Introduction to Programming

Week – *7*, Lecture – *2*

Global and Static Scopes in C

SAURABH SRIVASTAVA

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

IIT KANPUR

A review of “scopes”

We’ve discussed, till now, how variables defined in a particular block, are not accessible outside

- Now that we have discussed functions, it will be clearer

A review of “scopes”

We’ve discussed, till now, how variables defined in a particular block, are not accessible outside

- Now that we have discussed functions, it will be clearer

The variables that we “usually” create fall in `auto` storage class

- The variables in this storage class have their scopes limited to the block in which they are defined

A review of “scopes”

We’ve discussed, till now, how variables defined in a particular block, are not accessible outside

- Now that we have discussed functions, it will be clearer

The variables that we “usually” create fall in `auto` storage class

- The variables in this storage class have their scopes limited to the block in which they are defined

This is why, if we want to *pass* the value of these variables to another function…

- … we need to supply them as an argument

A review of “scopes”

We’ve discussed, till now, how variables defined in a particular block, are not accessible outside

- Now that we have discussed functions, it will be clearer

The variables that we “usually” create fall in `auto` storage class

- The variables in this storage class have their scopes limited to the block in which they are defined

This is why, if we want to *pass* the value of these variables to another function…

- … we need to supply them as an argument

The same is true for a value, which is returned from the called function

- It has to be stored in some local variable, to be used further in the calling function

A review of “scopes”

We’ve discussed, till now, how variables defined in a particular block, are not accessible outside

- Now that we have discussed functions, it will be clearer

The variables that we “usually” create fall in `auto` storage class

- The variables in this storage class have their scopes limited to the block in which they are defined

This is why, if we want to *pass* the value of these variables to another function…

- … we need to supply them as an argument

The same is true for a value, which is returned from the called function

- It has to be stored in some local variable, to be used further in the calling function

In some cases, a particular variable’s value may be required to be passed to many functions

- In this case, it may be convenient, to put the variable in a different storage class

Example: Creating some String Utilities

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

char* trim(char* str);
short startsWith(char* str, char* prefix);

int main()
{
        char* str = "  Hello  ";
        printf("%d\n", startsWith(str, "Hello"));
        str = trim(str);
        printf("%d\n", startsWith(str, "Hello"));
        free(str);
}
```

Example: Creating some String Utilities

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

char* trim(char* str);
short startsWith(char* str, char* prefix);

int main()
{
        char* str = "  Hello  ";
        printf("%d\n", startsWith(str, "Hello"));
        str = trim(str);
        printf("%d\n", startsWith(str, "Hello"));
        free(str);
}
```

Assume that we want to create two string-related functions

Example: Creating some String Utilities

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

char* trim(char* str);
short startsWith(char* str, char* prefix);

int main()
{
        char* str = "  Hello  ";
        printf("%d\n", startsWith(str, "Hello"));
        str = trim(str);
        printf("%d\n", startsWith(str, "Hello"));
        free(str);
}
```

The `trim()` function trims any leading and trailing spaces in the string

Example: Creating some String Utilities

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

char* trim(char* str);
short startsWith(char* str, char* prefix);

int main()
{
        char* str = "  Hello  ";
        printf("%d\n", startsWith(str, "Hello"));
        str = trim(str);
        printf("%d\n", startsWith(str, "Hello"));
        free(str);
}
```

The `startsWith()` function checks if one string is at the beginning of another given string or not

Example: Creating some String Utilities

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

char* trim(char* str);
short startsWith(char* str, char* prefix);

int main()
{
        char* str = "  Hello  ";
        printf("%d\n", startsWith(str, "Hello"));
        str = trim(str);
        printf("%d\n", startsWith(str, "Hello"));
        free(str);
}
```

We start with this string

Example: Creating some String Utilities

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

char* trim(char* str);
short startsWith(char* str, char* prefix);

int main()
{
        char* str = "  Hello  ";
        printf("%d\n", startsWith(str, "Hello"));
        str = trim(str);
        printf("%d\n", startsWith(str, "Hello"));
        free(str);
}
```

This should produce ”false”

Example: Creating some String Utilities

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

char* trim(char* str);
short startsWith(char* str, char* prefix);

int main()
{
        char* str = "  Hello  ";
        printf("%d\n", startsWith(str, "Hello"));
        str = trim(str);
        printf("%d\n", startsWith(str, "Hello"));
        free(str);
}
```

This should produce ”false”

By convention, `startsWith()` will return a `0` in this case

Example: Creating some String Utilities

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

char* trim(char* str);
short startsWith(char* str, char* prefix);

int main()
{
        char* str = "  Hello  ";
        printf("%d\n", startsWith(str, "Hello"));
        str = trim(str);
        printf("%d\n", startsWith(str, "Hello"));
        free(str);
}
```

This should strip the leading and trailing spaces from the string

Example: Creating some String Utilities

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

char* trim(char* str);
short startsWith(char* str, char* prefix);

int main()
{
        char* str = "  Hello  ";
        printf("%d\n", startsWith(str, "Hello"));
        str = trim(str);
        printf("%d\n", startsWith(str, "Hello"));
        free(str);
}
```

Now, this should produce “true”

Example: Creating some String Utilities

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

char* trim(char* str);
short startsWith(char* str, char* prefix);

int main()
{
        char* str = "  Hello  ";
        printf("%d\n", startsWith(str, "Hello"));
        str = trim(str);
        printf("%d\n", startsWith(str, "Hello"));
        free(str);
}
```

Now, this should produce “true”

By convention, `startsWith()` will return a `1` in this case

Example: Creating some String Utilities

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 7$ gcc StringUtils.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 7$ ./a.out
0
1
saurabh@saurabh-VirtualBox:~/C/examples/Week 7$
```

Example: Creating some String Utilities

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 7$ gcc StringUtils.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 7$ ./a.out
0
1
saurabh@saurabh-VirtualBox:~/C/examples/Week 7$
```

… and this is, indeed, the output !!

Example: Creating some String Utilities

```
char* trim(char *str)
{
        int len = strlen(str), beg = 0, end = len-1, ctr = 0;
        char* result = NULL;
        while(beg <= end && str[beg] == ' ')
                beg++;
        while(end >= beg && str[end] == ' ')
                end--;
        result = (char*) malloc(sizeof(char) * (end - beg + 2));
        while(beg <= end)
                result[ctr++] = str[beg++];
        result[ctr] = '\0';
        return result;
}

short startsWith(char* str, char* prefix)
{
        int len1 = strlen(str);
        int len2 = strlen(prefix);
        int i = 0, j = 0;
        if(len2 > len1)
                return 0;
        while(i < len2 && str[j] == prefix[i])
                i++, j++;
        return i == len2;
}
```

Example: Creating some String Utilities

```
char* trim(char *str)
{
        int len = strlen(str), beg = 0, end = len-1, ctr = 0;
        char* result = NULL;
        while(beg <= end && str[beg] == ' ')
                beg++;
        while(end >= beg && str[end] == ' ')
                end--;
        result = (char*) malloc(sizeof(char) * (end - beg + 2));
        while(beg <= end)
                result[ctr++] = str[beg++];
        result[ctr] = '\0';
        return result;
}

short startsWith(char* str, char* prefix)
{
        int len1 = strlen(str);
        int len2 = strlen(prefix);
        int i = 0, j = 0;
        if(len2 > len1)
                return 0;
        while(i < len2 && str[j] == prefix[i])
                i++, j++;
        return i == len2;
}
```

… and these are the definitions for the two methods !!

Global Variables

In the example, we had to pass the string to both functions – `trim()` and `startsWith()`

- This seems a rather wasted effort !!

Global Variables

In the example, we had to pass the string to both functions – `trim()` and `startsWith()`

- This seems a rather wasted effort !!

A global variable in C (and many other languages) is a variable whose scope is *global*

Global Variables

In the example, we had to pass the string to both functions – `trim()` and `startsWith()`

- This seems a rather wasted effort !!

A global variable in C (and many other languages) is a variable whose scope is *global*

By global, we mean that the variable is visible to all the functions in the program

Global Variables

In the example, we had to pass the string to both functions – `trim()` and `startsWith()`

- This seems a rather wasted effort !!

A global variable in C (and many other languages) is a variable whose scope is *global*

By global, we mean that the variable is visible to all the functions in the program

To do so, we must define the variable outside all the functions

Global Variables

In the example, we had to pass the string to both functions – `trim()` and `startsWith()`

- This seems a rather wasted effort !!

A global variable in C (and many other languages) is a variable whose scope is *global*

By global, we mean that the variable is visible to all the functions in the program

To do so, we must define the variable outside all the functions

This will make the variable visible to all the functions, defined after its definition

Global Variables

In the example, we had to pass the string to both functions – `trim()` and `startsWith()`

- This seems a rather wasted effort !!

A global variable in C (and many other languages) is a variable whose scope is *global*

By global, we mean that the variable is visible to all the functions in the program

To do so, we must define the variable outside all the functions

This will make the variable visible to all the functions, defined after its definition

You can always decide to use a separate *declaration* too

- To do so, you can use the *extern* keyword

Global Variables

In the example, we had to pass the string to both functions – `trim()` and `startsWith()`

- This seems a rather wasted effort !!

A global variable in C (and many other languages) is a variable whose scope is *global*

By global, we mean that the variable is visible to all the functions in the program

To do so, we must define the variable outside all the functions

This will make the variable visible to all the functions, defined after its definition

You can always decide to use a separate *declaration* too

- To do so, you can use the *extern* keyword
- However, as long as we are not splitting our code across multiple C files, it doesn't make much sense
- Still, check the homework :P

Global Variables

In the example, we had to pass the string to both functions – `trim()` and `startsWith()`

- This seems a rather wasted effort !!

A global variable in C (and many other languages) is a variable whose scope is *global*

By global, we mean that the variable is visible to all the functions in the program

To do so, we must define the variable outside all the functions

This will make the variable visible to all the functions, defined after its definition

You can always decide to use a separate *declaration* too

- To do so, you can use the *extern* keyword
- However, as long as we are not splitting our code across multiple C files, it doesn't make much sense
- Still, check the homework :P

In the example, it makes sense to make the `str` variable global

Example: Using a global string variable

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

char* trim();
short startsWith(char* prefix);

char* str = NULL;

int main()
{
        str = "  Hello  ";
        printf("%d\n", startsWith("Hello"));
        str = trim();
        printf("%d\n", startsWith("Hello"));
        free(str);
}
```

Example: Using a global string variable

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

char* trim();
short startsWith(char* prefix);

char* str = NULL;

int main()
{
        str = "  Hello  ";
        printf("%d\n", startsWith("Hello"));
        str = trim();
        printf("%d\n", startsWith("Hello"));
        free(str);
}
```

The `str` variable is now available to all functions – its scope is global now

Example: Using a global string variable

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

char* trim();
short startsWith(char* prefix);

char* str = NULL;

int main()
{
        str = "  Hello  ";
        printf("%d\n", startsWith("Hello"));
        str = trim();
        printf("%d\n", startsWith("Hello"));
        free(str);
}
```

So now, the number of arguments required to be passed to `trim()` and `startsWith()` also go down by 1

Example: Using a global string variable

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

char* trim();
short startsWith(char* prefix);

char* str = NULL;

int main()
{
        str = "  Hello  ";
        printf("%d\n", startsWith("Hello"));
        str = trim();
        printf("%d\n", startsWith("Hello"));
        free(str);
}
```

Our function calls too, become “thinner”, i.e., more compact

Example: Using a global string variable

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 7$ gcc StringUtilsWithGlobalString.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 7$ ./a.out
0
1
saurabh@saurabh-VirtualBox:~/C/examples/Week 7$
```

Example: Using a global string variable

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 7$ gcc StringUtilsWithGlobalString.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 7$ ./a.out
0
1
saurabh@saurabh-VirtualBox:~/C/examples/Week 7$
```

The output remains the same !!

Example: Using a global string variable

```
char* trim()
{
        int len = strlen(str), beg = 0, end = len-1, ctr = 0;
        char* result = NULL;
        while(beg <= end && str[beg] == ' ')
                beg++;
        while(end >= beg && str[end] == ' ')
                end--;
        result = (char*) malloc(sizeof(char) * (end - beg + 2));
        while(beg <= end)
                result[ctr++] = str[beg++];
        result[ctr] = '\0';
        return result;
}

short startsWith(char* prefix)
{
        int len1 = strlen(str);
        int len2 = strlen(prefix);
        int i = 0, j = 0;
        if(len2 > len1)
                return 0;
        while(i < len2 && str[j] == prefix[i])
                i++, j++;
        return i == len2;
}
```

Example: Using a global string variable

```
char* trim()
{
        int len = strlen(str), beg = 0, end = len-1, ctr = 0;
        char* result = NULL;
        while(beg <= end && str[beg] == ' ')
                beg++;
        while(end >= beg && str[end] == ' ')
                end--;
        result = (char*) malloc(sizeof(char) * (end - beg + 2));
        while(beg <= end)
                result[ctr++] = str[beg++];
        result[ctr] = '\0';
        return result;
}

short startsWith(char* prefix)
{
        int len1 = strlen(str);
        int len2 = strlen(prefix);
        int i = 0, j = 0;
        if(len2 > len1)
                return 0;
        while(i < len2 && str[j] == prefix[i])
                i++, j++;
        return i == len2;
}
```

… and these are the changed definitions for the two methods !!

Example: Using a global string variable

```
char* trim()
{
        int len = strlen(str), beg = 0, end = len-1, ctr = 0;
        char* result = NULL;
        while(beg <= end && str[beg] == ' ')
                beg++;
        while(end >= beg && str[end] == ' ')
                end--;
        result = (char*) malloc(sizeof(char) * (end - beg + 2));
        while(beg <= end)
                result[ctr++] = str[beg++];
        result[ctr] = '\0';
        return result;
}
```

```
short startsWith(char* prefix)
{
        int len1 = strlen(str);
        int len2 = strlen(prefix);
        int i = 0, j = 0;
        if(len2 > len1)
                return 0;
        while(i < len2 && str[j] == prefix[i])
                i++, j++;
        return i == len2;
}
```

… and these are the changed definitions for the two methods !!

The only difference is in the header actually…

The static storage class

An advantage of global variables is that their values *persist* across function calls…

- … i.e. unlike the auto variables, their values are not lost when a function returns

The static storage class

An advantage of global variables is that their values *persist* across function calls…

- … i.e. unlike the auto variables, their values are not lost when a function returns

The disadvantage of global variables is that they are accessible *to all* the functions

- You cannot, for example, restrict their scope to only a few selected functions

The static storage class

An advantage of global variables is that their values *persist* across function calls…

- … i.e. unlike the auto variables, their values are not lost when a function returns

The disadvantage of global variables is that they are accessible *to all* the functions

- You cannot, for example, restrict their scope to only a few selected functions

Sometimes, you may only wish that a variable's value persists across multiple invocations…

- … of the same function !!

The static storage class

An advantage of global variables is that their values *persist* across function calls…

- … i.e. unlike the auto variables, their values are not lost when a function returns

The disadvantage of global variables is that they are accessible *to all* the functions

- You cannot, for example, restrict their scope to only a few selected functions

Sometimes, you may only wish that a variable's value persists across multiple invocations…

- … of the same function !!

You can do that by making its scope global

- But then, the variable will become accessible to other functions as well

The static storage class

An advantage of global variables is that their values *persist* across function calls…

- … i.e. unlike the auto variables, their values are not lost when a function returns

The disadvantage of global variables is that they are accessible *to all* the functions

- You cannot, for example, restrict their scope to only a few selected functions

Sometimes, you may only wish that a variable's value persists across multiple invocations…

- … of the same function !!

You can do that by making its scope global

- But then, the variable will become accessible to other functions as well

A storage class in C which is somewhat in between auto and global variables is `static`

The static storage class

An advantage of global variables is that their values *persist* across function calls…

- … i.e. unlike the auto variables, their values are not lost when a function returns

The disadvantage of global variables is that they are accessible *to all* the functions

- You cannot, for example, restrict their scope to only a few selected functions

Sometimes, you may only wish that a variable's value persists across multiple invocations…

- … of the same function !!

You can do that by making its scope global

- But then, the variable will become accessible to other functions as well

A storage class in C which is somewhat in between auto and global variables is `static`

A static variable behaves like an auto variable, but its storage is not freed on function's return

- Thus, future invocations of the same function, can use the last set value for the variable

Example: A simple stop watch !!

```c
#include<stdio.h>
#include<time.h>

double click()
{
        static clock_t prior;
        clock_t now = clock();
        // printf("%ld, %ld\n", (long)prior, (long)now);
        // Uncomment the above line, to check the values
        double diff = (double)(now - prior)/ CLOCKS_PER_SEC;
        prior = now;
        return diff;
}

int main()
{
        double time;
        long ctr = 1000000;
        printf("Let me print something first...\n");
        printf("YAAAAWWN !!\n");
        click();
        while(ctr--);
        time = click();
        printf("I guess I dozed off for %lf seconds\n", time);
        printf("YAAAAWWN !!\n");
        ctr = 100000000;
        click();
        while(ctr--);
        time = click();
        printf("Did I sleep again for %lf seconds?\n", time);
}
```

Example: A simplistic stop watch !!

```c
#include<stdio.h>
#include<time.h>

double click()
{
        static clock_t prior;
        clock_t now = clock();
        // printf("%ld, %ld\n", (long)prior, (long)now);
        // Uncomment the above line, to check the values
        double diff = (double)(now - prior)/ CLOCKS_PER_SEC;
        prior = now;
        return diff;
}

int main()
{
        double time;
        long ctr = 1000000;
        printf("Let me print something first...\n");
        printf("YAAAAWWN !!\n");
        click();
        while(ctr--);
        time = click();
        printf("I guess I dozed off for %lf seconds\n", time);
        printf("YAAAAWWN !!\n");
        ctr = 100000000;
        click();
        while(ctr--);
        time = click();
        printf("Did I sleep again for %lf seconds?\n", time);
}
```

Here, we use a function called `click()`, which returns the rough amount of time that elapsed between its current, and previous call

Example: A simplistic stop watch !!

```c
#include<stdio.h>
#include<time.h>

double click()
{
        static clock_t prior;
        clock_t now = clock();
        // printf("%ld, %ld\n", (long)prior, (long)now);
        // Uncomment the above line, to check the values
        double diff = (double)(now - prior)/ CLOCKS_PER_SEC;
        prior = now;
        return diff;
}

int main()
{
        double time;
        long ctr = 1000000;
        printf("Let me print something first...\n");
        printf("YAAAAWWN !!\n");
        click();
        while(ctr--);
        time = click();
        printf("I guess I dozed off for %lf seconds\n", time);
        printf("YAAAAWWN !!\n");
        ctr = 100000000;
        click();
        while(ctr--);
        time = click();
        printf("Did I sleep again for %lf seconds?\n", time);
}
```

We create a `static` variable called `prior` in the function

Example: A simplistic stop watch !!

```c
#include<stdio.h>
#include<time.h>

double click()
{
        static clock_t prior;
        clock_t now = clock();
        // printf("%ld, %ld\n", (long)prior, (long)now);
        // Uncomment the above line, to check the values
        double diff = (double)(now - prior)/ CLOCKS_PER_SEC;
        prior = now;
        return diff;
}

int main()
{
        double time;
        long ctr = 1000000;
        printf("Let me print something first...\n");
        printf("YAAAAWWN !!\n");
        click();
        while(ctr--);
        time = click();
        printf("I guess I dozed off for %lf seconds\n", time);
        printf("YAAAAWWN !!\n");
        ctr = 100000000;
        click();
        while(ctr--);
        time = click();
        printf("Did I sleep again for %lf seconds?\n", time);
}
```

We create a `static` variable called `prior` in the function

Another property of static variables is that they are **zero-initialized**, i.e., they get initialized with `0` (as against the auto variables, which, if not initialized specifically, start with a garbage value)

Example: A simplistic stop watch !!

```c
#include<stdio.h>
#include<time.h>

double click()
{
        static clock_t prior;
        clock_t now = clock();
        // printf("%ld, %ld\n", (long)prior, (long)now);
        // Uncomment the above line, to check the values
        double diff = (double)(now - prior)/ CLOCKS_PER_SEC;
        prior = now;
        return diff;
}

int main()
{
        double time;
        long ctr = 1000000;
        printf("Let me print something first...\n");
        printf("YAAAAWWN !!\n");
        click();
        while(ctr--);
        time = click();
        printf("I guess I dozed off for %lf seconds\n", time);
        printf("YAAAAWWN !!\n");
        ctr = 100000000;
        click();
        while(ctr--);
        time = click();
        printf("Did I sleep again for %lf seconds?\n", time);
}
```

We make three calls to the `click()` function from `main()`

Example: A simplistic stop watch !!

```c
#include<stdio.h>
#include<time.h>

double click()
{
        static clock_t prior;
        clock_t now = clock();
        // printf("%ld, %ld\n", (long)prior, (long)now);
        // Uncomment the above line, to check the values
        double diff = (double)(now - prior)/ CLOCKS_PER_SEC;
        prior = now;
        return diff;
}

int main()
{
        double time;
        long ctr = 1000000;
        printf("Let me print something first...\n");
        printf("YAAAAWWN !!\n");
        click();
        while(ctr--);
        time = click();
        printf("I guess I dozed off for %lf seconds\n", time);
        printf("YAAAAWWN !!\n");
        ctr = 100000000;
        click();
        while(ctr--);
        time = click();
        printf("Did I sleep again for %lf seconds?\n", time);
}
```

We make three calls to the `click()` function from `main()`

There is some “artificial” work too, that we do between these invocations (the two `while` loops)

Example: A simplistic stop watch !!

```c
#include<stdio.h>
#include<time.h>

double click()
{
        static clock_t prior;
        clock_t now = clock();
        // printf("%ld, %ld\n", (long)prior, (long)now);
        // Uncomment the above line, to check the values
        double diff = (double)(now - prior)/ CLOCKS_PER_SEC;
        prior = now;
        return diff;
}

int main()
{
        double time;
        long ctr = 1000000;
        printf("Let me print something first...\n");
        printf("YAAAAWWN !!\n");
        click();
        while(ctr--);
        time = click();
        printf("I guess I dozed off for %lf seconds\n", time);
        printf("YAAAAWWN !!\n");
        ctr = 100000000;
        click();
        while(ctr--);
        time = click();
        printf("Did I sleep again for %lf seconds?\n", time);
}
```

Each time, we find the current relevant timestamp

Example: A simplistic stop watch !!

```c
#include<stdio.h>
#include<time.h>

double click()
{
        static clock_t prior;
        clock_t now = clock();
        // printf("%ld, %ld\n", (long)prior, (long)now);
        // Uncomment the above line, to check the values
        double diff = (double)(now - prior)/ CLOCKS_PER_SEC;
        prior = now;
        return diff;
}

int main()
{
        double time;
        long ctr = 1000000;
        printf("Let me print something first...\n");
        printf("YAAAAWWN !!\n");
        click();
        while(ctr--);
        time = click();
        printf("I guess I dozed off for %lf seconds\n", time);
        printf("YAAAAWWN !!\n");
        ctr = 100000000;
        click();
        while(ctr--);
        time = click();
        printf("Did I sleep again for %lf seconds?\n", time);
}
```

… and find its difference with the timestamp collected in the previous invocation

Example: A simplistic stop watch !!

```c
#include<stdio.h>
#include<time.h>

double click()
{
        static clock_t prior;
        clock_t now = clock();
        // printf("%ld, %ld\n", (long)prior, (long)now);
        // Uncomment the above line, to check the values
        double diff = (double)(now - prior)/ CLOCKS_PER_SEC;
        prior = now;
        return diff;
}

int main()
{
        double time;
        long ctr = 1000000;
        printf("Let me print something first...\n");
        printf("YAAAAWWN !!\n");
        click();
        while(ctr--);
        time = click();
        printf("I guess I dozed off for %lf seconds\n", time);
        printf("YAAAAWWN !!\n");
        ctr = 100000000;
        click();
        while(ctr--);
        time = click();
        printf("Did I sleep again for %lf seconds?\n", time);
}
```

… and then we store this timestamp in the `prior` variable

Example: A simplistic stop watch !!

```c
#include<stdio.h>
#include<time.h>

double click()
{
        static clock_t prior;
        clock_t now = clock();
        // printf("%ld, %ld\n", (long)prior, (long)now);
        // Uncomment the above line, to check the values
        double diff = (double)(now - prior)/ CLOCKS_PER_SEC;
        prior = now;
        return diff;
}

int main()
{
        double time;
        long ctr = 1000000;
        printf("Let me print something first...\n");
        printf("YAAAAWWN !!\n");
        click();
        while(ctr--);
        time = click();
        printf("I guess I dozed off for %lf seconds\n", time);
        printf("YAAAAWWN !!\n");
        ctr = 100000000;
        click();
        while(ctr--);
        time = click();
        printf("Did I sleep again for %lf seconds?\n", time);
}
```

… and then we store this timestamp in the `prior` variable

Since it is a static variable, it retains its value across the three invocations, allowing us to find the difference

Example: A simplistic stop watch !!

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 7$ gcc StopClock.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 7$ ./a.out
Let me print something first...
YAAAAWWN !!
I guess I dozed off for 0.002594 seconds
YAAAAWWN !!
Did I sleep again for 0.248393 seconds?
saurabh@saurabh-VirtualBox:~/C/examples/Week 7$
```

Example: A simplistic stop watch !!

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 7$ gcc StopClock.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 7$ ./a.out
Let me print something first...
YAAAAWWN !!
I guess I dozed off for 0.002594 seconds
YAAAAWWN !!
Did I sleep again for 0.248393 seconds?
saurabh@saurabh-VirtualBox:~/C/examples/Week 7$
```

This is the output…

Example: A simplistic stop watch !!

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 7$ gcc StopClock.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 7$ ./a.out
Let me print something first...
YAAAAWWN !!
I guess I dozed off for 0.002594 seconds
YAAAAWWN !!
Did I sleep again for 0.248393 seconds?
saurabh@saurabh-VirtualBox:~/C/examples/Week 7$
```

This is the output…

Try uncommenting some statements in the program, and convince yourself that you understand the use of the `prior` variable

About amalgamation of extern and static

We have not discussed breaking of C code across multiple files

- Mostly because it is something that is too specific for C development

About amalgamation of extern and static

We have not discussed breaking of C code across multiple files

- Mostly because it is something that is too specific for C development

However, if you do plan to work in C or C++ later, you will often see `static` and `extern` together

About amalgamation of extern and static

We have not discussed breaking of C code across multiple files

- Mostly because it is something that is too specific for C development

However, if you do plan to work in C or C++ later, you will often see `static` and `extern` together

The following information, is mostly for the sake of completeness (don't worry if you don't get it)

About amalgamation of extern and static

We have not discussed breaking of C code across multiple files

- Mostly because it is something that is too specific for C development

However, if you do plan to work in C or C++ later, you will often see `static` and `extern` together

The following information, is mostly for the sake of completeness (don't worry if you don't get it)

- A global variable declared in one C file, can be used in other C files as well…
- … provided that they are linked together during a build

About amalgamation of extern and static

We have not discussed breaking of C code across multiple files

- Mostly because it is something that is too specific for C development

However, if you do plan to work in C or C++ later, you will often see `static` and `extern` together

The following information, is mostly for the sake of completeness (don't worry if you don't get it)

- A global variable declared in one C file, can be used in other C files as well…
- … provided that they are linked together during a build
- In such case, you must provide a *declaration* of the variable in each file where you use the variable

About amalgamation of extern and static

We have not discussed breaking of C code across multiple files

- Mostly because it is something that is too specific for C development

However, if you do plan to work in C or C++ later, you will often see `static` and `extern` together

The following information, is mostly for the sake of completeness (don't worry if you don't get it)

- A global variable declared in one C file, can be used in other C files as well…
- … provided that they are linked together during a build
- In such case, you must provide a *declaration* of the variable in each file where you use the variable
- This declaration can be done by prefixing the keyword `extern` before the declaration

About amalgamation of extern and static

We have not discussed breaking of C code across multiple files

- Mostly because it is something that is too specific for C development

However, if you do plan to work in C or C++ later, you will often see `static` and `extern` together

The following information, is mostly for the sake of completeness (don't worry if you don't get it)

- A global variable declared in one C file, can be used in other C files as well…
- … provided that they are linked together during a build
- In such case, you must provide a *declaration* of the variable in each file where you use the variable
- This declaration can be done by prefixing the keyword `extern` before the declaration
- This is not required for functions though, because `extern` is implicitly applied to all functions

About amalgamation of extern and static

We have not discussed breaking of C code across multiple files

- Mostly because it is something that is too specific for C development

However, if you do plan to work in C or C++ later, you will often see `static` and `extern` together

The following information, is mostly for the sake of completeness (don't worry if you don't get it)

- A global variable declared in one C file, can be used in other C files as well…
- … provided that they are linked together during a build
- In such case, you must provide a *declaration* of the variable in each file where you use the variable
- This declaration can be done by prefixing the keyword `extern` before the declaration
- This is not required for functions though, because `extern` is implicitly applied to all functions
- If you want a global variable to be accessible only within the defined C file, you can make it `static`…
- … otherwise, it will remain accessible in other linked C code, and can be changed !!

Homework !!

If you are really interested, `extern` keyword can help you with building large C applications

- At least, read Section 1.10 and 4.3 from the C reference book (*The C Programming Language*)

Find out the other storage classes that exist in C

- This link can be helpful: https://www.geeksforgeeks.org/storage-classes-in-c/