A Brief Tour

"to get a deeper understanding of the language"

Deep C - a 3 day course
Jon Jagger & Olve Maudal

Exercise: Hello World!

- type in this code.
- compile and execute the program.
- what do you get?

hello.c

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

```
$ cc -o hello hello.c
$ ./hello
The answer is 42
$ echo $?
0
$
```

Was this the result you expected?

Of course it was!

But let's do a dry run and step through the code

(we will look at the compilation step later.)

The following is an example of what *might* happen when executing this code:

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

After some basic initialization the run-time environment will call the main function.

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}


int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

then preparation for the call to `calc()` starts by evaluating all the arguments to be passed to the function.

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}


int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{                         ?         ?                   ?
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

guess which argument is evaluated first?

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

unlike other popular programming laugages, the order of evaluation is mostly unspecified in C. In this case the compiler or runtime environment might choose to evaluate `life()` first

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}


int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}


int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}


int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}


int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe,    6    , everything());
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}


int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe,   6   , everything());
    printf("The answer is %d\n", a);
}
```

`universe` is replaced with 7.

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}


int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(     7    ,   6   , everything());
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}


int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7   ,   6   , everything());
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}


int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(     7    ,    6    , everything());
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}


int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7   ,   6   , everything());
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}


int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,   6   , everything());
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}


int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7    ,    6    ,        1        );
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7   ,   6   ,     1      );
    printf("The answer is %d\n", a);
}
```

the way arguments are passed to a function is defined by the calling convention (ABI) used by the compiler and runtime environment. Here is just an example...

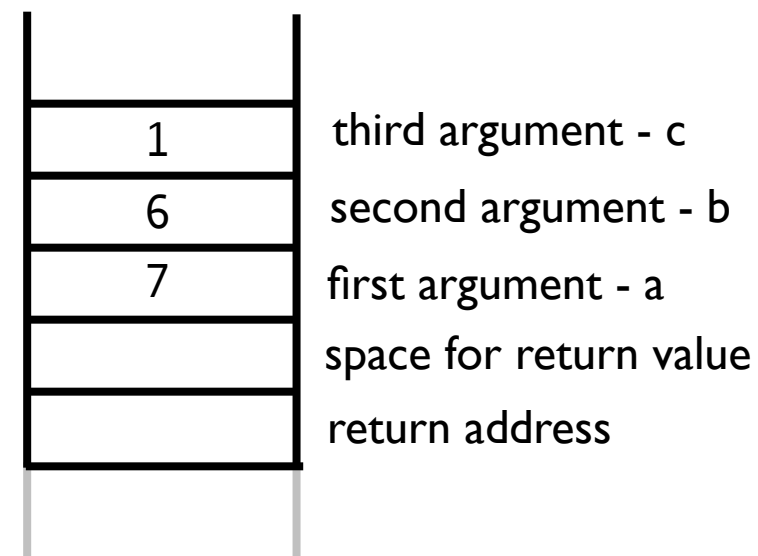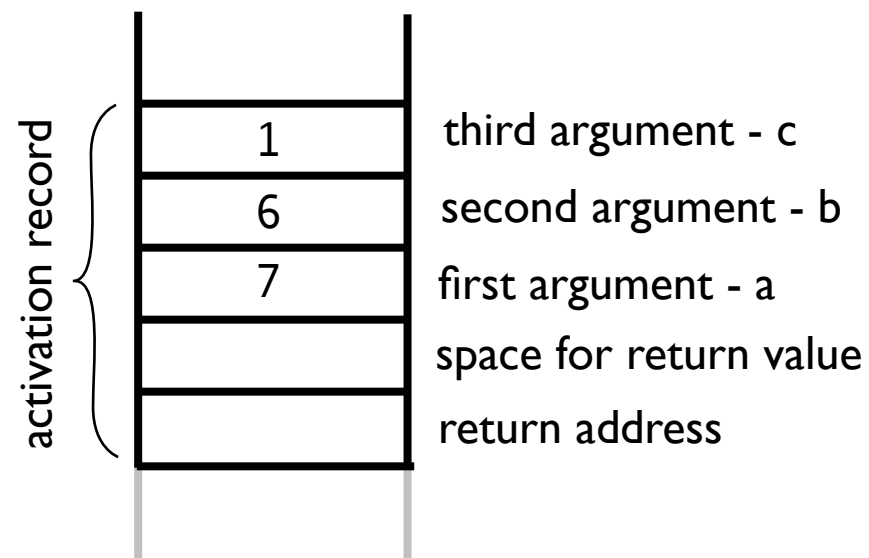|	
1	third argument - c
6	second argument - b
7	first argument - a
	space for return value
	return address

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7   ,   6   ,     1       );
    printf("The answer is %d\n", a);
}
```

and when the "activation record" is populated, the program can jump into the function.

activation record

1	third argument - c
6	second argument - b
7	first argument - a
	space for return value
	return address

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7   ,    6   ,      1      );
    printf("The answer is %d\n", a);
}
```

and then evaluate the expression.

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return 7 * 6 / 1;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7   ,   6   ,     1      );
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return   42  / 1;
}


int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7   ,   6   ,     1      );
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return    42   ;
}


int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7   ,   6   ,      1        );
    printf("The answer is %d\n", a);
}
```
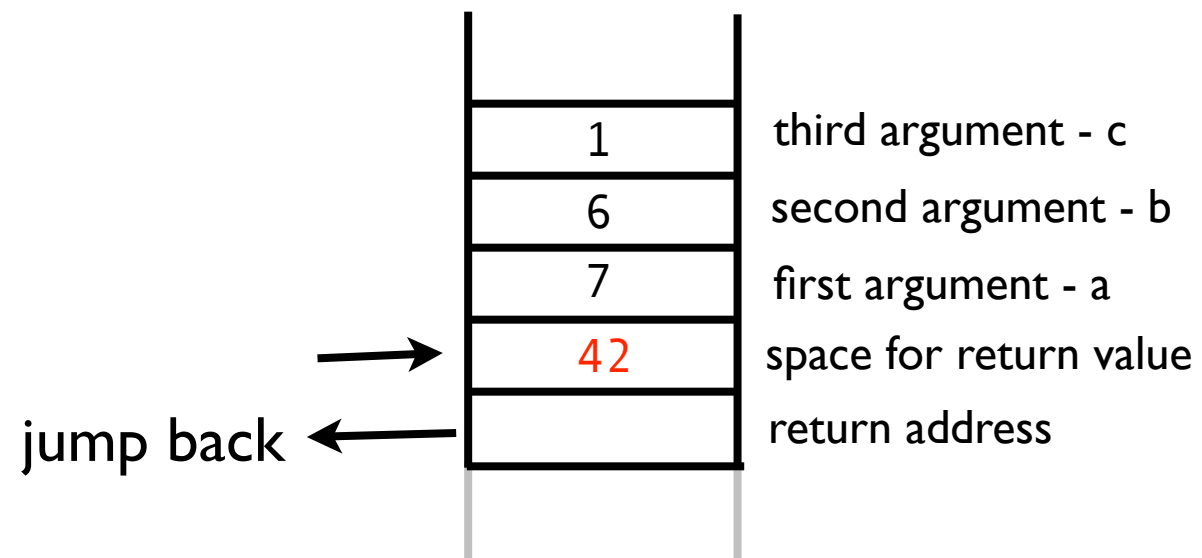
```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return    42   ;
}


int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7   ,   6   ,     1      );
    printf("The answer is %d\n", a);
}
```

1	third argument - c
6	second argument - b
7	first argument - a
42	space for return value
	return address

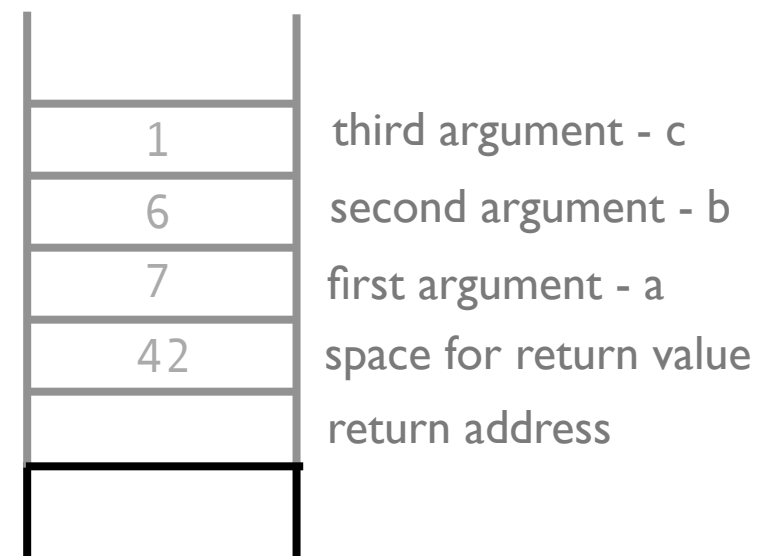jump back

```
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return    42   ;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(    7   ,    6    ,      1      );
    printf("The answer is %d\n", a);
}
```

1	third argument - c
6	second argument - b
7	first argument - a
42	space for return value
	return address

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return     42    ;
}


int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = 42;
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return     42    ;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = 42;
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return    42   ;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = 42;
    printf("The answer is %d\n", 42);
}
```

and then 42 and the pointer to the character string is pushed on the execution stack before the library function `printf()` is called

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return    42   ;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = 42;
    printf("The answer is %d\n", 42);
}
```

The answer is 42

The `printf()` function writes out to the standard output stream

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return    42   ;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = 42;
    printf("The answer is %d\n", 42);
}
```

```
The answer is 42
$ echo $?
0
```

and a default value to indicate success, in this case 0, is returned back to the run-time environment

Was this exactly what you expected?

Good!

This was just an example of what *might* happen.

Because...

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}


int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}


int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

maybe the compiler is clever and sees that `life()` and `everything()` always returns 6 and 1
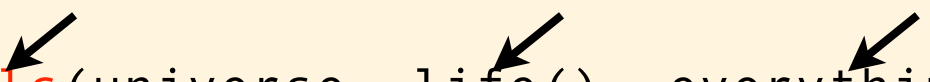
```c
#include <stdio.h>

static int calc(int a, int b, int c)
{
    return a * b / c;
}

int universe = 7;
static int life(void) { return 6; }
int everything(void) { return 1; }

int main(void)
{
    int a = calc(universe, life(), everything());
    printf("The answer is %d\n", a);
}
```

maybe the compiler is clever and sees that `life()` and `everything()` always returns 6 and 1

and then by inlining `calc()` perhaps the compiler choose to optimize the code into...

the C standard defines *what* the expected behaviour is, but says very little about *how* it should be implemented.

this is a key feature of C, and one of the reason why C is such a successful programming language on such a wide range of hardware!

Behavior

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

int main(void)
{
    // implementation-defined
    int i = ~0;
    i >>= 1;
    printf("%d\n", i);

    // unspecified
    printf("4") + printf("2");
    printf("\n");

    // undefined
    int k = INT_MAX;
    k += 1;
    printf("%d\n", k);
}
```

implementation-defined behavior: the construct is not incorrect; the code must compile; the compiler must document the behavior

unspecified behavior: the same as implementation-defined except the behavior need not be documented

undefined behavior: the standard imposes no requirements ; anything at all can happen, all bets are off, nasal demons might fly out of your nose.

Note that many compilers will not give you any warnings when compiling this code, and due to the undefined behavior caused by signed integer overflow above, the whole program is in theory undefined.

```c
#include <stdio.h>

int a(void) { printf("a"); return 3; }
int b(void) { printf("b"); return 4; }

int main(void)
{
    int c = a() + b();
    printf("%d\n", c);
}
```

According to the C standard, this program will print:

`ba7`   or   `ab7`

Unlike most modern programming languages, the evaluation order of most expressions are *not* specified in C

And while we are on a roll... What do you think might happen when we run this program?

```c
#include <stdio.h>

int main(void)
{
    int v[] = {9,7,5,3,1};
    int i = 2;
    int n = v[i++] - v[i++];
    printf("%d\n", n);
    printf("%d\n", i);
    return 0;
}
```

you might get:

```
42
0
```

Try it!

If you break the rules of the language, the behaviour of the whole program is undefined. Anything can happen! And the compiler will often not be able to give you a warning.

```c
#include <stdio.h>

int main(void)
{
    int v[] = {9,7,5,3,1};
    int i = 2;
    int n = v[i++] - v[i++];
    printf("%d\n", n);
    printf("%d\n", i);
    return 0;
}
```

Here is what I get on my machine:

```
$ gcc -O -Wall -Wextra -pedantic foo.c
$ ./a.out
0
4
$
```

The violation here is that we are violating the rules of sequencing, which, among other things says that a variable can not be updated twice between two sequence points.

```c
#include <stdio.h>

int main(void)
{
    int v[] = {9,7,5,3,1};
    int i = 2;
    int n = v[i++] - v[i++];
    printf("%d\n", n);
    printf("%d\n", i);
    return 0;
}
```

Here is what I get on my machine:

```
$ clang -O -Wall -Wextra -pedantic foo.c
error: multiple unsequenced modifications to 'i' [-Werror,-Wunsequenced]
    int n = v[i++] - v[i++];
              ^           ~~
```

We just had a few glimpses of what *might* happen when code is executed.

Scary stuff? Not really, but with only a shallow understanding of the language it is easy to make big mistakes.

The goal of this course is to give you a deep understanding of C, by starting from scratch an relearn the language in a systematic way.

A Brief Tour
tools and vocabulary

Summary
- hello world!
- implementation-defined behaviour
- unspecified behaviour
- undefined behaviour