The Clones of Mr C

ESC101: Fundamentals of Computing

Purushottam Kar

Announcements

- Last date for dropping Advanced Track October 12
  - Application must be an email to instructor, mentors, teammates
- Last date for dropping ESC101 course October 12
  - Application must be on standard DoAA course drop form – no email!
- Joint tutorial for B1 and B14 on October 12
  - 12 – 1 PM (same time), L19 - just an arrangement for this week ☺

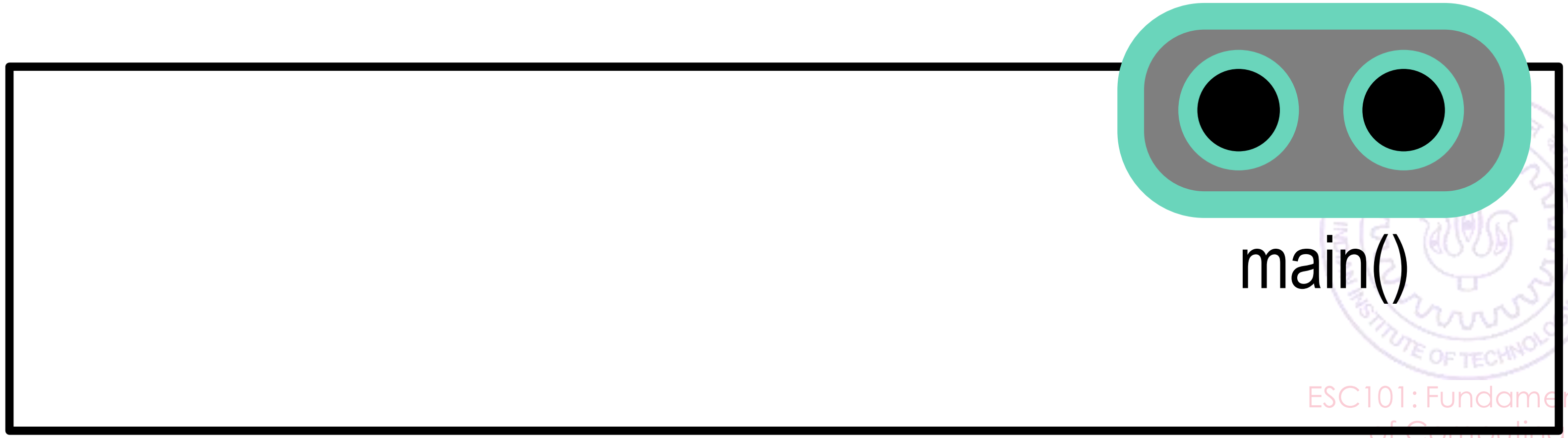
Clones!

Clones!

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + b;
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```

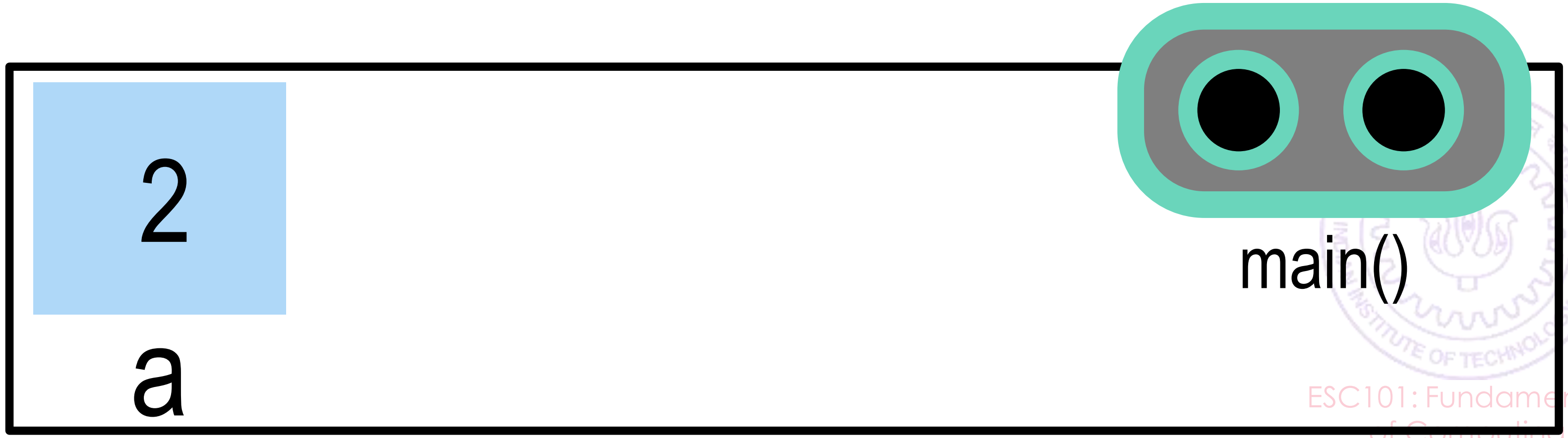
Clones!

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + b;
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



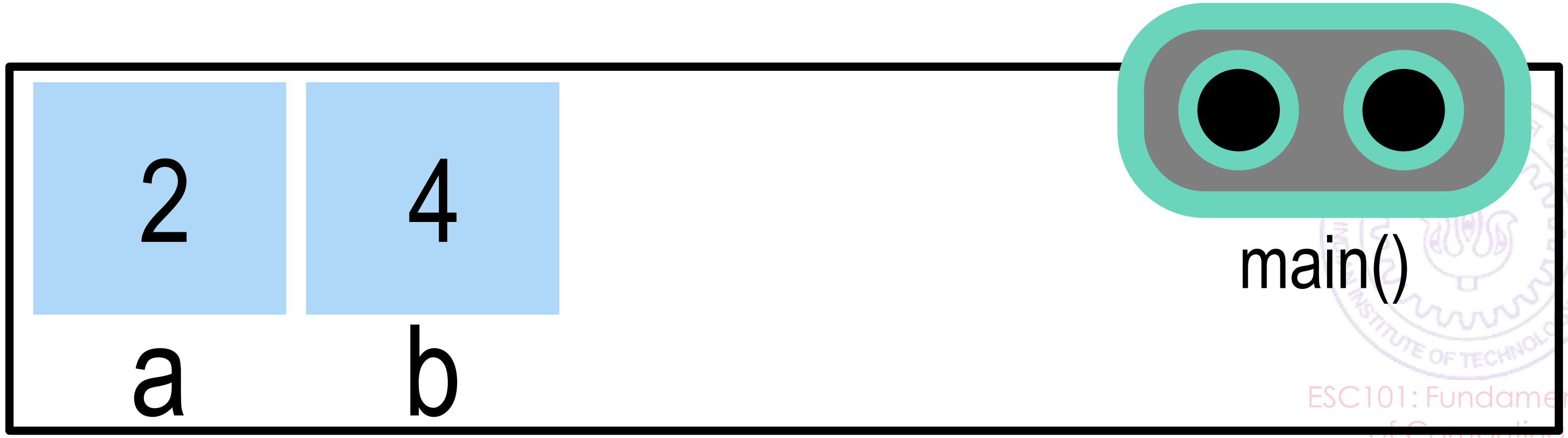
Clones!

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + b;
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



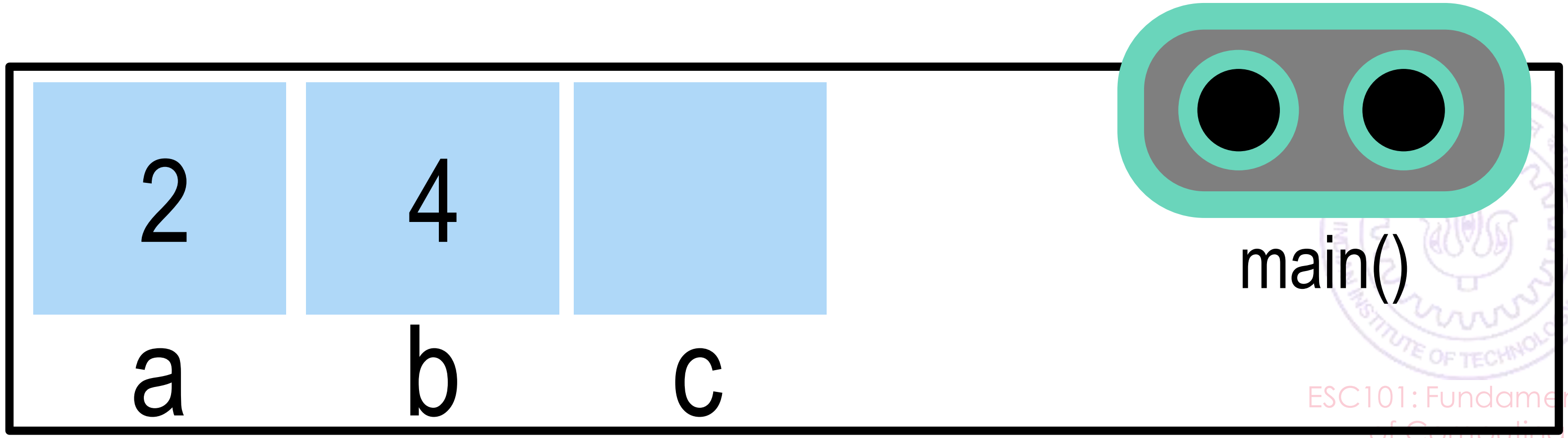
Clones!

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + b;
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



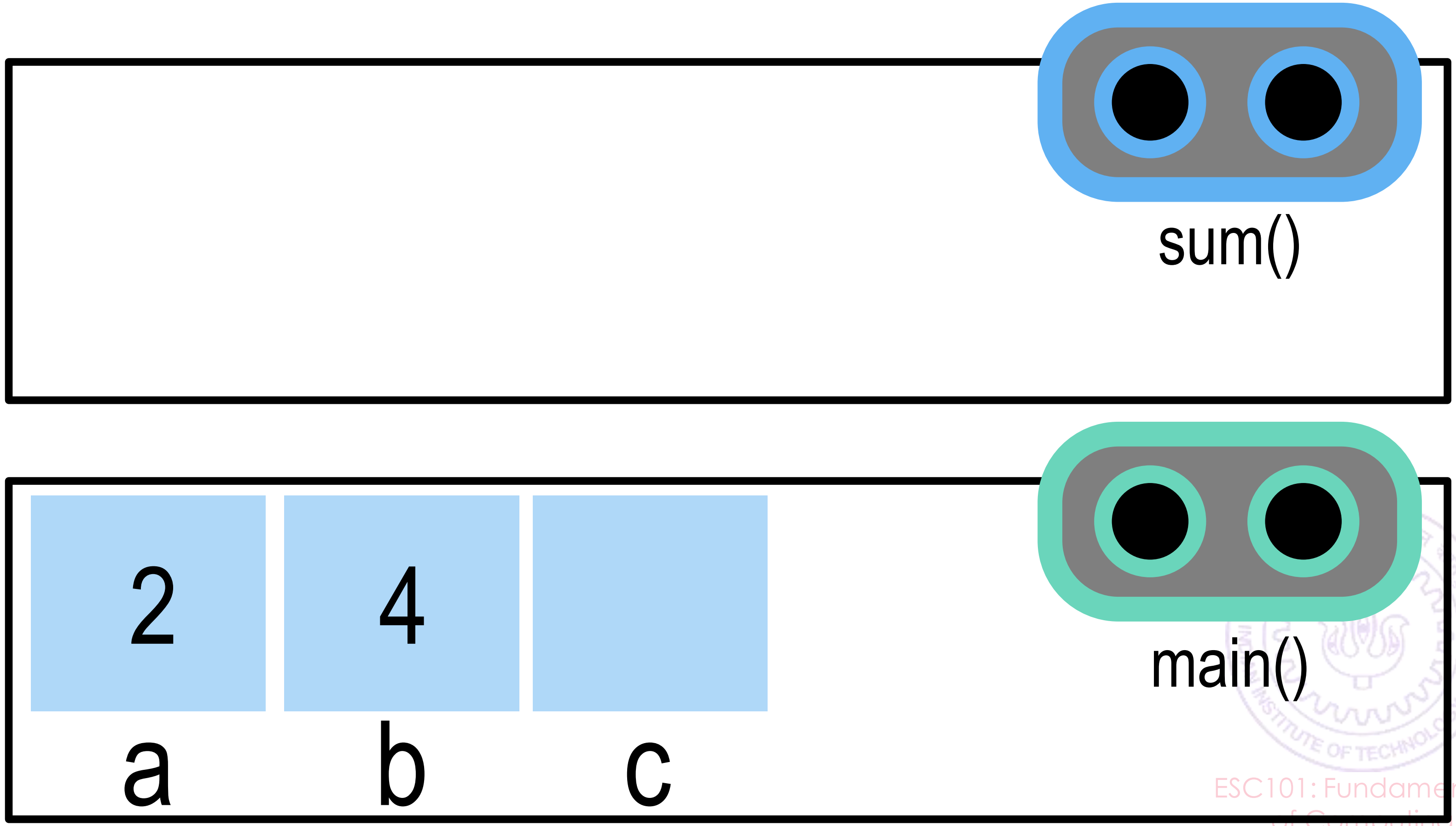
Clones!

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + b;
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



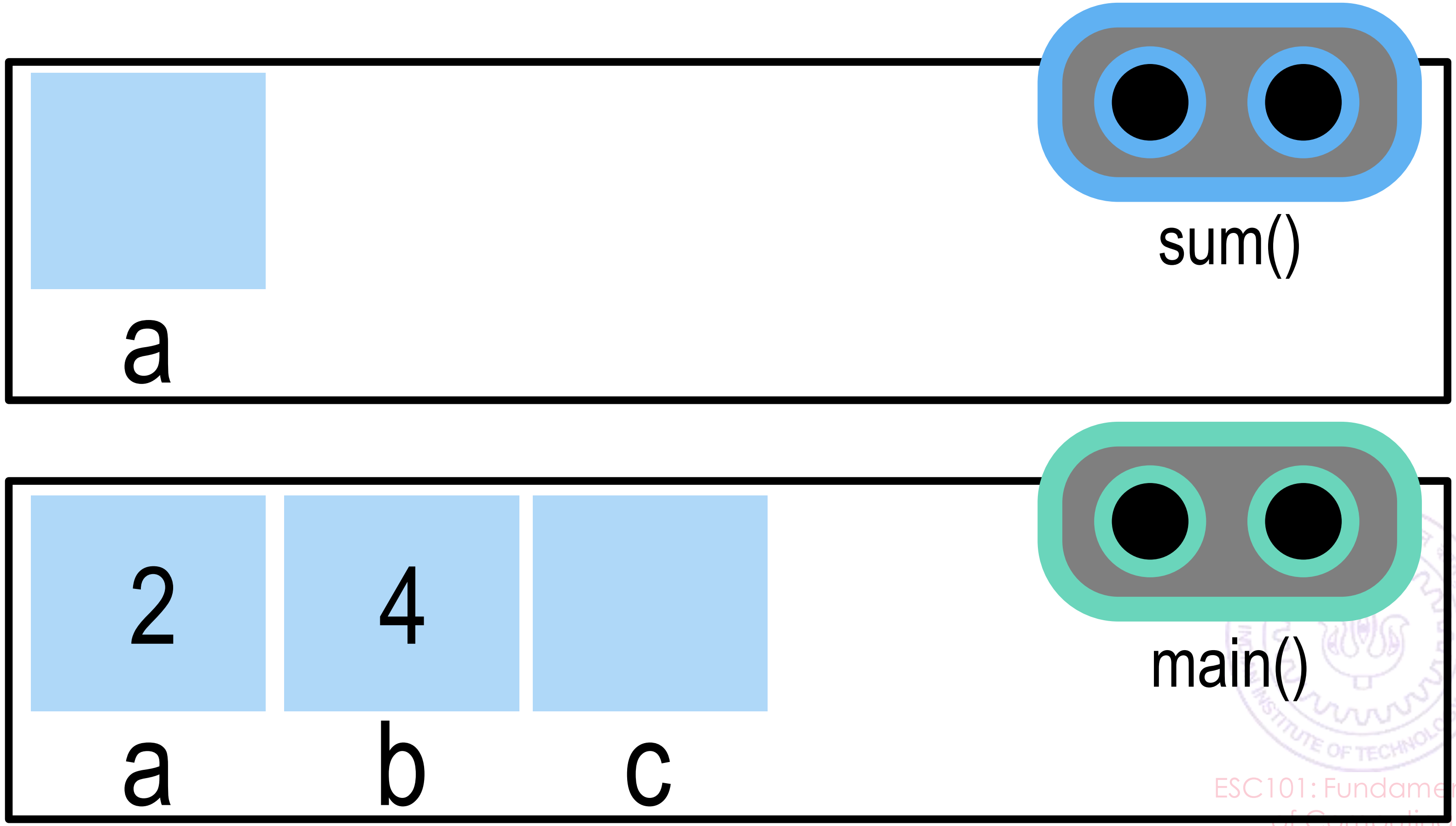
Clones!

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + b;
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



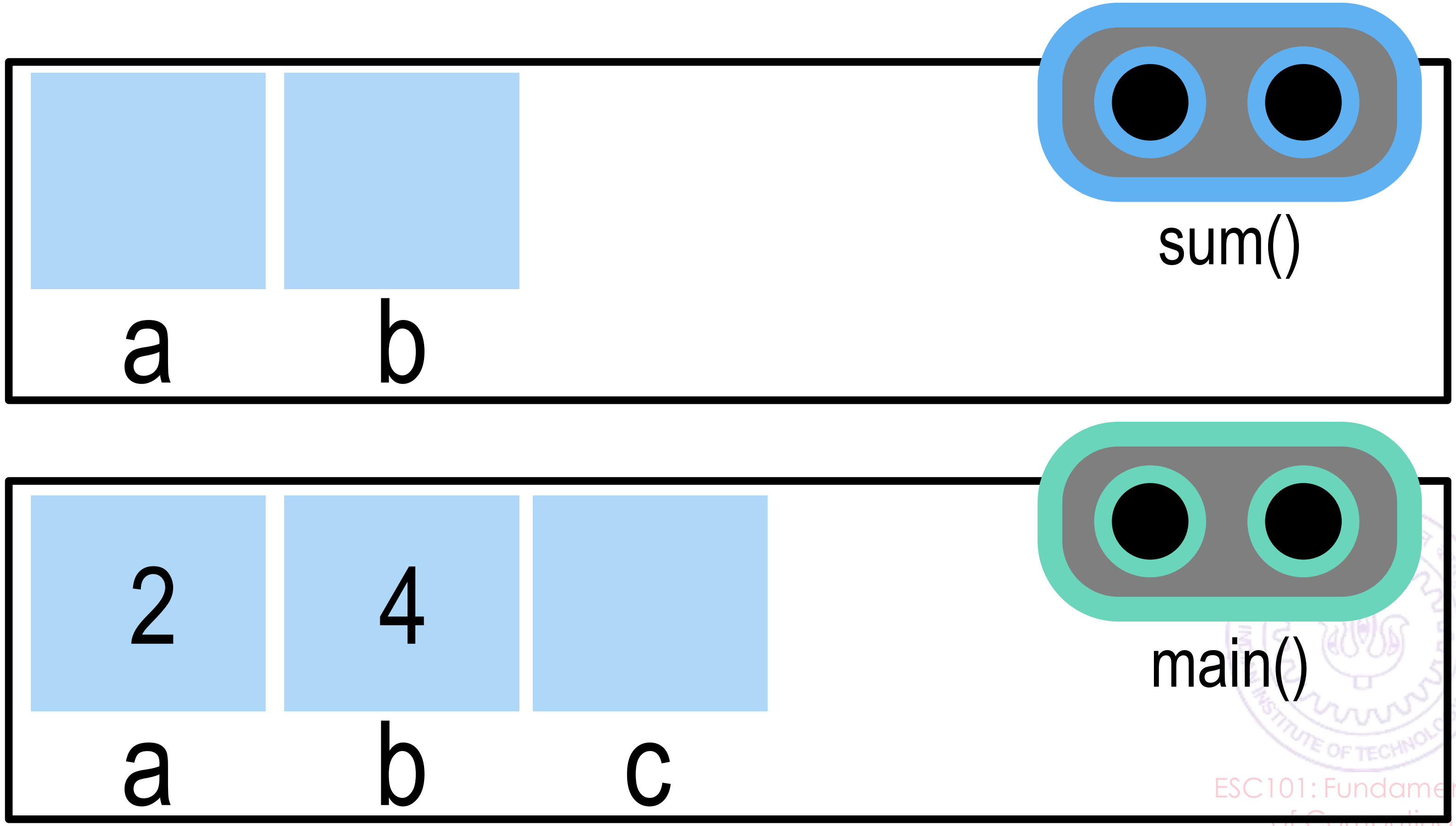
Clones!

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + b;
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



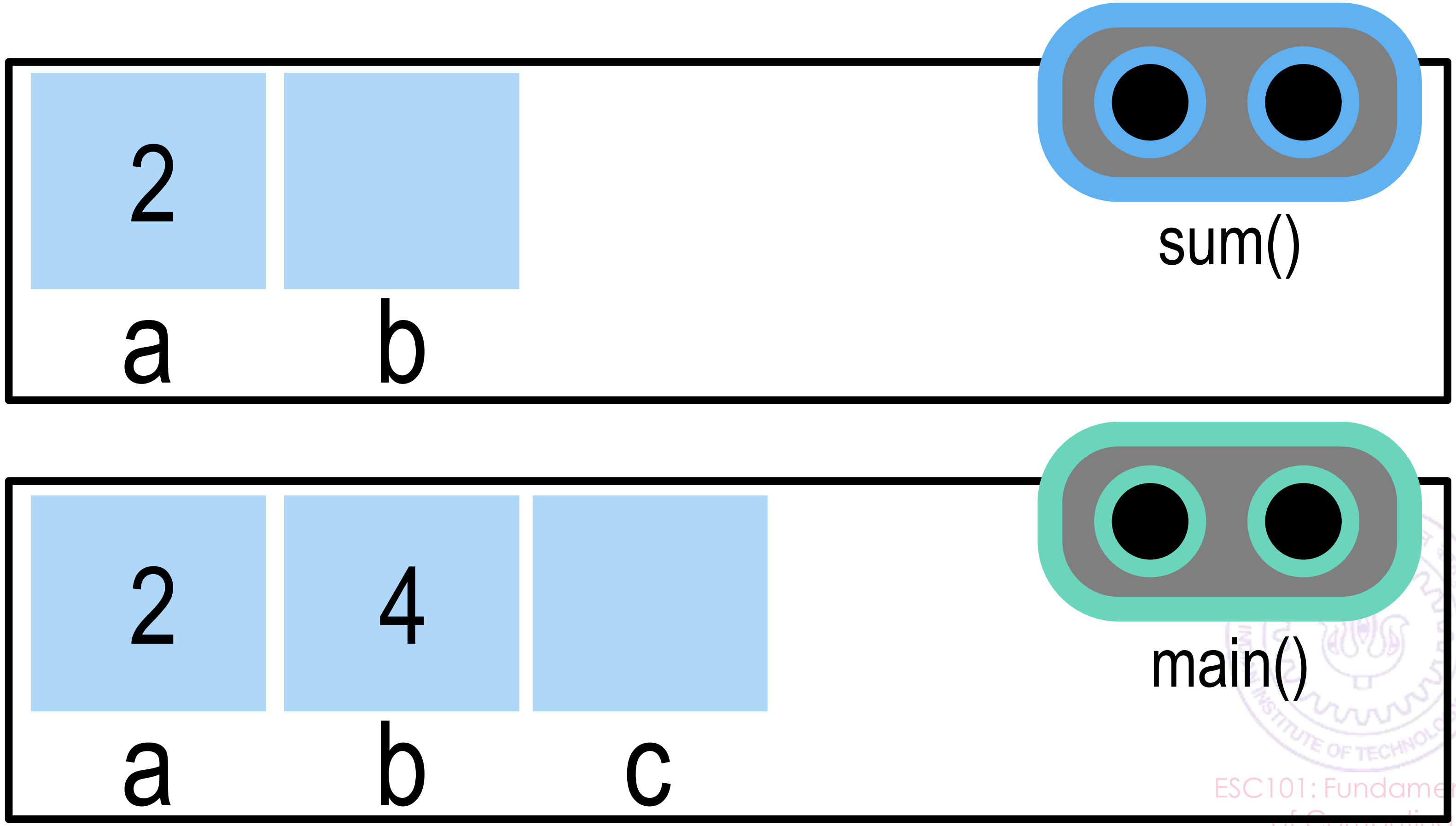
Clones!

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + b;
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



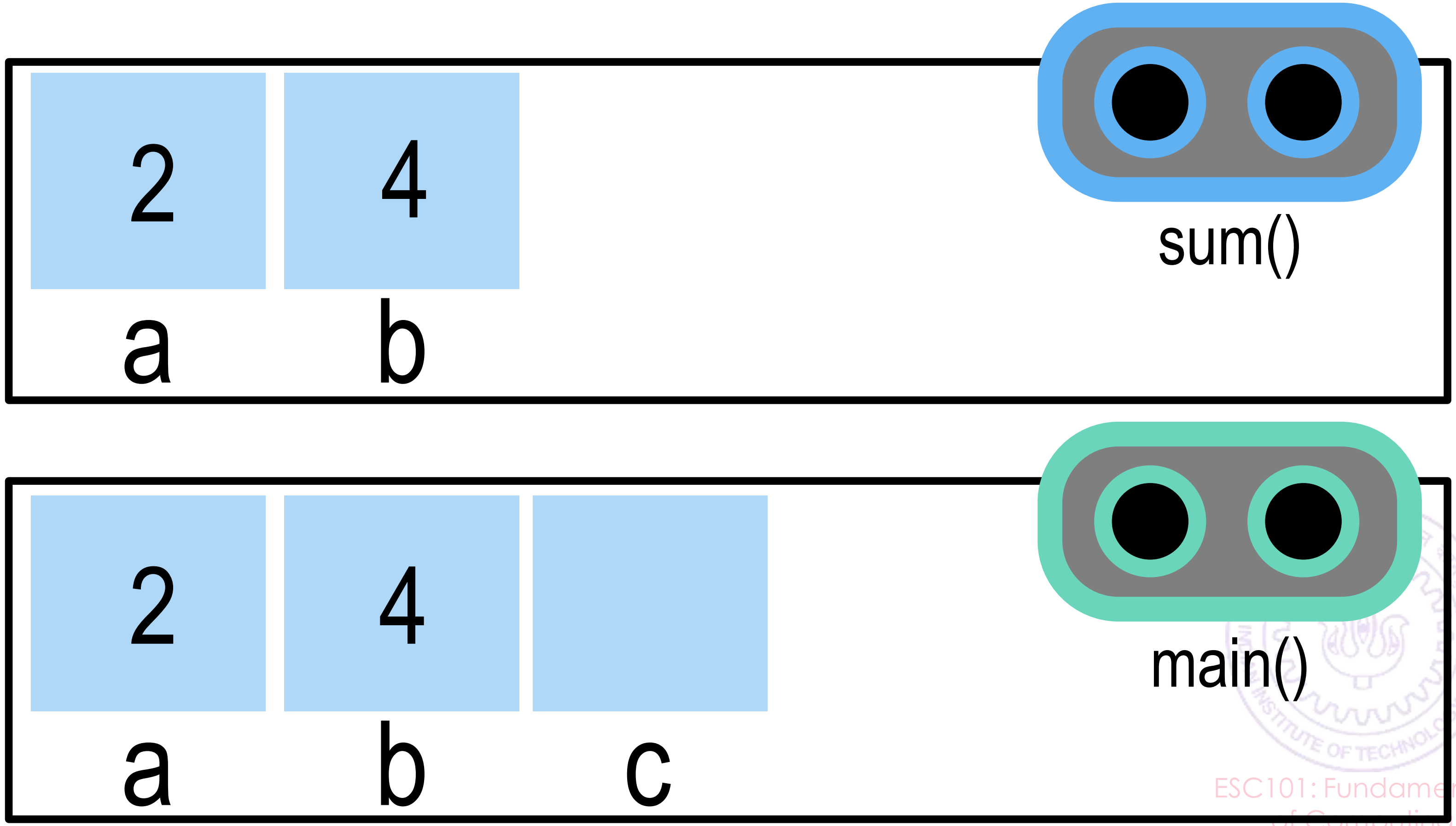
Clones!

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + b;
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



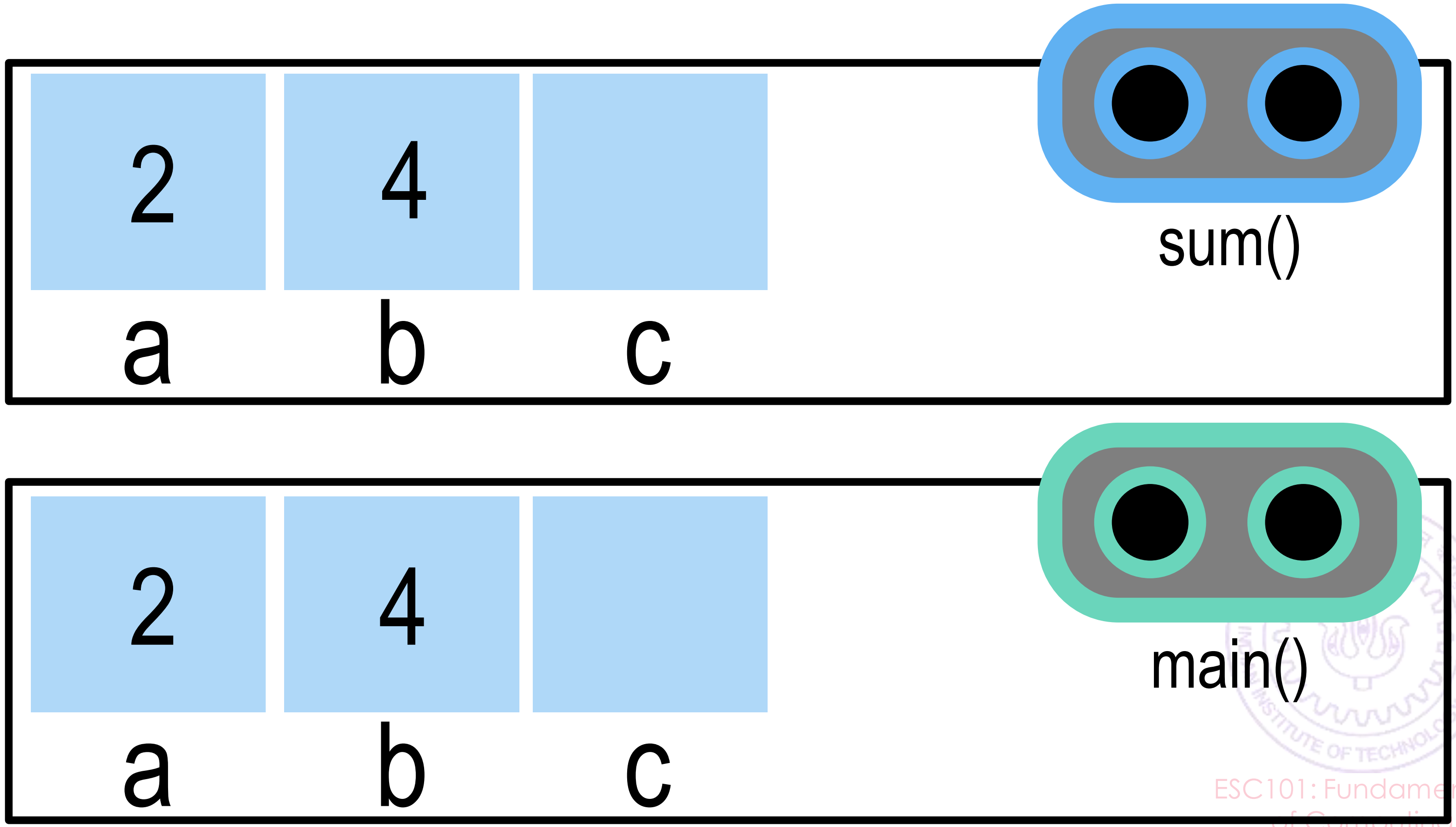
Clones!

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + b;
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



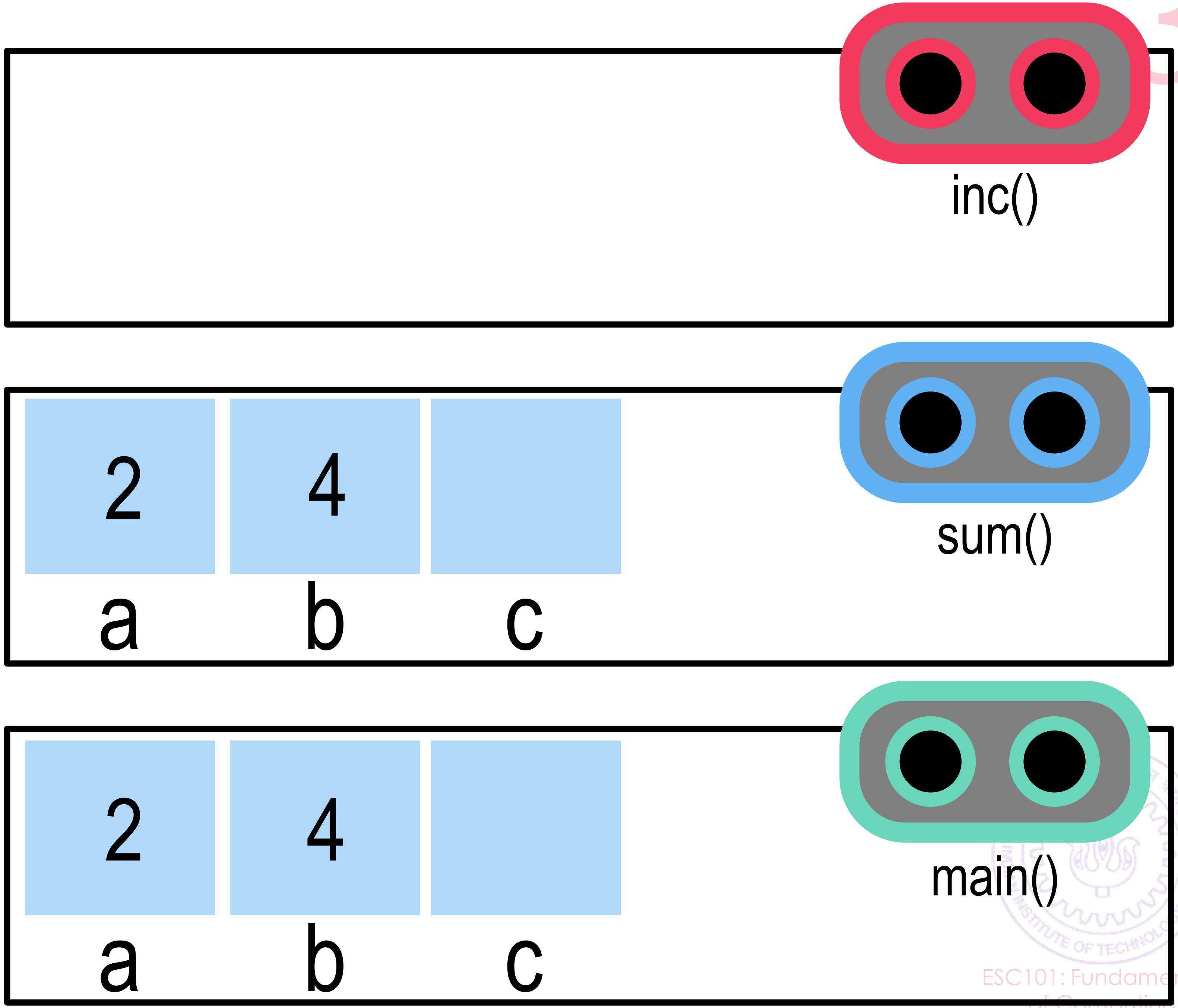
Clones!

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + b;
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



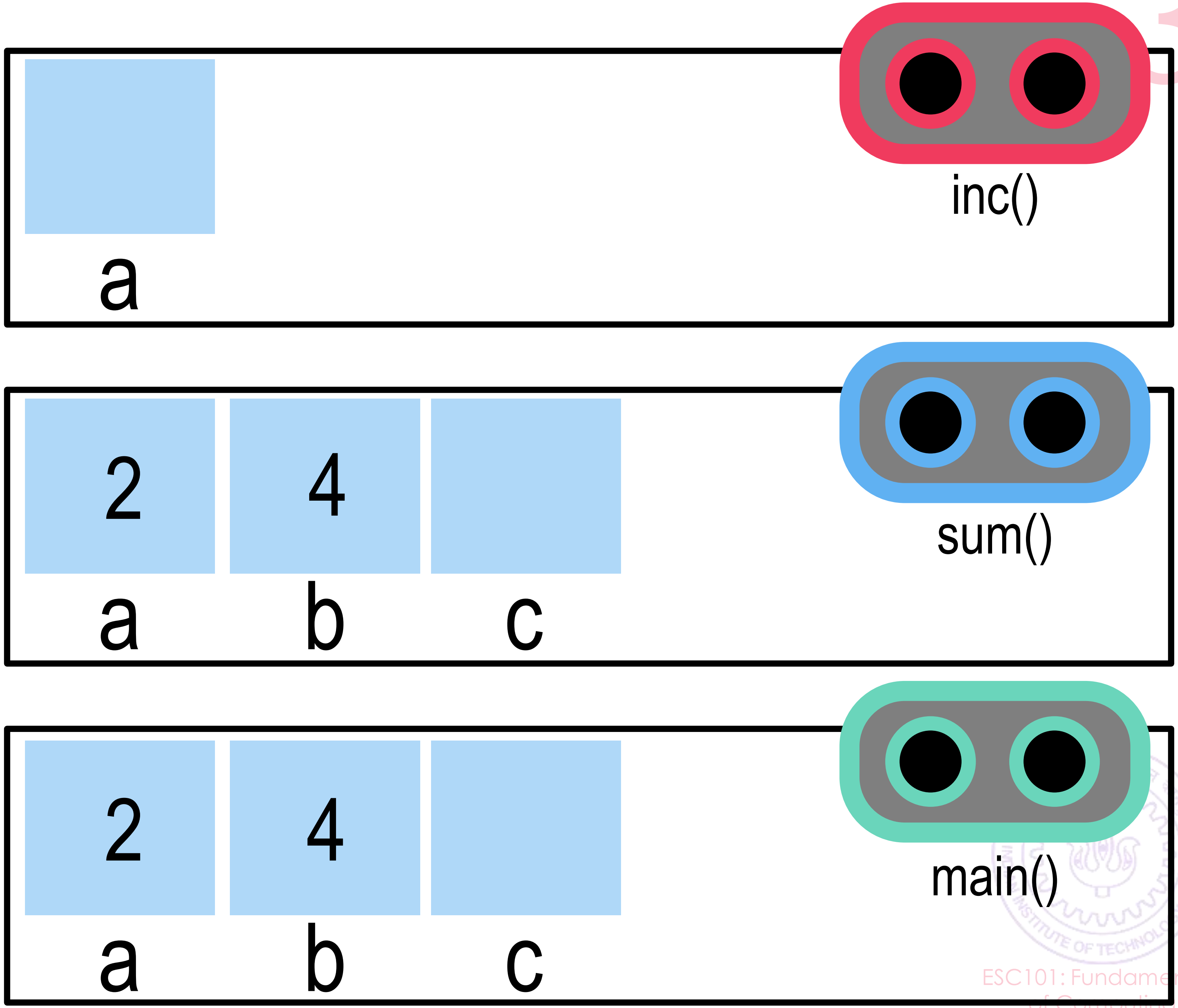
Clones!

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + b;
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



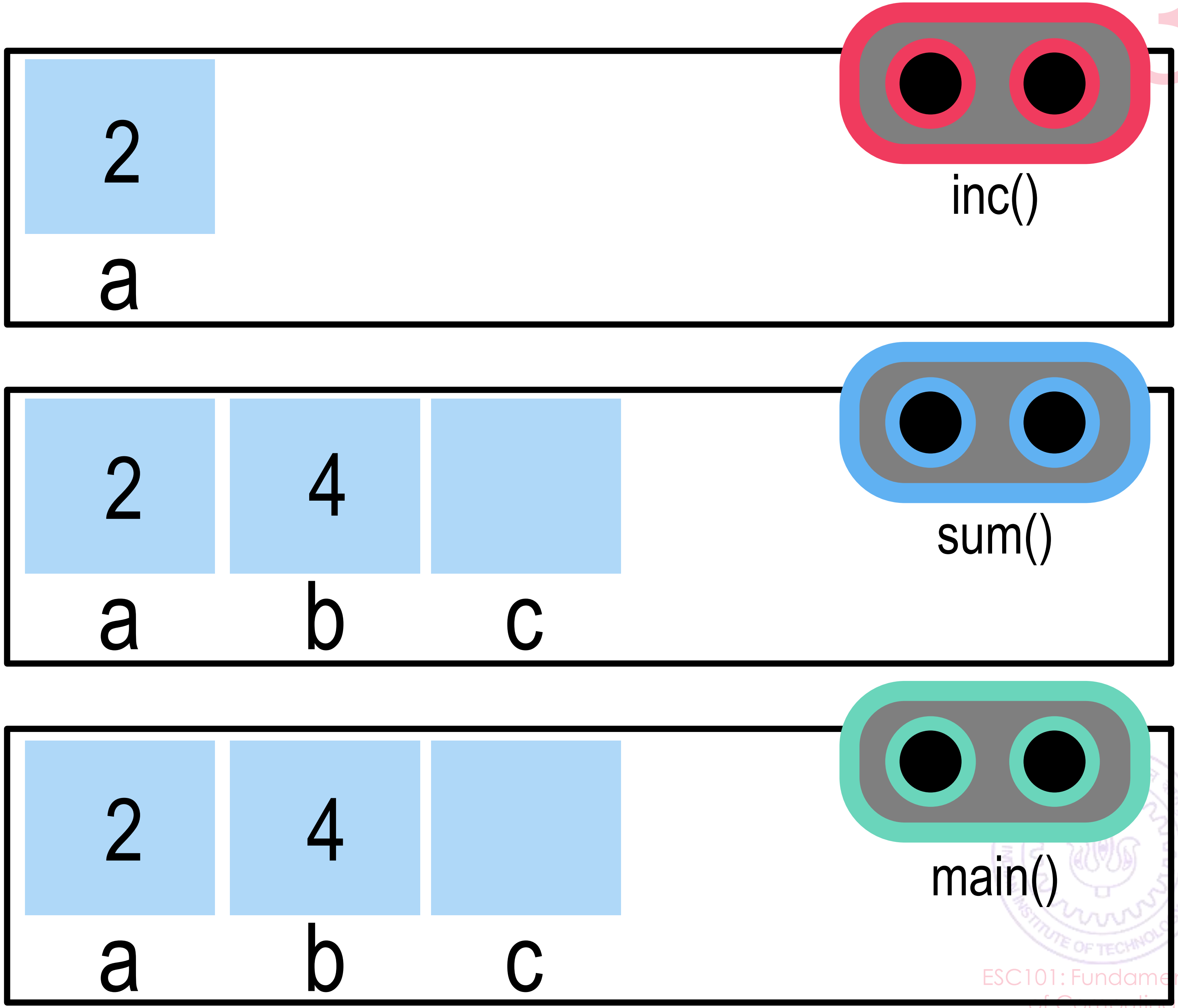
Clones!

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + b;
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



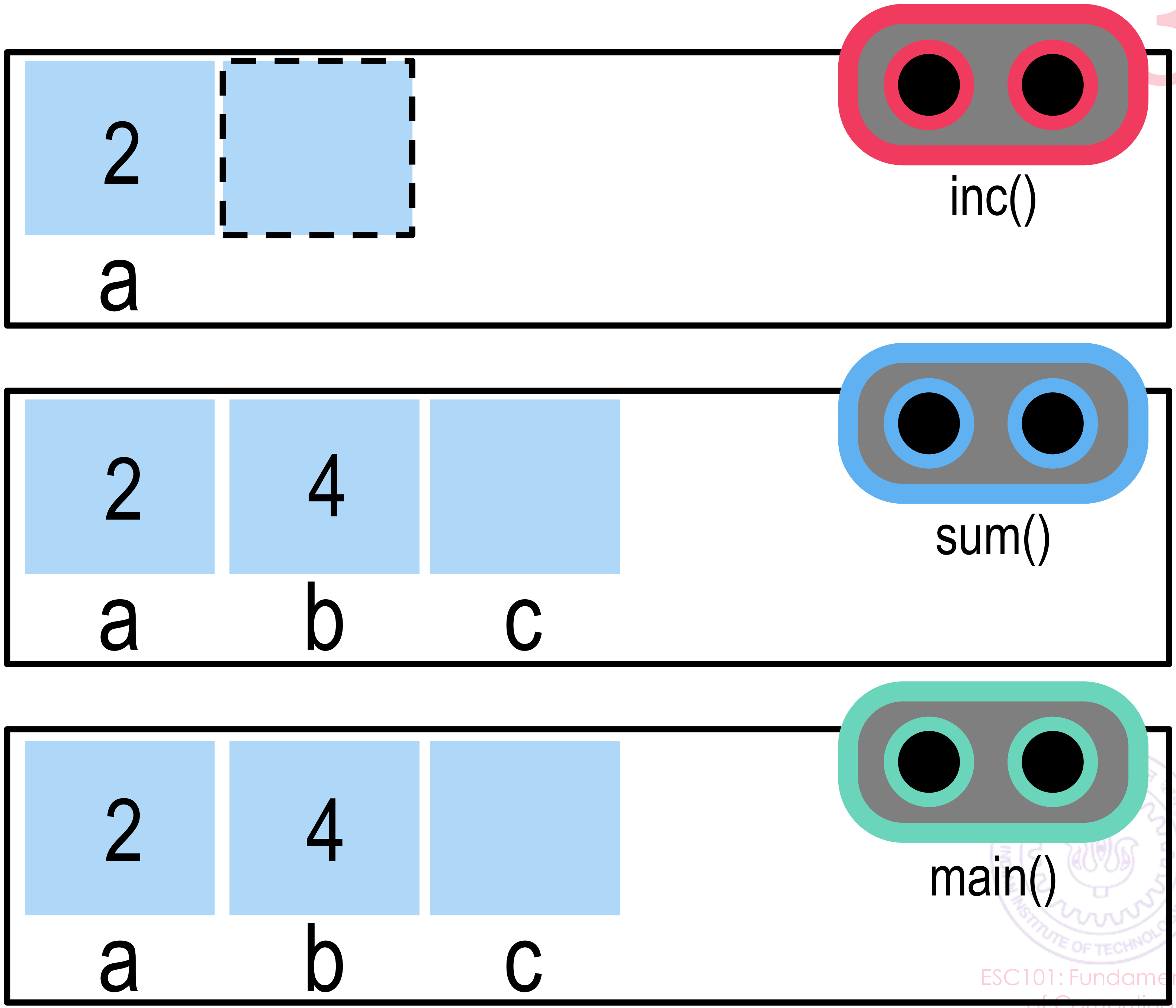
Clones!

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + b;
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



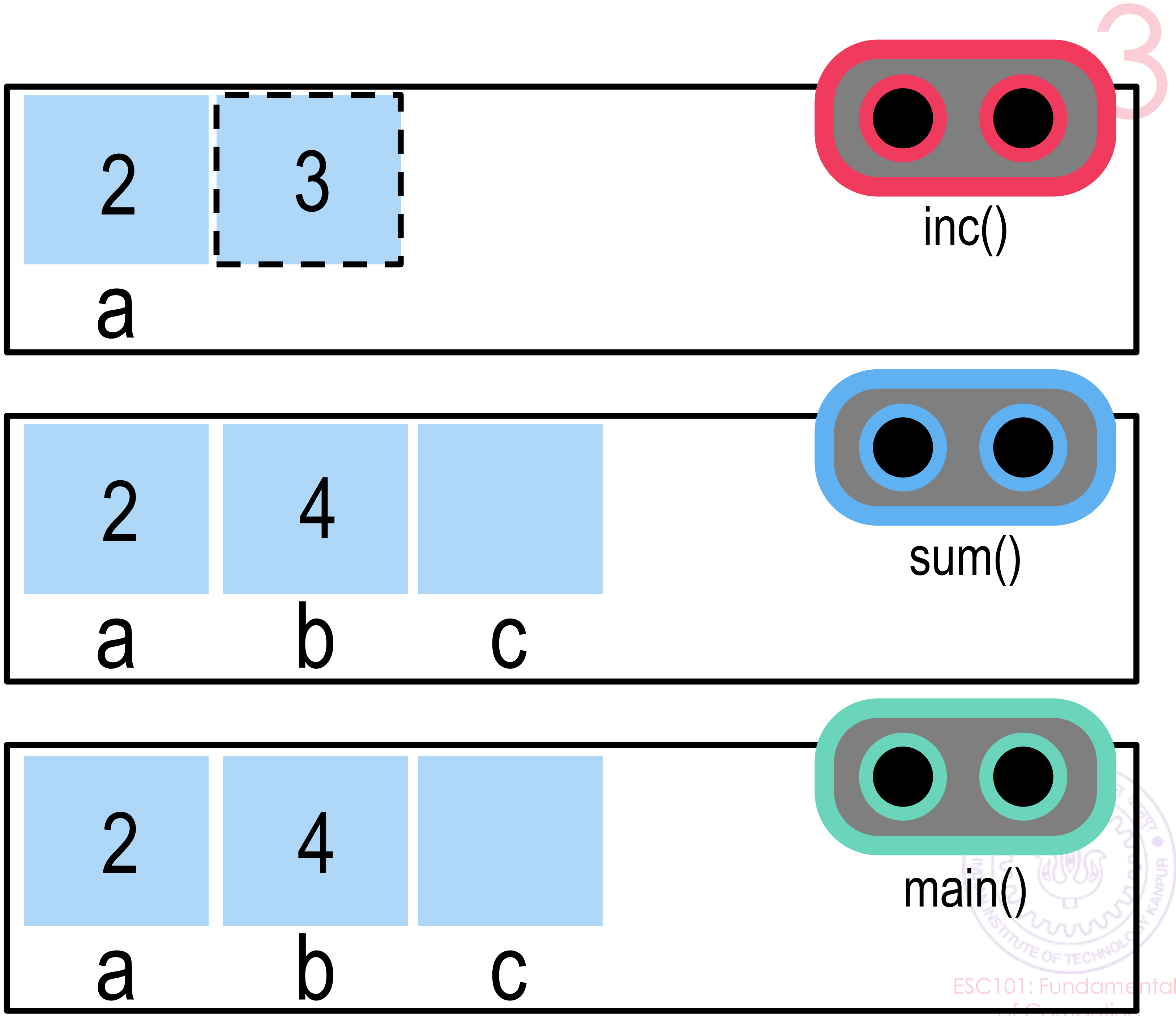
Clones!

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + b;
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



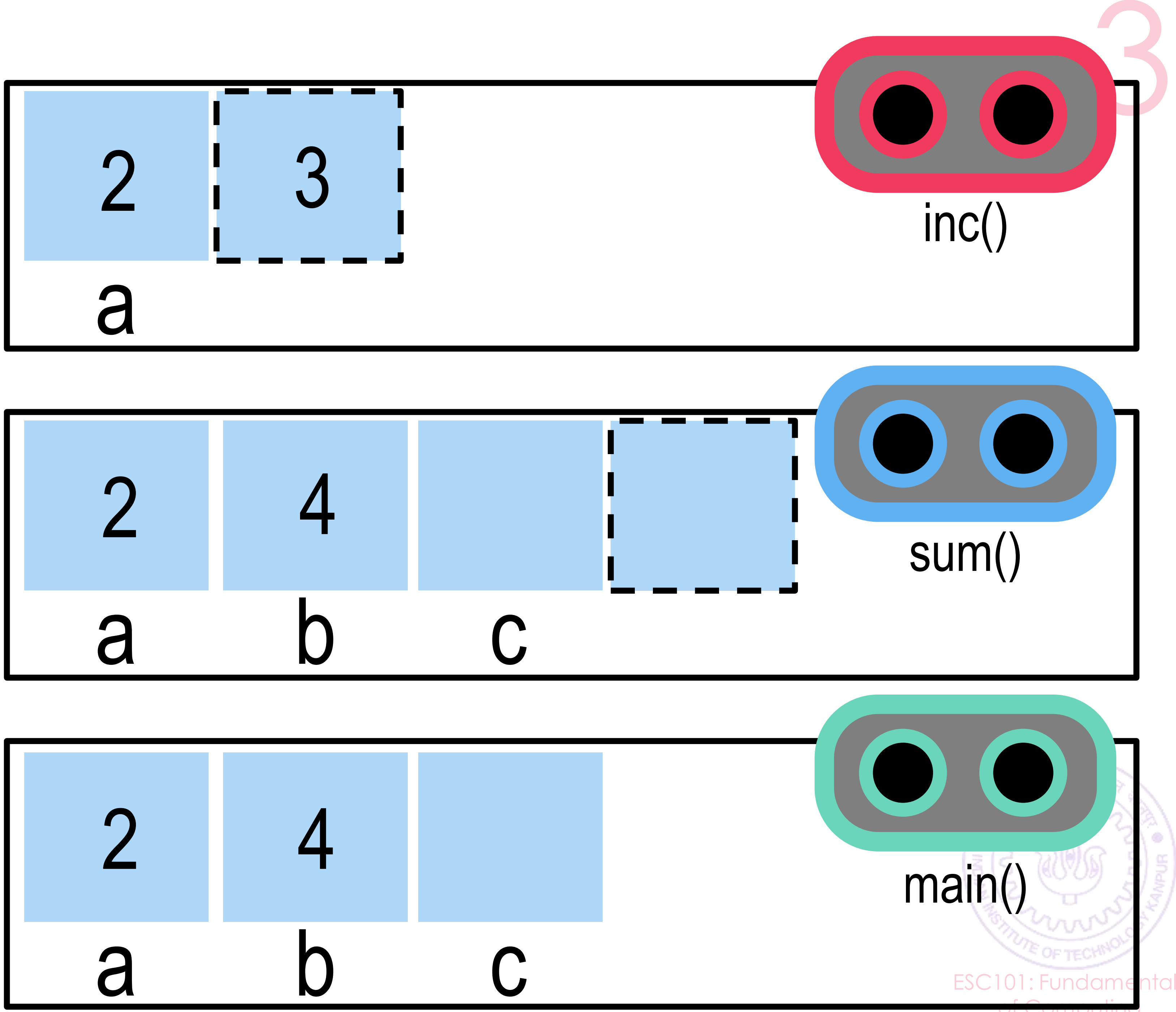
Clones!

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + b;
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



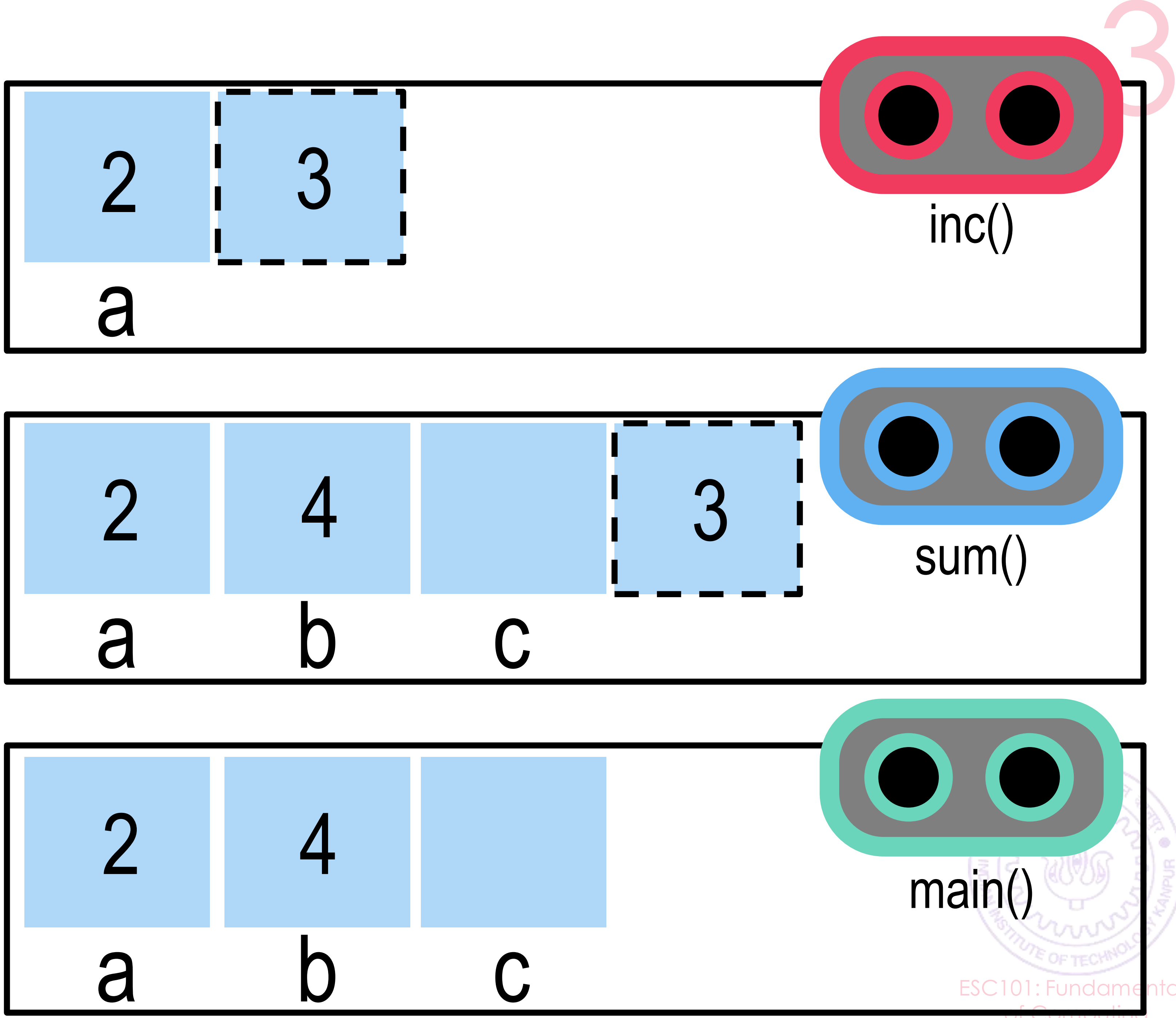
Clones!

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + b;
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



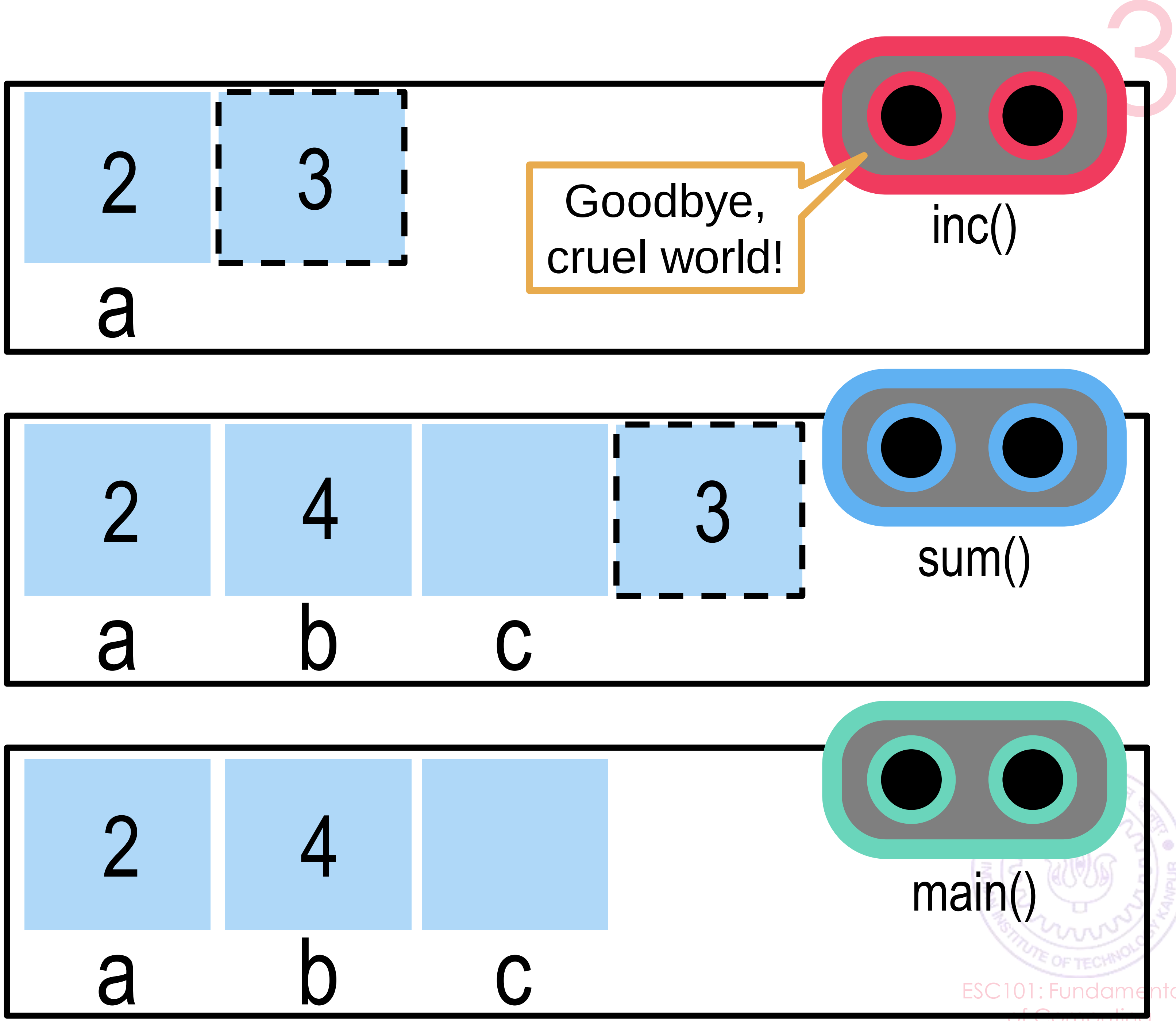
Clones!

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + b;
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



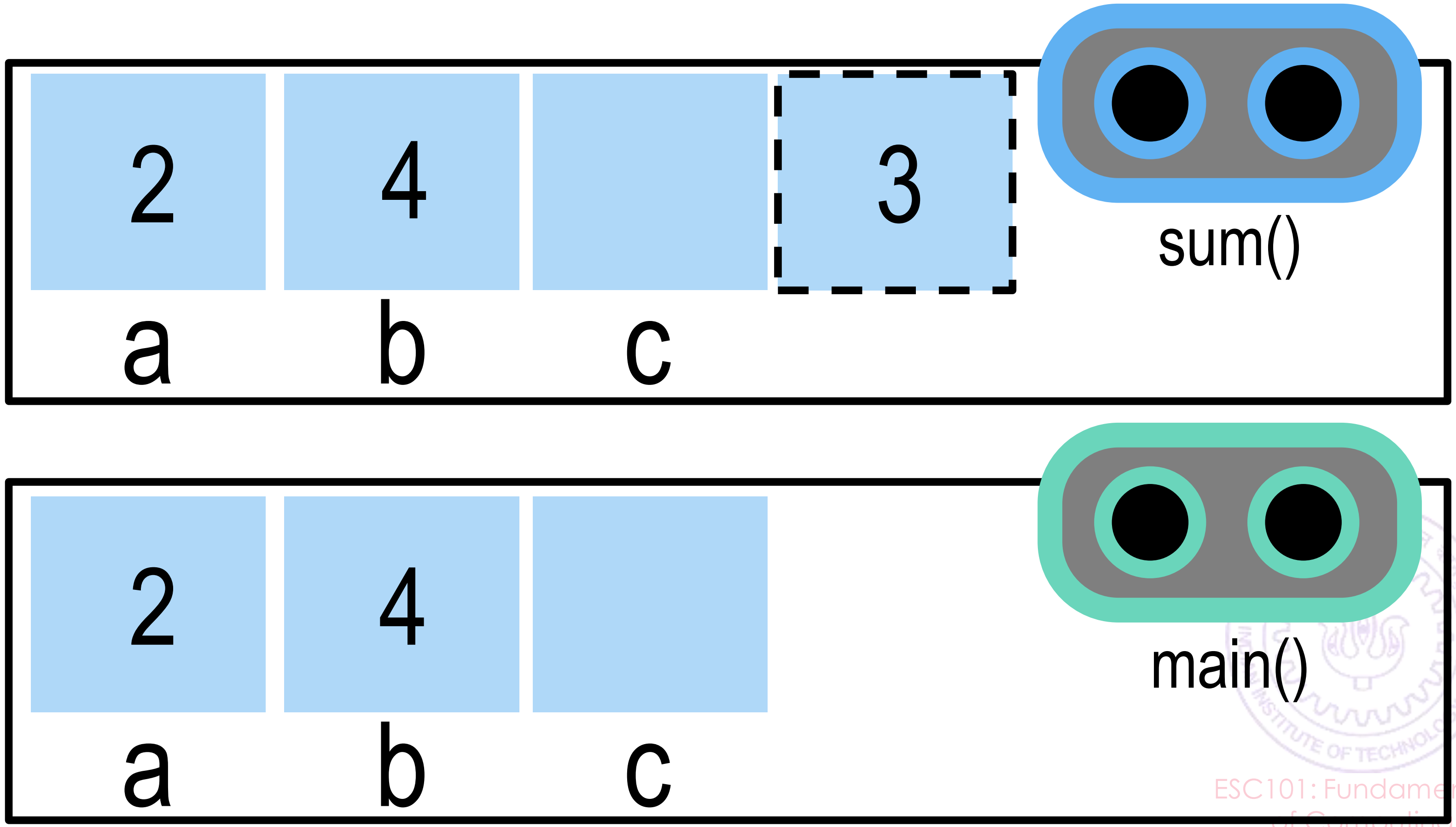
Clones!

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + b;
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



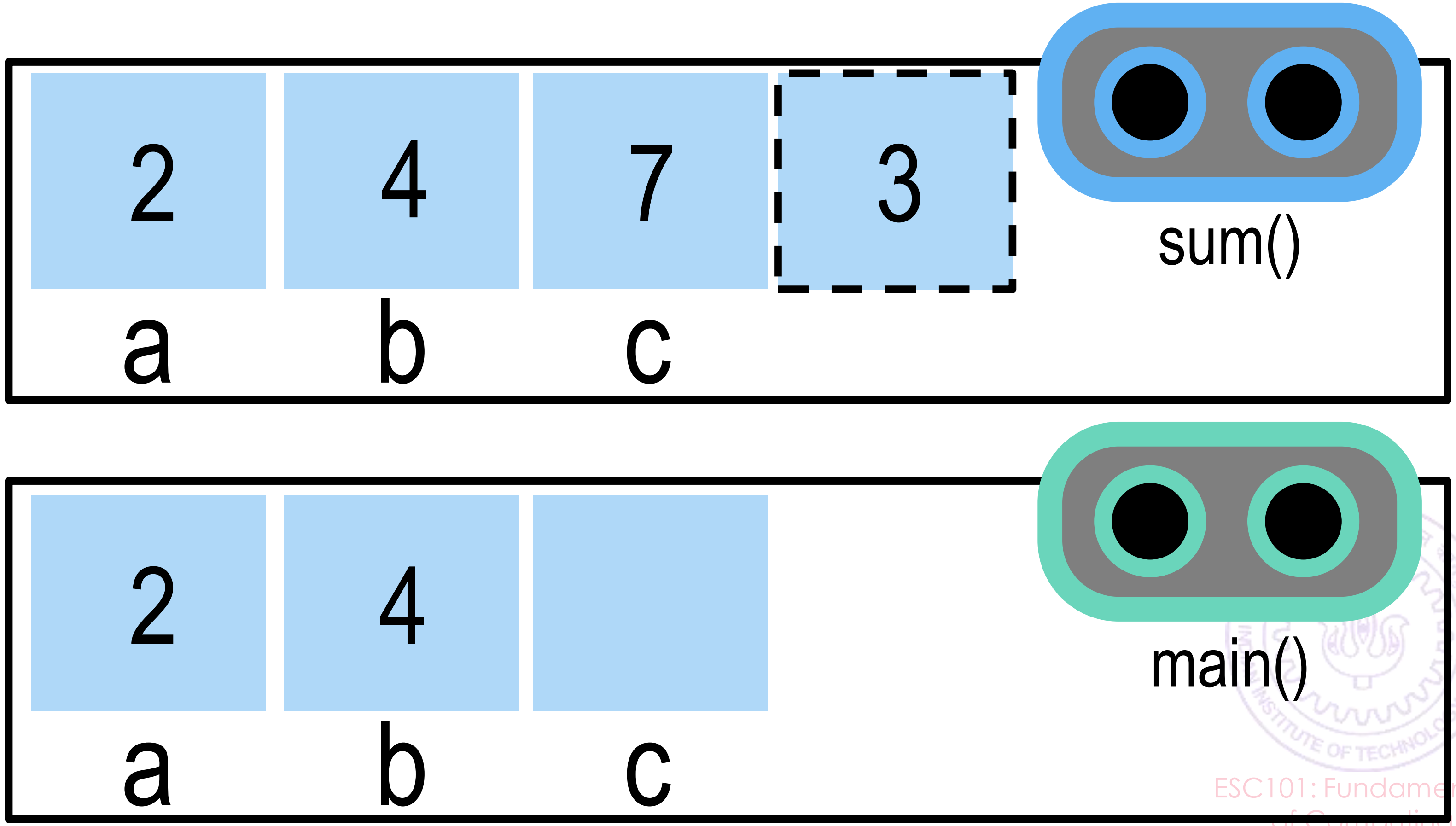
Clones!

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + b;
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



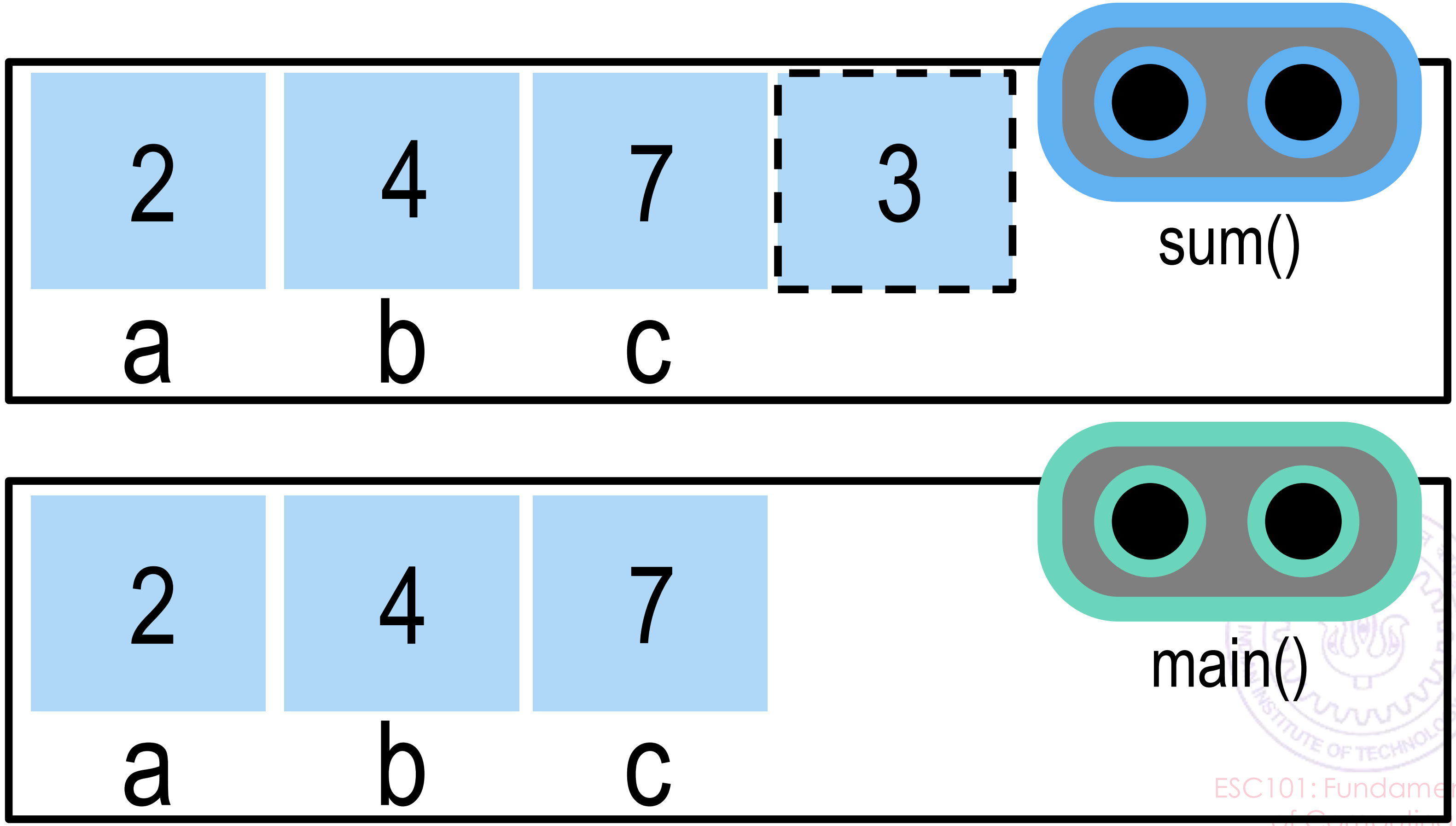
Clones!

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + b;
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



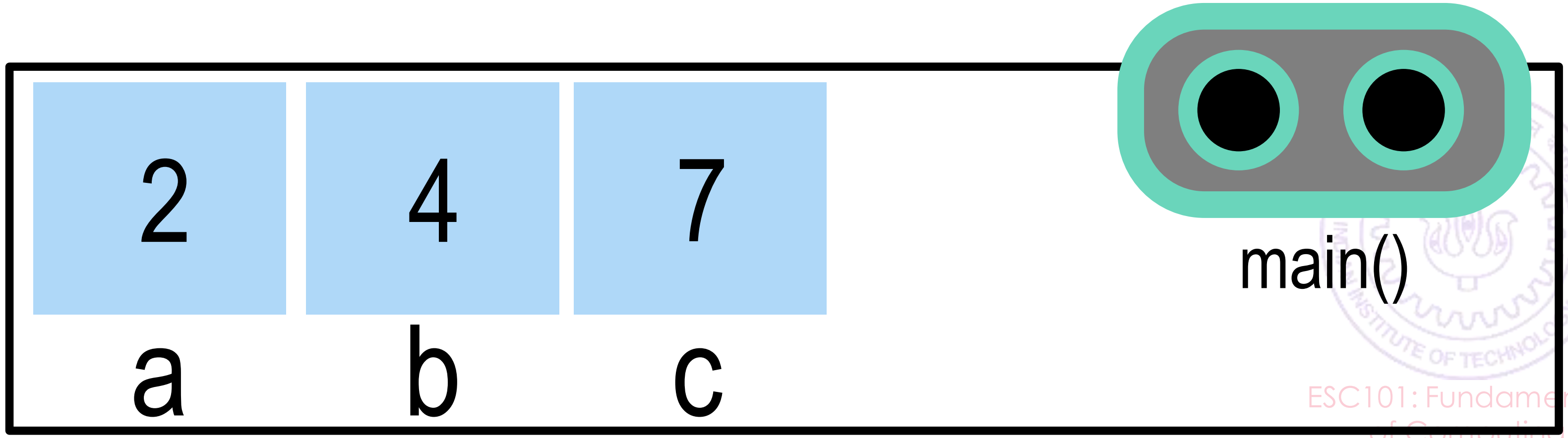
Clones!

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + b;
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



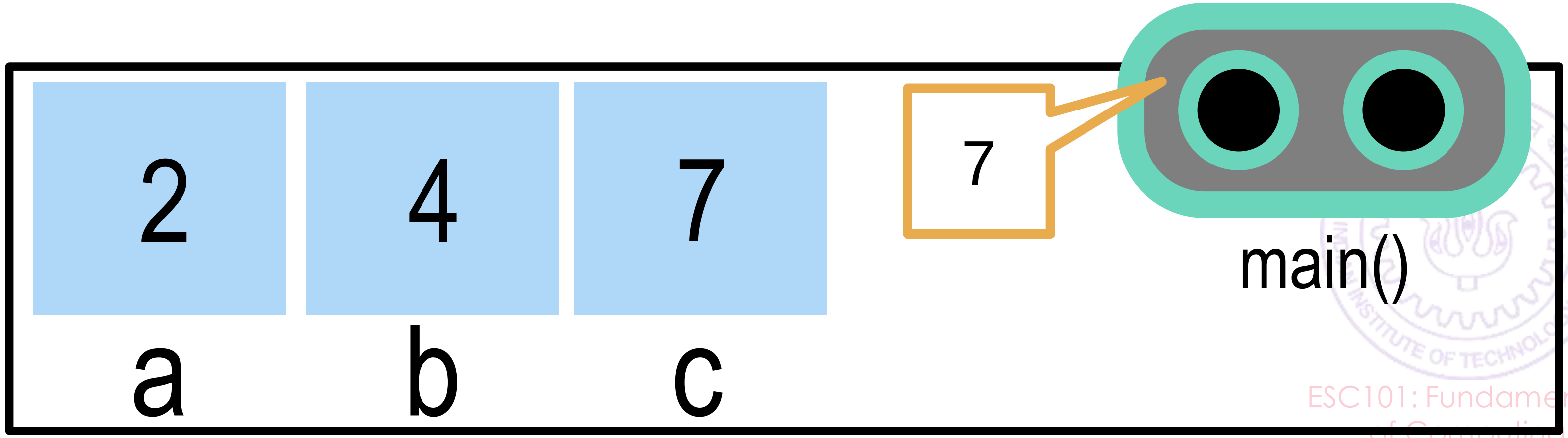
Clones!

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + b;
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



Clones!

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + b;
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



Clones!

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + b;
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```

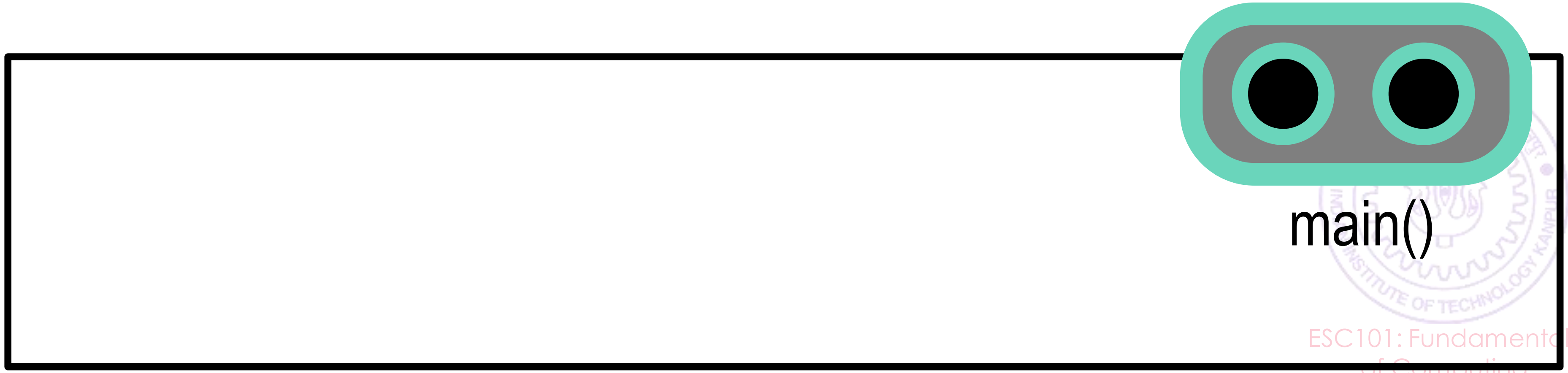
Same function called repeatedly

Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```

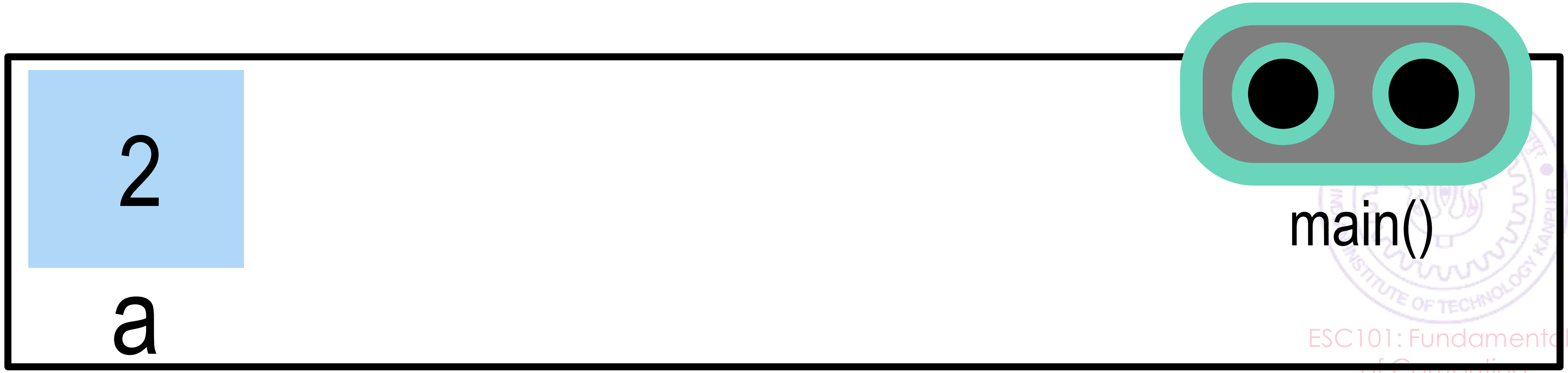
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



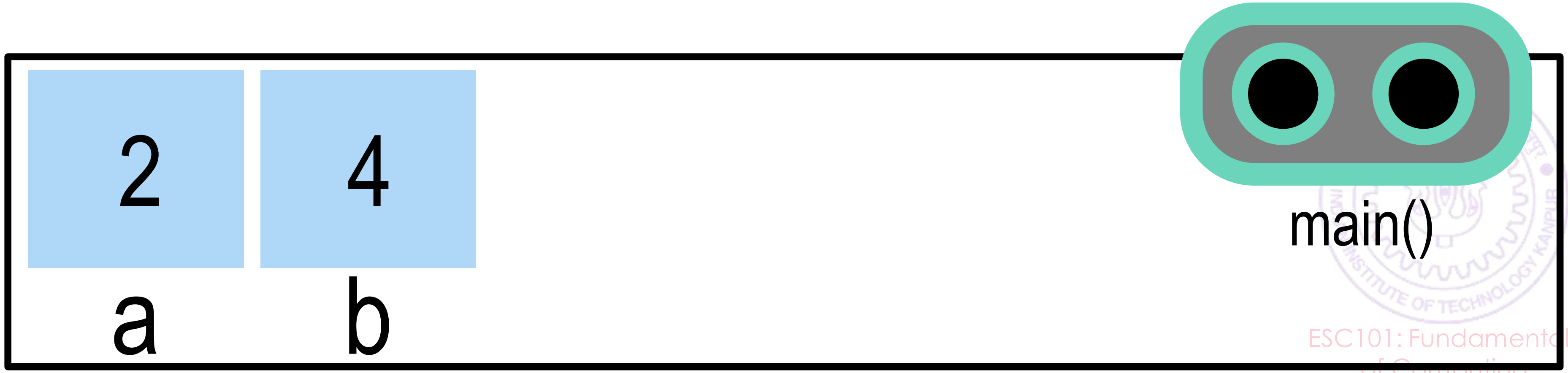
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



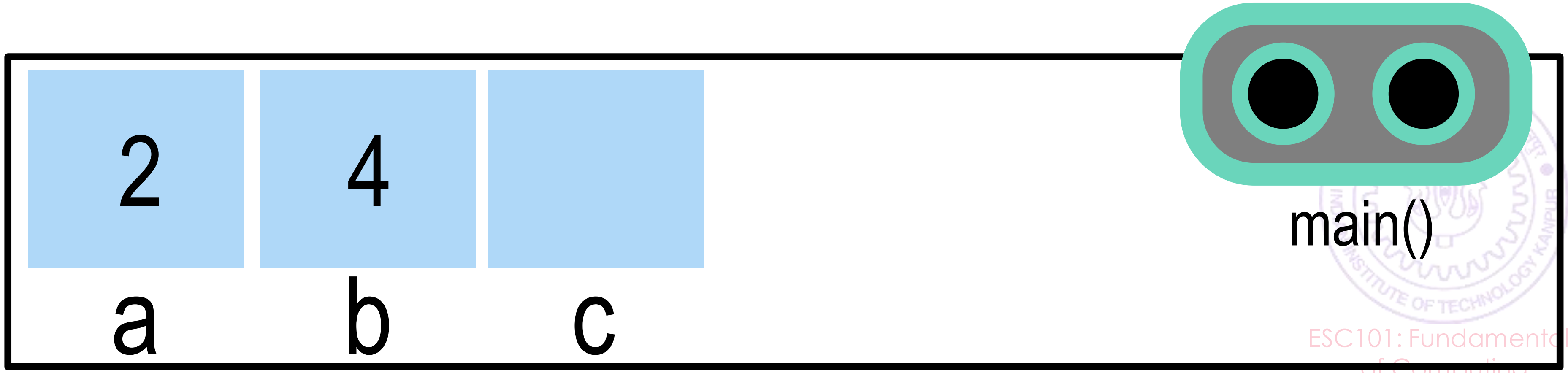
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



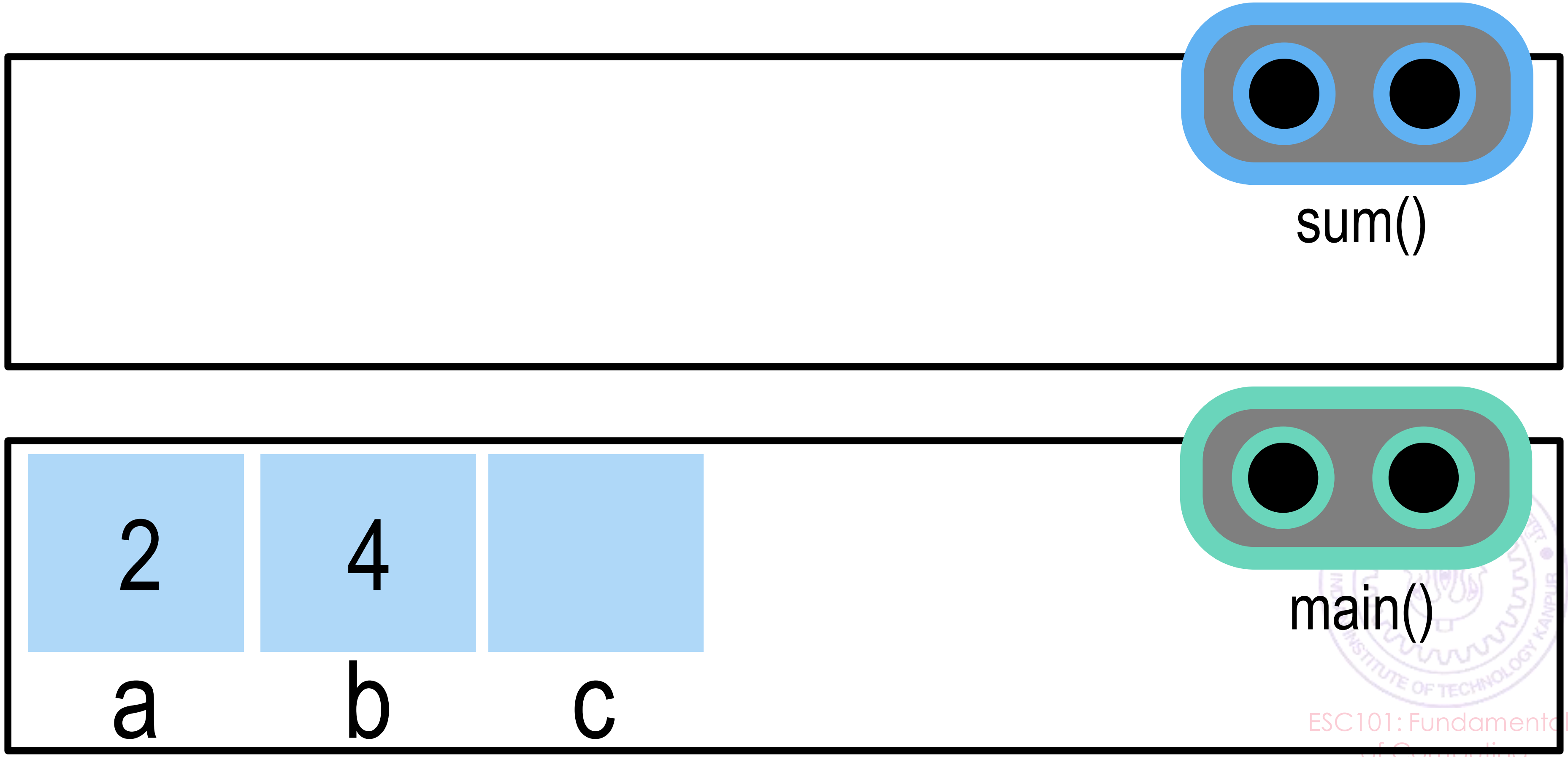
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



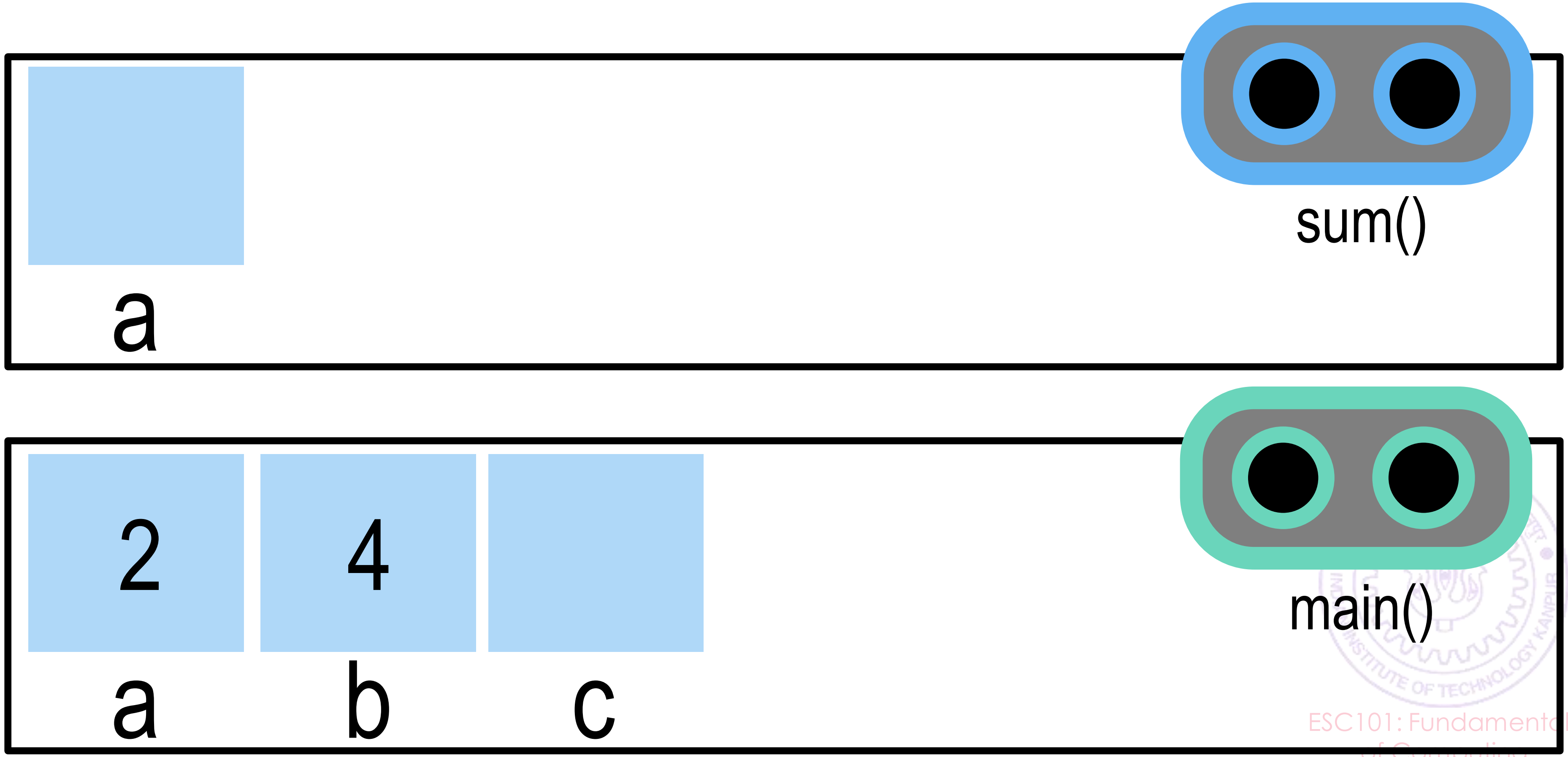
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



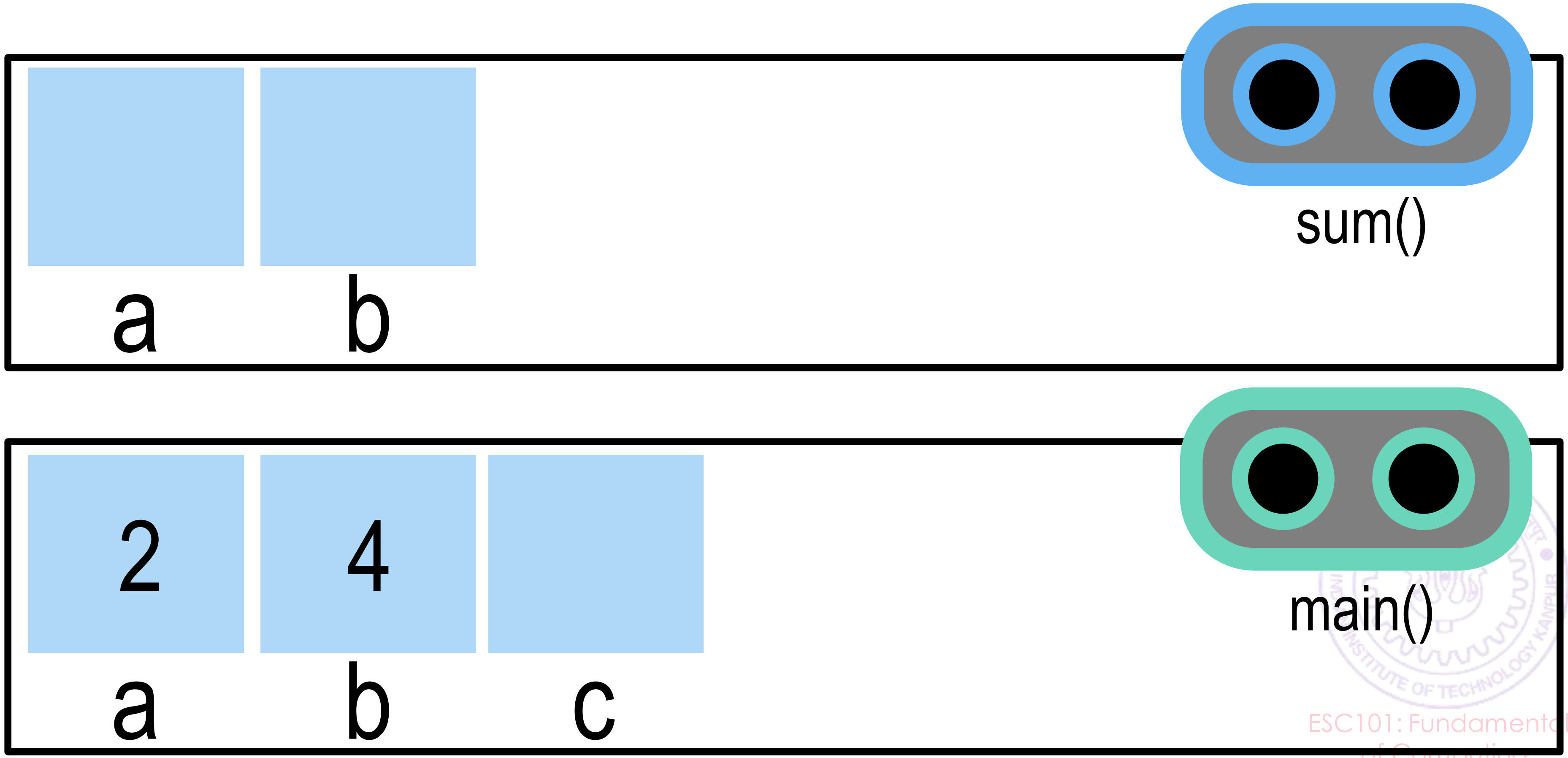
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



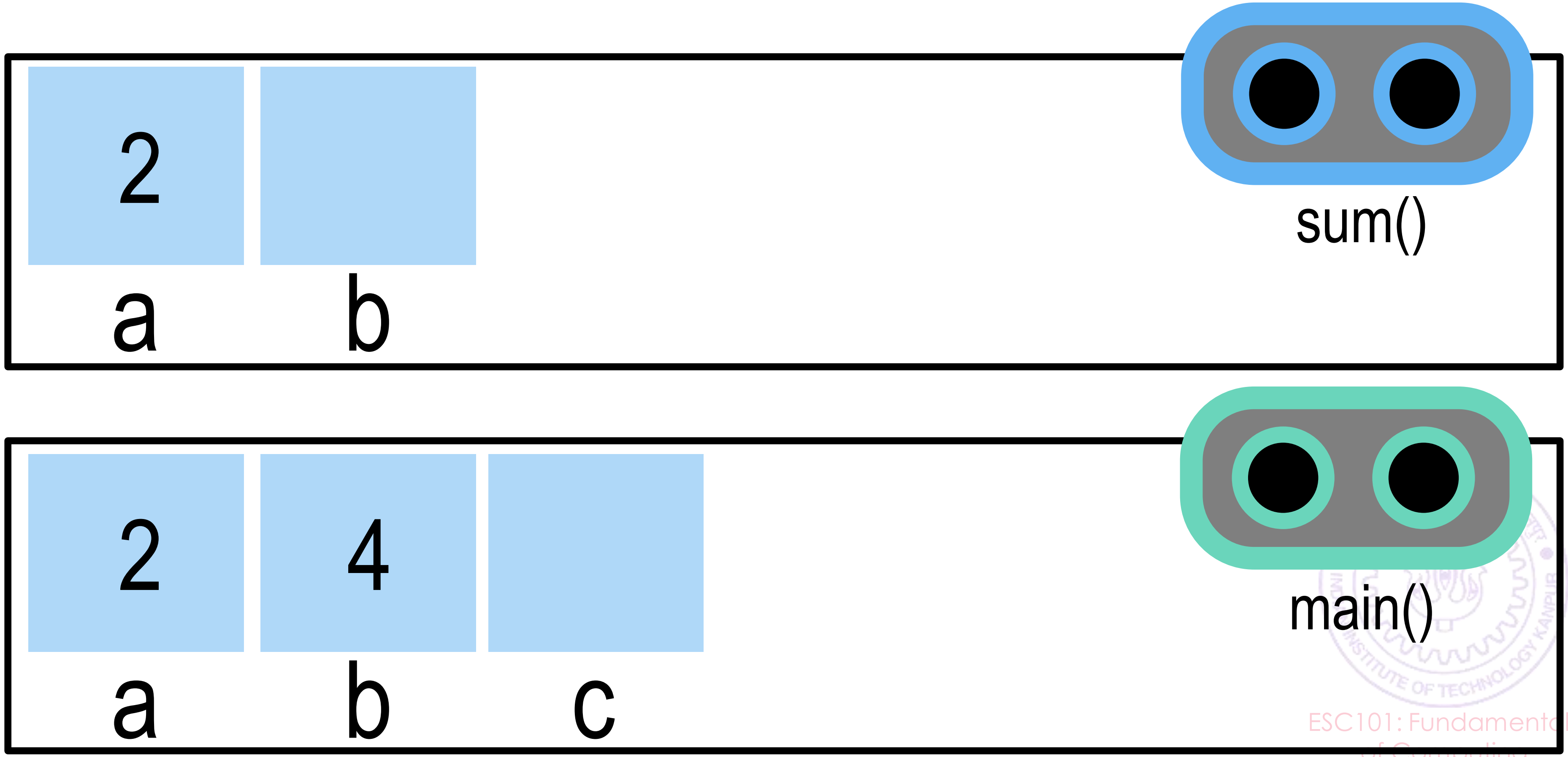
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



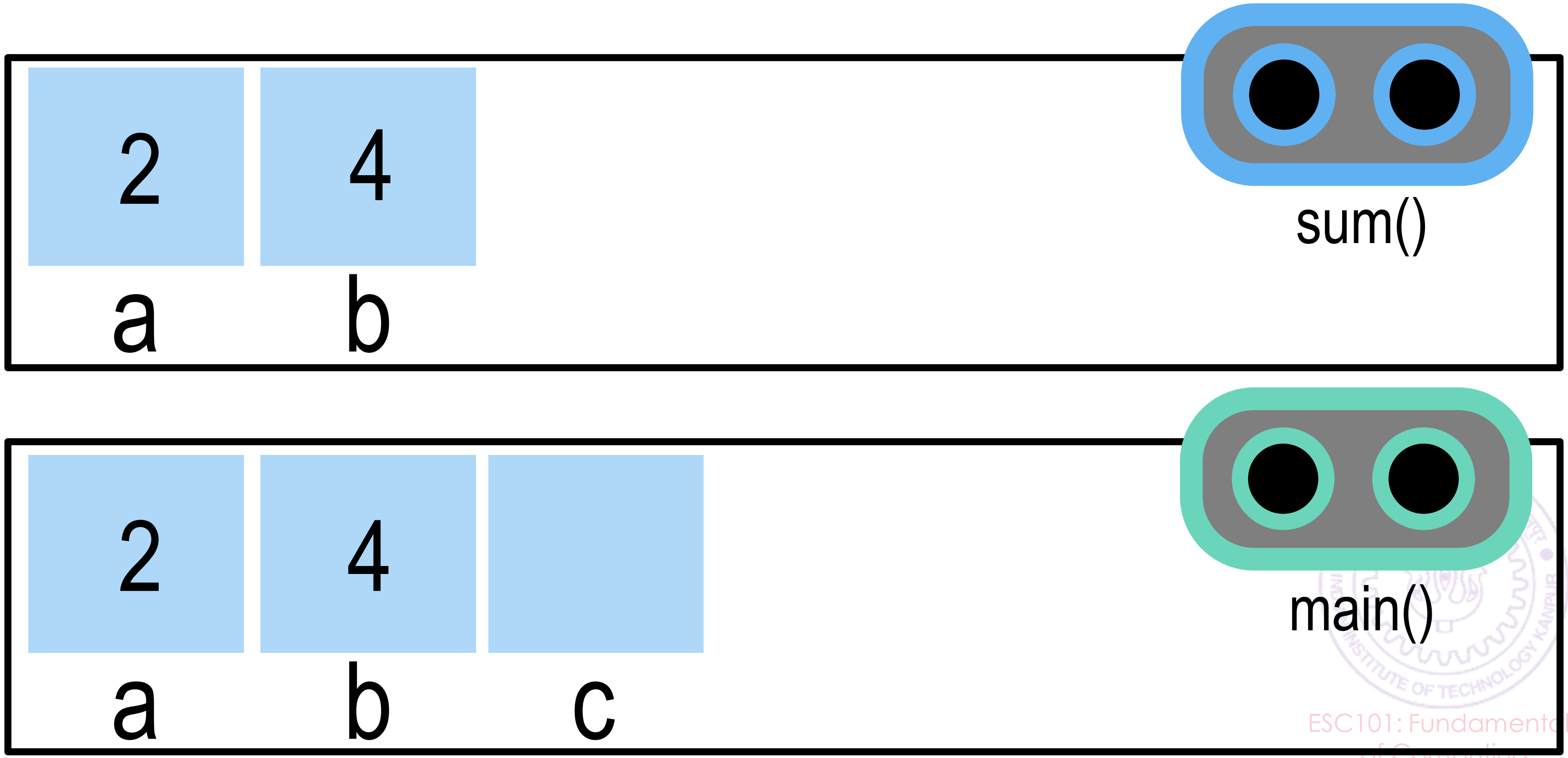
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



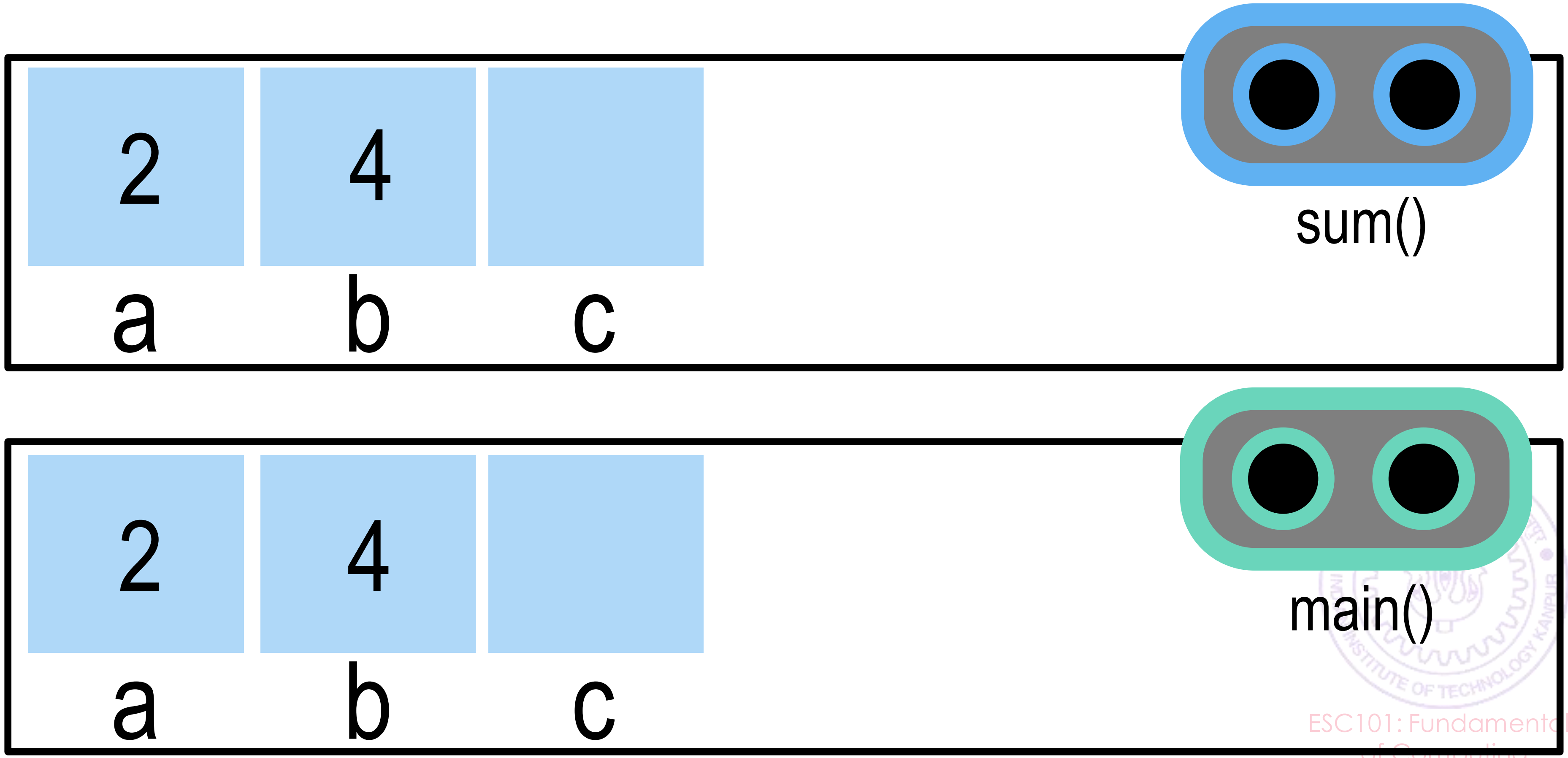
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



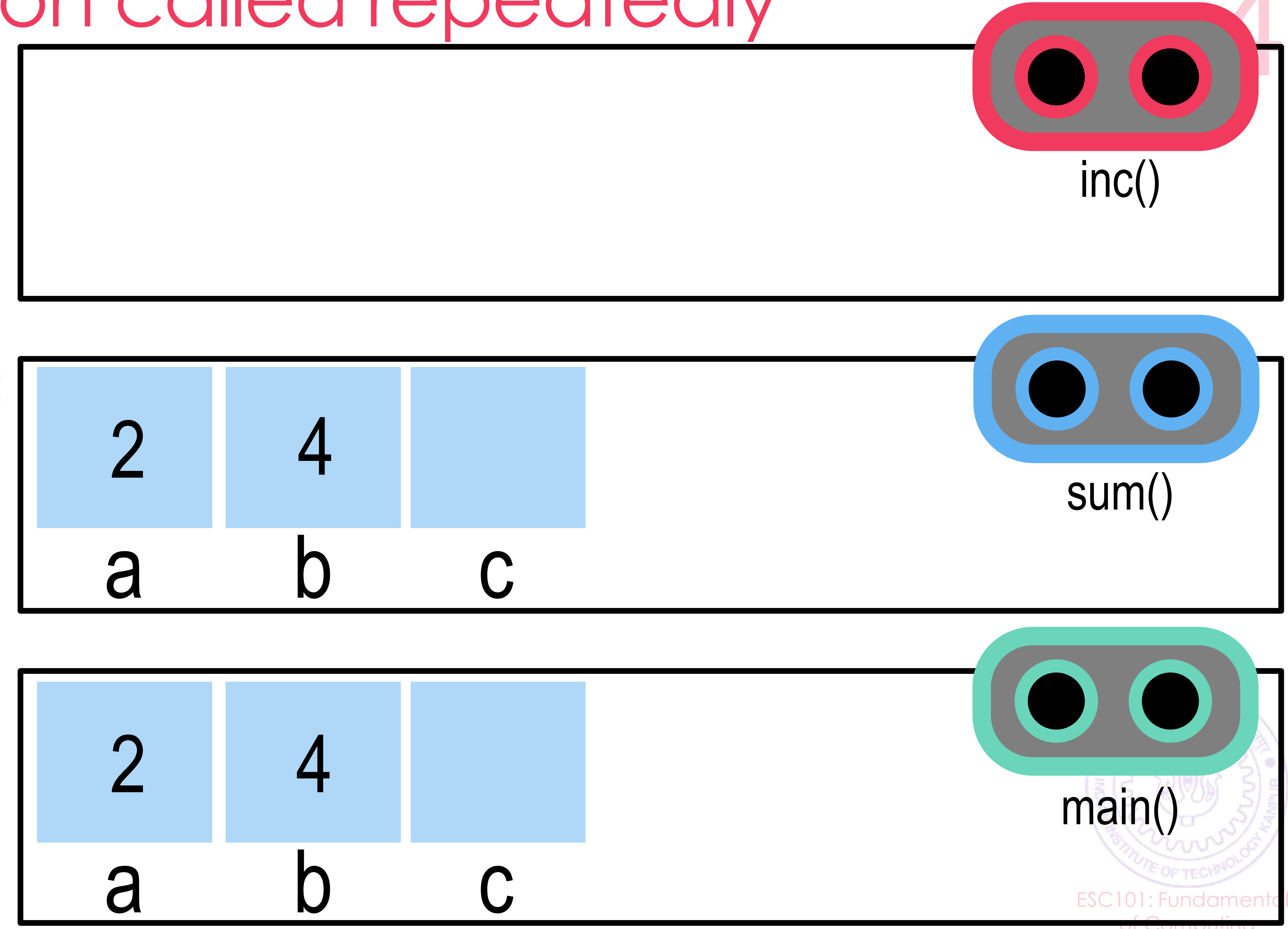
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



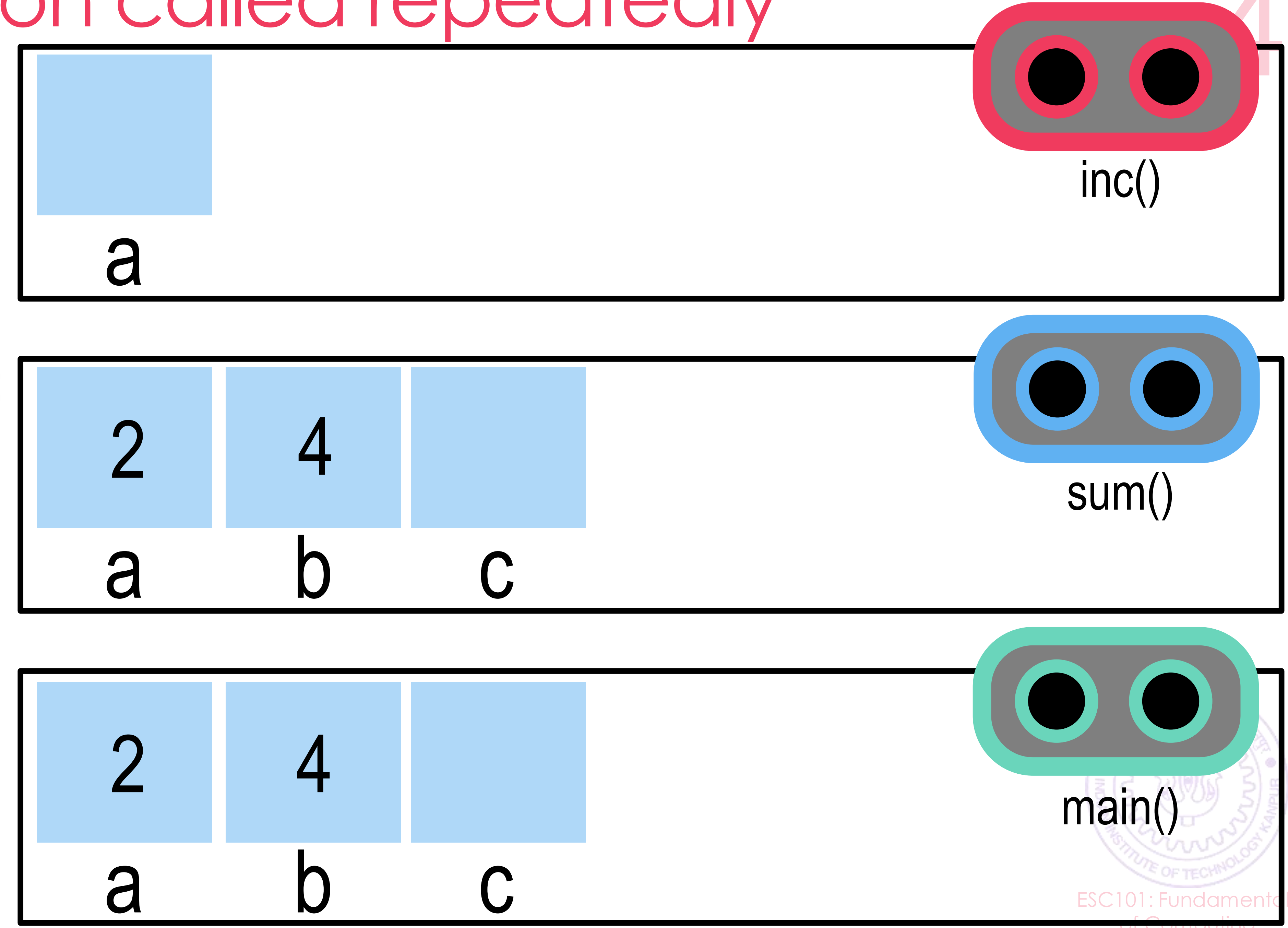
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



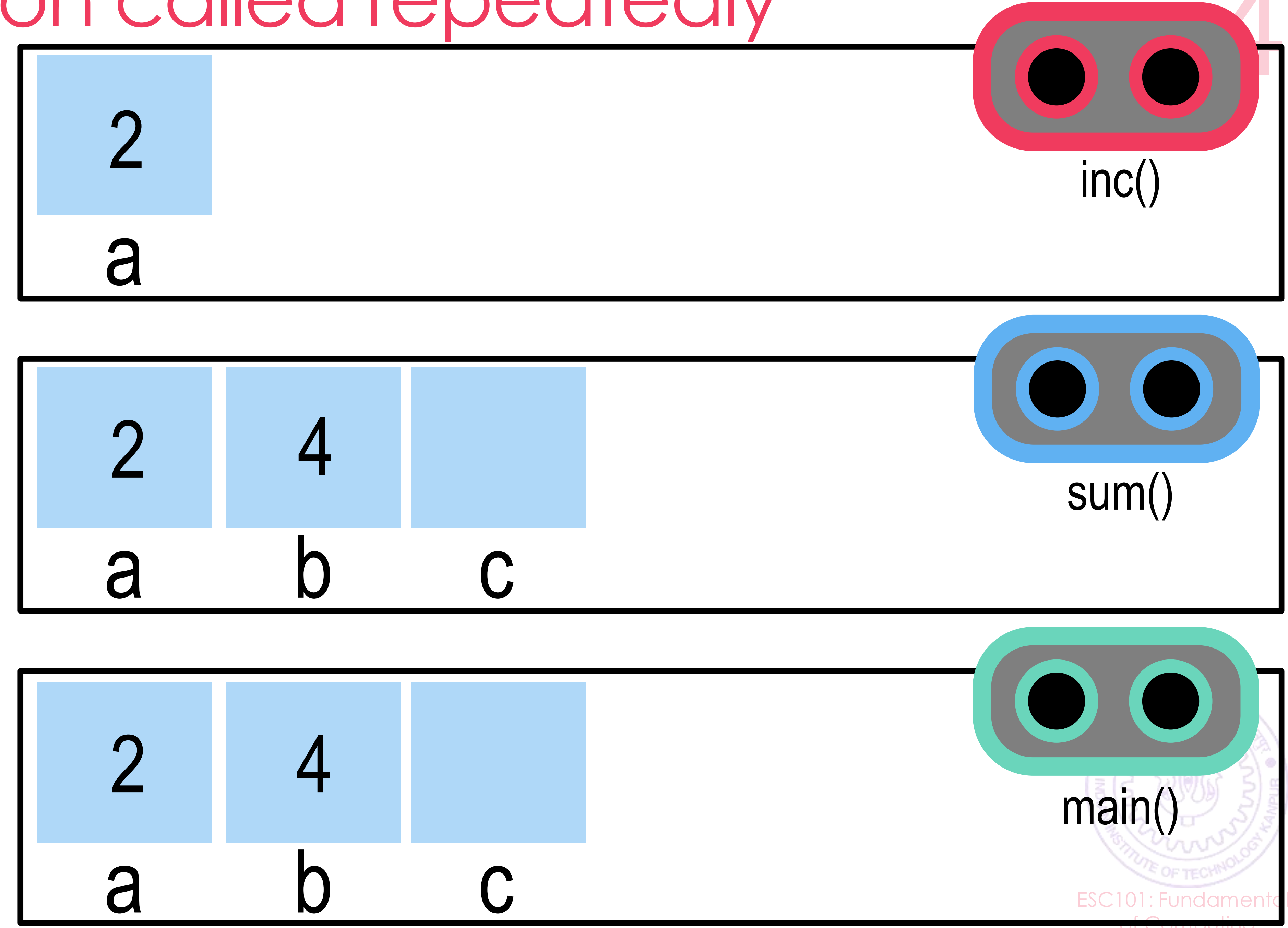
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



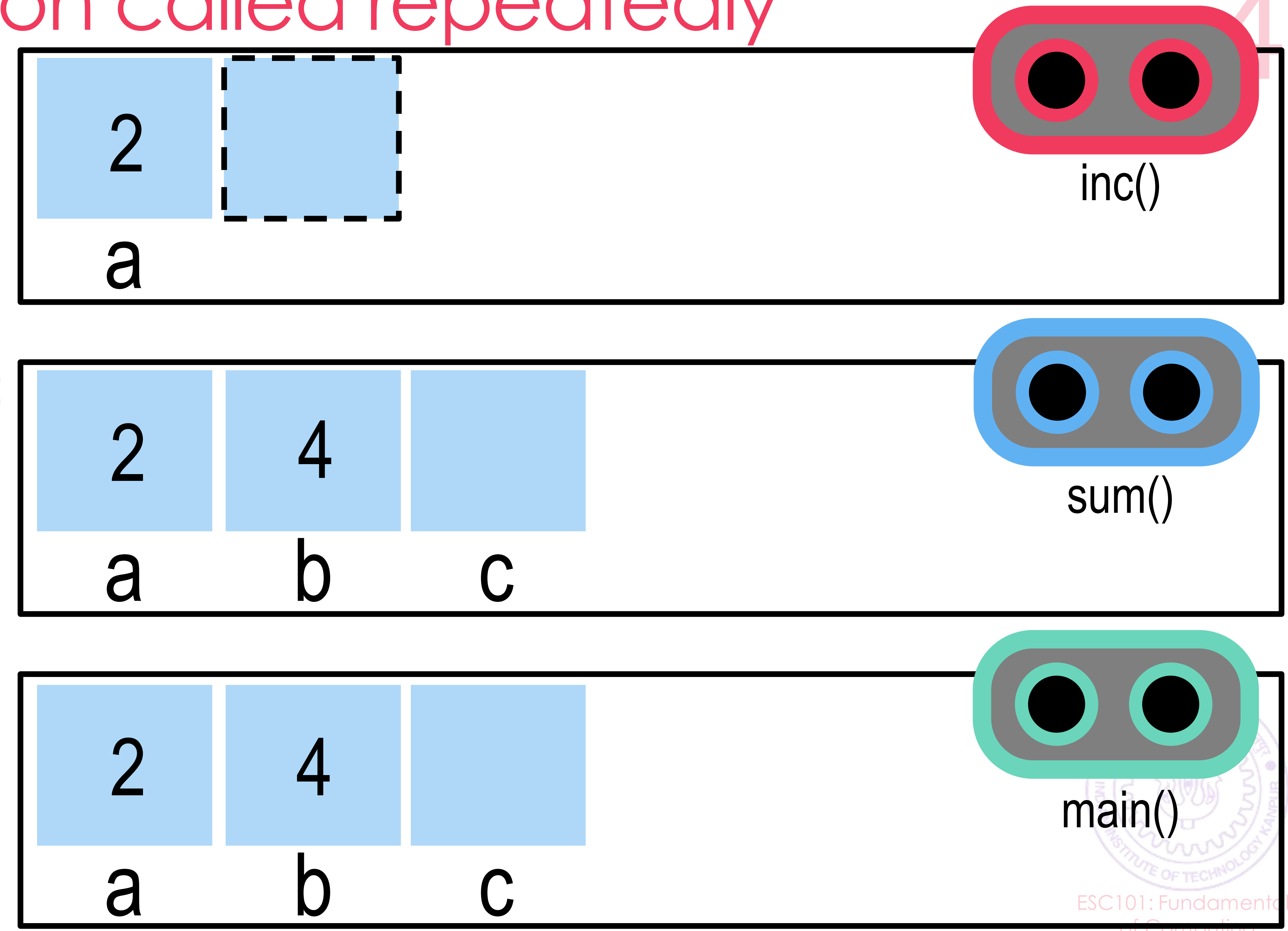
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



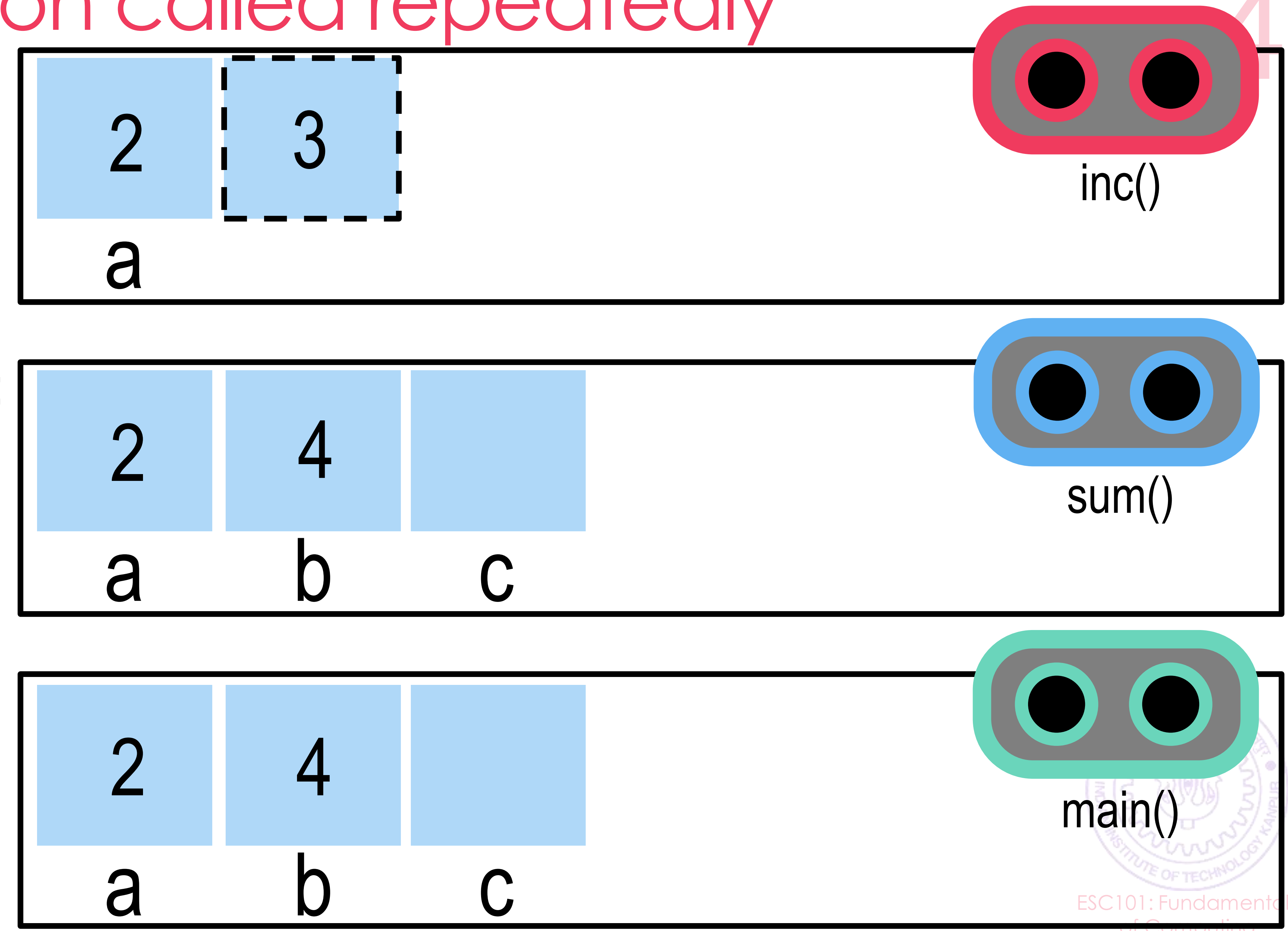
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



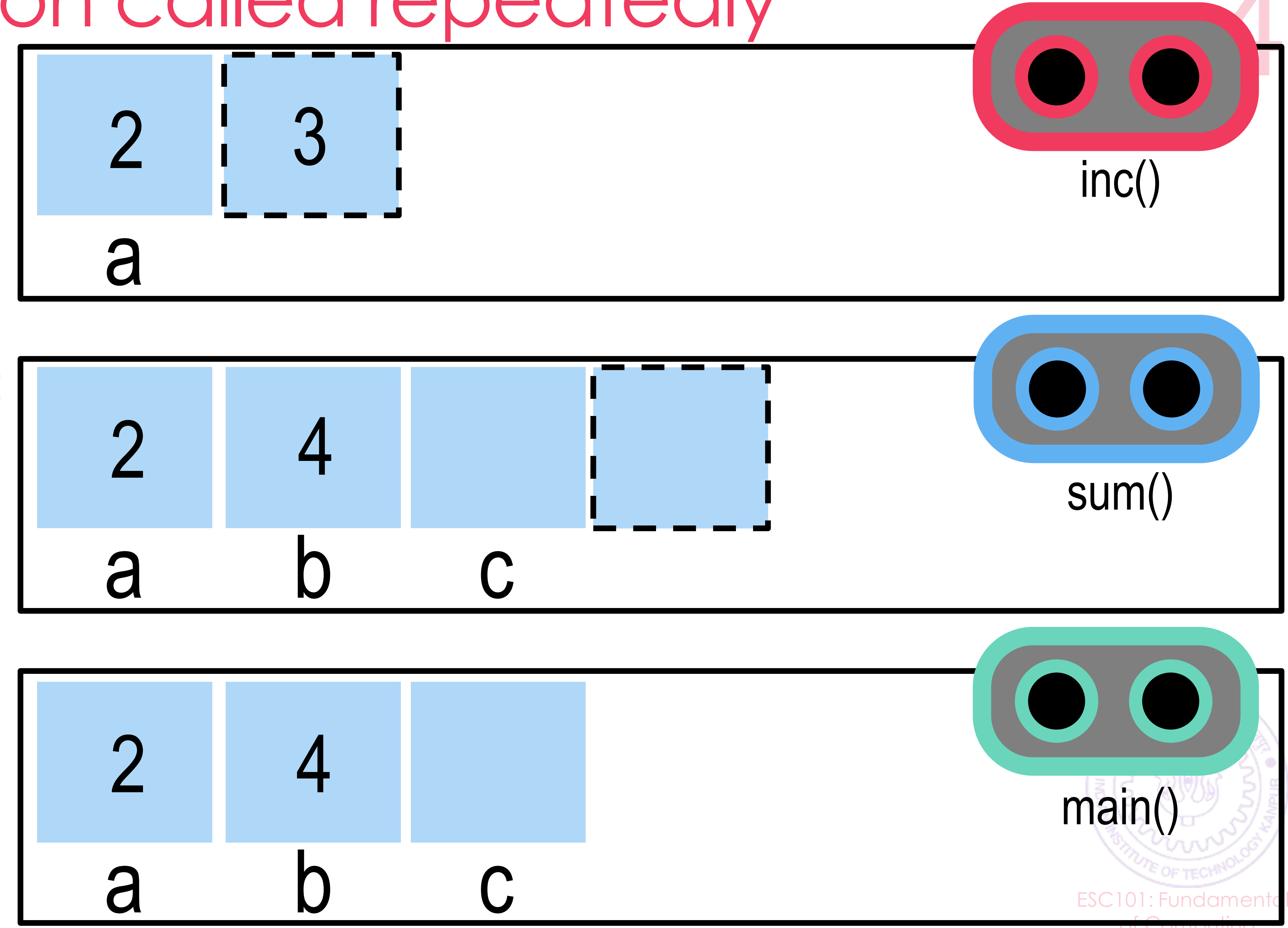
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



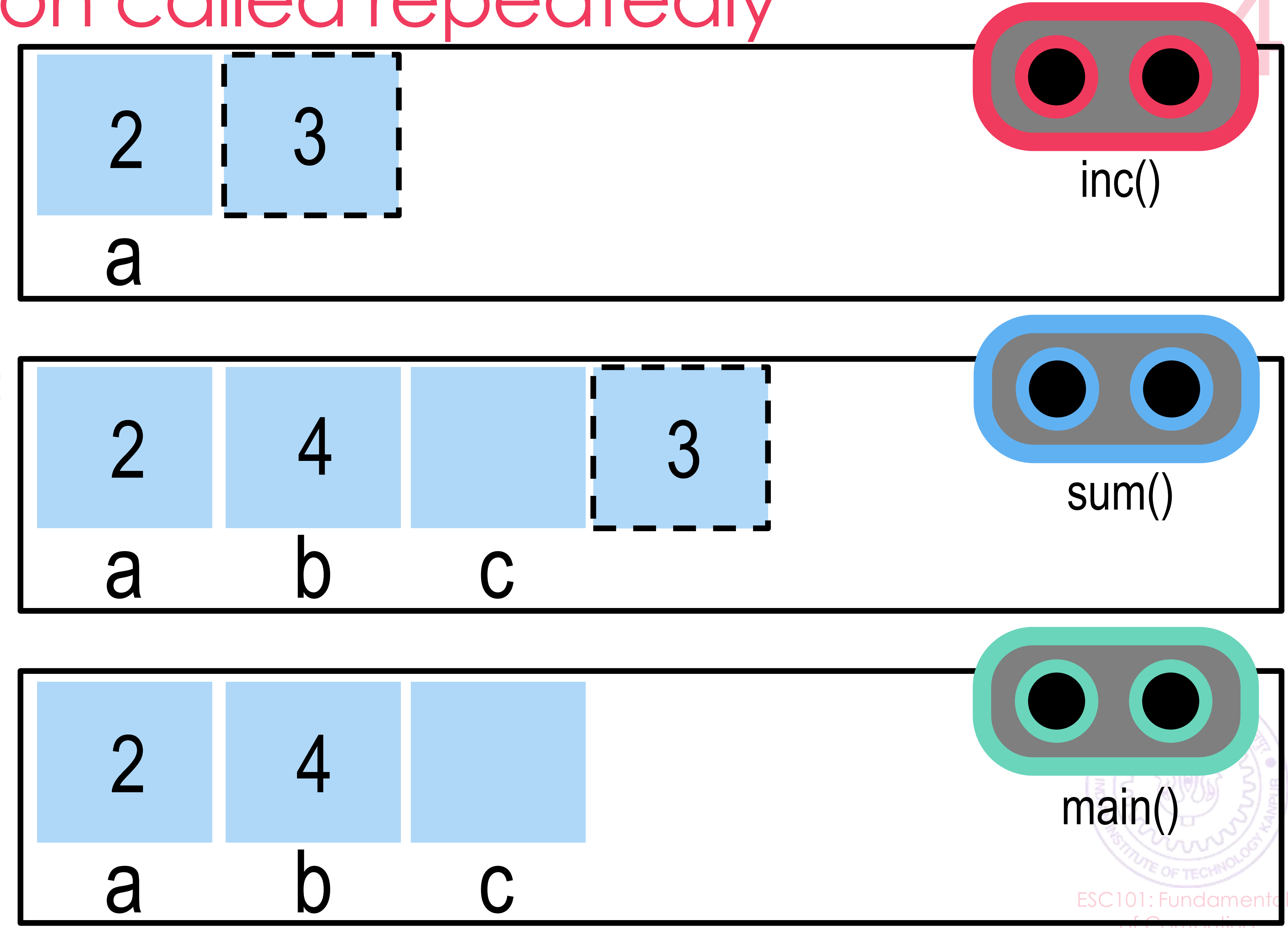
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



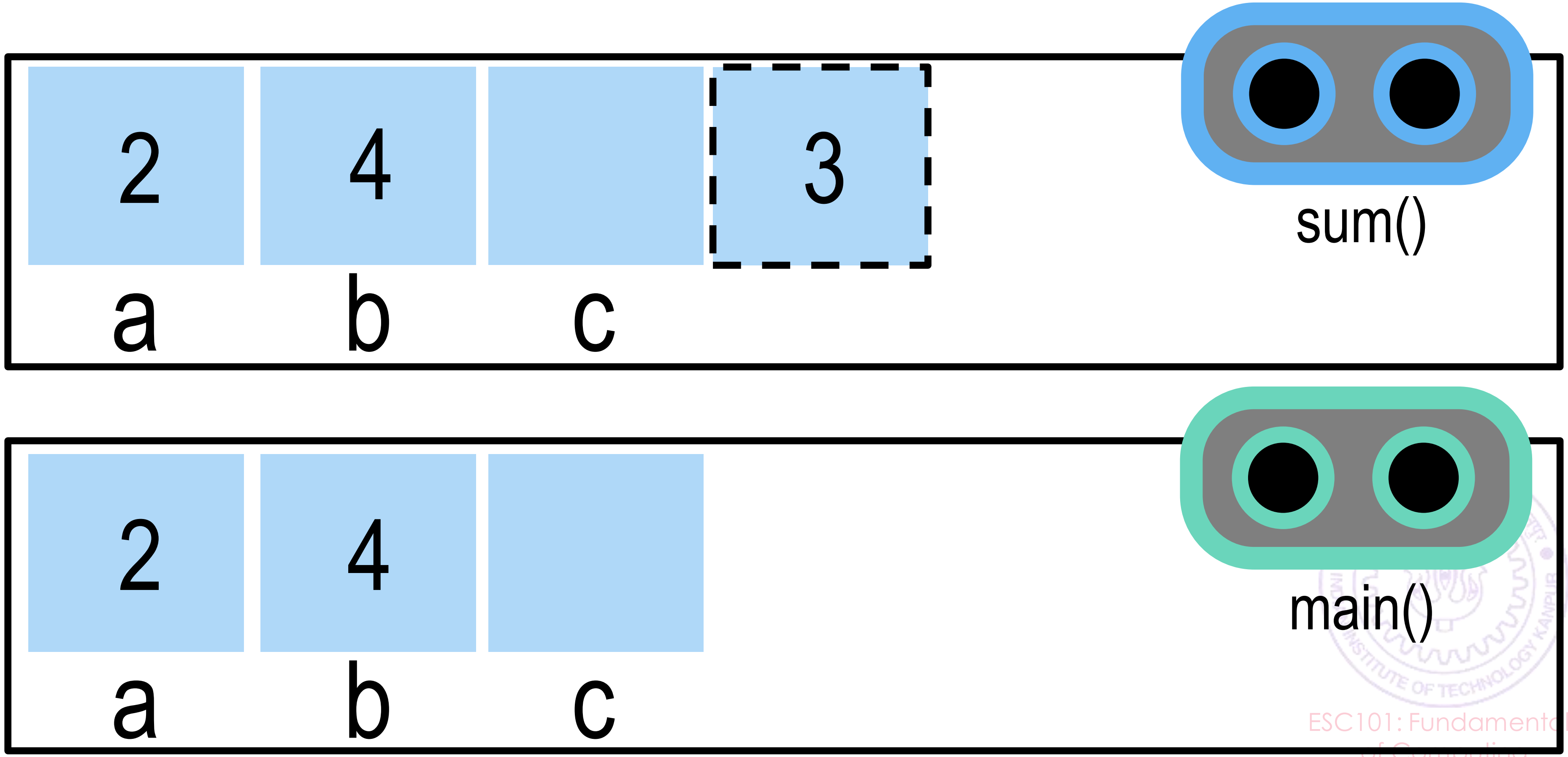
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



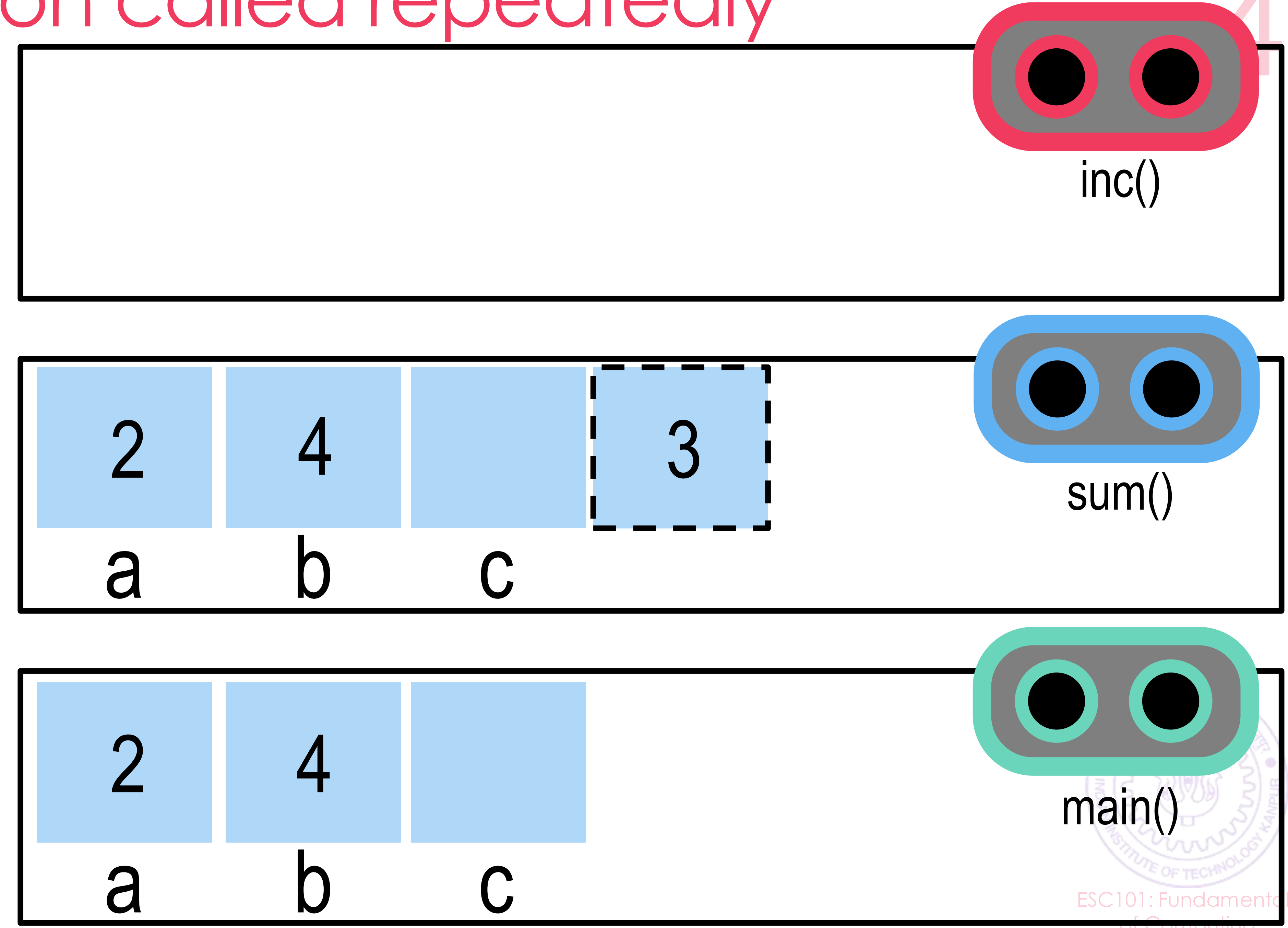
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



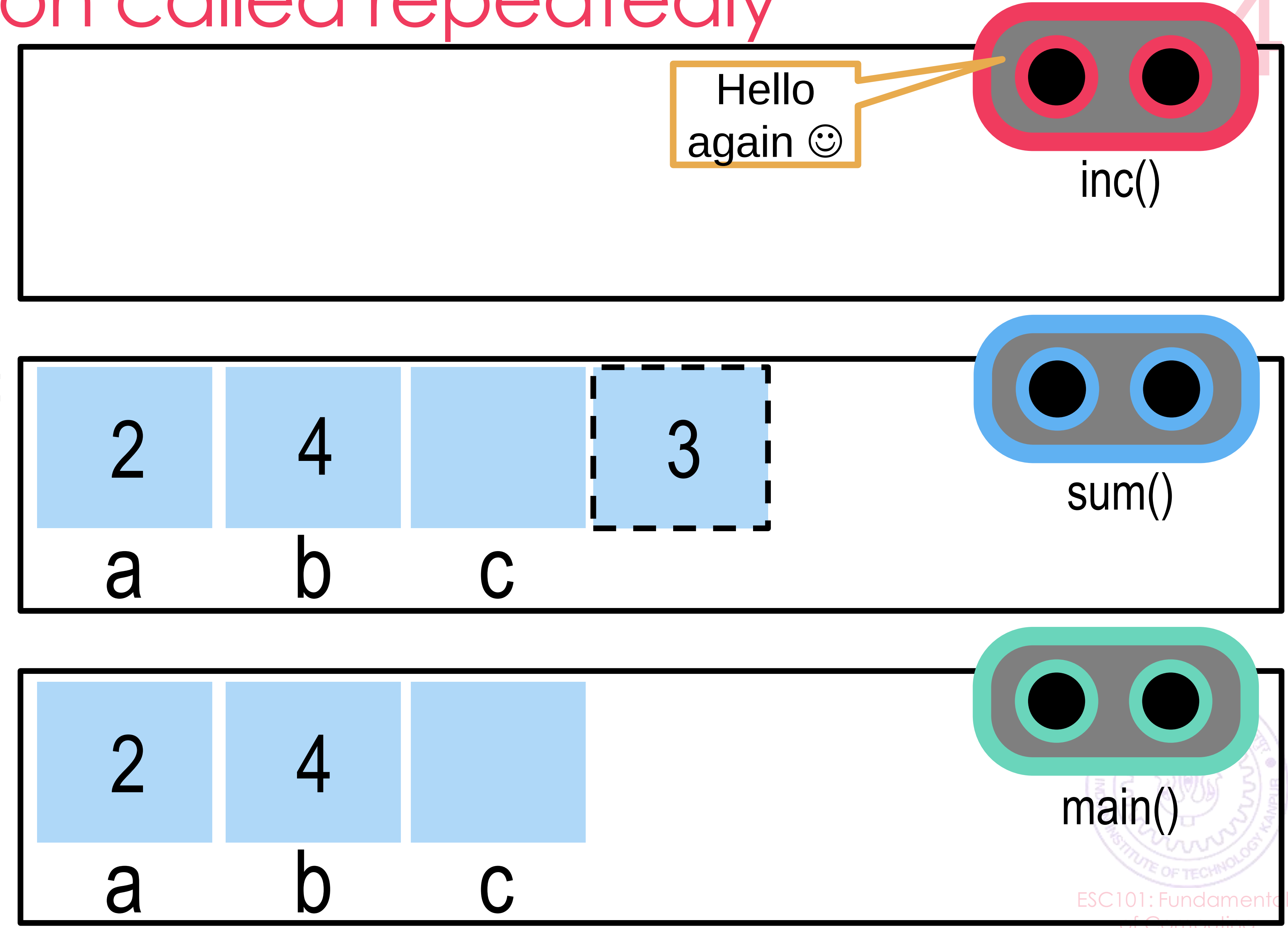
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



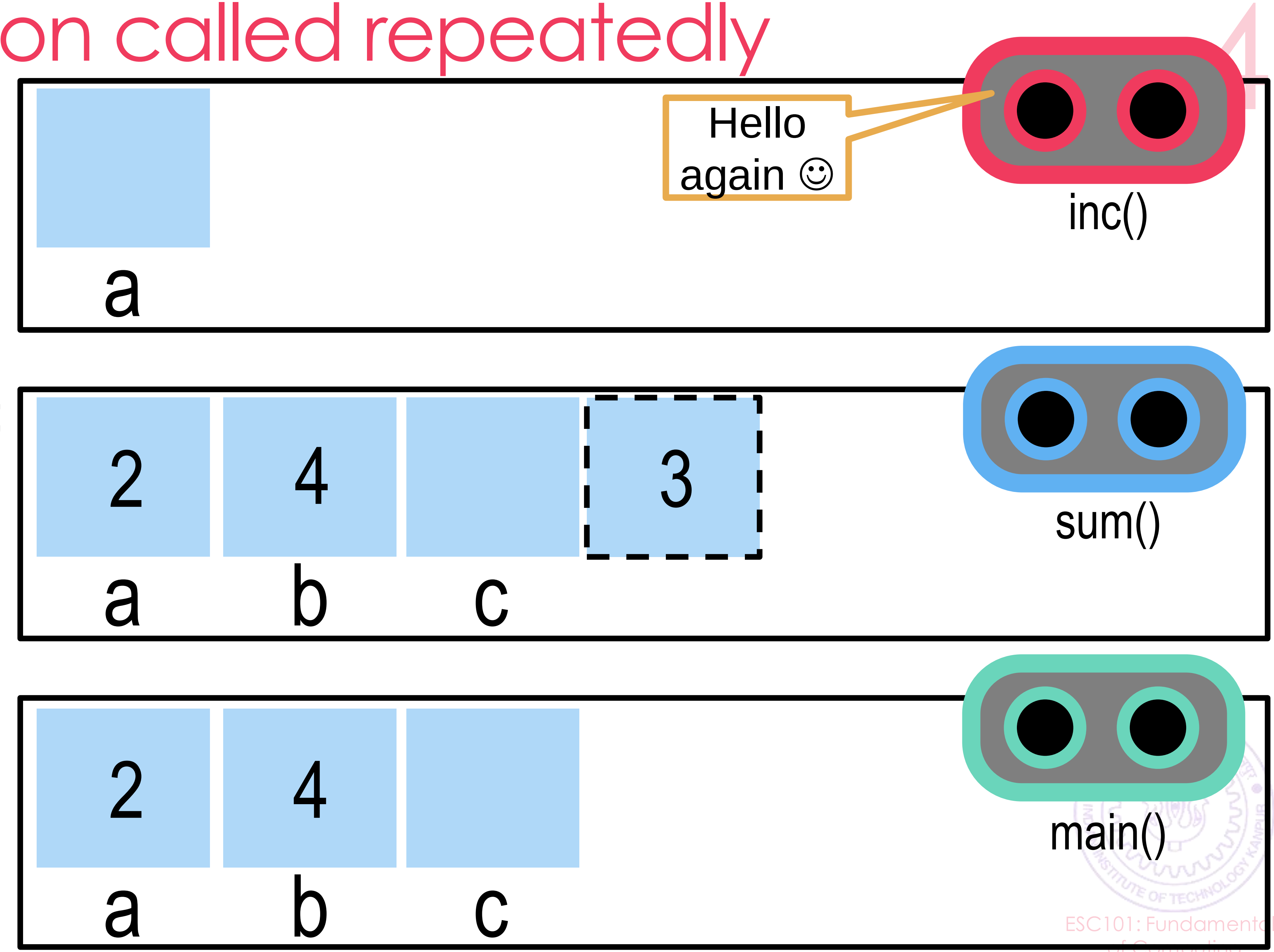
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



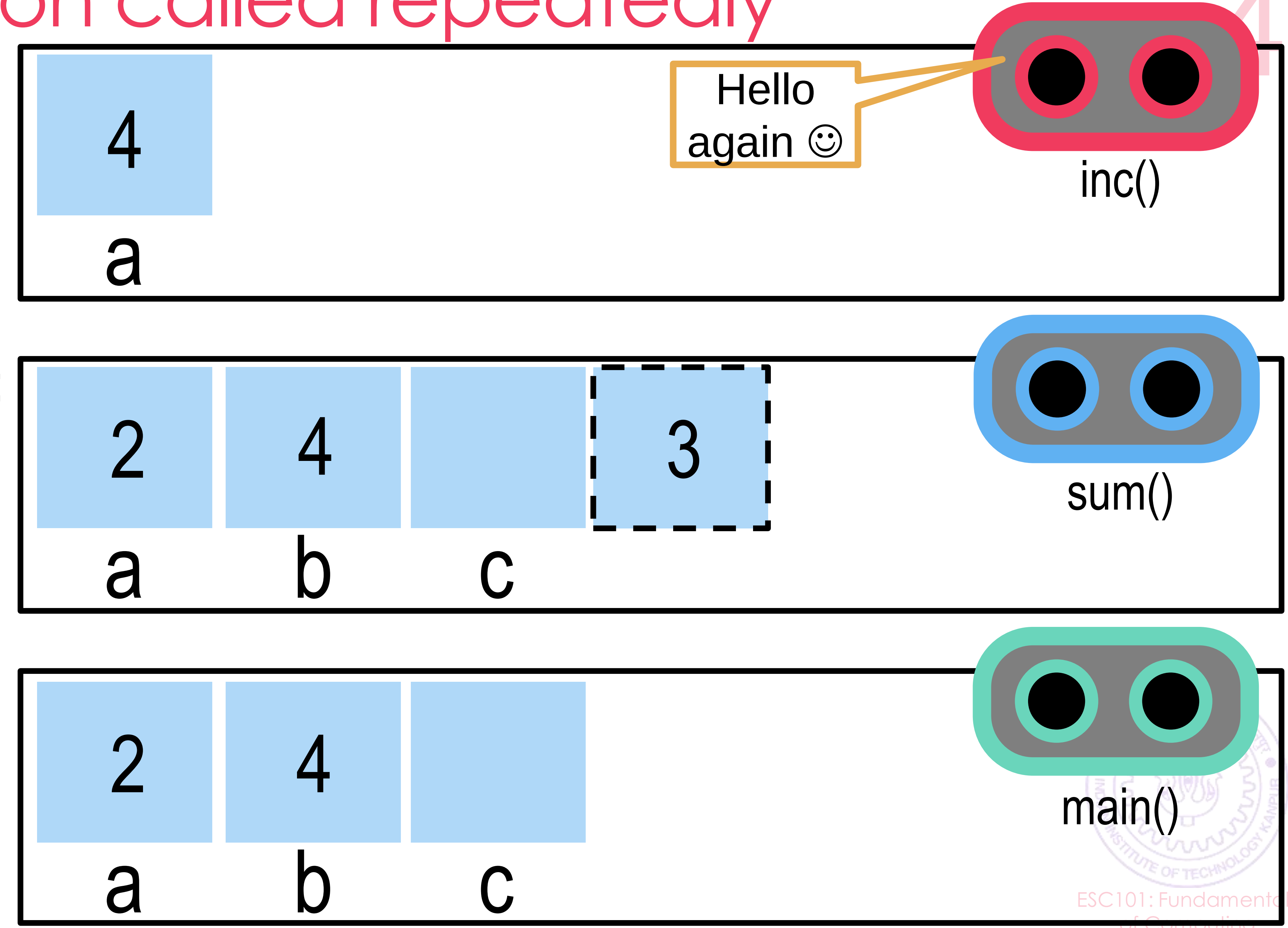
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



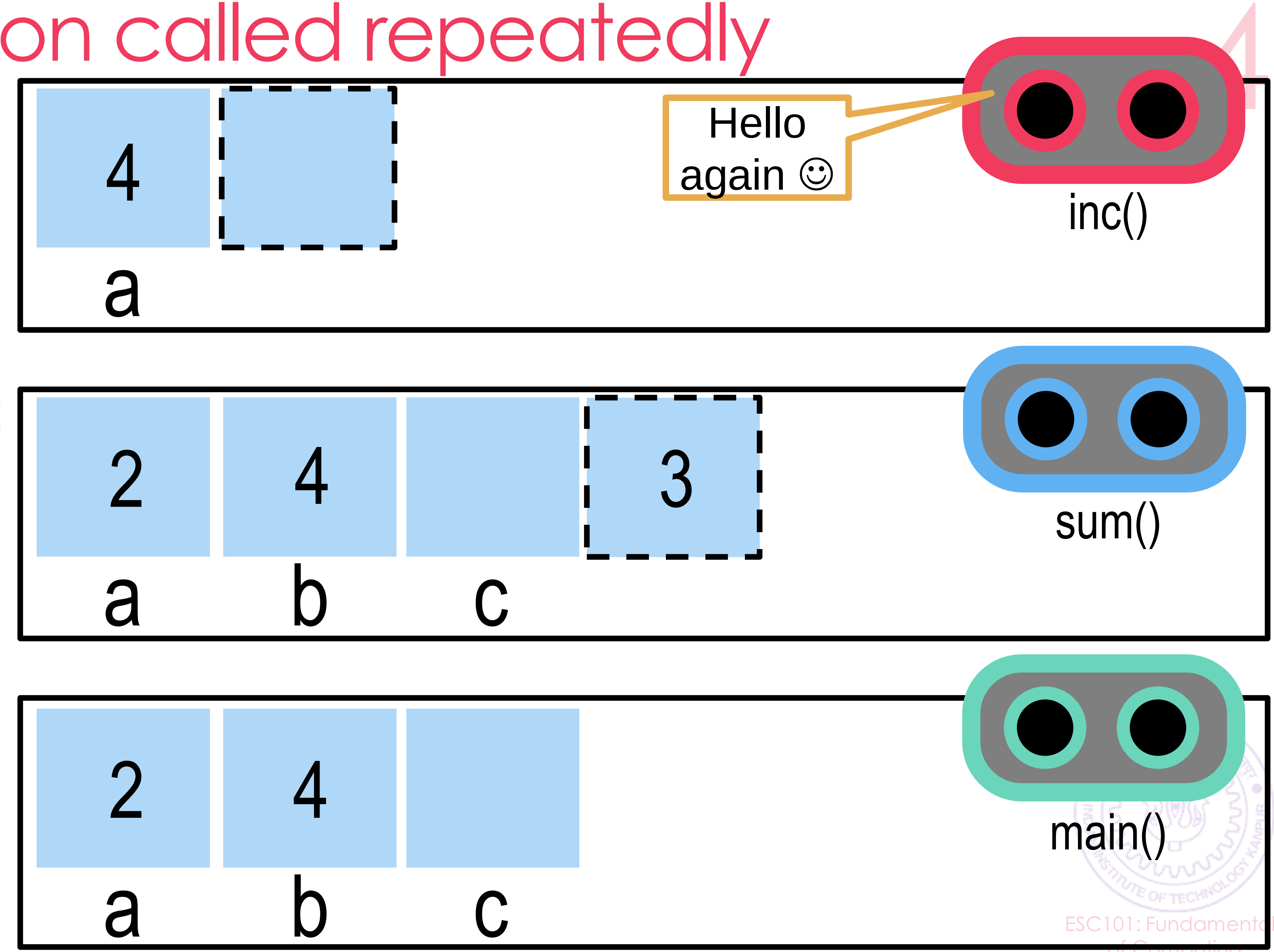
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



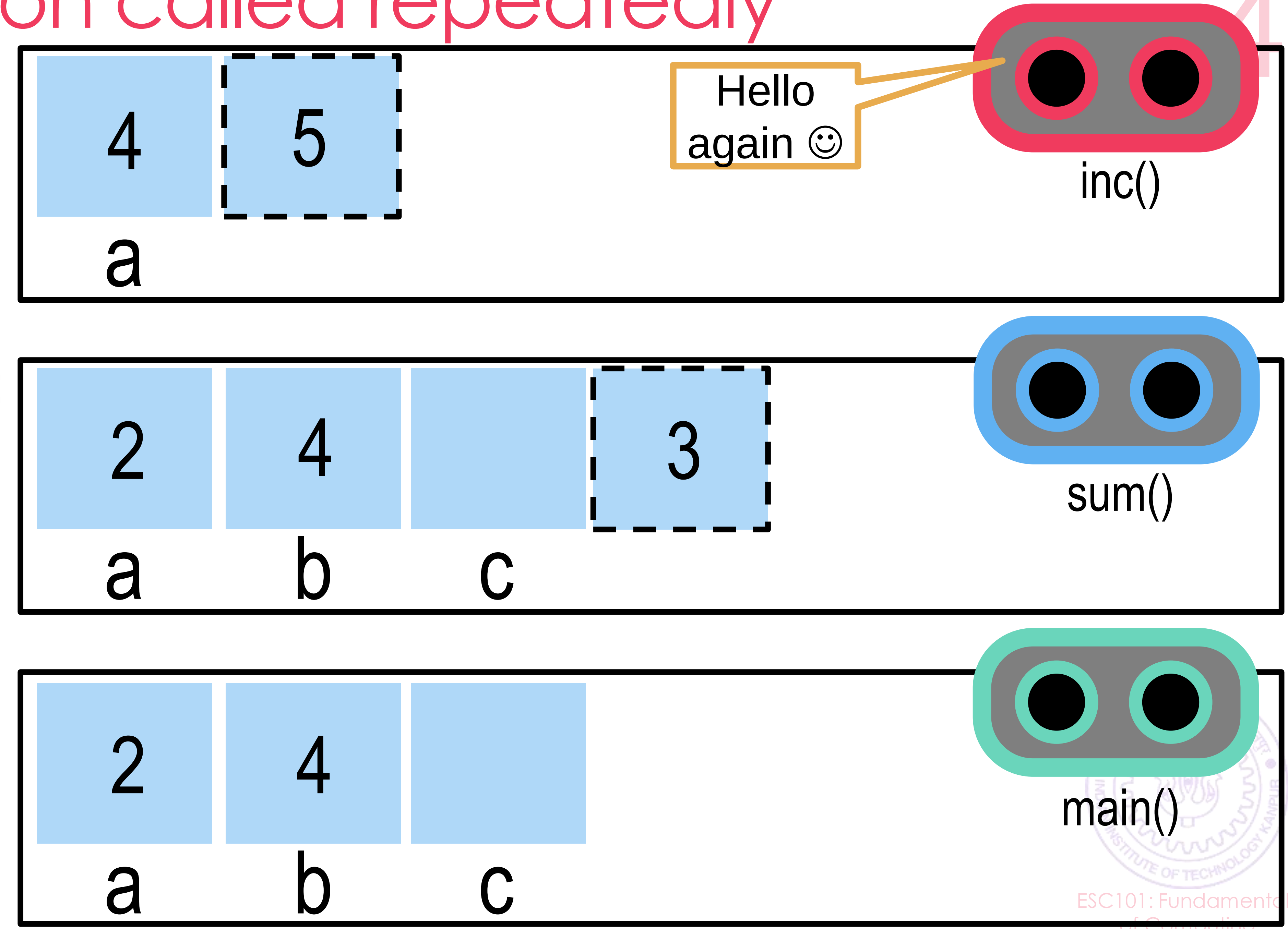
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



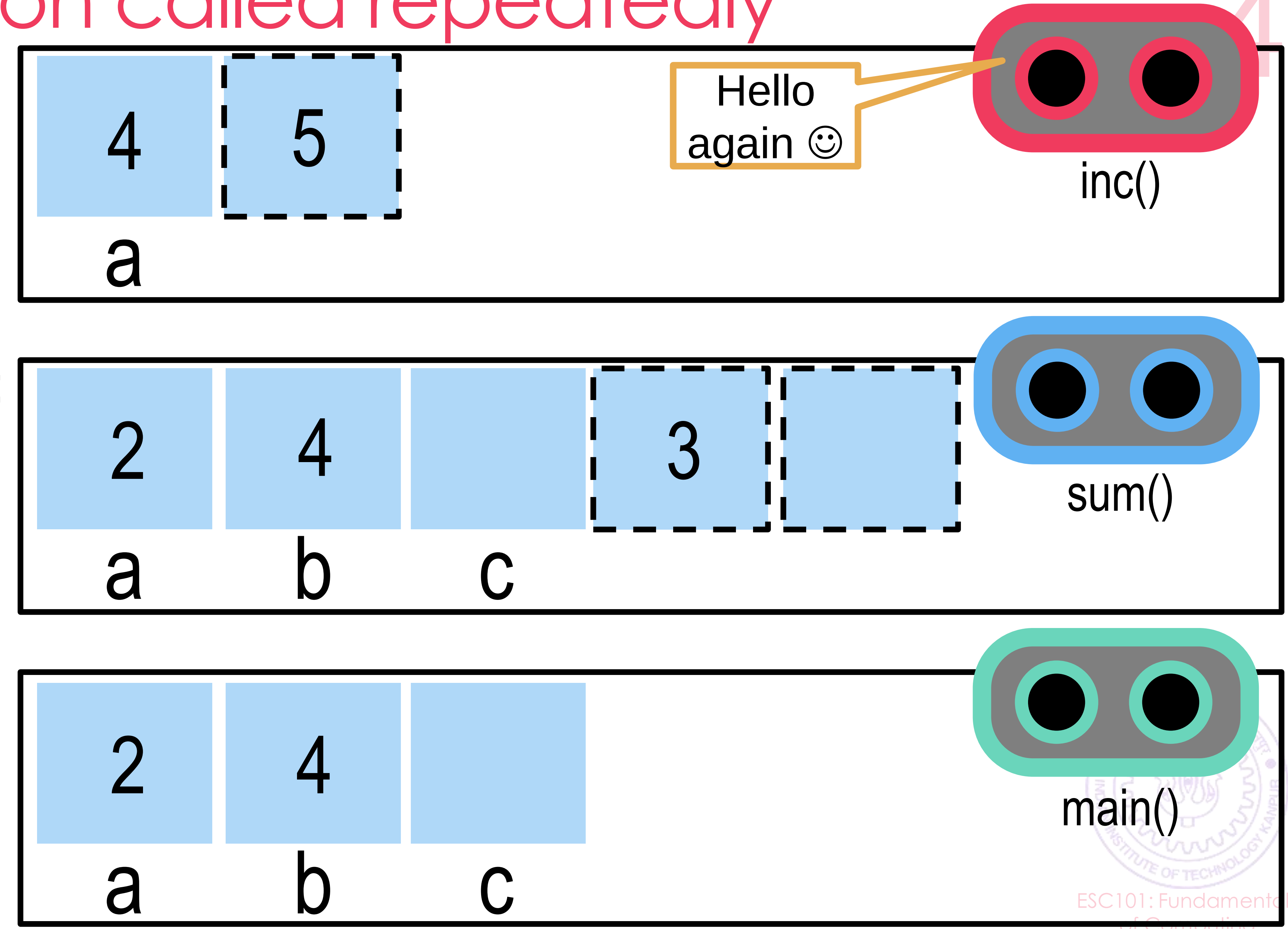
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



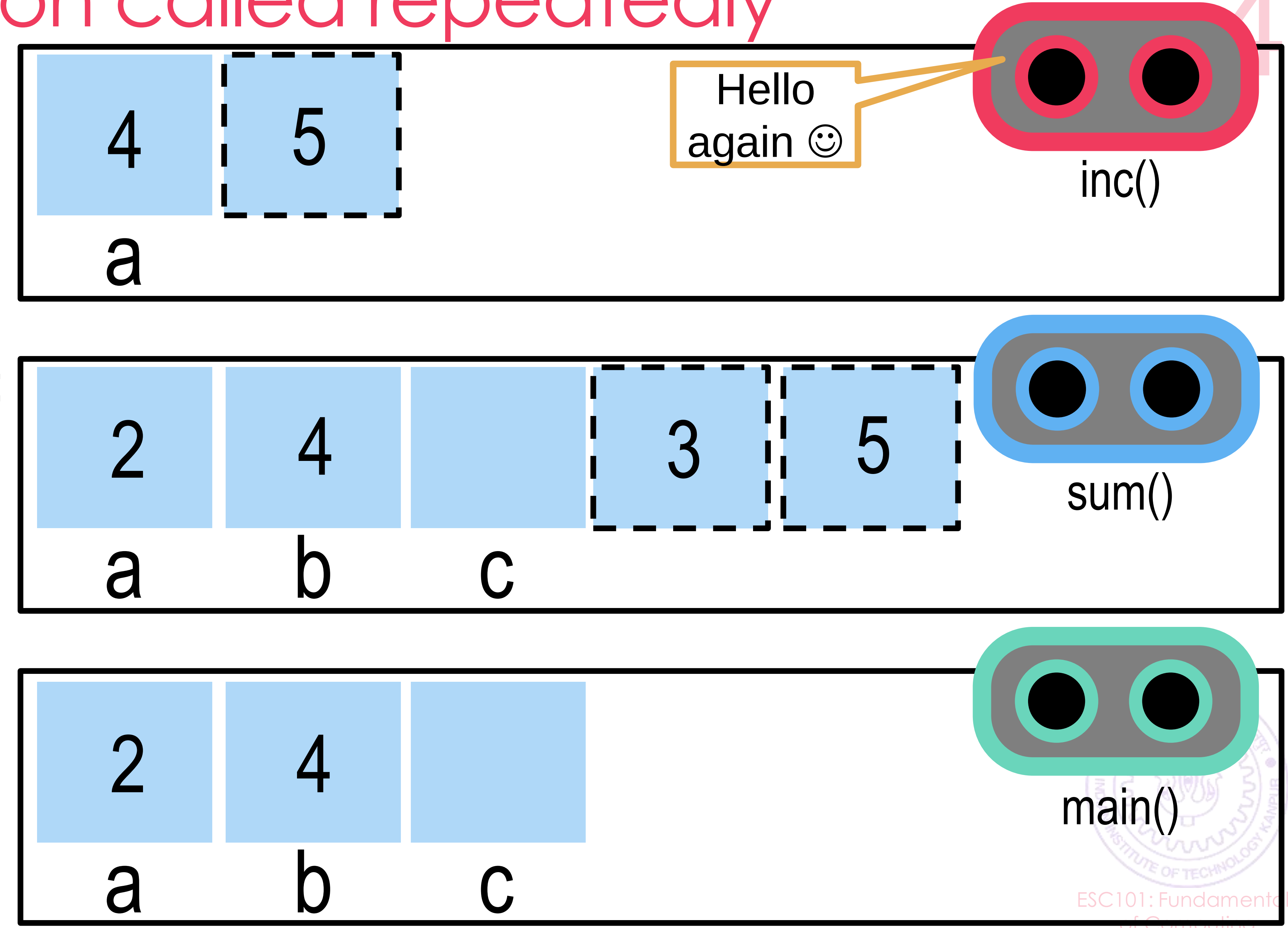
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



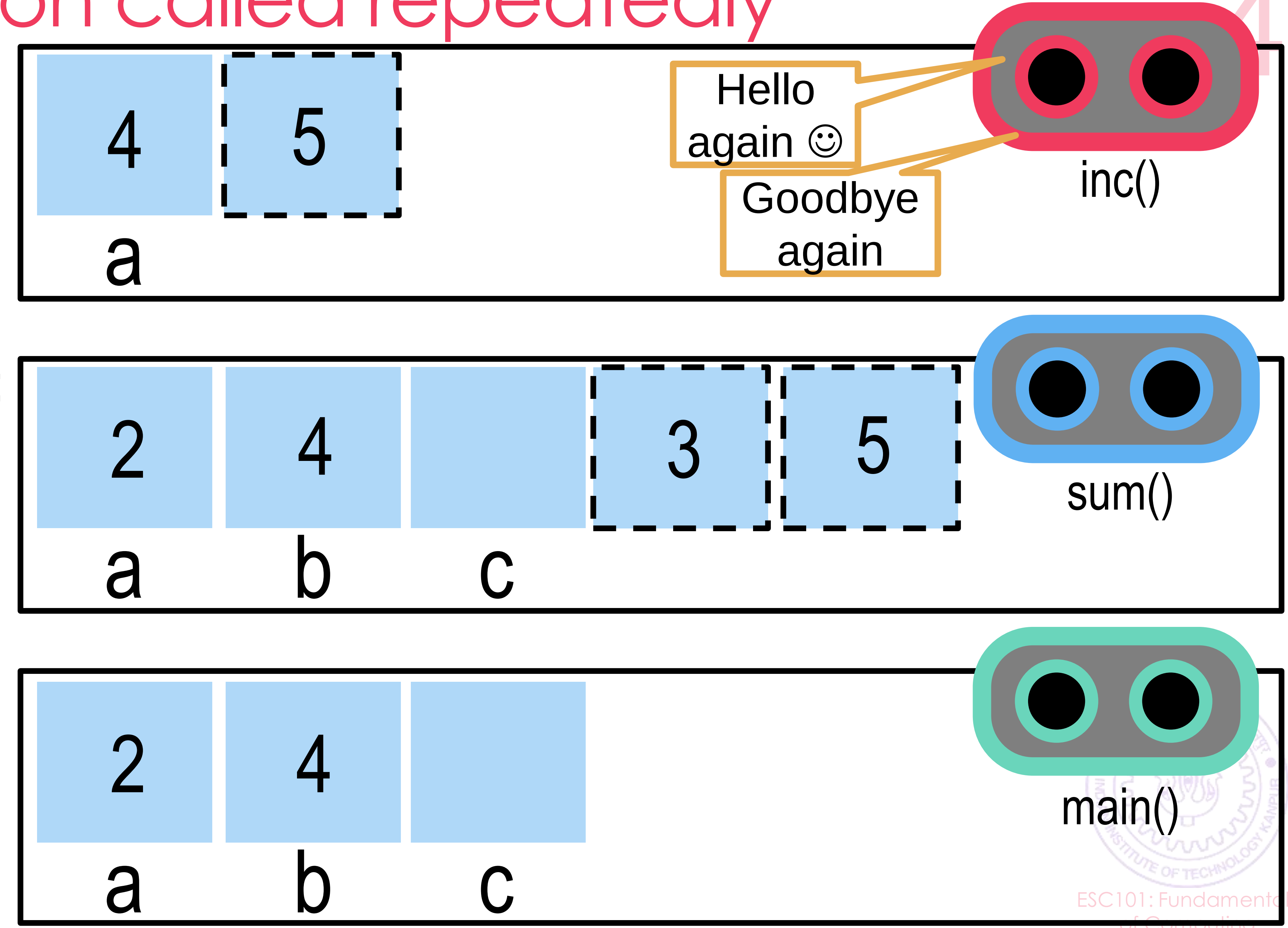
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



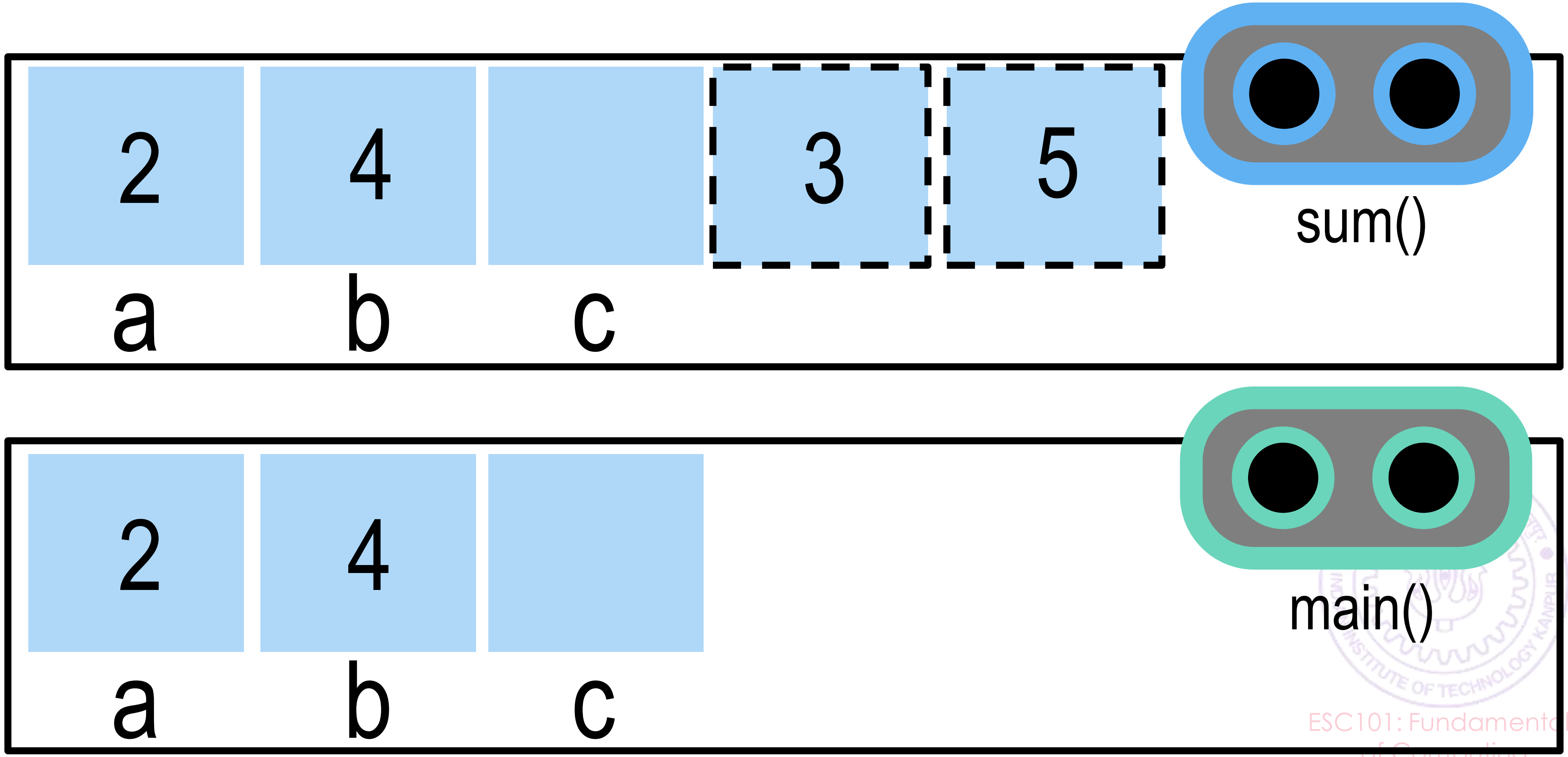
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



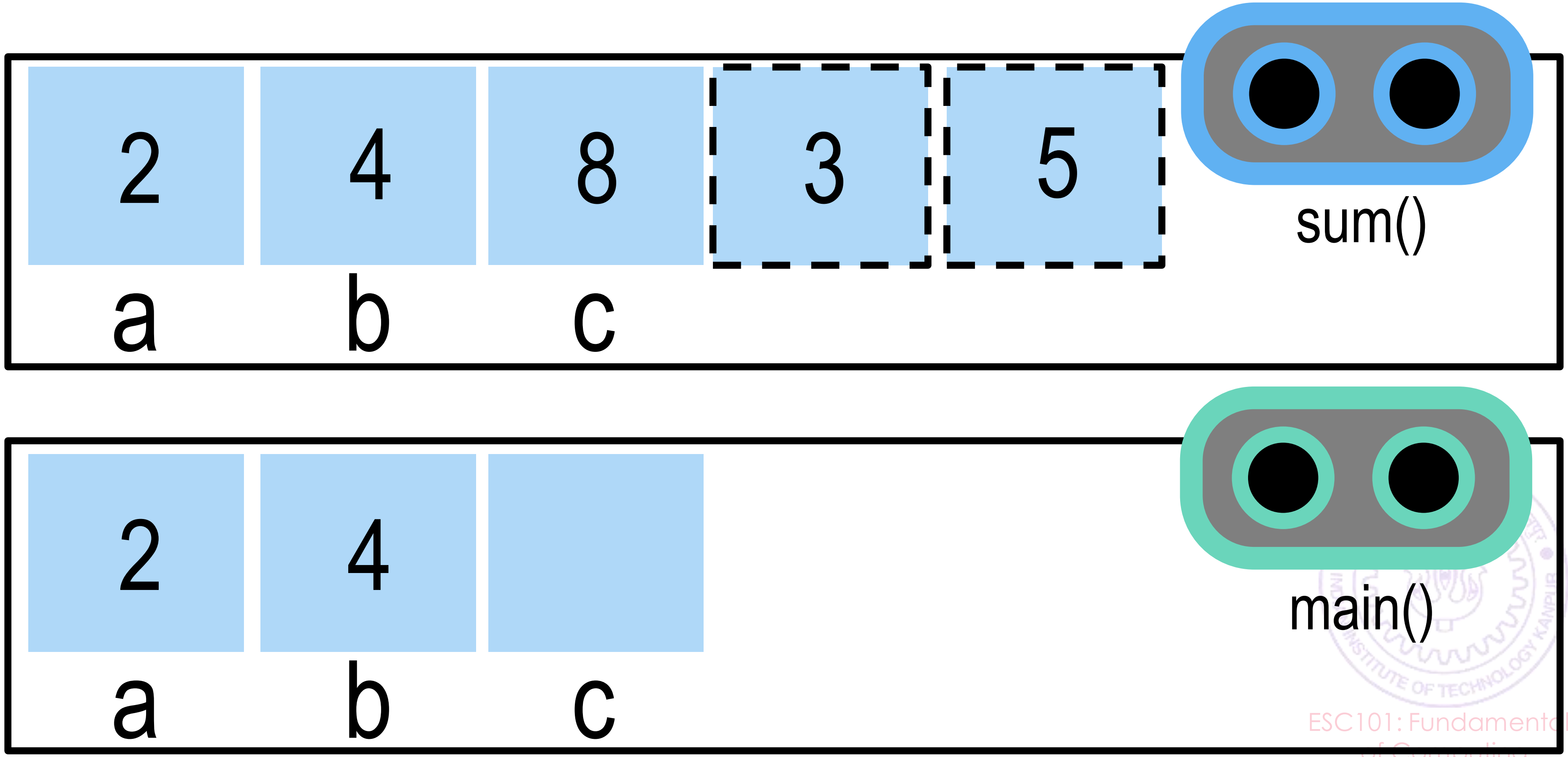
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



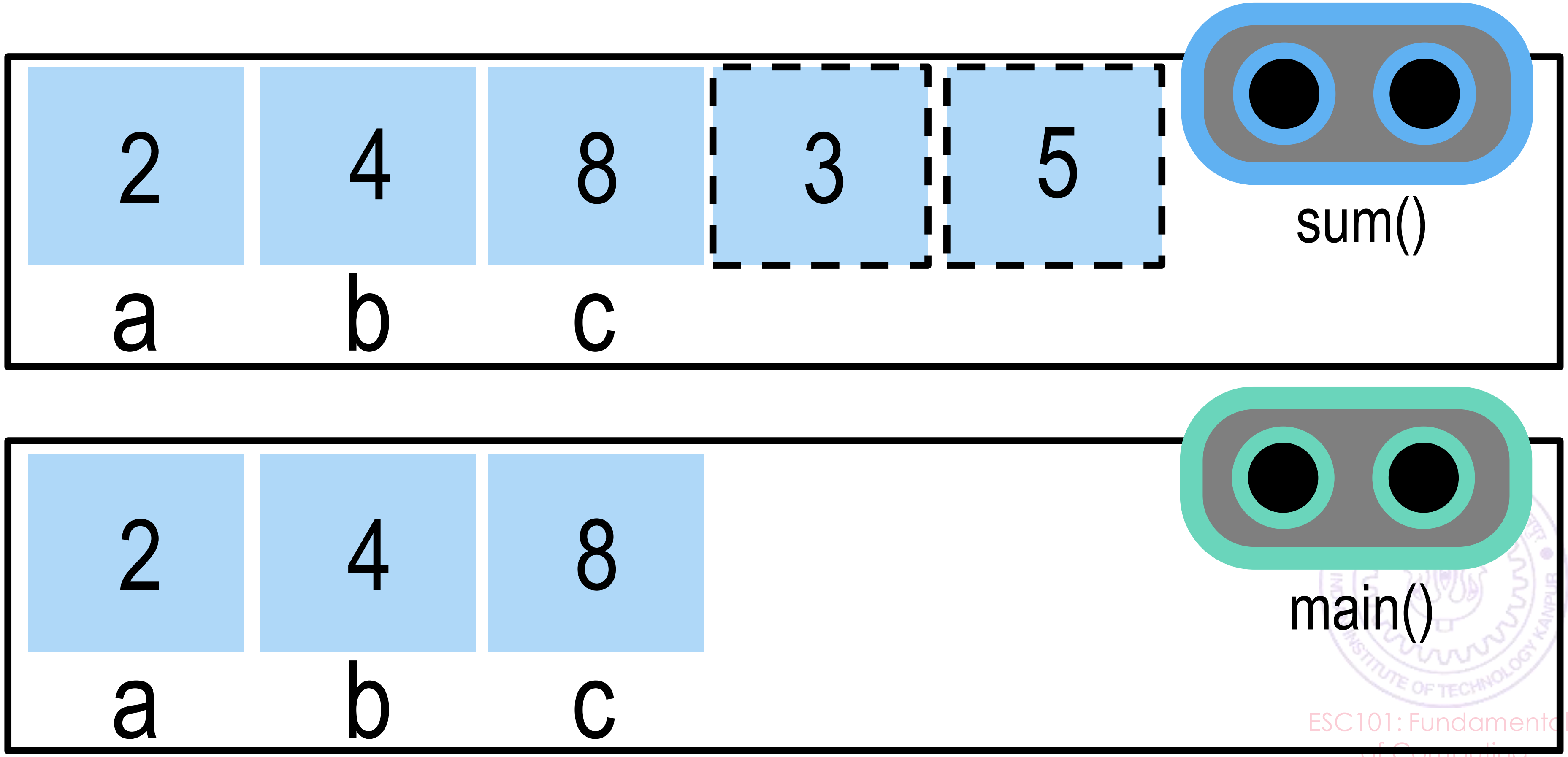
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



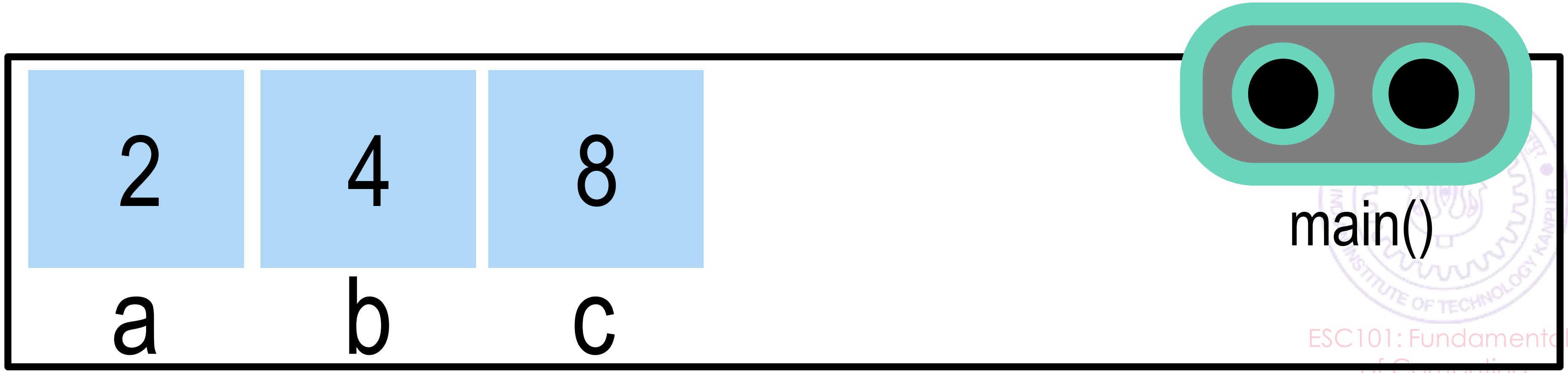
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



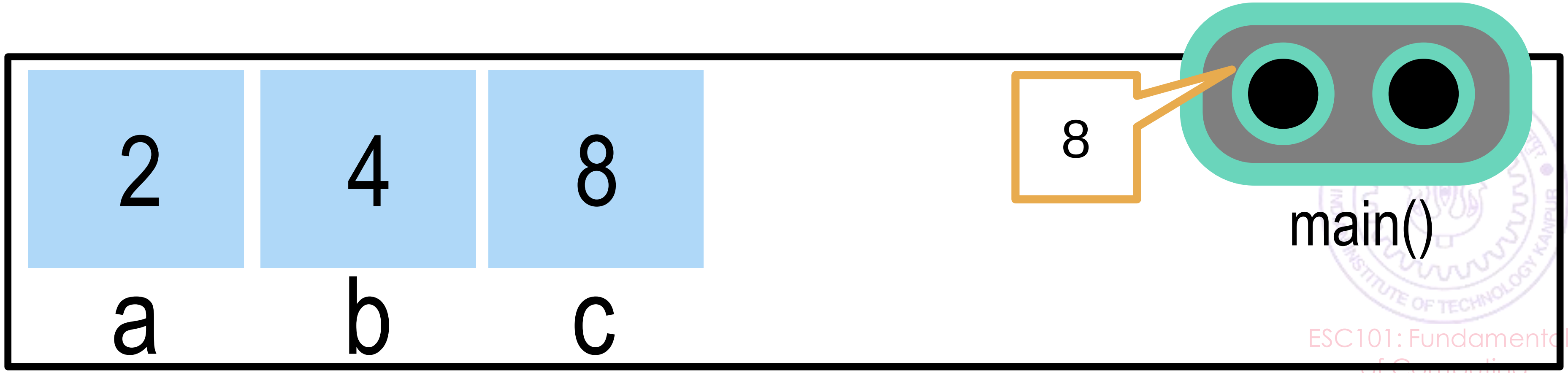
Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```



Same function called repeatedly

```
int inc(int a){
    return a+1;
}
int sum(int a, int b){
    int c = inc(a) + inc(b);
    return c;
}
int main(void){
    int a = 2, b = 4, c;
    c = sum(a, b);
    printf("%d", c);
    return 0;
}
```

The 4 Golden Rules of Functions

The 4 Golden Rules of Functions

RULE 1: When we give a variable as input, the value stored inside that variable gets passed as an argument

The 4 Golden Rules of Functions

RULE 1: When we give a variable as input, the value stored inside that variable gets passed as an argument

- Does not matter whether that variable is char, long, double or pointer

The 4 Golden Rules of Functions

RULE 1: When we give a variable as input, the value stored inside that variable gets passed as an argument

- Does not matter whether that variable is char, long, double or pointer
- If that variable is a pointer, address stored inside that pointer passed

The 4 Golden Rules of Functions

RULE 1: When we give a variable as input, the value stored inside that variable gets passed as an argument

- Does not matter whether that variable is char, long, double or pointer
- If that variable is a pointer, address stored inside that pointer passed
- Nothing new here – simply apply rule 1 of pointers

The 4 Golden Rules of Functions

RULE 1: When we give a variable as input, the value stored inside that variable gets passed as an argument

Does not matter whether that variable is char, long, double or pointer

If that variable is a pointer, address stored inside that pointer passed

Nothing new here – simply apply rule 1 of pointers

Rule 1 of pointers: all pointers (even pointers to pointers) store addresses

The 4 Golden Rules of Functions

RULE 1: When we give a variable as input, the value stored inside that variable gets passed as an argument

- Does not matter whether that variable is char, long, double or pointer
- If that variable is a pointer, address stored inside that pointer passed
- Nothing new here – simply apply rule 1 of pointers
- *Rule 1 of pointers*: all pointers (even pointers to pointers) store addresses

RULE 2: When we give an expression as input, the value generated by that expression gets passed as argument

The 4 Golden Rules of Functions

RULE 1: When we give a variable as input, the value stored inside that variable gets passed as an argument

- Does not matter whether that variable is char, long, double or pointer
- If that variable is a pointer, address stored inside that pointer passed
- Nothing new here – simply apply rule 1 of pointers
- *Rule 1 of pointers*: all pointers (even pointers to pointers) store addresses

RULE 2: When we give an expression as input, the value generated by that expression gets passed as argument

- Does not matter whether the value is a char, long, double or an address

The 4 Golden Rules of Functions

RULE 1: When we give a variable as input, the value stored inside that variable gets passed as an argument

Does not matter whether that variable is char, long, double or pointer

If that variable is a pointer, address stored inside that pointer passed

Nothing new here – simply apply rule 1 of pointers

Rule 1 of pointers: all pointers (even pointers to pointers) store addresses

RULE 2: When we give an expression as input, the value generated by that expression gets passed as argument

Does not matter whether the value is a char, long, double or an address

If we pass &a as input where a is a variable, address of a is passed

The 4 Golden Rules of Functions

RULE 1: When we give a variable as input, the value stored inside that variable gets passed as an argument

- Does not matter whether that variable is char, long, double or pointer
- If that variable is a pointer, address stored inside that pointer passed
- Nothing new here – simply apply rule 1 of pointers
- *Rule 1 of pointers*: all pointers (even pointers to pointers) store addresses

RULE 2: When we give an expression as input, the value generated by that expression gets passed as argument

- Does not matter whether the value is a char, long, double or an address
- If we pass &a as input where a is a variable, address of a is passed
- Nothing new here – simply apply rule 2 of pointers

The 4 Golden Rules of Functions

RULE 1: When we give a variable as input, the value stored inside that variable gets passed as an argument

- Does not matter whether that variable is char, long, double or pointer
- If that variable is a pointer, address stored inside that pointer passed
- Nothing new here – simply apply rule 1 of pointers
- *Rule 1 of pointers*: all pointers (even pointers to pointers) store addresses

RULE 2: When we give an expression as input, the value generated by that expression gets passed as argument

- Does not matter whether the value is a char, long, double or an address
- If we pass &a as input where a is a variable, address of a is passed
- Nothing new here – simply apply rule 2 of pointers
- *Rule 2 of pointers*: the expression &a generates the address of a as a value

The 4 Golden Rules of Functions

The 4 Golden Rules of Functions

RULE 3: If there is a mismatch between the type of input passed the type you promised when defining the function

The 4 Golden Rules of Functions

RULE 3: If there is a mismatch between the type of input passed the type you promised when defining the function

Mr C will try typecasting – may lose info e.g. float passed when int expected

The 4 Golden Rules of Functions

RULE 3: If there is a mismatch between the type of input passed the type you promised when defining the function

- Mr C will try typecasting – may lose info e.g. float passed when int expected
- We have seen Mr C do this several times in this course

The 4 Golden Rules of Functions

RULE 3: If there is a mismatch between the type of input passed the type you promised when defining the function

Mr C will try typecasting – may lose info e.g. float passed when int expected

We have seen Mr C do this several times in this course

If automatic typecasting not possible, Mr C will give error

The 4 Golden Rules of Functions

RULE 3: If there is a mismatch between the type of input passed the type you promised when defining the function

- Mr C will try typecasting – may lose info e.g. float passed when int expected
- We have seen Mr C do this several times in this course
- If automatic typecasting not possible, Mr C will give error

RULE 4: Whenever you pass an input value to a function, that value is stored in a fresh variable inside that function

The 4 Golden Rules of Functions

RULE 3: If there is a mismatch between the type of input passed the type you promised when defining the function

- Mr C will try typecasting – may lose info e.g. float passed when int expected
- We have seen Mr C do this several times in this course
- If automatic typecasting not possible, Mr C will give error

RULE 4: Whenever you pass an input value to a function, that value is stored in a fresh variable inside that function

- Modifying that value inside the function will **NOT** change the original value

The 4 Golden Rules of Functions

RULE 3: If there is a mismatch between the type of input passed the type you promised when defining the function

- Mr C will try typecasting – may lose info e.g. float passed when int expected
- We have seen Mr C do this several times in this course
- If automatic typecasting not possible, Mr C will give error

RULE 4: Whenever you pass an input value to a function, that value is stored in a fresh variable inside that function

- Modifying that value inside the function will **NOT** change the original value
- Does not matter whether the value passed is char or long or an address

The 4 Golden Rules of Functions

RULE 3: If there is a mismatch between the type of input passed the type you promised when defining the function

Mr C will try typecasting – may lose info e.g. float passed when int expected

We have seen Mr C do this several times in this course

If automatic typecasting not possible, Mr C will give error

RULE 4: Whenever you pass an input value to a function, that value is stored in a fresh variable inside that function

Modifying that value inside the function will **NOT** change the original value

Does not matter whether the value passed is char or long or an address

This rule may seem confusing when we are passing pointers to functions but make no mistake – rule 4 ALWAYS applies unless global variables involved

More on Return

More on Return

May write return statement many times inside a function

More on Return

May write return statement many times inside a function

When Mr C (his clone actually) sees a return statement, he immediately generates the output and function execution stops there.

More on Return

May write return statement many times inside a function

When Mr C (his clone actually) sees a return statement, he immediately generates the output and function execution stops there.

The clone dies and the original Mr C takes over ☺

More on Return

May write return statement many times inside a function

When Mr C (his clone actually) sees a return statement, he immediately generates the output and function execution stops there.

The clone dies and the original Mr C takes over ☺

Warning: if you have promised that a function returns an integer, all return statements in that function must return an integer value – otherwise compilation error!

More on Return

May write return statement many times inside a function

When Mr C (his clone actually) sees a return statement, he immediately generates the output and function execution stops there.

The clone dies and the original Mr C takes over ☺

Warning: if you have promised that a function returns an integer, all return statements in that function must return an integer value – otherwise compilation error!

If you return a float/double value from a function with int return type, automatic typecasting will take place.

More on Return

May write return statement many times inside a function

When Mr C (his clone actually) sees a return statement, he immediately generates the output and function execution stops there.

The clone dies and the original Mr C takes over ☺

Warning: if you have promised that a function returns an integer, all return statements in that function must return an integer value – otherwise compilation error!

If you return a float/double value from a function with int return type, automatic typecasting will take place.

Be careful to not make typecasting mistakes

More on Return

May write return statement many times inside a function

When Mr C (his clone actually) sees a return statement, he immediately generates the output and function execution stops there.

The clone dies and the original Mr C takes over ☺

Warning: if you have promised that a function returns an integer, all return statements in that function must return an integer value – otherwise compilation error!

If you return a float/double value from a function with int return type, automatic typecasting will take place.

Be careful to not make typecasting mistakes

More o

For functions that do not need to return anything i.e. void return type, you can either say return; or else not write return at all inside the function body in which case the entire body will get executed

May write re

function

When Mr C (his clone actually) sees a return statement, he immediately generates the output and function execution stops there.

The clone dies and the original Mr C takes over ☺

Warning: if you have promised that a function returns an integer, all return statements in that function must return an integer value – otherwise compilation error!

If you return a float/double value from a function with int return type, automatic typecasting will take place.

Be careful to not make typecasting mistakes

More on Return

More on Return

The value that is returned can be used safely just as a normal variable of that same data type

More on Return

The value that is returned can be used safely just as a normal variable of that same data type

You can freely use returned values in expressions

More on Return

The value that is returned can be used safely just as a normal variable of that same data type

You can freely use returned values in expressions

- Be careful of type though

More on Return

- The value that is returned can be used safely just as a normal variable of that same data type
- You can freely use returned values in expressions
  - Be careful of type though
- Did you know that the printf function also returns an integer (the number of characters printed) ☺

More on Return

The value that is returned can be used safely just as a normal variable of that same data type

You can freely use returned values in expressions

- Be careful of type though

Did you know that the printf function also returns an integer (the number of characters printed) ☺

scanf() also returns an integer – find out what that is !!

More on Return

The value that is returned can be used safely just as a normal variable of that same data type

You can freely use returned values in expressions

- Be careful of type though

Did you know that the printf function also returns an integer (the number of characters printed) ☺

scanf() also returns an integer – find out what that is !!

```
int sum(int x, int y){
    return x + y;
}
```

More on Return

The value that is returned can be used safely just as a normal variable of that same data type

You can freely use returned values in expressions

- Be careful of type though

Did you know that the printf function also returns an integer (the number of characters printed) ☺

scanf() also returns an integer – find out what that is !!

```
int sum(int x, int y){
    return x + y;
}
```

```
int main(){
    printf("%d", sum(3,4) - sum(5,6));
    return 0;
}
```

More on Return

The value that is returned can be used safely just as a normal variable of that same data type

You can freely use returned values in expressions

- Be careful of type though

Did you know that the printf function also returns an integer (the number of characters printed) ☺

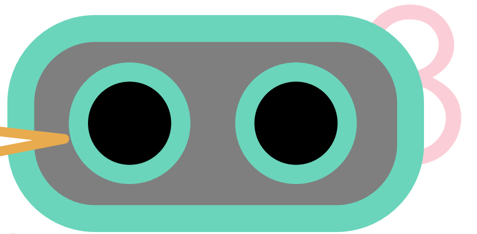
scanf() also returns an integer – find out what that is !!

```
int sum(int x, int y){
    return x + y;
}
```

```
int main(){
    printf("%d", sum(3,4) - sum(5,6));
    return 0;
}
```

More on Return

main() is also a function with return type int



The value that is returned can be used safely just as a normal variable of that same data type

You can freely use returned values in expressions

- Be careful of type though

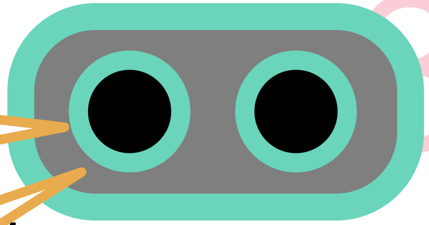
Did you know that the printf function also returns an integer (the number of characters printed) ☺

scanf() also returns an integer – find out what that is !!

```
int sum(int x, int y){
    return x + y;
}
```

```
int main(){
    printf("%d", sum(3,4) - sum(5,6));
    return 0;
}
```

More on Return



main() is also a function with return type int

main() is like a reserved function name. Cannot name your function main

The value that is re ust as a normal variable of

You can freely use returned values in expressions

- Be careful of type though

Did you know that the printf function also returns an integer (the number of characters printed) ☺

scanf() also returns an integer – find out what that is !!

```
int sum(int x, int y){
    return x + y;
}
```

```
int main(){
    printf("%d", sum(3,4) - sum(5,6));
    return 0;
}
```

Benefits of writing functions

Benefits of writing functions

Allows you to think very clearly

Benefits of writing functions

Allows you to think very clearly

E.g. if you want to do something if the integer n is a prime number or if it is divisible by 11

Benefits of writing functions

Allows you to think very clearly

E.g. if you want to do something if the integer n is a prime number or if it is divisible by 11

```
if(isPrime(n) || isDivby11(n)){
    …
}
```

Benefits of writing functions

Allows you to think very clearly

E.g. if you want to do something if the integer n is a prime number or if it is divisible by 11

```
if(isPrime(n) || isDivby11(n)){
    …
}
```

Write the body of the if condition without worrying about primality testing etc and then define the functions later ☺

Benefits of writing functions

Allows you to think very clearly

E.g. if you want to do something if the integer n is a prime number or if it is divisible by 11

```
if(isPrime(n) || isDivby11(n)){
    ...
}
```

Write the body of the if condition without worrying about primality testing etc and then define the functions later ☺

You can break your code into chunks – called modules

Benefits of writing functions

Allows you to think very clearly

E.g. if you want to do something if the integer n is a prime number or if it is divisible by 11

```
if(isPrime(n) || isDivby11(n)){
    ...
}
```

Write the body of the if condition without worrying about primality testing etc and then define the functions later ☺

You can break your code into chunks – called modules

Each module handled using a separate function

Benefits of writing functions

Allows you to think very clearly

E.g. if you want to do something if the integer n is a prime number or if it is divisible by 11

```
if(isPrime(n) || isDivby11(n)){
    …
}
```

Write the body of the if condition without worrying about primality testing etc and then define the functions later ☺

You can break your code into chunks – called modules

Each module handled using a separate function

Benefits of writing functions

Allows you to think very clearly

E.g. if you want to do sor a prime number or if it is divisible

E.g. in this case, primality testing is one module, checking for divisibility by 11 is another module

```
if(isPrime(n) || isDivby11(n)){
    ...
}
```

Write the body of the if condition without worrying about primality testing etc and then define the functions later ☺

You can break your code into chunks – called modules

Each module handled using a separate function

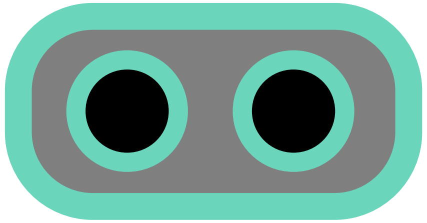
Benefits of writing functions

Allows you to think very clearly

E.g. if you want to do sor a prime number or if it is divisible

E.g. in this case, primality testing is one module, checking for divisibility by 11 is another module

```
if(isPrime(n) || isDivby11(n)){
    ...
}
```



Write the body of the if condition without worrying about primality testing etc and then define the functions later ☺

You can break your code into chunks – called modules

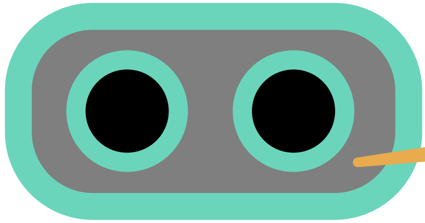
Each module handled using a separate function

Benefits of writing functions

Allows you to think very clearly

E.g. if you want to do sor a prime number or if it is divisible

```
if(isPrime(n) || isDivby11(n)){
    …
}
```



E.g. in this case, primality testing is one module, checking for divisibility by 11 is another module

Writing code that has modules is a type of *modular programming* – it is the the industry standard!

Write the body of the if condition without worrying about primality testing etc and then define the functions later ☺

You can break your code into chunks – called modules

Each module handled using a separate function

Benefits of writing functions

Benefits of writing functions

Functions allow you to write very neat, readable code

Benefits of writing functions

Functions allow you to write very neat, readable code

Use function names that describe what the function does

Benefits of writing functions

Functions allow you to write very neat, readable code

Use function names that describe what the function does

Your co-workers/team-mates will be able to understand your code much better if it has nice readable functions

Benefits of writing functions

Functions allow you to write very neat, readable code

Use function names that describe what the function does

Your co-workers/team-mates will be able to understand your code much better if it has nice readable functions

Functions allow you to debug your program faster

Benefits of writing functions

Functions allow you to write very neat, readable code

Use function names that describe what the function does

Your co-workers/team-mates will be able to understand your code much better if it has nice readable functions

Functions allow you to debug your program faster

If code is broken into function, to debug, find out which function is not working properly ☺

Benefits of writing functions

Functions allow you to write very neat, readable code

Use function names that describe what the function does

Your co-workers/team-mates will be able to understand your code much better if it has nice readable functions

Functions allow you to debug your program faster

If code is broken into function, to debug, find out which function is not working properly ☺

Rest of code need not be touched, only faulty function needs to be fixed – again the industry standard of code maintenance.

Benefits of writing functions

Benefits of writing functions

Functions allow you to reuse code

Benefits of writing functions

Functions allow you to reuse code

We are so grateful some one wrote functions like sqrt(), abs() in math.h that we are able to use again and again ☺

Benefits of writing functions

Functions allow you to reuse code

We are so grateful some one wrote functions like sqrt(), abs() in math.h that we are able to use again and again ☺

printf() and scanf() are also functions. Think of how much we use them in every single program

Benefits of writing functions

Functions allow you to reuse code

We are so grateful some one wrote functions like sqrt(), abs() in math.h that we are able to use again and again ☺

printf() and scanf() are also functions. Think of how much we use them in every single program

We are reusing code that some helpful C expert wrote in the printf(), scanf(), sqrt(), abs() and other functions

Benefits of writing functions

Functions allow you to reuse code

We are so grateful some one wrote functions like sqrt(), abs() in math.h that we are able to use again and again ☺

printf() and scanf() are also functions. Think of how much we use them in every single program

We are reusing code that some helpful C expert wrote in the printf(), scanf(), sqrt(), abs() and other functions

If some piece of code keeps getting used in your program again and again – put it inside a function!

Benefits of writing functions

Functions allow you to reuse code

We are so grateful some one wrote functions like sqrt(), abs() in math.h that we are able to use again and again ☺

printf() and scanf() are also functions. Think of how much we use them in every single program

We are reusing code that some helpful C expert wrote in the printf(), scanf(), sqrt(), abs() and other functions

If some piece of code keeps getting used in your program again and again – put it inside a function!

We reused code in today's codes – didn't have to rewrite code – may make mistakes if you write same code again