Fun with Functions

ESC101: Fundamentals of Computing

Purushottam Kar

Announcements

- No tutorial this week (Students Gymkhana holiday)
  - Doubt clearing session: Friday 26 October 6PM KD101
  - Students of all sections welcome to doubt clearing session
  - Tutorial sheet will still be released on website – read it!
- AT mid-term presentations this week – see schedule!
- Next week – major quiz and end sem lab exam
  - Refer to course calendar on course website to clarify dates

The 6 Golden Rules of Functions

The 6 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 6 Golden Rules of Functions

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

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

The 6 Golden Rules of Functions

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

- If that variable is a pointer, the address stored inside that pointer gets passed
- *Rule 1 of pointers*: all pointers (even pointers to pointers) store addresses

The 6 Golden Rules of Functions

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

- If that variable is a pointer, the address stored inside that pointer gets passed
- *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 6 Golden Rules of Functions

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

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

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

If the expression is generating an address (e.g. &a), that address gets passed

The 6 Golden Rules of Functions

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

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

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

If the expression is generating an address (e.g. &a), that address gets passed

Rule 2 of pointers: the expression &a generates the address of a as a value

The 6 Golden Rules of Functions

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

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

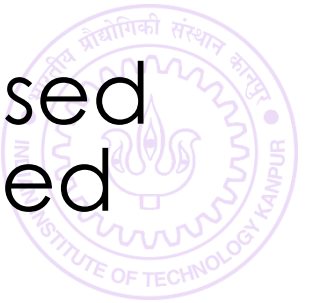
Rule 1 of pointers: all pointers (even pointers to pointers) store addresses

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

If the expression is generating an address (e.g. &a), that address gets passed

Rule 2 of pointers: the expression &a generates the address of a as a value

RULE 3: In case of a mismatch b/w type of arg promised and type of arg passed, typecasting will be attempted



The 6 Golden Rules of Functions

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

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

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

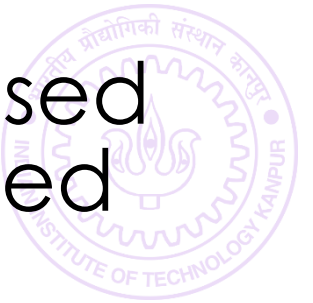
RULE 2: When we give an expression as input, the value generated by that expression gets passed as argument

If the expression is generating an address (e.g. &a), that address gets passed

Rule 2 of pointers: the expression &a generates the address of a as a value

RULE 3: In case of a mismatch b/w type of arg promised and type of arg passed, typecasting will be attempted

WARNING: may cause loss of information or unexpected behavior



The 6 Golden Rules of Functions

The 6 Golden Rules of Functions

RULE 4: All values passed to a function get stored in a fresh variable inside that function

The 6 Golden Rules of Functions

RULE 4: All values passed to a function get stored in a fresh variable inside that function

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

The 6 Golden Rules of Functions

RULE 4: All values passed to a function get 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 6 Golden Rules of Functions

RULE 4: All values passed to a function get 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

RULE 5: Value returned by a function can be used freely in any way values of that data-type could have been used

The 6 Golden Rules of Functions

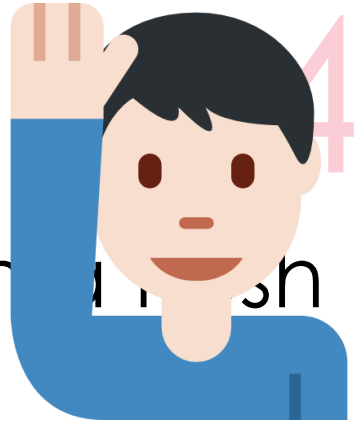
RULE 4: All values passed to a function get stored in a fresh variable inside that function

- 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

RULE 5: Value returned by a function can be used freely in any way values of that data-type could have been used

- If function is returning a float, feel free to take square root with it

The 6 Golden Rules of Functions (4)



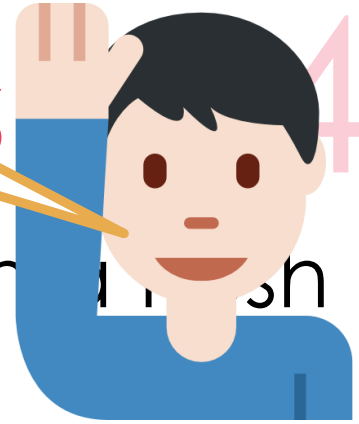
RULE 4: All values passed to a function get stored in fresh variable inside that function

- 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

RULE 5: Value returned by a function can be used freely in any way values of that data-type could have been used

- If function is returning a float, feel free to take square root with it

The 6 Golden Rules



RULE 4: All values passed to a function get stored in 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

RULE 5: Value returned by a function can be used freely in any way values of that data-type could have been used

- If function is returning a float, feel free to take square root with it

The 6 Golden Rules

However, verify that the float returned is not negative

RULE 4: All values passed to a function get 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

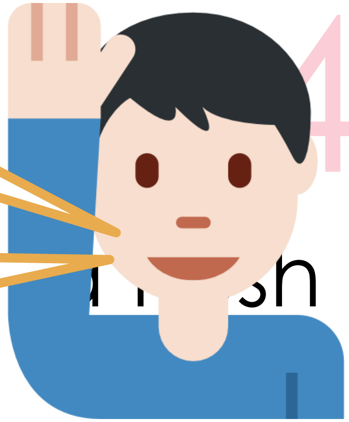
RULE 5: Value returned by a function can be used freely in any way values of that data-type could have been used

- If function is returning a float, feel free to take square root with it
- If function is returning an int, feel fee to use it as an array index

The 6 Golden Rule… s … 4

However, verify that the float returned is not negative

However, verify that the int gives an index within bounds



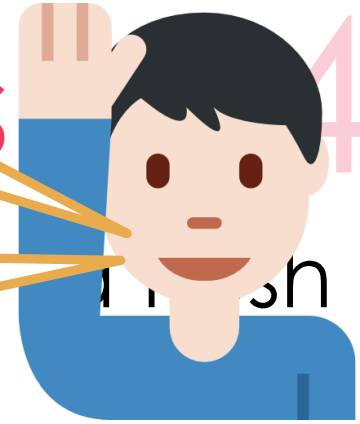
RULE 4: All values passed to … i … sh variable inside that function

- 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

RULE 5: Value returned by a function can be used freely in any way values of that data-type could have been used

- If function is returning a float, feel free to take square root with it
- If function is returning an int, feel fee to use it as an array index

The 6 Golden Rules



RULE 4: All values passed to a function are passed in through a 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

RULE 5: Value returned by a function can be used freely in any way values of that data-type could have been used

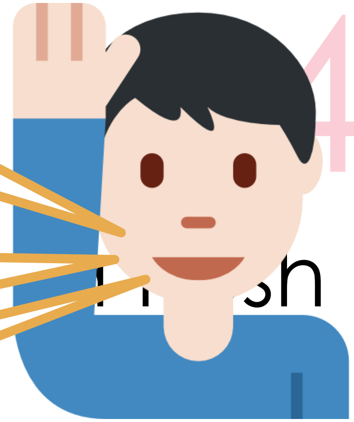
- If function is returning a float, feel free to take square root with it
- If function is returning an int, feel fee to use it as an array index
- If function is returning an address, feel free to dereference that address

The 6 Golden Rule

However, verify that the float returned is not negative

However, verify that the int gives an index within bounds

However, verify that the address returned isn't NULL



RULE 4: All values passed to

variable inside that function

Modifying that value inside the func ginal value

Does not matter whether the value address

RULE 5: Value returned by a function can be used freely in any way values of that data-type could have been used

If function is returning a float, feel free to take square root with it

If function is returning an int, feel fee to use it as an array index

If function is returning an address, feel free to dereference that address

The 6 Golden Rule

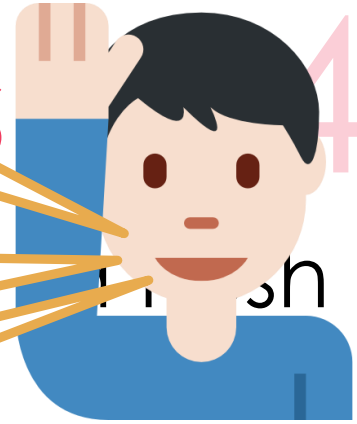
RULE 4: All values passed to
variable inside that function

Modifying that value inside the func
Does not matter whether the value

However, verify that the float returned is not negative

However, verify that the int gives an index within bounds

However, verify that the address returned isn't NULL



RULE 5: Value returned by a function can be used fr
any way values of that data-type could have bee

If function is returning a float, feel free to take square root with it
If function is returning an int, feel fee to use it as an array index
If function is returning an address, feel free to dereference that address

The 6 Golden Rule

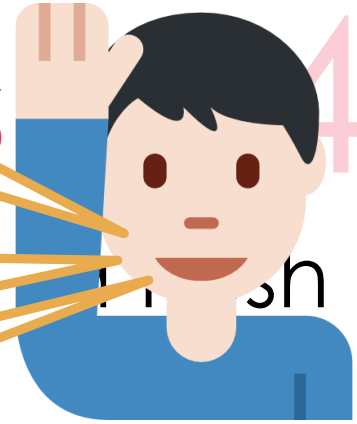
RULE 4: All values passed to
variable inside that function

Modifying that value inside the func ginal value
Does not matter whether the value address

However, verify that the float returned is not negative

However, verify that the int gives an index within bounds

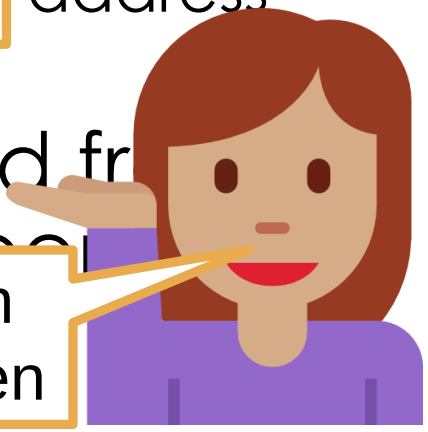
However, verify that the address returned isn't NULL



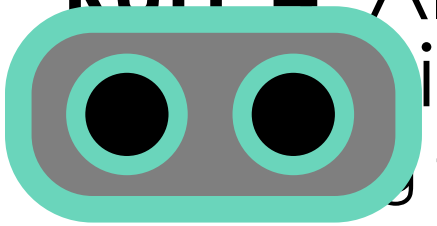
RULE 5: Value returned by a function can be used fr
any way values of th

If function is returning a flo
If function is returning an in
If function is returning an address, feel free to dereference that address

Remember, Mr C terminates a function the moment any return statement is seen



The 6 Golden Rule



RULE 4: All values passed to inside that function

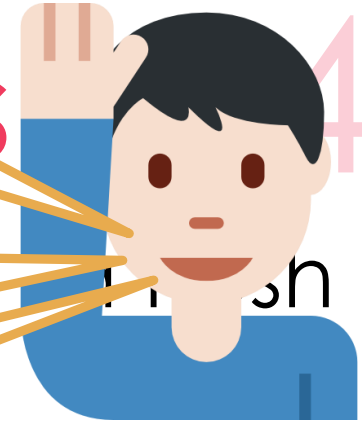
that value inside the func ginal value

Does not matter whether the value address

However, verify that the float returned is not negative

However, verify that the int gives an index within bounds

However, verify that the address returned isn't NULL



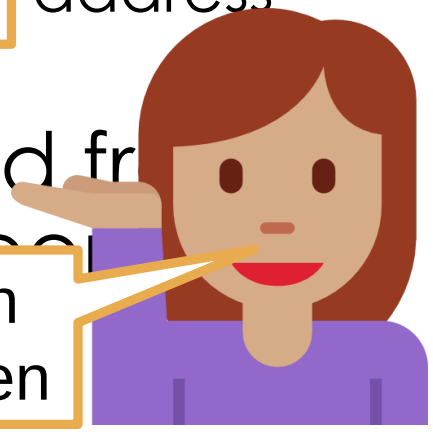
RULE 5: Value returned by a function can be used fr any way values of th

If function is returning a flo

If function is returning an in

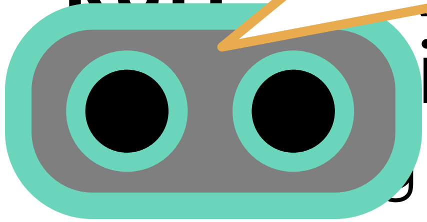
If function is returning an address, feel free to dereference that address

Remember, Mr C terminates a function the moment any return statement is seen



Th… …s 4

RU… …l values passed to … …sh
…inside that function
…g that value inside the func… …ginal value
Does not matter whether the value … address

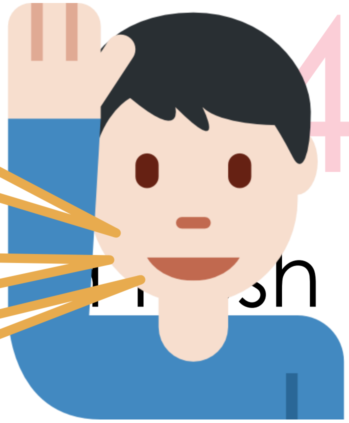


Yes, you may have multiple return statements but the clone will die the moment any one of them is seen

However, verify that the float returned is not negative

However, verify that the int gives an index within bounds

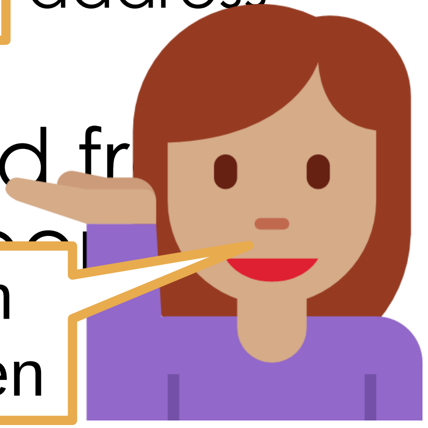
However, verify that the address returned isn't NULL



RULE 5: Value returned by a function can be used fr… any way values of th… data type could have bee…

If function is returning a flo…
If function is returning an int, feel free to use it as an array index
If function is returning an address, feel free to dereference that address

Remember, Mr C terminates a function the moment any return statement is seen



Th…s …4

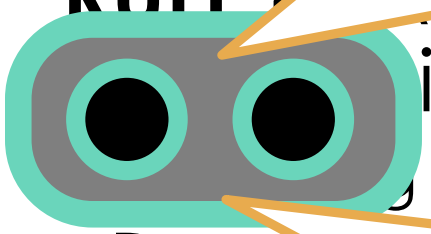
R…values passed to …sh
…inside that function
…g that value inside the func… …ginal value
…address

R… function can be used fr…
any way values of th…

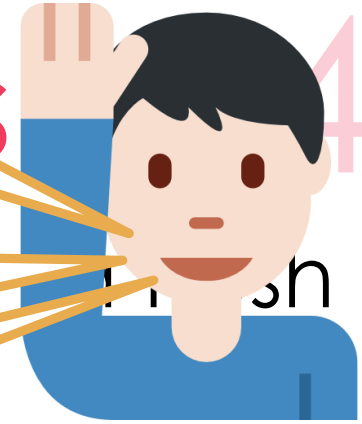
If function is returning a flo…
If function is returning an int, feel free to use it as an array index
If function is returning an address, feel free to dereference that address

Yes, you may have multiple return statements but the clone will die the moment any one of them is seen

However, verify that the float returned is not negative

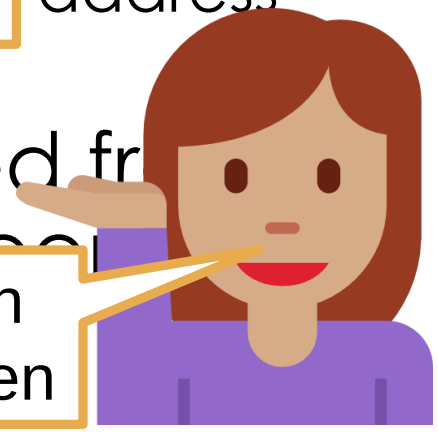


However, verify that the int gives an index within bounds



However, verify that the address returned isn't NULL

Careful, all return statements must return only one value, and that too of the type promised in the function



Remember, Mr C terminates a function the moment any return statement is seen

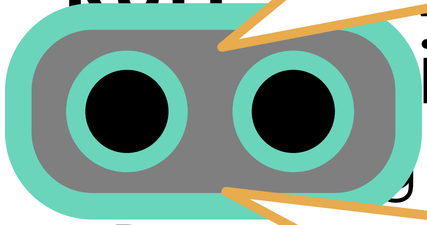
Yes, you may have multiple return statements but the clone will die the moment any one of them is seen

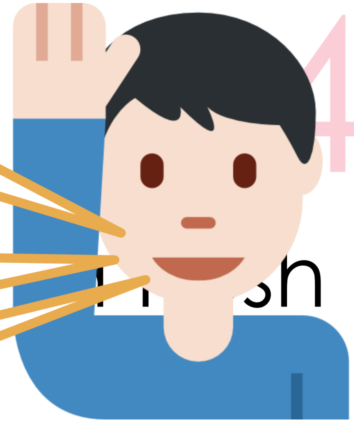
However, verify that the float returned is not negative

However, verify that the int gives an index within bounds

However, verify that the address returned isn't NULL

Careful, all return statements must return only one value, and that too of the type promised in the function





RU... values passed to ... inside that function

...g that value inside the func... ...ginal value

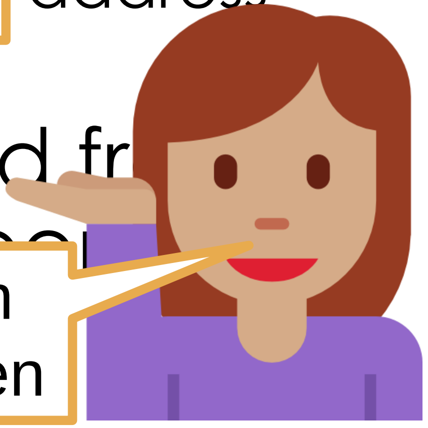
... address

RU... function can be used fr... any way values of th...

If function is returning a flo...

If function is returning an int, ...

If function is returning an address, feel free to dereference that address



Remember, Mr C terminates a function the moment any return statement is seen

RULE 6: All clones share the memory address space

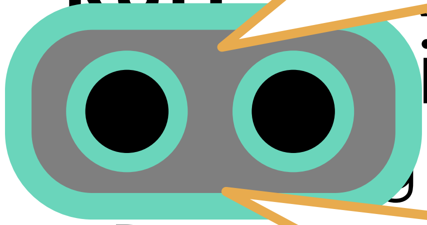
Th... s 4

RULE ... values passed to ... inside that function

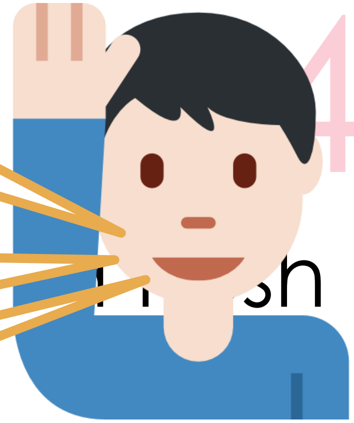
...g that value inside the func... ginal value

... address

Yes, you may have multiple return statements but the clone will die the moment any one of them is seen



Careful, all return statements must return only one value, and that too of the type promised in the function

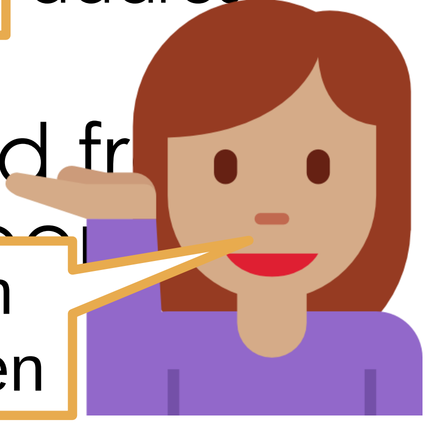


However, verify that the float returned is not negative

However, verify that the int gives an index within bounds

However, verify that the the address returned isn't NULL

RU... function can be used fr... any way values of that data type could have been



Remember, Mr C terminates a function the moment any return statement is seen

If function is returning a flo...

If function is returning an int, feel free to use it as an array index

If function is returning an address, feel free to dereference that address

RULE 6: All clones share the memory address space

Let us look at this rule more closely

RULE 6: the address rule

RULE 6: the address rule

We have seen that the clones do not care what names other clones have given to variables – all passed values are copied

RULE 6: the address rule

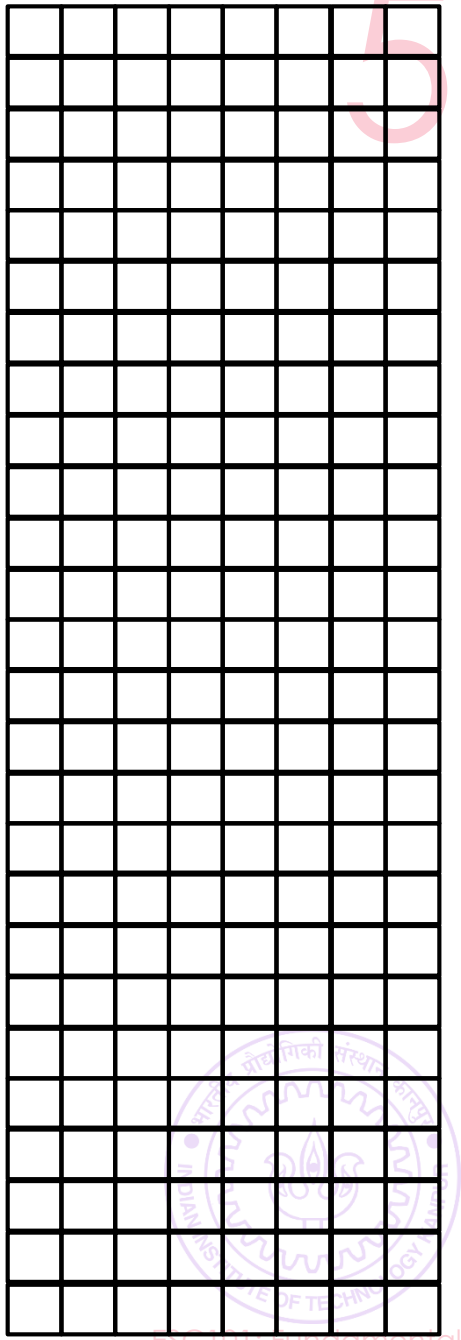
We have seen that the clones do not care what names other clones have given to variables – all passed values are copied

However, all clones work with the same set of memory addresses

RULE 6: the address rule

We have seen that the clones do not care what names other clones have given to variables – all passed values are copied

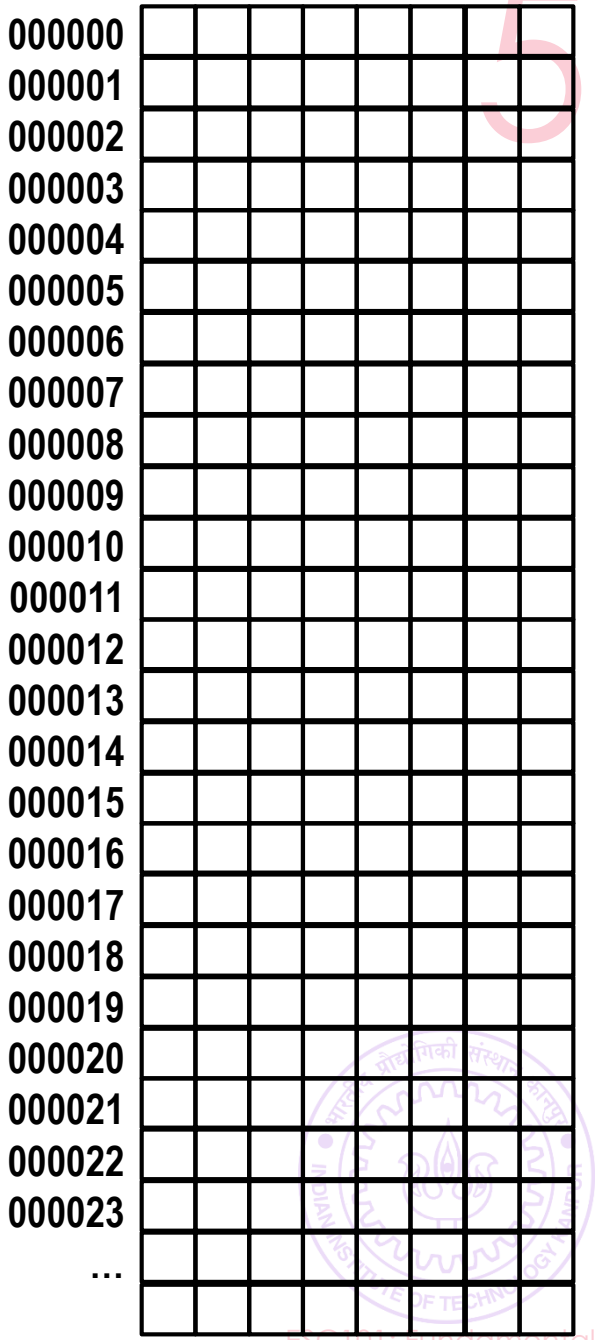
However, all clones work with the same set of memory addresses



RULE 6: the address rule

We have seen that the clones do not care what names other clones have given to variables – all passed values are copied

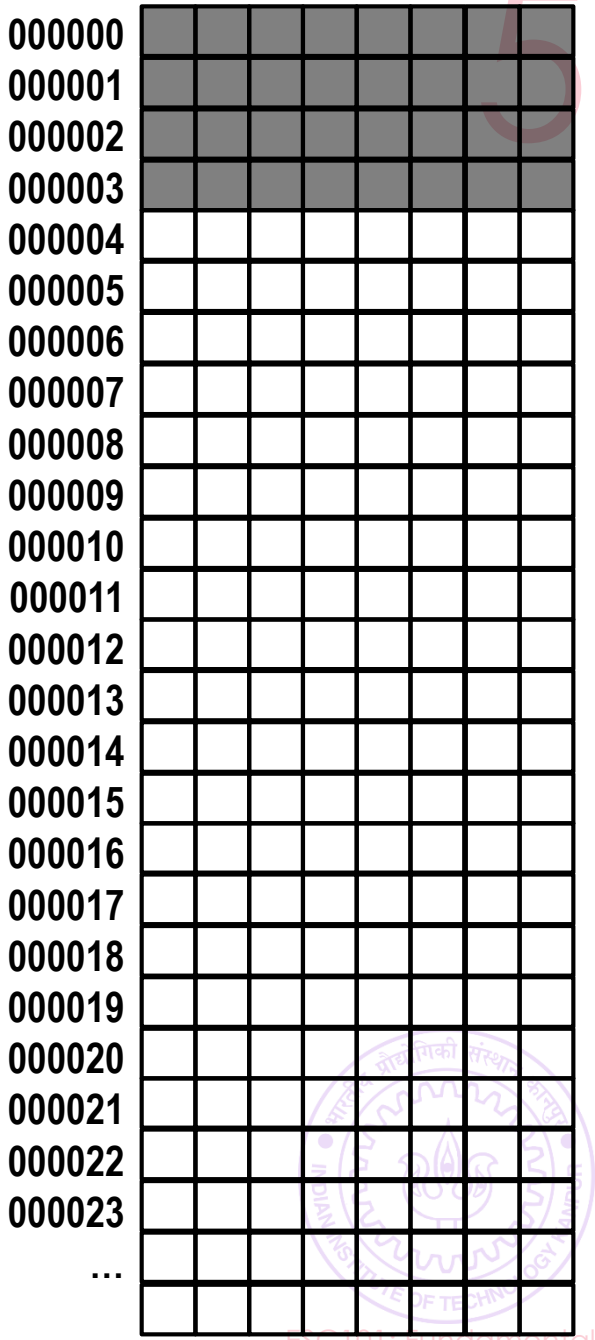
However, all clones work with the same set of memory addresses



RULE 6: the address rule

We have seen that the clones do not care what names other clones have given to variables – all passed values are copied

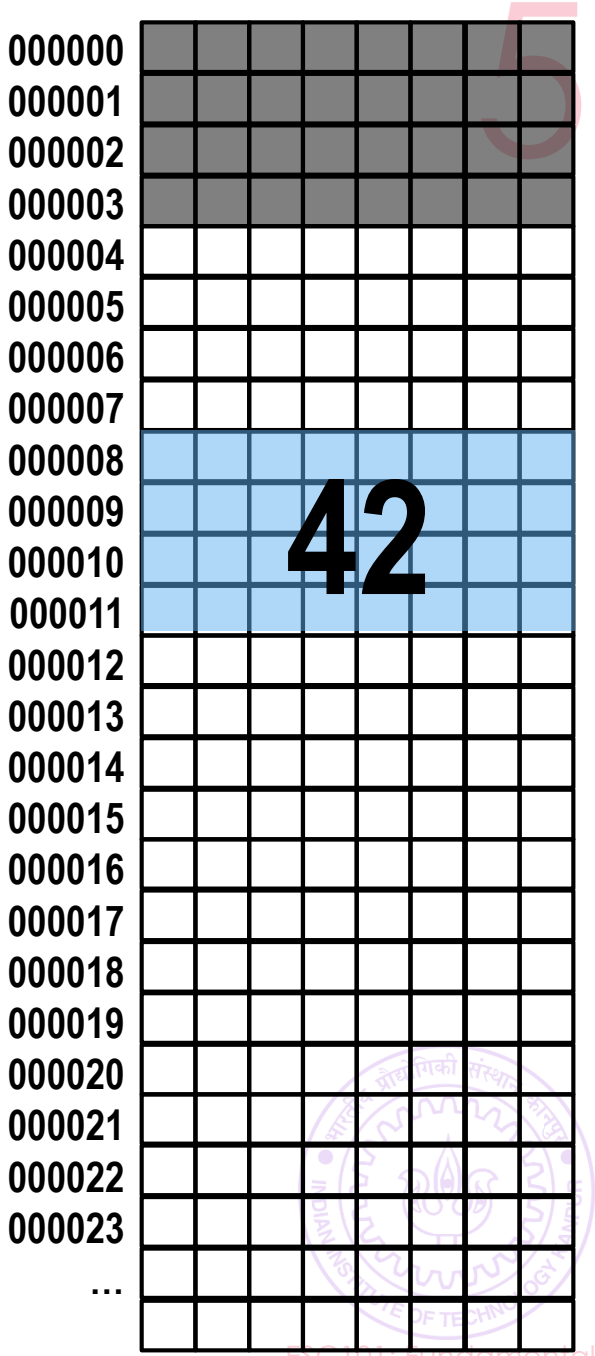
However, all clones work with the same set of memory addresses



RULE 6: the address rule

We have seen that the clones do not care what names other clones have given to variables – all passed values are copied

However, all clones work with the same set of memory addresses

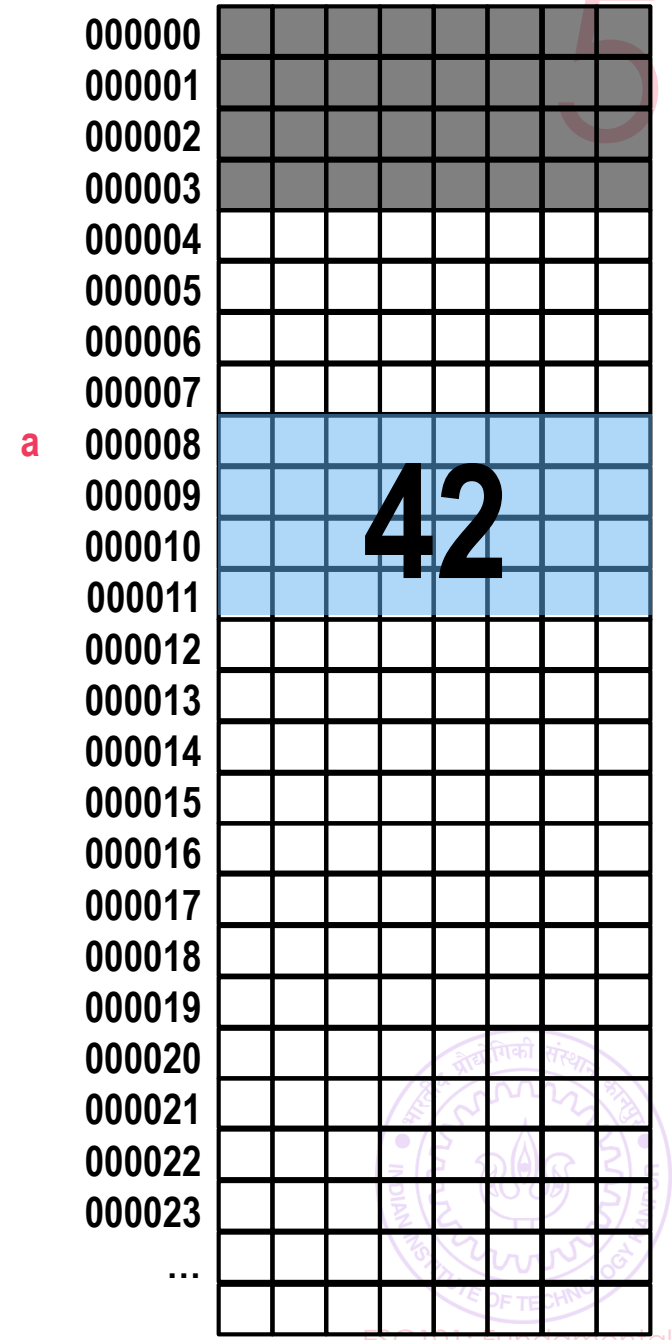


RULE 6: the address rule

We have seen that the clones do not care what names other clones have given to variables – all passed values are copied

However, all clones work with the same set of memory addresses

Consider an address 000008 – no matter which clone tries to read from, or write to, address 000008, they will all do so from the exact same address



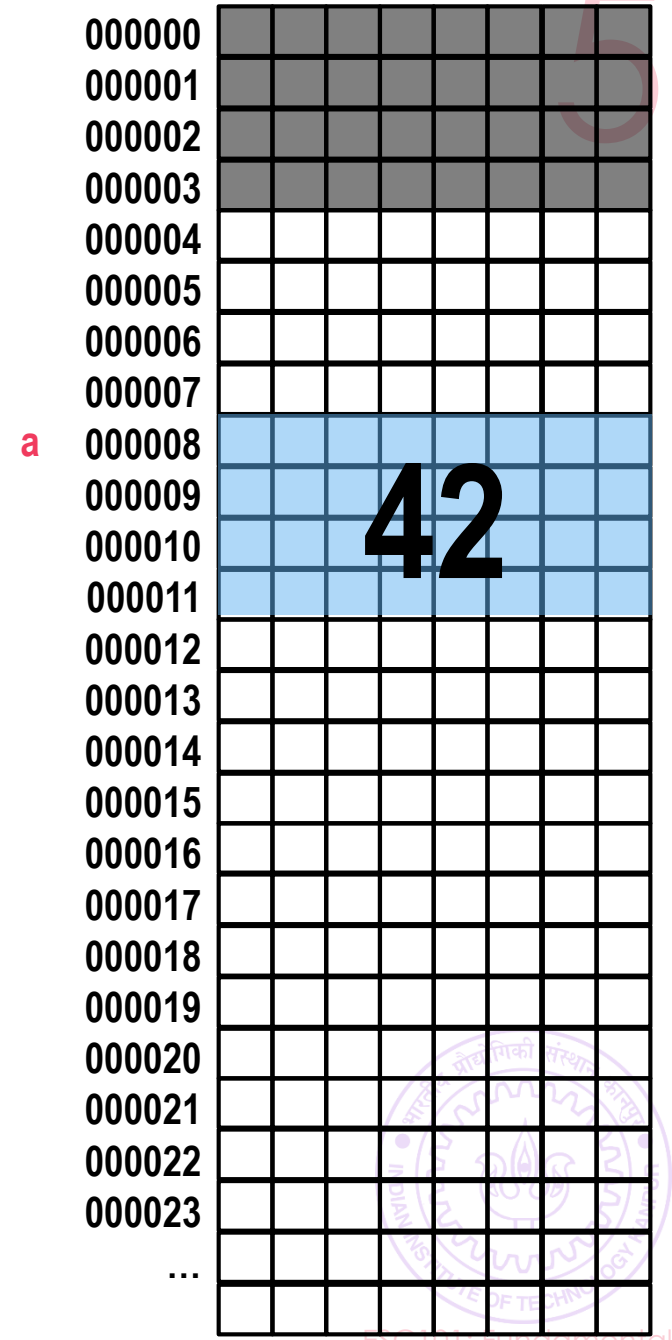
RULE 6: the address rule

We have seen that the clones do not care what names other clones have given to variables – all passed values are copied

However, all clones work with the same set of memory addresses

Consider an address 000008 – no matter which clone tries to read from, or write to, address 000008, they will all do so from the exact same address

Will exploit this feature very soon!



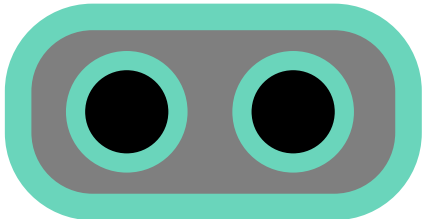
RULE 6: the address rule

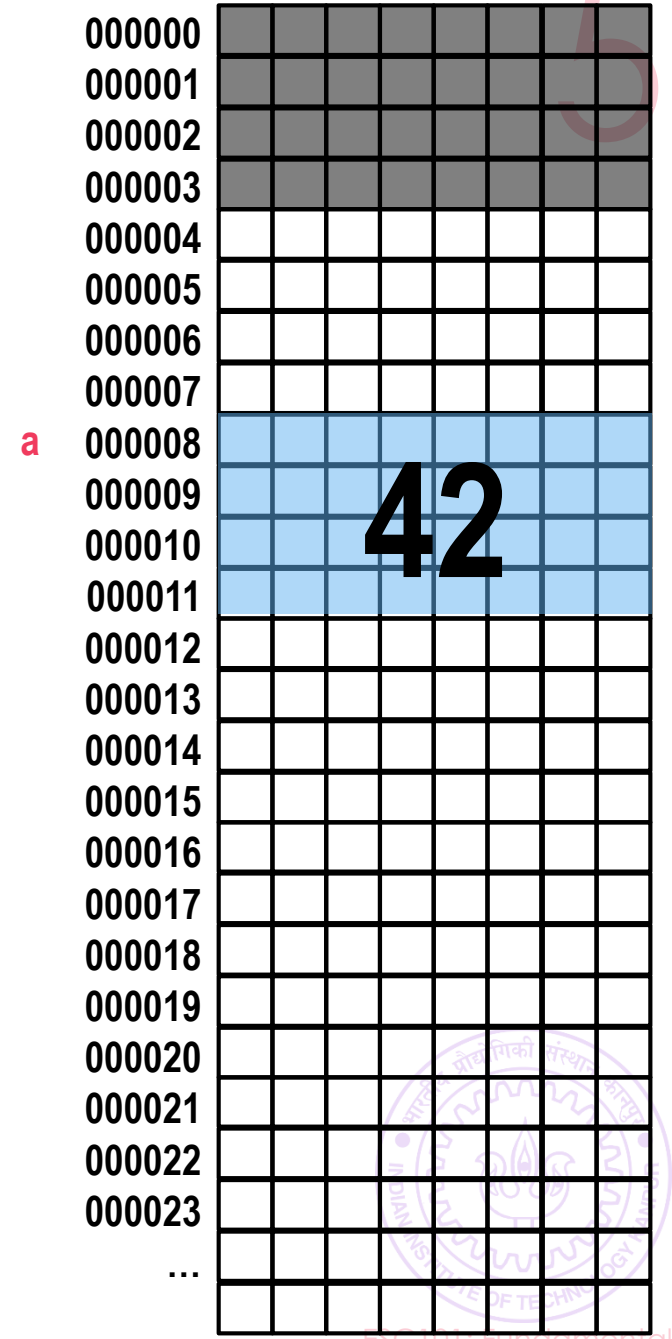
We have seen that the clones do not care what names other clones have given to variables – all passed values are copied

However, all clones work with the same set of memory addresses

Consider an address 000008 – no matter which clone tries to read from, or write to, address 000008, they will all do so from the exact same address

Will exploit this feature very soon!





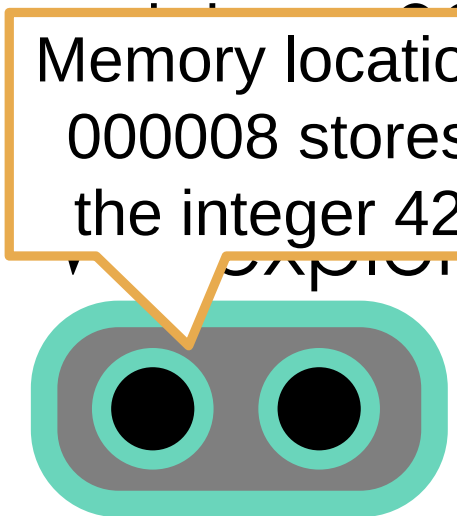
RULE 6: the address rule

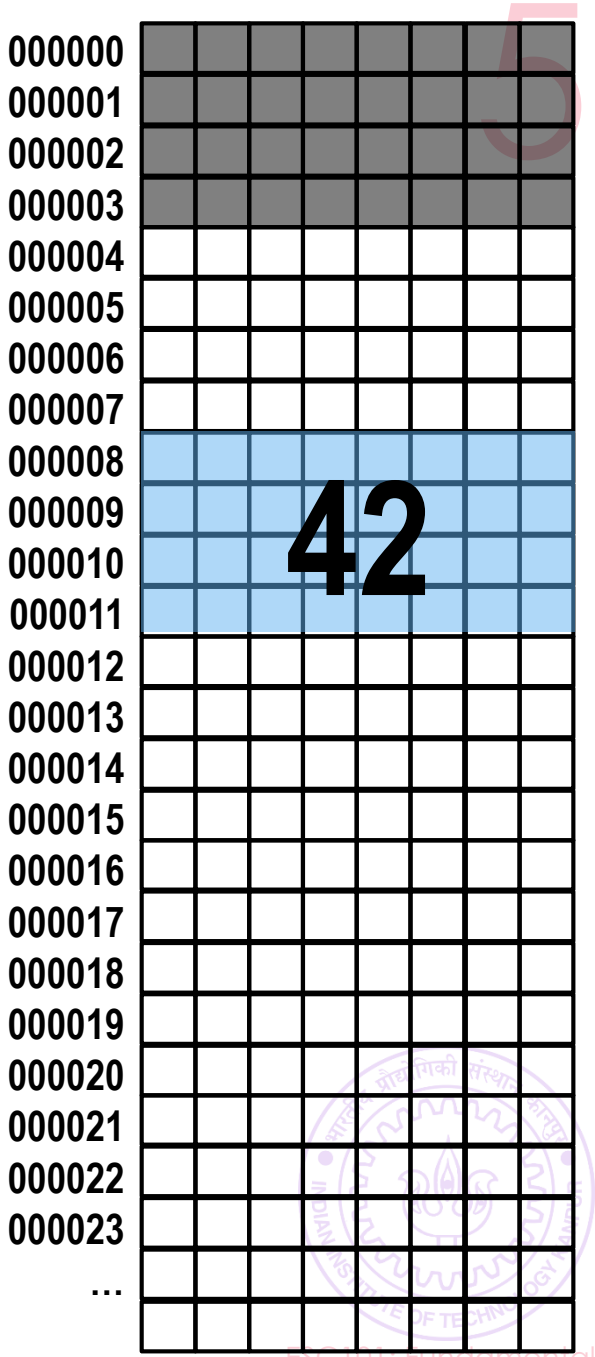
We have seen that the clones do not care what names other clones have given to variables – all passed values are copied

However, all clones work with the same set of memory addresses

Consider an address 000008 – no matter which clone tries to read from, or write to, [illegible]008, they will all do so from the [illegible] address

[illegible]xplain this feature very soon!





RULE 6: the address rule

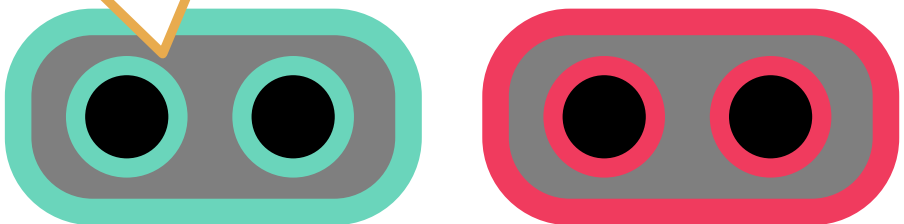
We have seen that the clones do not care what names other clones have given to variables – all passed values are copied

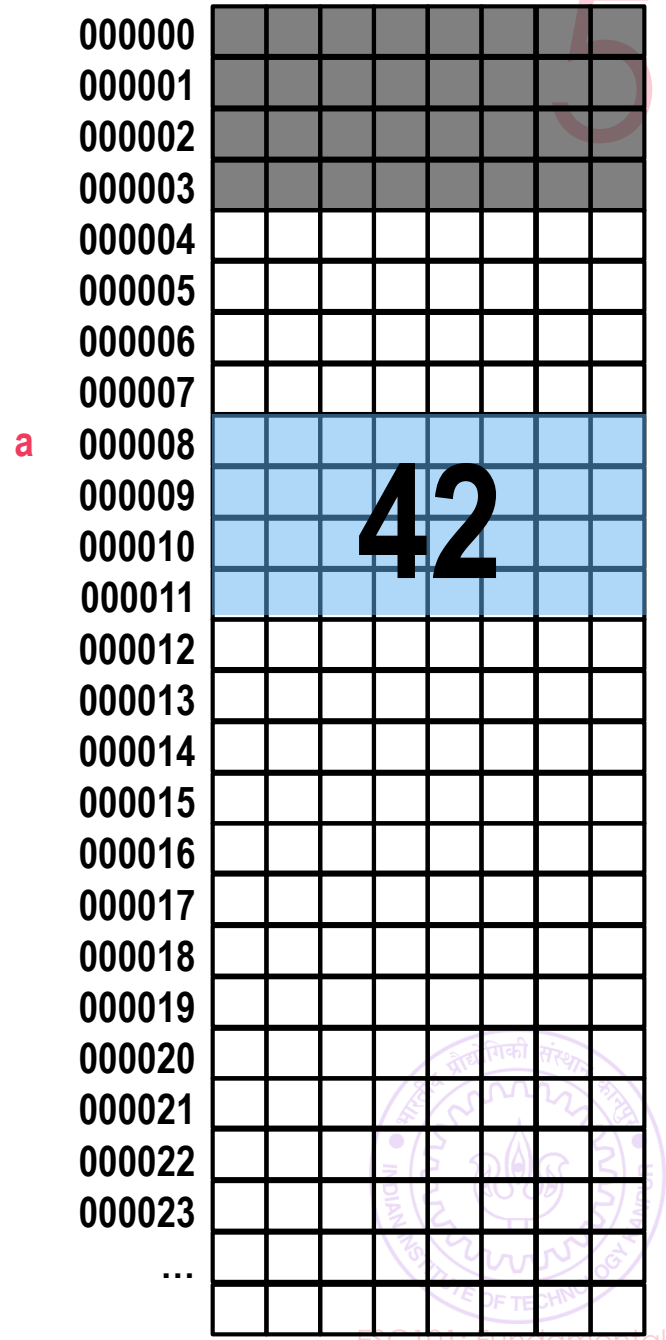
However, all clones work with the same set of memory addresses

Consider an address 000008 – no matter which clone tries to read from, or write to, [illegible]008, they will all do so from the [illegible] address

[illegible]his feature very soon!

Memory location 000008 stores the integer 42





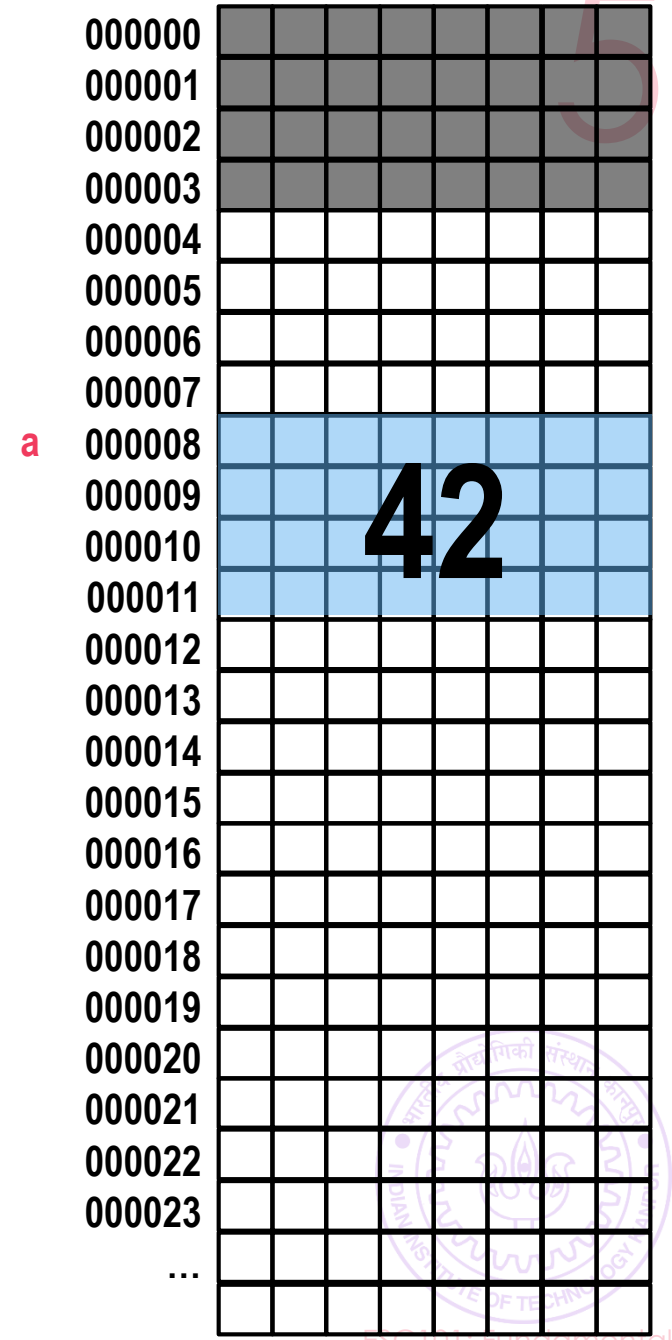
RULE 6: the address rule

We have seen that the clones do not care what names other clones have given to variables – all passed values are copied

However, all clones work with the same set of memory addresses

Consider an address 000008 – no matter which clone tries to read from, or write to, ll all do so from the

very soon!



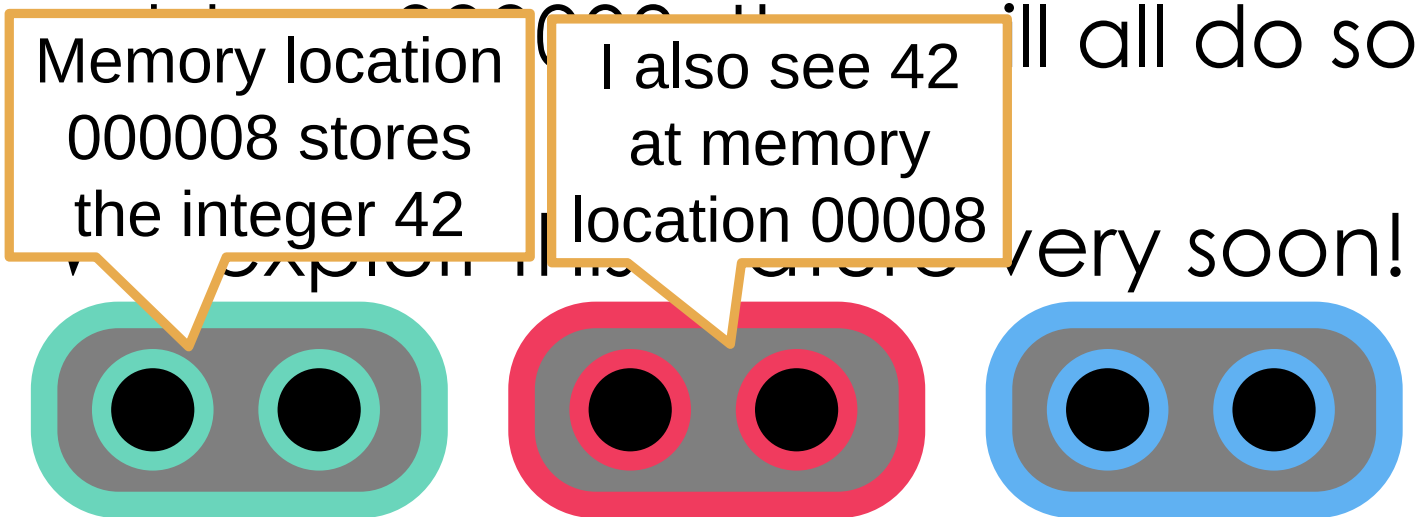
RULE 6: the address rule

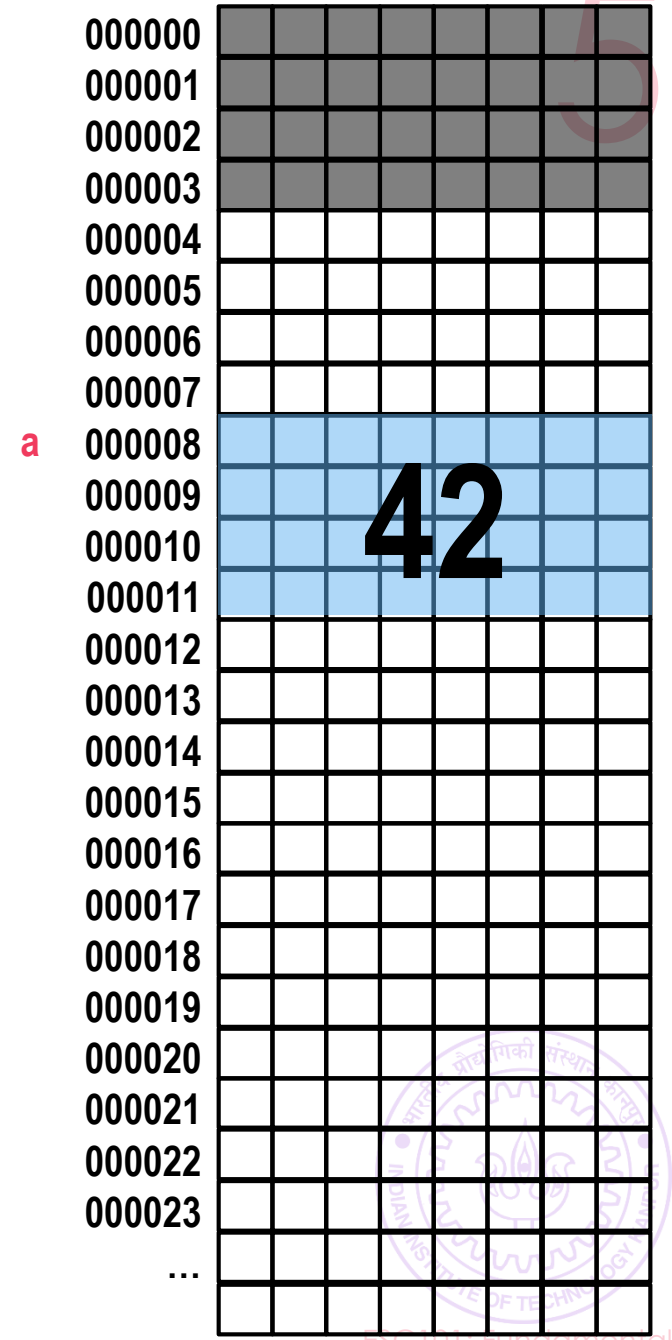
We have seen that the clones do not care what names other clones have given to variables – all passed values are copied

However, all clones work with the same set of memory addresses

Consider an address 000008 – no matter which clone tries to read from, or write to, ll all do so from the

very soon!



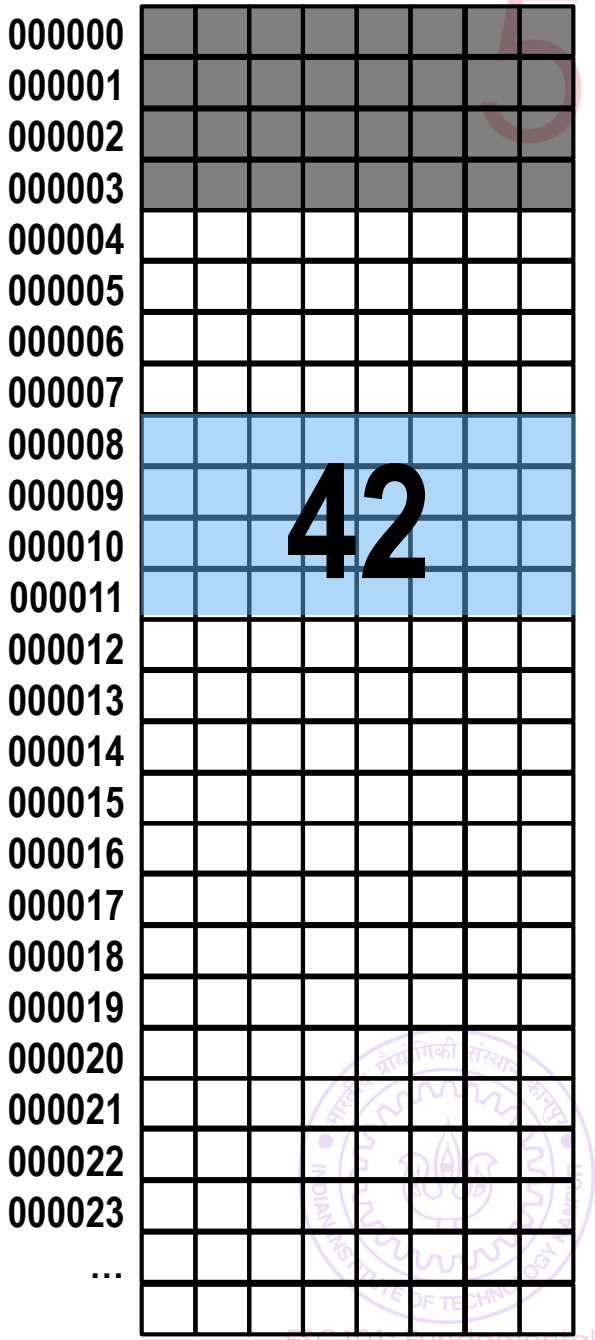


RULE 6: the address rule

We have seen that the clones do not care what names other clones have given to variables – all passed values are copied

However, all clones work with the same set of memory addresses

Consider an address 000008 – no matter which clone tries to read from, or write to, ll all do so from the

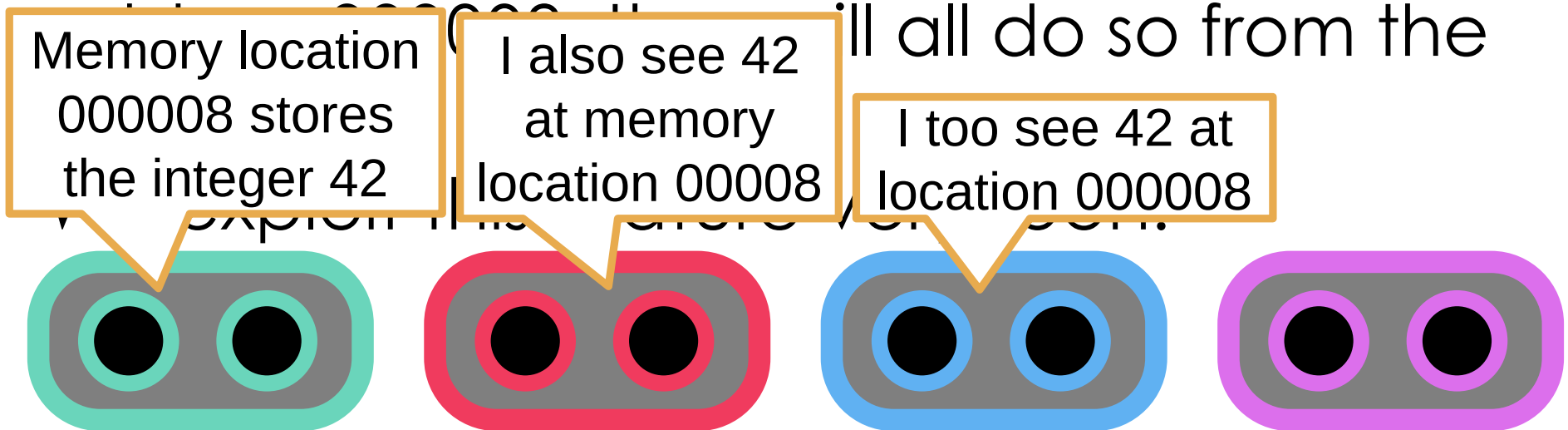


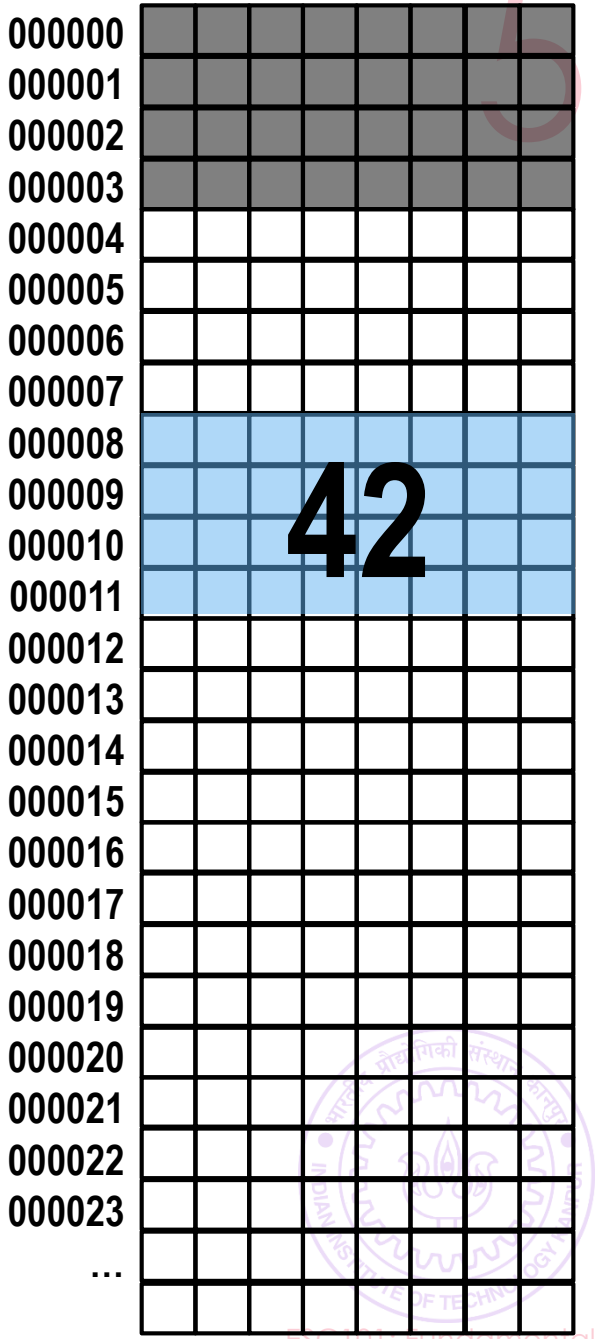
RULE 6: the address rule

We have seen that the clones do not care what names other clones have given to variables – all passed values are copied

However, all clones work with the same set of memory addresses

Consider an address 000008 – no matter which clone tries to read from, or write to, ll all do so from the



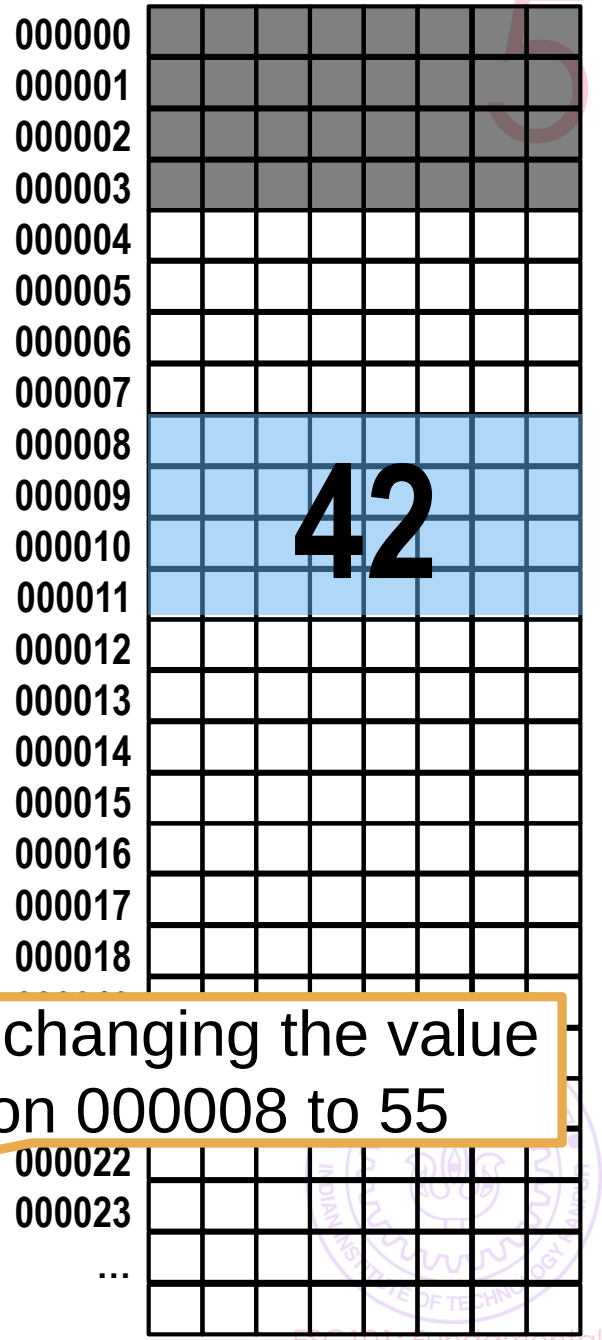


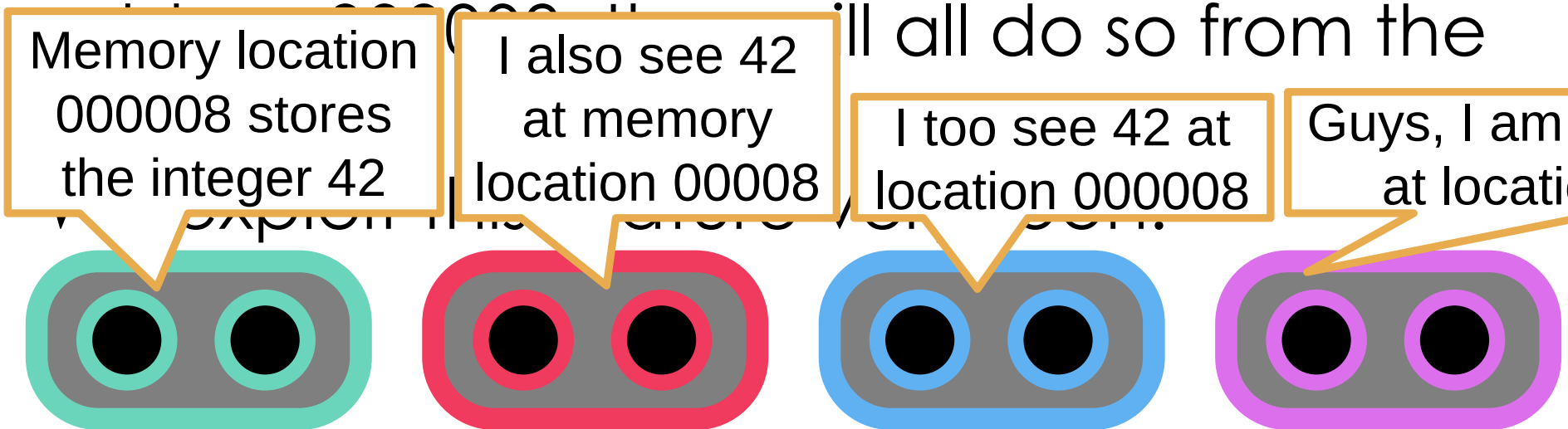
RULE 6: the address rule

We have seen that the clones do not care what names other clones have given to variables – all passed values are copied

However, all clones work with the same set of memory addresses

Consider an address 000008 – no matter which clone tries to read from, or write to, ll all do so from the



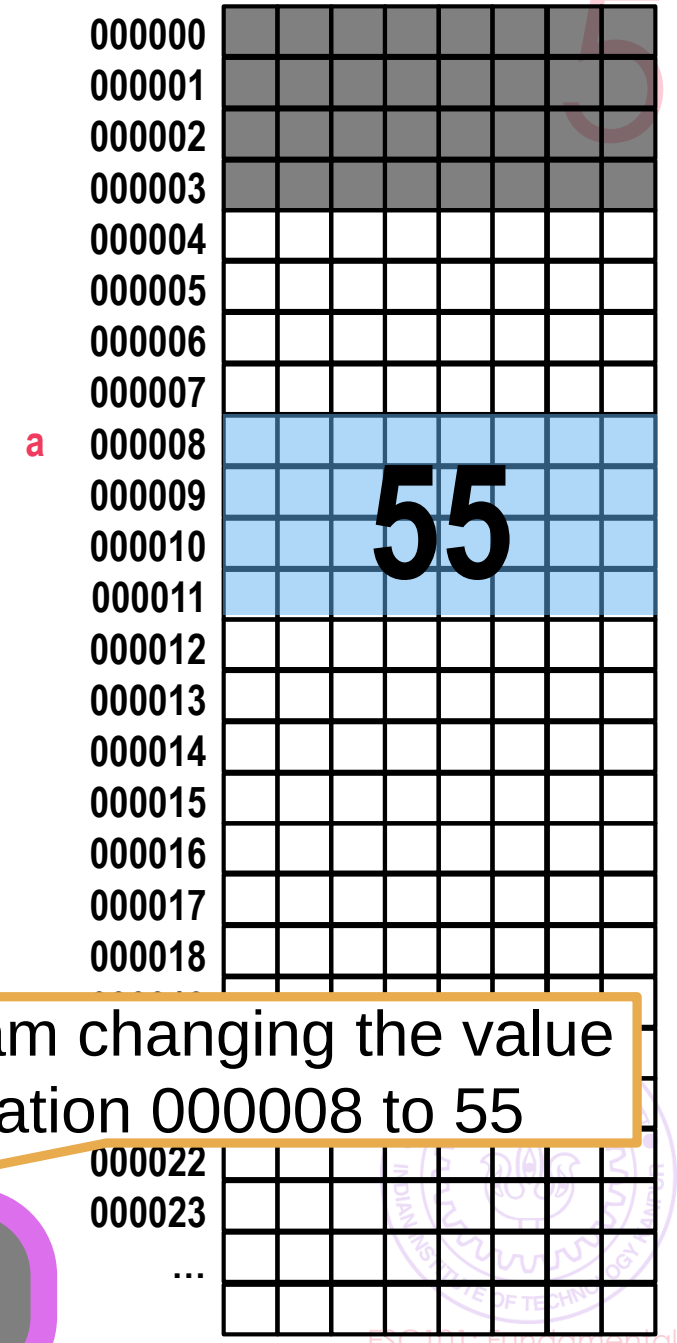


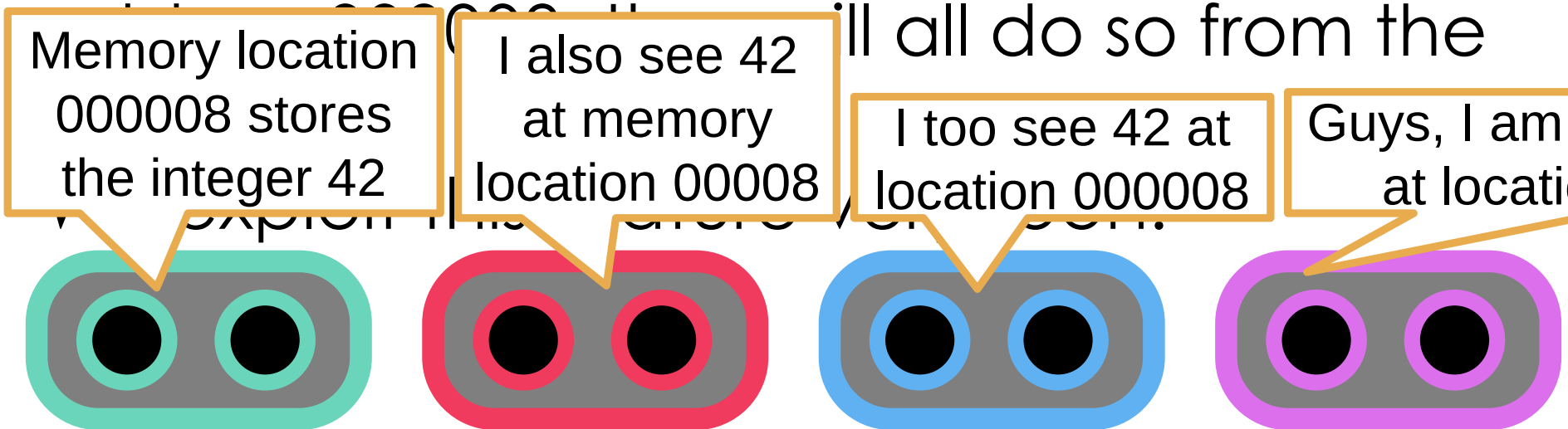
RULE 6: the address rule

We have seen that the clones do not care what names other clones have given to variables – all passed values are copied

However, all clones work with the same set of memory addresses

Consider an address 000008 – no matter which clone tries to read from, or write to, ll all do so from the



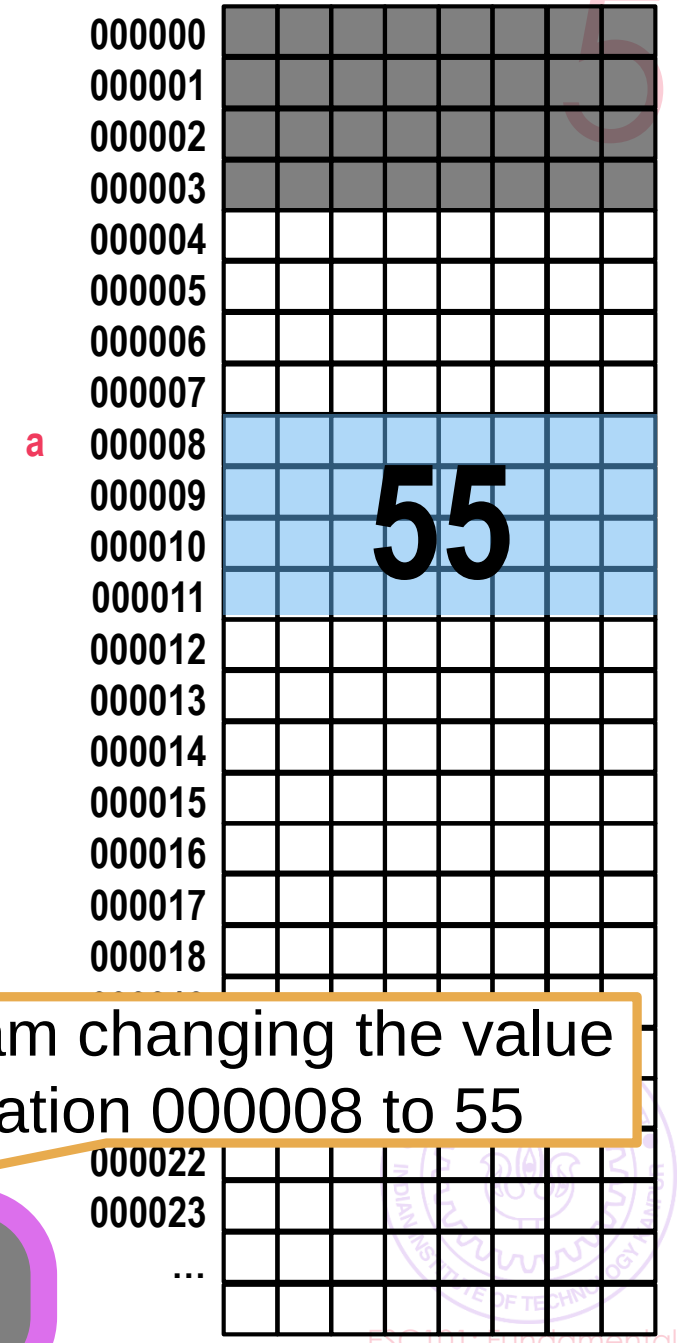


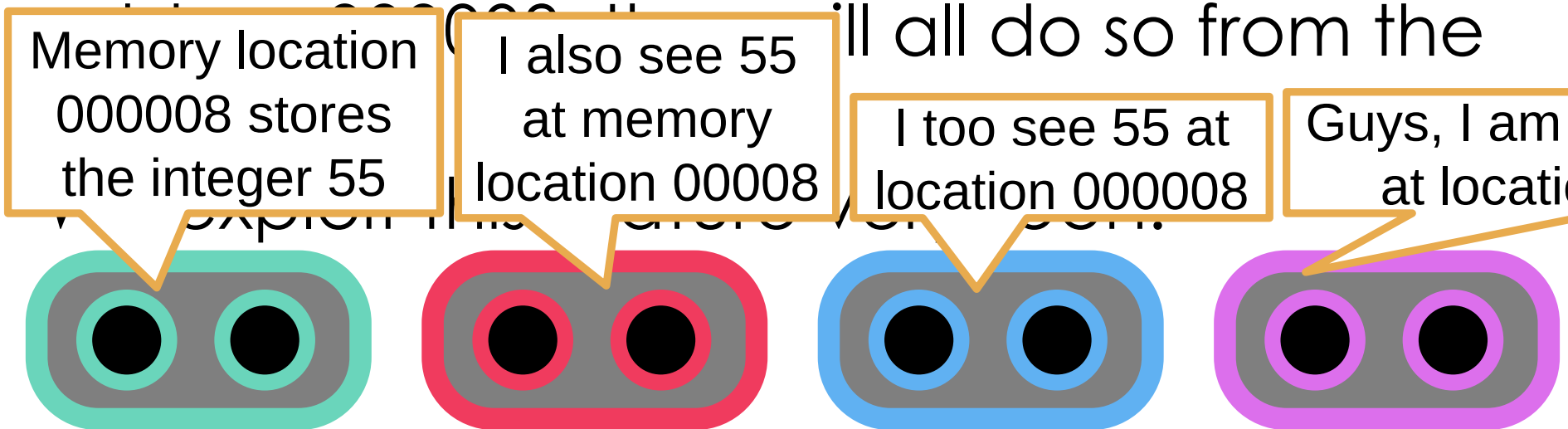
RULE 6: the address rule

We have seen that the clones do not care what names other clones have given to variables – all passed values are copied

However, all clones work with the same set of memory addresses

Consider an address 000008 – no matter which clone tries to read from, or write to, [illegible] will all do so from the [illegible]





A word of caution

A word of caution

When you go on internet websites or read books, you will find several terms such as *pass-by-value, pass-by-reference, pass-by-array,* and *pass-by-pointer*.

A word of caution

When you go on internet websites or read books, you will find several terms such as *pass-by-value, pass-by-reference, pass-by-array,* and *pass-by-pointer*.

If you follow the 6 rules of functions you will **not** have to worry about these pass-by rules separately

A word of caution

When you go on internet websites or read books, you will find several terms such as *pass-by-value, pass-by-reference, pass-by-array,* and *pass-by-pointer*.

If you follow the 6 rules of functions you will **not** have to worry about these pass-by rules separately

All these pass-by cases will follow automatically

A word of caution

When you go on internet websites or read books, you will find several terms such as *pass-by-value, pass-by-reference, pass-by-array,* and *pass-by-pointer*.

If you follow the 6 rules of functions you will **not** have to worry about these pass-by rules separately

All these pass-by cases will follow automatically

We will take examples to understand each of them

Passing simple variables/expressions

Passing simple variables/expressions

This is the case when the input to the function is either a variable (Rule 1) or an expression (Rule 2)

Passing simple variables/expressions

This is the case when the input to the function is either a variable (Rule 1) or an expression (Rule 2)

- Rule 4 will always apply no matter what is passed as input

Passing simple variables/expressions

This is the case when the input to the function is either a variable (Rule 1) or an expression (Rule 2)

Rule 4 will always apply no matter what is passed as input

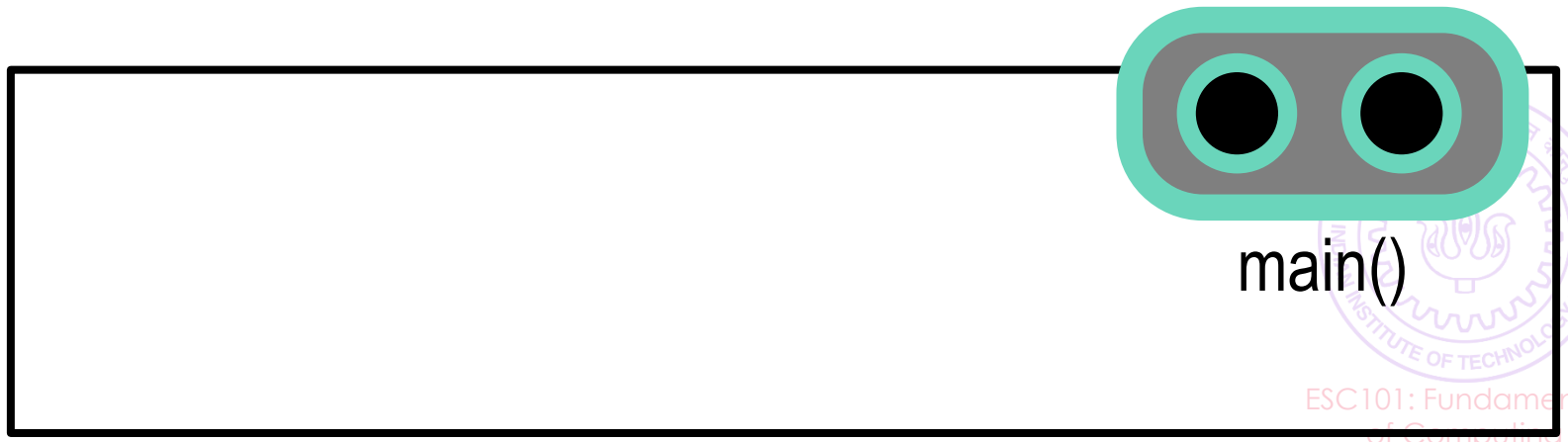
```
int inc(int a){
    return -a;
}
int main(void){
    int a = 2;
    printf("%d", neg(a));
    printf("%d", neg(4*2);
    return 0;
}
```

Passing simple variables/expressions

This is the case when the input to the function is either a variable (Rule 1) or an expression (Rule 2)

Rule 4 will always apply no matter what is passed as input

```
int inc(int a){
    return -a;
}
int main(void){
    int a = 2;
    printf("%d", neg(a));
    printf("%d", neg(4*2);
    return 0;
}
```

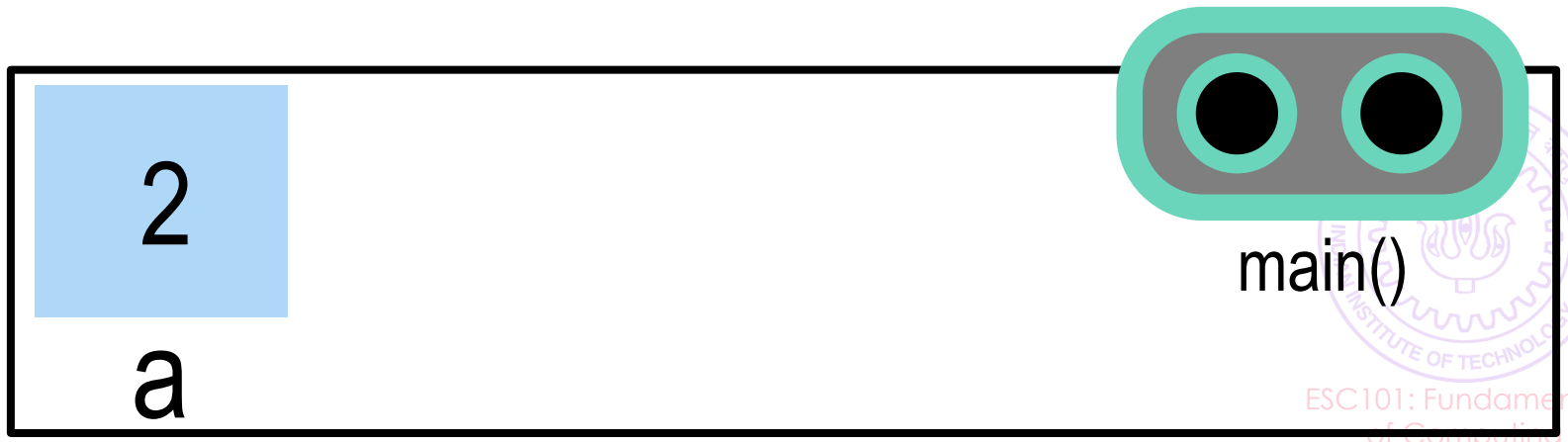


Passing simple variables/expressions

This is the case when the input to the function is either a variable (Rule 1) or an expression (Rule 2)

Rule 4 will always apply no matter what is passed as input

```
int inc(int a){
    return -a;
}
int main(void){
    int a = 2;
    printf("%d", neg(a));
    printf("%d", neg(4*2);
    return 0;
}
```

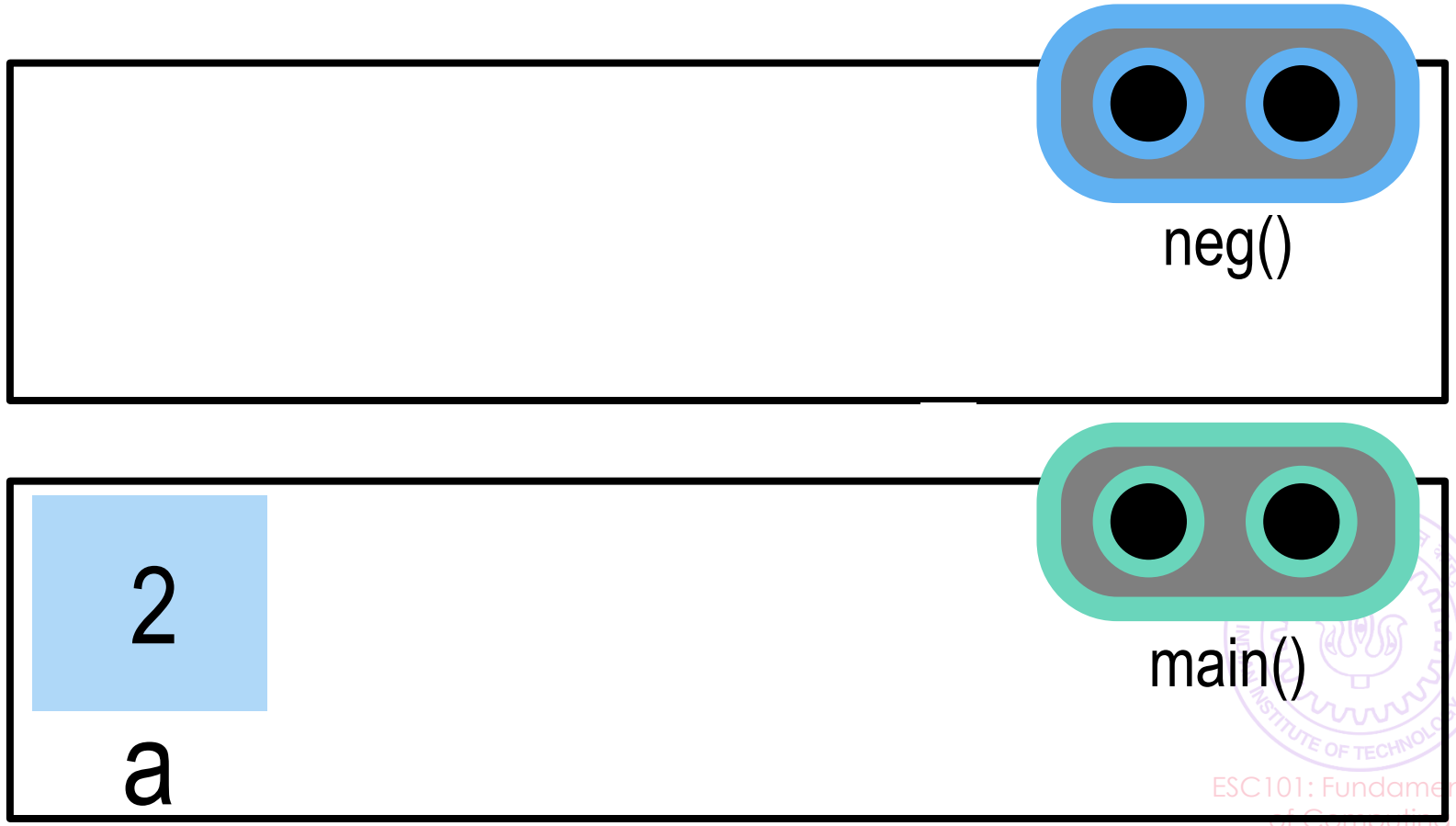


Passing simple variables/expressions

This is the case when the input to the function is either a variable (Rule 1) or an expression (Rule 2)

Rule 4 will always apply no matter what is passed as input

```
int inc(int a){
    return -a;
}
int main(void){
    int a = 2;
    printf("%d", neg(a));
    printf("%d", neg(4*2);
    return 0;
}
```

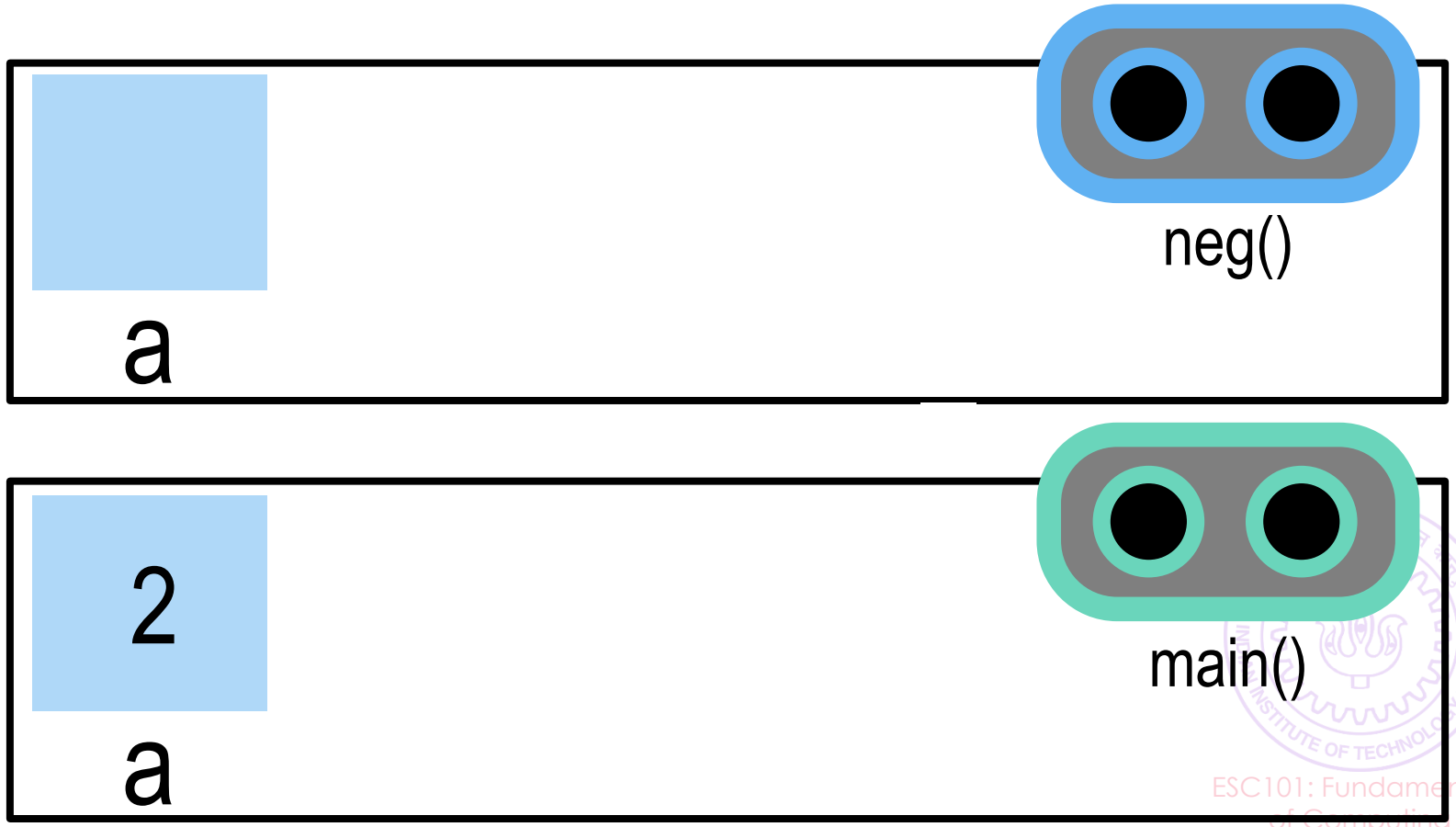


Passing simple variables/expressions

This is the case when the input to the function is either a variable (Rule 1) or an expression (Rule 2)

Rule 4 will always apply no matter what is passed as input

```
int inc(int a){
    return -a;
}
int main(void){
    int a = 2;
    printf("%d", neg(a));
    printf("%d", neg(4*2);
    return 0;
}
```

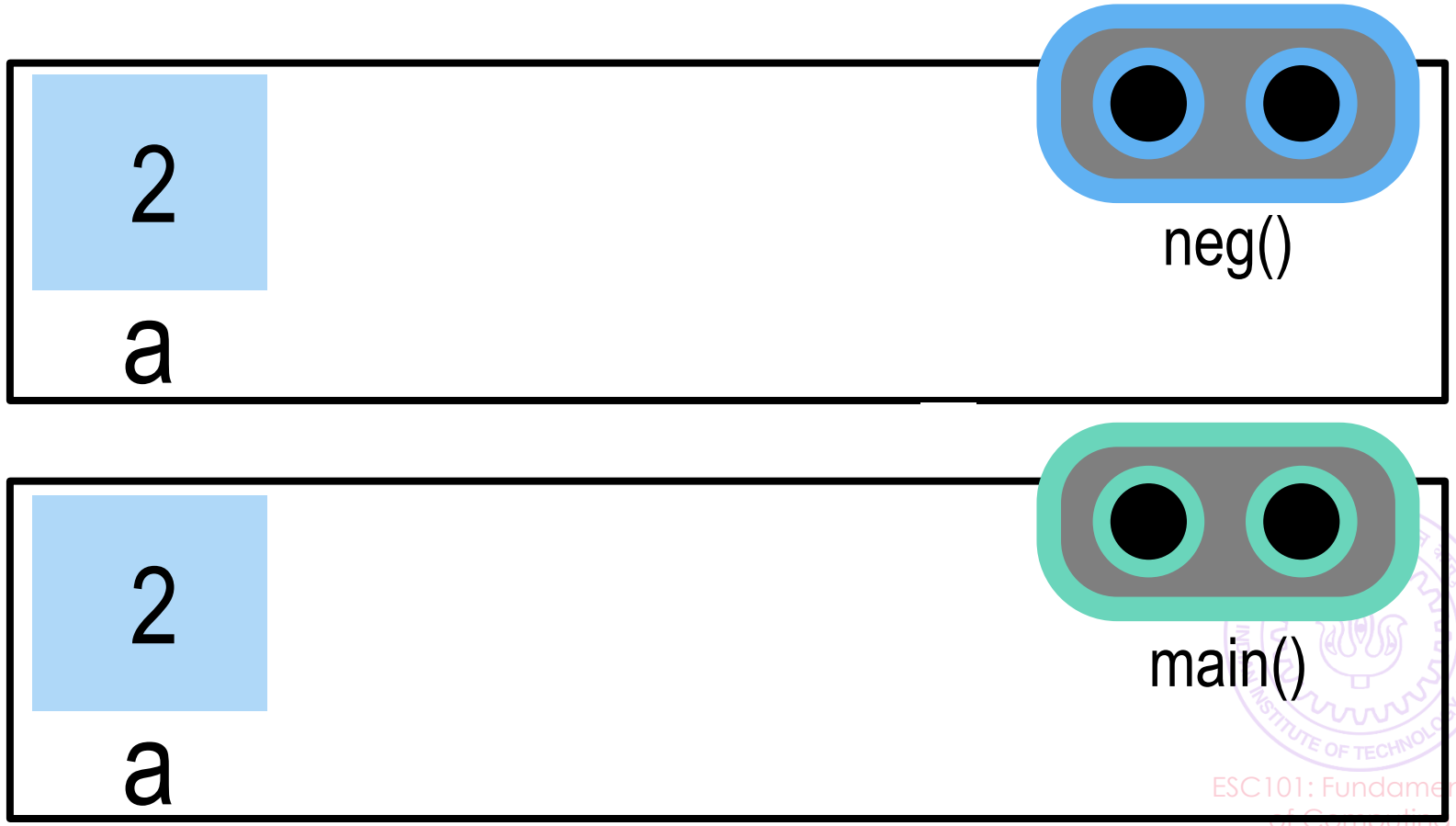


Passing simple variables/expressions

This is the case when the input to the function is either a variable (Rule 1) or an expression (Rule 2)

Rule 4 will always apply no matter what is passed as input

```
int inc(int a){
    return -a;
}
int main(void){
    int a = 2;
    printf("%d", neg(a));
    printf("%d", neg(4*2);
    return 0;
}
```

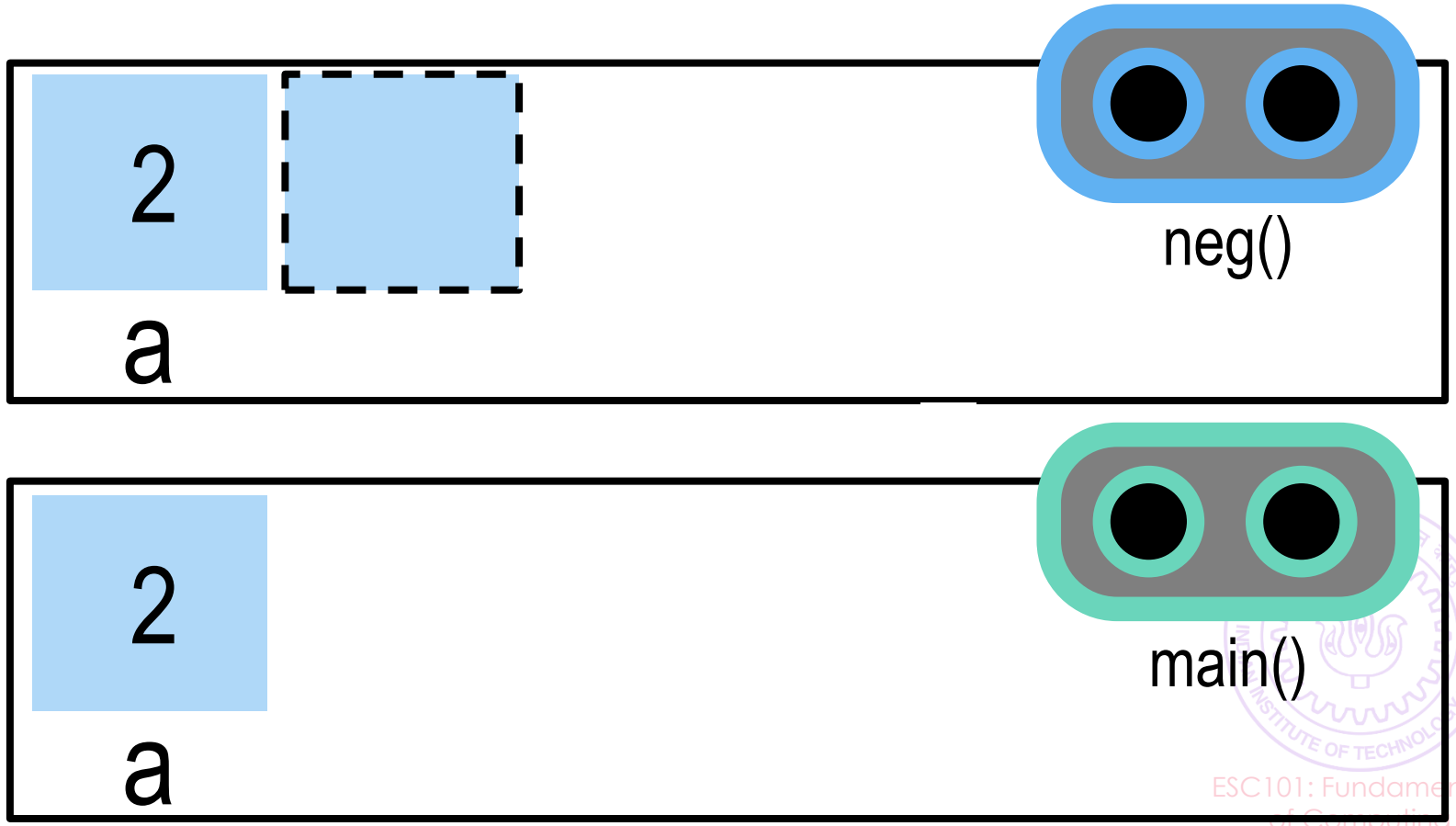


Passing simple variables/expressions

This is the case when the input to the function is either a variable (Rule 1) or an expression (Rule 2)

Rule 4 will always apply no matter what is passed as input

```
int inc(int a){
    return -a;
}
int main(void){
    int a = 2;
    printf("%d", neg(a));
    printf("%d", neg(4*2);
    return 0;
}
```

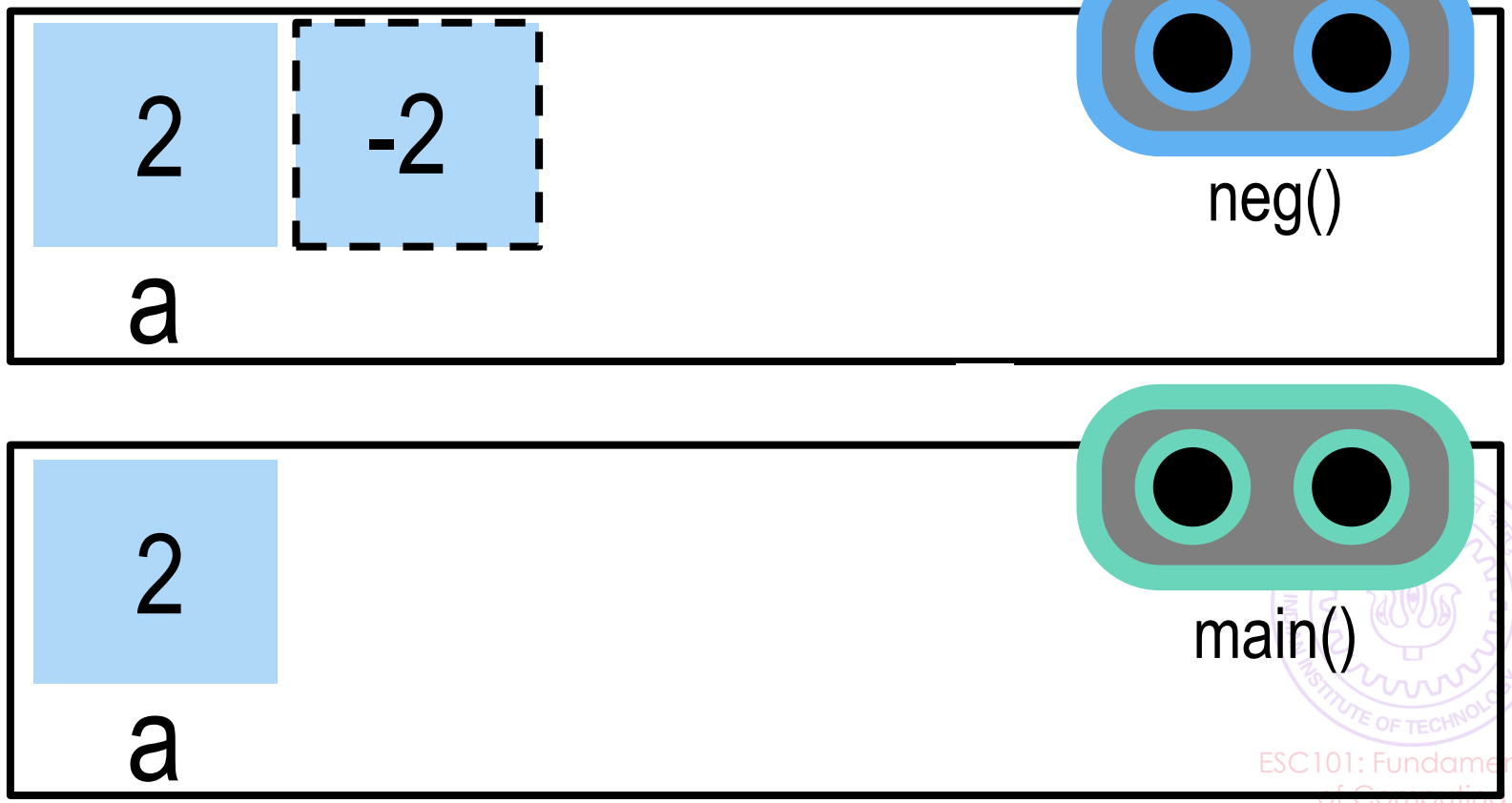


Passing simple variables/expressions

This is the case when the input to the function is either a variable (Rule 1) or an expression (Rule 2)

Rule 4 will always apply no matter what is passed as input

```
int inc(int a){
    return -a;
}
int main(void){
    int a = 2;
    printf("%d", neg(a));
    printf("%d", neg(4*2);
    return 0;
}
```

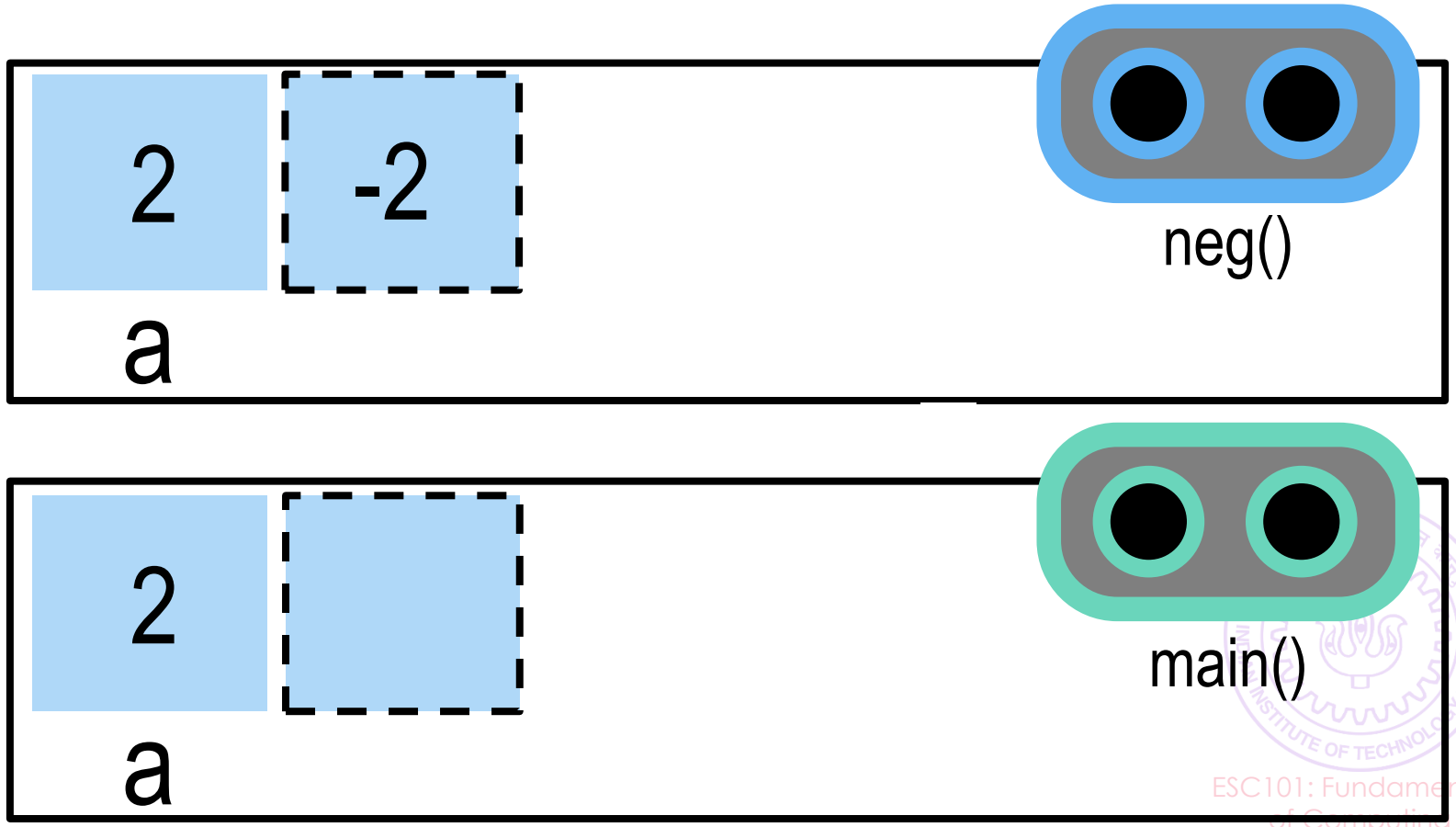


Passing simple variables/expressions

This is the case when the input to the function is either a variable (Rule 1) or an expression (Rule 2)

Rule 4 will always apply no matter what is passed as input

```
int inc(int a){
    return -a;
}
int main(void){
    int a = 2;
    printf("%d", neg(a));
    printf("%d", neg(4*2);
    return 0;
}
```

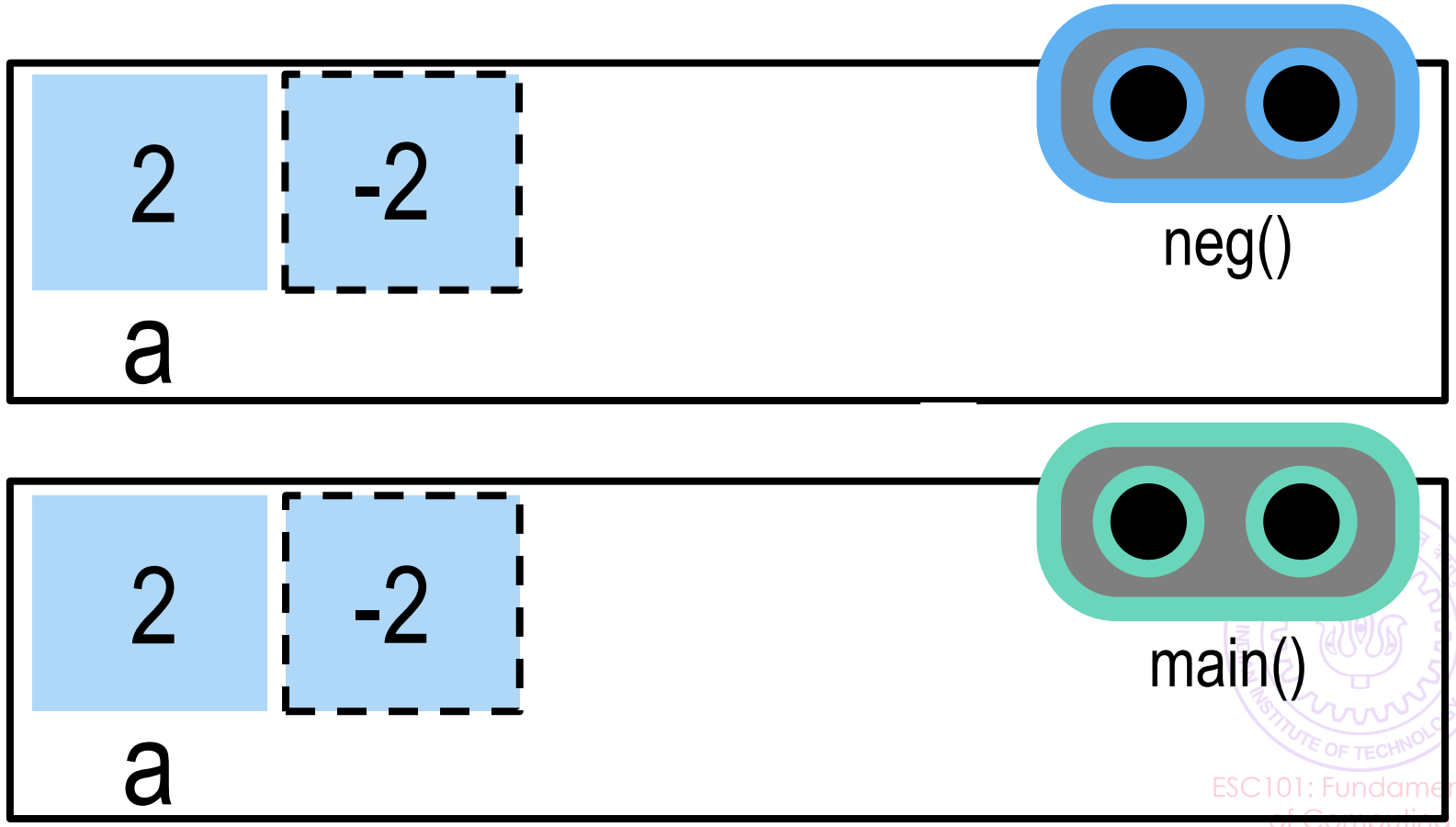


Passing simple variables/expressions

This is the case when the input to the function is either a variable (Rule 1) or an expression (Rule 2)

Rule 4 will always apply no matter what is passed as input

```
int inc(int a){
    return -a;
}
int main(void){
    int a = 2;
    printf("%d", neg(a));
    printf("%d", neg(4*2);
    return 0;
}
```

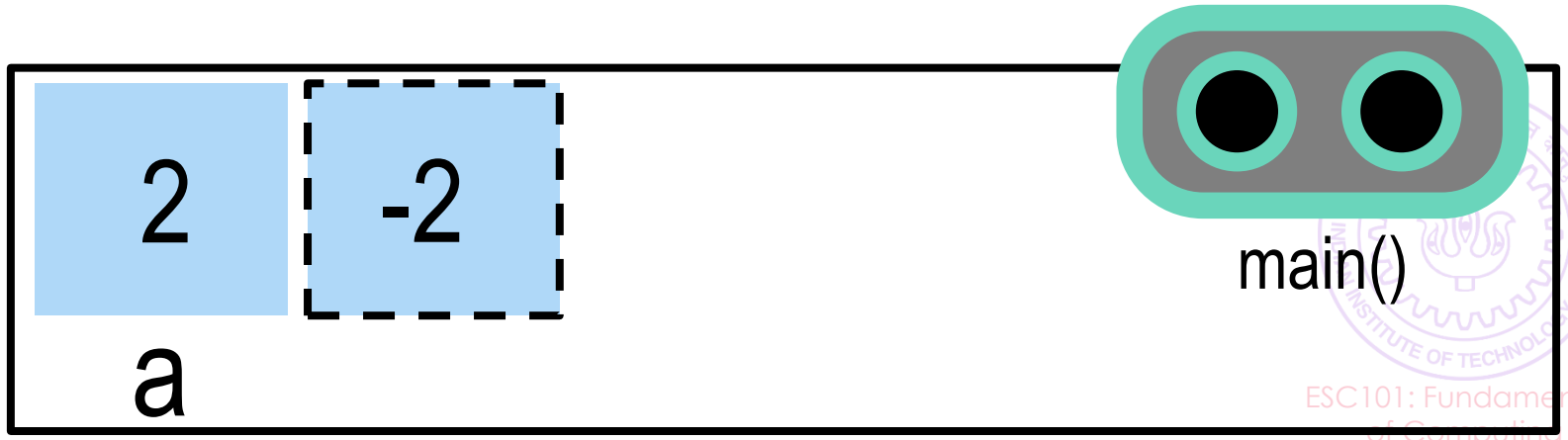


Passing simple variables/expressions

This is the case when the input to the function is either a variable (Rule 1) or an expression (Rule 2)

Rule 4 will always apply no matter what is passed as input

```
int inc(int a){
    return -a;
}
int main(void){
    int a = 2;
    printf("%d", neg(a));
    printf("%d", neg(4*2);
    return 0;
}
```

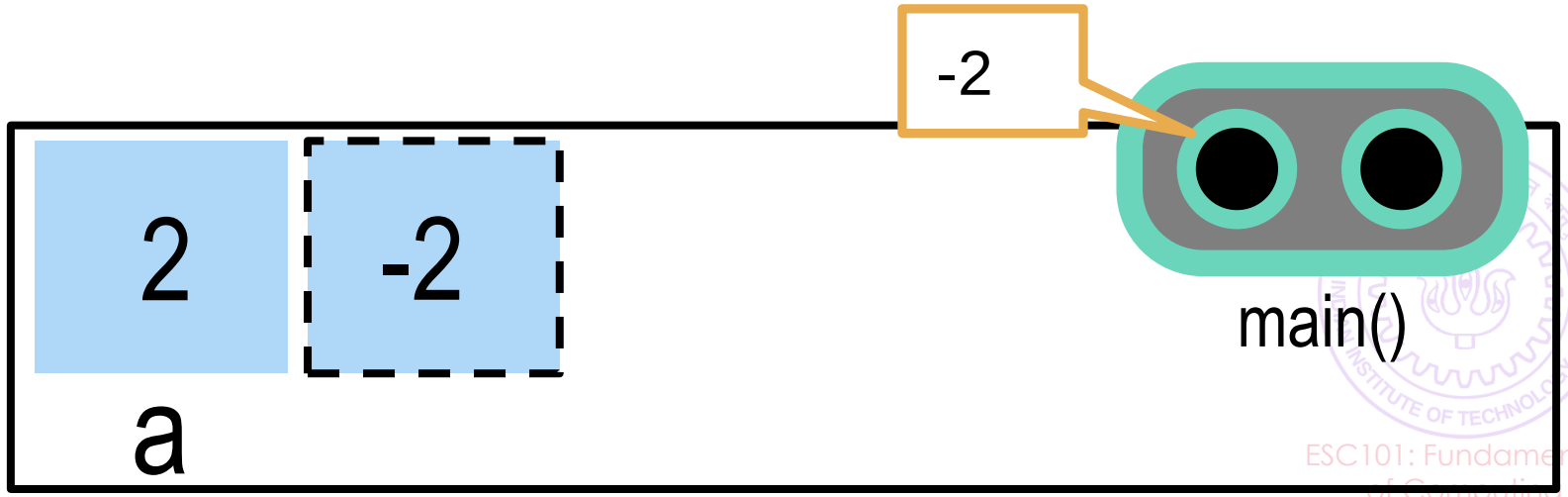


Passing simple variables/expressions

This is the case when the input to the function is either a variable (Rule 1) or an expression (Rule 2)

Rule 4 will always apply no matter what is passed as input

```
int inc(int a){
    return -a;
}
int main(void){
    int a = 2;
    printf("%d", neg(a));
    printf("%d", neg(4*2);
    return 0;
}
```

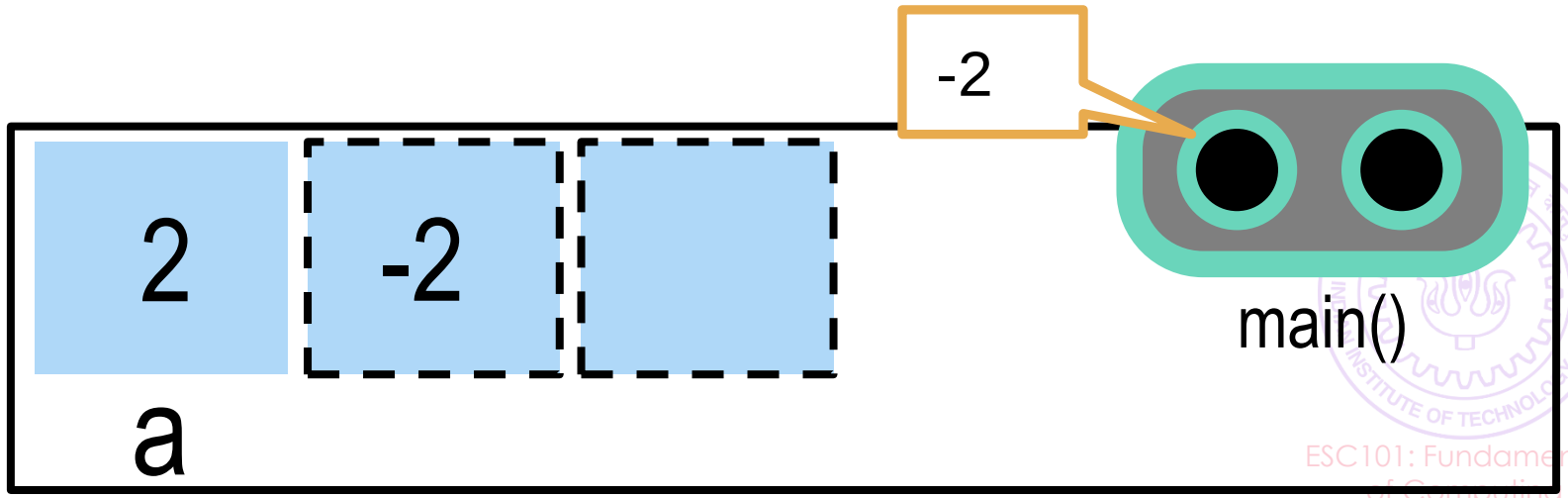


Passing simple variables/expressions

This is the case when the input to the function is either a variable (Rule 1) or an expression (Rule 2)

Rule 4 will always apply no matter what is passed as input

```
int inc(int a){
    return -a;
}
int main(void){
    int a = 2;
    printf("%d", neg(a));
    printf("%d", neg(4*2);
    return 0;
}
```

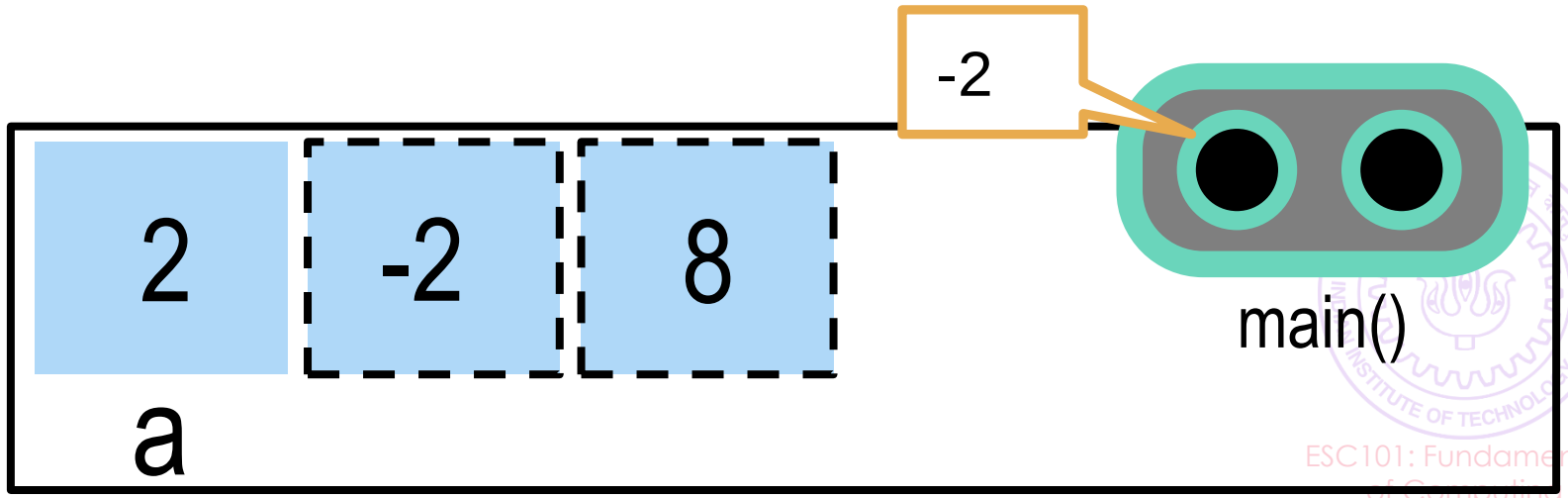


Passing simple variables/expressions

This is the case when the input to the function is either a variable (Rule 1) or an expression (Rule 2)

Rule 4 will always apply no matter what is passed as input

```
int inc(int a){
    return -a;
}
int main(void){
    int a = 2;
    printf("%d", neg(a));
    printf("%d", neg(4*2);
    return 0;
}
```

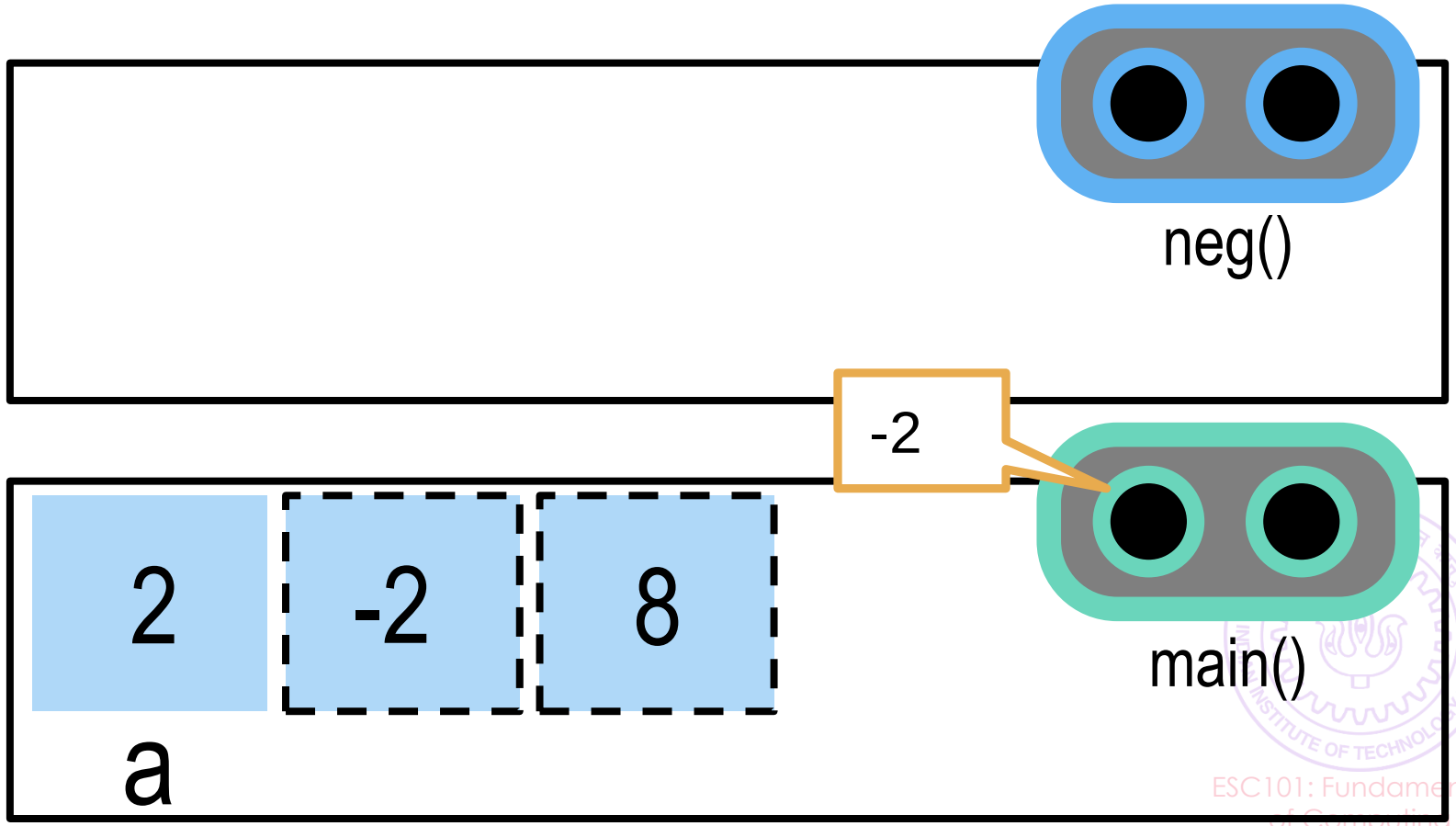


Passing simple variables/expressions

This is the case when the input to the function is either a variable (Rule 1) or an expression (Rule 2)

Rule 4 will always apply no matter what is passed as input

```
int inc(int a){
    return -a;
}
int main(void){
    int a = 2;
    printf("%d", neg(a));
    printf("%d", neg(4*2);
    return 0;
}
```

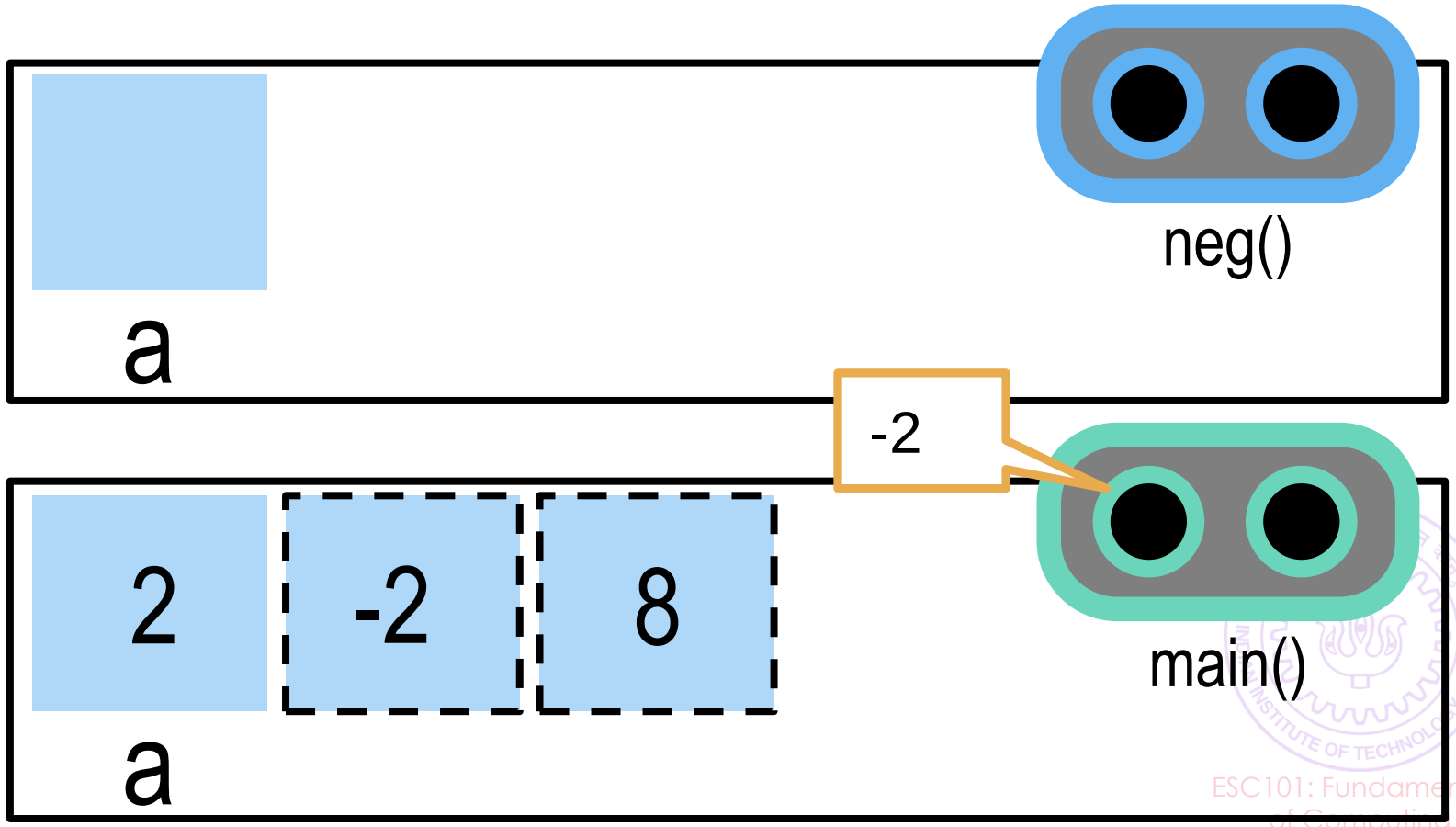


Passing simple variables/expressions

This is the case when the input to the function is either a variable (Rule 1) or an expression (Rule 2)

Rule 4 will always apply no matter what is passed as input

```
int inc(int a){
    return -a;
}
int main(void){
    int a = 2;
    printf("%d", neg(a));
    printf("%d", neg(4*2);
    return 0;
}
```

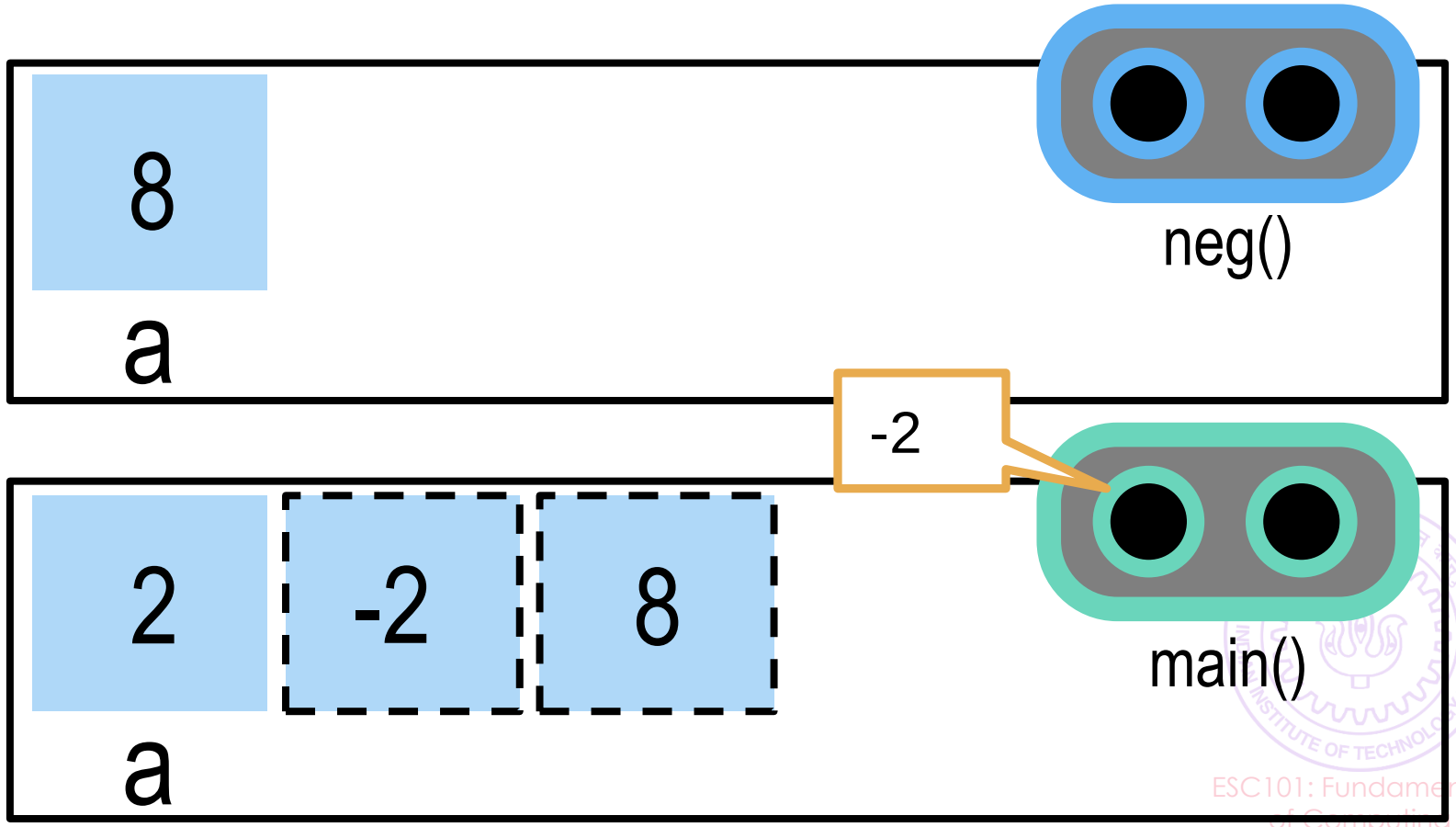


Passing simple variables/expressions

This is the case when the input to the function is either a variable (Rule 1) or an expression (Rule 2)

Rule 4 will always apply no matter what is passed as input

```
int inc(int a){
    return -a;
}
int main(void){
    int a = 2;
    printf("%d", neg(a));
    printf("%d", neg(4*2);
    return 0;
}
```

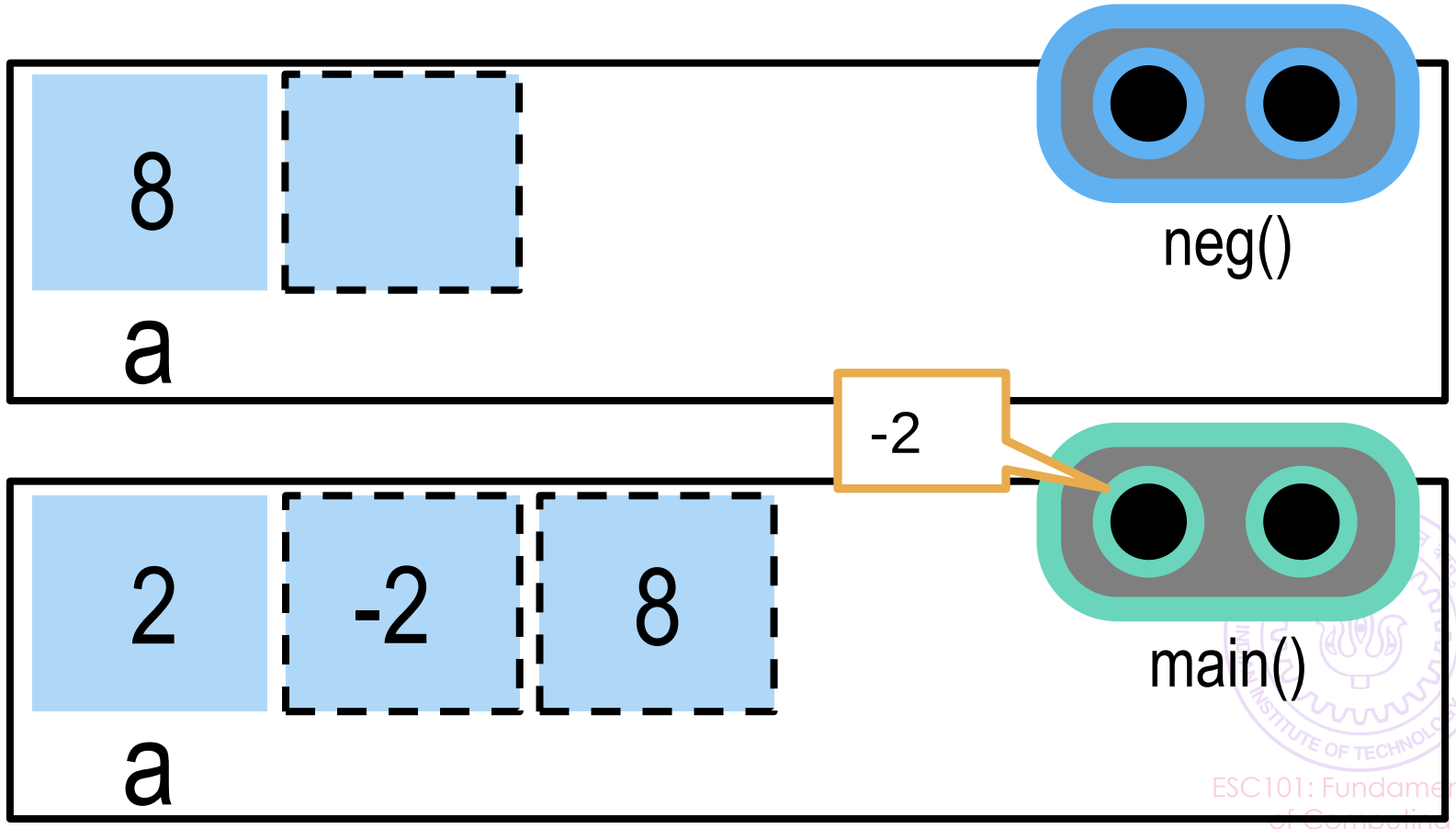


Passing simple variables/expressions

This is the case when the input to the function is either a variable (Rule 1) or an expression (Rule 2)

Rule 4 will always apply no matter what is passed as input

```
int inc(int a){
    return -a;
}
int main(void){
    int a = 2;
    printf("%d", neg(a));
    printf("%d", neg(4*2);
    return 0;
}
```

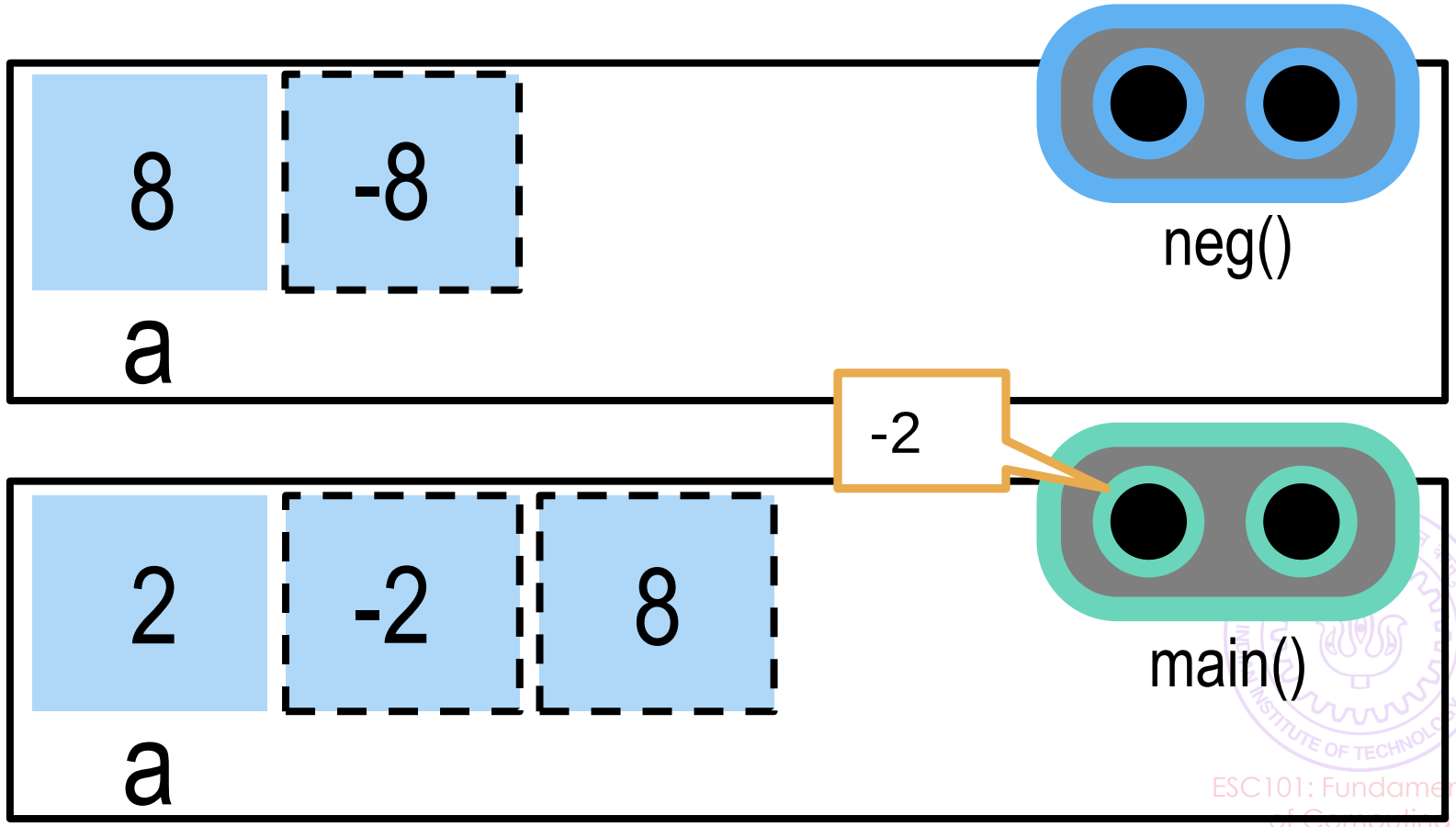


Passing simple variables/expressions

This is the case when the input to the function is either a variable (Rule 1) or an expression (Rule 2)

Rule 4 will always apply no matter what is passed as input

```
int inc(int a){
    return -a;
}
int main(void){
    int a = 2;
    printf("%d", neg(a));
    printf("%d", neg(4*2);
    return 0;
}
```

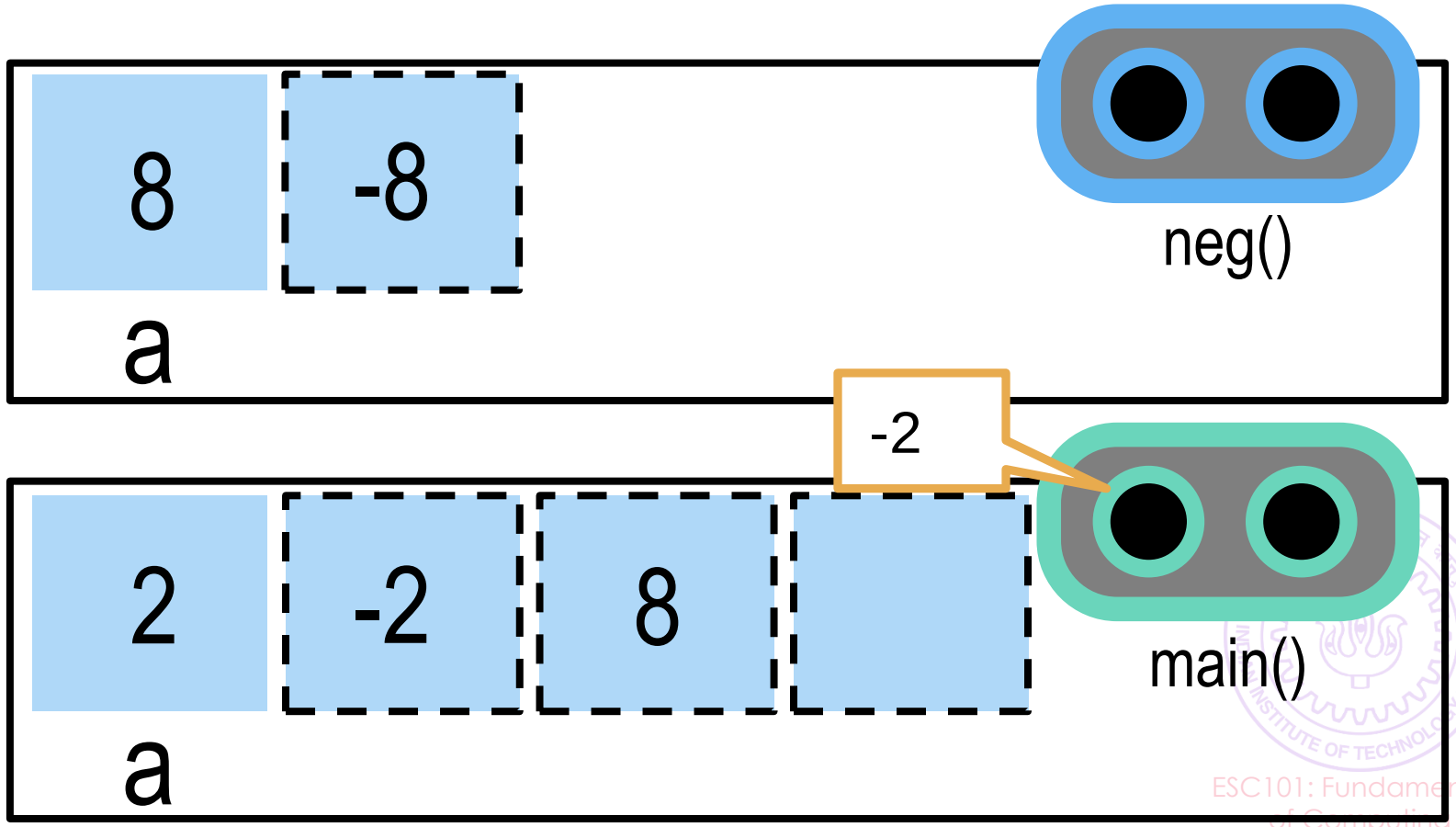


Passing simple variables/expressions

This is the case when the input to the function is either a variable (Rule 1) or an expression (Rule 2)

Rule 4 will always apply no matter what is passed as input

```
int inc(int a){
    return -a;
}
int main(void){
    int a = 2;
    printf("%d", neg(a));
    printf("%d", neg(4*2);
    return 0;
}
```

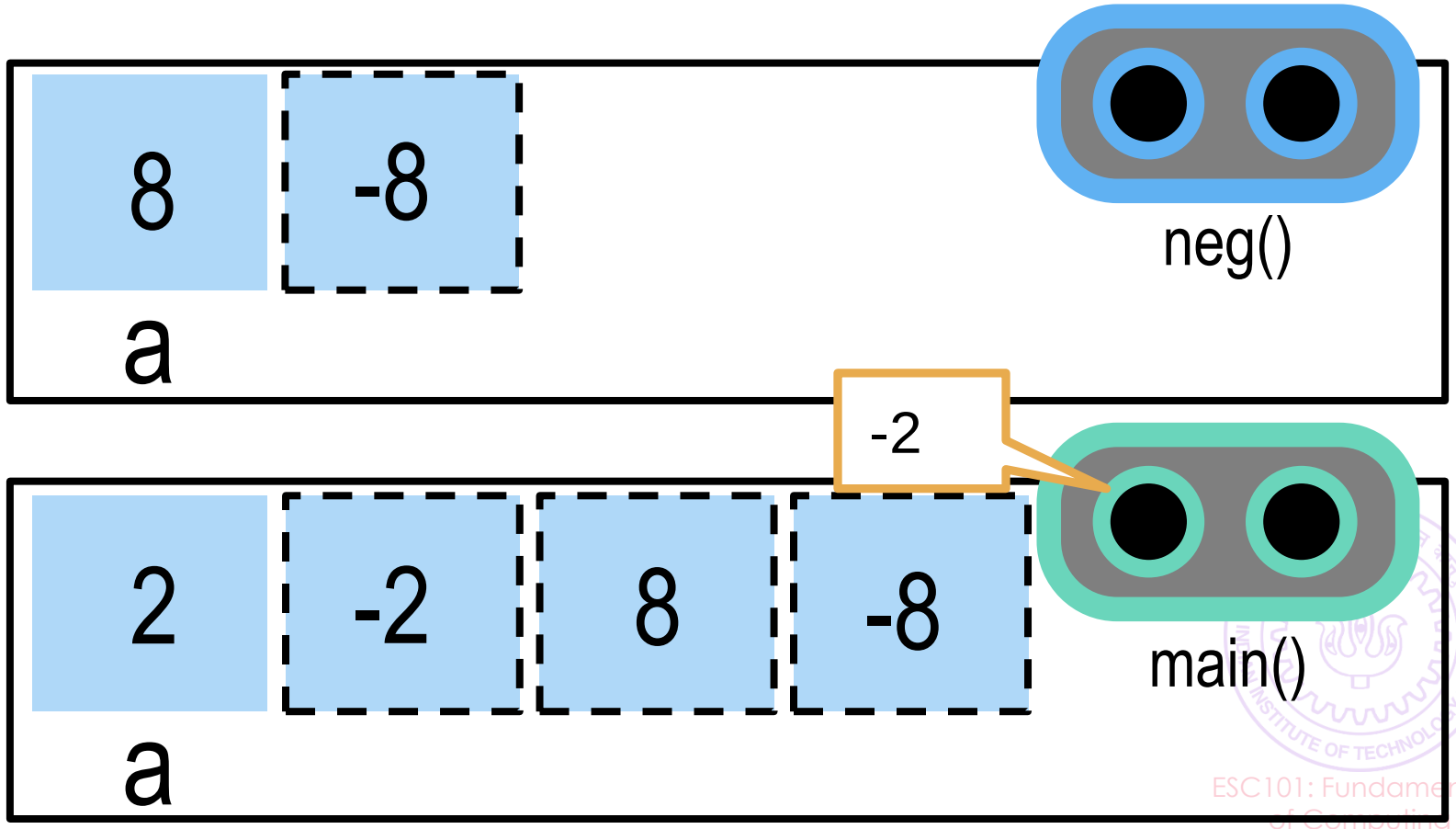


Passing simple variables/expressions

This is the case when the input to the function is either a variable (Rule 1) or an expression (Rule 2)

Rule 4 will always apply no matter what is passed as input

```
int inc(int a){
    return -a;
}
int main(void){
    int a = 2;
    printf("%d", neg(a));
    printf("%d", neg(4*2);
    return 0;
}
```

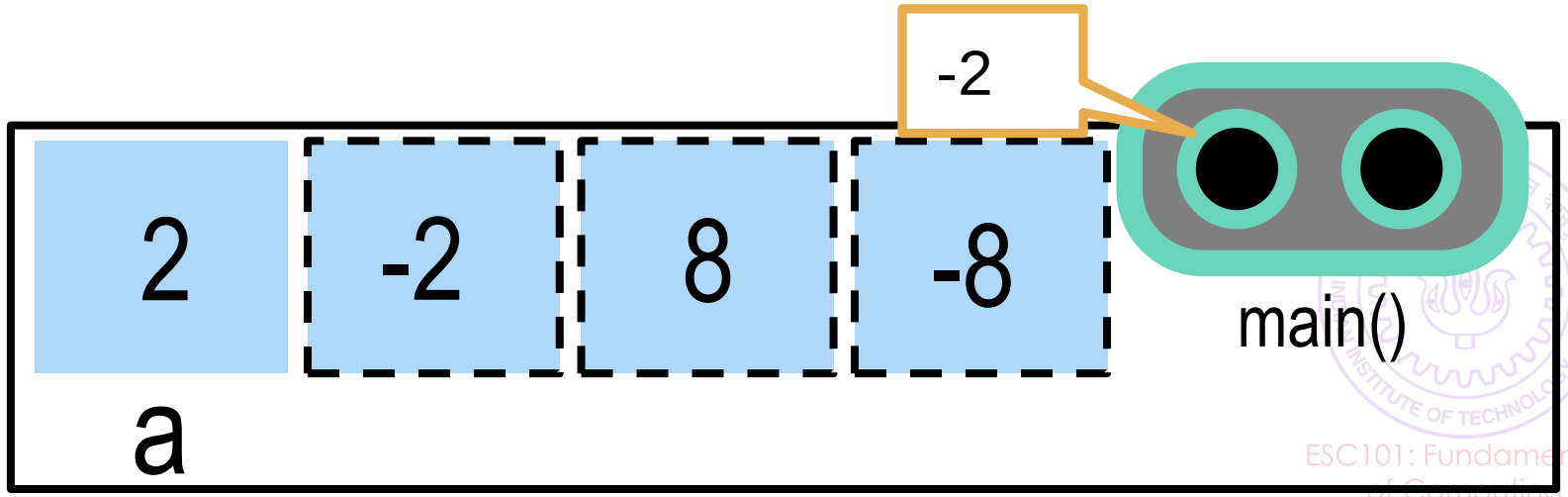


Passing simple variables/expressions

This is the case when the input to the function is either a variable (Rule 1) or an expression (Rule 2)

Rule 4 will always apply no matter what is passed as input

```
int inc(int a){
    return -a;
}
int main(void){
    int a = 2;
    printf("%d", neg(a));
    printf("%d", neg(4*2);
    return 0;
}
```

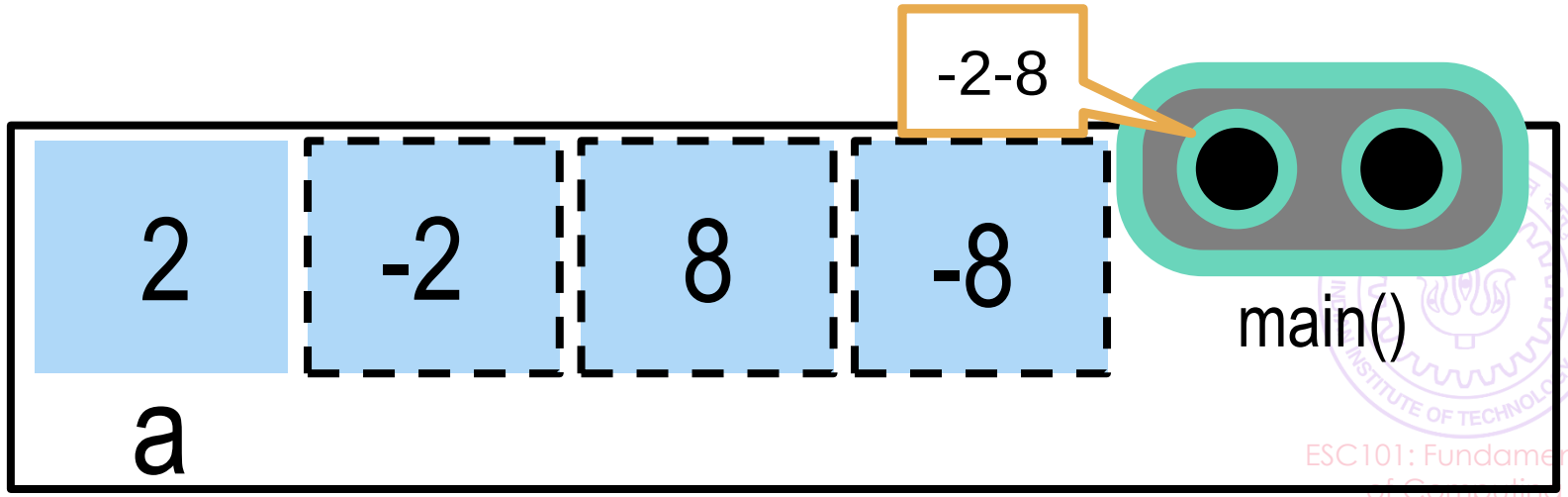


Passing simple variables/expressions

This is the case when the input to the function is either a variable (Rule 1) or an expression (Rule 2)

Rule 4 will always apply no matter what is passed as input

```
int inc(int a){
    return -a;
}
int main(void){
    int a = 2;
    printf("%d", neg(a));
    printf("%d", neg(4*2);
    return 0;
}
```

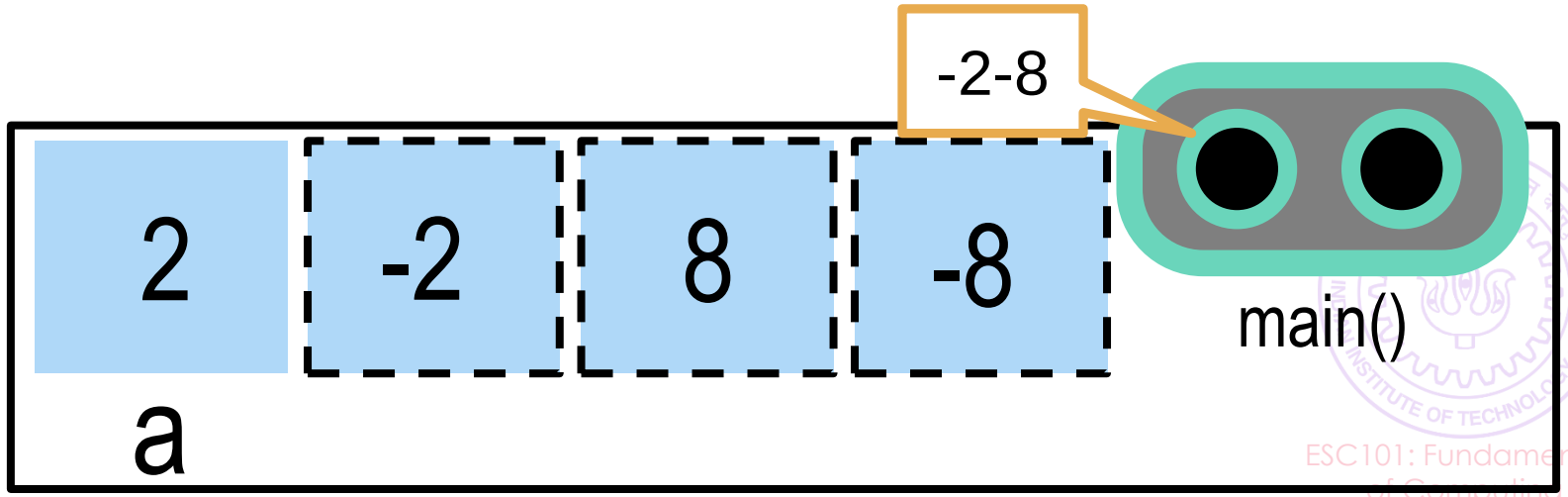


Passing simple variables/expressions

This is the case when the input to the function is either a variable (Rule 1) or an expression (Rule 2)

- Rule 4 will always apply no matter what is passed as input
- Books, websites often call this technique *pass-by-value*

```
int inc(int a){
    return -a;
}
int main(void){
    int a = 2;
    printf("%d", neg(a));
    printf("%d", neg(4*2);
    return 0;
}
```



Passing pointers/addresses

Passing pointers/addresses

Case 1: the input to the function is a pointer variable

Passing pointers/addresses

Case 1: the input to the function is a pointer variable

Rules 1, 4 and 6 apply here
(remember – rule 4 always applies)

Passing pointers/addresses

Case 1: the input to the function is a pointer variable

- Rules 1, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-pointer*

Passing pointers/addresses

Case 1: the input to the function is a pointer variable

- Rules 1, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-pointer*

Case 2: the input to the function is an expression that generates an address value

Passing pointers/addresses

Case 1: the input to the function is a pointer variable

- Rules 1, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-pointer*

Case 2: the input to the function is an expression that generates an address value

- Rules 2, 4 and 6 apply here (remember – rule 4 always applies)

Passing pointers/addresses

Case 1: the input to the function is a pointer variable

- Rules 1, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-pointer*

Case 2: the input to the function is an expression that generates an address value

- Rules 2, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-reference*

Passing pointers/addresses

Case 1: the input to the function is a pointer variable

- Rules 1, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-pointer*

Case 2: the input to the function is an expression that generates an address value

- Rules 2, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-reference*

```
void prn(int *ptr){
    printf("%d", *ptr);
    *ptr++;
}
int main(void){
    int a = 42;
    int *b= &a;
    prn(b);
    prn(&a);
    return 0;
}
```

Passing pointers/addresses

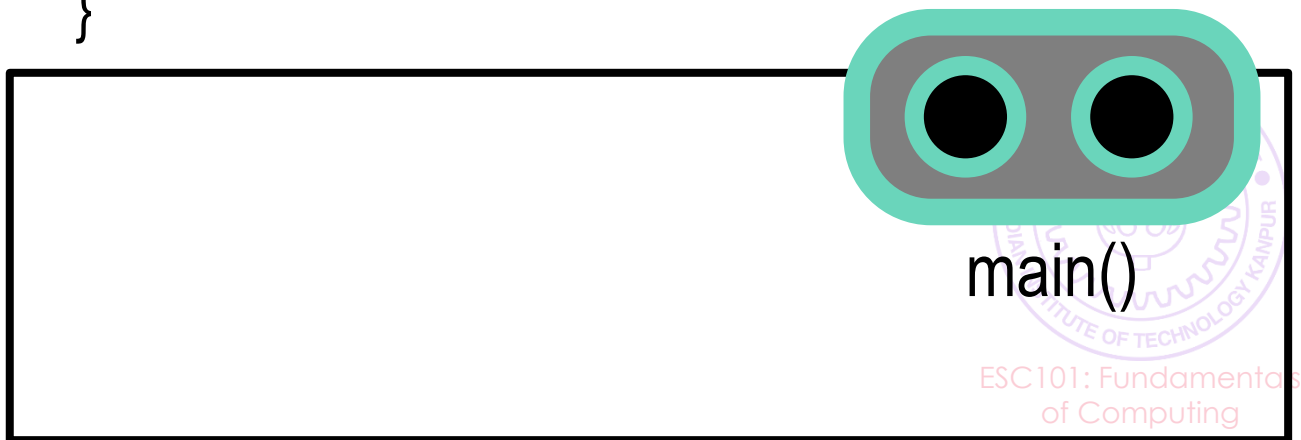
Case 1: the input to the function is a pointer variable

- Rules 1, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-pointer*

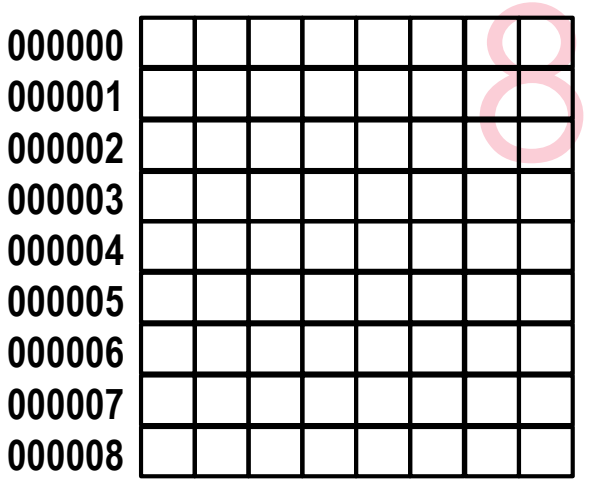
Case 2: the input to the function is an expression that generates an address value

- Rules 2, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-reference*

```
void prn(int *ptr){
    printf("%d", *ptr);
    *ptr++;
}
int main(void){
    int a = 42;
    int *b= &a;
    prn(b);
    prn(&a);
    return 0;
}
```



Passing pointers/addresses



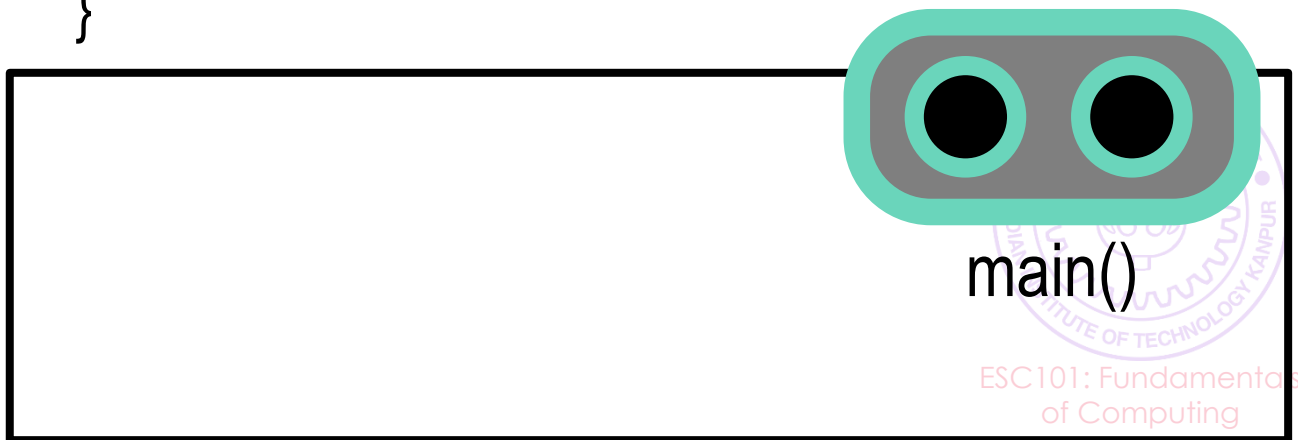
Case 1: the input to the function is a pointer variable

- Rules 1, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-pointer*

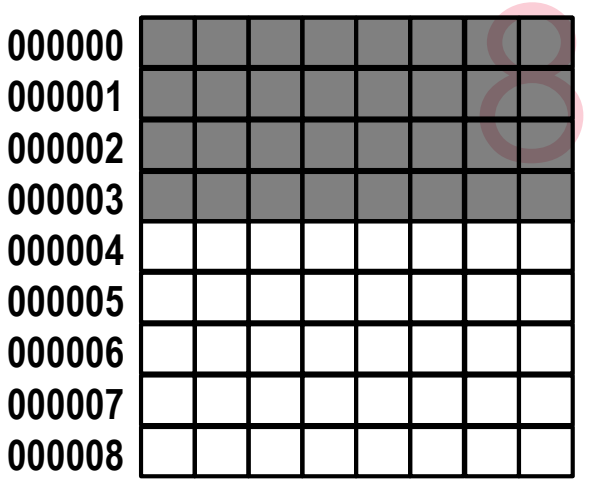
```
void prn(int *ptr){
    printf("%d", *ptr);
    *ptr++;
}
int main(void){
    int a = 42;
    int *b= &a;
    prn(b);
    prn(&a);
    return 0;
}
```

Case 2: the input to the function is an expression that generates an address value

- Rules 2, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-reference*



Passing pointers/addresses



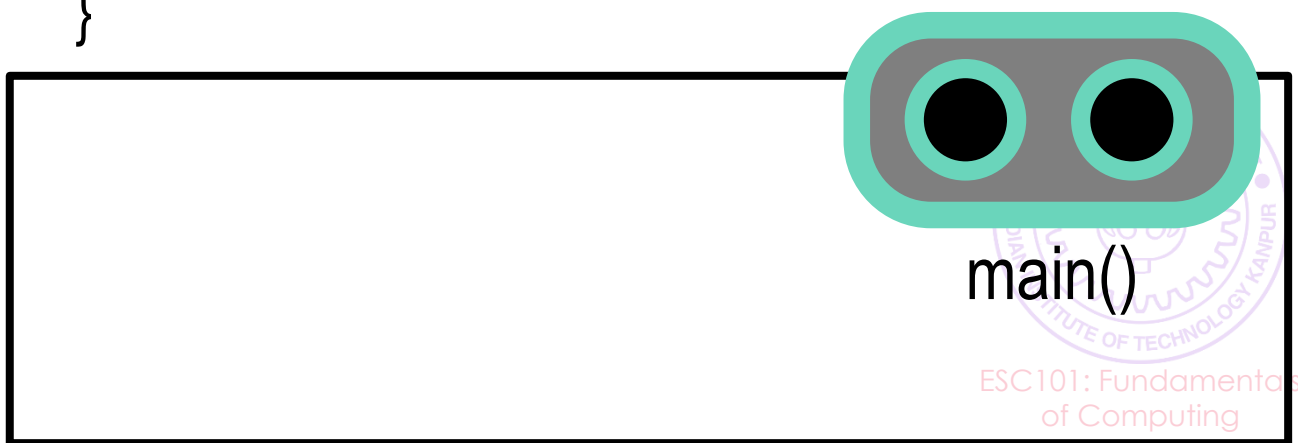
Case 1: the input to the function is a pointer variable

- Rules 1, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-pointer*

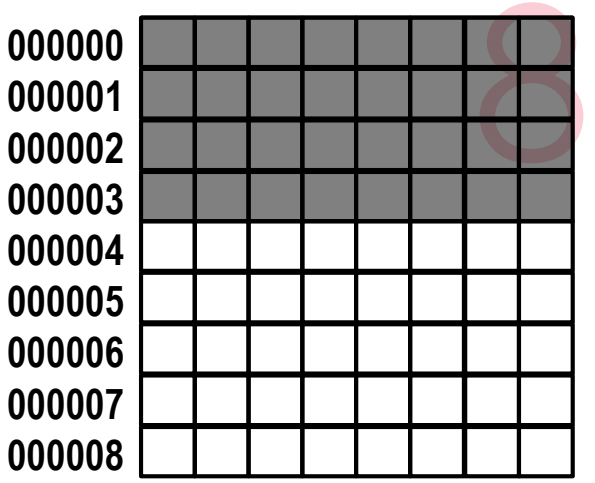
```
void prn(int *ptr){
    printf("%d", *ptr);
    *ptr++;
}
int main(void){
    int a = 42;
    int *b= &a;
    prn(b);
    prn(&a);
    return 0;
}
```

Case 2: the input to the function is an expression that generates an address value

- Rules 2, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-reference*



Passing pointers/addresses



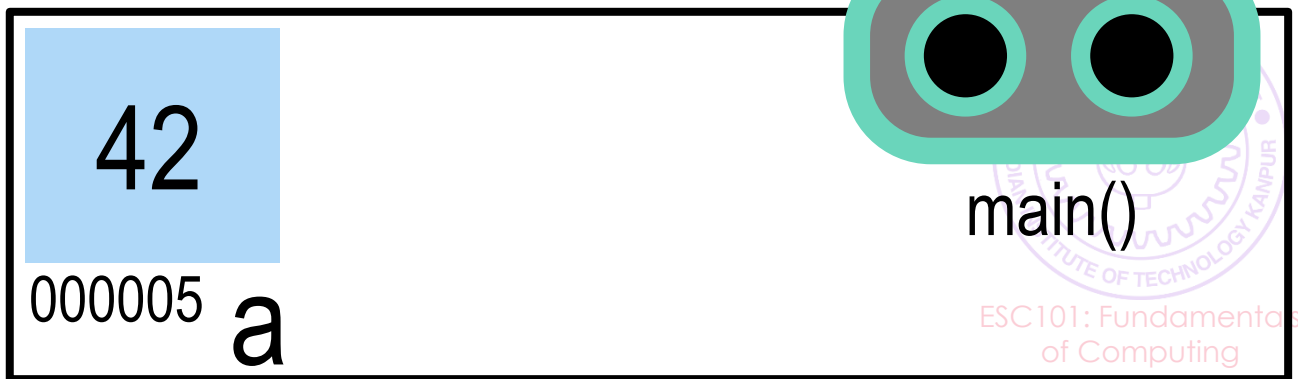
Case 1: the input to the function is a pointer variable

- Rules 1, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-pointer*

Case 2: the input to the function is an expression that generates an address value

- Rules 2, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-reference*

```
void prn(int *ptr){
    printf("%d", *ptr);
    *ptr++;
}
int main(void){
    int a = 42;
    int *b= &a;
    prn(b);
    prn(&a);
    return 0;
}
```



Passing pointers/addresses

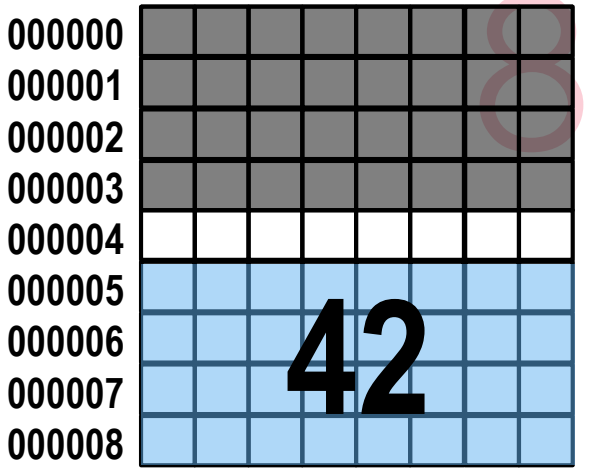
Case 1: the input to the function is a pointer variable

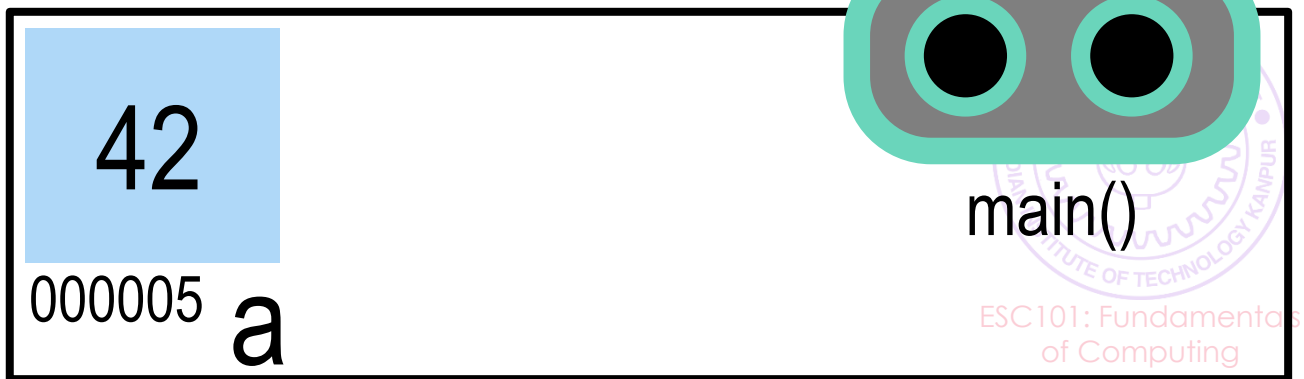
- Rules 1, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-pointer*

Case 2: the input to the function is an expression that generates an address value

- Rules 2, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-reference*

```
void prn(int *ptr){
    printf("%d", *ptr);
    *ptr++;
}
int main(void){
    int a = 42;
    int *b= &a;
    prn(b);
    prn(&a);
    return 0;
}
```





Passing pointers/addresses

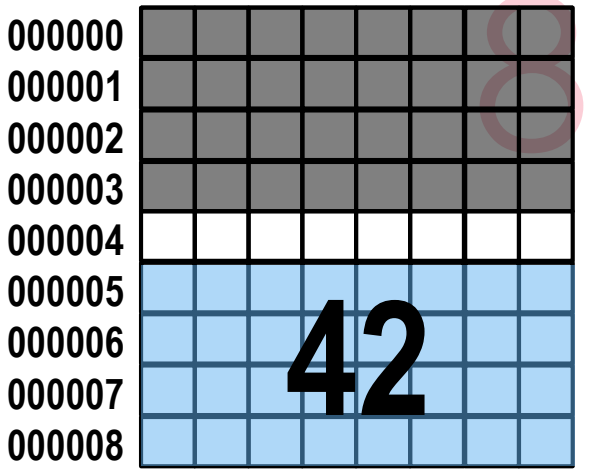
Case 1: the input to the function is a pointer variable

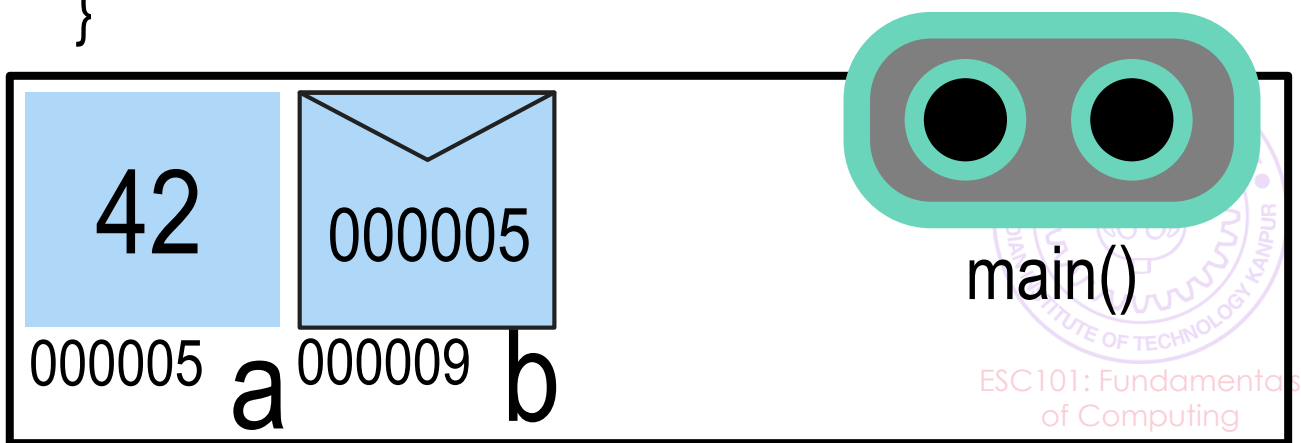
- Rules 1, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-pointer*

Case 2: the input to the function is an expression that generates an address value

- Rules 2, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-reference*

```
void prn(int *ptr){
    printf("%d", *ptr);
    *ptr++;
}
int main(void){
    int a = 42;
    int *b= &a;
    prn(b);
    prn(&a);
    return 0;
}
```





Passing pointers/addresses

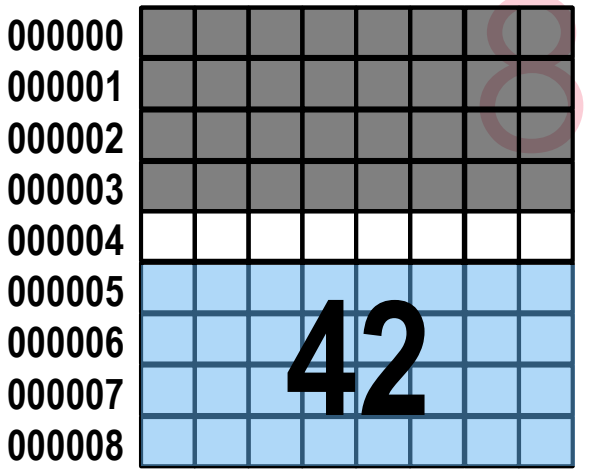
Case 1: the input to the function is a pointer variable

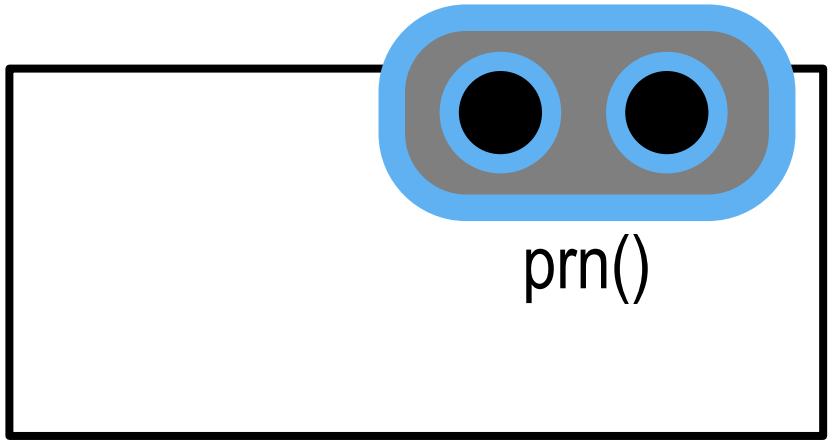
- Rules 1, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-pointer*

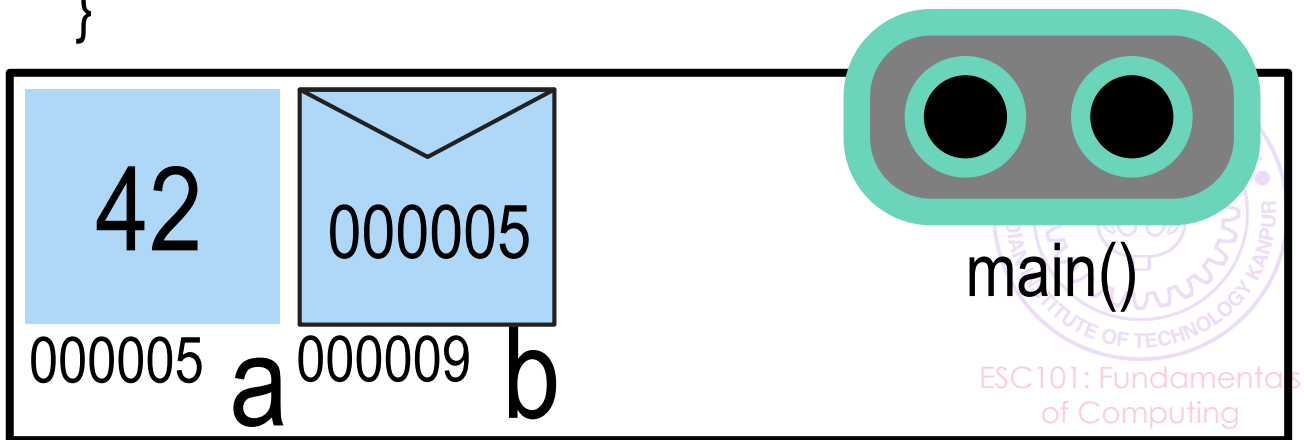
Case 2: the input to the function is an expression that generates an address value

- Rules 2, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-reference*

```
void prn(int *ptr){
    printf("%d", *ptr);
    *ptr++;
}
int main(void){
    int a = 42;
    int *b= &a;
    prn(b);
    prn(&a);
    return 0;
}
```







Passing pointers/addresses

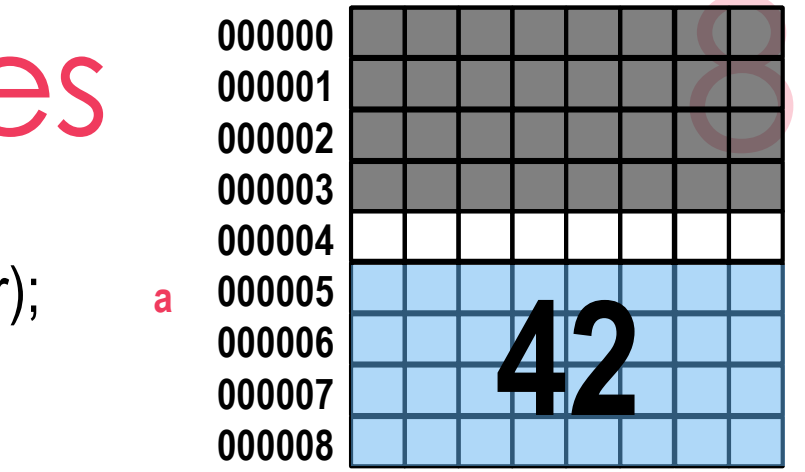
Case 1: the input to the function is a pointer variable

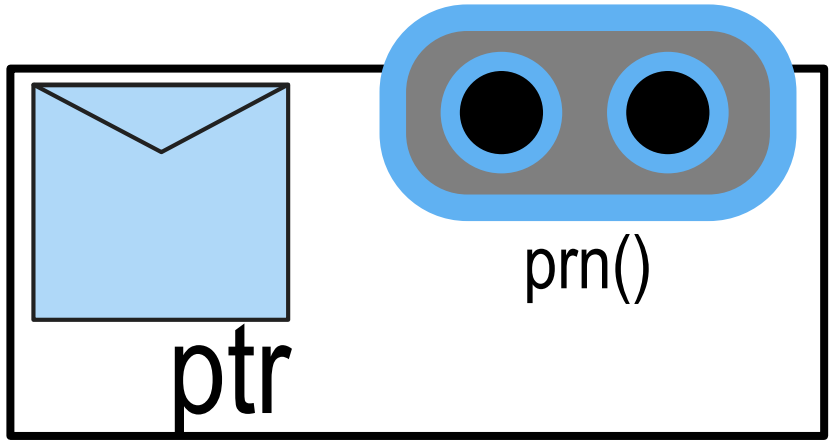
- Rules 1, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-pointer*

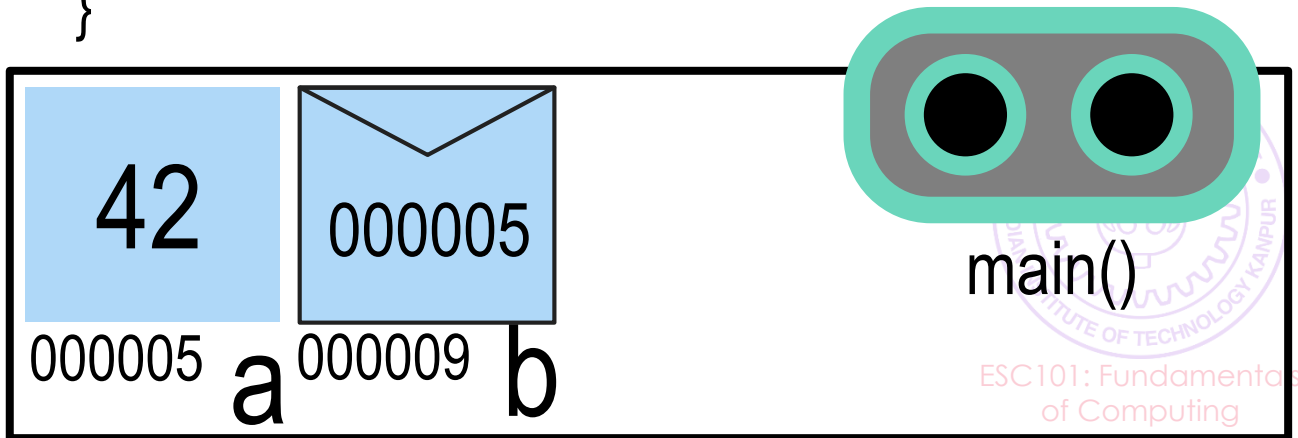
Case 2: the input to the function is an expression that generates an address value

- Rules 2, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-reference*

```
void prn(int *ptr){
    printf("%d", *ptr);
    *ptr++;
}
int main(void){
    int a = 42;
    int *b= &a;
    prn(b);
    prn(&a);
    return 0;
}
```







Passing pointers/addresses

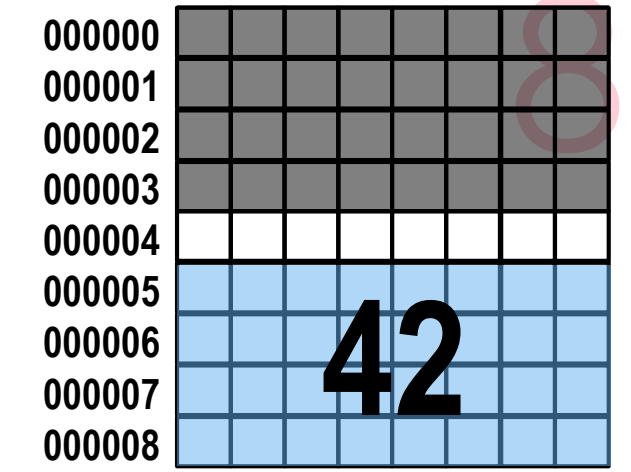
Case 1: the input to the function is a pointer variable

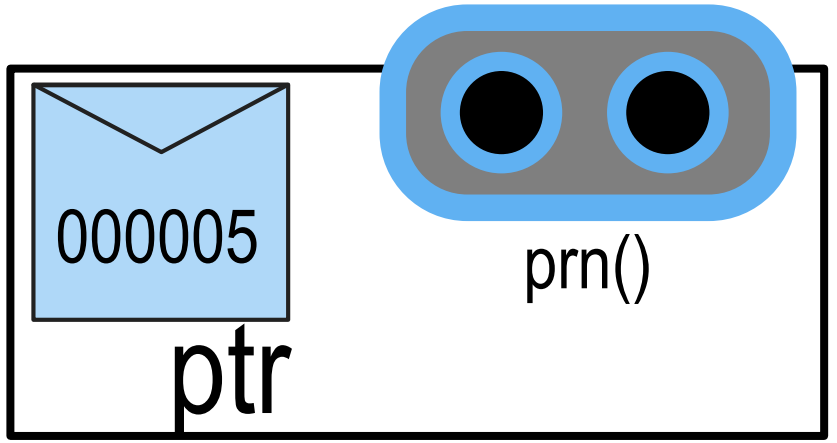
- Rules 1, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-pointer*

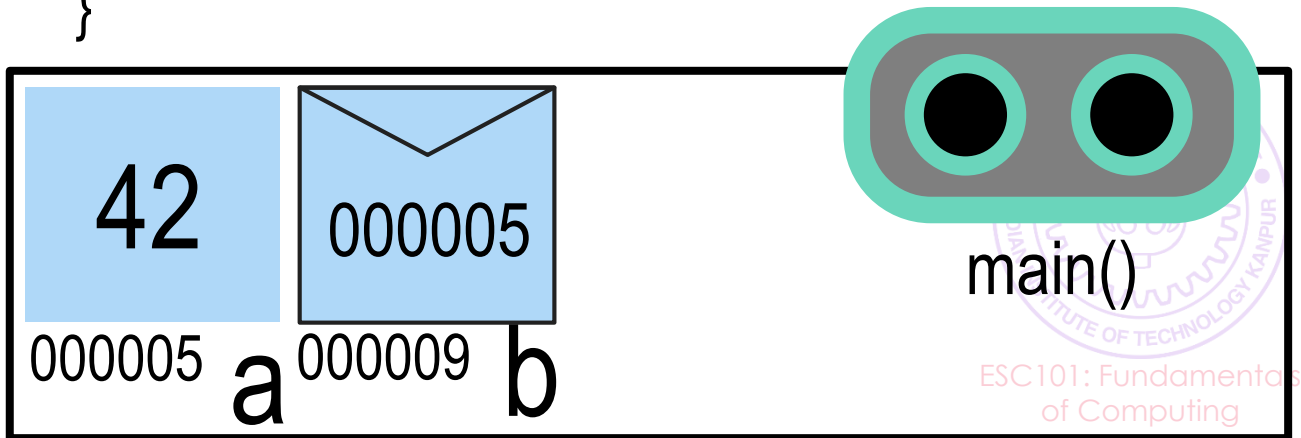
Case 2: the input to the function is an expression that generates an address value

- Rules 2, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-reference*

```
void prn(int *ptr){
    printf("%d", *ptr);
    *ptr++;
}
int main(void){
    int a = 42;
    int *b= &a;
    prn(b);
    prn(&a);
    return 0;
}
```





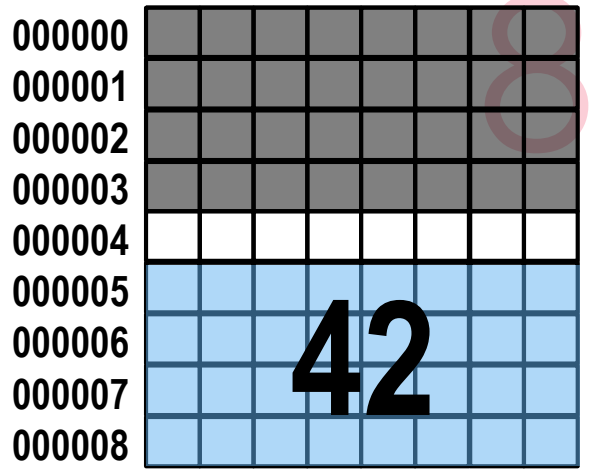


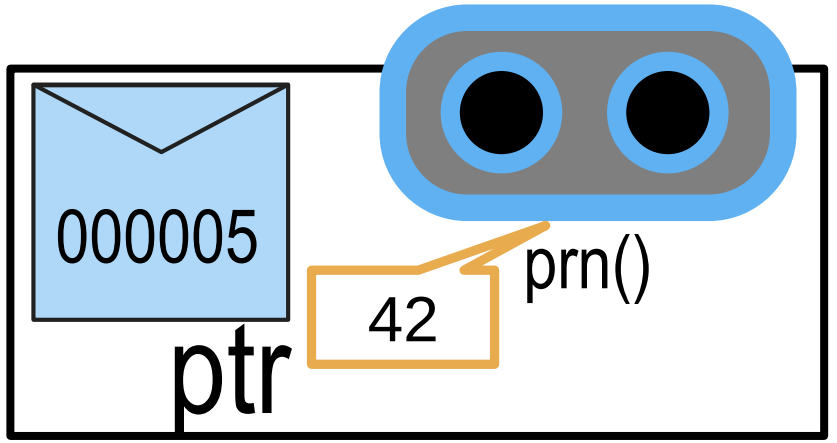
Passing pointers/addresses

Case 1: the input to the function is a pointer variable

- Rules 1, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-pointer*

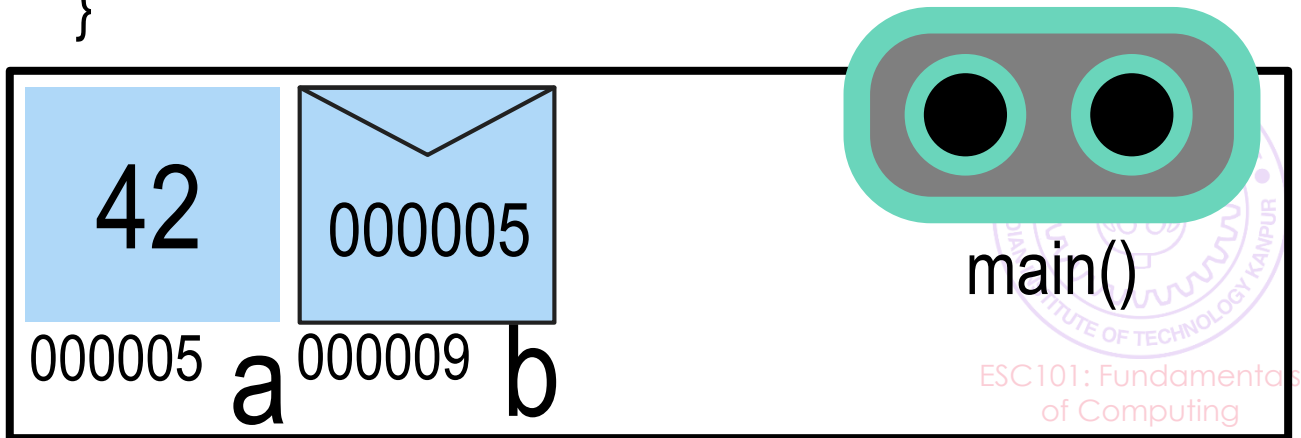
```
void prn(int *ptr){
    printf("%d", *ptr);
    *ptr++;
}
int main(void){
    int a = 42;
    int *b= &a;
    prn(b);
    prn(&a);
    return 0;
}
```





Case 2: the input to the function is an expression that generates an address value

- Rules 2, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-reference*

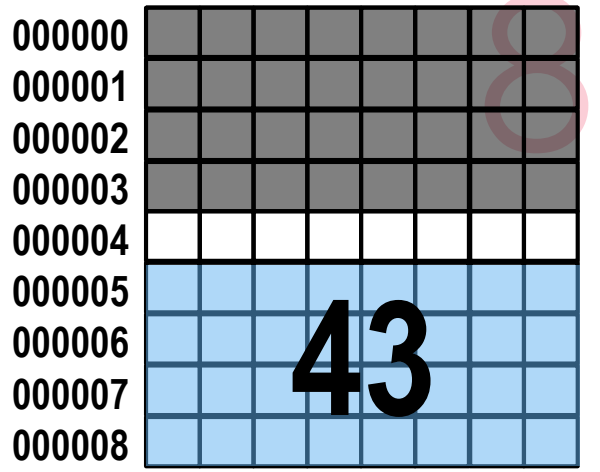


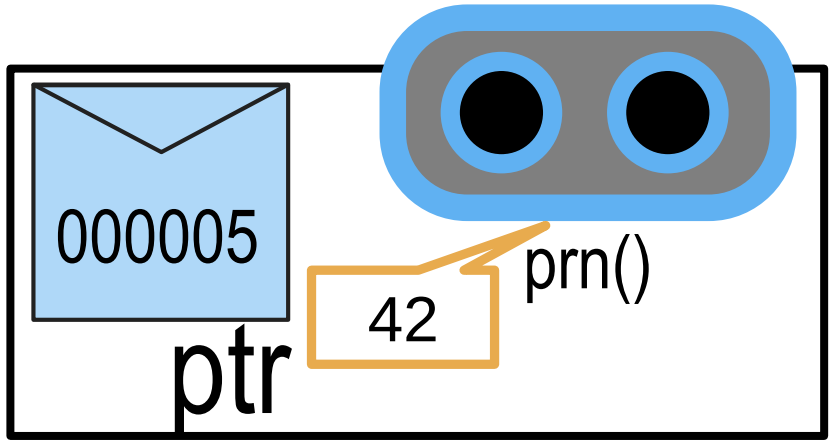
Passing pointers/addresses

Case 1: the input to the function is a pointer variable

- Rules 1, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-pointer*

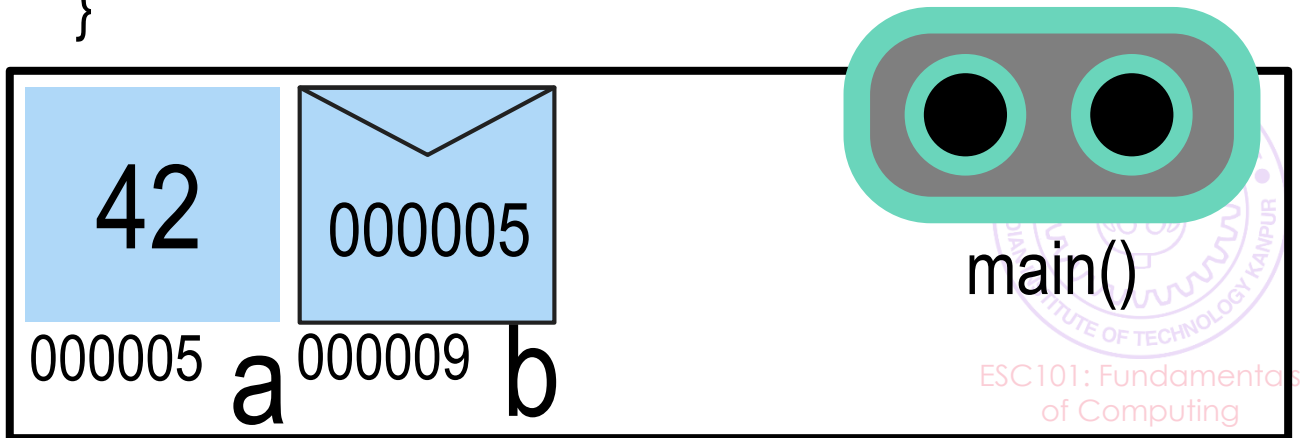
```
void prn(int *ptr){
    printf("%d", *ptr);
    *ptr++;
}
int main(void){
    int a = 42;
    int *b= &a;
    prn(b);
    prn(&a);
    return 0;
}
```





Case 2: the input to the function is an expression that generates an address value

- Rules 2, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-reference*



Passing pointers/addresses

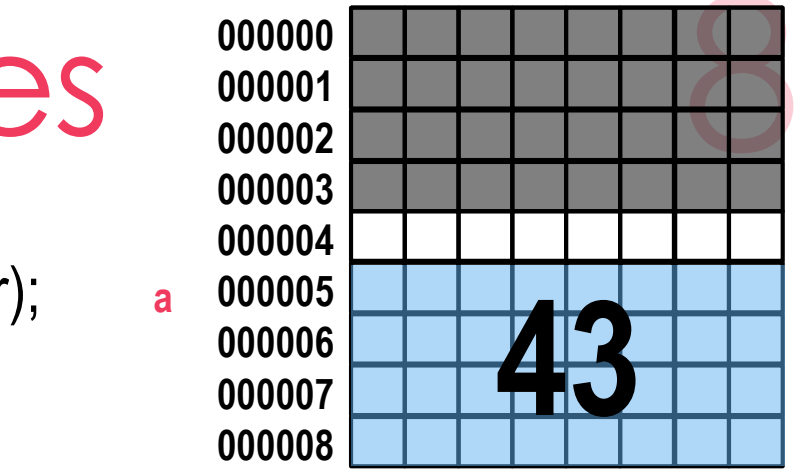
Case 1: the input to the function is a pointer variable

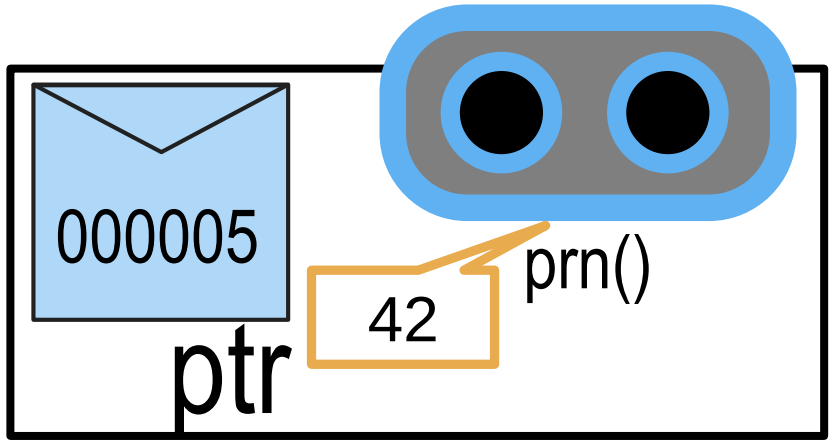
- Rules 1, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-pointer*

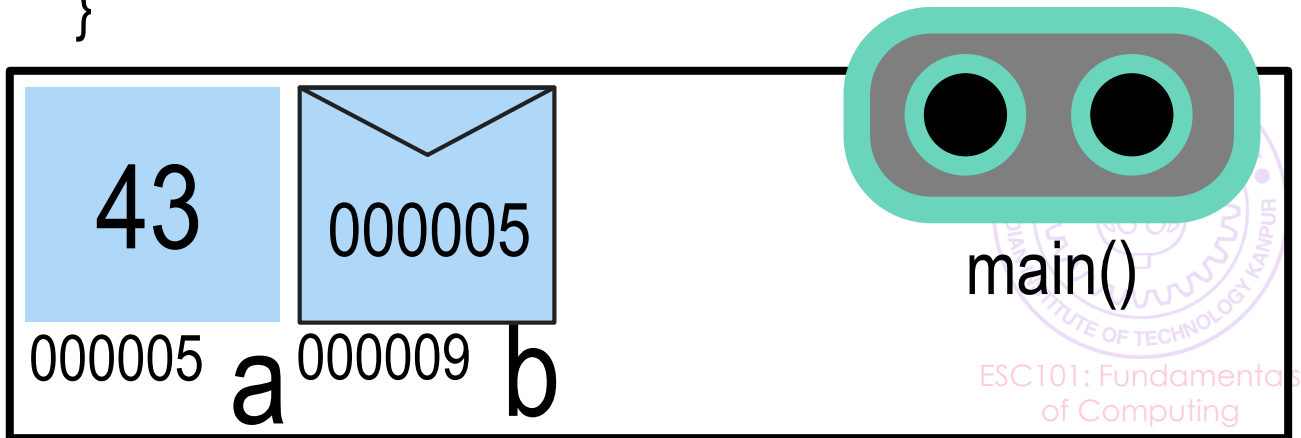
Case 2: the input to the function is an expression that generates an address value

- Rules 2, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-reference*

```
void prn(int *ptr){
    printf("%d", *ptr);
    *ptr++;
}
int main(void){
    int a = 42;
    int *b= &a;
    prn(b);
    prn(&a);
    return 0;
}
```







Passing pointers/addresses

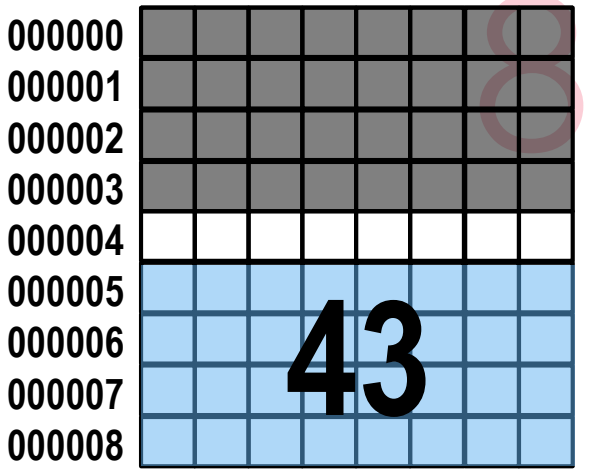
Case 1: the input to the function is a pointer variable

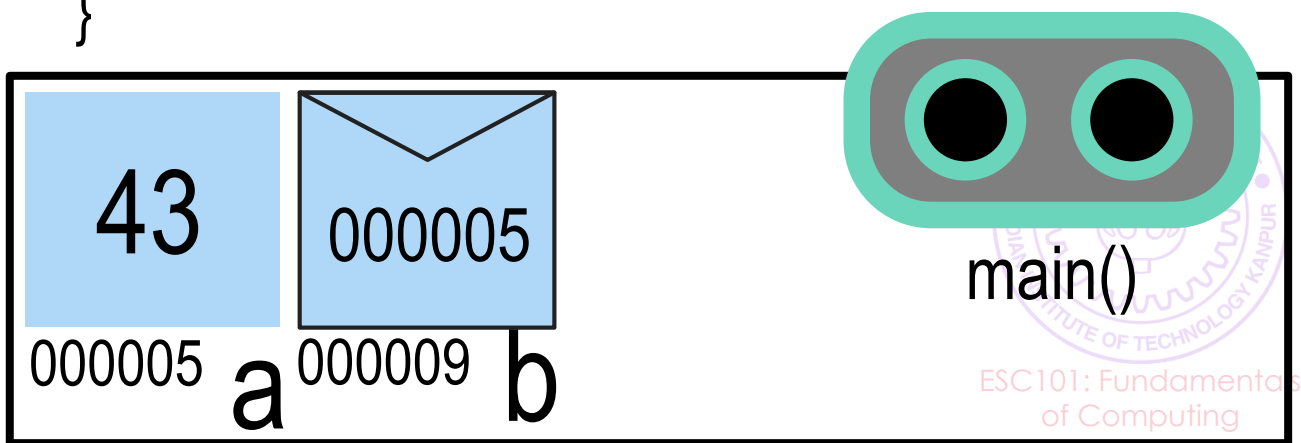
- Rules 1, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-pointer*

Case 2: the input to the function is an expression that generates an address value

- Rules 2, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-reference*

```
void prn(int *ptr){
    printf("%d", *ptr);
    *ptr++;
}
int main(void){
    int a = 42;
    int *b= &a;
    prn(b);
    prn(&a);
    return 0;
}
```





Passing pointers/addresses

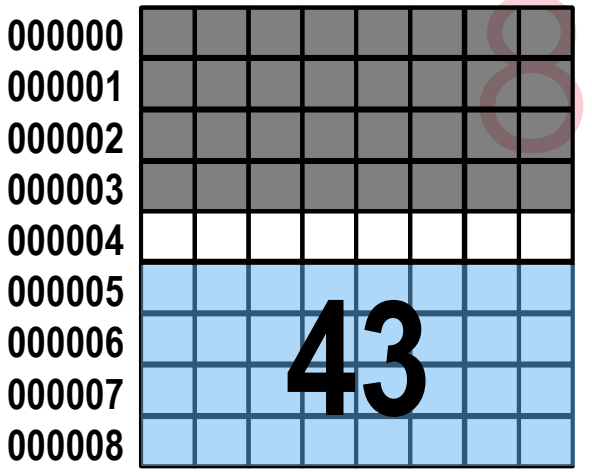
Case 1: the input to the function is a pointer variable

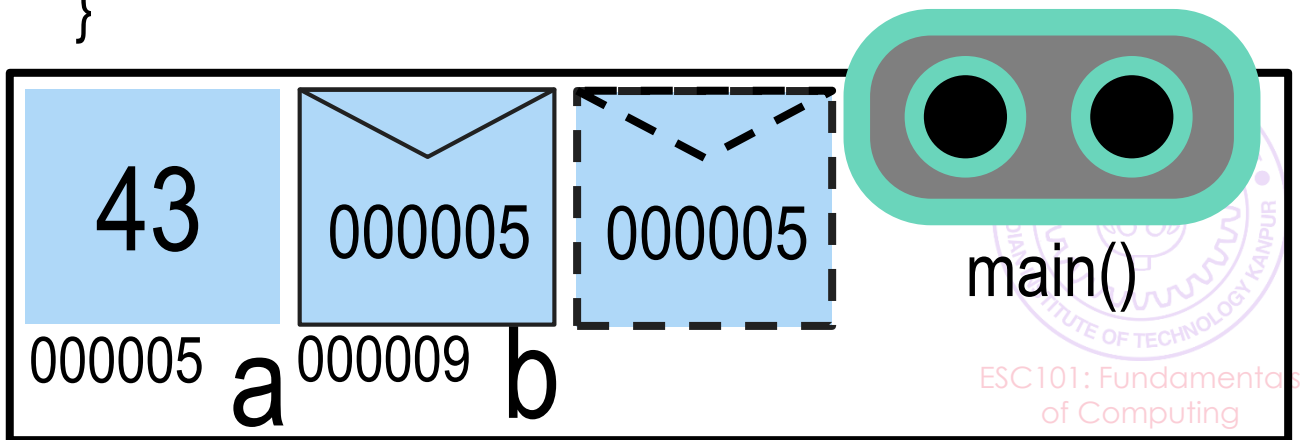
- Rules 1, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-pointer*

Case 2: the input to the function is an expression that generates an address value

- Rules 2, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-reference*

```
void prn(int *ptr){
    printf("%d", *ptr);
    *ptr++;
}
int main(void){
    int a = 42;
    int *b= &a;
    prn(b);
    prn(&a);
    return 0;
}
```





Passing pointers/addresses

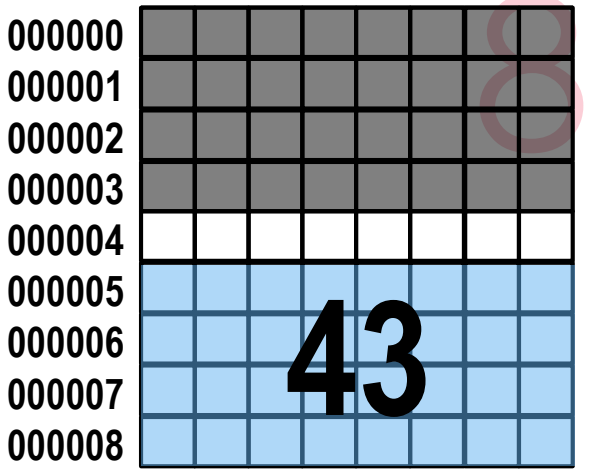
Case 1: the input to the function is a pointer variable

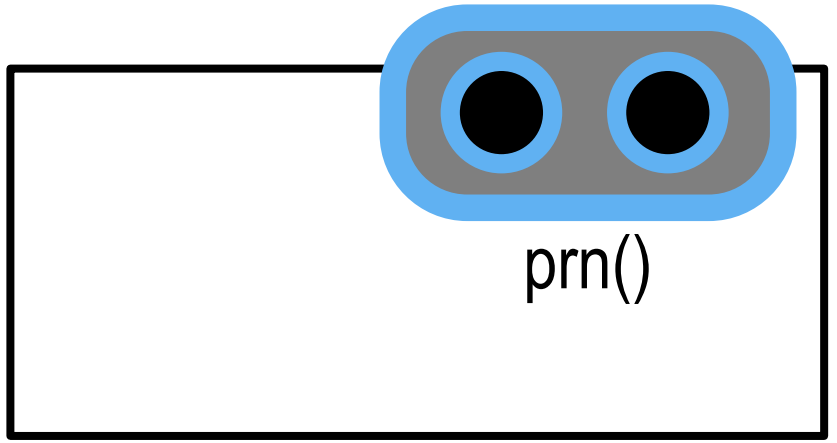
- Rules 1, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-pointer*

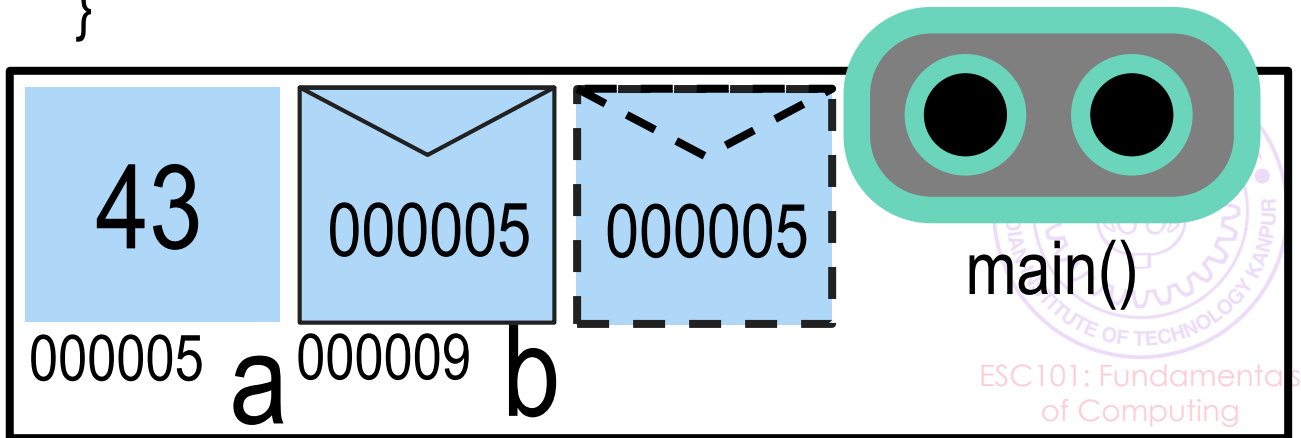
Case 2: the input to the function is an expression that generates an address value

- Rules 2, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-reference*

```
void prn(int *ptr){
    printf("%d", *ptr);
    *ptr++;
}
int main(void){
    int a = 42;
    int *b= &a;
    prn(b);
    prn(&a);
    return 0;
}
```







Passing pointers/addresses

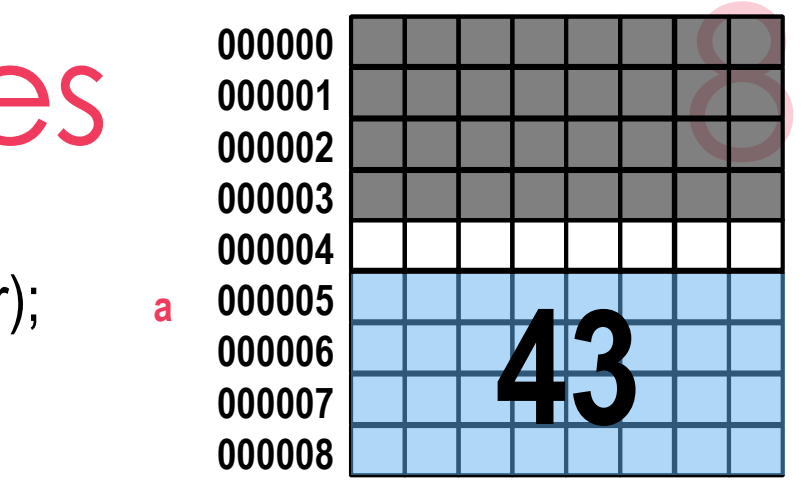
Case 1: the input to the function is a pointer variable

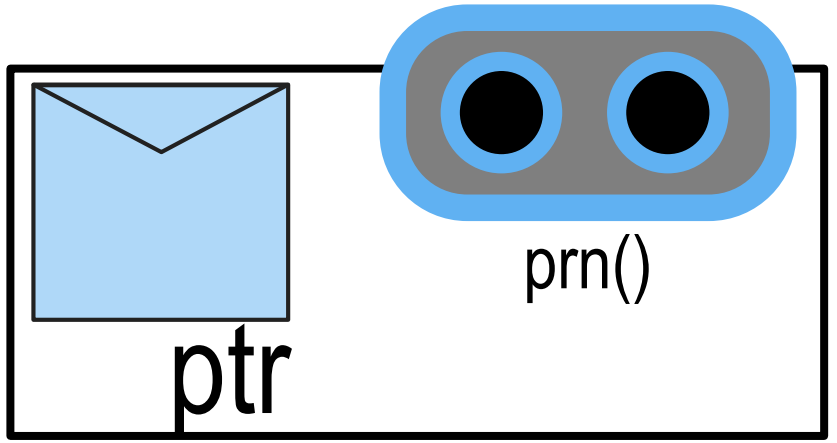
- Rules 1, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-pointer*

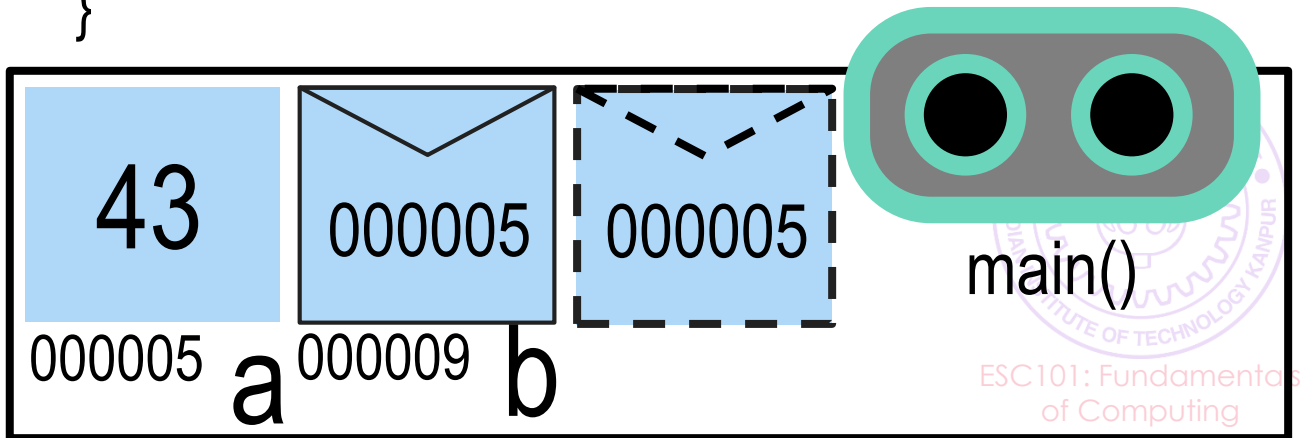
Case 2: the input to the function is an expression that generates an address value

- Rules 2, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-reference*

```
void prn(int *ptr){
    printf("%d", *ptr);
    *ptr++;
}
int main(void){
    int a = 42;
    int *b= &a;
    prn(b);
    prn(&a);
    return 0;
}
```







Passing pointers/addresses

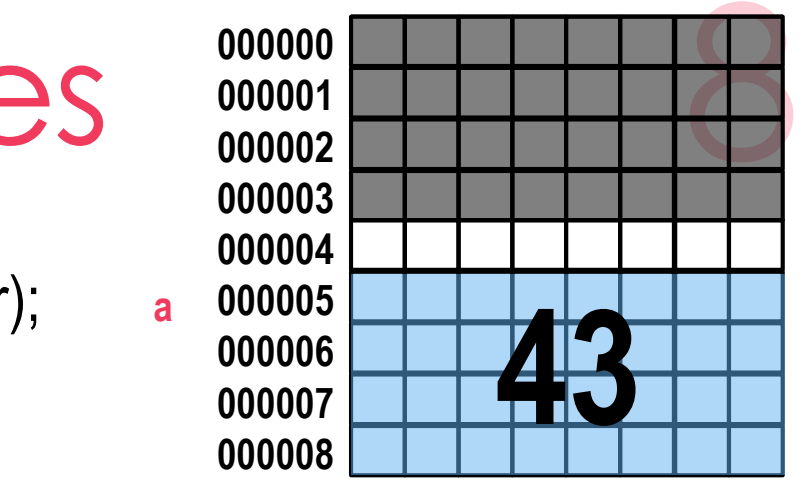
Case 1: the input to the function is a pointer variable

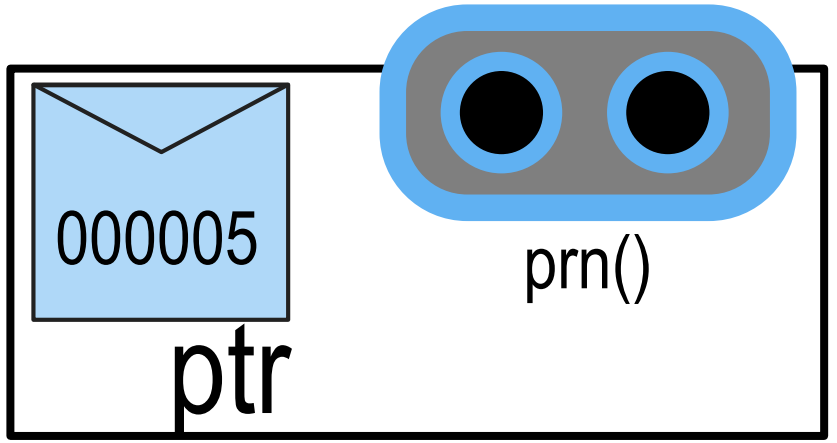
- Rules 1, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-pointer*

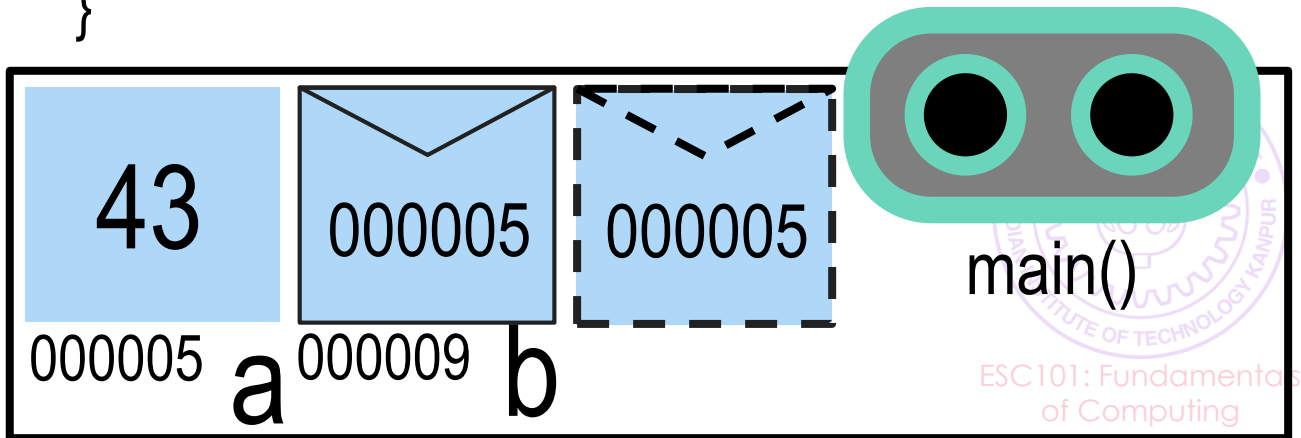
Case 2: the input to the function is an expression that generates an address value

- Rules 2, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-reference*

```
void prn(int *ptr){
    printf("%d", *ptr);
    *ptr++;
}
int main(void){
    int a = 42;
    int *b= &a;
    prn(b);
    prn(&a);
    return 0;
}
```







Passing pointers/addresses

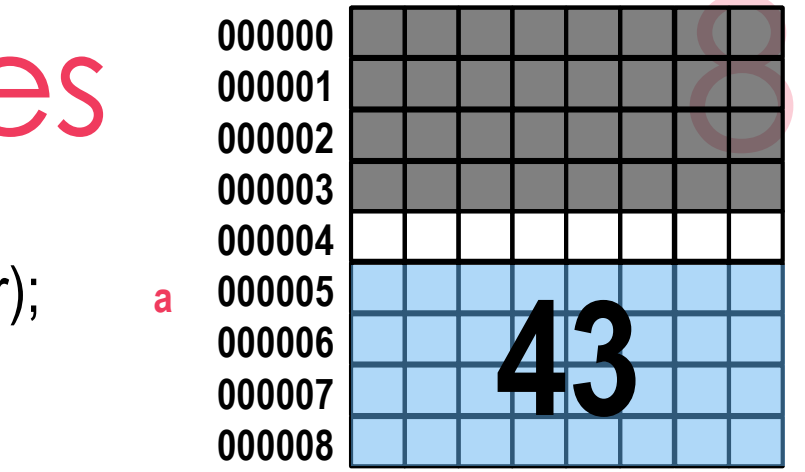
Case 1: the input to the function is a pointer variable

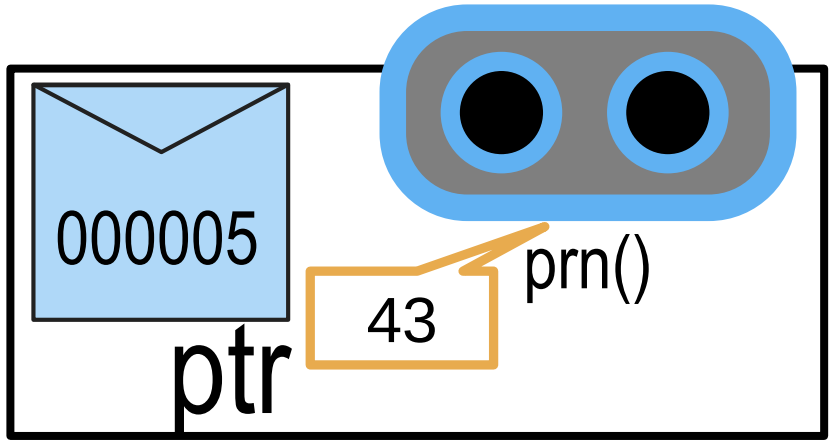
- Rules 1, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-pointer*

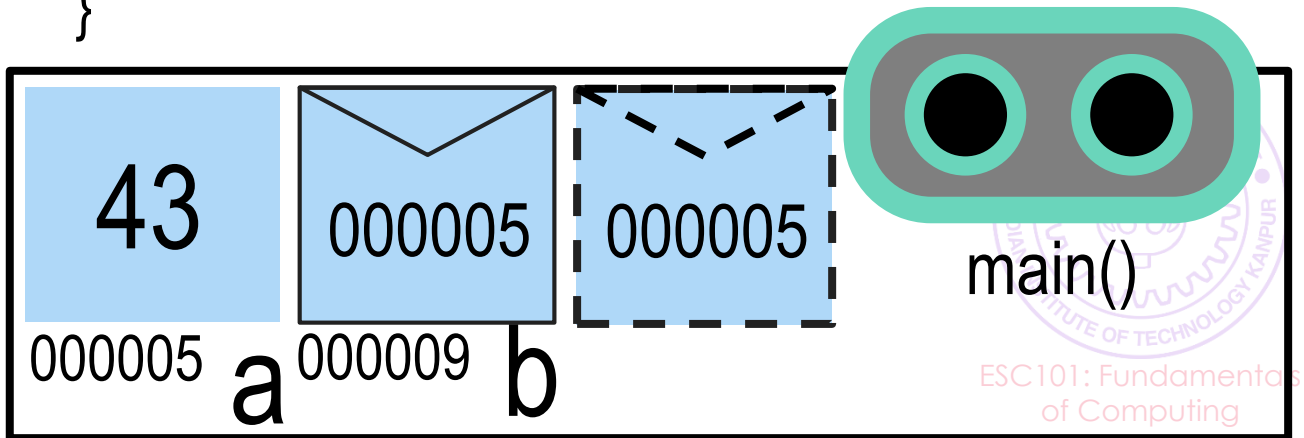
Case 2: the input to the function is an expression that generates an address value

- Rules 2, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-reference*

```
void prn(int *ptr){
    printf("%d", *ptr);
    *ptr++;
}
int main(void){
    int a = 42;
    int *b= &a;
    prn(b);
    prn(&a);
    return 0;
}
```





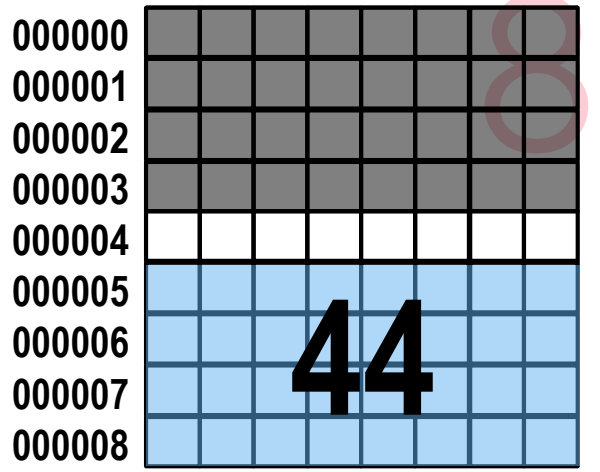


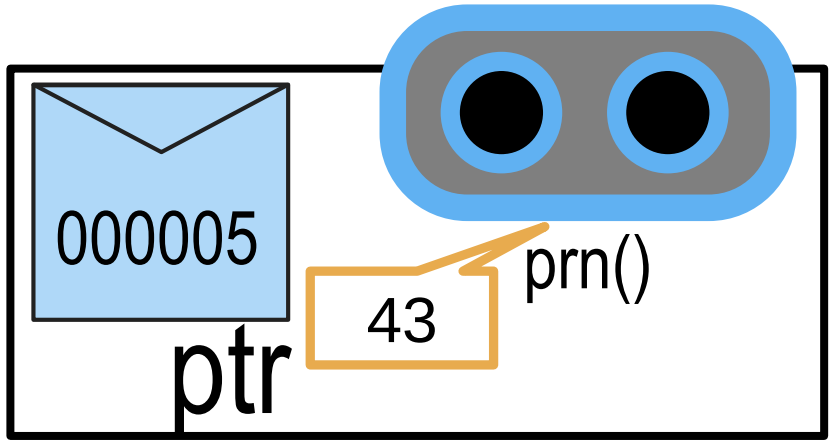
Passing pointers/addresses

Case 1: the input to the function is a pointer variable

- Rules 1, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-pointer*

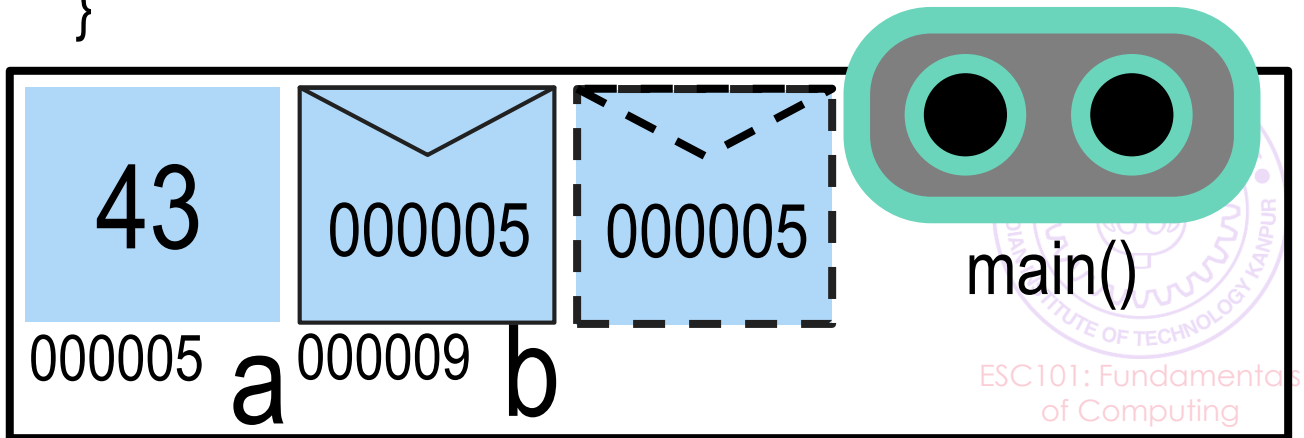
```
void prn(int *ptr){
    printf("%d", *ptr);
    *ptr++;
}
int main(void){
    int a = 42;
    int *b= &a;
    prn(b);
    prn(&a);
    return 0;
}
```





Case 2: the input to the function is an expression that generates an address value

- Rules 2, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-reference*



Passing pointers/addresses

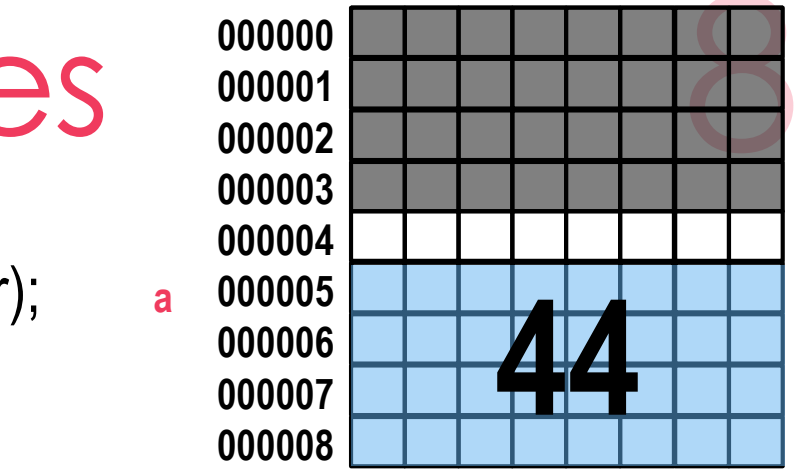
Case 1: the input to the function is a pointer variable

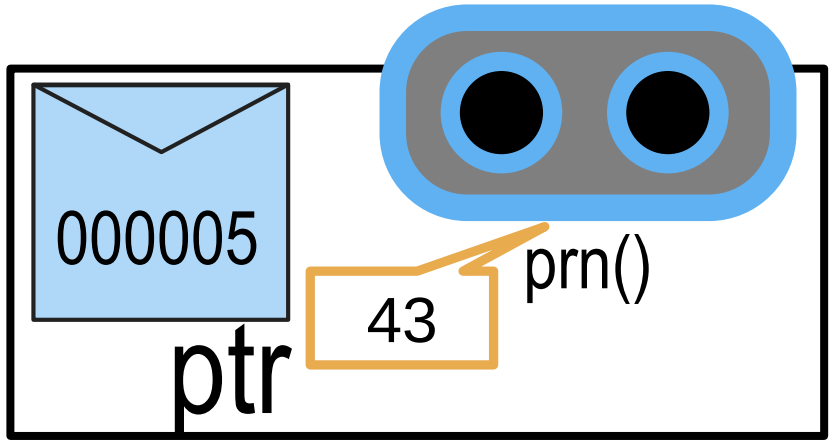
- Rules 1, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-pointer*

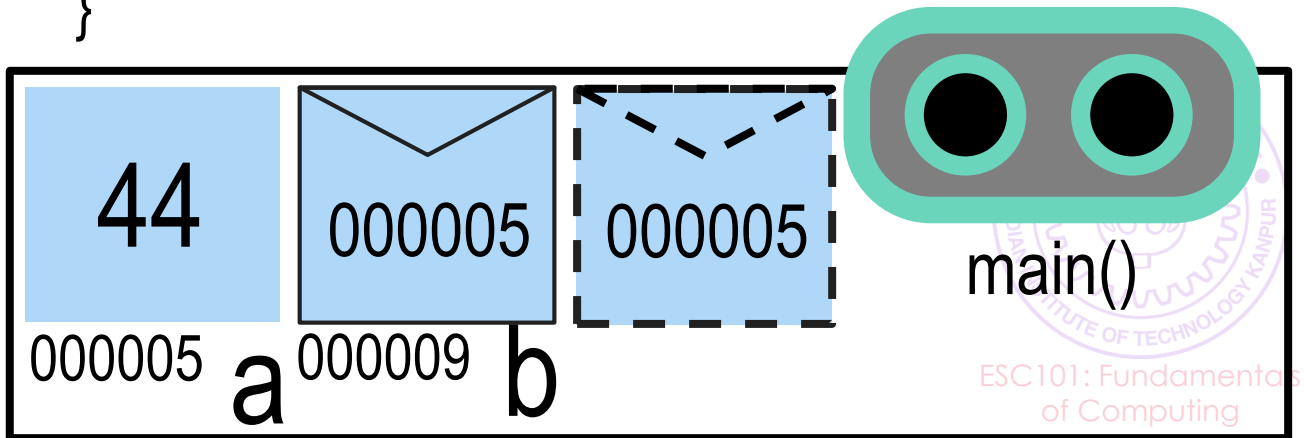
Case 2: the input to the function is an expression that generates an address value

- Rules 2, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-reference*

```
void prn(int *ptr){
    printf("%d", *ptr);
    *ptr++;
}
int main(void){
    int a = 42;
    int *b= &a;
    prn(b);
    prn(&a);
    return 0;
}
```







Passing pointers/addresses

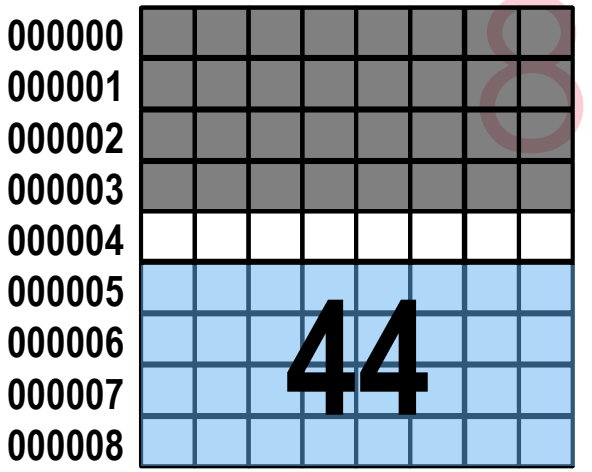
Case 1: the input to the function is a pointer variable

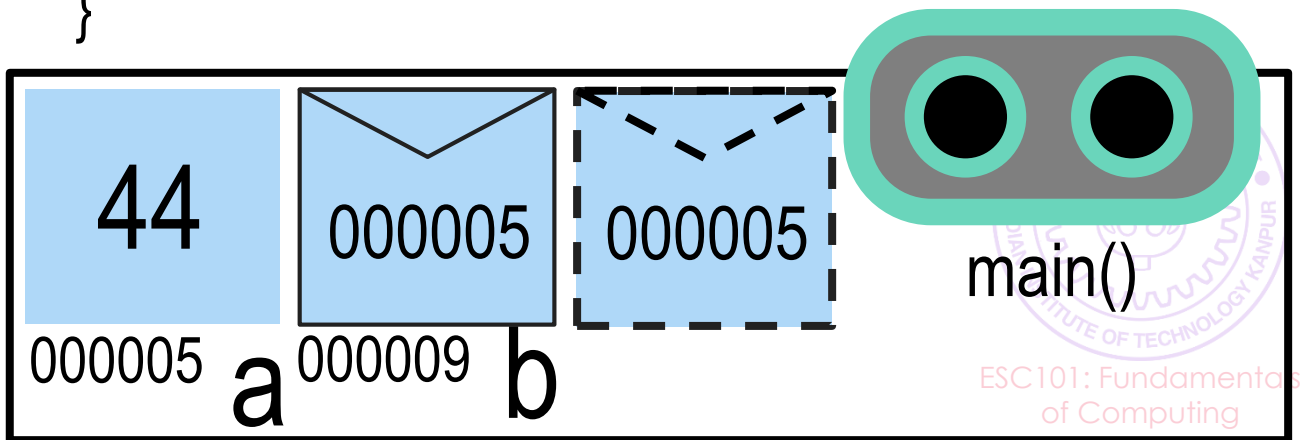
- Rules 1, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-pointer*

Case 2: the input to the function is an expression that generates an address value

- Rules 2, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-reference*

```
void prn(int *ptr){
    printf("%d", *ptr);
    *ptr++;
}
int main(void){
    int a = 42;
    int *b= &a;
    prn(b);
    prn(&a);
    return 0;
}
```





Passing pointers/addresses

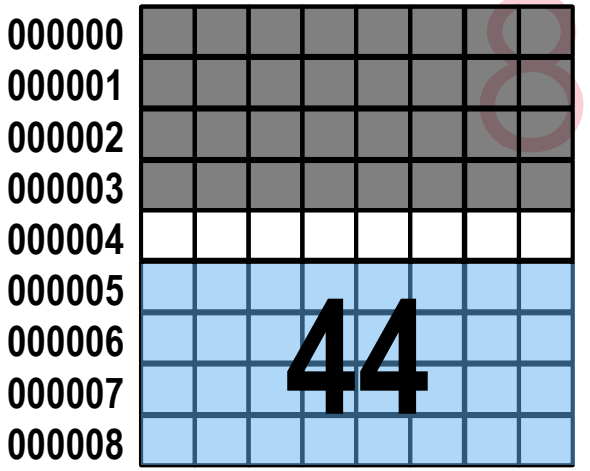
Case 1: the input to the function is a pointer variable

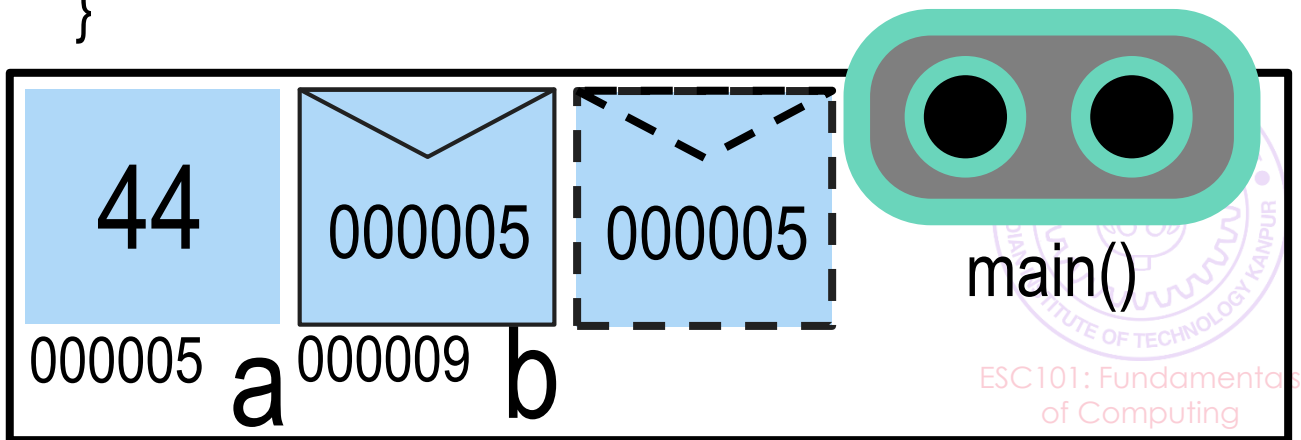
- Rules 2, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-pointer*

Case 2: the input to the function is an expression that generates an address value

- Rules 2, 4 and 6 apply here (remember – rule 4 always applies)
- Books/websites call this technique *pass-by-reference*

```
void prn(int *ptr){
    printf("%d", *ptr);
    *ptr++;
}
int main(void){
    int a = 42;
    int *b= &a;
    prn(b);
    prn(&a);
    return 0;
}
```





Passing … esses

Ca… t…
…n is a pointer variable

Rule… …and 6 apply here
(re…mber – rule 4 always applies)

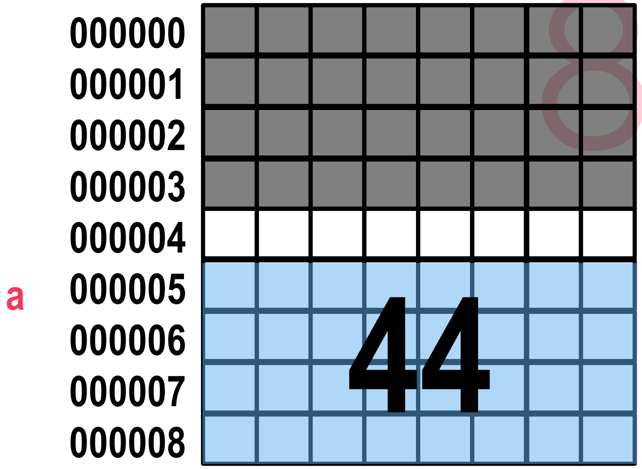
Books/websites call this technique *pass-by-pointer*

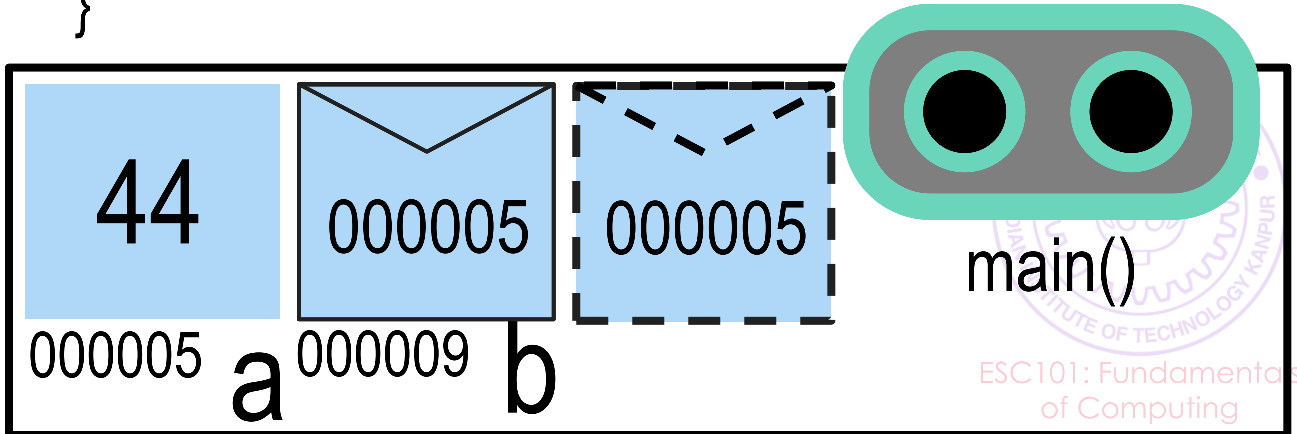
Case 2: the input to the function is an expression that generates an address value

Rules 2, 4 and 6 apply here
(remember – rule 4 always applies)

Books/websites call this technique *pass-by-reference*

```
…nt *ptr){
    …%d", *ptr);
    *ptr++;
}
int main(void){
    int a = 42;
    int *b= &a;
    prn(b);
    prn(&a);
    return 0;
}
```





Summary

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

Summary

We have seen how normal variables (int, float, char) can be passed to functions (rule 1) and how expressions of these (rule 2) can be passed to functions

Summary

We have seen how normal variables (int, float, char) can be passed to functions (rule 1) and how expressions of these (rule 2) can be passed to functions

- Sometimes called pass-by-value

Summary

We have seen how normal variables (int, float, char) can be passed to functions (rule 1) and how expressions of these (rule 2) can be passed to functions

- Sometimes called pass-by-value

We have seen how pointers (rule 1) and expressions that generate addresses (rule 2) can be passed to functions

Summary

We have seen how normal variables (int, float, char) can be passed to functions (rule 1) and how expressions of these (rule 2) can be passed to functions

- Sometimes called pass-by-value

We have seen how pointers (rule 1) and expressions that generate addresses (rule 2) can be passed to functions

- Sometimes called pass-by-pointer or pass-by-reference

Summary

We have seen how normal variables (int, float, char) can be passed to functions (rule 1) and how expressions of these (rule 2) can be passed to functions

- Sometimes called pass-by-value

We have seen how pointers (rule 1) and expressions that generate addresses (rule 2) can be passed to functions

- Sometimes called pass-by-pointer or pass-by-reference

Remember - rule 4 always applies, no matter what!

Summary

We have seen how normal variables (int, float, char) can be passed to functions (rule 1) and how expressions of these (rule 2) can be passed to functions

- Sometimes called pass-by-value

We have seen how pointers (rule 1) and expressions that generate addresses (rule 2) can be passed to functions

- Sometimes called pass-by-pointer or pass-by-reference

Remember - rule 4 always applies, no matter what!

Will see pass-by-array tomorrow

Summary

We have seen how normal variables (int, float, char) can be passed to functions (rule 1) and how expressions of these (rule 2) can be passed to functions

- Sometimes called pass-by-value

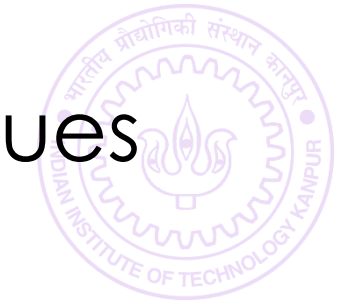
We have seen how pointers (rule 1) and expressions that generate addresses (rule 2) can be passed to functions

- Sometimes called pass-by-pointer or pass-by-reference

Remember - rule 4 always applies, no matter what!

Will see pass-by-array tomorrow

For now, let us see a neat trick to return multiple values



Returning more than one value

Returning more than one value

Can trick Mr C into returning more than one value

Returning more than one value

Can trick Mr C into returning more than one value

METHOD: pass the address of a variable and ask the function to directly modify that variable – exploit rule 6

Returning more than one value

Can trick Mr C into returning more than one value

METHOD: pass the address of a variable and ask the function to directly modify that variable – exploit rule 6

- Does not "return" the variable in strictest sense of the term but effectively we get back more values from the function

Returning more than one value

Can trick Mr C into returning more than one value

METHOD: pass the address of a variable and ask the function to directly modify that variable – exploit rule 6

- Does not "return" the variable in strictest sense of the term but effectively we get back more values from the function
- Advantage: can return different datatypes using this trick

Returning more than one value

Can trick Mr C into returning more than one value

METHOD: pass the address of a variable and ask the function to directly modify that variable – exploit rule 6

- Does not "return" the variable in strictest sense of the term but effectively we get back more values from the function
- Advantage: can return different datatypes using this trick
- Advantage: can return multiple arrays using this trick ☺ (wait for next class)

Returning more than one value

Can trick Mr C into returning more than one value

METHOD: pass the address of a variable and ask the function to directly modify that variable – exploit rule 6

- Does not "return" the variable in strictest sense of the term but effectively we get back more values from the function
- Advantage: can return different datatypes using this trick
- Advantage: can return multiple arrays using this trick ☺ (wait for next class)
- Disadvantage: have to be careful handling pointers, memory leaks

Returning more than one value

Can trick Mr C into returning more than one value

METHOD: pass the address of a variable and ask the function to directly modify that variable – exploit rule 6

- Does not "return" the variable in strictest sense of the term but effectively we get back more values from the function
- Advantage: can return different datatypes using this trick
- Advantage: can return multiple arrays using this trick ☺ (wait for next class)
- Disadvantage: have to be careful handling pointers, memory leaks

Think of this technique as a controlled way of cheating

Returning more than one value

Can trick Mr C into returning more than one value

METHOD: pass the address of a variable and ask the function to directly modify that variable – exploit rule 6

- Does not "return" the variable in strictest sense of the term but effectively we get back more values from the function
- Advantage: can return different datatypes using this trick
- Advantage: can return multiple arrays using this trick ☺ (wait for next class)
- Disadvantage: have to be careful handling pointers, memory leaks

Think of this technique as a controlled way of cheating

- **Government rule**: to get passport, only INR 100 can be paid to official

Returning more than one value

Can trick Mr C into returning more than one value

METHOD: pass the address of a variable and ask the function to directly modify that variable – exploit rule 6

- Does not "return" the variable in strictest sense of the term but effectively we get back more values from the function
- Advantage: can return different datatypes using this trick
- Advantage: can return multiple arrays using this trick ☺ (wait for next class)
- Disadvantage: have to be careful handling pointers, memory leaks

Think of this technique as a controlled way of cheating

- **Government rule**: to get passport, only INR 100 can be paid to official
- **Our desire**: pay INR 500 bribe to get it quickly

Returning more than one value

Can trick Mr C into returning more than one value

METHOD: pass the address of a variable and ask the function to directly modify that variable – exploit rule 6

- Does not "return" the variable in strictest sense of the term but effectively we get back more values from the function
- Advantage: can return different datatypes using this trick
- Advantage: can return multiple arrays using this trick ☺ (wait for next class)
- Disadvantage: have to be careful handling pointers, memory leaks

Think of this technique as a controlled way of cheating

- **Government rule**: to get passport, only INR 100 can be paid to official
- **Our desire**: pay INR 100 for passport but also INR 500 bribe to get it quickly
- **Problem**: Govt rule prevents us from paying more than INR 100 to official

Returning more than one value

Can trick Mr C into returning more than one value

METHOD: pass the address of a variable and ask the function to directly modify that variable – exploit rule 6

- Does not "return" the variable in strictest sense of the term but effectively we get back more values from the function
- Advantage: can return different datatypes using this trick
- Advantage: can return multiple arrays using this trick ☺ (wait for next class)
- Disadvantage: have to be careful handling pointers, memory leaks

Think of this technique as a controlled way of cheating

- **Government rule**: to get passport, only INR 100 can be paid to official
- **Our desire**: pay INR 100 for passport but also INR 500 bribe to get it quickly
- **Problem**: Govt rule prevents us from paying more than INR 100 to official
- **Solution**: Corrupt official gives us his bank account number. We pay official INR 100 the normal route but also directly deposit INR 500 in his bank account

Returning more than one value

Co[illegible]k Mr C into returning more than one value

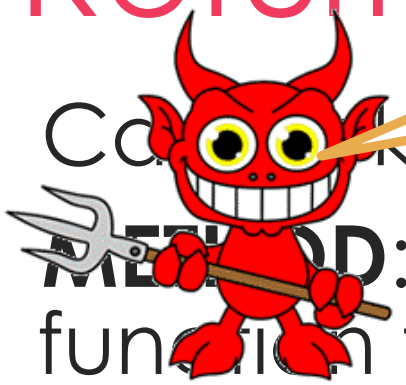
METHOD: pass the address of a variable and ask the function to directly modify that variable – exploit rule 6

- Does not "return" the variable in strictest sense of the term but effectively we get back more values from the function
- Advantage: can return different datatypes using this trick
- Advantage: can return multiple arrays using this trick ☺ (wait for next class)
- Disadvantage: have to be careful handling pointers, memory leaks

Think of this technique as a controlled way of cheating

- **Government rule**: to get passport, only INR 100 can be paid to official
- **Our desire**: pay INR 100 for passport but also INR 500 bribe to get it quickly
- **Problem**: Govt rule prevents us from paying more than INR 100 to official
- **Solution**: Corrupt official gives us his bank account number. We pay official INR 100 the normal route but also directly deposit INR 500 in his bank account

Returning more than one value



The govt rule on paying only INR 100 is like Mr C's rule of returning only one variable

Ca[illegible] value

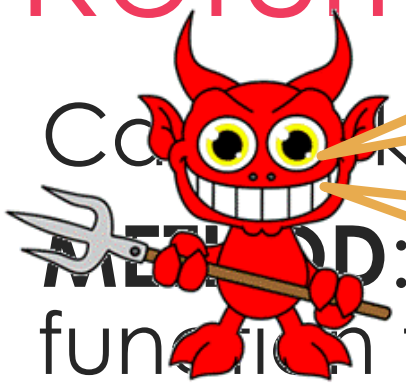
METHOD: pass the address of a variable and ask the function to directly modify that variable – exploit rule 6

- Does not "return" the variable in strictest sense of the term but effectively we get back more values from the function
- Advantage: can return different datatypes using this trick
- Advantage: can return multiple arrays using this trick ☺ (wait for next class)
- Disadvantage: have to be careful handling pointers, memory leaks

Think of this technique as a controlled way of cheating

- **Government rule**: to get passport, only INR 100 can be paid to official
- **Our desire**: pay INR 500 for passport but also INR 500 bribe to get it quickly
- **Problem**: Govt rule prevents us from paying more than INR 100 to official
- **Solution**: Corrupt official gives us his bank account number. We pay official INR 100 the normal route but also directly deposit INR 500 in his bank account

Returning more than one value



The govt rule on paying only INR 100 is like Mr C's rule of returning only one variable

The official giving us his account number is like main() passing the address of a variable to be modified directly

Ca[illegible] value

METHOD: p[illegible] e function to directly modify that variable – explo[illegible]le 6

Does not "return" the variable in strictest sense of the term but effectively we get back more values from the function

Advantage: can return different datatypes using this trick

Advantage: can return multiple arrays using this trick ☺ (wait for next class)

Disadvantage: have to be careful handling pointers, memory leaks

Think of this technique as a controlled way of cheating

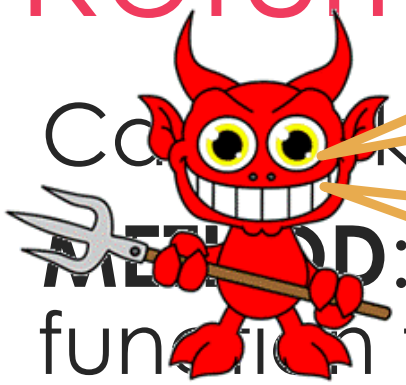
Government rule: to get passport, only INR 100 can be paid to official

Our desire: pay INR 100 for passport but also INR 500 bribe to get it quickly

Problem: Govt rule prevents us from paying more than INR 100 to official

Solution: Corrupt official gives us his bank account number. We pay official INR 100 the normal route but also directly deposit INR 500 in his bank account

Returning more than one value



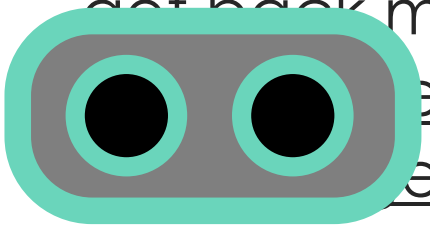
The govt rule on paying only INR 100 is like Mr C's rule of returning only one variable

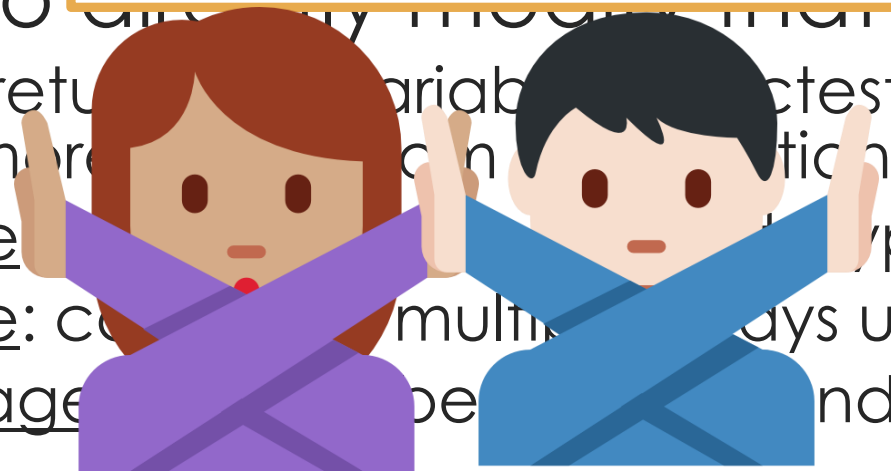
The official giving us his account number is like main() passing the address of a variable to be modified directly

Ca… Mr C like returning more than one value

METHOD: p… e function to already modified that variable – explo… rule 6

- Does not "retu… ariab… ctest sense of the term but effectively we get back m… n … tion
- …e … types using this trick
- …e: co… mult… ays using this trick ☺ (wait for next class)
- Disadvantage… pe… ndling pointers, memory leaks





Think of this technique as a controlled way of cheating

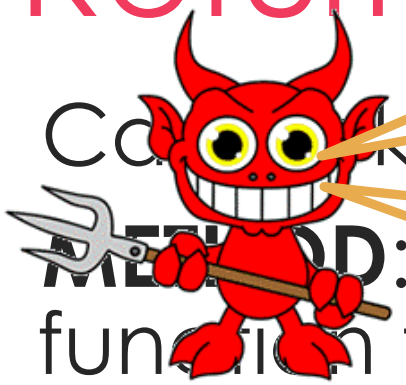
Government rule: to get passport, only INR 100 can be paid to official

Our desire: pay INR 100 for passport but also INR 500 bribe to get it quickly

Problem: Govt rule prevents us from paying more than INR 100 to official

Solution: Corrupt official gives us his bank account number. We pay official INR 100 the normal route but also directly deposit INR 500 in his bank account

Returning more than one value



The govt rule on paying only INR 100 is like Mr C's rule of returning only one variable

The official giving us his account number is like main() passing the address of a variable to be modified directly

Never pay bribes, kickbacks in real life

Ca… Mr C's rule returning more than one value

METHOD: p… e function to already modified that variable – explain rule 6

Does not "retu… ariab… ctest s… effectively we get back m… n… tio…

…e… types using this trick

…e: c… multi… ays using this trick ☺ (wait for next class)

Disadvantage… pe… ndling pointers, memory leaks

Think of this technique as a controlled way of cheating

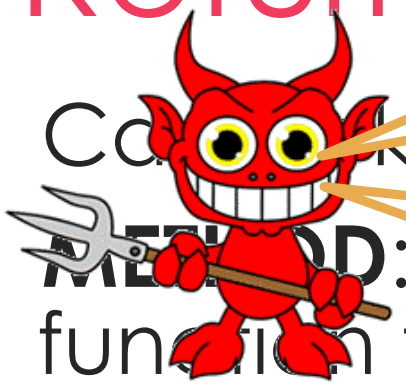
Government rule: to get passport, only INR 100 can be paid to official

Our desire: pay INR 500 bribe to get it quickly

Problem: Govt rule prevents us from paying more than INR 100 to official

Solution: Corrupt official gives us his bank account number. We pay official INR 100 the normal route but also directly deposit INR 500 in his bank account

Returning more than one value



The govt rule on paying only INR 100 is like Mr C's rule of returning only one variable

The official giving us his account number is like main() passing the address of a variable to be modified directly

Ca… Mr C like returning more than one value

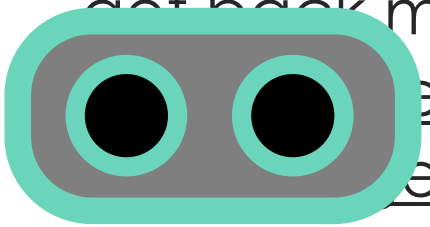
METHOD: p… e function to already modified that variable – explain rule 6

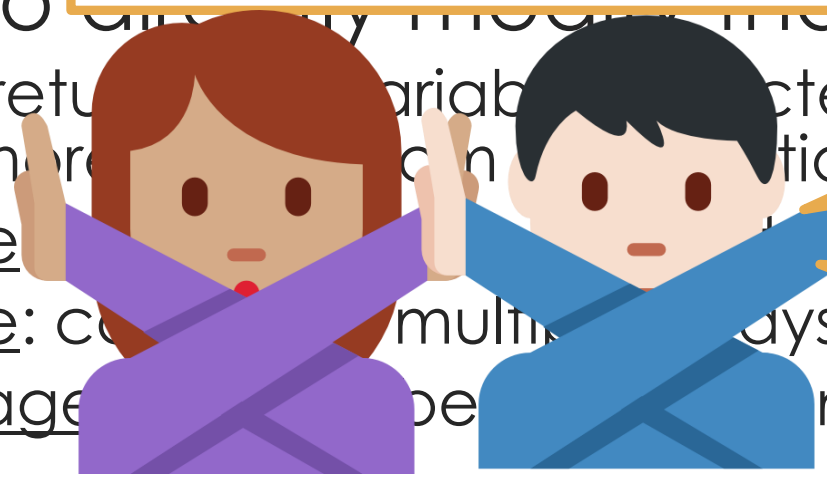
Does not "retu… variable… ctest s… effectively we get back m… n… tion

…e… types using this trick

…e: c… multi… days usi… next class)

Disadvantage… pe… ndlin… eaks





Never pay bribes, kickbacks in real life

Corrupt behaviour is detrimental to society

Think of this technique as a controlled way of cheating

Government rule: to get passport, only INR 100 can be paid to official

Our desire: pay INR 100 for passport but also INR 500 bribe to get it quickly

Problem: Govt rule prevents us from paying more than INR 100 to official

Solution: Corrupt official gives us his bank account number. We pay official INR 100 the normal route but also directly deposit INR 500 in his bank account