Lecture 3 - Scopes, Lifetime, Inline Functions, and Modularity

1. Scopes

1.1. Function Prototype Scope:

A variable defined as a parameter for the prototype isn't automatically defined outside the prototype.

The code

```
void myFunction(int x);

int main(void)
{
    printf("%d", x);
}
```

will result in

```
error: 'x' undeclared (first use in this function)
     printf("%d", x);
                  ^
```

1.2. Function Scope

A variable defined as a parameter for a function definition can't be accessed automatically outside the function even in the function decleration.

The code

```
void myFunction(int x)
{
    x = 4;
    printf("%d", x);
}

int main(void)
{
    printf("%d", x);
    return 0;
}
```

will result in

```
error: 'x' undeclared (first use in this function)
     printf("%d", x);
                  ^
```

Also the code

```
int addNumbers(int a, int b);

int main(void)
{
    return 0;
}

int addNumbers(int x, int y)
{
    return a + b;
}
```

will cause the same compilation error.

1.3. Block Scope

A scope can be created using curly braces { and }. A variable defined inside the curly braces are only accessible inside this scope and any scopes encapsulated inside the variable scope.

The code

```
int main(void)
{
    {
        int x = 5;
    }
    printf("%d", x);
}
```

will result in

```
warning: unused variable 'x' [-Wunused-variable]
         int x = 5;
             ^

error: 'x' undeclared (first use in this function)
     printf("%d", x);
                  ^
```

1.4. File Scope

A global variable is available to access through its file only.

For `src1.c`

```
// src1.c
int x = 3;

int main(void)
{
    printf("Hello\n");
}
```

If we tried to build `src2.c`

```
// src2.c
void myFunction(void)
{
    printf("%d", x);
}
```

A compilation error will occur.

```
error: 'x' undeclared (first use in this function)
     printf("%d", x);
                  ^
```

To solve this issue, we use `extern` key word in `src2.c`

```
//src2.c
extern int x;
void myFunction(void)
{
    printf("%d", x);
}
```

If we want to keep the variable `x` private only to its file `src1.c`, we can use the keyword `static`.

```
// src1.c
static int x = 5;
```

However, with keeping the `extern` keyword in `src2.c` will result an error when compiling the whole project.

note: If a variable is used just for one function, define the variable inside this function scope.

DON'T

```
int x = 3;

void myFucntion(void)
{
    printf("%d", x);
}

int main(void)
{
    myFunction();
    return 0;
}
```

DO

```
void myFucntion(void)
{

    int x = 3;
    printf("%d", x);
}

int main(void)
{
    myFunction();
    return 0;
}
```

That also includes using `static` keyword when declaring a variable that will be used in its file scope only.

Variables of different scopes can be named the same. When accessing the variable, we go from current scope then go outwards.

For example,

```
int x = 5;

int main(void)
{
    printf("%d ", x);
    {
        int x = 10;
        printf("%d ", x);
        x++;
        printf("%d ", x);
    }
    printf("%d ", x);
    int x = 8;
    printf("%d ", x);
    return 0;
}
```

will result in

```
5 10 11 5 8
```

where,

```
 5 --> global scope
10 --> inner scope inside main
11 --> inner scope inside main incremented
5 --> global scope
8 --> main scope
```

However, naming different-scope variables with the same name **isn't preferable**.

---

2. Lifetime

A variable that is defined in global or file scope if it,

- was initalized with a value `int x = 3;`, the variable `x` is stored in the data segment.
- was not initialized with a value `int x;`, the variable `x` will be stored in the BSS segment.

Both data and BSS segments addresses are constant during run-time and the values inside them are preserved. **lifetime: the entire runtime of the program**.

A variable that is defined inside a function is called *automatic variable* and is allocated int the stack segment and only when the function is called. Once the function is over, the variable is discarded. When the function is called again, the variable is again created in the stack and may have different address from the previous time. That's why the modifications of this variable aren't preserved and can't be accessed outside the function scope.

lifetime: begins when program execution enters the function or statement block or compound and ends when execution leaves the block.

When using the keyword `static` with a variale that's not int the global(file) scope, it informs the compiler to create the variable inside the data segment, if initialized, instead of the stack segment. So, any modification that happened to the variable are preserved when the function is called again. Still, it can't be accessed outside its scope.

lifetime: lifetime of the program

For example,

```
void myFunction(void)
{
    static int x = 10;
    printf("%d ", x);
    x++;
}

int main(void)
{
    myFunction();
    myFunction();
    myFunction();
    return 0;
}
```

wil result in

```
10 11 12
```

The variable `x` is printed as 10 then incremented to 11. When the function is called again, the `static` keywork keeps the value 11 inside the constant address of `x` and use the value of 11, print it, then increment it to 12. When the function is called for the third time, 12 will be printed and then incremented to 13.

---

3. Inline Functions

When calling a function, a `JUMP` instruction is executed in addition to creating the variables in the stack segment.

Using `inline` keyword before the function definition tells the compiler to replace the function call with the machine code of the function.

NOTE When coding, it's preferable to write `static inline` instead of `inline`. Why?

Although using `inline` keyword means that it is the same if we wrote the lines of function manually instead of calling a function, it is * very useful in time-sensitive programs.

- very useful in reducing the complexity of the code and improve the readability.

But of course it costs more memory.

---

4. Modular Programming

Instead of writing all the code in one single file which is very difficult when working in a team. We separte the functions, which we created to solve some problems in our program, into external files which are linked to the main file through the linker.

The definitions of the function are written in a source files (.c) and their prototypes in header files (.h) and the header files are included in the main file.

```
// Math.h
int Add(int x, int y);
```

```
// Math.c
int Add(int x, int y)
{
    return x + y;
}
```

```
// main.c

#include "Math.h"

int main()
{
    int z = Add(10, 15);
    return 0;
}
```