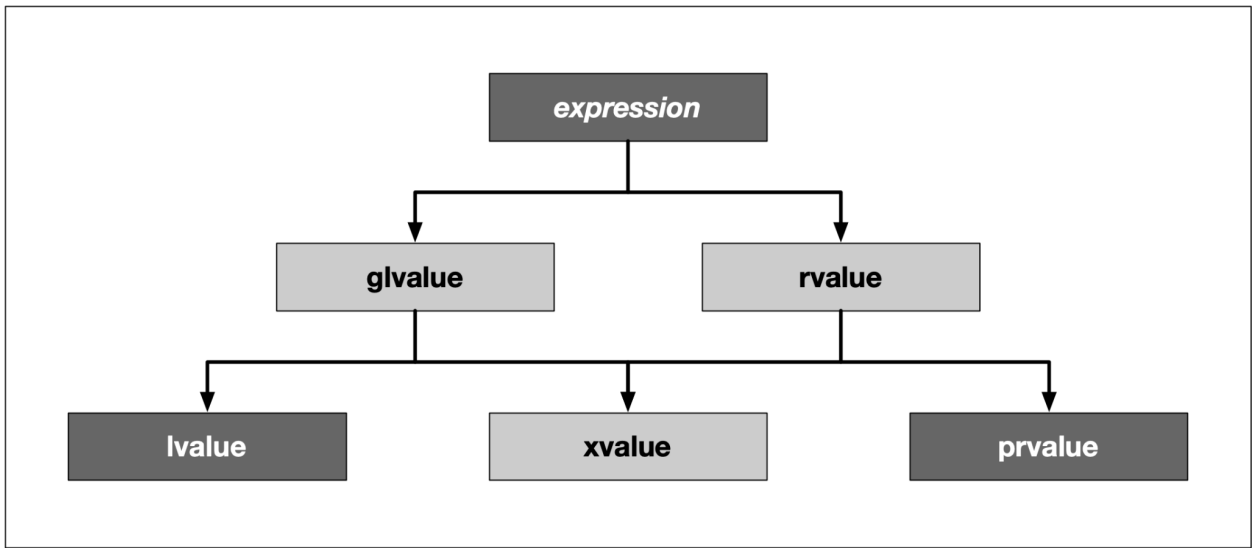
66. Lvalues and Rvalues

a. Understanding lvalues and rvalues is essential to understanding the more advanced concepts of rvalue references and motion semantics. Motion semantics is a prerequisite for smart pointers.

   i. Concept learning order:
      1. Copy policies > Lvalues/Rvalues > Motion Semantics > Smart Pointers

b. Let us start by stating that every expression in C++ has a type and belongs to a value category. When objects are created, copied or moved during the evaluation of an expression, the compiler uses these value expressions to decide which method to call or which operator to use.
   i. Deeper explanations:
      1. https://medium.com/@dhaneshvb/how-to-crack-the-confusing-world-of-lvalues-and-rvalues-in-c-it-is-easy-61c32ced51ce
      2. https://docs.microsoft.com/en-us/cpp/cpp/lvalues-and-rvalues-visual-cpp?view=msvc-160

c. Prior to C++ 11, there were only two value categories, now there are as many as five of them:



d. However, we will only cover the main two:
   i. **Lvalues**
      1. Have an address that can be accessed. They are expressions whose evaluation by the compiler determines the identity of objects or

functions. *(An lvalue is an entity that points to a specific memory location, i.e named containers for rvalues.)*

ii. **Prvalues(aka rvalues)**

1. Do not have an address that is accessible directly. They are temporary expressions used to initialize objects or compute the value of the operand of an operator. *(An rvalue is usually a short-lived object, which is only needed in a narrow local scope.)*

e. The two characters l and r are derived from the perspective of the assignment operator =

i. `int i = 42;  // lvalue = rvalue;`

f. Lvalue References

i. The code sample below declares an integer `i` and a reference `j` which can be used as an alias for the existing object.

```
#include <iostream>

int main()
{
    int i = 1;
    int &j = i;
    ++i;
    ++j;

    std::cout << "i = " << i << ", j = " << j <<
std::endl;

    return 0;
}
```

ii. The lvalue reference `j` can be used just as `i` can. A change to either will affect the same memory location on the stack. The output of the code above is i = 3, j = 3

iii. One of the primary use-cases for **lvalue references** is the pass-by-reference semantics in function calls as in the example below.

```
#include <iostream>

void myFunction(int &val)
{
    ++val;
}


// myFunction has an lvalue reference as a
parameter, which establishes an alias to the
integer i which is passed to it in main.
int main()
{
    int i = 1;
    myFunction(i);

    std::cout << "i = " << i << std::endl;

    return 0;
}
```

g. Rvalue references

i. In the code below, the function myFunction takes an lvalue reference as its argument.

```
#include <iostream>

void myFunction(int &val)
{
    std::cout << "val = " << val << std::endl;
}

int main()
{
    int j = 42;
    myFunction(j);

    myFunction(42);
```

```
    int k = 23;
    myFunction(j+k);

    return 0;
}
```

ii. In main, the call myFunction(j) works just fine while myFunction(42) as well as myFunction(j+k) produces the following compiler error on Mac:

```
candidate function not viable: expects an l-value
 for 1st argument
```

iii. and the following error in the workspace with g++:

```
error: cannot bind non-const lvalue reference of type
‘int&’ to an rvalue of type ‘int’
```

iv. While the number 42 is obviously an rvalue, with j+k things might not be so obvious, as j and k are variables and thus lvalues. To compute the result of the addition, the compiler has to create a temporary object to place it in - and this object is an rvalue.

1. Detailed explanations:
    a. http://thbecker.net/articles/rvalue_references/section_01.html

v. Since C++11, there is a new type available called **rvalue reference**, which can be identified from the double ampersand **&&** after a type name. With this operator, it is possible to store and even modify an rvalue, i.e. a temporary object which would otherwise be lost quickly.

```
#include <iostream>

int main()
{

    int i = 1;
    int j = 2;
```

```
    // The sum i+j is created as an rvalue,
      which holds the result of the addition
      before being copied into the memory location
      of k on the stack
    int k = i + j;



    // Instead of first creating the rvalue i+j ,
      then copying it and finally deleting it, we
      can now hold the temporary object in memory.
    int &&l = i + j;

    std::cout << "k = " << k << ", l = " << l <<
std::endl;

    return 0;
}
```

67. Move Semantics

Code Examples location:

\ProjectsUdacityCPP\MoveSemantics

a. A mighty technique in modern C++ *(C++ 11 and up)* to **optimize memory usage and processing speed**. Move semantics and rvalue references make it possible to write code that transfers resources such as dynamically allocated memory from one object to another in a very efficient manner and also supports the concept of **exclusive ownership**, both of which are important to understanding **smart pointers**.

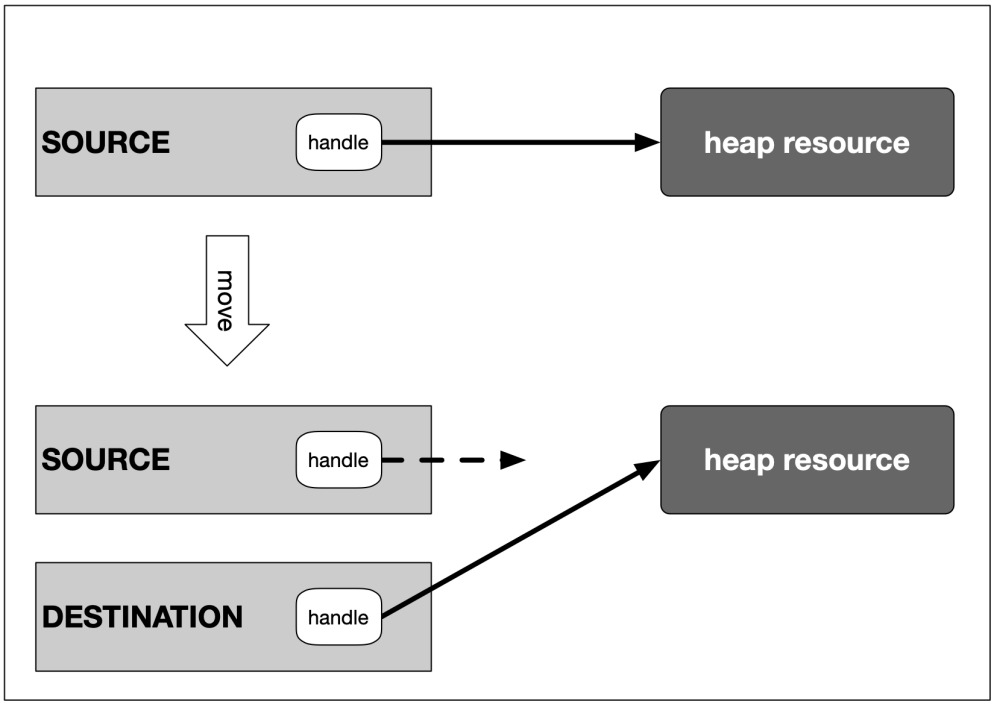
   i. *Put simply: The idea is to create the object a single time and then move it around using rvalue references.*

b. RValue references are themselves LValues.

c. The move constructor

   i. The basic idea to optimize the code in **rule_of_three.cpp** for obj4 and obj5 is to "steal" the rvalue generated by the compiler during the return-by-value operation and move the expensive data in the source object to the target object - not by copying it but by redirecting the data handles.

   ii. Moving data in such a way is always cheaper than making copies, which is why programmers are highly encouraged to use this powerful tool.



iii. To achieve this, we will be using a construct called **"move constructor"**, which is similar to the copy constructor with the key difference being the re-use of existing data without unnecessarily copying it.

iv. The **"move assignment operator"** works in a similar way. The data handle is copied from source to target. Afterwards, the data members of source are invalidated. The rest of the code is identical with the copy constructor we have already implemented. *(code for move constructor and move assignment operator are also in the rule_of_three.cpp file in \ProjectsUdacityCPP)*

1. Source argument of this function will always be an **rvalue reference**.

d. The Rule of Five

i. If you have to write one of the functions listed below then you should consider implementing all of them with a proper resource management policy in place. *(If you forget to implement one or more, the compiler will usually generate the missing ones, but the default versions might not be suitable for the purpose you have in mind.)*

1. **Destructor**
2. **Assignment Operator**
3. **Copy Constructor**

^ Rule of three

4. **Move Constructor**
5. **Move assignment Operator**

ii. The rule of five is especially important in resource management, where unnecessary copying needs to be avoided due to limited resources and performance reasons.

e. When are move semantics used
   i. One of the primary areas of application are cases, where heavy-weight objects need to be passed around in a program. Copying these without move semantics can cause serious performance issues.
   ii. Another application is where ownership needs to be transferred (such as with unique pointers). The primary difference to shared references is that with move semantics we are not sharing anything but instead we are ensuring through a smart policy that only a single object at a time owns the resource.

f. Moving lvalues
   i. It can make sense to treat lvalues like rvalues. At some point in your code, you might want to transfer ownership of a resource to another part of your program as it is not needed anymore in the current scope. For this you can use **std::move**
      1. This function accepts an lvalue argument and returns it as an rvalue without triggering copy construction.

```
void useObject(MyMovableClass obj)
{
    std::cout << "using object " << &obj << std::endl;
}



int main()
{
    MyMovableClass obj1(100); // constructor

    useObject(std::move(obj1));

    return 0;
}
```

g. Note that after the call to useObject, the instance obj1 has been invalidated by setting its internal handle to null and thus may not be used anymore within the scope of main.

Smart Pointers

Code Examples location:
\ProjectsUdacityCPP\SmartPointers

68. Problems with **new** and **delete**
    a. **A proper pairing of new and delete:**
        i. Every dynamically allocated object that is created with new must be followed by a manual deallocation at a "proper" place in the program or else you will have memory leaks.

    b. **Correct operator pairing:**
        i. For example a dynamically allocated array initialized with **new[]** may only be deleted with the operator **delete[]**. If the wrong operator is used, program behavior will be undefined.

    c. **Memory ownership**
        i. If a third-party function returns a pointer to a data structure, the only way of knowing who will be responsible for resource deallocation is by looking into either the code or the docs. If both are not available, there is no way to infer the ownership from the return type.
            1. *(You could have a framework that handles its own object deletions but if the programmer doesn't know this they might try to call delete and interfere with the inner workings of the library.)*

69. Benefits of smart pointers

a. When a smart pointer is no longer needed (which is the case as soon as it goes out of scope), the memory to which it points is automatically deallocated. This makes them easier to use than “conventional” or “raw” pointers.

b. Smart pointers are classes that are wrapped around raw pointers. By overloading the **->** and ***** operators, smart pointer objects make sure that the memory that their internal raw pointer refers to is properly deallocated. This makes it possible to use smart pointers with the same syntax as raw pointers.
    i. This technique of wrapping a management class around a resource is called **Resource Acquisition Is Initialization (RAII)**

70. RAII

a. The major idea of RAII revolves around object ownership and information hiding. Allocation and deallocation are hidden within the **management class**, so a programmer using the class does not have to worry about memory management responsibilities *(e.g new and delete should disappear from the surface level code)*. Whoever owns a resource deals with it.

b. RAII is a widespread **programming paradigm** that can be used to protect a resource such as a file stream, a network connection or a block of memory which needs proper management.

c. In most programs of reasonable size, there will be many situations where a certain action at some point will need a proper reaction at another point, such as:

	Acquire	Release
Files	`fopen`	`fclose`
Memory	`new, new[]`	`delete, delete[]`
Locks	`lock, try_lock`	`unlock`

    i. Problems that RAII solve:

1. The program might throw an exception during resource use and thus the point of release might never be reached
2. We might forget to release the resource
3. There might be several points where the resource could potentially be released, making it hard for a programmer to keep track of all eventualities.

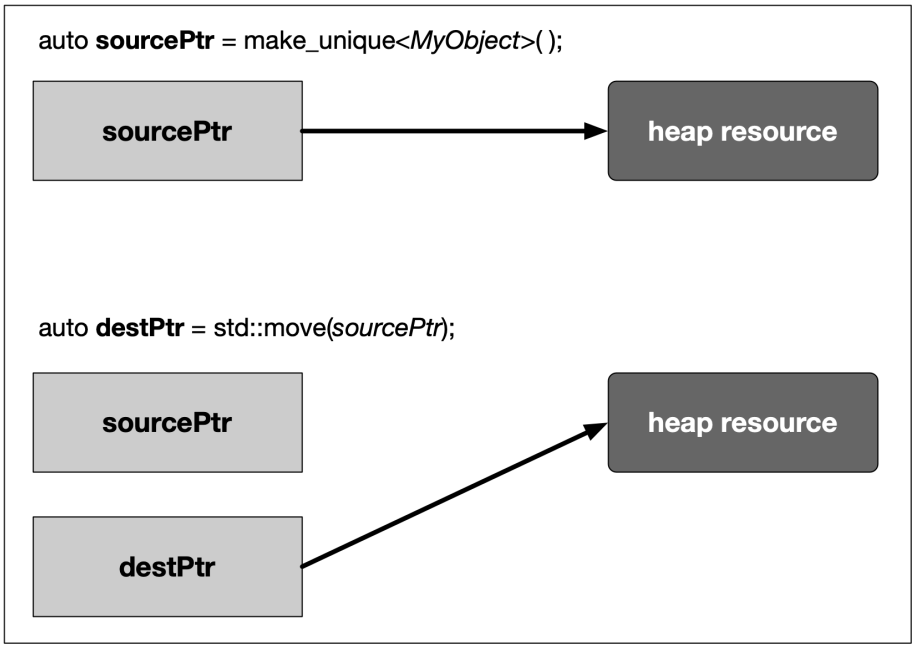
71. Smart Pointers
    a. Modern C++ includes smart pointes, which help to ensure that programs are free of memory leaks while also remaining exception-safe. With smart pointers, resource acquisition occurs at the same time that the object is initialized (when instantiated with **make_shared** or **make_unique**), so that all resources for the object are created and initialized in a single line of code.
    b. Types of smart pointers
        i. **std::unique_ptr** exclusively owns a dynamically allocated resource on the heap. There must not be a second unique pointer to the same resource.

        ii. **std::shared_ptr** points to a heap resource but does not explicitly own it. There may even be several shared pointers to the same resource, each of which will increase as internal reference count. As soon as this count reaches zero, the resource will automatically be deallocated.

        iii. **std::weak_ptr** behaves similar to the shared pointer but does not increase the reference counter

    *Note: Prior to C++ 11 there was a concept called **std::auto_ptr** but it has been deprecated and should not be used.*

    c. Unique Pointer
        i. There must not be a second unique pointer to the same memory resource, otherwise there will be a compiler error. As soon as the unique pointer goes out of scope, the memory resource is deallocated again.

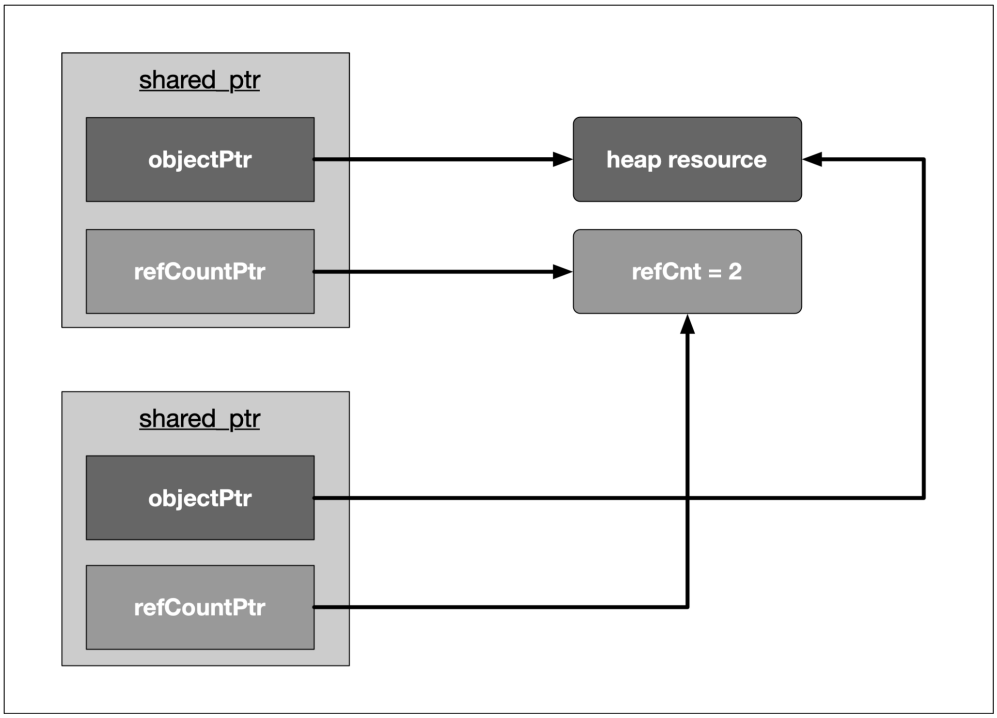


ii. Unique pointers should be the default choice unless you know for certain that sharing is required at a later stage. Internally, the unique pointer uses the Rule of Five and Move semantics, along with RAII to encapsulate a resource (the raw pointer) and transfer it between pointer objects when either the move assignment operator or the move constructor are called.

iii. Also, a key feature of a unique pointer, which makes it so well-suited as a return type for many functions, is the possibility to convert it to a shared pointer.

d. Shared Pointer

i. This smart pointer type is useful for cases where you require access to a memory location on the heap in multiple parts of your program and you want to make sure that whoever owns a shared pointer to the memory can rely on the fact that it will be accessible throughout the lifetime of that pointer.



e. Weak Pointer
   i. Similar to shared pointers, there can be multiple weak pointers to the same resource. The main difference is that weak pointers do not increase the reference count. Weak pointers hold a non-owning reference to an object that is managed by another shared pointer.
   ii. **Rule**:
      1. You can only create a weak pointer out of a shared pointer or out of another weak pointer.
         a. ***Good***

```
std::shared_ptr<int> mySharedPtr(new int);
std::weak_ptr<int> myWeakPtr1(mySharedPtr);
std::weak_ptr<int> myWeakPtr2(myWeakPtr1);
```

         b. ***Bad***

```
std::weak_ptr<int> myWeakPtr3(new int); // Will
ERROR
```

72. Converting Between Smart Pointer Types

Code Examples location:
\ProjectsUdacityCPP\SmartPointers\SmartPointerConversion

a. In some cases it might be necessary to convert from one smart pointer type to another. There code examples in the location above shows how to do this along with detailed comment explanations of what is happening under the hood.
b. However, there are no options for converting away from a **shared pointer**. Once you have created a shared pointer, you must stick to it *(or a copy of it) for the remainder of your program.*

73. When to use raw pointers and smart pointers?

a. **As a general rule of thumb with modern C++, smart pointers should be used over raw pointers in most cases. However, raw pointers have their place also.**
    i. C++ core guidelines for resource management
    http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#r-resource-management

        1. Rule 10 : Avoid malloc and free
        2. Rule 11 : Avoid calling new and delete explicitly
        3. Rule 12 : Immediately give the result of an explicit resource allocation to a manager object

b. **Rules for using smart pointers**
    http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#rsmart-smart-pointers

        1. Rule 20 : Use unique_ptr or shared_ptr to represent ownership
        2. Rule 21 : Prefer unique_ptr over std::shared_ptr unless you need to share ownership

74. Transferring Ownership

a. Take smart pointers as parameters only to explicitly express lifetime semantics
    i. The core idea behind this rule is the notion that functions that only manipulate objects without affecting its lifetime in any way should not be concerned with a particular kind of smart pointer. A function that does not manipulate the lifetime or ownership should use raw pointers or references instead.
    ii. A function should take smart pointers as parameters only if it examines or manipulates the smart pointer itself. As we have seen, smart pointers are

classes that provide several features such as counting references of a **shared_ptr** or increasing them by making a copy.

iii. Also, data can be moved from one **unique_ptr** to another and thus transferring the ownership. A particular function should accept smart pointers only if it expects to do something of this sort. If a function just needs to operate on the underlying object without the need of using any smart pointer property, it should accept the objects via raw pointers references instead.

1. The following examples are **pass-by-value** types that lend the ownership of the underlying object:

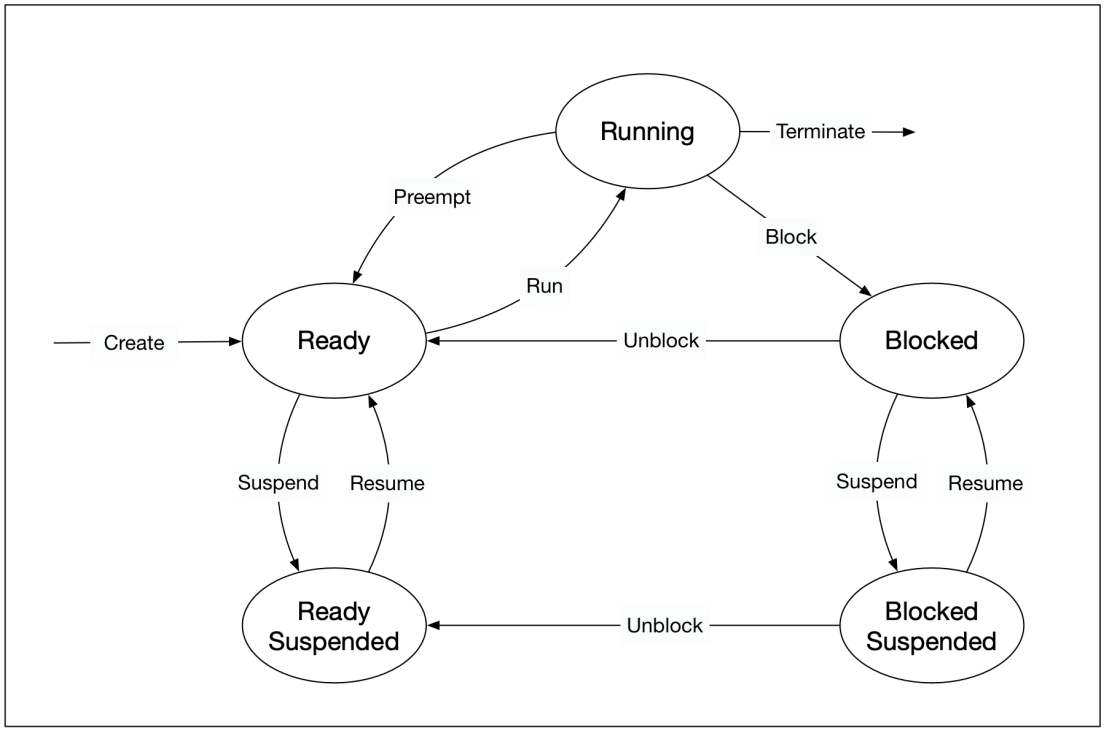
```
a. void f(std::unique_ptr<MyObject> ptr)
b. void f(std::shared_ptr<MyObject> ptr)
c. void f(std::weak_ptr<MyObject> ptr)
```

2. Passing smart pointers by value means to lend their ownership to a particular function f. In the above examples 1-3, all pointers are passed by value *(i.e, the function f has a private copy of it which it should modify)*.

Concurrency

75. Processes

a. A process (also called a Task) is a computer program at runtime. It consists of the runtime environment provided by the operating system, as well as the embedded binary code of the program during execution.
b. A process is controlled by the OS through certain actions with which it sets the process into one of several carefully defined states:



c. **Ready:** After its creation, a process enters the ready state and is loaded into main memory. The process now is ready to run and is waiting for CPU time to be executed. Processes that are ready for execution by the CPU are stored in a queue managed by the OS.

d. **Running:** The OS has selected the process for execution and the instructions within the process are executed on one or more of the available CPU cores.

e. **Blocked:** A process that is blocked is one that is waiting for an event (such as a system resource becoming available) or the completion of an I/O operation.

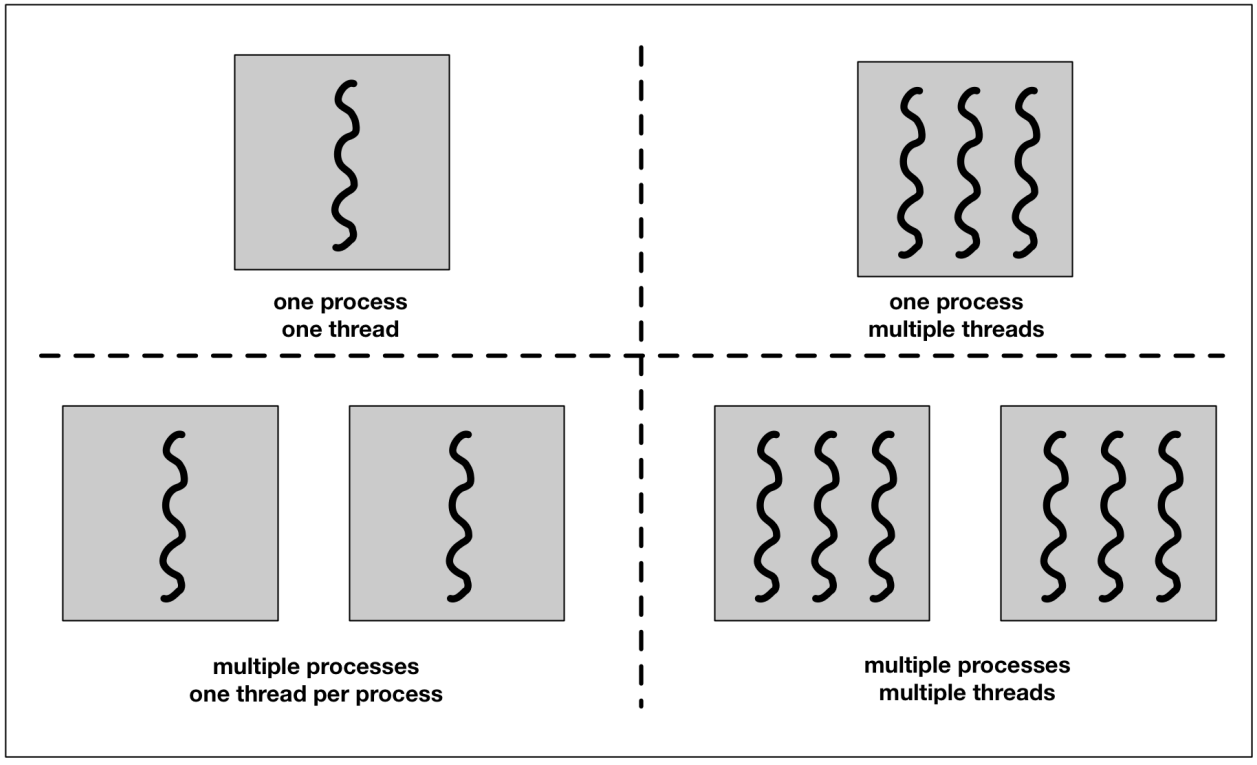
f. **Terminated:** When a process completes its execution or when it is being explicitly killed, it changes to the "terminated" state. The underlying program is no longer executing, but the process remains in the process table as a "zombie process". When it is finally removed from the process table, its lifetime ends.

g. **Ready Suspended:** A process that was initially in ready state but has been swapped out of main memory and placed onto external storage is said to be in suspend ready state. The process will transition back to ready state whenever it is moved to main memory again.

h. **Blocked Suspended**: A process that is blocked may also be swapped out of main memory. It may be swapped back in again under the same conditions as a "ready suspended" process. In such a case, the process will move to the blocked state, and may still be waiting for a resource to become available.

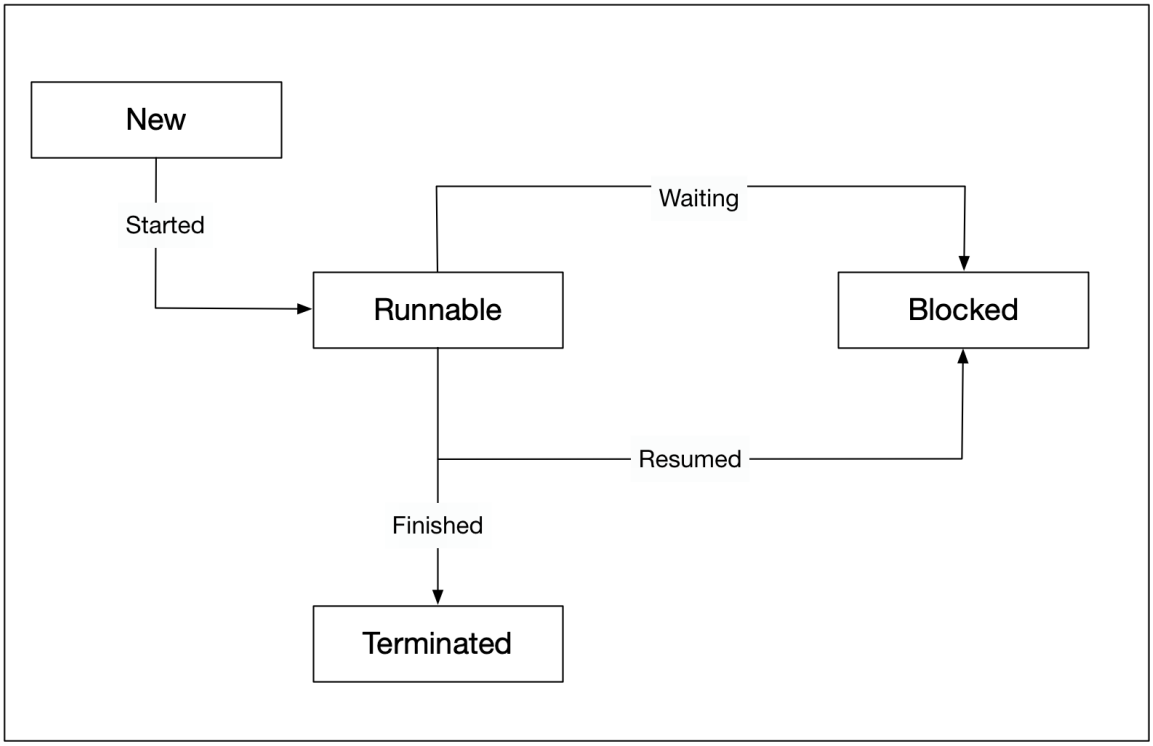
i. Scheduler
    i. The component of the OS that manages all Processes.
    ii. The scheduler can either let a process run until it ends or blocks (non-interrupting scheduler), or it can ensure that the currently running process is interrupted after a short period of time.
    iii. The scheduler can switch back and forth between different active processes (interrupting scheduler), alternately assigning them CPU time.

76. Threads
    a. A concurrent execution unit within a process.
    b. Threads are characterized as **light-weight processes (LWP)**. In many systems the creation of a thread is up to 100 faster than the creation of a process. This is especially advantageous in situations, when the need for concurrent operations changes dynamically.



c. Threads exist within processes and share their resources. As shown above, a process can contain several threads or - if no parallel processing is provided for the program flow - only a single thread.

d. A major difference between a process and a thread is that each process has its own address space, while a thread does not require a new address space to be created. All the threads in a process can access its shared memory. Threads also share other OS dependent resources such as processors, files, and network connections. As a result, the management overhead for threads is typically less than for processes. Threads, however, are not protected against each other and must carefully synchronize when accessing the shared process resources to avoid conflicts.



e. Thread states:
   i. **New:** A thread is in this state once it has been created. Until it is actually running, it will not take any CPU resources.
   ii. **Runnable:** In this state, a thread might actually be running or it might be ready to run. It is the responsibility of the thread scheduler to assign CPU time to the thread.
   iii. **Blocked:** A thread might be in this state, when it is waiting for I/O operations to complete. When blocked, a thread cannot continue its execution any further until it is moved to the runnable state again. It will

not consume any CPU time in this state. The thread scheduler is responsible for reactivating the thread.

f.

- How does C++ handle dependencies? (npm install equivalent)
    - C++ Boost Libraries
- How to build a cmake file for multi-file programs?
    - (*add_executable() for each class? Or just parent dir?*)
- How to build a library in C++ ?
    - https://stackoverflow.com/questions/8153519/how-to-automatically-download-c-dependencies-in-a-cross-platform-way-cmake
- What is the difference between a module and a library ?
- What is a Functor?
- CPP Unit
- CPP Check (static code analysis tool)
- CPP Style guide (find the best one and make your C++ code syntax follow those guidelines)
- How to see the assembly of your C++ build files?

```
C:\> objdump -D a.exe
```

- When you finish this course. Download all of the videos from each sections resources tab.