1. **Functions in Shell Scripts:**
   - Also called subroutines.
   - Arguments are passed through positional parameters.
   - Variables defined outside the function are visible within as long as they are declared before the function calling.
   - Variables defined inside the function are visible outside.
   - Return value is stored in "$?".
   - Must return a number.
   - E.g. Consider the shell script below:

```
#!/bin/sh
myfunc() {
        arg1=$1
        arg2=$2
        echo $globalvar $arg1
        return 100
}
globalvar="I like to eat"
myfunc pizza spaghetti
echo $?
```

     The output is:
     I like to eat pizza.
     100
   - E.g. Consider the shell script below:

```
#!/bin/sh
myfunc() {
        arg1=$1
        arg2=$2
        echo $globalvar $arg1
        return 100
}
globalvar="I like to eat"
myfunc pizza spaghetti
echo $arg2
echo $?
```

     The output is:
     I like to eat pizza.
     spaghetti
     0

   - E.g. Consider the shell script below:

```
#!/bin/bash
func(){
var1=3
echo $var2
return 100
```

```
}
func ← Function calling
var2=5
echo $var1
```
The output is:

3

- E.g. Consider the shell script below:

```
#!/bin/bash
func(){
var1=3
echo $var2
return 100
}
var2=5
func ← Function calling
echo $var1
```
The output is:
5
3

2. **Introduction to C:**
   - C is a low-level language: machine access
   - C is a high-level language: structured
   - C is a small language, extendable with libraries
   - C is permissive: assumes you know what you're doing
   - Good: efficient, powerful, portable, flexible
   - Bad: easy to make errors
3. **Basic Data Types:**
   - char: Integer; 8 bits **Note:** 8 bits = 1 byte
   - int: Integer; 32 bits
   - long: Integer; 64 bits
   - float: Real number; 32 bits
   - double: Real number; 64 bits
   - In an assignment statement make sure that the variable on the left is at least as wide as the expression on the right.
   - **Note:** In C, there is no string type. We have to use char arrays.
4. **Variables:**
   - Variable names cannot start with a number or an underscore.
   - Variables need to be declared.
   - Variables have no default value. If a variable is not initialized, there's no guarantees about what its content is.
   - One function's variables are not visible to other functions. (I.e. they are local)
   - Variables are only visible inside the block they were declared in, with the exception of global variables declared outside of the main function.

- Examples of variable declaration:
  - int a;
  - int a = 10;
  - char a, b, c;
  - char a = “a”, b = “b”, c = “c”;
- The type of a variable must be a valid C type.

5. **If statement:**
   - You must have an if, but the else if and else are optional. Furthermore, you can have as many else if’s and at most, one else. The if is followed by the else if’s, if there are any, followed by an else, if there is any. Lastly, once a condition is met, the rest of the else if’s and else won’t be tested.
   - If there is any variables created in the if, else if, else block, then once you exit the block, the variable is no longer accessible.
   - We can’t mix declarations and non-declaration in a block.
   - Syntax:

```
if ( condition 1 ){
        some code …
}
else if ( condition 2 ){
        some code …
}
else{
        some code …
}
```

   - Example:

```
int a = 10;
if ( a < 20 ){
        printf(“a is less than 20.”);
}
else if ( a == 20){
        printf(“a is equal to 20.”);
}
else{
        printf(“a is greater than 20.”);
}
```

6. **While loop:**
   - If there is any variables created in the while loop, then once you exit the loop, the variable is no longer accessible.
   - We can’t mix declarations and non-declaration in a block.
   - Syntax:

```
while ( condition ){
        some code ...
}
```

- Example:

```
int a = 10;
while (a < 20){
        printf("%d", a);
        a++;
}
```

7. **For loop:**
   - If there is any variables created in the for loop, then once you exit the loop, the variable is no longer accessible.
   - Syntax:

```
for ( init; condition; increment ) {
   statement(s);
}
```

   Note: In older versions of C, you cannot have the initialization in the for loop. I.e. You can't do for (int i = 1; …)
     - Here is the flow of control in a 'for' loop:
       1. The init step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
       2. Next, the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the 'for' loop.
       3. After the body of the 'for' loop executes, the flow of control jumps back up to the increment statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
       4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the 'for' loop terminates.
   - Example:

```
for (int i = 1; i < 20; i++){
        printf("%d \n", i);
}
```

8. **Functions:**
   - Every C program has the main function, main().
   - A **function declaration** tells the compiler about a function's name, return type, and parameters.
     The syntax is **return_type function_name(parameter list);**
   - A **function definition** provides the actual body of the function.
   - Functions must be declared before first use then defined.

- **return_type function_name (parameters) {**
  some code …
  }

  Return Type: A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value.

  Function Name: This is the actual name of the function. The function name and the parameter list together constitute the function signature.

  Parameters: A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

  Function Body: The function body contains a collection of statements that define what the function does.

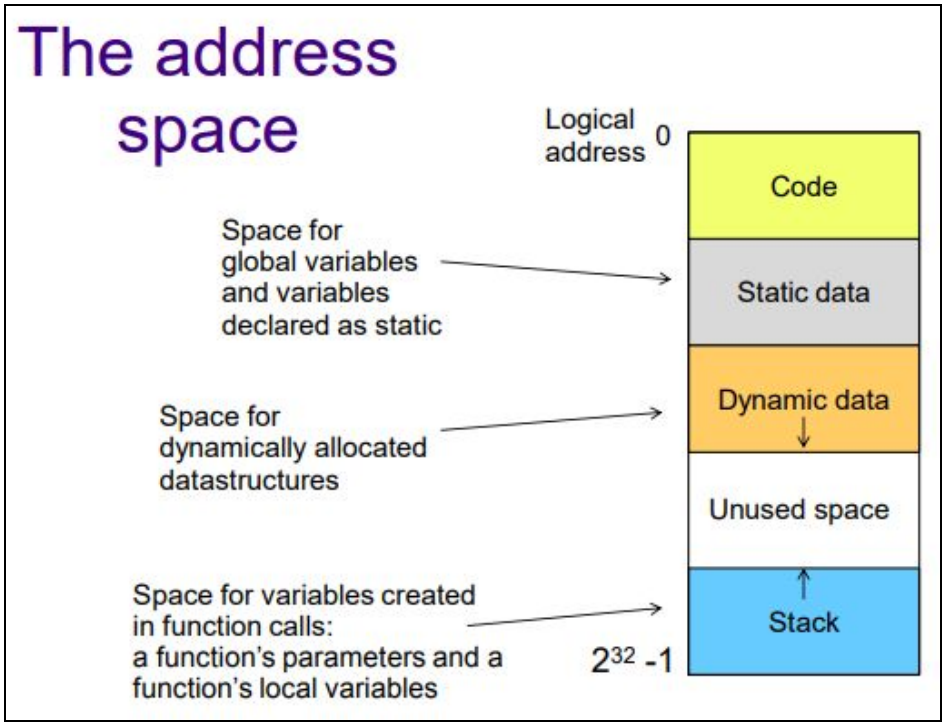
- If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function. Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.
- While calling a function, there are two ways in which arguments can be passed to a function:
  1. **Call by value:** This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
  2. **Call by reference:** This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

     By default, C uses call by value to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

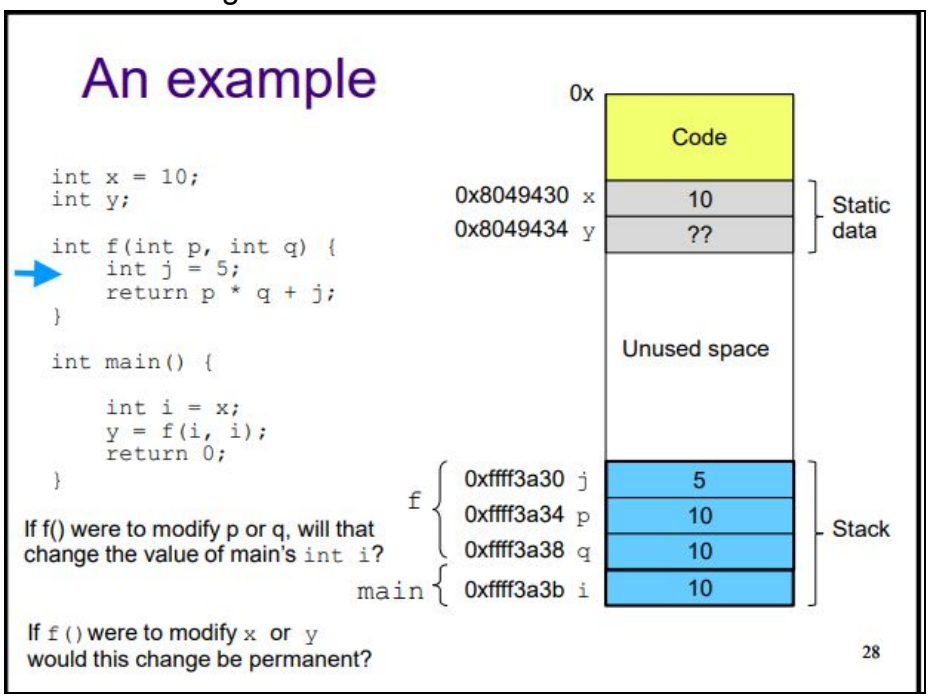
- Example:

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

int min(int num1, int num2); /* Method declaration */

int main (){
   int a = 0;
   int b = 100;
   int result;
   result = min(a, b);
   printf("The minimum value is %d", result);
   return 0;
}

/* Method Definition */
int min (int num1, int num2){

   int result;

   if (num1 < num2){
      result = num1;
   }

   else {
      result = num2;
   }

   return result;
}
```

9. **Memory Model:**
    - The memory for a **process** (a running program) is called it's **address space**.
    - Memory is just a sequence of bytes.
    - A memory location (a byte) is identified by an address.



    - E.g.



The answer to both questions is no.

10. Arrays:

- Arrays in C are a contiguous chunk of memory that contain a list of items of the same type.
- In particular, the size of the array is not stored with the array. There is no runtime checking.
- To declare a single-dimensional array, you do:
  type arrayName [arraySize]
  type can be any valid C type.
  arraySize must be an integer constant greater than zero.
  E.g.
  int a[10]; This creates an array of size 10 that holds ints.
- You can initialize an array by any of the following methods:
  1. **Initialization Loops:**
     ```
     for (int i = 0; i < N; i++){
             a[i] = i;
     }
     ```
  2. **Static Initialization:**
     int a[5] = {1,2,3,4,5};
     Note: The number of values in the curly brackets cannot exceed the size of the array.
     int a[] = {1,2,3,4,5}; also works.
- **Note:** The first element of an array is its 0th index.
  I.e. a[0] will get the first element of a.
- You can set or change values for a specific element, too.
  E.g.
  int a[4] = 50; will set 50 as the 5th element of array a.
- An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array.
  E.g.
  int x = a[9]; This will set x to a's 10th element.
- There are multi-dimensional arrays, too.
- The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size [x][y], you do type arrayName [x][y]. Here, x is the number of rows and y is the number of columns.
  E.g.
  int a[3][4]; This creates a 2-D array with 3 rows and 4 columns.

- You can initialize an array by any of the following methods:
  1. **Initialization Loops:**

```
for (int i = 0; i < N; i++){
        for (int j = 0; j < M; j++){
                a[i][j] = i*j;
        }
}
```

  2. **Static Initialization:**

```
int a[3][4] = {
                {0, 1, 2, 3} ,
                {4, 5, 6, 7} ,
                {8, 9, 10, 11}
               };
```

     int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11}; also works.
- To access an element of a 2-D array, you do
  int x = a[i][j];

11. **Strings:**
    - Strings are actually one-dimensional array of characters terminated by a null character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a null.
    - Since there's the null character at the end, the array's size is one bigger than the length of the word. If a word has 3 letters, the size of the array is 4.
    - E.g.
      char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
      char greeting[] = "Hello";
      Both ways are valid.
12. **Pointers:**
    - A pointer is a higher-level version of an address.
      I.e. A pointer is a variable whose value is the address of another variable.
    - The general form of a pointer declaration is: **type *var;**
      E.g.
      char *cptr;
    - Assigning a value to a pointer:
      char c = 'a';
      cptr = &c;
      - cptr gets the value of the address of c.
      - The value stored in the variable cptr is the address of the memory location where variable c is stored.
    - Dereferencing a pointer:
      cptr = 'b';
      - Stores 'b' at the memory address that is stored in cptr.

      char d = *cptr;
      - Takes the contents of the memory address stored in cptr and stores them in the variable d.

Week 3 Notes

- Without an asterisk, pointer references an address.
- With an asterisk, pointer references the value at that address.
- Always use the same type of pointer as the variable it examines. (I.e. ints for ints, chars for chars)
- Initialize the pointer before using it. Set it to an address of some variable.
- When you declare an array, you are creating a pointer to the array's 0th element. (In C, pointers and arrays are very similar.)
  E.g.
  int a[3] = {1, 3, 5};
  int *p = a; // same as p = &a[0];
  Note: When pointing to an array, you do not need to use &.
- The array access operator [ ] is really only a shorthand for pointer arithmetic + dereference.
- These are equivalent in C: a[i] == *(a + i)
  E.g.
  a[0] == *p
  a[1] == *(p+1)
  a[2] == *(p+2)
- Pointer arithmetic respects the type of the pointer.
  E.g.
  int i[2] = {1, 2};
  int *ip;
  ip = i;
  ***(ip + 1) += 2; // This puts 4 in i[1].**
  E.g.
  char c[2] = {'a','z'};
  char *cp;
  cp = c;
  ***(cp + 1) = 'b'; // This puts b to c[1].**
- C knows the size of what is being pointed at from the type of the pointer.
- When you pass arrays as parameters, what is being passed to the function is the name of the array which decays to a pointer to the first element. Since C doesn't store the size of arrays, you should also input the size of the array.
- int sum(int *a, int size) and int sum(int a[], int size) are both legal, but it's bad form to use the latter. This is because you really are passing a pointer-to-int not an array, you still don't know how big the array is.
- **Note:** Use array notation rather than pointer arithmetic whenever you have an array.

13. **Scope Rules:**
    - There are three places where variables can be declared in C programming language:
        - Inside a function or a block which is called **local variables**.
        - Outside of all functions which is called **global variables**.
        - In the definition of function parameters which are called **formal parameters**.
    - **Local Variables**: Variables that are declared inside a function or block. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own.
    - **Global Variables**: Variables that are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program. A global variable can be accessed by any function.
      I.e. A global variable is available for use throughout your entire program after its declaration.
    - **Formal Parameters**: Formal parameters, are treated as local variables with-in a function and they take precedence over global variables.
14. **Special Characters and Miscellaneous Information:**
    - **#include <stdio.h>**: Include this at the top of your program. It allows you to access some of C's built in functions and libraries.
      Note: There are other, similar tags.
    - **printf**: If you have #include <stdio.h>, then this is one of the built in features you can use. It allows you to print things on the console.
    - **%d:** Take the next argument and print it as an int.
    - **%s:** Take the next argument and print it as a string.
    - **\n:** Newline
    - **/* */:** Comments
    - **&**: Allows the user to access an item's memory location.
    - Because C is permissive, the error messages are often random and/or weird.
    - In C, 0 is false and 1 is true.
    - To compile a C program, do the following:
      gcc -Wall -o programName programName.c
      gcc is the compiler.
      -Wall may help create some meaningful error messages.
      -o programName creates the object file.