CSCI 2270: Data Structures

Lecture 04: C++ Review: Streams, Structs, and Functions

Ashutosh Trivedi

Department of Computer Science
UNIVERSITY OF COLORADO BOULDER

C++ Review: Streams

C++ Review: Arrays and Structs

C++ Review: Functions and Memory

Streams: basic input-output mechanism

- *Streams* are the sequence of bytes flowing in and out of the programs (just like water and oil flowing through a pipe).

Streams: basic input-output mechanism

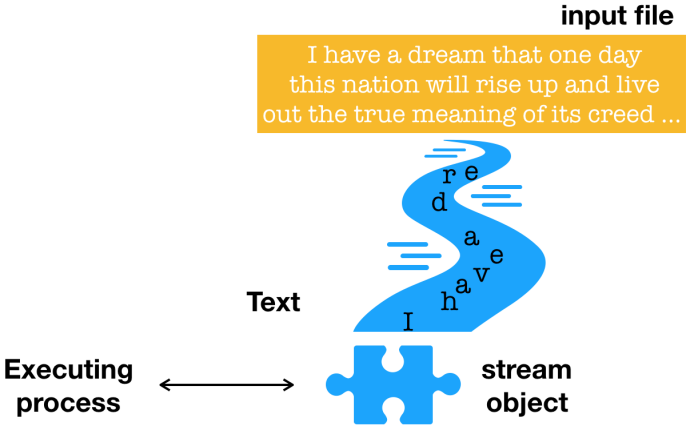
- *Streams* are the sequence of bytes flowing in and out of the programs (just like water and oil flowing through a pipe).
- In *input operations*, data bytes flow from an input source (keyboard, file, network, strings, and other programs) to the program, and

Streams: basic input-output mechanism

- *Streams* are the sequence of bytes flowing in and out of the programs (just like water and oil flowing through a pipe).
- In *input operations*, data bytes flow from an input source (keyboard, file, network, strings, and other programs) to the program, and
- in output operations, data bytes flow from the program to an output sink (such as console, file, network or another program).
- Streams act as an abstract interface between the program and the actual IO devices in such a way that frees a programmer from hardware concerns.

Streams: basic input-output mechanism

- *Streams* are the sequence of bytes flowing in and out of the programs (just like water and oil flowing through a pipe).



Streams: basic input-output mechanism

- We have seen how to use the standard I/O stream by including the following pre-compiler directive:

```
#include <iostream>
```

Streams: basic input-output mechanism

- We have seen how to use the standard I/O stream by including the following pre-compiler directive:

```
#include <iostream>
```

- In order to output to screen, we merely use a statement like,

```
std::cout << " X = "<< X;
```

Streams: basic input-output mechanism

- We have seen how to use the standard I/O stream by including the following pre-compiler directive:

```
#include <iostream>
```

- In order to output to screen, we merely use a statement like,

```
std::cout << " X = "<< X;
```

- The “insertion operator” (<<) points in the direction of data flow.

Streams: basic input-output mechanism

- We have seen how to use the standard I/O stream by including the following pre-compiler directive:

```
#include <iostream>
```

- In order to output to screen, we merely use a statement like,

```
std::cout << " X = "<< X;
```

- The "insertion operator" (<<) points in the direction of data flow.
- Similarly, input stream reads from the keyboard into a variable.
- The extraction operator (>>) is "smart enough" to consider the type of the target variable when it determines how much to read from the input stream.

Input from the keyboard

```
// program14.cpp
#include<iostream>
#include<string>
int main()
{
  std::cout << "Please input a string, an integer, a character, and a float (space separated): \n";
  std::string w;
  int x;
  char y;
  float z;
  std::cin >> w;
  std::cin >> x;
  std::cin >> y;
  std::cin >> z;
  std::cout << "w= " << w << " x= " <<  x << " y=" << y  <<  " z= " << z << std::endl;
  return 0;
}
```

File Output

```cpp
// program15.cpp
#include <iostream>
#include <fstream>
using namespace std;
int main () {
  ofstream myfile;
  myfile.open("example.txt");
  myfile << "Writing this to a file.\n";
  myfile.close();
  return 0;
}
```

File Output: Parameters (Append)

```
// program16.cpp
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile("example.txt", ios::binary | ios::app | ios::out);
  myfile << "Writing this to a file.\n";
  myfile.close();
  return 0;
}
```

Notice:

- The "constructor" instead of `open()`.
- Parameters passed while opening the file.

File Input

```cpp
// program17.cpp
#include <iostream>
#include <fstream>
int main (int argc, char *argv[]) {
  std::ifstream fin("addresses.txt");
  if (fin.is_open()) {
    std::cout << "File is open as fin stream\n";
    char c;
    fin >>  c;
    std::cout << "first char is " << c << " \n";
  }
  else std::cerr << "File addresses.txt not found!";
  fin.close(); // Don't forget to close!
  return 0;
}
```

File Input: Eat it line by line!

```cpp
// program18.cpp
#include <iostream>
#include <fstream>
#include <string>
int main (int argc, char *argv[]) {
  std::ifstream fin("addresses.txt");
  if (fin.is_open()) {
    std::string line;
    while (getline(fin, line)) {
      std::size_t found = line.find("TX");
      if (found!=std::string::npos) {
  std::cout << line << std::endl;
      }
    }
  }
  else std::cerr << "File addresses.txt not found!";
  fin.close(); // Don't forget to close!
  return 0;
}
```

String Streams: input and output to strings

```
    while (getline(fin, line)) {
      std::stringstream sin(line);
      std::string id, name, phone, email, street, zip, city, state, lat, lon;
      getline(sin, id, ',');
      getline(sin, name, ',');
      getline(sin, phone, ',');
      getline(sin, email, ',');
      getline(sin, street, ',');
      getline(sin, city, ',');
      getline(sin, state, ',');
      getline(sin, zip, ',');
      getline(sin, lat, ',');
      getline(sin, lon, ' ');
      std:: cout << name  << "lives in " << state << std::endl;
    }
  }
```

C++ Review: Streams

C++ Review: Arrays and Structs

C++ Review: Functions and Memory

Arrays

- An array is a collection of elements of the same type.
- Given a variable of type T, and array of type $T[N]$ holds an array of N elements, each of type N.
- Each element of the array can be referenced by its index that is a number of 0 to $N - 1$.

Arrays

- An array is a collection of elements of the same type.
- Given a variable of type T, and array of type $T[N]$ holds an array of N elements, each of type N.
- Each element of the array can be referenced by its index that is a number of 0 to $N-1$.

```cpp
// program8.cpp
#include<iostream>
int main(int argc, char* argv[])
{
  int ia[3]; //Array of 3 ints with garbage values
  std::cout << ia[1] << std::endl;
  float fa[] = {1, 2, 3}; //Array of 3 floats initialzed: size automatically computed
  std::cout << fa[2] << std::endl; // Read different values
  return 0;
}
```

Arrays (Statically Declared Arrays)

```cpp
// program8.cpp
#include<iostream>
int main(int argc, char* argv[])
{
  int ia[3]; //Array of 3 ints with garbage values
  std::cout << ia[1] << std::endl;
  float fa[] = {1, 2, 3}; //Array of 3 floats initialzed: size automatically computed
  std::cout << fa[2] << std::endl; // Read different values
  return 0;
}
```

1. Static Array storage is contiguous.
2. Array bound must be a constant expression. If you need variable bounds, use a `vector`.
3. What happens when initialization and array size mismatch?
4. Multi-dimensional arrays (contiguous in row-order fashion!).

Structures (Our first data-structure!)

- A structure is useful for storing an aggregation of elements.
- Unlike an array, the elements of a structure may be of different types.
- Each element of field is referred by a given name.

Structures (Our first data-structure!)

- A structure is useful for storing an aggregation of elements.
- Unlike an array, the elements of a structure may be of different types.
- Each element of field is referred by a given name.

```
// program11.cpp
#include<iostream>
int main(int argc, char* argv[])  {
  struct address {
    long id; // unique ID: 1614011140000
    std::string name;  // Name: Ashutosh Trivedi
    std::string phone; //Phone number: (720) 707-9663
    std::string email; //ashutosh.trivedi@gmail.com
    std::string street; //4141 Spruce Street
    std::string city; // Philadelphia
    std::string state; //PA
    int zip;  // 19104
    float lat; //39.948610
    float lon; //-75.177830
  };
```

Structures (Our first data-structure!)

```cpp
// program11.cpp
#include<iostream>
int main(int argc, char* argv[])  {
  struct address {
    long id; // unique ID: 1614011140000
    std::string name;  // Name: Ashutosh Trivedi
    std::string phone; //Phone number: (720) 707-9663
    std::string email; //ashutosh.trivedi@gmail.com
    std::string street; //4141 Spruce Street
    std::string city; // Philadelphia
    std::string state; //PA
    int zip;  // 19104
    float lat; //39.948610
    float lon; //-75.177830
  };
  address myadress = {1614011140000, "Ashutosh Trivedi", "(720) 707-9663", "ashutosh.trivedi@gmail.com", "4141 Spruce Street","Philadelphia
        ","PA",19104,39.948610,-75.177830};
  std::cout << myadress.name << " lives in " << myadress.state << std::endl;
  return 0;
```

Structures (Constructors)

```cpp
// program20.cpp
#include<iostream>
int main(int argc, char* argv[])  {
  struct Address {
    long id; // unique ID: 16140111140000
    std::string name;  // Name: Ashutosh Trivedi
    std::string phone; //Phone number: (720) 707-9663
    std::string email; //ashutosh.trivedi@gmail.com
    std::string street; //4141 Spruce Street
    std::string city; // Philadelphia
    std::string state; //PA
    int zip;  // 19104
    float lat; //39.948610
    float lon; //-75.177830
    Address() {};
    Address(long _id, std::string _name, std::string _phone, std::string _email, std::string _street, std::string _city, std::string _state
          , int _zip,float _lat, float _lon) {
      id = _id;
      name = _name;
      phone = _phone;
      email = _email;
      street = _street;
      city = _city;
      state = _state;
      zip = _zip;
      lat = _lat;
      lon = _lon;
    }
```

Structures (Member functions)

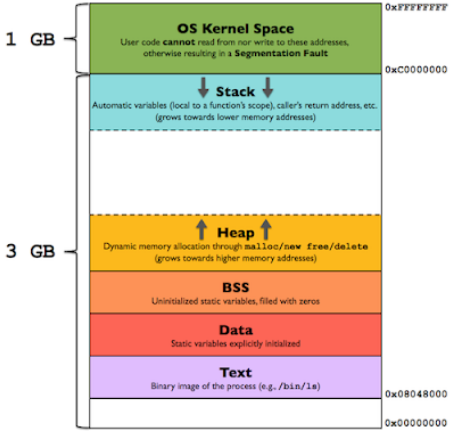
```
1  struct Address {
2    long id; // unique ID: 1614011140000
3    std::string name;  // Name: Ashutosh Trivedi
4    std::string phone; //Phone number: (720) 707-9663
5    std::string email; //ashutosh.trivedi@gmail.com
6    std::string street; //4141 Spruce Street
7    std::string city; // Philadelphia
8    std::string state; //PA
9    int zip;  // 19104
10   float lat; //39.948610
11   float lon; //-75.177830
12   Address() {};
13   Address(long _id, std::string _name, std::string _phone, std::string _email, std::string _street, std::string _city, std::string _state
           , int _zip,float _lat, float _lon) {
14     id = _id;
15     name = _name;
16     phone = _phone;
17     email = _email;
18     street = _street;
19     city = _city;
20     state = _state;
21     zip = _zip;
22     lat = _lat;
23     lon = _lon;
24   }
25   void prettyPrint() {
26     std:: cout << name << std::endl;
27     std::cout << "    Id: \n        " << id << std::endl;
28     std::cout << "    Ph: \n        " << phone << std::endl;
29     std::cout << "    E-mail: \n        " << email << std::endl;
30     std::cout << "    Address: "<< std::endl;
31     std::cout << "        "<< street << ", " << city << ", " << state << "-" << zip << std::endl;
32     std::cout << "        Location: ("<< lat << ", " << lon << ")" << std::endl;
33     std::cout<< "_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/" << std::endl;
```

C++ Review: Streams

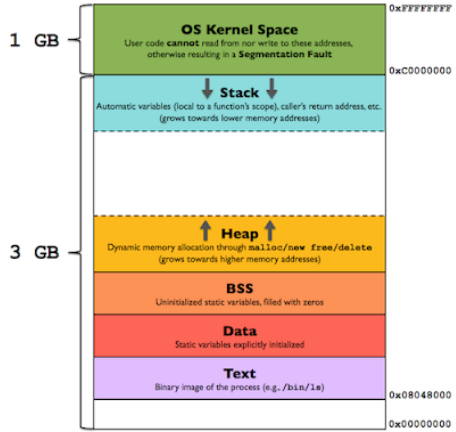
C++ Review: Arrays and Structs

C++ Review: Functions and Memory

In-Memory Layout of a Program



In-Memory Layout of a Program



- Segmentation fault: *the program has attempted to access a restricted area of memory (memory access violation)*
- core dump: *dump of the process state before segmentation fault*

Functions: Declaration and Definition

```
// program21.cpp
#include<iostream>
#include <cmath>
int square(int x);
long exponentiation(int x, int y);
int main(int argc, char* argv[])  {
  if (argc == 2)  std::cout << square(std::stoi(argv[1]));
  if (argc == 3)  std::cout << exponentiation(std::stoi(argv[1]), std::stoi(argv[2]));
  return 0;
}
int square(int x) {
  int y = x*x;
  return y;
}
long exponentiation(int x, int y) {
  long z = x;
  while (y-->1) z = z*x;
  return z;
}
```

Functions: Declaration and Definition

What is wrong with the following program?

```
// program22.cpp
#include<iostream>
#include <cmath>
int square(int x);
long exponentiation(int x, int y);
int main(int argc, char* argv[])  {
  if (argc == 2)  std::cout << square(std::stoi(argv[1]));
  if (argc == 3) {
    std::cout << exponentiation(std::stoi(argv[1]), std::stoi(argv[2]));
    std::cout << w;
    std::cout << power(std::stoi(argv[1]));
  }
  return 0;
}
int square(int x) {
  int y = x*x;
  return y;
}
long exponentiation(int x, int y) {
  long z = x;
  int w = x*y;
  while (y-->1) z = z*x;
  return z;
}
```

Functions: Declaration and Definition

What is wrong with the following program?

```
// program22.cpp
#include<iostream>
#include <cmath>
int square(int x);
long exponentiation(int x, int y);
int main(int argc, char* argv[])  {
  if (argc == 2)  std::cout << square(std::stoi(argv[1]));
  if (argc == 3) {
    std::cout << exponentiation(std::stoi(argv[1]), std::stoi(argv[2]));
    std::cout << w;
    std::cout << power(std::stoi(argv[1]));
  }
  return 0;
}
int square(int x) {
  int y = x*x;
  return y;
}
long exponentiation(int x, int y) {
  long z = x;
  int w = x*y;
  while (y-->1) z = z*x;
  return z;
}
```

- Variable w is not accessible inside the function `main`.

Functions: Declaration and Definition

What is wrong with the following program?

```
// program22.cpp
#include<iostream>
#include <cmath>
int square(int x);
long exponentiation(int x, int y);
int main(int argc, char* argv[])  {
  if (argc == 2)  std::cout << square(std::stoi(argv[1]));
  if (argc == 3) {
    std::cout << exponentiation(std::stoi(argv[1]), std::stoi(argv[2]));
    std::cout << w;
    std::cout << power(std::stoi(argv[1]));
  }
  return 0;
}
int square(int x) {
  int y = x*x;
  return y;
}
long exponentiation(int x, int y) {
  long z = x;
  int w = x*y;
  while (y-->1) z = z*x;
  return z;
}
```

- Variable w is not accessible inside the function `main`.
- Notice that the variable `argv` is still available to `main` after the exponentiation procedure returns. Why?

Functions: Factorial

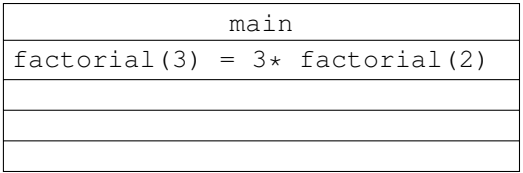
```
// program23.cpp
// Factorial(n) = n * (n-1) * (n-2) * ... * 1
#include<iostream>
#include <cmath>
int factorial(int x);
int main(int argc, char* argv[])  {
  if (argc == 2)  std::cout << factorial(std::stoi(argv[1]));
  return 0;
}
int factorial(int x) {
  int res = x;
  while (x-- > 1) res = res * x;
  return res;
}
```

Functions: Factorial (recursive)

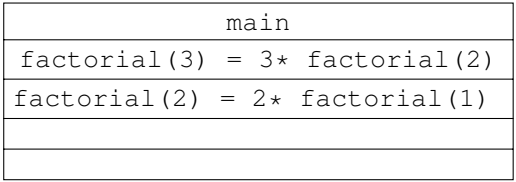
```
// program24.cpp
// Factorial(n) = n * Factorial(n-1)
#include<iostream>
#include <cmath>
int factorial(int x);
int main(int argc, char* argv[])  {
  if (argc == 2)  std::cout << factorial(std::stoi(argv[1]));
  return 0;
}
int factorial(int x) {
  int y = 1;
  std::cout << "Entering function Factorial(" << x << ")" << std::endl;
  if (x == 0) y = 1;
  else y =  x * factorial(x-1);
  std::cout << "Computed  Factorial(" << x << ")" << std::endl;
  return y;
}
```

Function

Function



Function



Function

main
factorial(3) = 3* factorial(2)
factorial(2) = 2* factorial(1)
factorial(1) = 1* factorial(0)

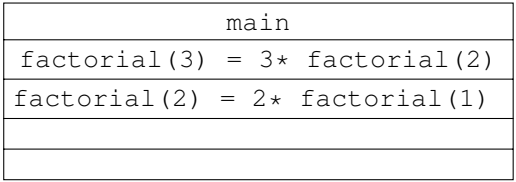
Function

main
factorial(3) = 3* factorial(2)
factorial(2) = 2* factorial(1)
factorial(1) = 1* factorial(0)
factorial(0) = 1

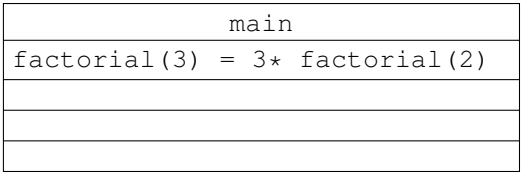
Function

main
factorial(3) = 3* factorial(2)
factorial(2) = 2* factorial(1)
factorial(1) = 1* factorial(0)

Function



Function



Function

Stack Memory

- A “stack frame” is created when a function is called. Henceforth, all the local variables of that function are created within the confines of this stack frame.
- When the function returns, its stack frame is “deleted”. The deletion of all variables happens “automagically.’
- Programmer does not need to concern herself to create or to delete copy of variables. The run-time system provides this facility.

Stack Memory

- A “stack frame” is created when a function is called. Henceforth, all the local variables of that function are created within the confines of this stack frame.
- When the function returns, its stack frame is “deleted”. The deletion of all variables happens “automagically.’
- Programmer does not need to concern herself to create or to delete copy of variables. The run-time system provides this facility.
- A big limitation of stack: how to store variables that one can access across function calls?

Stack Memory

- A “stack frame” is created when a function is called. Henceforth, all the local variables of that function are created within the confines of this stack frame.
- When the function returns, its stack frame is “deleted”. The deletion of all variables happens “automagically.’
- Programmer does not need to concern herself to create or to delete copy of variables. The run-time system provides this facility.
- A big limitation of stack: how to store variables that one can access across function calls?
- Such a memory to store global variables is called the heap memory.

Stack Memory: summary

1. The stack grows and shrinks as functions push and pop local variables.
2. There is no need to manage the memory yourself, variables are allocated and freed automagically.
3. The stack has size limits. (Check yours with `ulimit -a` and set with `ulimit -s 33333`.
4. The stack variables only exist while the function that created them, is running.

The Heap

- The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU.

The Heap

- The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU.
- To allocate memory on the heap, you must use operator `new`.

The Heap

- The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU.
- To allocate memory on the heap, you must use operator `new`.
- Once you have allocated memory on the heap, you are responsible for deleting it using `delete` operator to deallocate that memory once you don't need it any more.

The Heap

- The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU.
- To allocate memory on the heap, you must use operator `new`.
- Once you have allocated memory on the heap, you are responsible for deleting it using `delete` operator to deallocate that memory once you don't need it any more.
- If you fail to do this, your program will have what is known as a memory leak. That is, memory on the heap will still be set aside (and won't be available to other processes).

The Heap

- The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU.
- To allocate memory on the heap, you must use operator `new`.
- Once you have allocated memory on the heap, you are responsible for deleting it using `delete` operator to deallocate that memory once you don't need it any more.
- If you fail to do this, your program will have what is known as a memory leak. That is, memory on the heap will still be set aside (and won't be available to other processes).
- There are debuggers (valgrind) that can help you detect memory leaks.

The Heap

- The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU.
- To allocate memory on the heap, you must use operator `new`.
- Once you have allocated memory on the heap, you are responsible for deleting it using `delete` operator to deallocate that memory once you don't need it any more.
- If you fail to do this, your program will have what is known as a memory leak. That is, memory on the heap will still be set aside (and won't be available to other processes).
- There are debuggers (valgrind) that can help you detect memory leaks.
- Unlike the stack, the heap does not have size restrictions on variable size (apart from the obvious physical limitations of your computer).

The Heap

- The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU.
- To allocate memory on the heap, you must use operator `new`.
- Once you have allocated memory on the heap, you are responsible for deleting it using `delete` operator to deallocate that memory once you don't need it any more.
- If you fail to do this, your program will have what is known as a memory leak. That is, memory on the heap will still be set aside (and won't be available to other processes).
- There are debuggers (valgrind) that can help you detect memory leaks.
- Unlike the stack, the heap does not have size restrictions on variable size (apart from the obvious physical limitations of your computer).
- Unlike the stack, variables created on the heap are accessible by any function, anywhere in your program. Heap variables are essentially global in scope.

The Heap

- The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU.
- To allocate memory on the heap, you must use operator `new`.
- Once you have allocated memory on the heap, you are responsible for deleting it using `delete` operator to deallocate that memory once you don't need it any more.
- If you fail to do this, your program will have what is known as a memory leak. That is, memory on the heap will still be set aside (and won't be available to other processes).
- There are debuggers (valgrind) that can help you detect memory leaks.
- Unlike the stack, the heap does not have size restrictions on variable size (apart from the obvious physical limitations of your computer).
- Unlike the stack, variables created on the heap are accessible by any function, anywhere in your program. Heap variables are essentially global in scope.
- Heap memory is slightly slower to be read from and written to, because one has to use pointers to access memory on the heap. We will talk about pointers in the next lecture.

Stack Vs Heap: Pros and Cons

Stack:

- very fast access
- don't have to explicitly free variables
- space is managed efficiently by CPU,
- memory will not become fragmented
- local variables only
- limit on stack size (OS-dependent)
- variables cannot be resized

Heap

- variables can be accessed globally
- no limit on memory size
- (relatively) slower access
- no guaranteed efficient use of space, memory may become fragmented over time as blocks of memory are allocated, then freed
- you must manage memory (you're in charge of allocating and freeing variables)

When to use the Heap?

- If you need to allocate a large block of memory (e.g. a large array, or a big struct), and you need to keep that variable around a long time (like a global), then you should allocate it on the heap.

When to use the Heap?

- If you need to allocate a large block of memory (e.g. a large array, or a big struct), and you need to keep that variable around a long time (like a global), then you should allocate it on the heap.
- If you are dealing with relatively small variables that only need to persist as long as the function using them is alive, then you should use the stack.

When to use the Heap?

- If you need to allocate a large block of memory (e.g. a large array, or a big struct), and you need to keep that variable around a long time (like a global), then you should allocate it on the heap.
- If you are dealing with relatively small variables that only need to persist as long as the function using them is alive, then you should use the stack.
- If you need variables like arrays and structs that can change size dynamically (e.g. arrays that can grow or shrink as needed) then you will likely need to allocate them on the heap, and use dynamic memory allocation functions to manage that memory "by hand".