C Language for advanced level:

In a previous level we learn about the if statement.

Now we learn about more advanced level

Switch Statement:

Instead of writing **many** `if..else` statements, you can use the `switch` statement.

The `switch` statement selects one of many code blocks to be executed:

Syntax

```
switch (expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

This is how it works:

- The `switch` expression is evaluated once
- The value of the expression is compared with the values of each `case`
- If there is a match, the associated block of code is executed
- The `break` statement breaks out of the switch block and stops the execution
- The `default` statement is optional, and specifies some code to run if there is no case match

The example below uses the weekday number to calculate the weekday name:

Example

```
int day = 4;

switch (day) {
  case 1:
    printf("Monday");
```

```
    break;
  case 2:
    printf("Tuesday");
    break;
  case 3:
    printf("Wednesday");
    break;
  case 4:
    printf("Thursday");
    break;
  case 5:
    printf("Friday");
    break;
  case 6:
    printf("Saturday");
    break;
  case 7:
    printf("Sunday");
    break;
}

// Outputs "Thursday" (day 4)
```

The break Keyword:

When C reaches a `break` keyword, it breaks out of the switch block.

This will stop the execution of more code and case testing inside the block.

When a match is found, and the job is done, it's time for a break. There is no need for more testing.

The default Keyword

The `default` keyword specifies some code to run if there is no case match:

Example

```
int day = 4;

switch (day) {
  case 6:
    printf("Today is Saturday");
    break;
  case 7:
```

```
    printf("Today is Sunday");
    break;
  default:
    printf("Looking forward to the Weekend");
}

// Outputs "Looking forward to the Weekend"
```

Loops:

Loops can execute a block of code as long as a specified condition is reached.

Loops are handy because they save time, reduce errors, and they make code more readable.

While Loop:

The `while` loop loops through a block of code as long as a specified condition is `true`:

Syntax

```
while (condition) {
  // code block to be executed
}
```

In the example below, the code in the loop will run, over and over again, as long as a variable (`i`) is less than 5:

Example

```
int i = 0;

while (i < 5) {
  printf("%d\n", i);
  i++;
}
```

The Do/While Loop:

The `do/while` loop is a variant of the `while` loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax

```
do {
  // code block to be executed
}
while (condition);
```

The example below uses a `do/while` loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

Example

```
int i = 0;

do {
  printf("%d\n", i);
  i++;
}
while (i < 5);
```

For Loop:

When you know exactly how many times you want to loop through a block of code, use the `for` loop instead of a `while` loop:

Syntax

```
for (expression 1; expression 2; expression 3) {
  // code block to be executed
}
```

Expression 1 is executed (one time) before the execution of the code block.

Expression 2 defines the condition for executing the code block.

Expression 3 is executed (every time) after the code block has been executed.

The example below will print the numbers 0 to 4:

Example:

```
int i;

for (i = 0; i < 5; i++) {
  printf("%d\n", i);
}
```

Example explained:

Expression 1 sets a variable before the loop starts (int i = 0).

Expression 2 defines the condition for the loop to run (i must be less than 5). If the condition is true, the loop will start over again, if it is false, the loop will end.

Expression 3 increases a value (i++) each time the code block in the loop has been executed.

Nested Loops:

It is also possible to place a loop inside another loop. This is called a **nested loop**.

The "inner loop" will be executed one time for each iteration of the "outer loop":

Example:

```
int i, j;

// Outer loop
for (i = 1; i <= 2; ++i) {
  printf("Outer: %d\n", i);  // Executes 2 times

  // Inner loop
  for (j = 1; j <= 3; ++j) {
    printf(" Inner: %d\n", j);  // Executes 6 times (2 * 3)
  }
}
```

C Break and Continue:

Break:

You have already seen the `break` statement used in an earlier chapter of this tutorial. It was used to "jump out" of a `switch` statement.

The `break` statement can also be used to jump out of a **loop**.

This example jumps out of the **for loop** when `i` is equal to 4:

Example:

```
int i;

for (i = 0; i < 10; i++) {
  if (i == 4) {
    break;
  }
  printf("%d\n", i);
}
```

Continue:

The `continue` statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

This example skips the value of 4:

Example:

```
int i;

for (i = 0; i < 10; i++) {
  if (i == 4) {
    continue;
  }
  printf("%d\n", i);
}
```

Break and Continue in While Loop:

You can also use `break` and `continue` in while loops:

Break example:

```
int i = 0;

while (i < 10) {
  if (i == 4) {
    break;
  }
  printf("%d\n", i);
  i++;
}
```

Continue Example:

```
int i = 0;

while (i < 10) {
  if (i == 4) {
    i++;
    continue;
  }
  printf("%d\n", i);
  i++;
}
```

Memory Address:

When a variable is created in C, a memory address is assigned to the variable.

The memory address is the location of where the variable is stored on the computer.

When we assign a value to the variable, it is stored in this memory address.

To access it, use the reference operator (&), and the result represents where the variable is stored:

Example:

```c
int myAge = 43;
printf("%p", &myAge); // Outputs 0x7ffe5367e044
```

Creating Pointers:

You learned from the previous chapter, that we can get the **memory address** of a variable with the reference operator `&`:

Example

```c
int myAge = 43; // an int variable

printf("%d", myAge);  // Outputs the value of myAge (43)
printf("%p", &myAge); // Outputs the memory address of myAge
(0x7ffe5367e044)
```

A **pointer** is a variable that **stores** the **memory address** of another variable as its value.

A **pointer variable points** to a **data type** (like `int`) of the same type, and is created with the `*` operator.

The address of the variable you are working with is assigned to the pointer:

Example

```c
int myAge = 43;     // An int variable
int* ptr = &myAge;  // A pointer variable, with the name ptr, that stores
the address of myAge

// Output the value of myAge (43)
printf("%d\n", myAge);

// Output the memory address of myAge (0x7ffe5367e044)
printf("%p\n", &myAge);

// Output the memory address of myAge with the pointer (0x7ffe5367e044)
printf("%p\n", ptr);
```

Example explained

Create a pointer variable with the name `ptr`, that **points to** an `int` variable (`myAge`). Note that the type of the pointer has to match the type of the variable you're working with (`int` in our example).

Use the `&` operator to store the memory address of the `myAge` variable, and assign it to the pointer.

Now, `ptr` holds the value of `myAge`'s memory address.

Dereference:

In the example above, we used the pointer variable to get the memory address of a variable (used together with the `&` **reference** operator).

You can also get the value of the variable the pointer points to, by using the `*` operator (the **dereference** operator):

Example

```
int myAge = 43;     // Variable declaration
int* ptr = &myAge;  // Pointer declaration

// Reference: Output the memory address of myAge with the pointer
(0x7ffe5367e044)
printf("%p\n", ptr);

// Dereference: Output the value of myAge with the pointer (43)
printf("%d\n", *ptr);
```

Pointers & Arrays:

You can also use pointers to access arrays.

Consider the following array of integers:

Example

```
int myNumbers[4] = {25, 50, 75, 100};
```

Example

```
int myNumbers[4] = {25, 50, 75, 100};
int i;

for (i = 0; i < 4; i++) {
  printf("%d\n", myNumbers[i]);
}
```

Result:

```
25
50
75
100
```

Instead of printing the value of each array element, let's print the memory address of each array element:

Example

```
int myNumbers[4] = {25, 50, 75, 100};
int i;

for (i = 0; i < 4; i++) {
  printf("%p\n", &myNumbers[i]);
}
```

Result:

```
0x7ffe70f9d8f0
0x7ffe70f9d8f4
0x7ffe70f9d8f8
0x7ffe70f9d8fc
```

Note that the last number of each of the elements' memory address is different, with an addition of 4.

It is because the size of an `int` type is typically 4 bytes, remember:

Example

```c
// Create an int variable
int myInt;

// Get the memory size of an int
printf("%lu", sizeof(myInt));
```

Result:

```
4
```

Example

```c
int myNumbers[4] = {25, 50, 75, 100};

// Get the size of the myNumbers array
printf("%lu", sizeof(myNumbers));
```

Result:

```
16
```

The **memory address** of the **first element** is the same as the **name of the array**:

Example

```c
int myNumbers[4] = {25, 50, 75, 100};

// Get the memory address of the myNumbers array
printf("%p\n", myNumbers);

// Get the memory address of the first array element
printf("%p\n", &myNumbers[0]);
```

Result:

```
0x7ffe70f9d8f0
0x7ffe70f9d8f0
```

This basically means that we can work with arrays through pointers!

Example

```
int myNumbers[4] = {25, 50, 75, 100};

// Get the value of the first element in myNumbers
printf("%d", *myNumbers);
```

Result:

```
25
```

To access the rest of the elements in myNumbers, you can increment the pointer/array (+1, +2, etc):

Example

```
int myNumbers[4] = {25, 50, 75, 100};

// Get the value of the second element in myNumbers
printf("%d\n", *(myNumbers + 1));

// Get the value of the third element in myNumbers
printf("%d", *(myNumbers + 2));

// and so on..
```

Result:

```
50
75
```

Or loop through it:

Example

```
int myNumbers[4] = {25, 50, 75, 100};
int *ptr = myNumbers;
int i;

for (i = 0; i < 4; i++) {
  printf("%d\n", *(ptr + i));
}
```

Result:

```
25
50
75
100
```

It is also possible to change the value of array elements with pointers:

Example

```c
int myNumbers[4] = {25, 50, 75, 100};

// Change the value of the first element to 13
*myNumbers = 13;

// Change the value of the second element to 17
*(myNumbers +1) = 17;

// Get the value of the first element
printf("%d\n", *myNumbers);

// Get the value of the second element
printf("%d\n", *(myNumbers + 1));
```

Result:

```
13
17
```

Structures:

Structures (also called structs) are a way to group several related variables into one place. Each variable in the structure is known as a **member** of the structure.

Unlike an array, a structure can contain many different data types (int, float, char, etc.).

Create a Structure:

You can create a structure by using the `struct` keyword and declare each of its members inside curly braces:

```
struct MyStructure {   // Structure declaration
  int myNum;           // Member (int variable)
  char myLetter;       // Member (char variable)
}; // End the structure with a semicolon
```

To access the structure, you must create a variable of it.

Use the `struct` keyword inside the `main()` method, followed by the name of the structure and then the name of the structure variable:

Create a struct variable with the name "s1":

```
struct myStructure {
  int myNum;
  char myLetter;
};

int main() {
  struct myStructure s1;
  return 0;
}
```

Access Structure Members:

To access members of a structure, use the dot syntax (`.`):

Example

```
// Create a structure called myStructure
struct myStructure {
  int myNum;
  char myLetter;
};

int main() {
  // Create a structure variable of myStructure called s1
  struct myStructure s1;

  // Assign values to members of s1
  s1.myNum = 13;
  s1.myLetter = 'B';

  // Print values
  printf("My number: %d\n", s1.myNum);
```

```
  printf("My letter: %c\n", s1.myLetter);

  return 0;
}
```

Now you can easily create multiple structure variables with different values, using just one structure:

Example

```
// Create different struct variables
struct myStructure s1;
struct myStructure s2;

// Assign values to different struct variables
s1.myNum = 13;
s1.myLetter = 'B';

s2.myNum = 20;
s2.myLetter = 'C';
```

What About Strings in Structures?

Remember that strings in C are actually an array of characters, and unfortunately, you can't assign a value to an array like this:

Example

```
struct myStructure {
  int myNum;
  char myLetter;
  char myString[30];  // String
};

int main() {
  struct myStructure s1;

  // Trying to assign a value to the string
  s1.myString = "Some text";

  // Trying to print the value
  printf("My string: %s", s1.myString);
```

```
  return 0;
}
```

Example

```
struct myStructure {
  int myNum;
  char myLetter;
  char myString[30]; // String
};

int main() {
  struct myStructure s1;

  // Assign a value to the string using the strcpy function
  strcpy(s1.myString, "Some text");

  // Print the value
  printf("My string: %s", s1.myString);

  return 0;
}
```

Result:

```
My string: Some text
```

Simpler Syntax:

You can also assign values to members of a structure variable at declaration time, in a single line.

Just insert the values in a comma-separated list inside curly braces `{}`. Note that you don't have to use the `strcpy()` function for string values with this technique:

Example

```
// Create a structure
struct myStructure {
  int myNum;
  char myLetter;
  char myString[30];
};

int main() {
  // Create a structure variable and assign values to it
  struct myStructure s1 = {13, 'B', "Some text"};
```

```
  // Print values
  printf("%d %c %s", s1.myNum, s1.myLetter, s1.myString);

  return 0;
}
```

Copy Structures:

You can also assign one structure to another.

In the following example, the values of s1 are copied to s2:

Example

```
struct myStructure s1 = {13, 'B', "Some text"};
struct myStructure s2;

s2 = s1;
```

Modify Values:

If you want to change/modify a value, you can use the dot syntax (`.`).

And to modify a string value, the `strcpy()` function is useful again:

Example

```
struct myStructure {
  int myNum;
  char myLetter;
  char myString[30];
};

int main() {
  // Create a structure variable and assign values to it
  struct myStructure s1 = {13, 'B', "Some text"};

  // Modify values
  s1.myNum = 30;
  s1.myLetter = 'C';
  strcpy(s1.myString, "Something else");

  // Print values
  printf("%d %c %s", s1.myNum, s1.myLetter, s1.myString);
```

```
  return 0;
}
```

Modifying values are especially useful when you copy structure values:

Example

```
// Create a structure variable and assign values to it
struct myStructure s1 = {13, 'B', "Some text"};

// Create another structure variable
struct myStructure s2;

// Copy s1 values to s2
s2 = s1;

// Change s2 values
s2.myNum = 30;
s2.myLetter = 'C';
strcpy(s2.myString, "Something else");

// Print values
printf("%d %c %s\n", s1.myNum, s1.myLetter, s1.myString);
printf("%d %c %s\n", s2.myNum, s2.myLetter, s2.myString);
```

C Enums:

An **enum** is a special type that represents a group of constants (unchangeable values).

To create an enum, use the `enum` keyword, followed by the name of the enum, and separate the enum items with a comma:

```
enum Level {
  LOW,
  MEDIUM,
  HIGH
};
```

To access the enum, you must create a variable of it.

Inside the main() method, specify the enum keyword, followed by the name of the enum (Level) and then the name of the enum variable (myVar in this example):

```
enum Level myVar;
```

Now that you have created an enum variable (myVar), you can assign a value to it.

The assigned value must be one of the items inside the enum (LOW, MEDIUM or HIGH):

```
enum Level myVar = MEDIUM;
```

By default, the first item (LOW) has the value 0, the second (MEDIUM) has the value 1, etc.

If you now try to print myVar, it will output 1, which represents MEDIUM:

```
int main() {
  // Create an enum variable and assign a value to it
  enum Level myVar = MEDIUM;

  // Print the enum variable
  printf("%d", myVar);

  return 0;
}
```

Change Values:

the first item of an enum has the value 0. The second has the value 1, and so on.

To make more sense of the values, you can easily change them:

```
enum Level {
  LOW = 25,
  MEDIUM = 50,
  HIGH = 75
};
printf("%d", myVar); // Now outputs 50
```

Note that if you assign a value to one specific item, the next items will update their numbers accordingly:

```
enum Level {
  LOW = 5,
  MEDIUM, // Now 6
  HIGH // Now 7
};
```

Enum in a Switch Statement

Enums are often used in switch statements to check for corresponding values:

```
enum Level {
  LOW = 1,
  MEDIUM,
  HIGH
};

int main() {
  enum Level myVar = MEDIUM;

  switch (myVar) {
    case 1:
      printf("Low Level");
      break;
    case 2:
      printf("Medium level");
      break;
    case 3:
      printf("High level");
      break;
  }
  return 0;
}
```