Introduction to Programming (C/C++)

03: C Memory

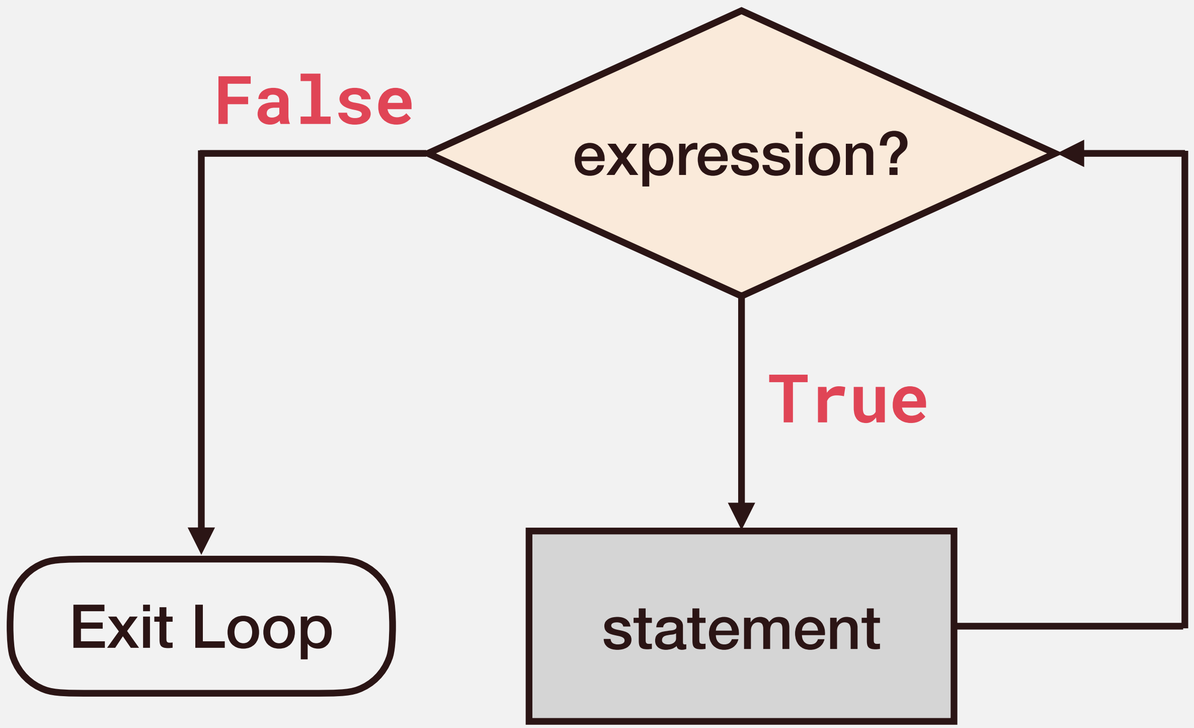
Huanchen Zhang

清華大學
Tsinghua University

交叉信息研究院
Institute for Interdisciplinary
Information Sciences

Loops

```
while (expression)
   statement
```



Task: Given a positive integer n user, print out “#” n times

```
int i = 0;
```

Task: Given a positive integer n user, print out “#” n times

```
int i = 0;
while () {
  printf(“#”);
}
```

Task: Given a positive integer n user, print out “#” n times

```
int i = 0;
while (1) {
  printf(“#”);
}
```

Task: Given a positive integer n user, print out “#” n times

```
int i = 0;
while (i <= n) {
  printf(“#”);
}
```

Task: Given a positive integer n user, print out “#” n times

```
int i = 0;
while (i <= n) {
  printf(“#”);
  i++;
}
```

Task: Given a positive integer n user, print out “#” n times

```
int i = 0;
while (i < n) {
  printf(“#”);
  i++;
}
```

Task: Given a positive integer n user, print out “#” n times

```
int i = 0;
while (i < n) { // easy to have off-by-one error here
  printf(“#”);
  i++;
}
```

Task: Given a positive integer n user, print out “#” n times

```
int i = 0;
while (i < n) { // easy to have off-by-one error here
  printf(“#”);
  i++; // infinite loop if this is forgotten
}
```

Be careful about infinite-loops and off-by-one errors

For Loops

```
for (i = 0; i < n; i++)
  statement
```

For Loops

```
for (i = 0; i < n; i++)
  statement
```

← Execute this statement **n** times

For Loops

```
for (e_init; e_test; e_modify)
  statement
```

For Loops

```
for (e_init; e_test; e_modify)
  statement
```

For Loops

```
for (e_init; e_test; e_modify)
  statement
```

For Loops

```
for (e_init; e_test; e_modify)
  statement
```

For Loops

```
for (e_init; e_test; e_modify)
  statement
```

For Loops

```
for (e_init; e_test; e_modify)
  statement
```

For Loops

```
for (e_init; e_test; e_modify)
  statement
```

For Loops

```
for (e_init; e_test; e_modify)
  statement
```

For Loops

```
for (e_init; e_test; e_modify)
  statement
```

For Loops

```
for (e_init; e_test; e_modify)
  statement

...
```

For Loops

```
for (e_init; e_test; e_modify)
  statement

...

e_init;
```

For Loops

```
for (e_init; e_test; e_modify)
  statement

...

e_init;
while (e_test) {
```

For Loops

```
for (e_init; e_test; e_modify)
  statement

...

e_init;
while (e_test) {
  statement
```

For Loops

```
for (e_init; e_test; e_modify)
  statement

...

e_init;
while (e_test) {
  statement
  e_modify
}
```

Task: Given a positive integer n user, print out “#” n times

```
int i = 0;
while (i < n) {
  printf(“#”);
  i++;
}
```

Task: Given a positive integer n user, print out “#” n times

```
int i = 0;
while (i < n) {
  printf(“#”);
  i++;
}

for (i = 0; i < n; i++) {
  printf(“#”);
}
```

Exercise: print out a pyramid

n rows

```
   #
  # #
 # # #
# # # #
# # # # #
```

space

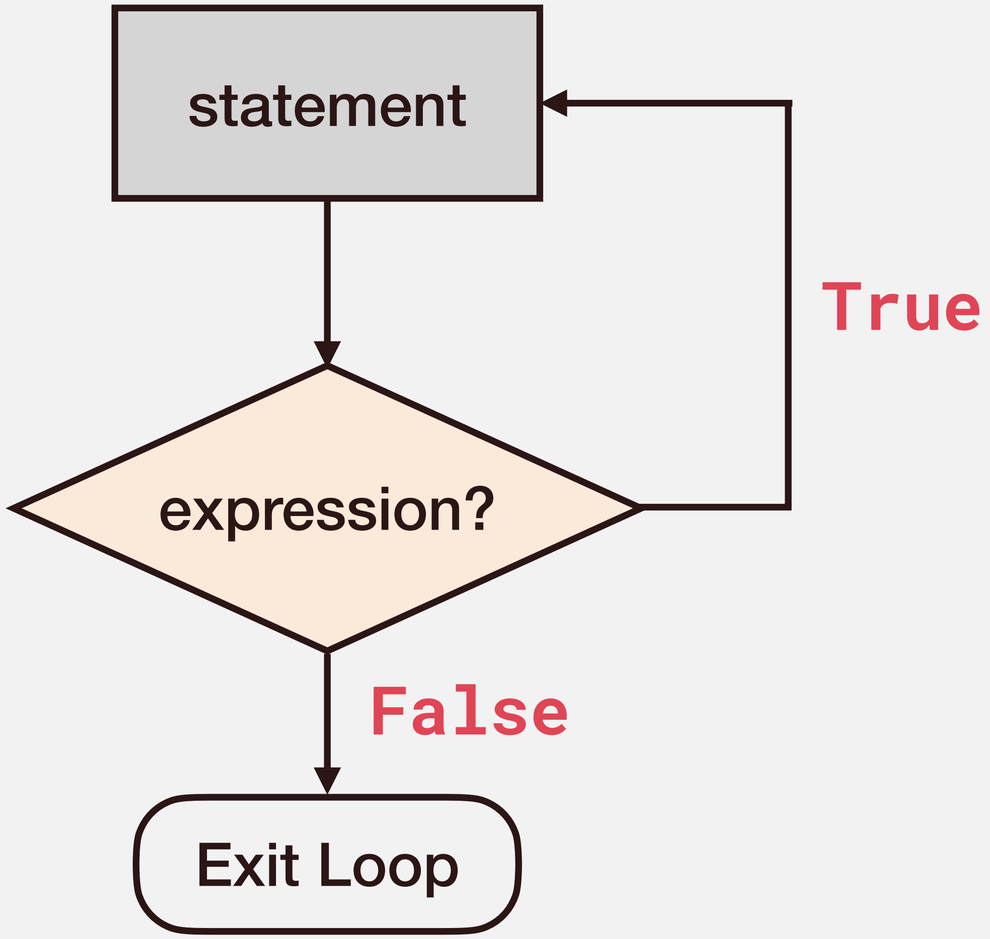
Exercise: print out a pyramid

n rows

```
,,,,#
,,,#,#
,,#,#,#
,#,#,#,#
#,#,#,#,#
```

Loops

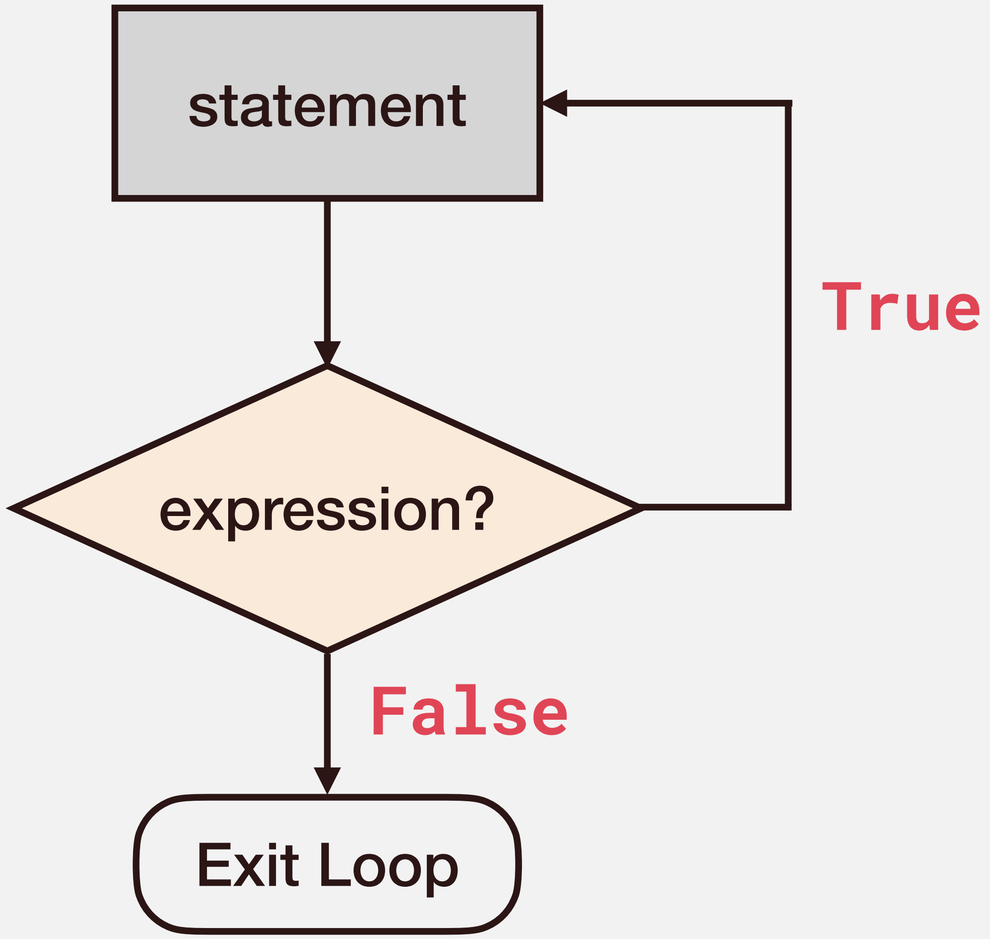
```
do
   statement
while (expression)
```



Loops

```
do
   statement
while (expression)
```

Less Used



Early-Exit

```
while (expression) {
   ...
   break;
   ...
}
...
```

Early-Exit

```
while (expression) {
   ...
   break;
   ...
}
...
```

Early-Exit

```
while (expression) {
  ...
  if (condition)
    break;
  ...
}
...
```

Early-Exit

```
for (e_init; e_test; e_modify) {
  ...
  if (condition)
    break;
  ...
}
...
```

Early-Exit

```
for (e_init; e_test; e_modify) {
  ...
  if (condition)
    break;
  ...
}
...
```

Used in **loop** and **switch** **only**

Early-Exit

```
int n = 0;
int sum = 0;
for (i = 0; i < 10; i++) {
  scanf("%d", &n);
  if (n < 0)
    break;
  sum += n;
}
```

Early-Exit

```
int n = 0;
int sum = 0;
for (i = 0; i < 10; i++) {
  scanf("%d", &n);
  if (n < 0)
    break;
  sum += n;
}
```

← **Exit** loop upon negative input

Next Iteration

```
while (expression) {
   ...
   continue;
   ...
}
```

Next Iteration

```
while (expression) {
   ...
   continue;
   ...
}
```

Next Iteration

```
while (expression) {
  ...
  if (condition)
    continue;
  ...
}
```

Next Iteration

```
while (expression) {
  ...
  if (condition)
    continue;
  ...
}
```

Used in **loop only**

Next Iteration

```
int n = 0;
int sum = 0;
for (i = 0; i < 10; i++) {
  scanf("%d", &n);
  if (n < 0)
    continue;
  sum += n;
}
```

Next Iteration

```
int n = 0;
int sum = 0;
for (i = 0; i < 10; i++) {
  scanf("%d", &n);
  if (n < 0)
    continue;
  sum += n;
}
```

Skip negative inputs

Control Flow Summary

- Execute with **conditions**
  - `if, else if, else`
  - `switch`
- **Loops**
  - `while, do-while, for`
- Interrupt loops
  - `break, continue`

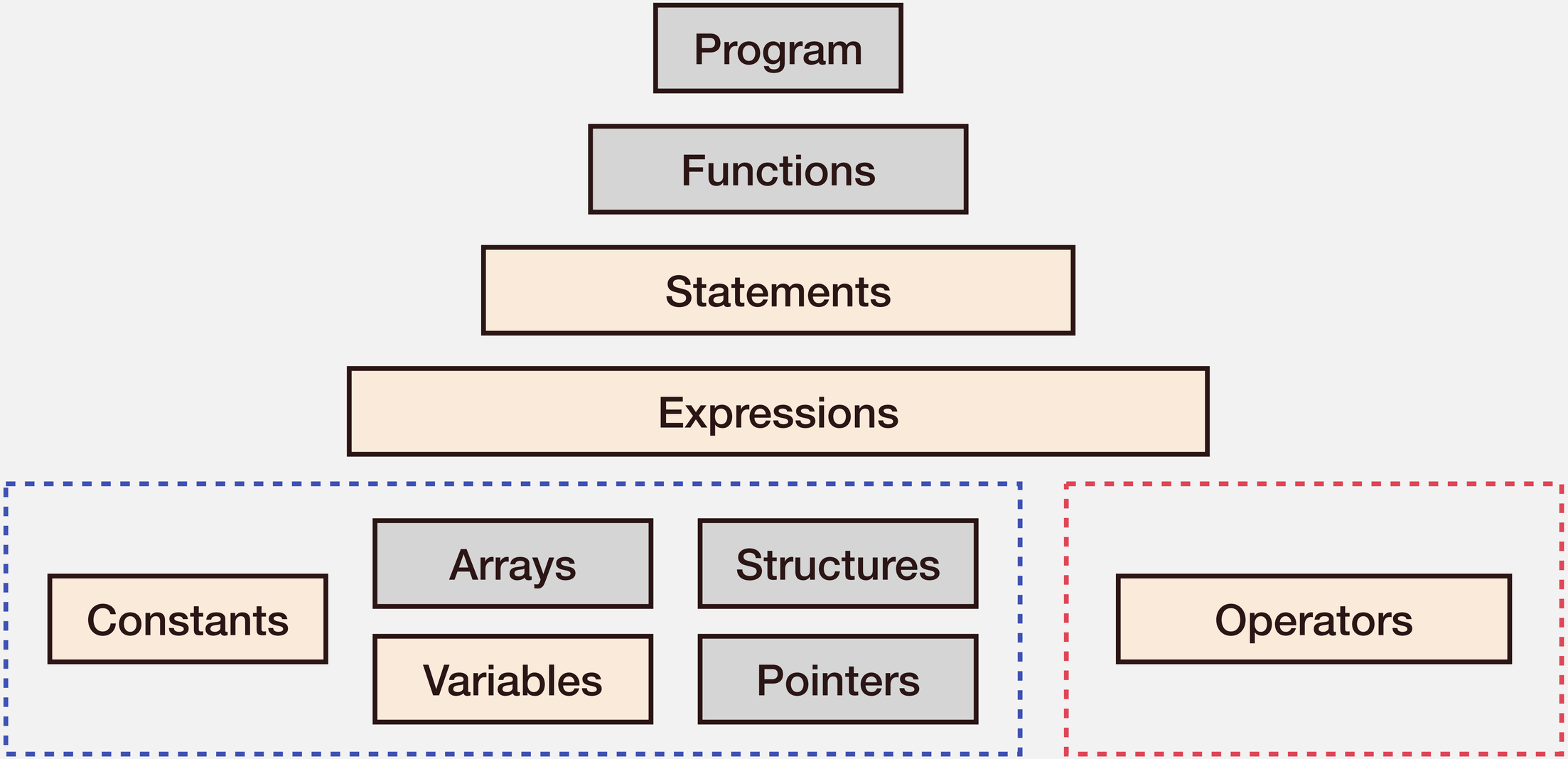
Control Flow Summary

- Execute with **conditions**
  - `if, else if, else`
  - `switch`
- **Loops**
  - `while, do-while, for`
- Interrupt loops
  - `break, continue`
- Jump to arbitrary location
  - `goto`

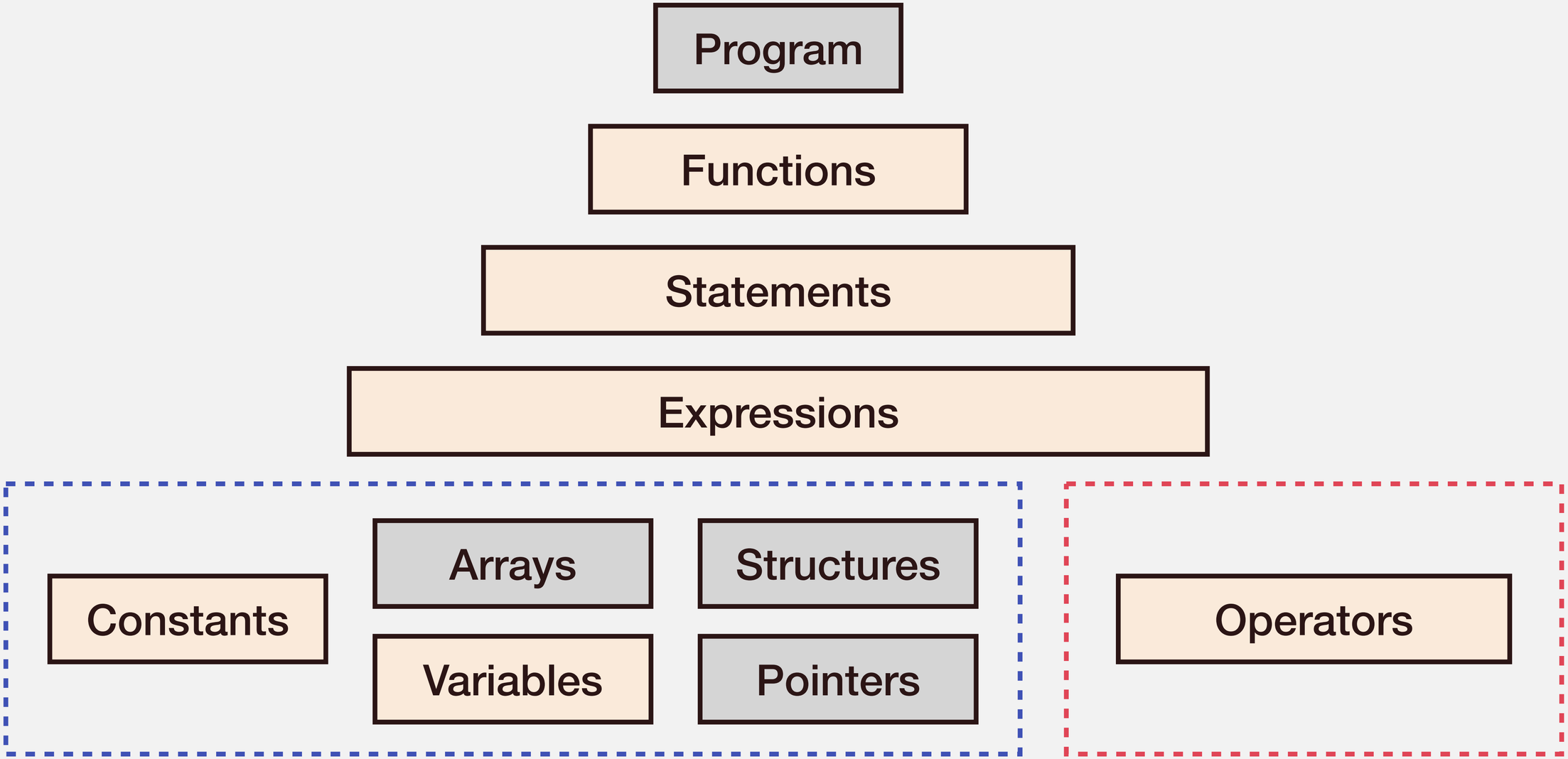
Control Flow Summary

- Execute with **conditions**
  - `if, else if, else`
  - `switch`
- **Loops**
  - `while, do-while, for`
- Interrupt loops
  - `break, continue`
- Jump to arbitrary location **Usually bad**
  - `goto`

Road Map

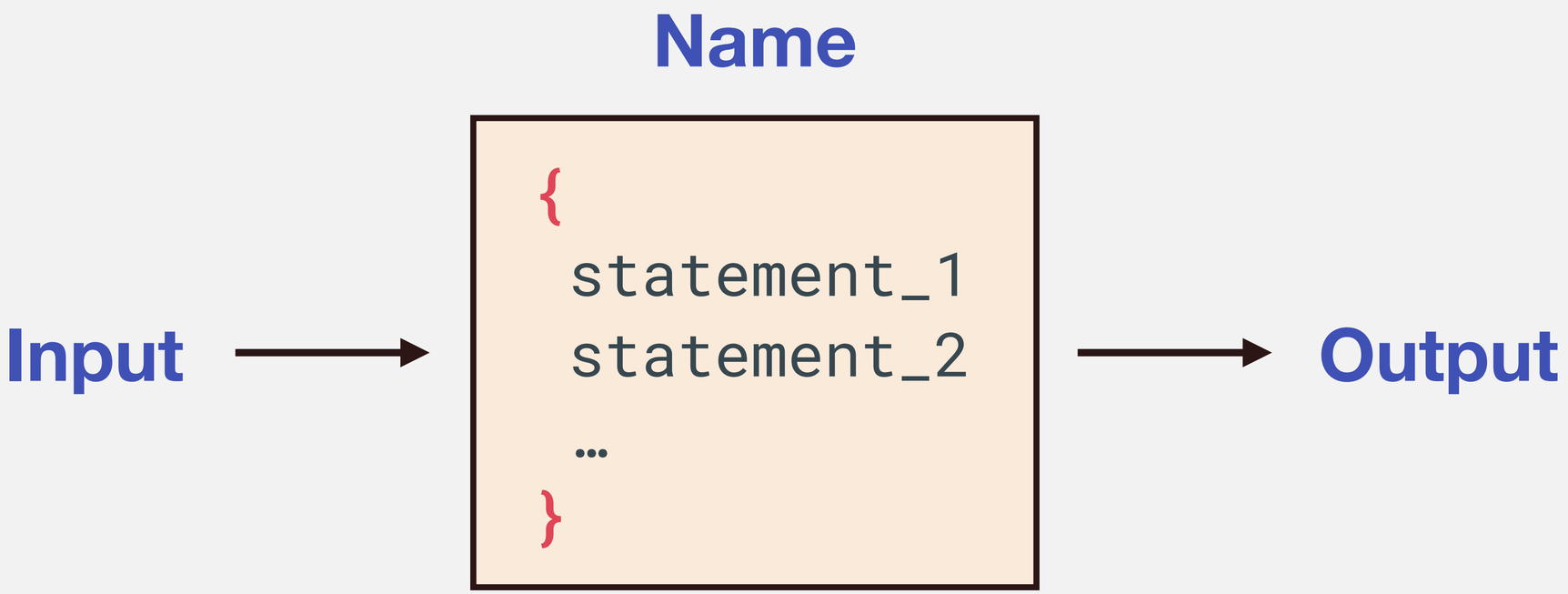

Program
Functions
Statements
Expressions
Constants
Arrays
Variables
Structures
Pointers
Operators

Road Map


Program
Functions
Statements
Expressions
Constants
Arrays
Variables
Structures
Pointers
Operators

Functions

➜ A group of statements that together forms a **task**



Why Functions?

- Break large tasks into **smaller** and more **manageable** pieces

Why Functions?

- ➔ Break large tasks into **smaller** and more **manageable** pieces
- ➔ Facilitate **Reusing** code
  - Built on what others have done

Copy-paste code usually indicates bad design

Why Functions?

- ➔ Break large tasks into **smaller** and more **manageable** pieces
- ➔ Facilitate **Reusing** code
  - Built on what others have done
- ➔ Hide unnecessary details
  - More readable code
  - Easier to make changes

Function Definition

```
return_type func_name(arg declarations) {
  statements
}
```

Function Definition

```
return_type func_name(arg declarations) {
  statements
}

int max2(int x, int y) {
  return ((x > y) ? x : y);
}
```

Function Call

```
int max2(int x, int y) {
  return ((x > y) ? x : y);
}

int main() {
  int a, b;
  scanf("%d%d", &a, &b);
  int c = max2(a, b);
  ...
}
```

Function Call

```
int max2(int x, int y) {
  return ((x > y) ? x : y);
}

int main() {
  int a, b;
  scanf("%d%d", &a, &b);
  int c = max2(a, b);
  ...
}
```

$$f(x, y) = \ldots$$

Function Call

```
int max2(int x, int y) {
  return ((x > y) ? x : y);
}

int main() {
  int a, b;
  scanf("%d%d", &a, &b);
  int c = max2(a, b);
  ...
}
```

$$f(x, y) = \ldots$$

$$c = f(a, b)$$

Function Call

```
int max2(int x, int y) {
  return ((x > y) ? x : y);
}

int main() {
  int a, b;
  scanf("%d%d", &a, &b);
  int c = max2(a, b);
  ...
}
```

$$f(x, y) = \ldots$$

$$x = a$$
$$y = b$$

$$c = f(a, b)$$

Function Call

```
int max2(int x, int y) {
  return ((x > y) ? x : y);
}

int main() {
  int a, b;
  scanf("%d%d", &a, &b);
  int c = max2(a, b);
  ...
}
```

$$f(x, y) = \ldots$$

$$x = a$$
$$y = b$$

$$c = f(a, b)$$

Function Call

```
int max2(int x, int y) {
  return ((x > y) ? x : y);
}

int main() {
  int a, b;
  scanf("%d%d", &a, &b);
  int c = max2(a, b);
  ...
}
```

$$f(x, y) = \ldots$$

$$\boxed{\begin{aligned} x &= a \\ y &= b \end{aligned}}$$

Pass-by-Value
(make a copy)

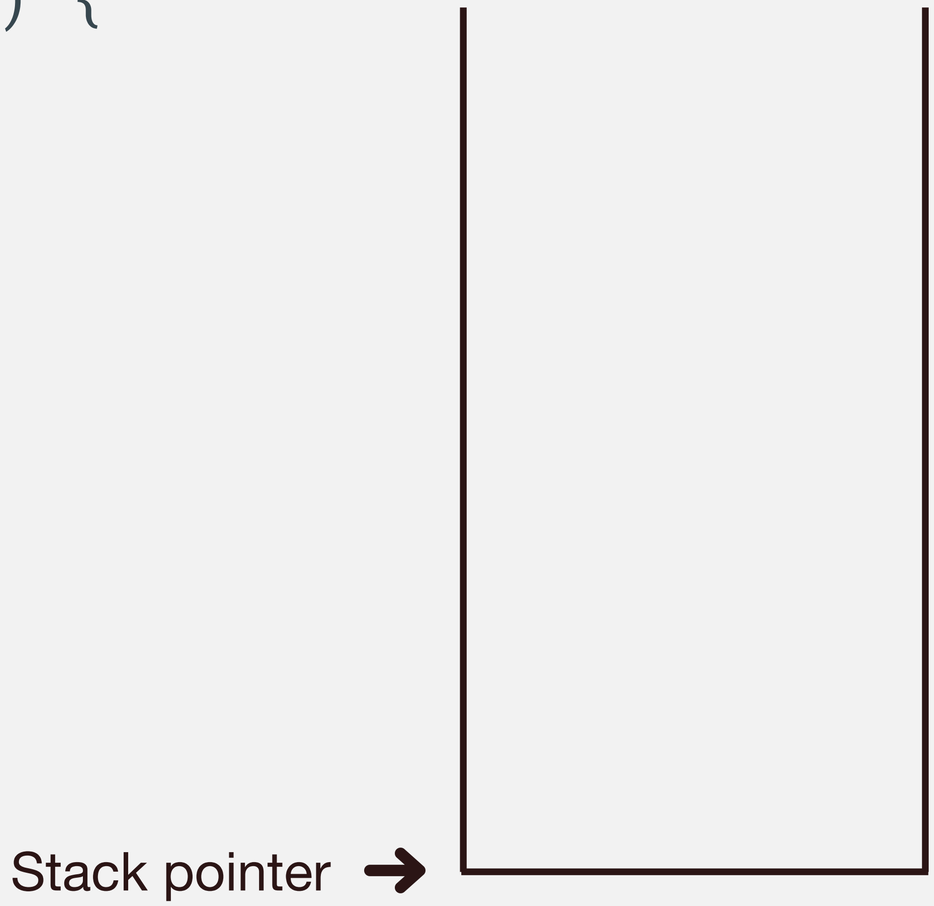
$$c = f(a, b)$$

Swap.c

```c
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
  swap(a, b);
}
```

Call Stack / Stack Memory

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
  swap(a, b);
}
```

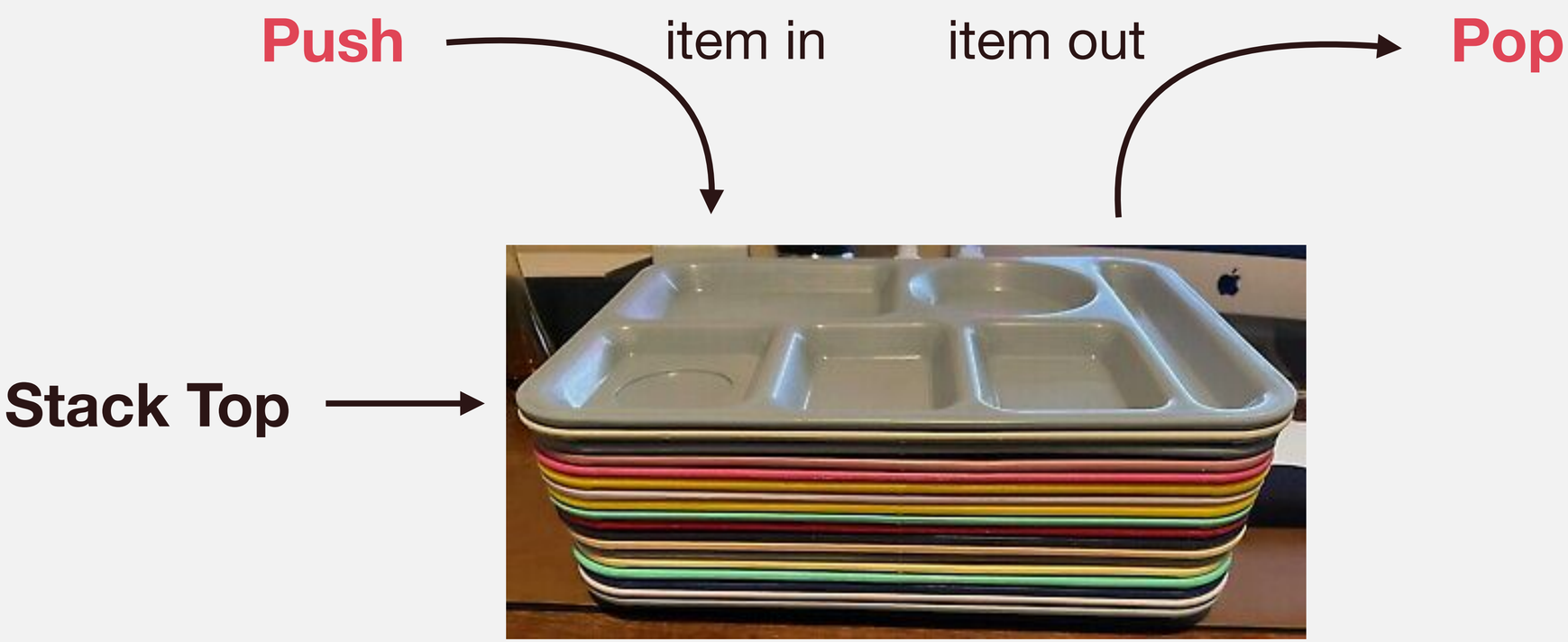


Stack

Stack

Stack Top

Stack



Stack

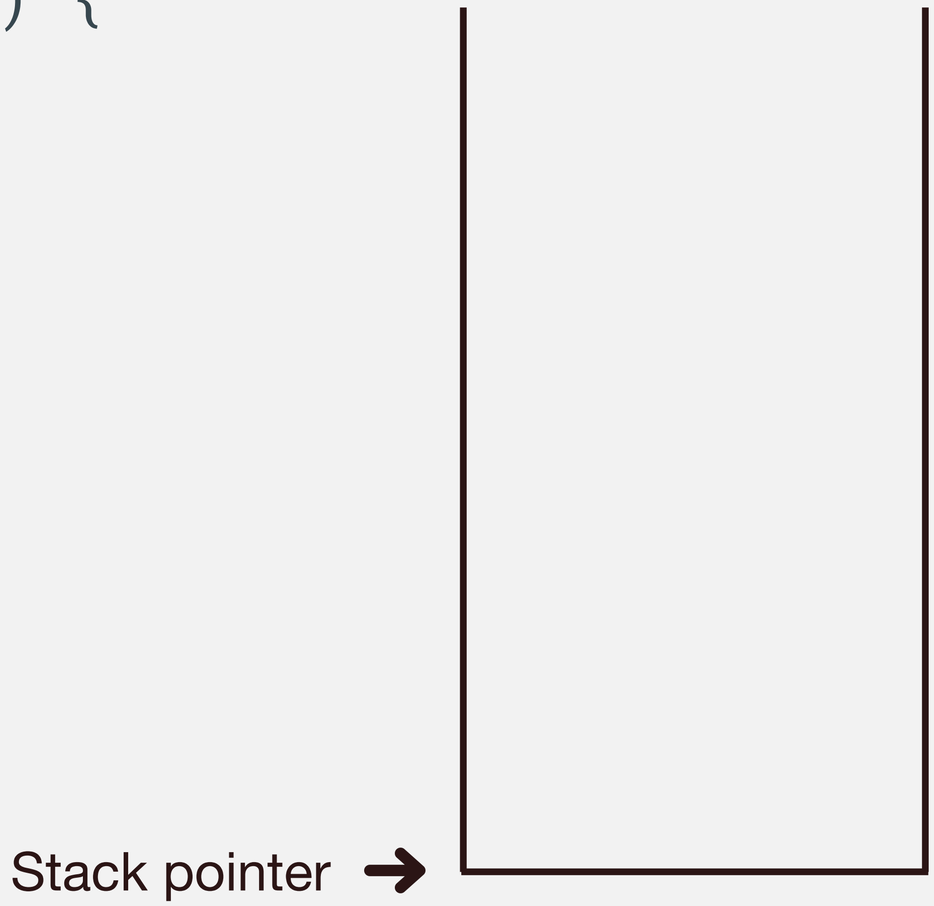
Push item in item out Pop

Stack Top

Last-In-First-Out (LIFO)

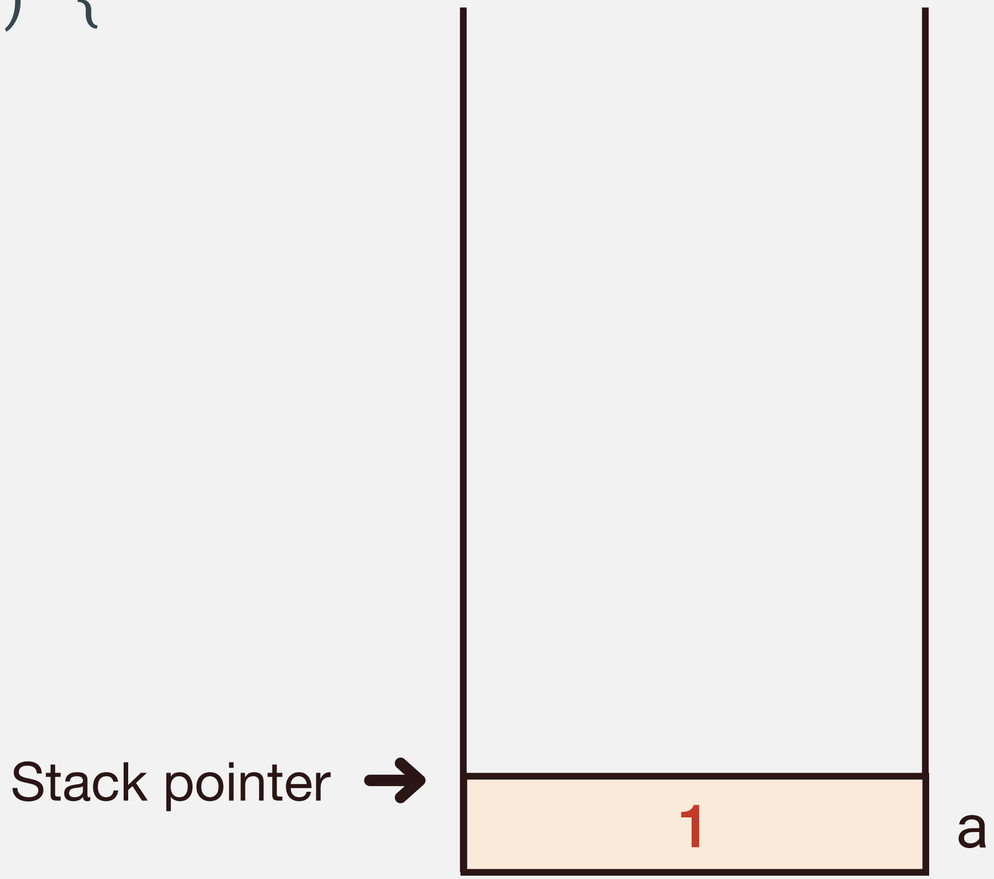
Call Stack

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
➔ int a = 1;
  int b = 2;
  swap(a, b);
}
```



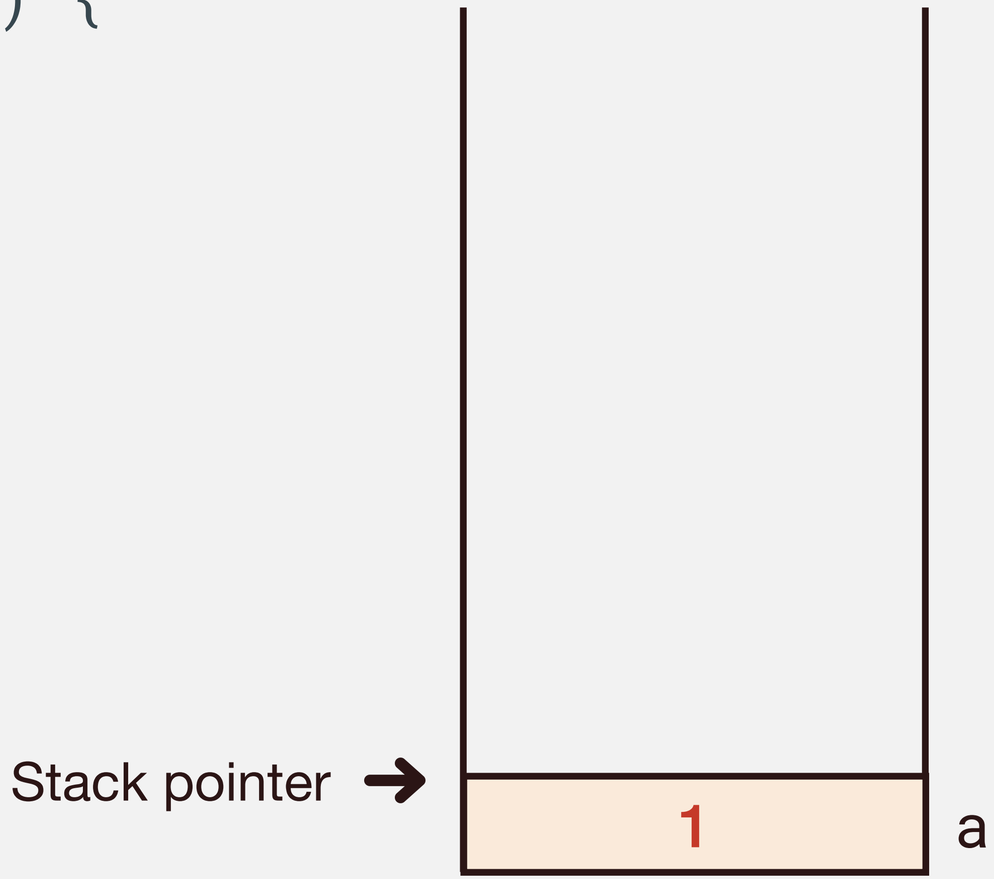
Call Stack

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
➜ int a = 1;
  int b = 2;
  swap(a, b);
}
```



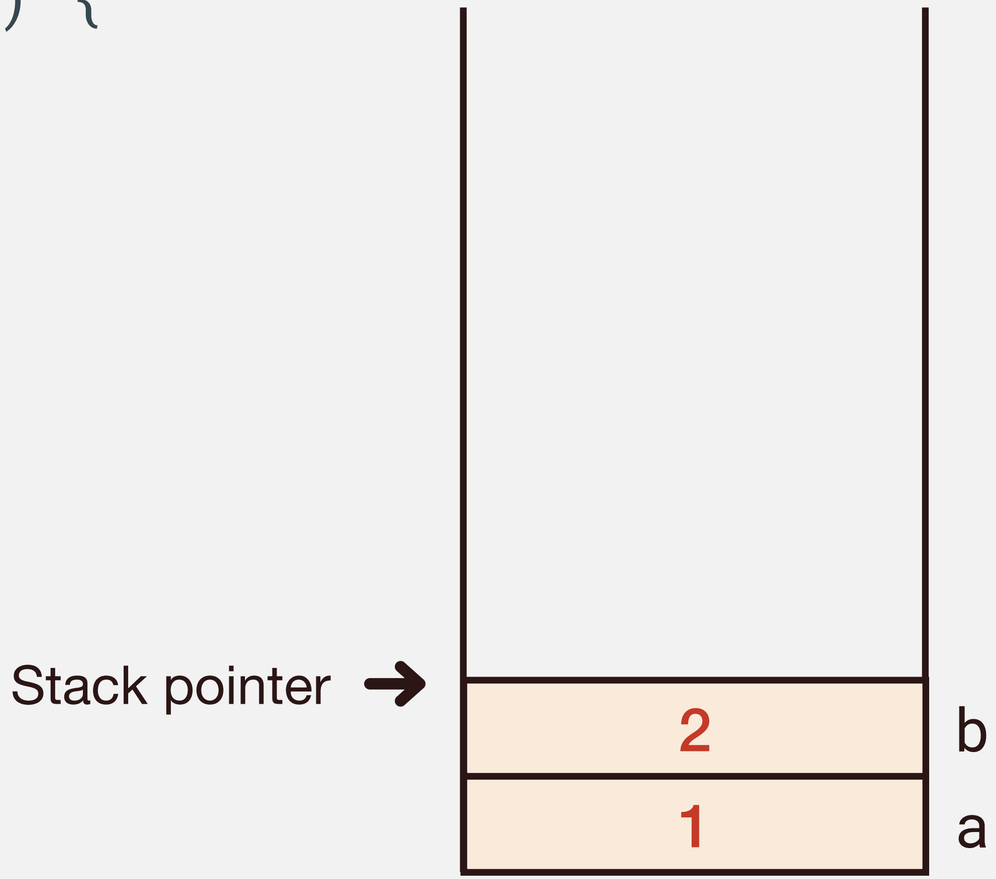
Call Stack

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int a = 1;
➜ int b = 2;
  swap(a, b);
}
```



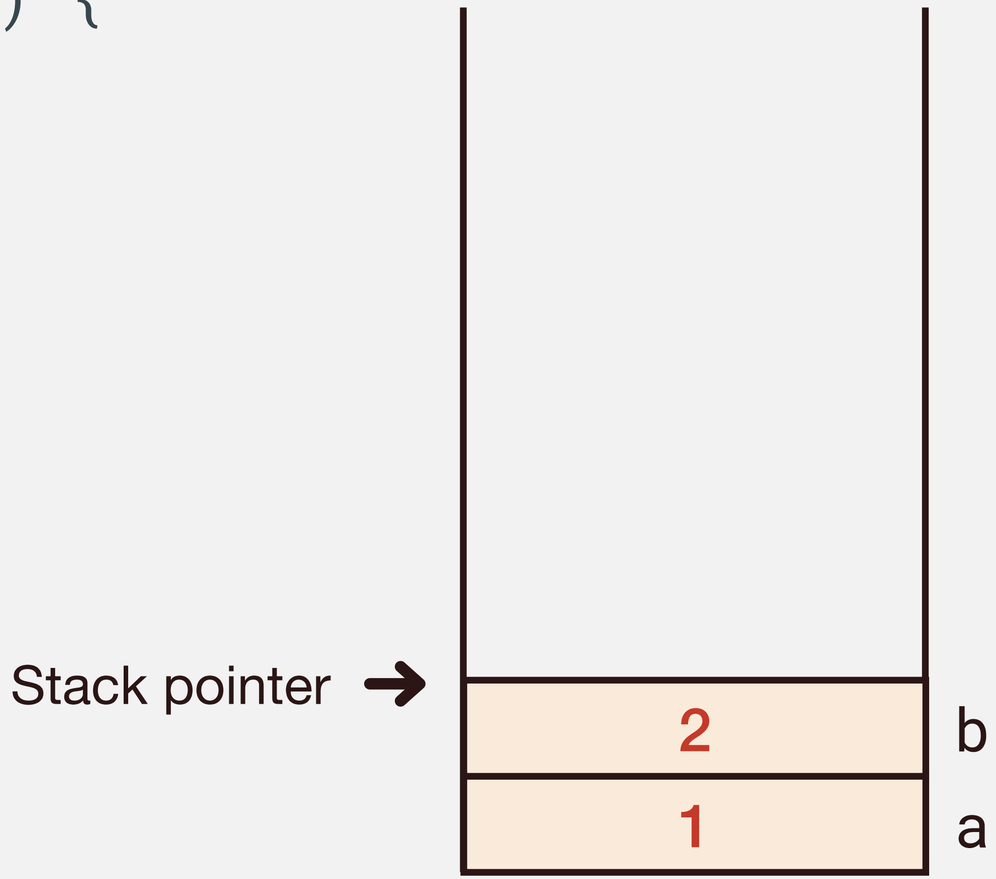
Call Stack

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int a = 1;
→ int b = 2;
  swap(a, b);
}
```



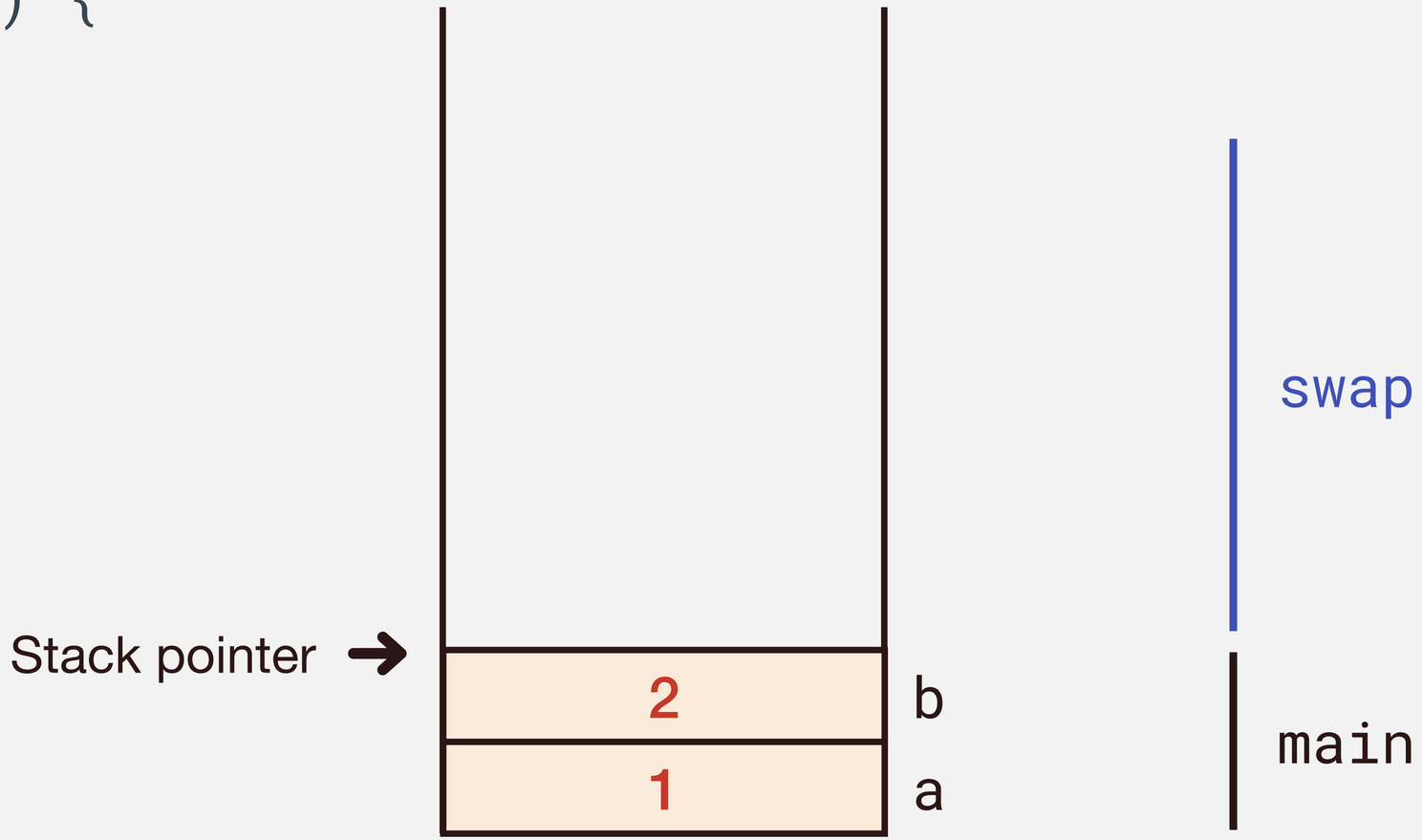
Call Stack

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
➔ swap(a, b);
}
```



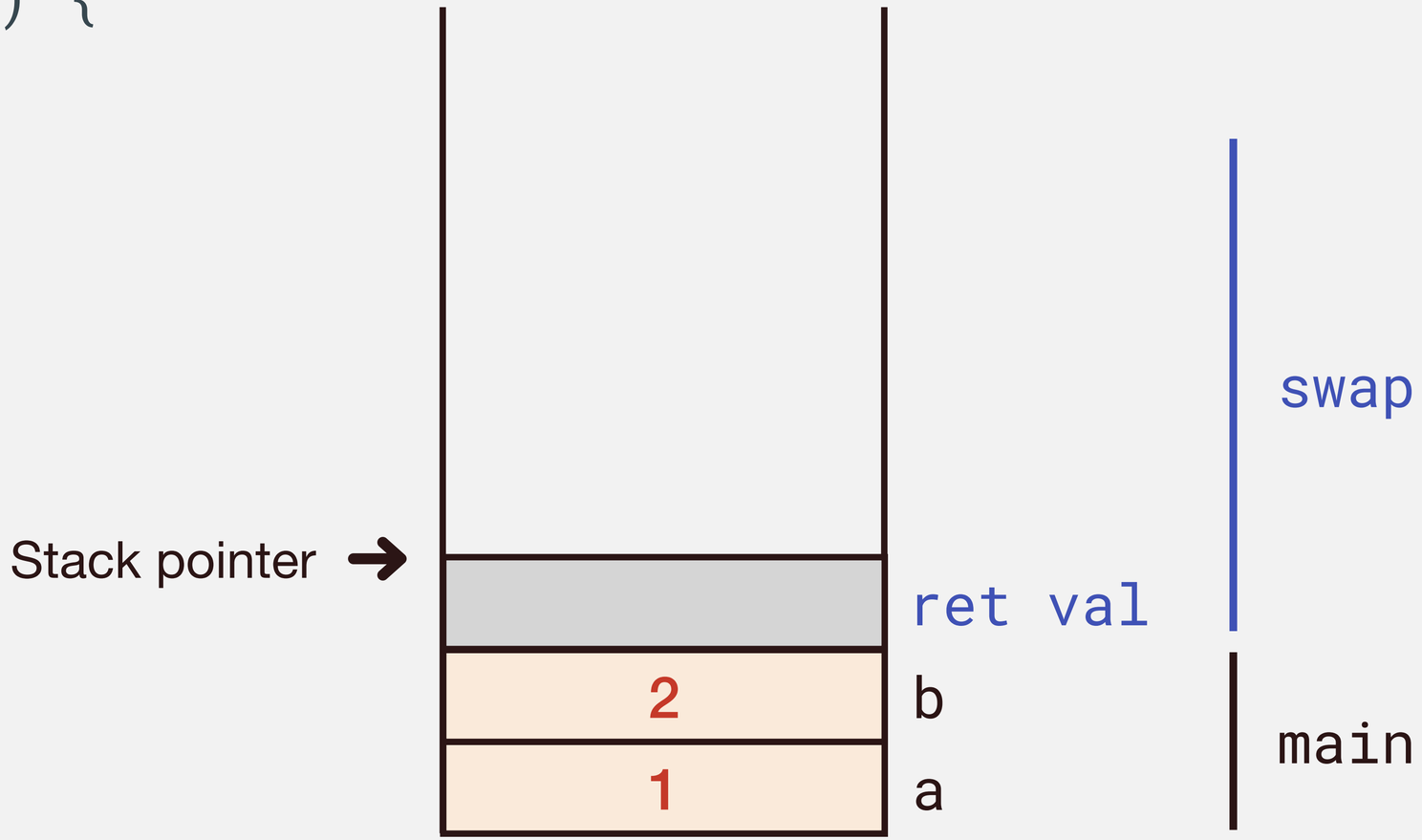
Call Stack

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
➔ swap(a, b);
}
```



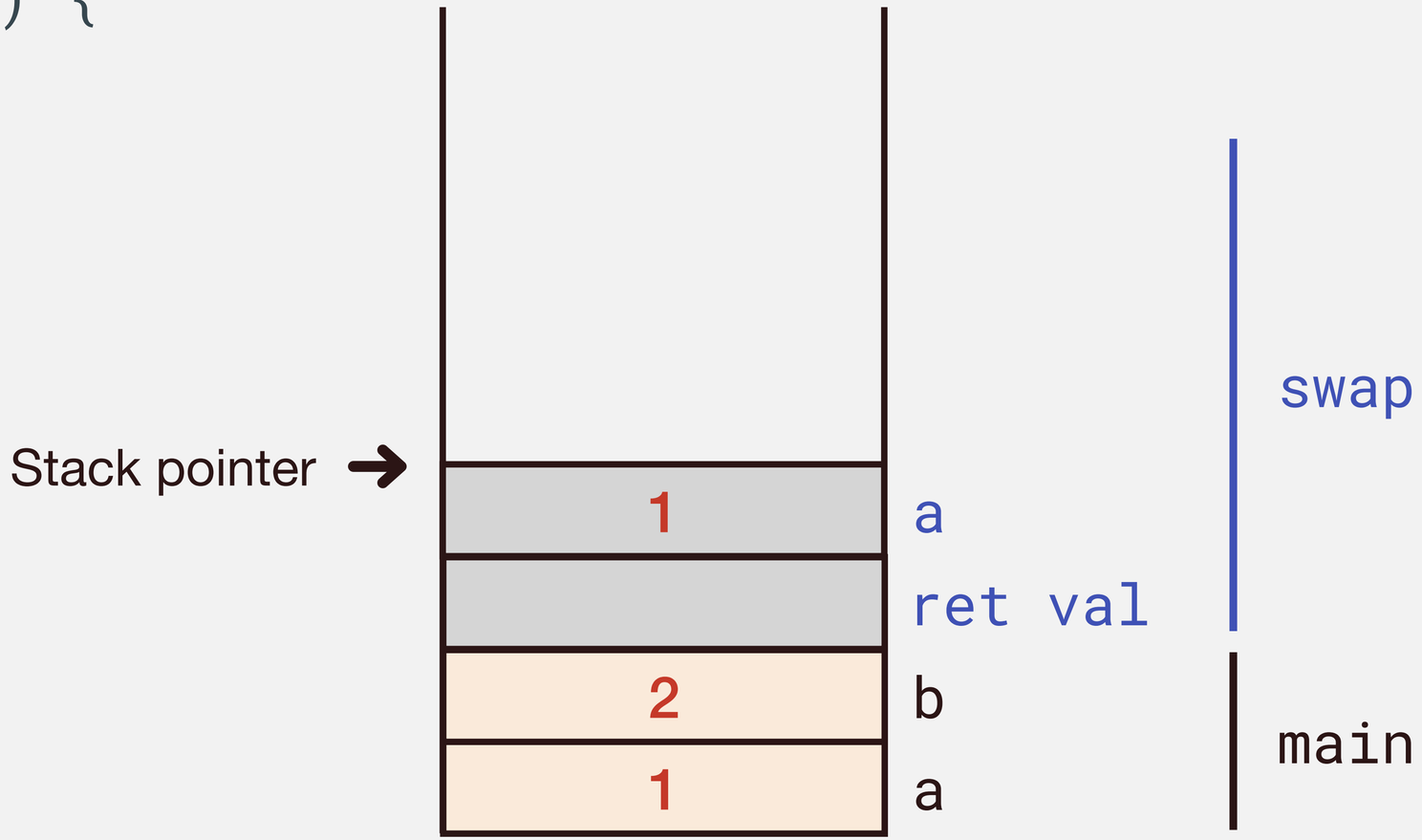
Call Stack

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
➔ swap(a, b);
}
```



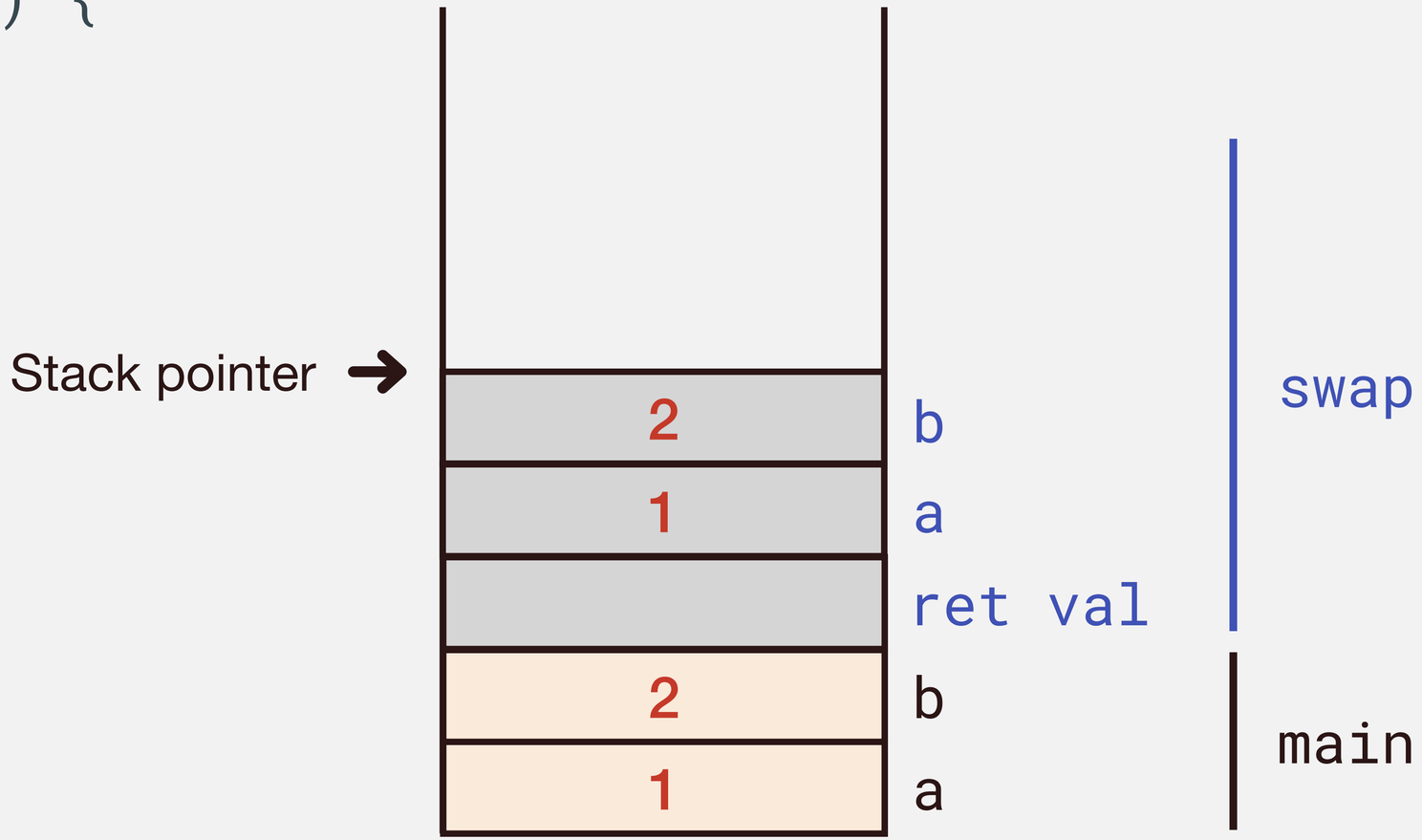
Call Stack

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
➔ swap(a, b);
}
```



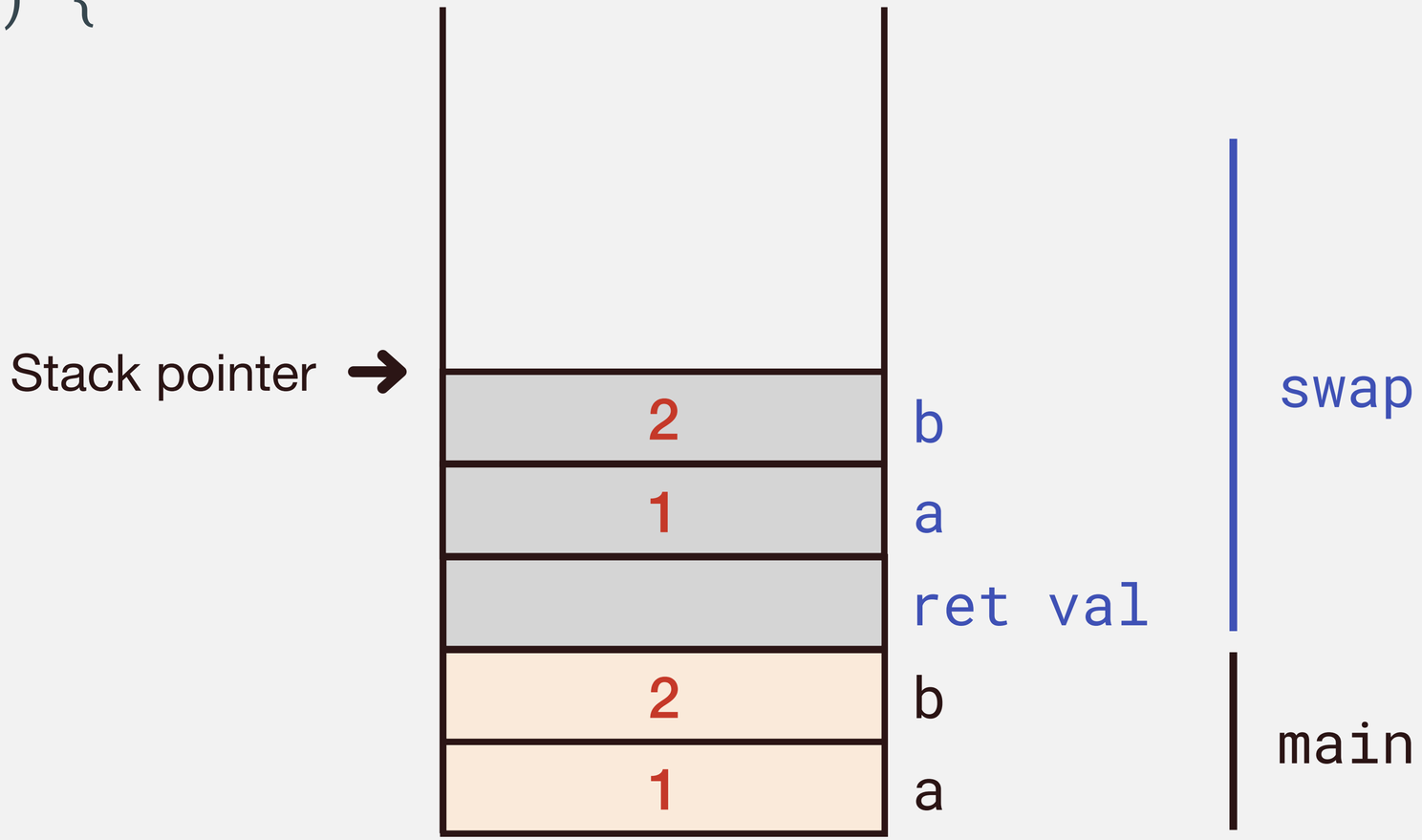
Call Stack

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
➔ swap(a, b);
}
```



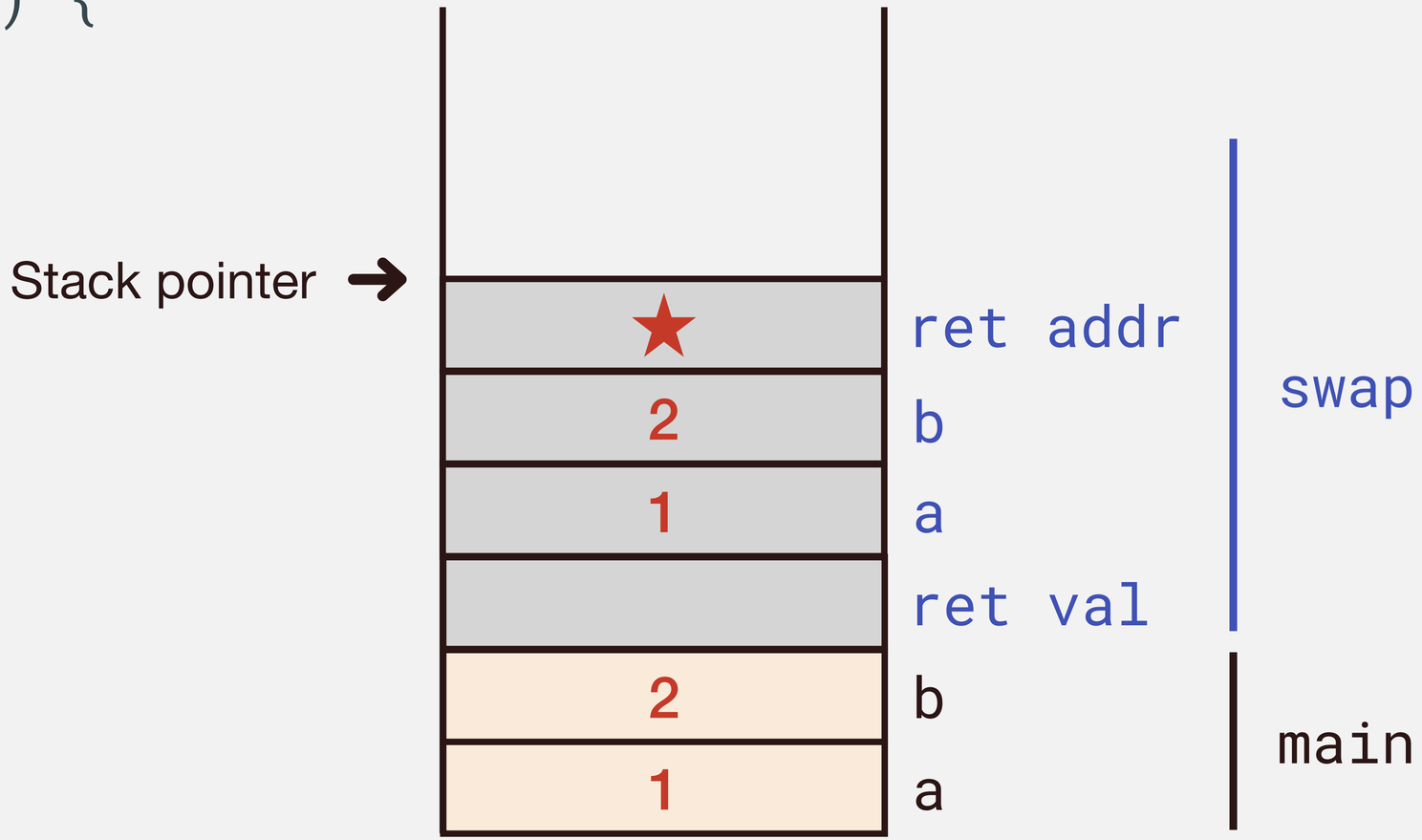
Call Stack

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
➜ swap(a, b); ★
}
```



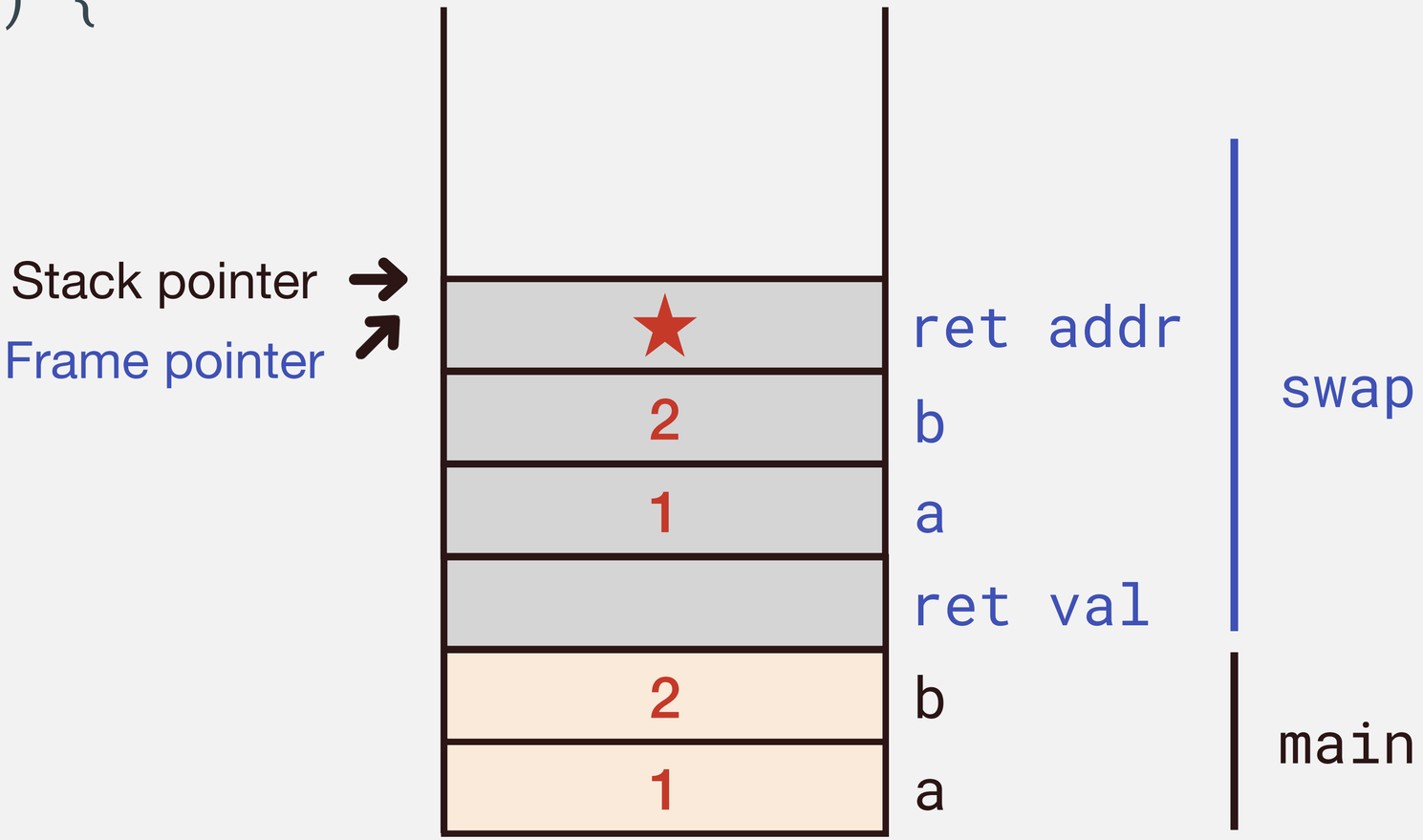
Call Stack

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
➔ swap(a, b); ★
}
```



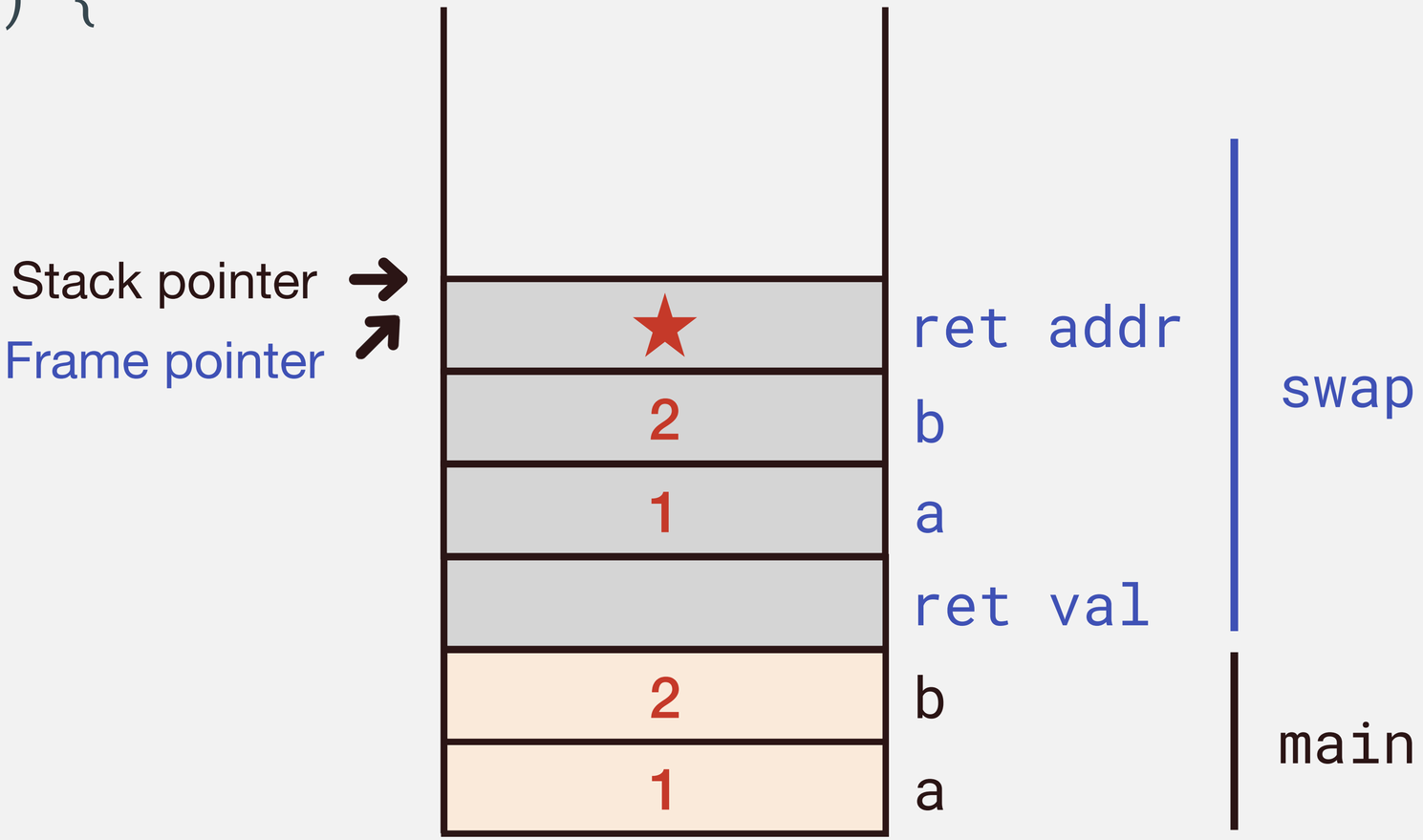
Call Stack

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
➔ swap(a, b); ★
}
```



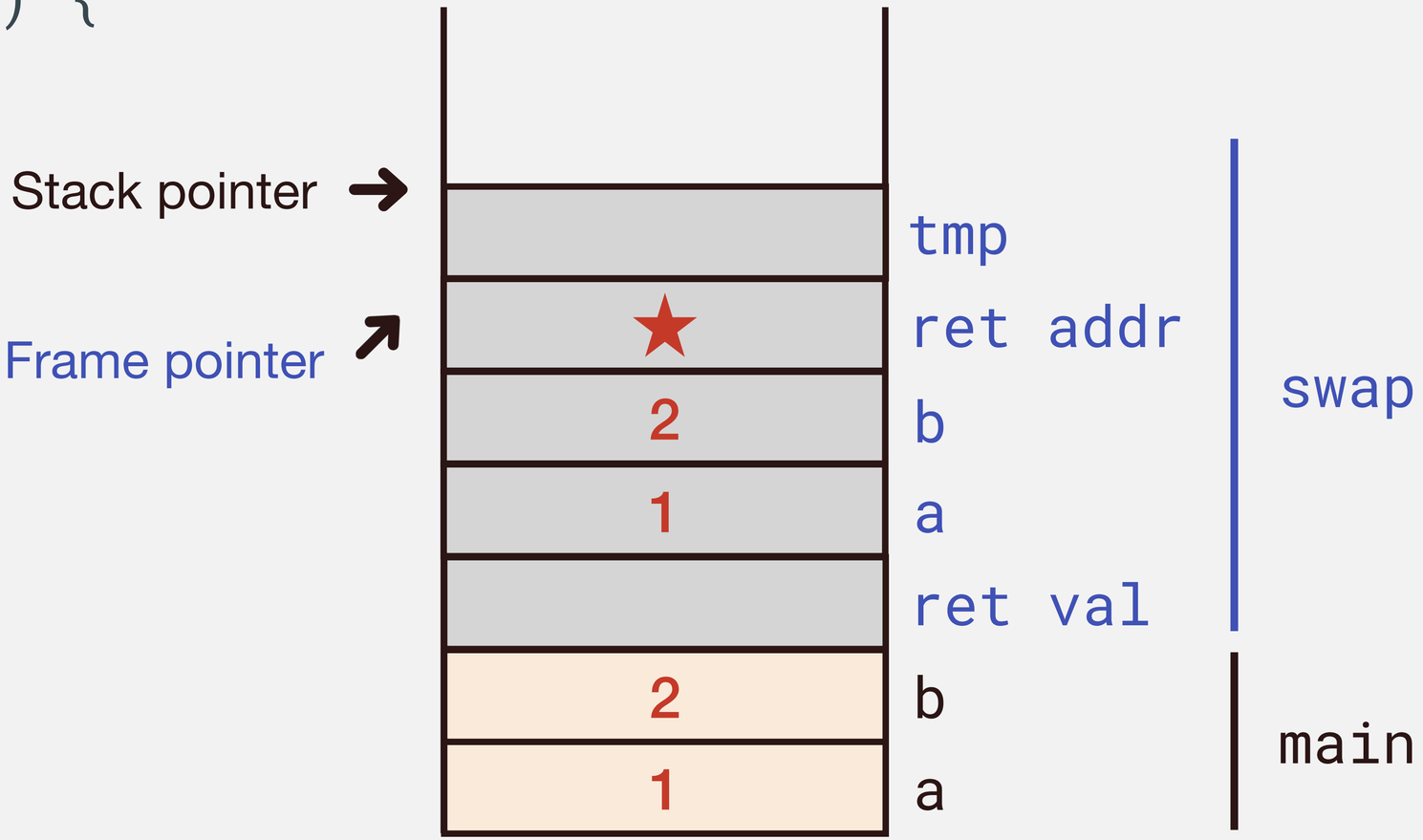
Call Stack

```
void swap(int a, int b) {
➔ int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
  swap(a, b); ★
}
```



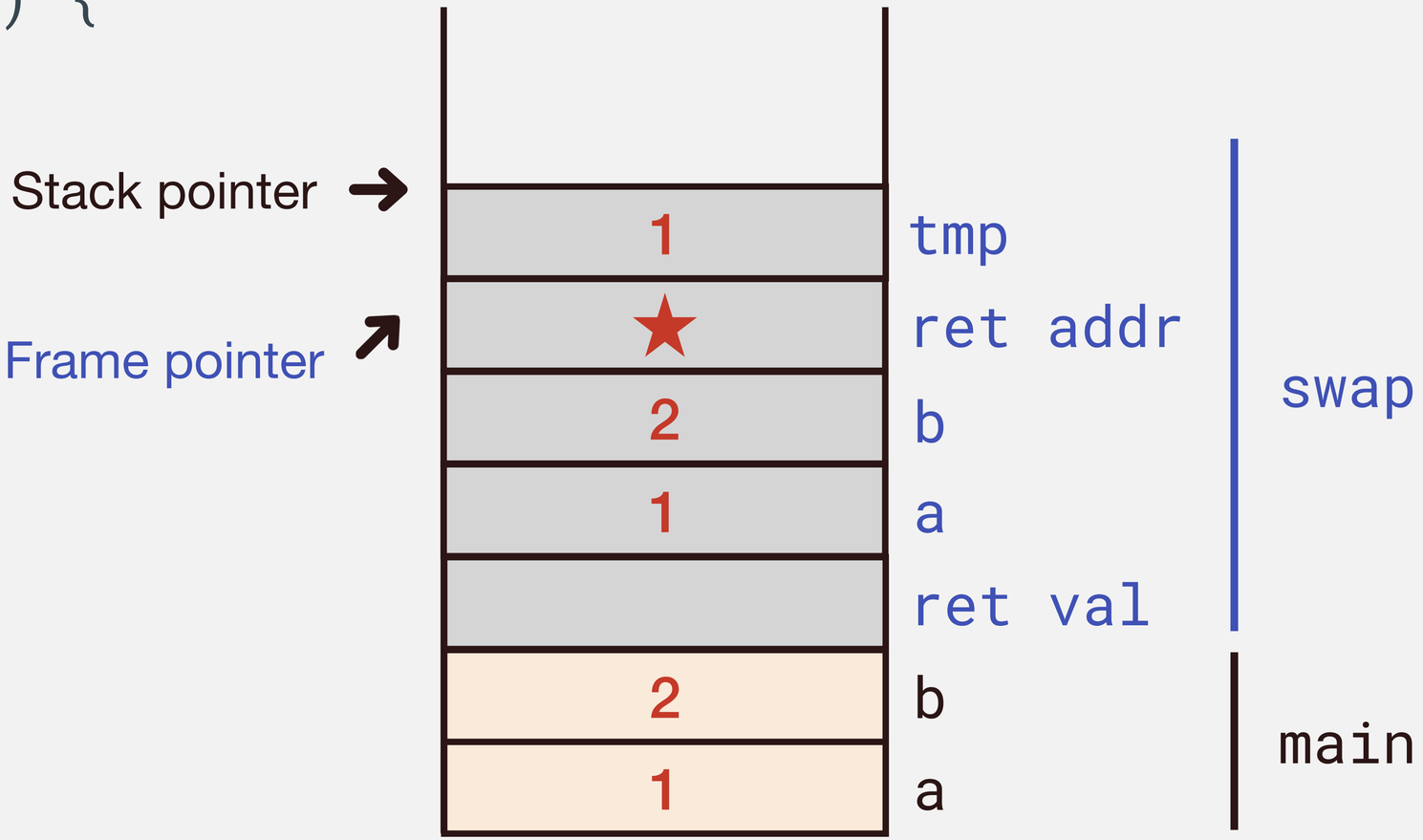
Call Stack

```
void swap(int a, int b) {
➔ int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
  swap(a, b); ★
}
```



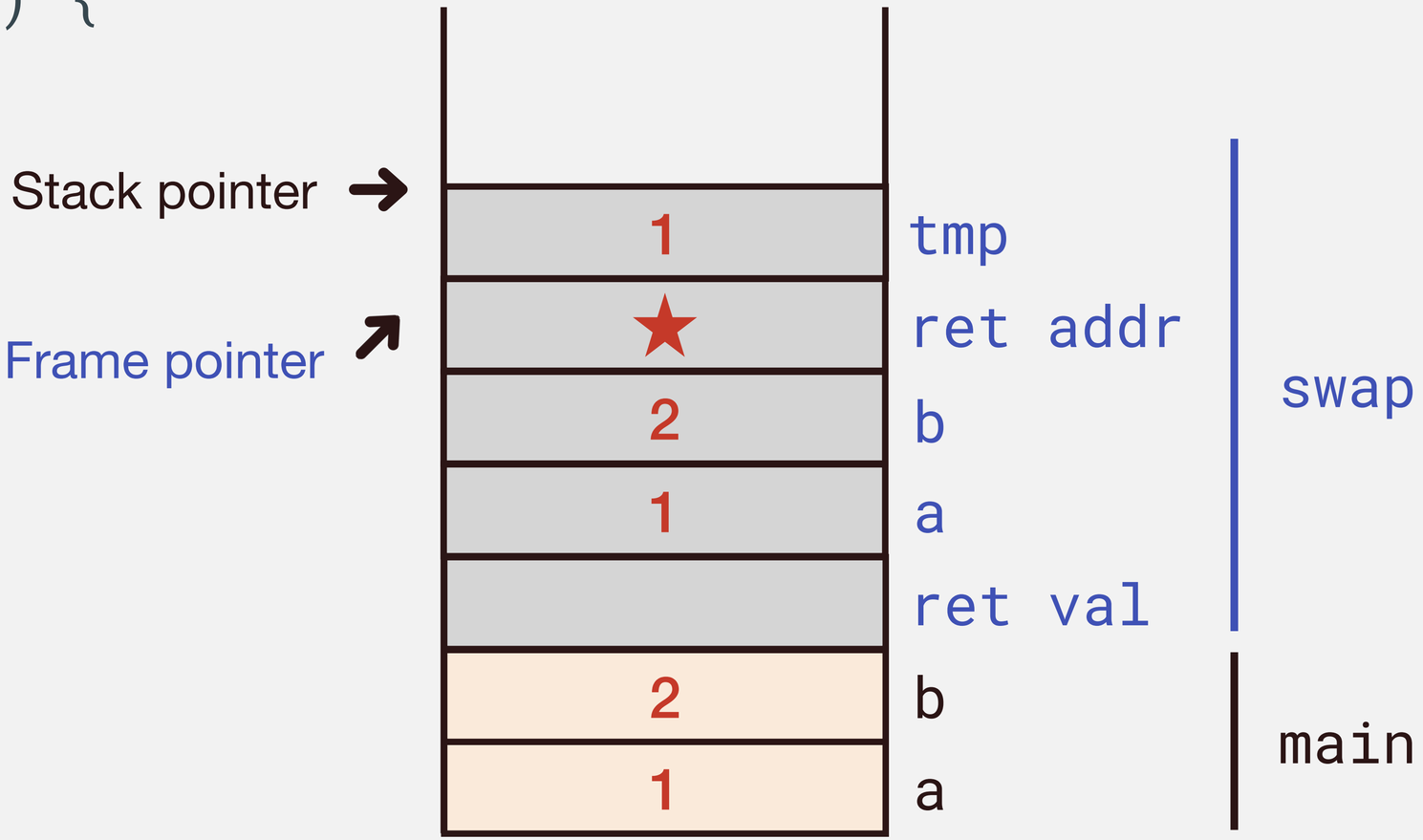
Call Stack

```
void swap(int a, int b) {
➜ int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
  swap(a, b); ★
}
```



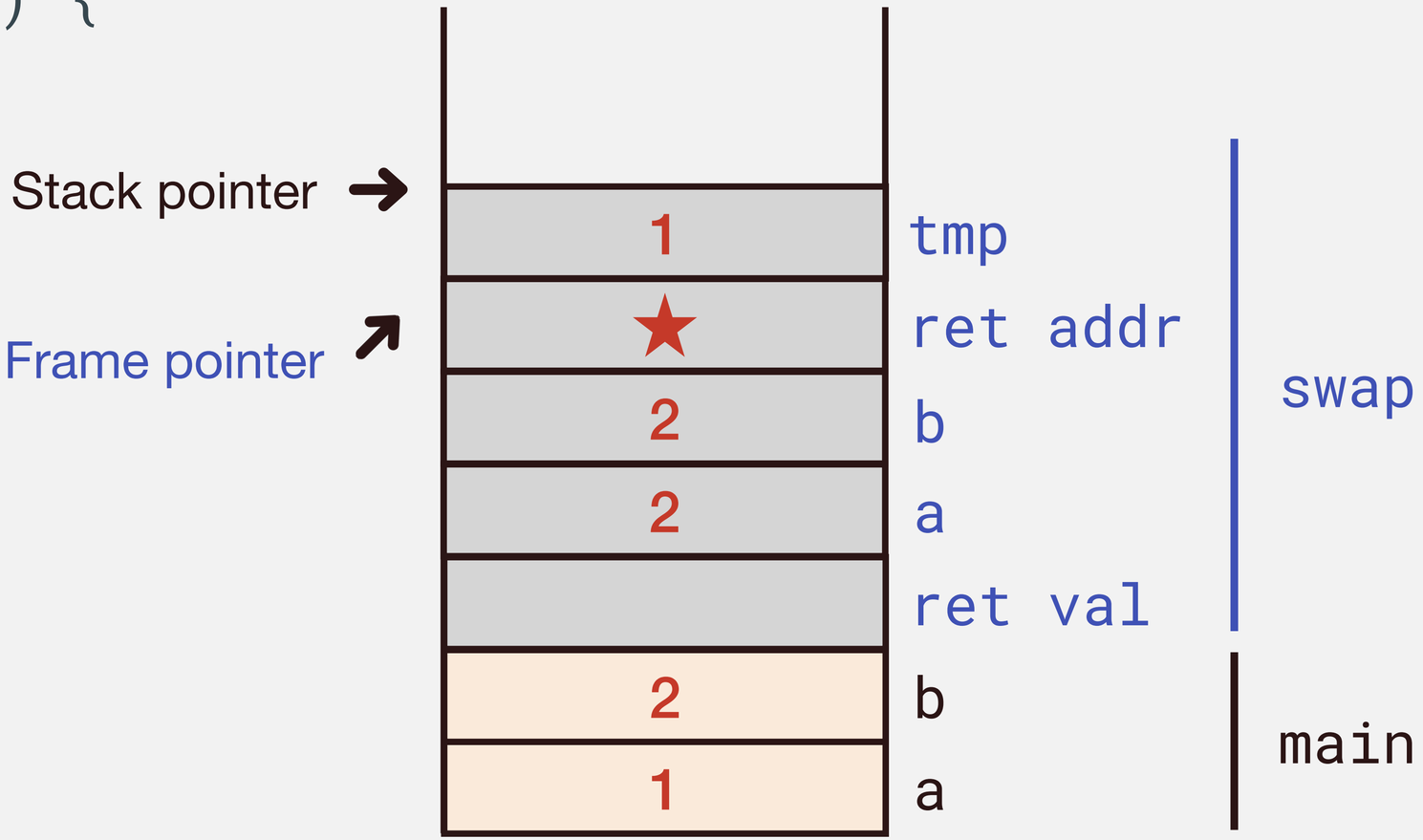
Call Stack

```
void swap(int a, int b) {
  int tmp = a;
→ a = b;
  b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
  swap(a, b); ★
}
```



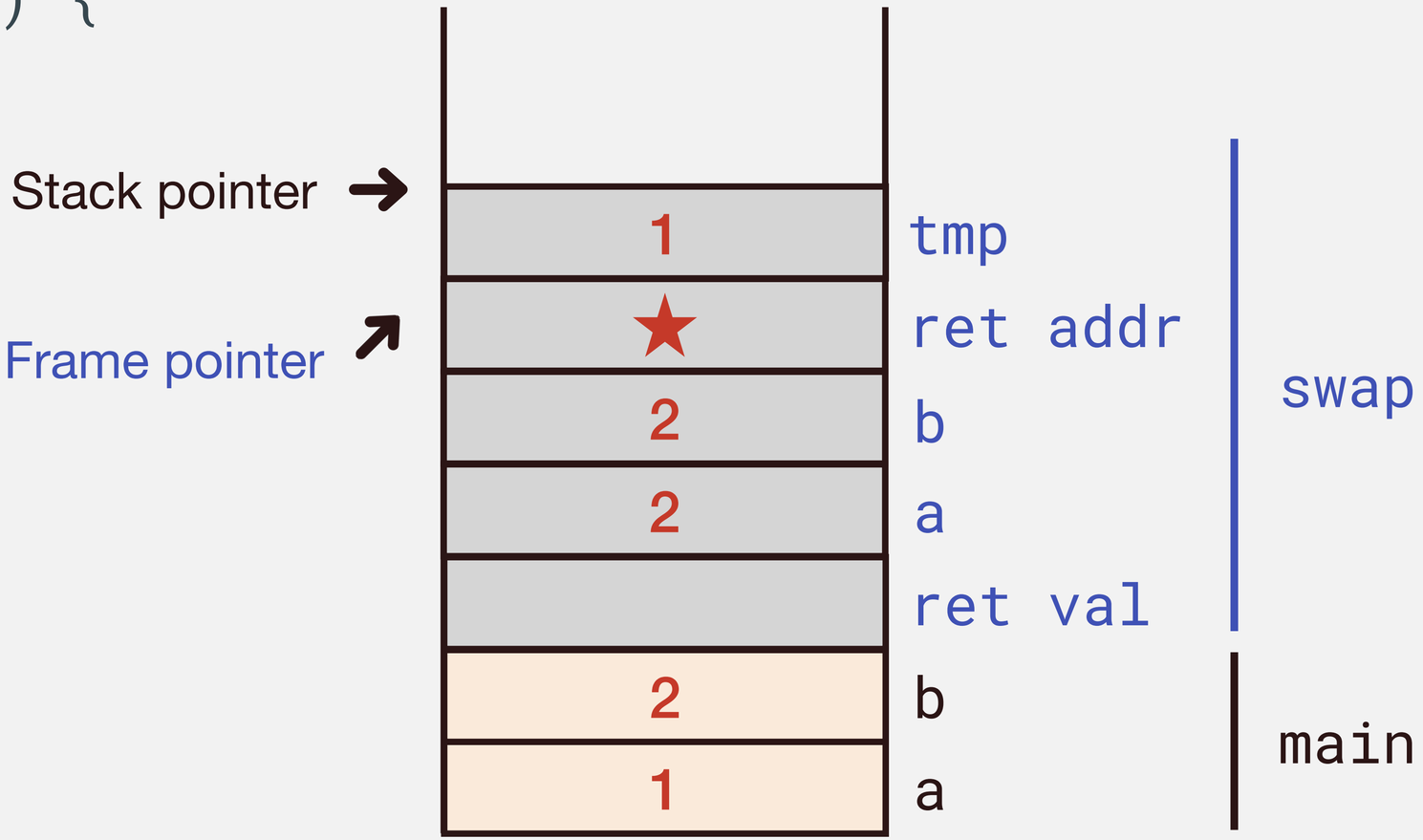
Call Stack

```
void swap(int a, int b) {
  int tmp = a;
➔ a = b;
  b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
  swap(a, b); ★
}
```



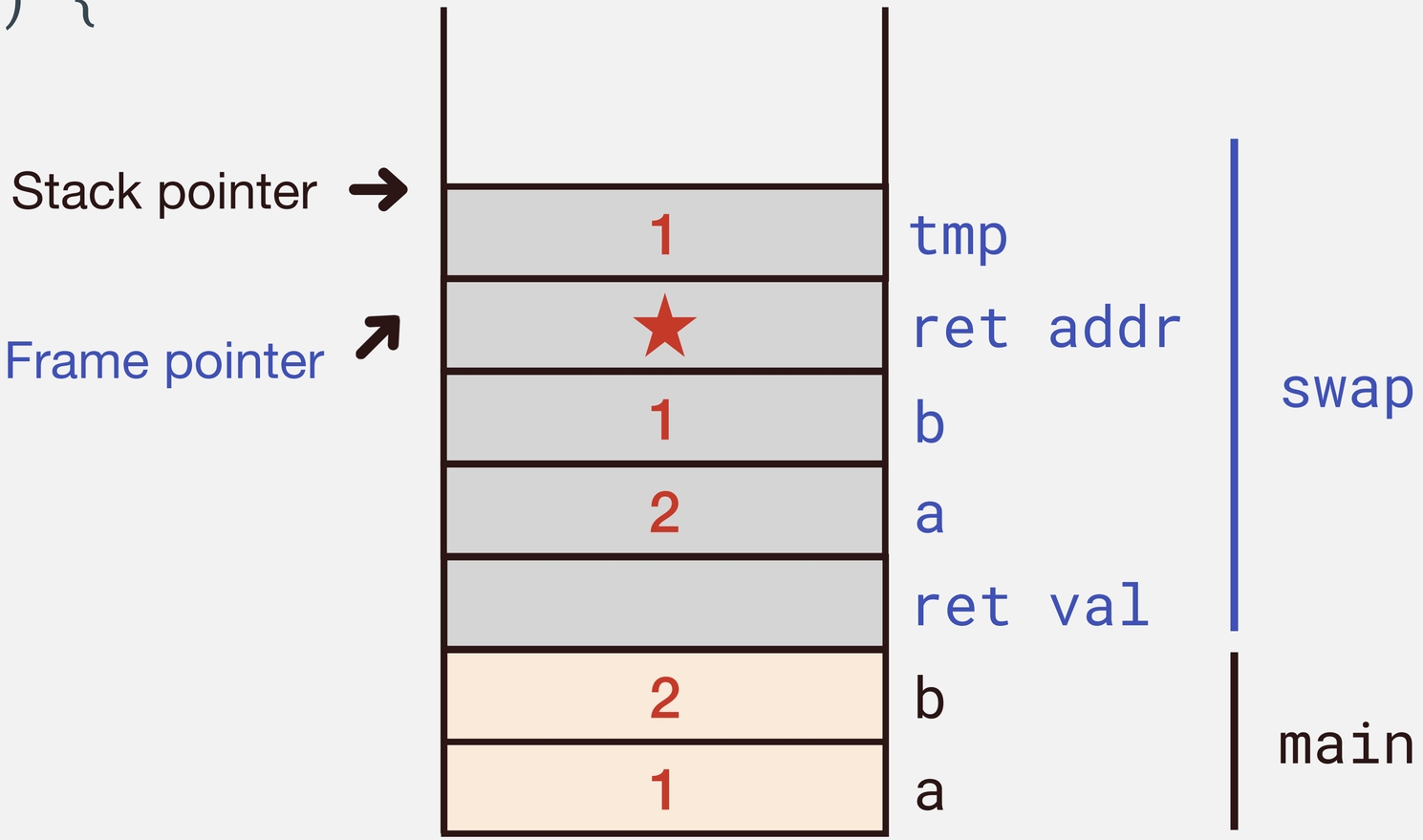
Call Stack

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
→ b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
  swap(a, b); ★
}
```



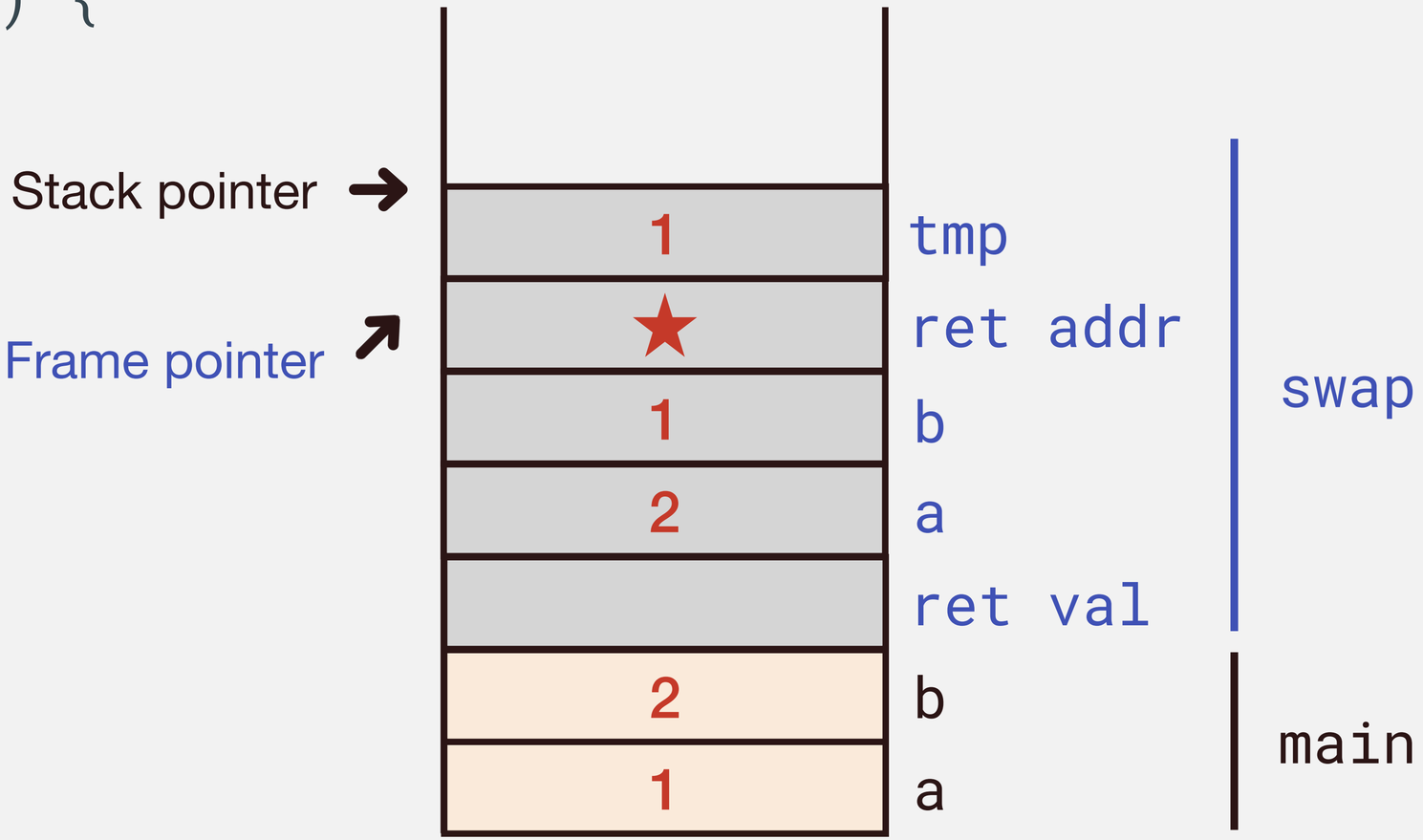
Call Stack

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
→ b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
  swap(a, b); ★
}
```



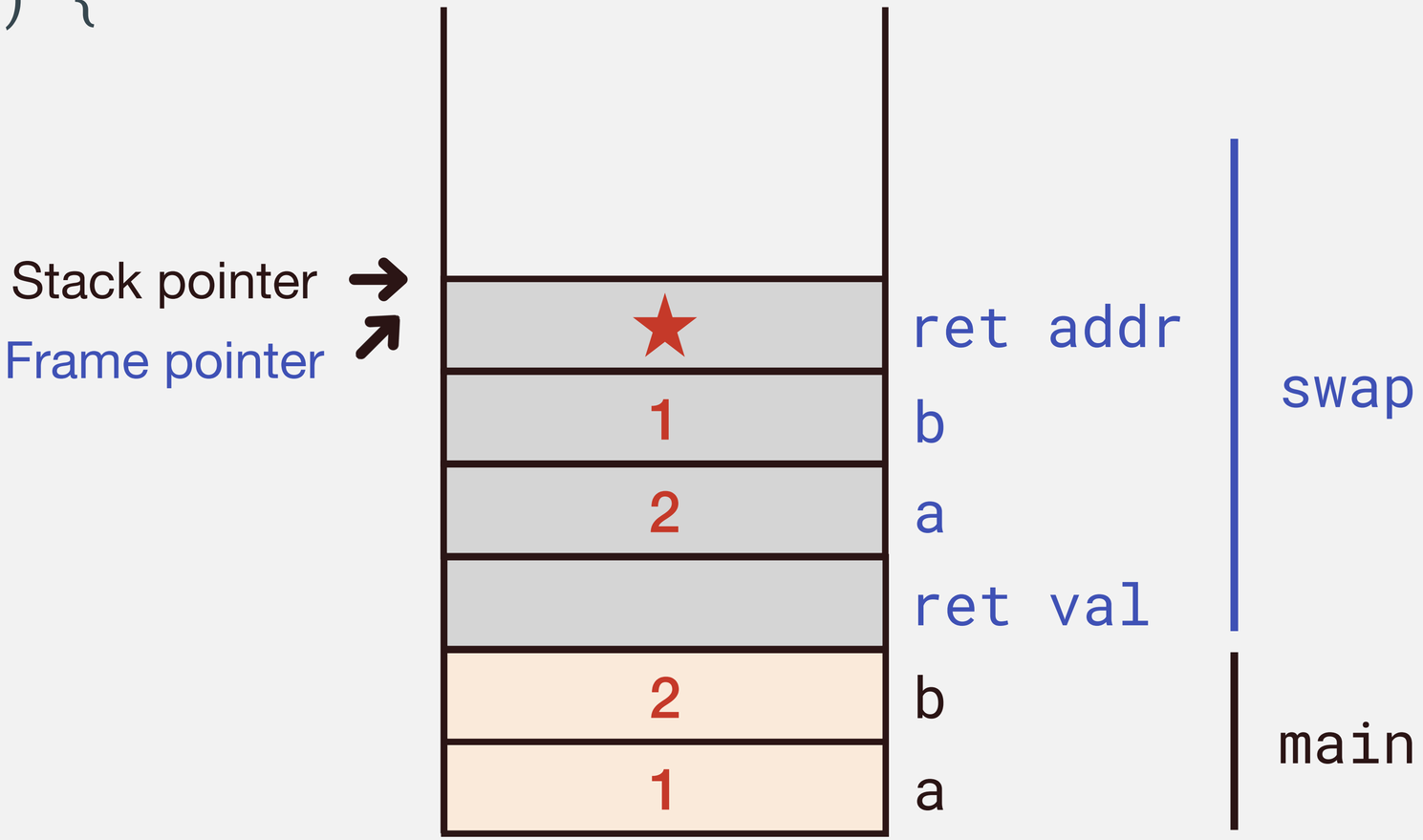
Call Stack

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
→ }

int main() {
  int a = 1;
  int b = 2;
  swap(a, b); ★
}
```



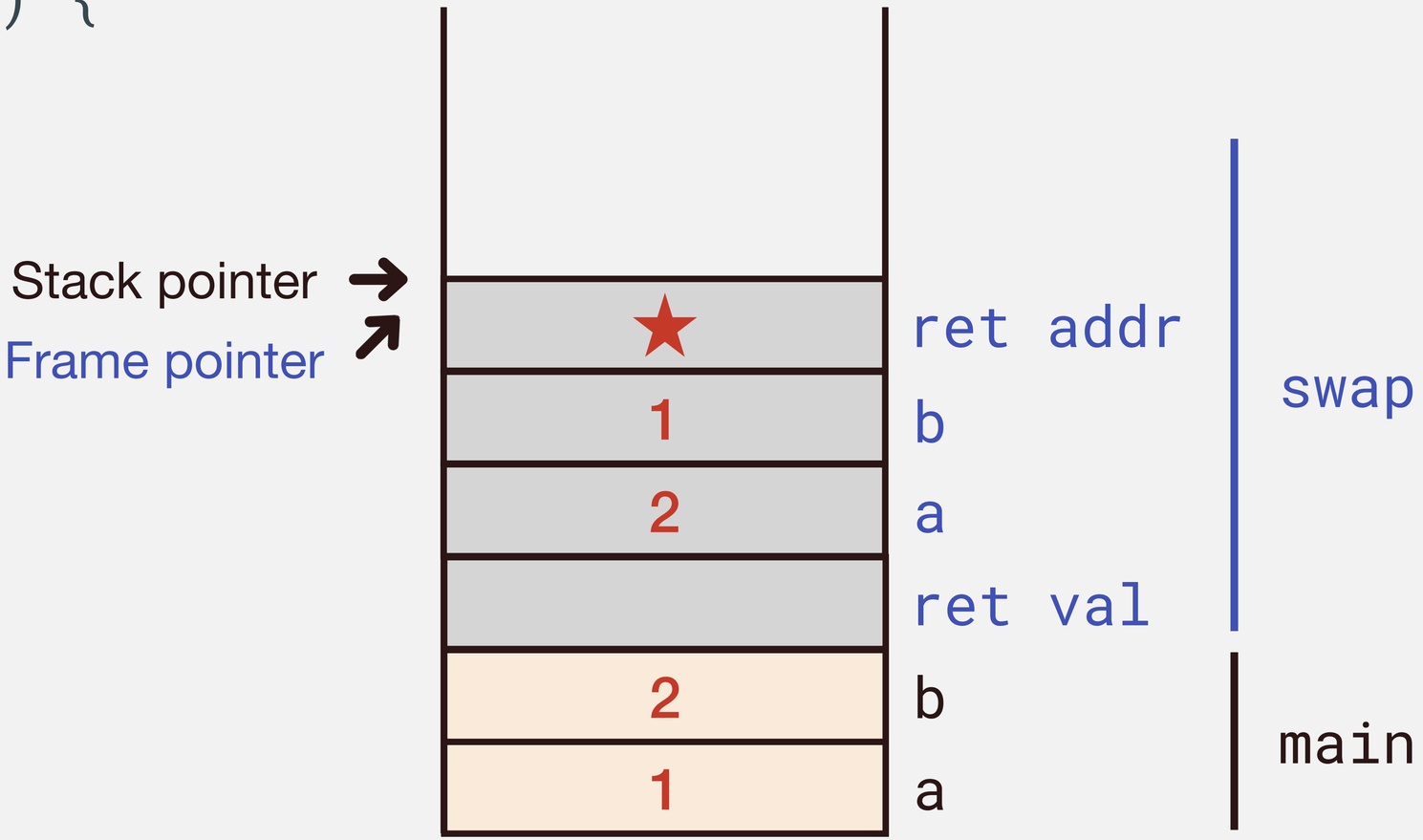
Call Stack

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
→ }

int main() {
  int a = 1;
  int b = 2;
  swap(a, b); ★
}
```



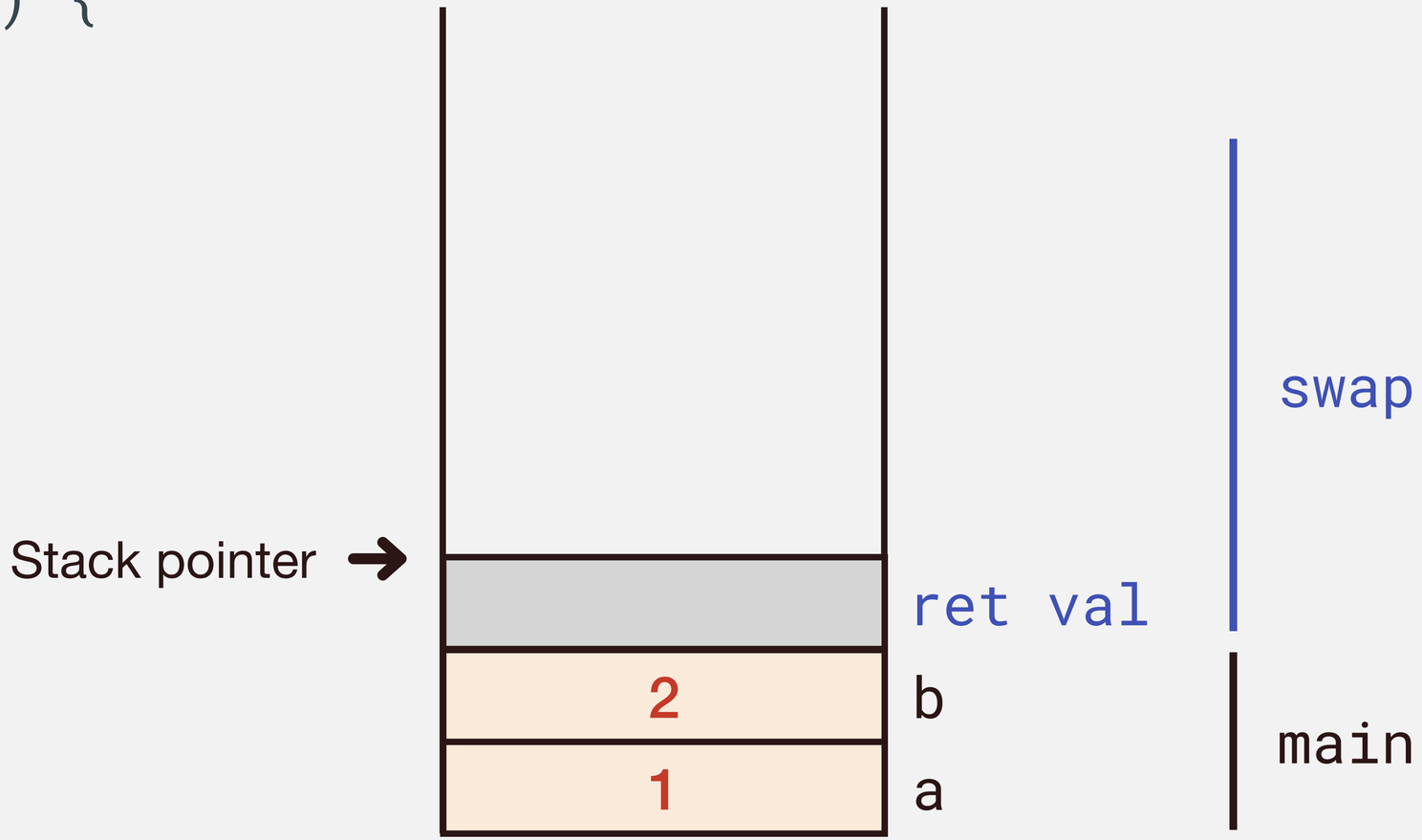
Call Stack

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
➔ swap(a, b); ★
}
```



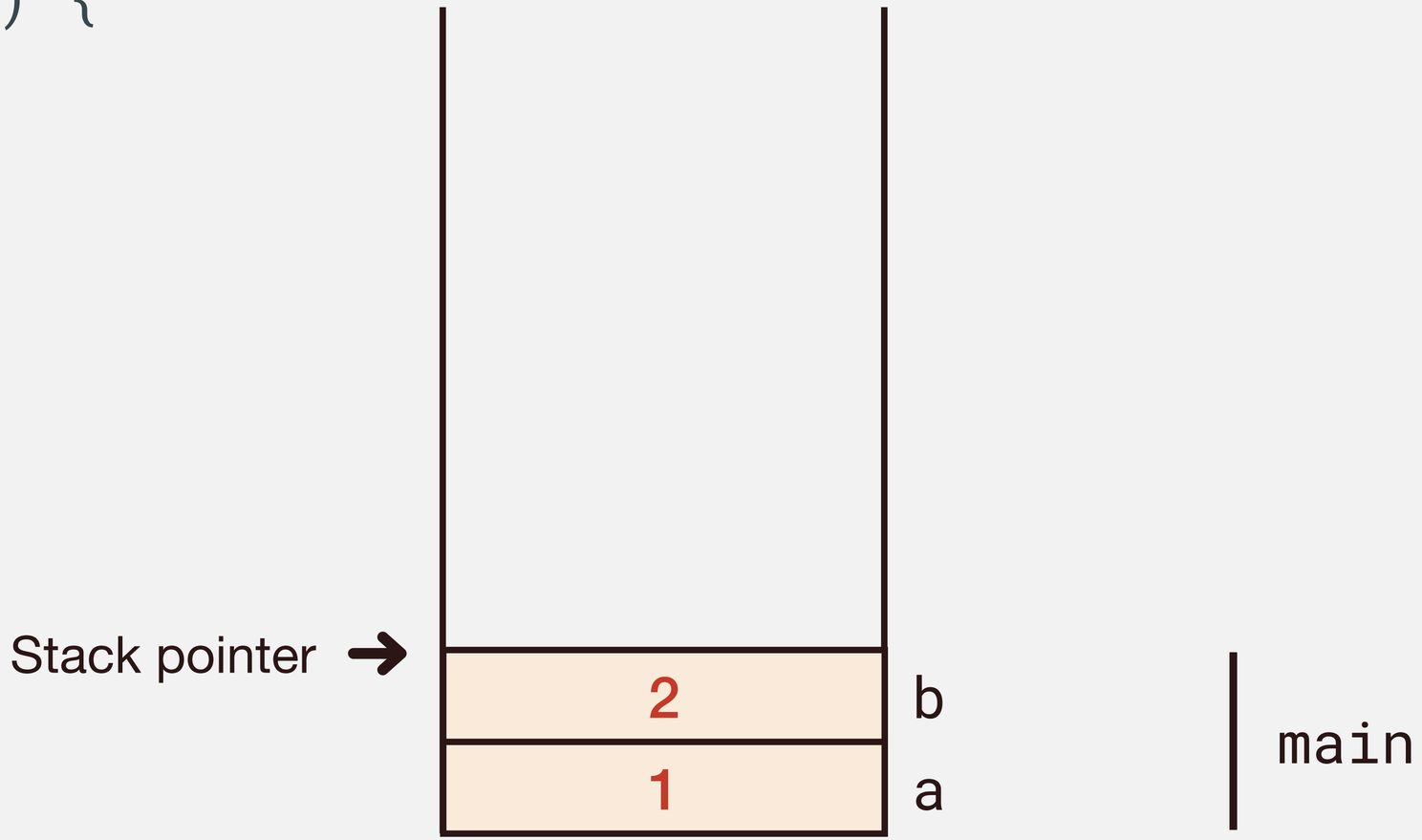
Call Stack

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
➔ swap(a, b); ★
}
```



Call Stack

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
→ swap(a, b); ★
}
```



Phenomena Explained by the Stack

1. Why are function arguments **passed by value**?

Phenomena Explained by the Stack

1. Why are function arguments **passed by value**?
   - Makes a local copy of the passed-in values on stack at function call

Phenomena Explained by the Stack

1. Why are function arguments **passed by value**?
   - Makes a local copy of the passed-in values on stack at function call
2. Why are local variables limited to the **scope** of the function?

Phenomena Explained by the Stack

1. Why are function arguments **passed by value**?
   - Makes a local copy of the passed-in values on stack at function call
2. Why are local variables limited to the **scope** of the function?
   - They no-longer exist on stack after function returns

Variable Scope

```
int squareSum(int n) {
  int s = 0;
  for (int i = 1; i <= n; i++) {
    int i_sqr = i * i;
    s += i_sqr;
  }
  return s;
}

int main() {
  int n = 100;
  squareSum(n);
}
```

Variable Scope

```
int squareSum(int n) {
  int s = 0;
  for (int i = 1; i <= n; i++) {
    int i_sqr = i * i;
    s += i_sqr;
  }
  return s;
}

int main() {
  int n = 100;
  squareSum(n);
}
```

Variable Scope

```
int squareSum(int n) {
  int s = 0;
  for (int i = 1; i <= n; i++) {
    int i_sqr = i * i;
    s += i_sqr;
  }
  return s;
}
```

```
int main() {
  int n = 100;
  squareSum(n);
}
```

Variable Scope

```
int squareSum(int n) {
  int s = 0;
  for (int i = 1; i <= n; i++) {
    int i_sqr = i * i;
    s += i_sqr;
  }
  return s;
}

int main() {
  int n = 100;
  squareSum(n);
}
```

Variable Scope

```
int squareSum(int n) {
  int s = 0;
  for (int i = 1; i <= n; i++) {
    int i_sqr = i * i;
    s += i_sqr;
  }
  return s;
}
```

```
int main() {
  int n = 100;
  squareSum(n);
}
```

Variable Scope

```
int squareSum(int n) {
  int s = 0;
  for (int i = 1; i <= n; i++) {
    int i_sqr = i * i;
    s += i_sqr;
  }
  return s;
}

int main() {
  int n = 100;
  squareSum(n);
}
```

Variable Scope

```
int squareSum(int n) {
  int s = 0;
  for (int i = 1; i <= n; i++) {
    int i_sqr = i * i;
    s += i_sqr;
  }
  return s;
}

int main() {
  int n = 100;
  squareSum(n);
}
```

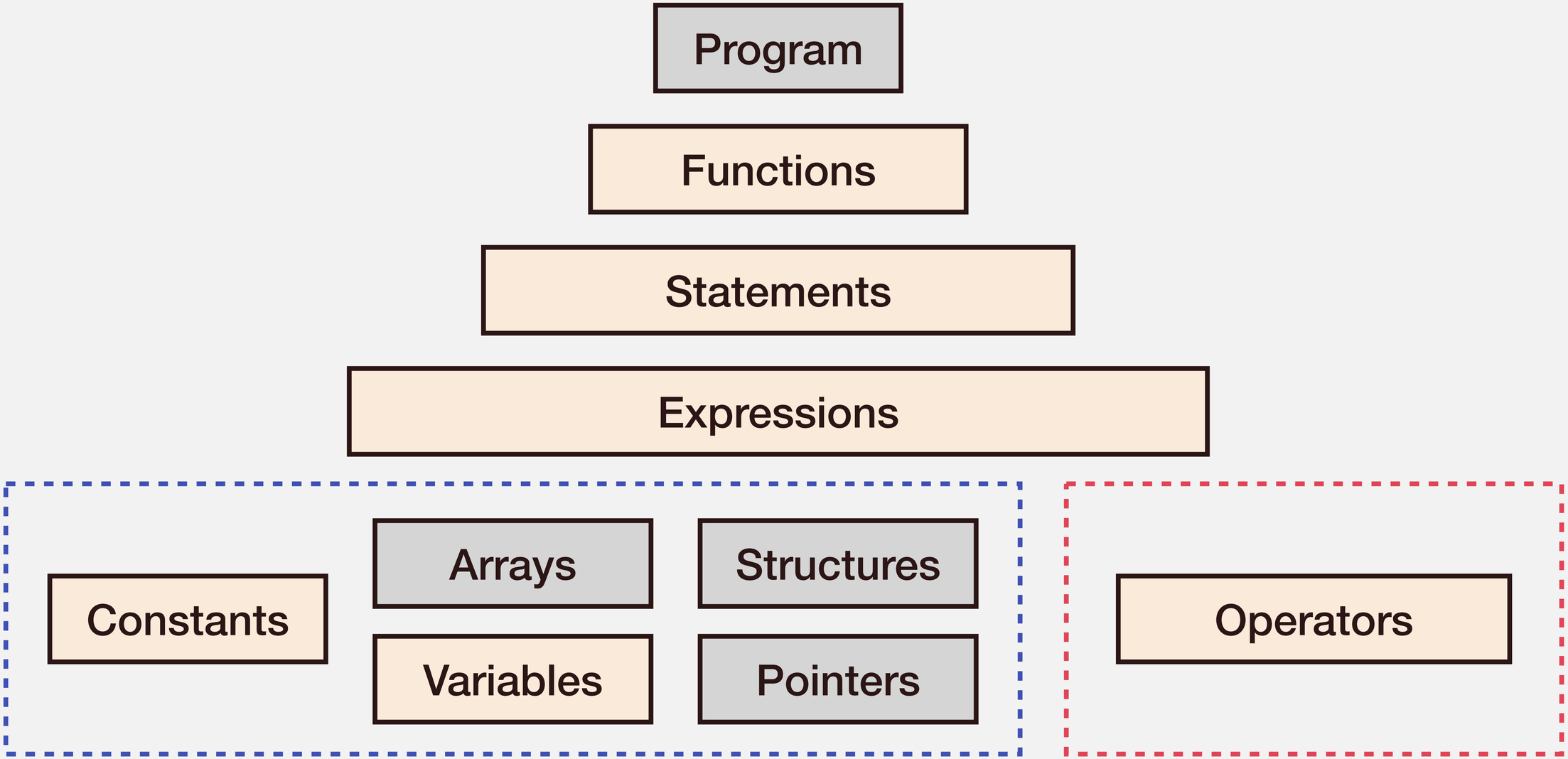
Phenomena Explained by the Stack

1. Why are function arguments **passed by value**?
   - Makes a local copy of the passed-in values on stack at function call
2. Why do local variables limited to the **scope** of the function?
   - They no-longer exist on stack after function returns
3. Why don’t local variables need garbage collection?

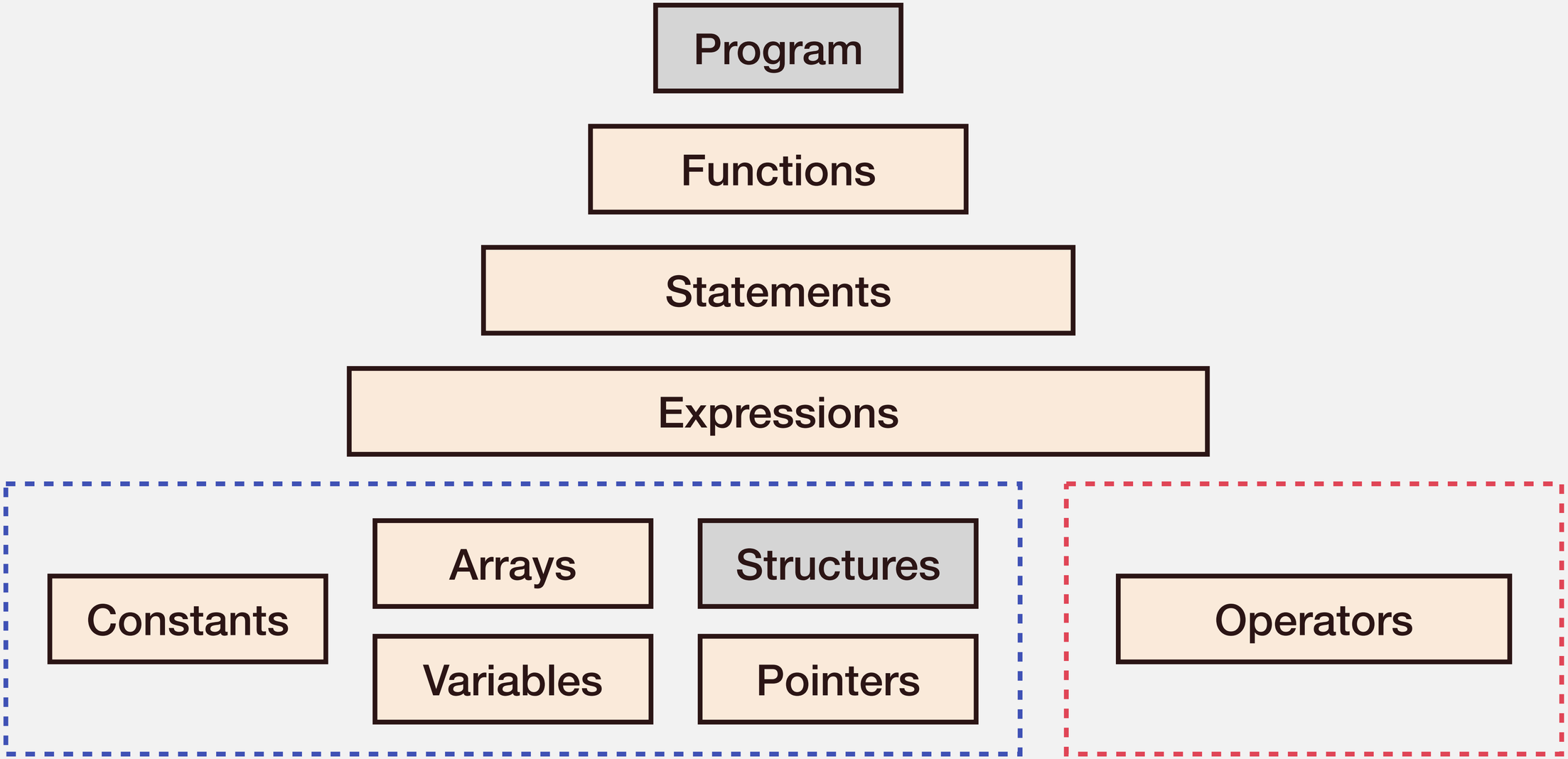
Phenomena Explained by the Stack

1. Why are function arguments **passed by value**?
   - Makes a local copy of the passed-in values on stack at function call
2. Why do local variables limited to the **scope** of the function?
   - They no-longer exist on stack after function returns
3. Why don’t local variables need garbage collection?
   - Automatically done by the stack

Road Map

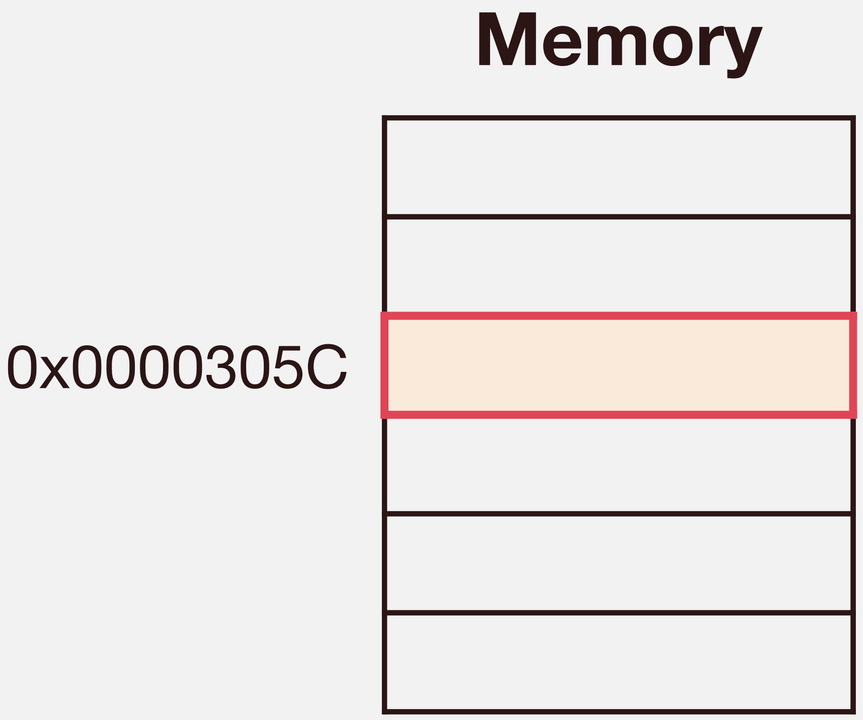

Program
Functions
Statements
Expressions
Constants
Arrays
Variables
Structures
Pointers
Operators

Road Map


Program
Functions
Statements
Expressions
Constants
Arrays
Variables
Structures
Pointers
Operators

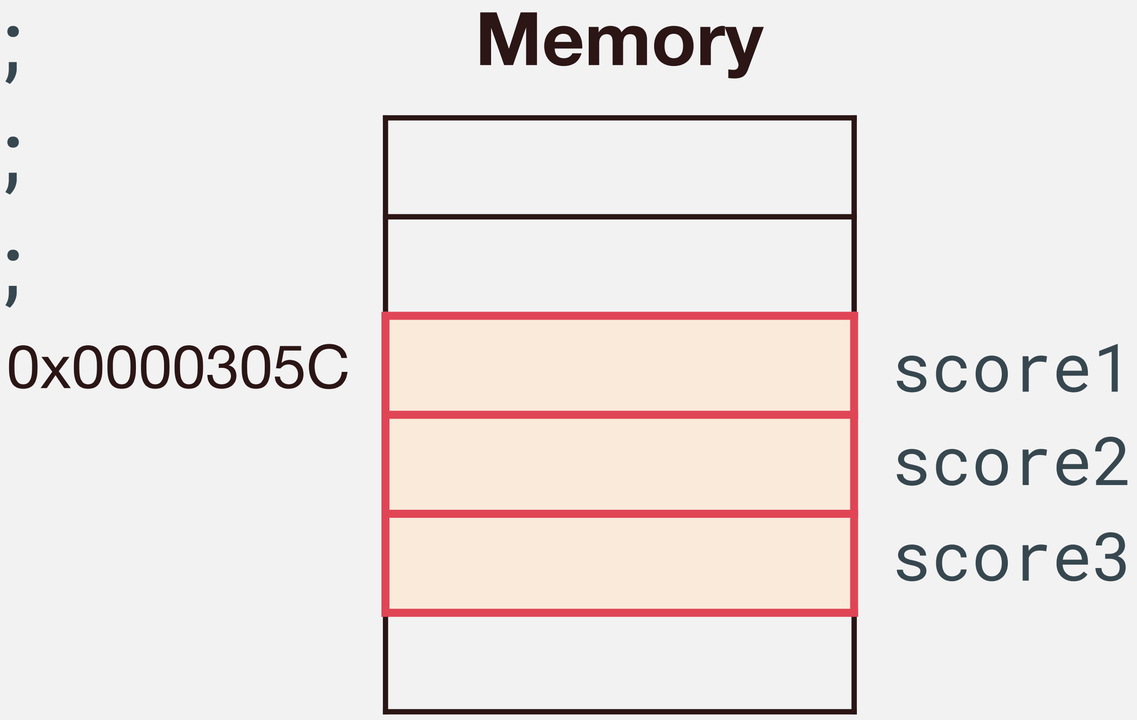
Array Declaration

```
int score;
```



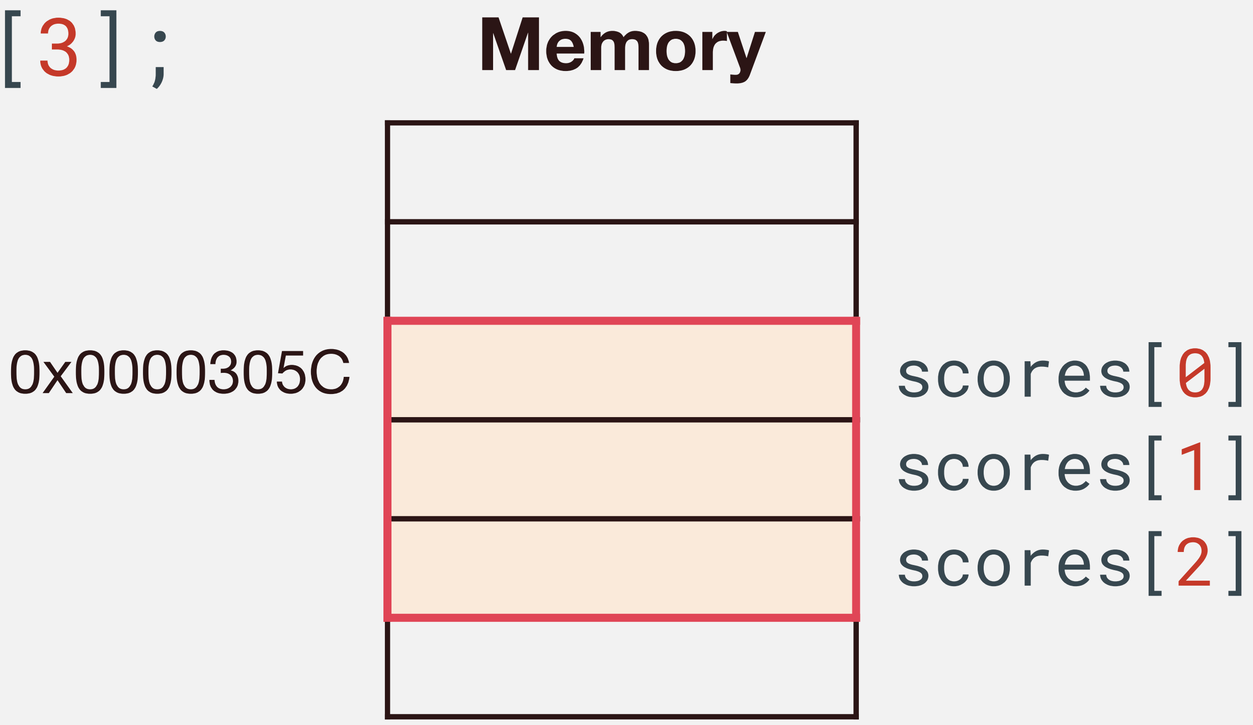
Array Declaration

```
int score1;
int score2;
int score3;
```



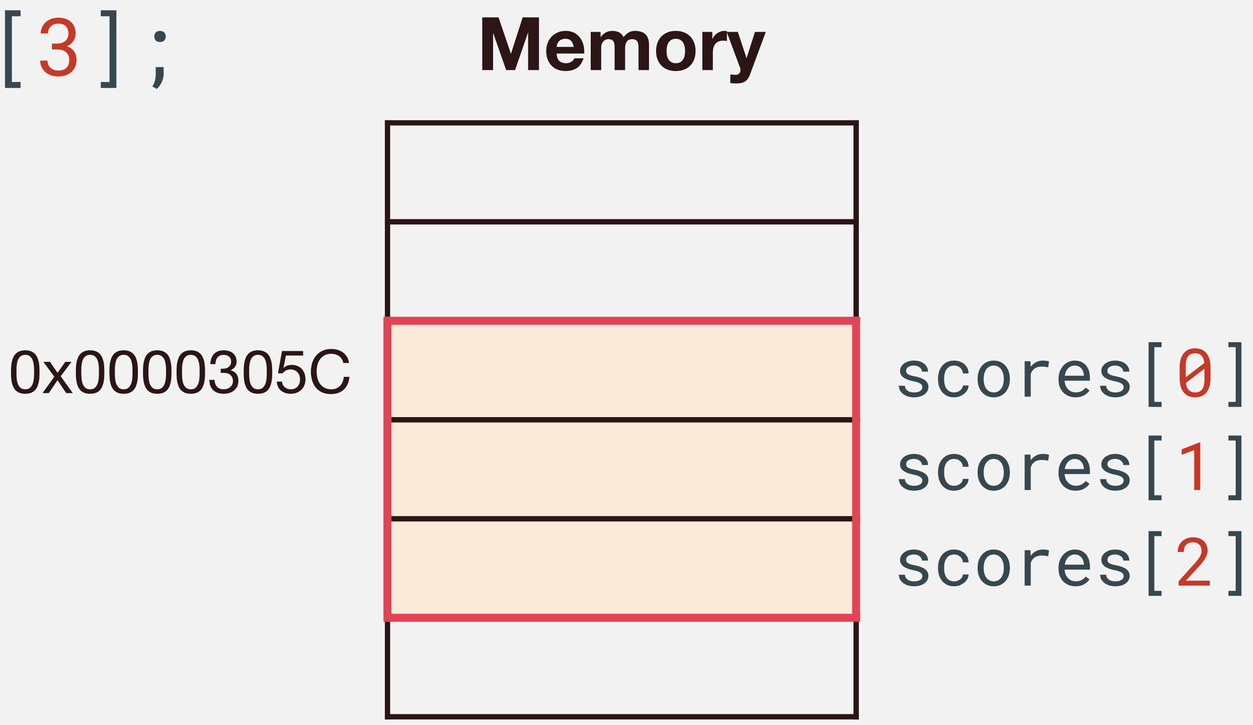
Array Declaration

```
int scores[3];
```



Array Declaration

```
int scores[3];
```



Same Type

Back-to-back

Array Initialization

```
int array[5] = {1, 2, 3, 4, 5}
```

Array Initialization

```
int array[5] = {1, 2, 3, 4, 5}
```

must be **constant**

Array Initialization

```
int array[5] = {1, 2, 3, 4, 5}
```

must be **constant**

```
int n = 5;
int array[n];
```

Array Initialization

```
int array[5] = {1, 2, 3, 4, 5}
```

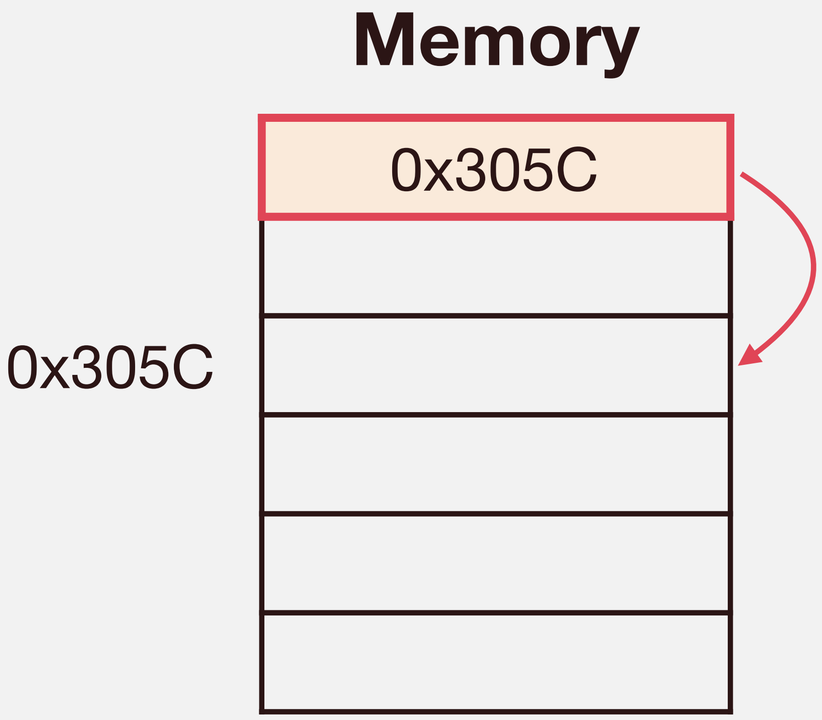
must be **constant**

```
int n = 5;
int array[n];
int i;
for (i = 0; i < n; i++) {
  array[i] = i + 1;
}
```

Pointer Declaration

➜ **Pointer**: a variable that holds the **address** of a memory location

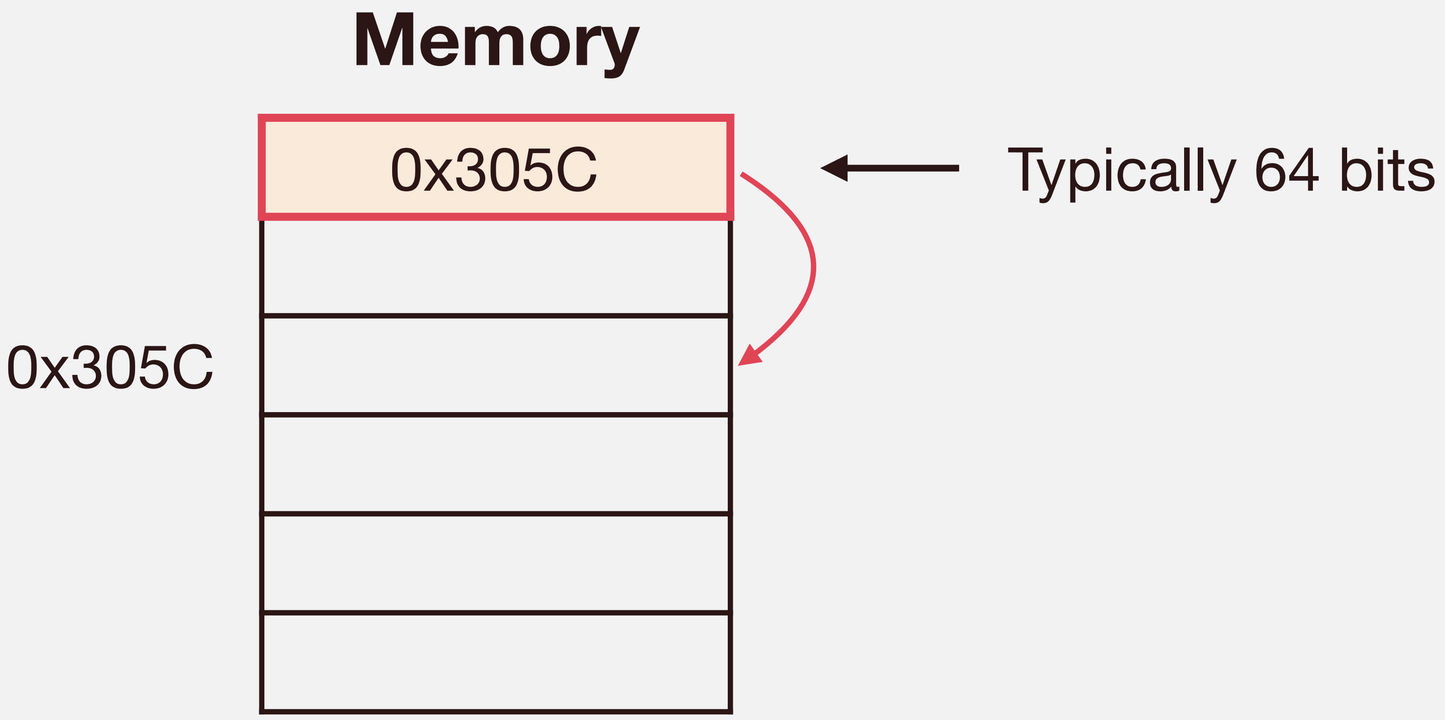
```
int *num_primes;
```



Pointer Declaration

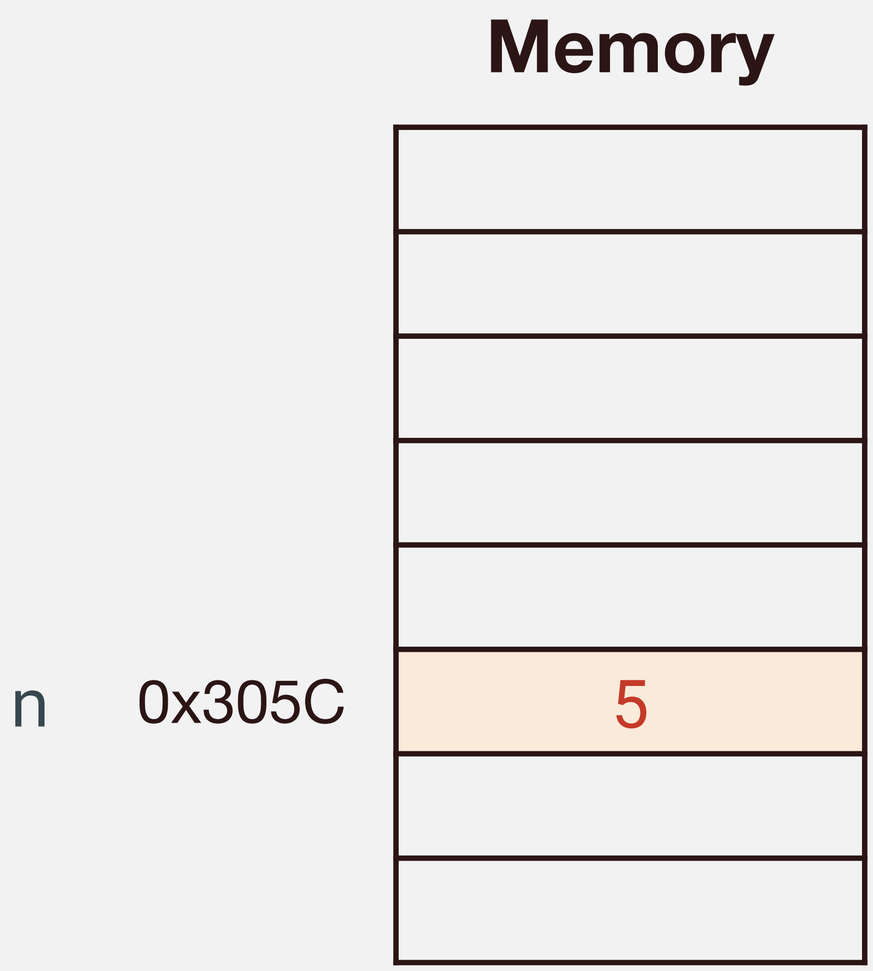
➜ **Pointer**: a variable that holds the **address** of a memory location

```
int *num_primes;
```



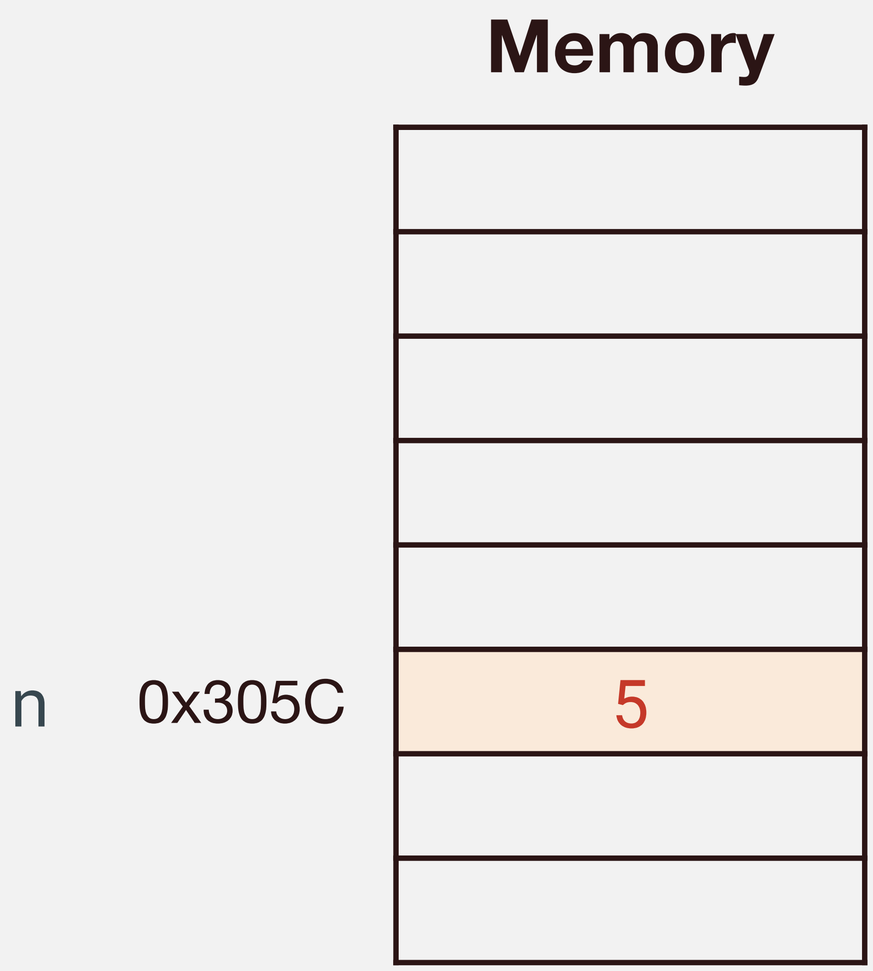
Address <—> Content

```
int main() {
  int n = 5;

}
```



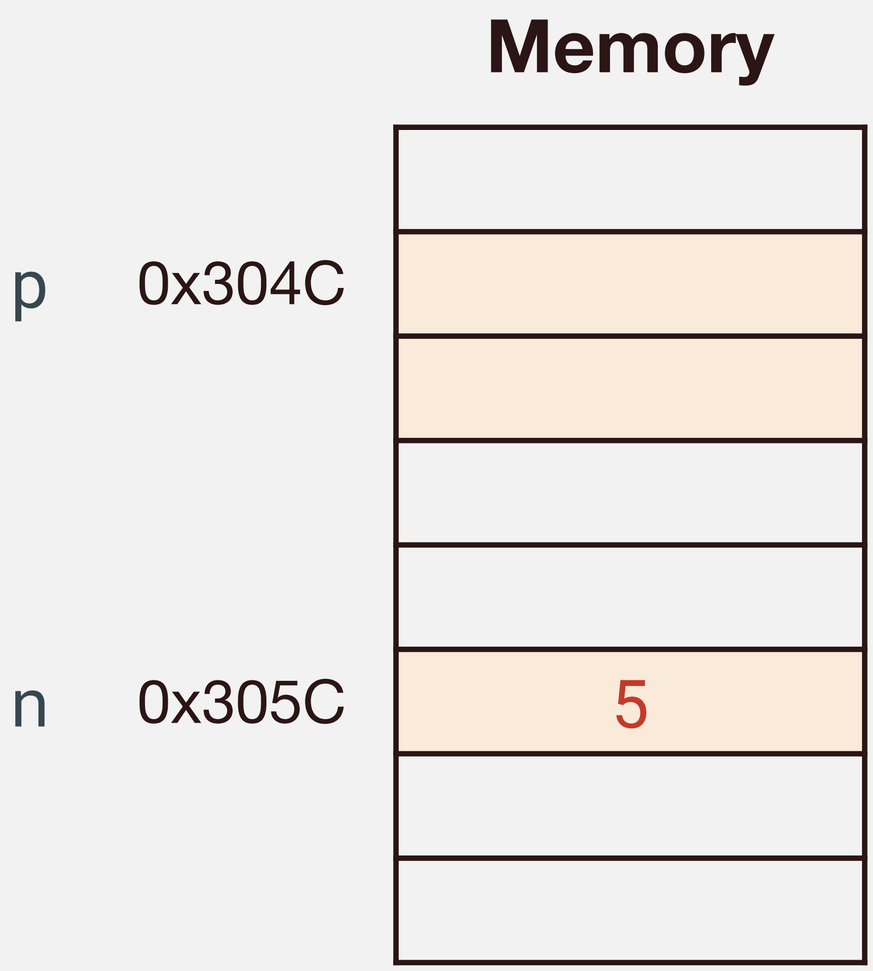
Address <—> Content

```
int main() {
  int n = 5;
  int *p;

}
```



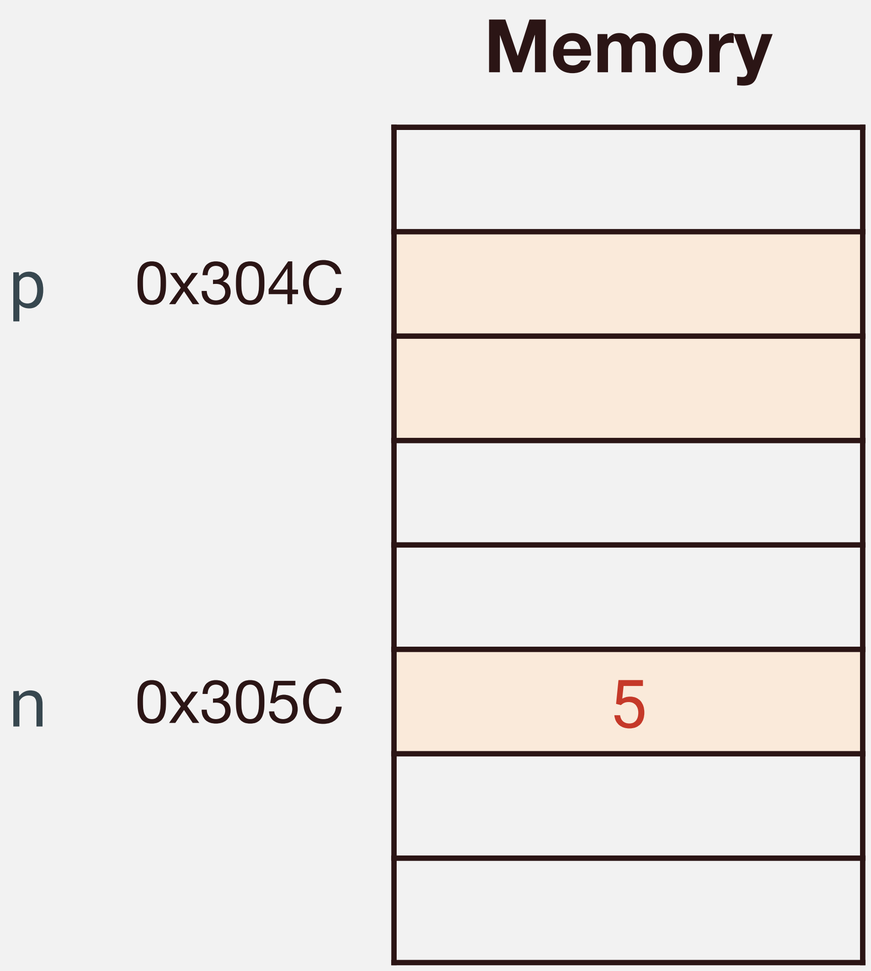
Address <—> Content

```
int main() {
  int n = 5;
  int *p;

}
```



Address <—> Content

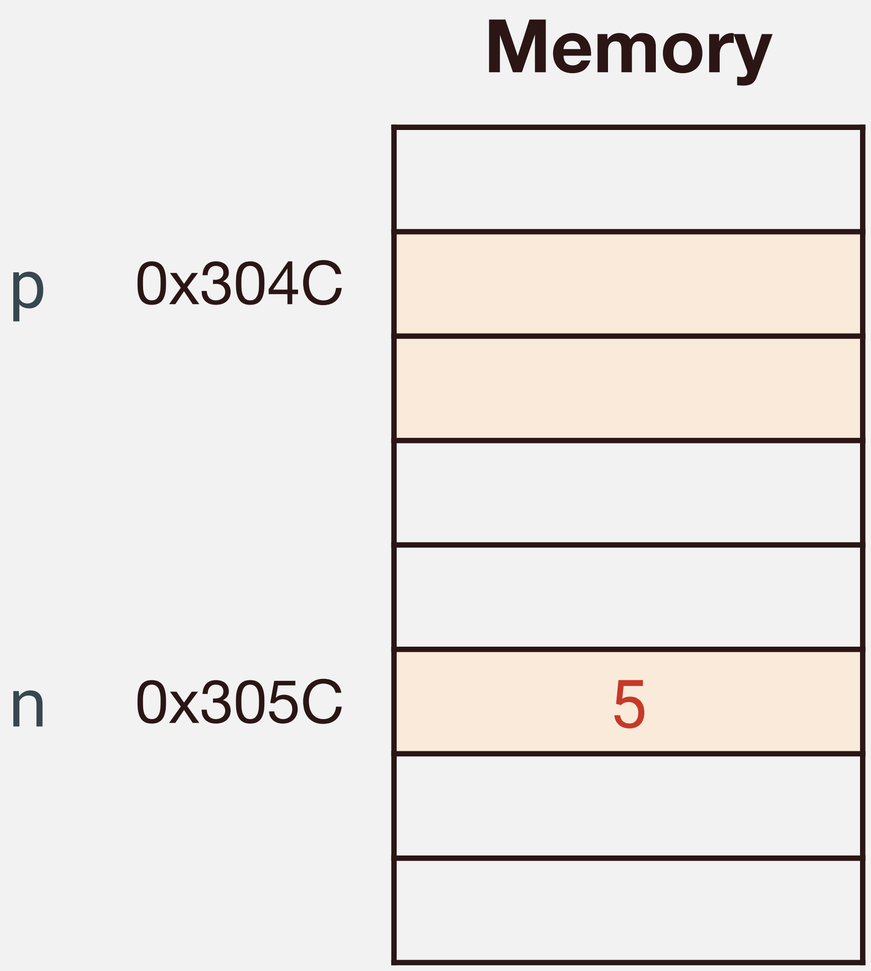
```
int main() {
  int n = 5;
  int *p;
  p = &n;



}
```



Address <—> Content

```
int main() {
  int n = 5;
  int *p;
  p = &n;
}
```

← Take the address of



Address <—> Content

```
int main() {
  int n = 5;
  int *p;
  p = &n;
}
```

← Take the address of

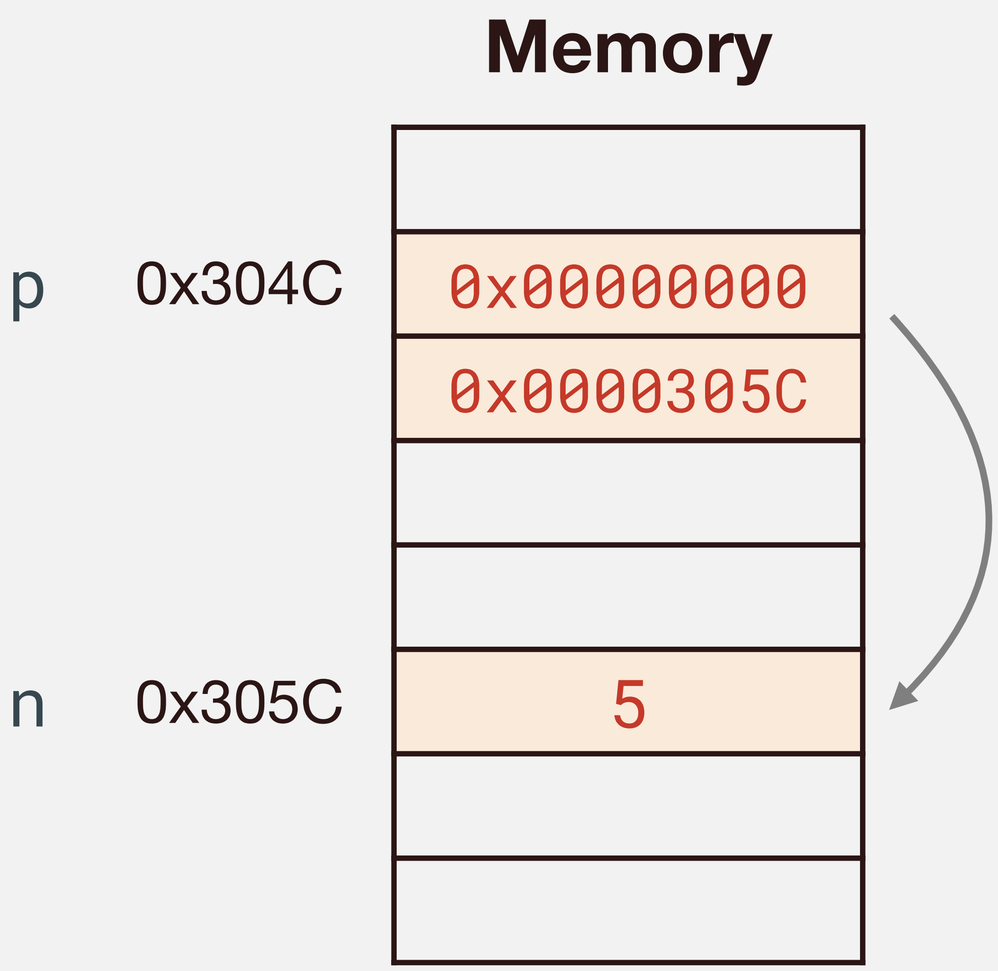
Memory

p	0x304C	0x00000000
		0x0000305C
n	0x305C	5

Address <—> Content

```
int main() {
  int n = 5;
  int *p;
  p = &n;
}
```

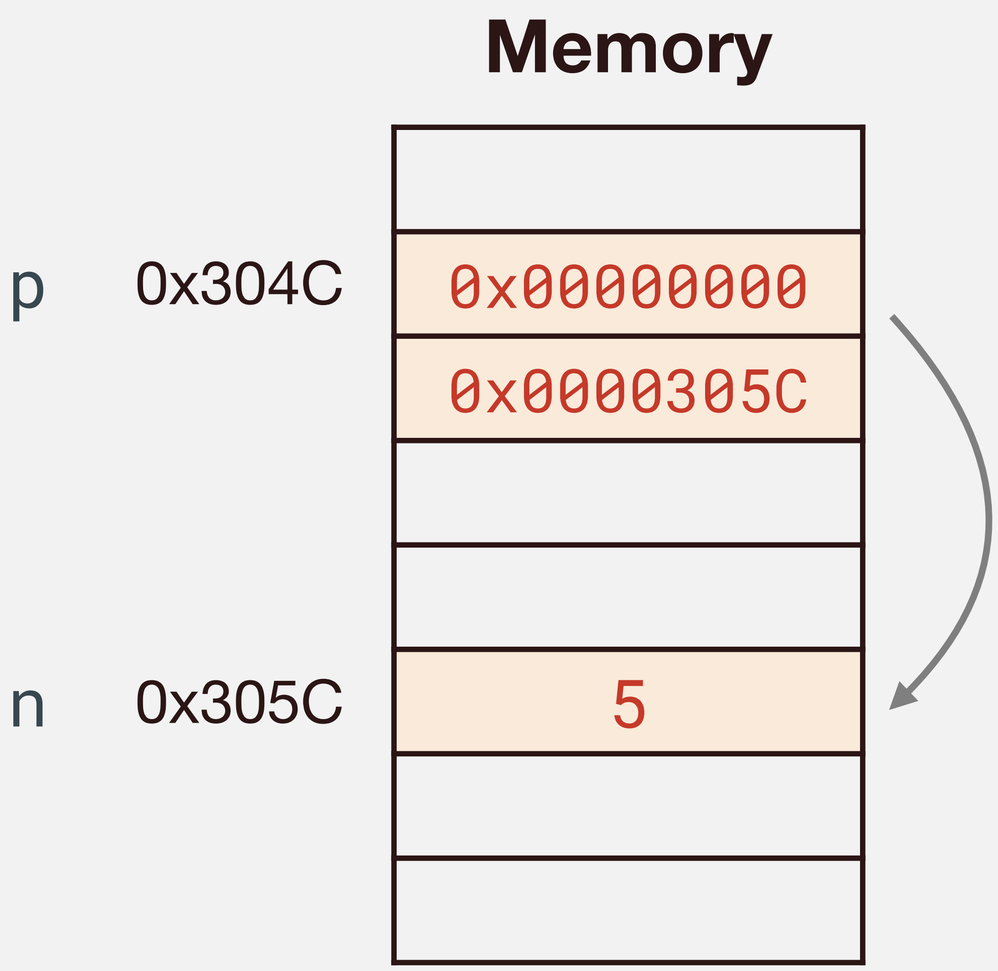
← Take the address of



Address <—> Content

```
int main() {
  int n = 5;
  int *p;
  p = &n;
  int t = *p;

}
```

← Take the address of …

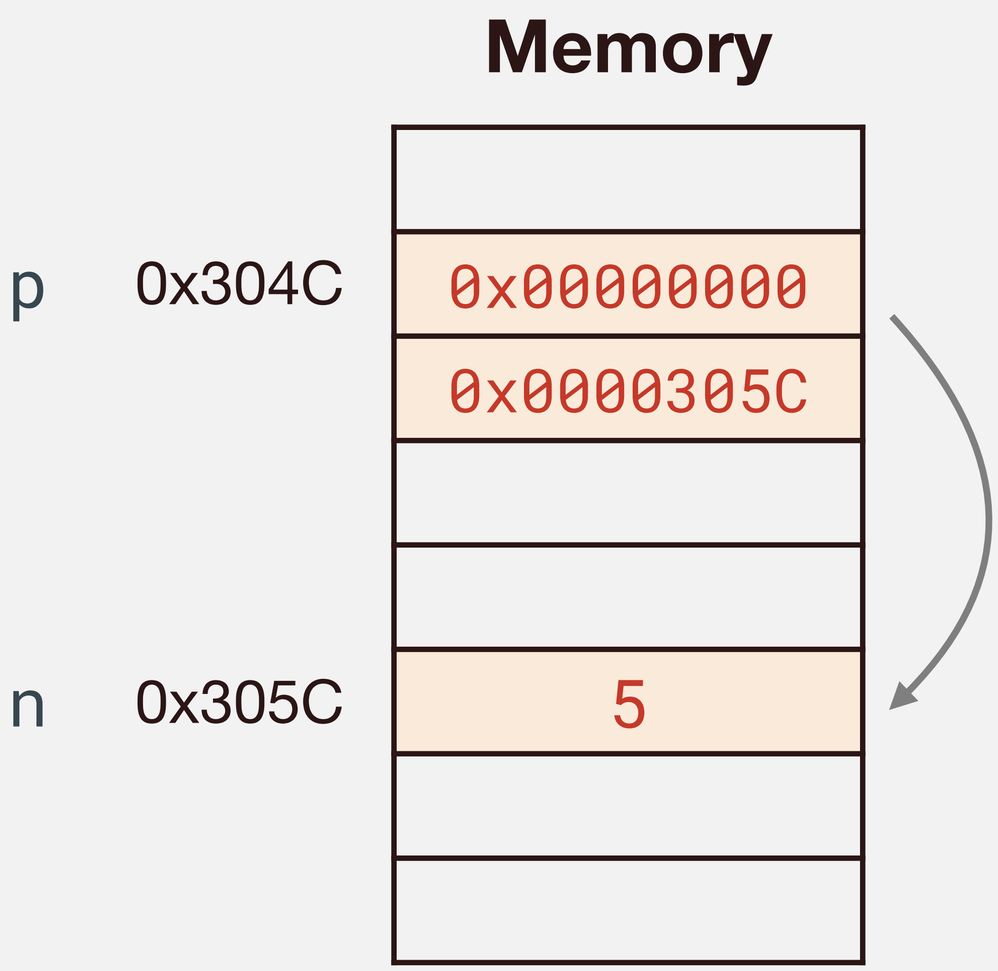


Address <—> Content

```
int main() {
  int n = 5;
  int *p;
  p = &n;
  int t = *p;

}
```

← Take the address of …

← Get the value stored at address …

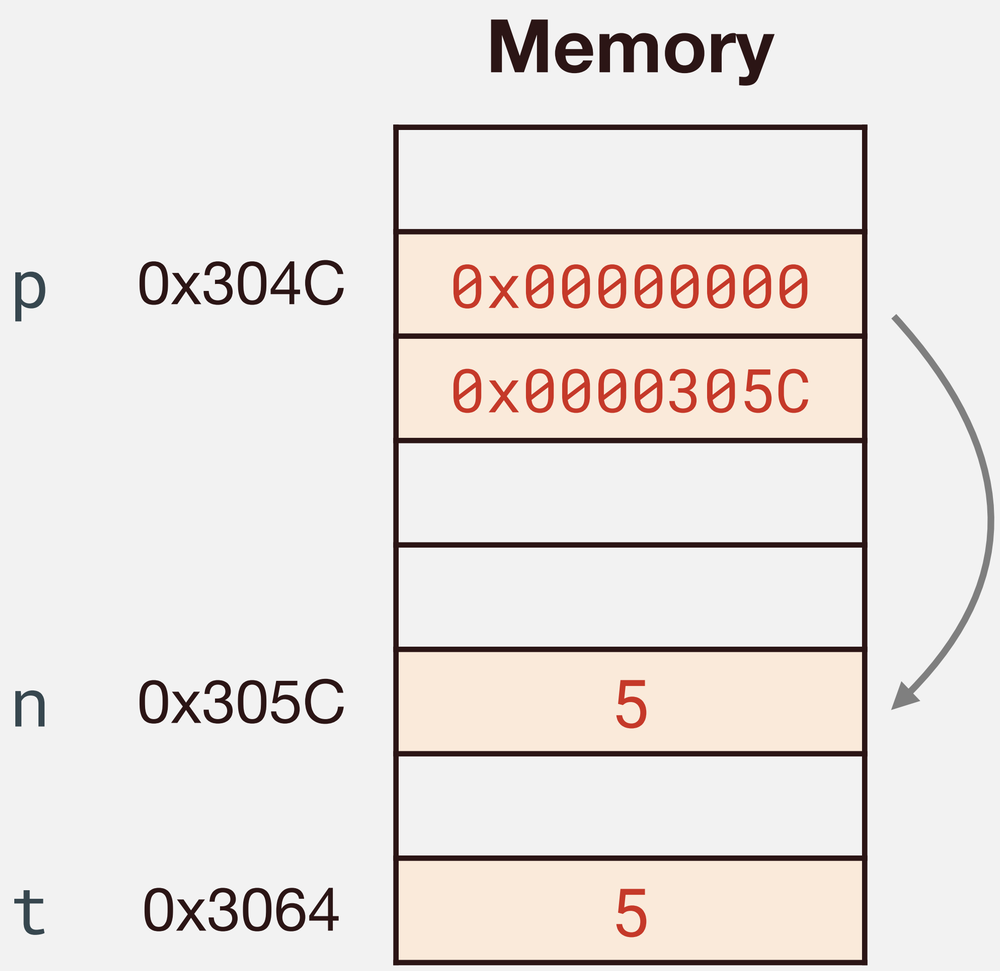


Address <—> Content

```
int main() {
  int n = 5;
  int *p;
  p = &n;
  int t = *p;

}
```

← Take the address of …

← Get the value stored at address …

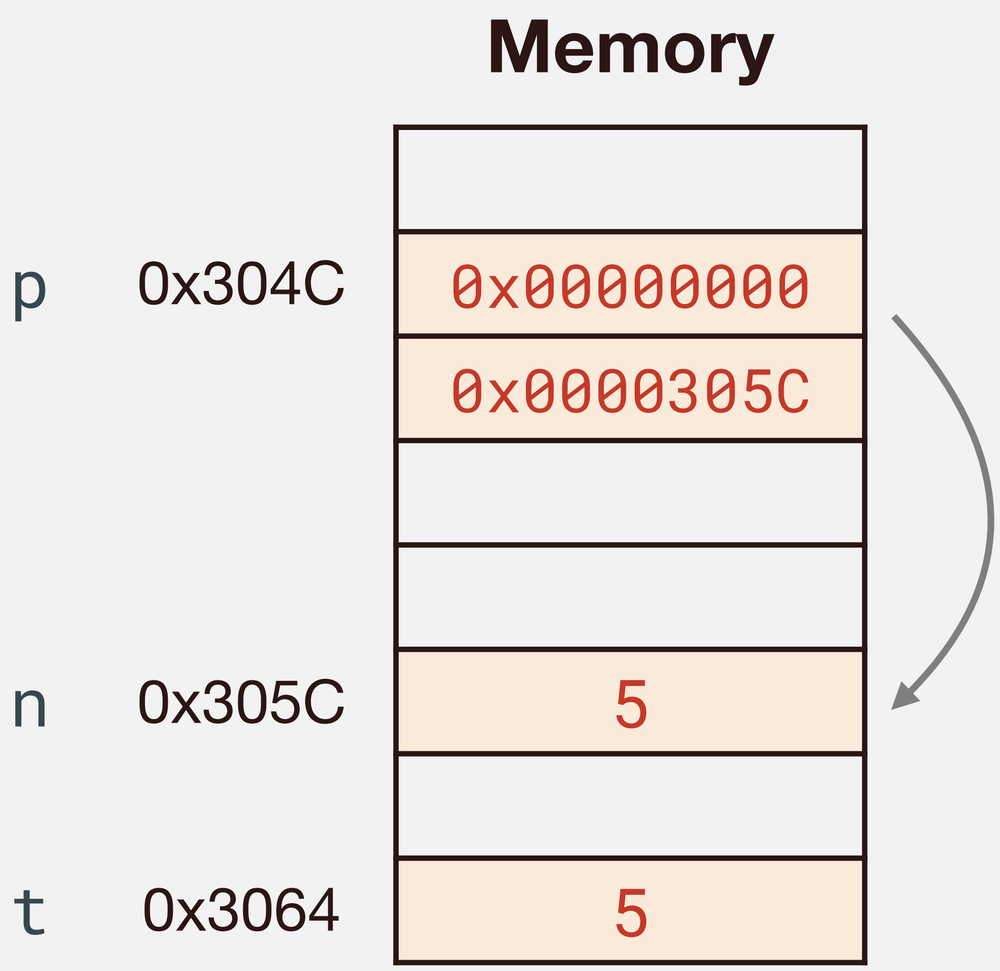


Address <—> Content

```
int main() {
  int n = 5;
  int *p;
  p = &n;
  int t = *p;
  *p = 6;

}
```

Take the address of …

Get the value stored at address …

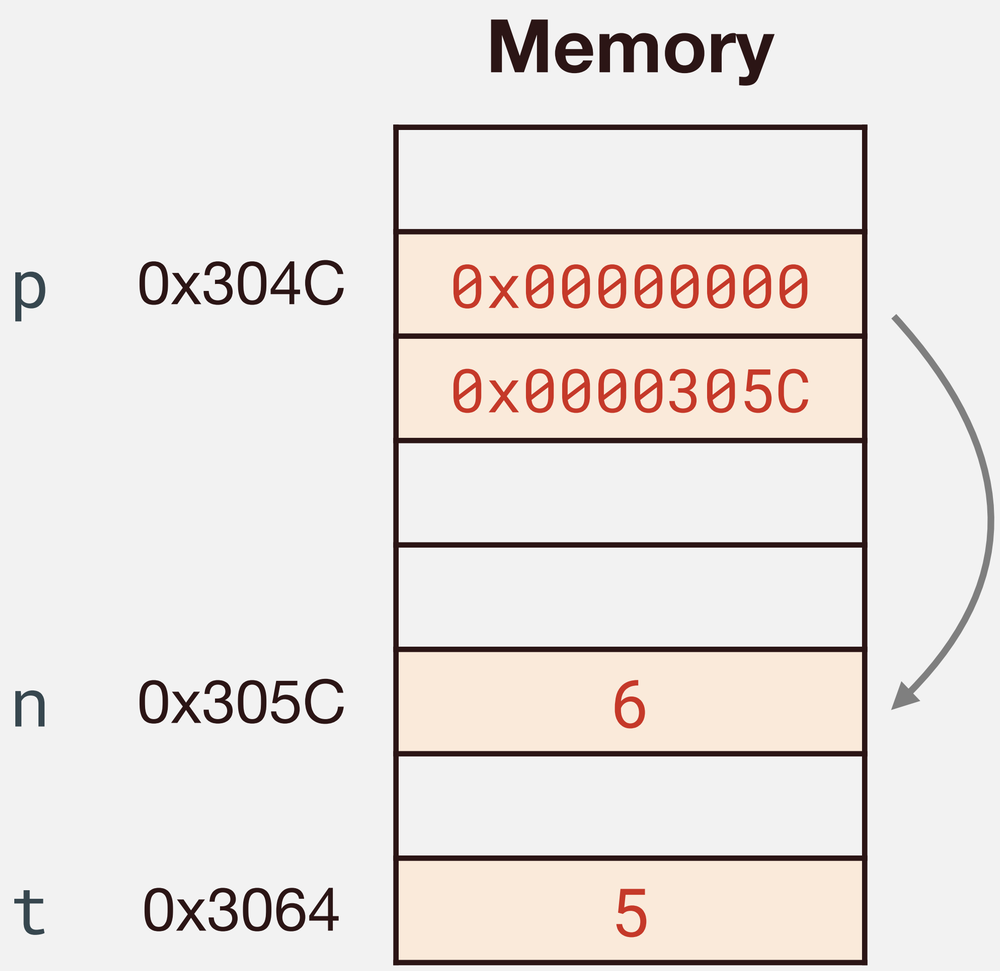


Address <—> Content

```
int main() {
  int n = 5;
  int *p;
  p = &n;
  int t = *p;
  *p = 6;

}
```

Take the address of …

Get the value stored at address …

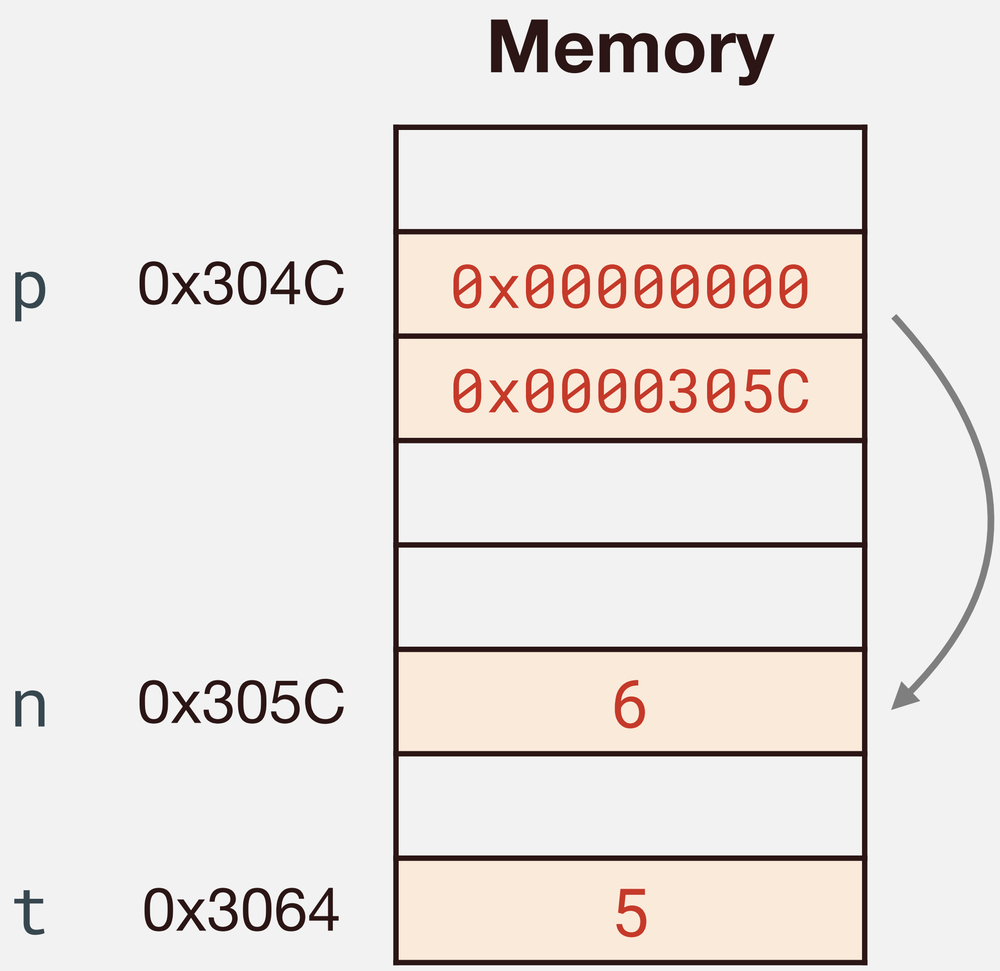


Address <—> Content

```
int main() {
  int n = 5;
  int *p;
  p = &n;
  int t = *p;
  *p = 6;
  printf("%d\n", n);


}
```

← Take the address of …

← Get the value stored at address …

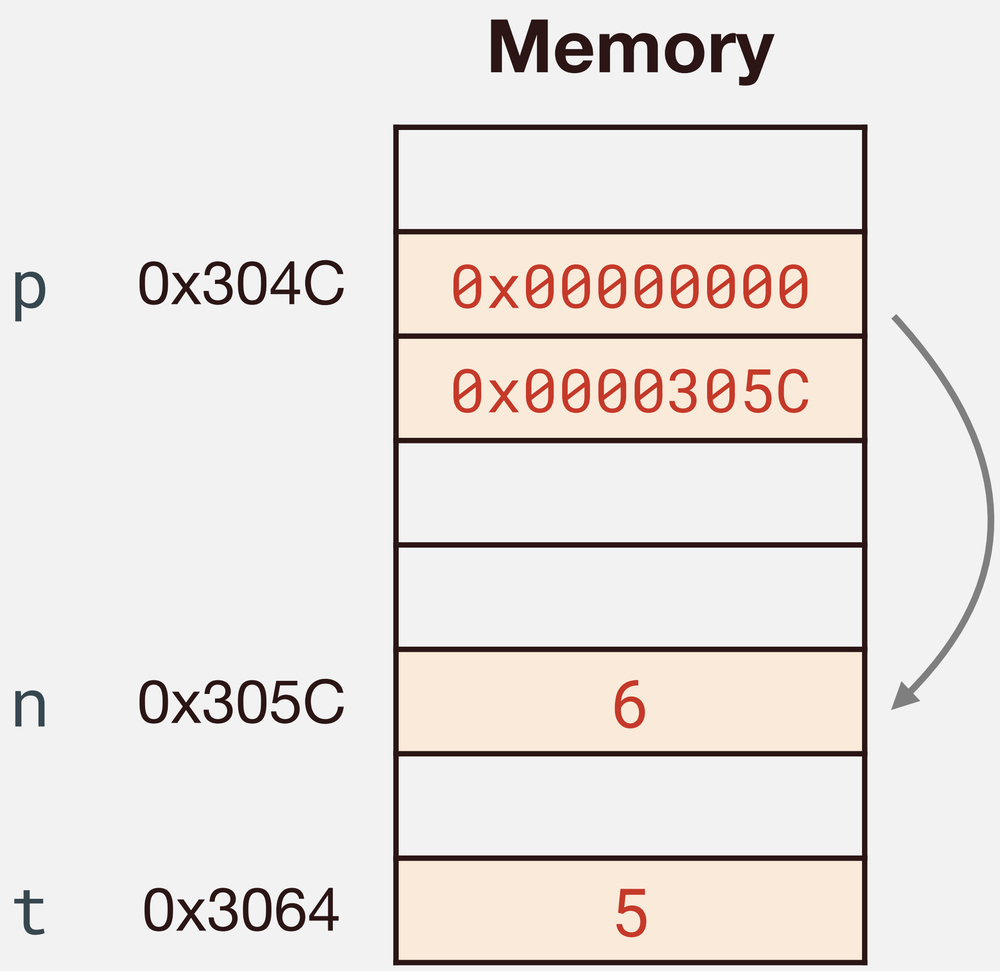


Address <—> Content

```
int main() {
  int n = 5;
  int *p;
  p = &n;
  int t = *p;
  *p = 6;
  printf("%d\n", n);
  printf("%p\n", &n);

}
```

← Take the address of …

← Get the value stored at address …

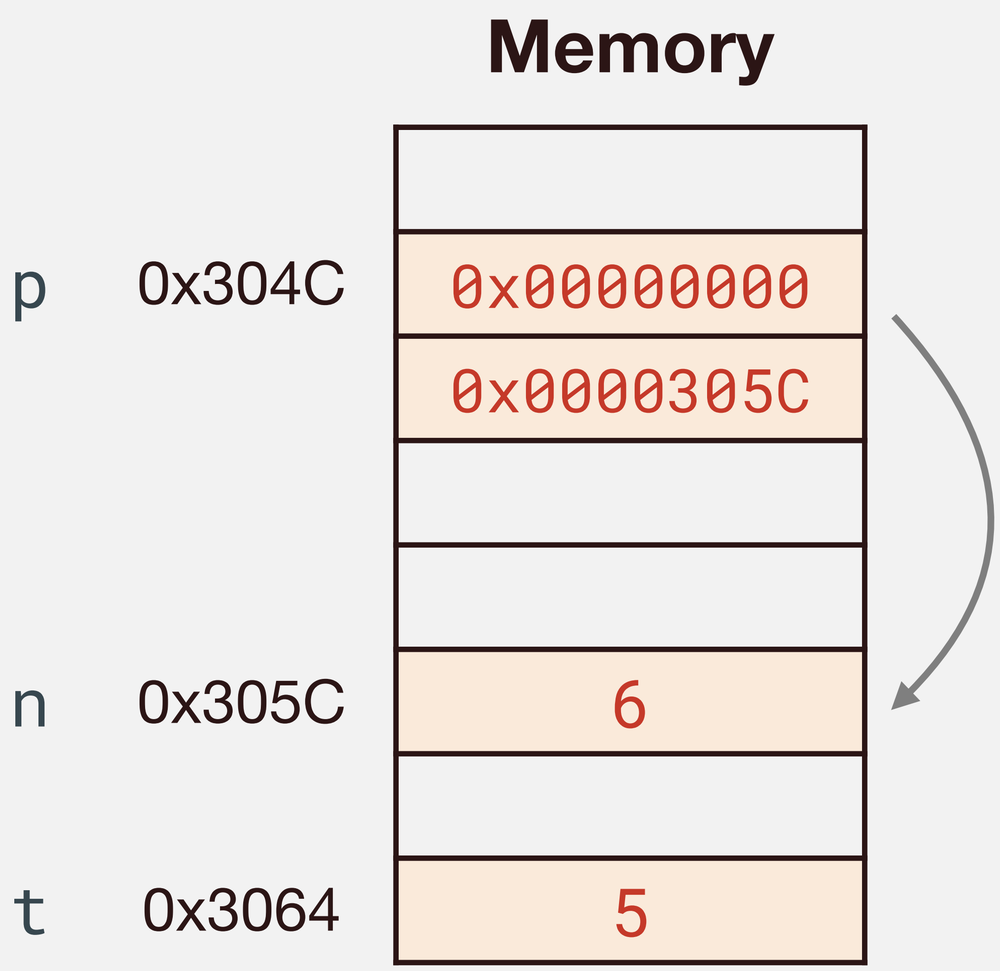


Address <—> Content

```
int main() {
  int n = 5;
  int *p;
  p = &n;
  int t = *p;
  *p = 6;
  printf("%d\n", n);
  printf("%p\n", &n);
  printf("%d\n", t);
}
```

Take the address of …

Get the value stored at address …



Pointer Declaration Styles

Style 1

```
int *p;
```

Style 2

```
int* p;
```

Pointer Declaration Styles

Style 1

```
int *p;

int *p = &n;
```

Style 2

```
int* p;

int* p = &n;
```

Pointer Declaration Styles

Style 1

```
int *p;

int *p = &n;
```

Style 2

```
int* p;

★ int* p = &n;
```

Pointer Declaration Styles

Style 1

```
int *p;
int *p = &n;
int *p, c;
```

Style 2

```
int* p;
int* p = &n;
int* p, c;
```

Pointer Declaration Styles

Style 1

```
int *p;
int *p = &n;
int *p, c;
```

Style 2

```
int* p;
int* p = &n;
int* p, c;
```

Pointers as Function Arguments

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
  swap(a, b);
}
```

Pointers as Function Arguments

```
void swap(int *a, int *b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
  swap(a, b);
}
```

Pointers as Function Arguments

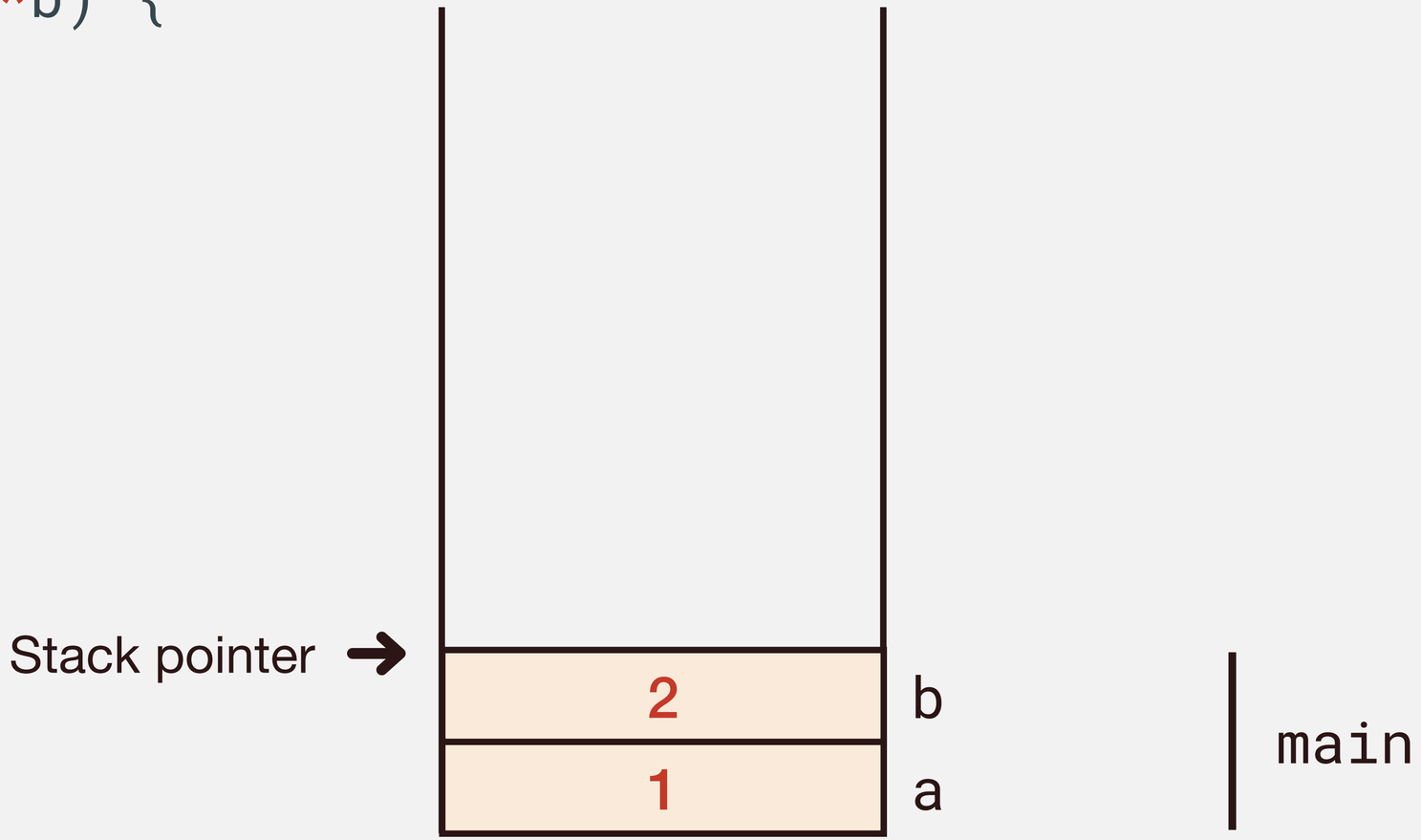
```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
  swap(a, b);
}
```

Pointers as Function Arguments

```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
  swap(&a, &b);
}
```

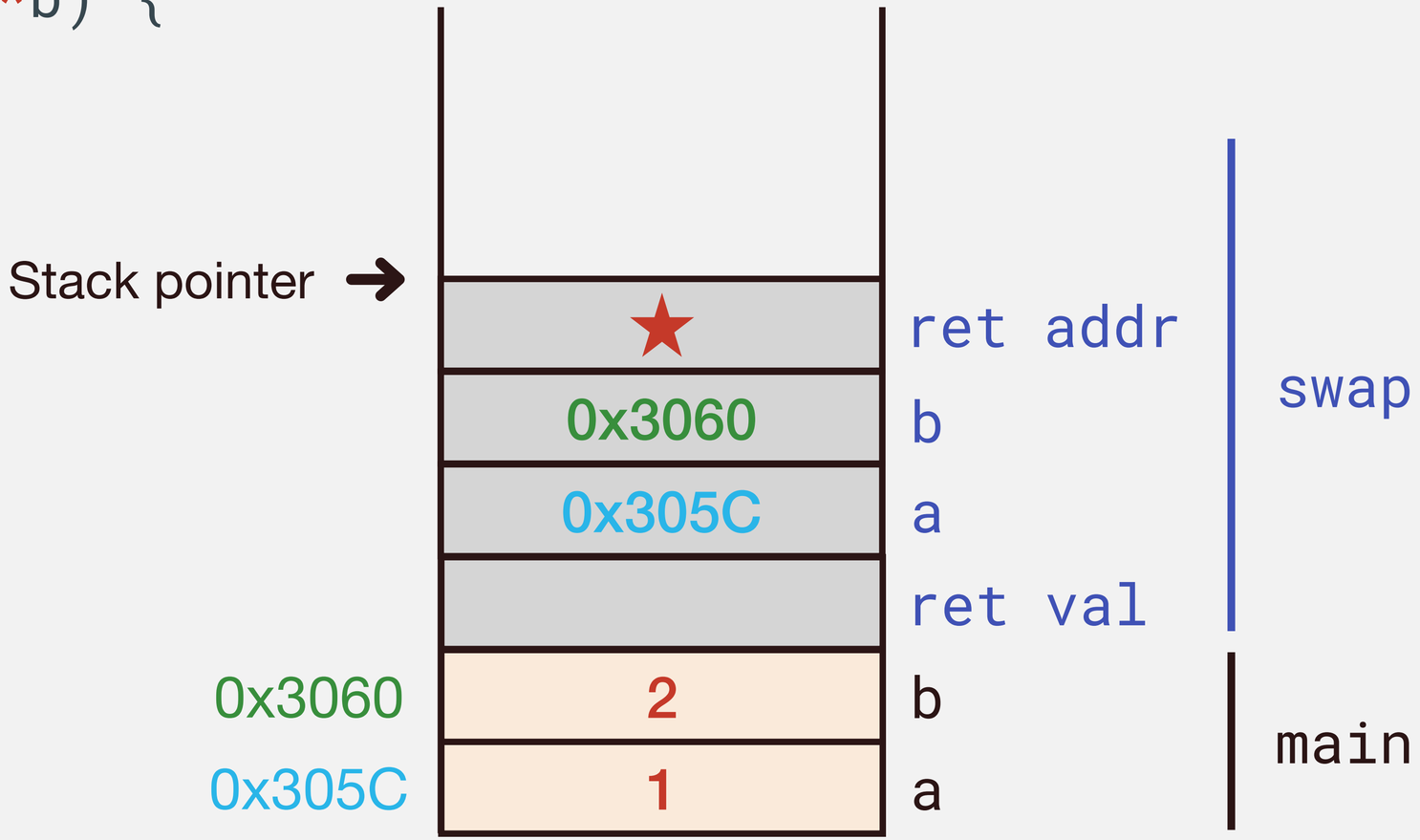
Pointers as Function Arguments

```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
➜ swap(&a, &b);
}
```



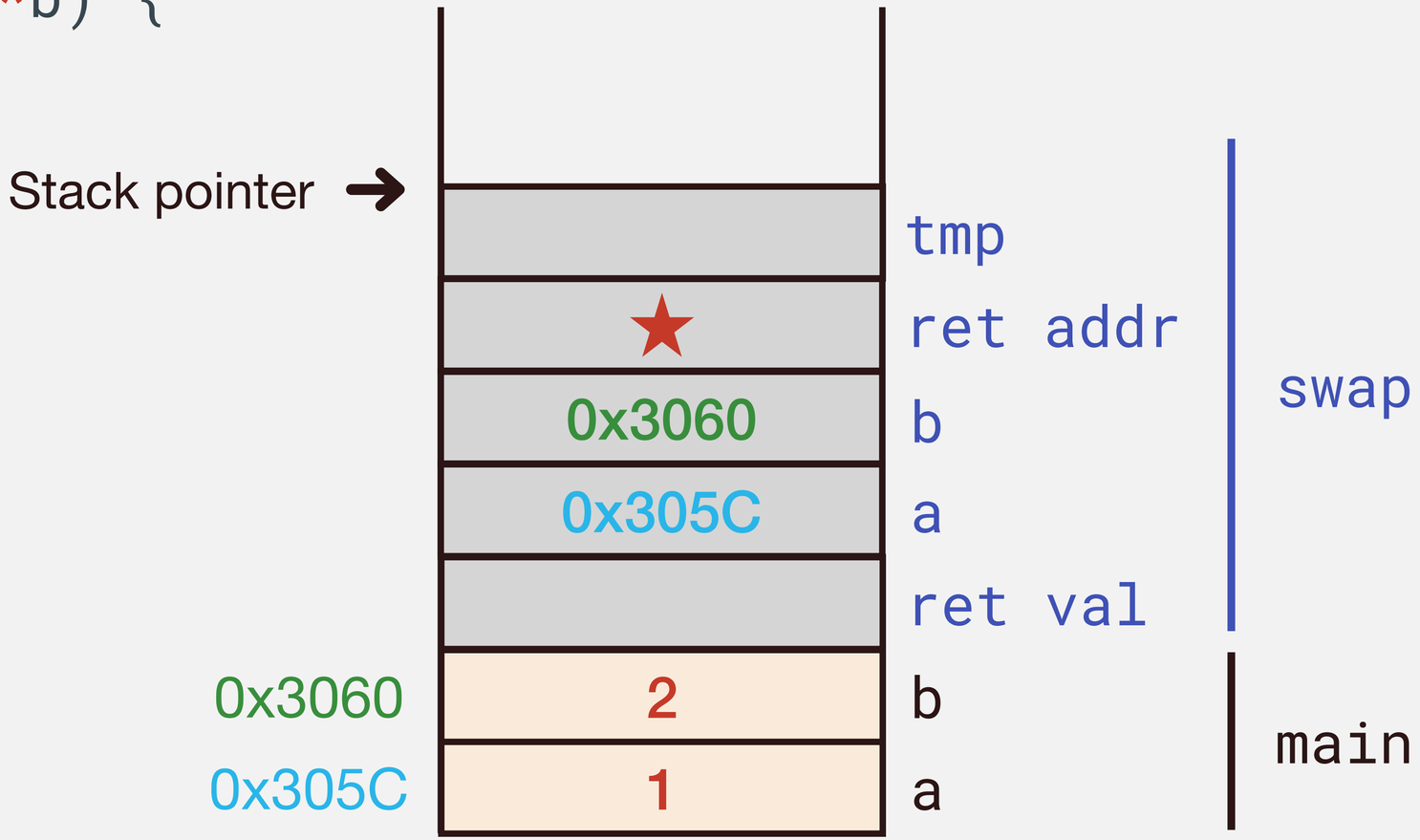
Pointers as Function Arguments

```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
➜ swap(&a, &b);
}
```



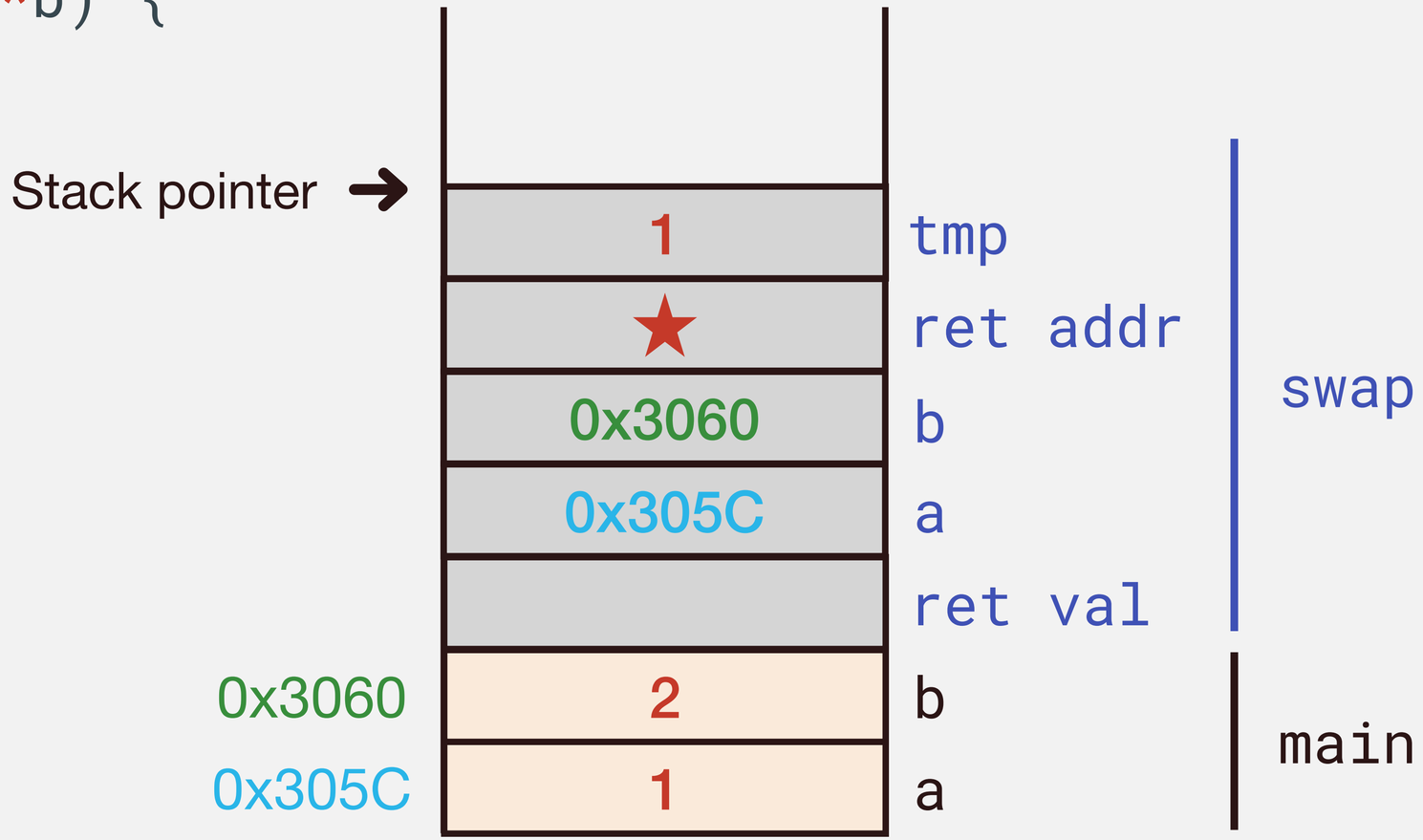
Pointers as Function Arguments

```
void swap(int *a, int *b) {
➔ int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
  swap(&a, &b);
}
```



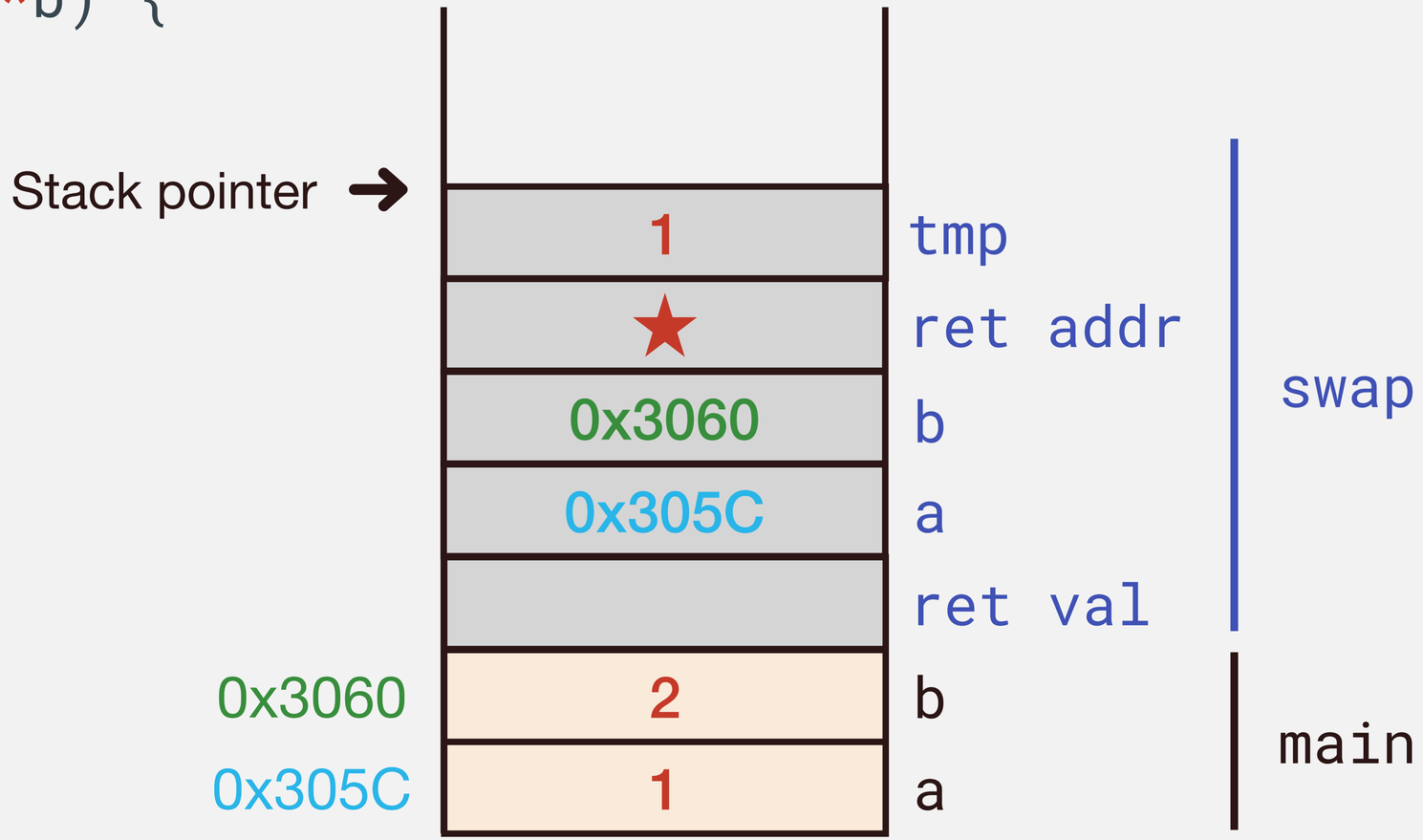
Pointers as Function Arguments

```
void swap(int *a, int *b) {
➔ int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
  swap(&a, &b);
}
```



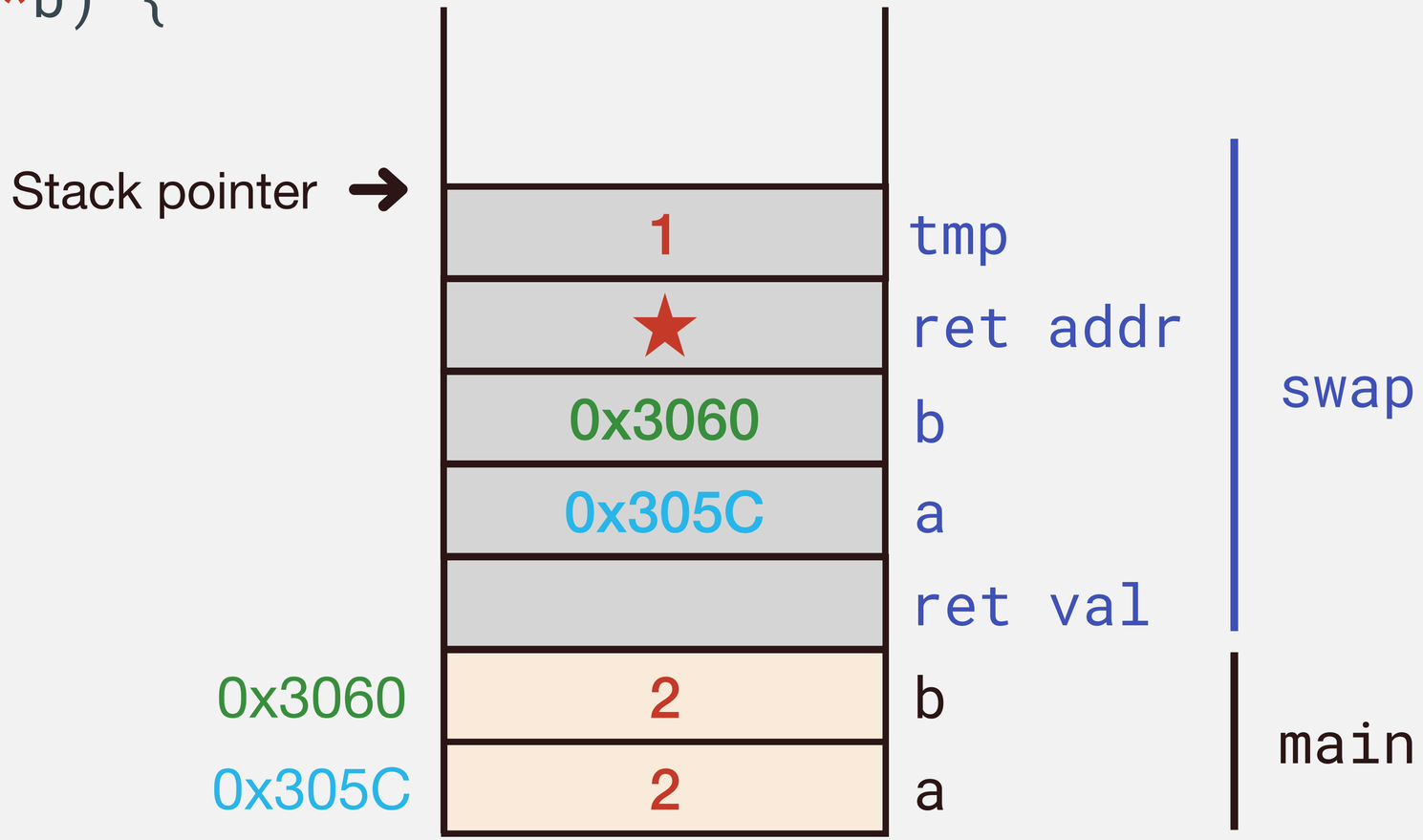
Pointers as Function Arguments

```
void swap(int *a, int *b) {
  int tmp = *a;
➔ *a = *b;
  *b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
  swap(&a, &b);
}
```



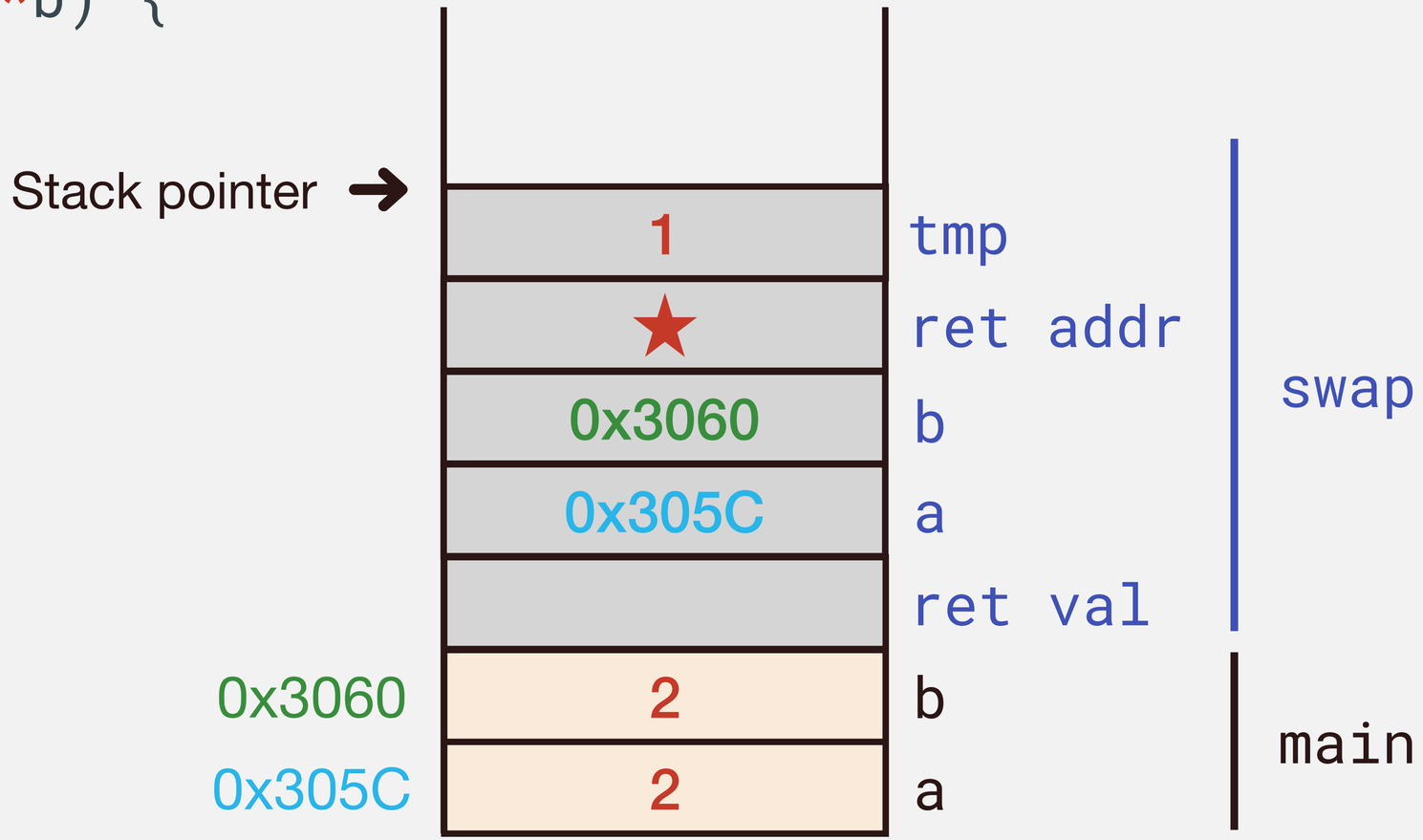
Pointers as Function Arguments

```
void swap(int *a, int *b) {
  int tmp = *a;
➔ *a = *b;
  *b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
  swap(&a, &b);
}
```



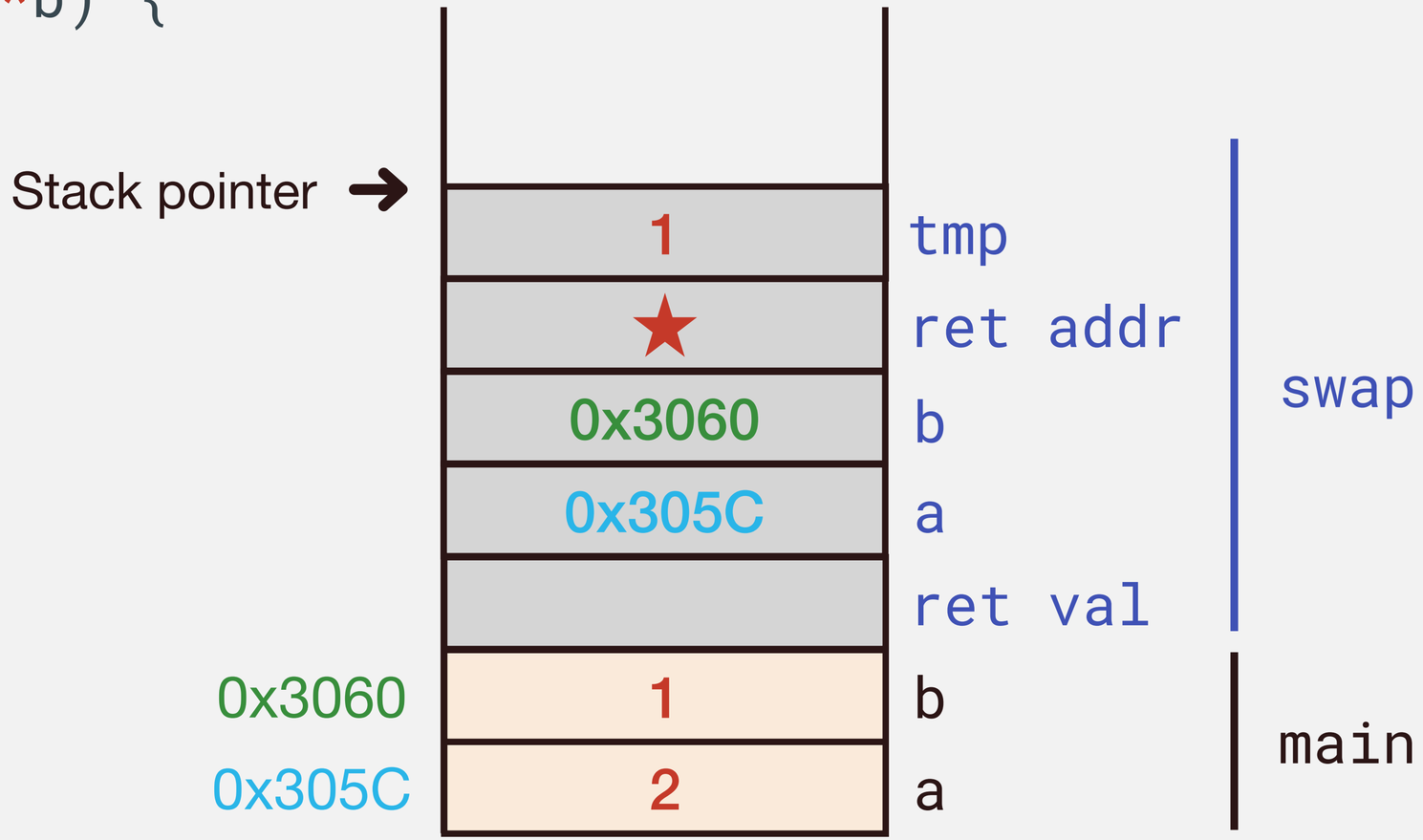
Pointers as Function Arguments

```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
➔ *b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
  swap(&a, &b);
}
```



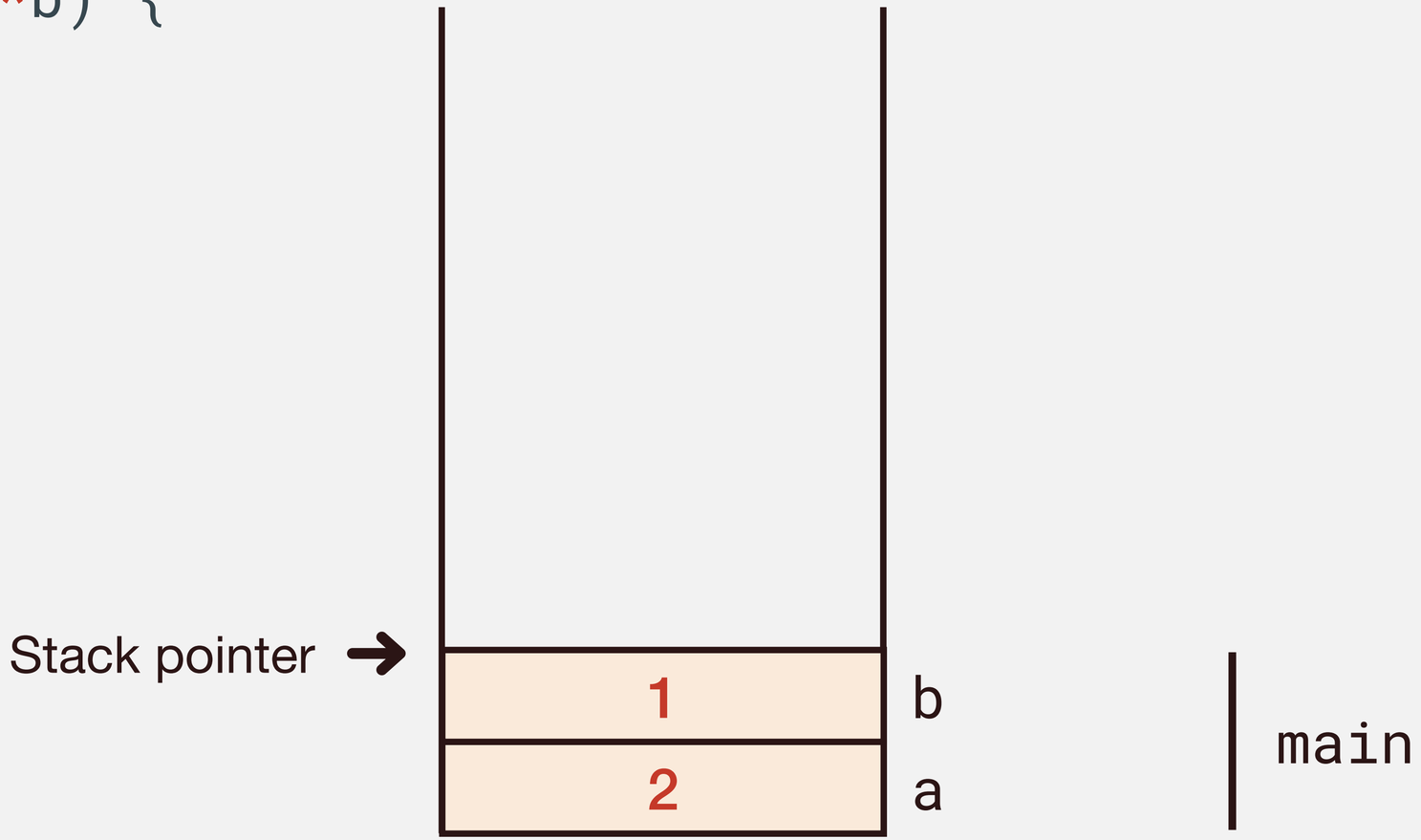
Pointers as Function Arguments

```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
➔ *b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
  swap(&a, &b);
}
```

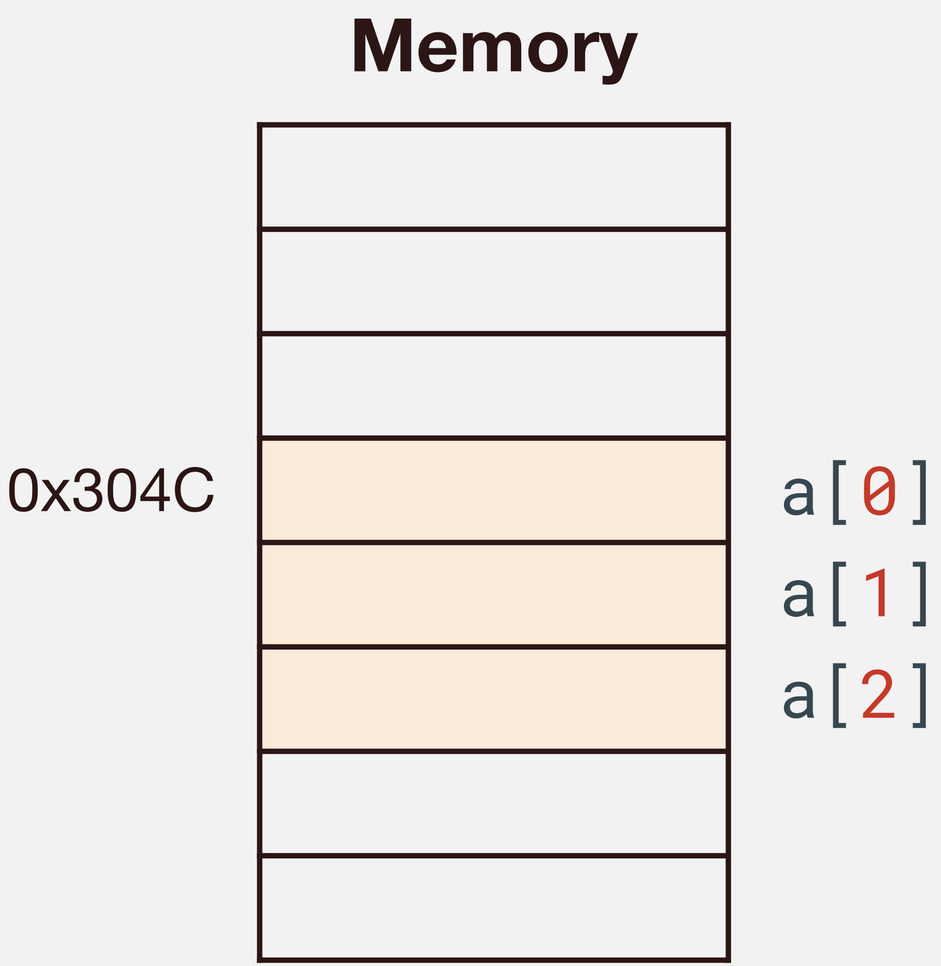


Pointers as Function Arguments

```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
➜ swap(&a, &b);
}
```

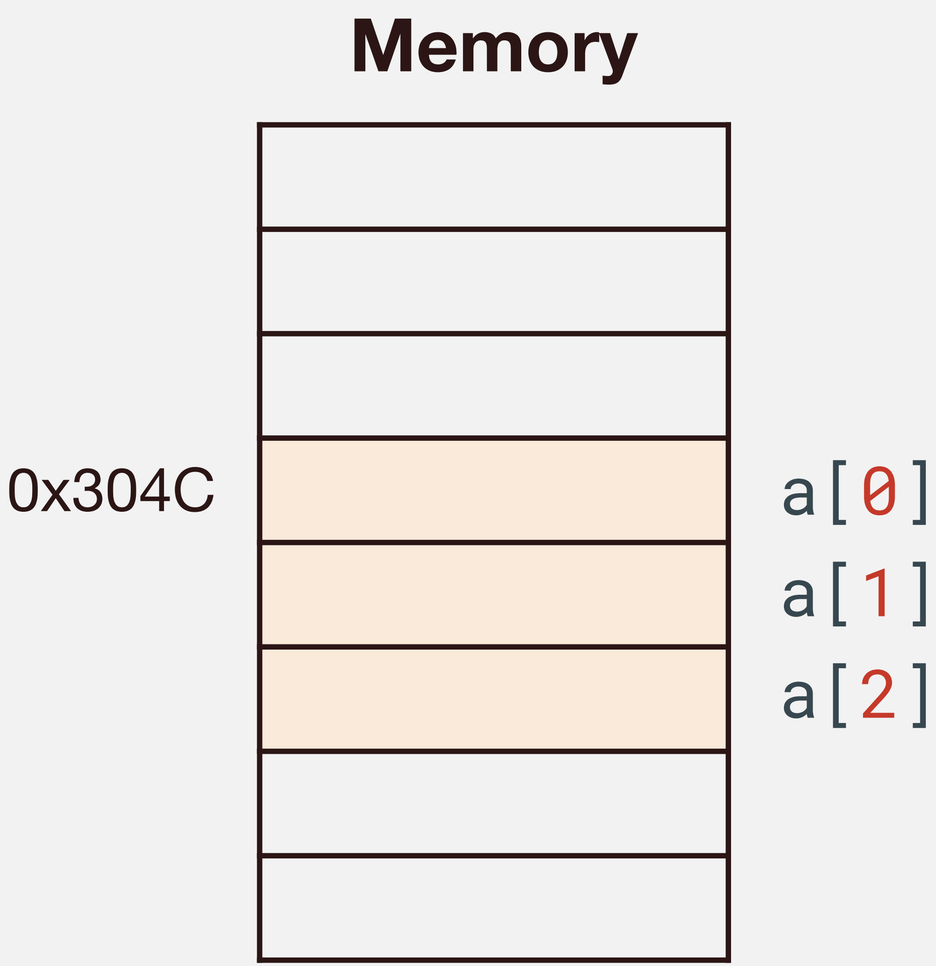


Pointers and Arrays




```
int a[3];
```

Pointers and Arrays

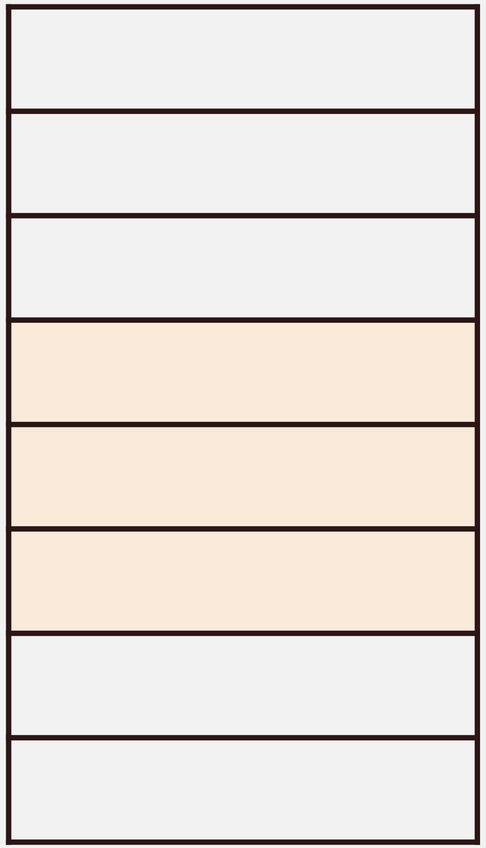



```
int a[3];
```

pointer to the first element

Pointers and Arrays

Memory



0x304C

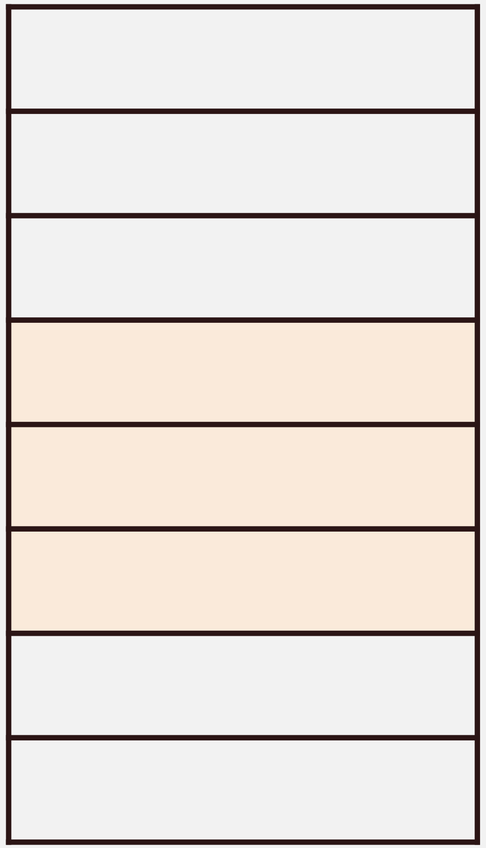
```
int a[3];
```

pointer to the first element

```
a[0] ⟷ *a
a[1]
a[2]
```

Pointers and Arrays

Memory



0x304C

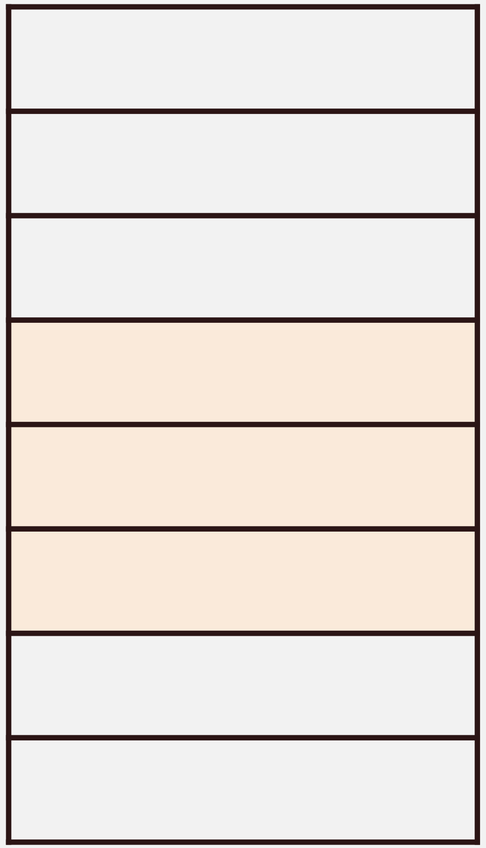
```
int a[3];
```

pointer to the first element

```
a[0] ⟷ *a
a[1] ⟷ *(a + 1)
a[2]
```

Pointers and Arrays

Memory



0x304C

```
int a[3];
```

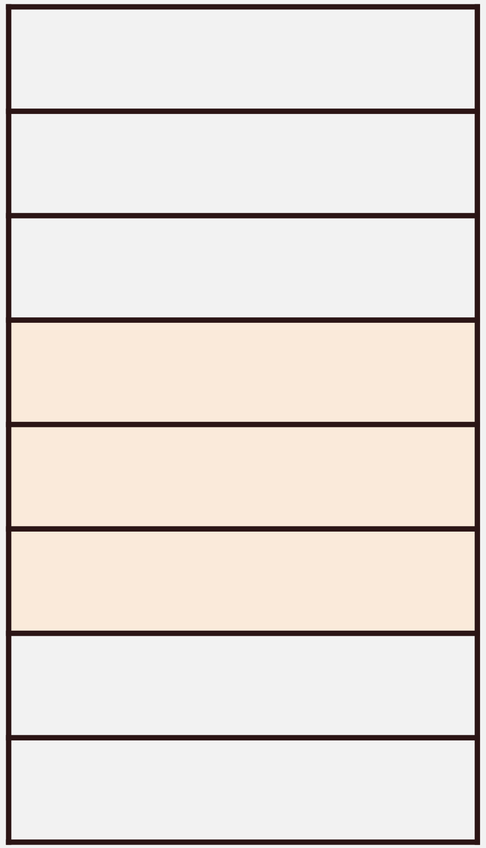
pointer to the first element

```
a[0] ⟷ *a
a[1] ⟷ *(a + 1)
a[2]
```

pointer arithmetic

Pointers and Arrays

Memory



0x304C

```
int a[3];
```

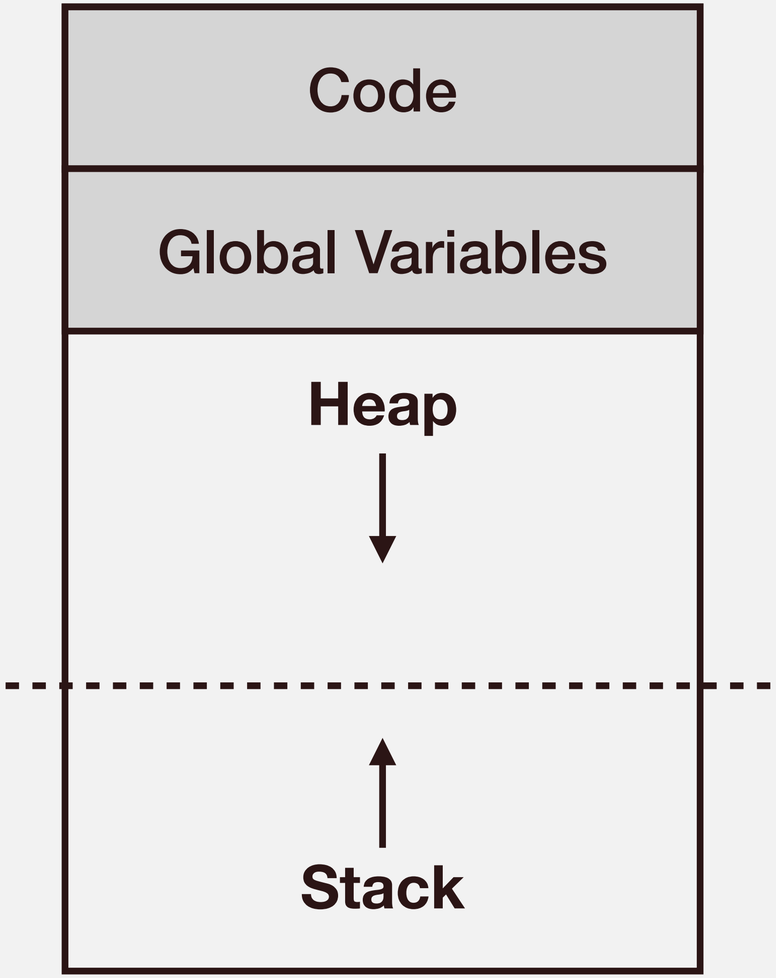
pointer to the first element

```
a[0] ⟷ *a
a[1] ⟷ *(a + 1)
a[2]
```

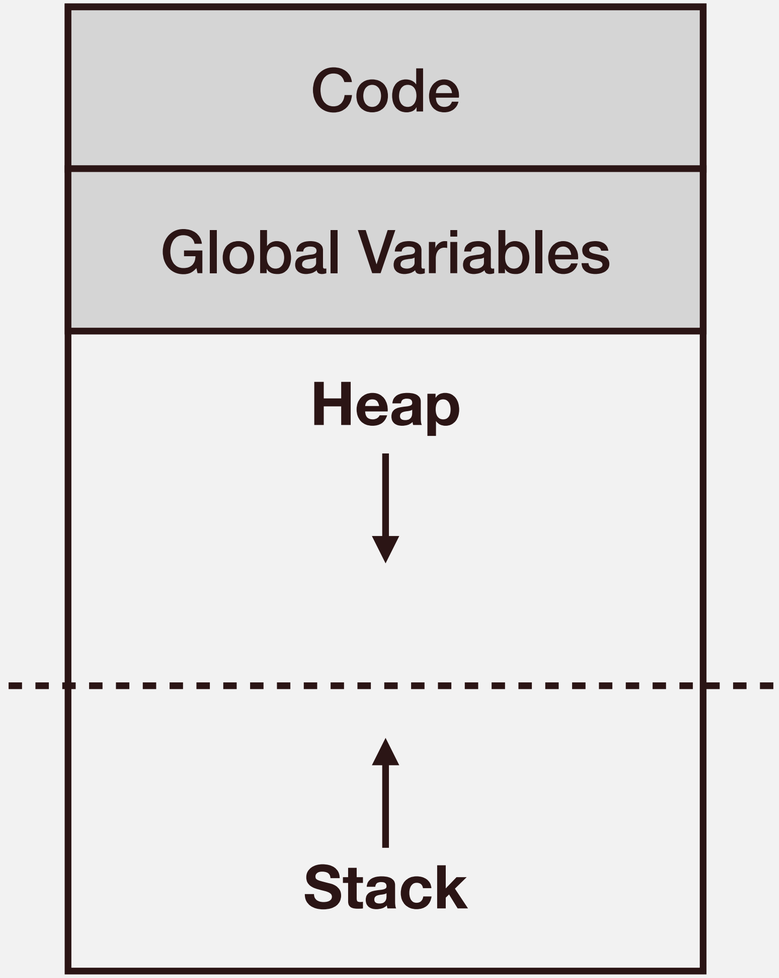
pointer arithmetic

```
a + 1
0x304C + 1 * sizeof(int)
```

Memory Layout

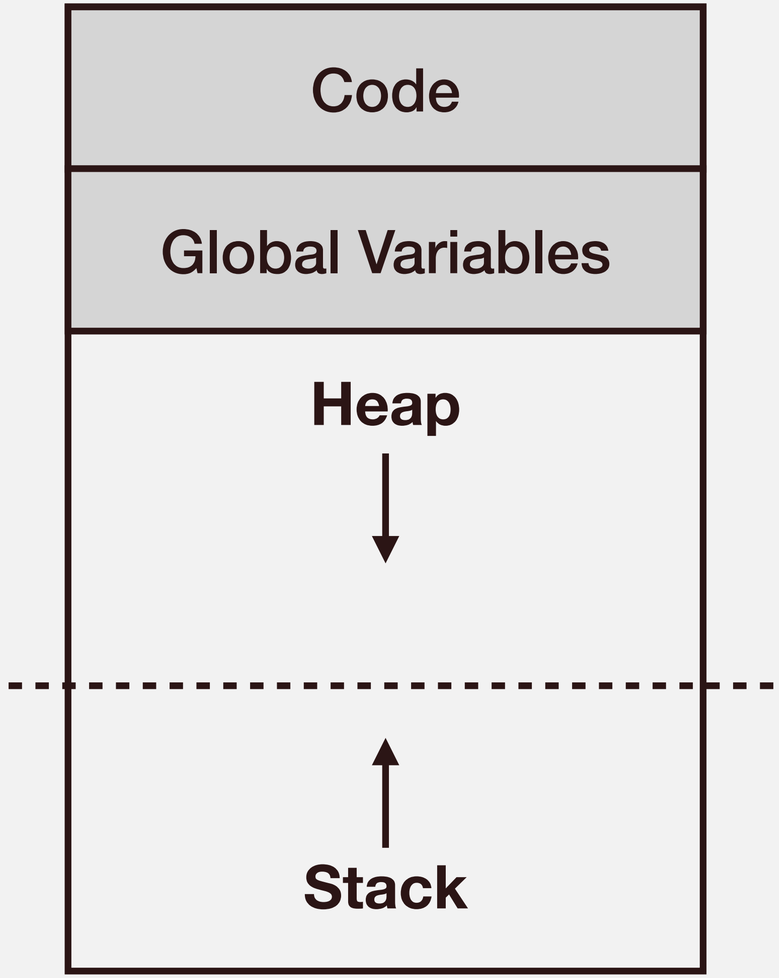


Memory Layout




```
int x;
char a[64];
```

Memory Layout




```
int *p = malloc(size)
```

```
int x;
char a[64];
```

Dynamic Memory Allocation

```
malloc(byte_size)
```

Dynamic Memory Allocation

`malloc(byte_size)` → Allocates a chunk of memory from heap of size `byte_size`

Dynamic Memory Allocation

`malloc(byte_size)` → Allocates a chunk of memory from heap of size `byte_size`

Returns the start address with type `void*`

Dynamic Memory Allocation

`malloc(byte_size)` → Allocates a chunk of memory from heap of size `byte_size`

Returns the start address with type `void*`

```
int *a = (int *)malloc(100 * sizeof(int));
```

Dynamic Memory Allocation

`malloc(byte_size)` → Allocates a chunk of memory from heap of size `byte_size`

Returns the start address with type `void*`

```
int *a = (int *)malloc(100 * sizeof(int));
int a[100];
```

Dynamic Memory Allocation

`malloc(byte_size)` → Allocates a chunk of memory from heap of size `byte_size`

Returns the start address with type `void*`

```
int *a = (int *)malloc(100 * sizeof(int));
```
Heap Array

```
int a[100];
```
Stack Array

Dynamic Memory Allocation

`malloc(byte_size)` → Allocates a chunk of memory from heap of size `byte_size`

Returns the start address with type `void*`

```
int *a = (int *)malloc(100 * sizeof(int));
```
Heap Array

```
int a[100];
```
Stack Array

```
free(pointer)
```

Dynamic Memory Allocation

`malloc(byte_size)` → Allocates a chunk of memory from heap of size `byte_size`

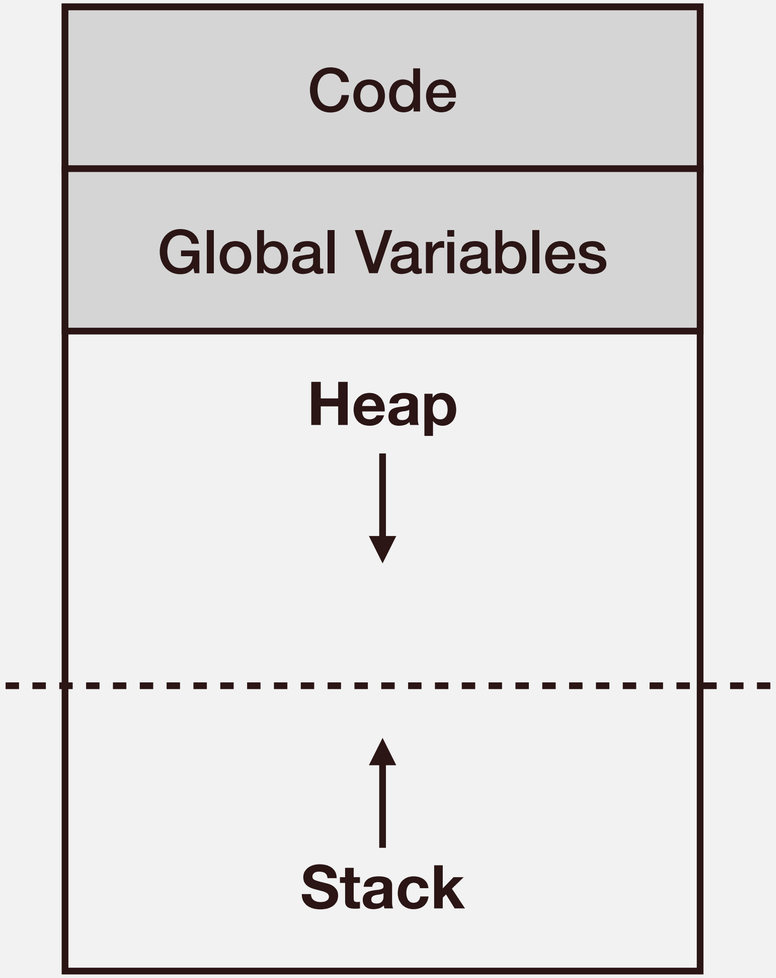
Returns the start address with type `void*`

```
int *a = (int *)malloc(100 * sizeof(int));
```
Heap Array

```
int a[100];
```
Stack Array

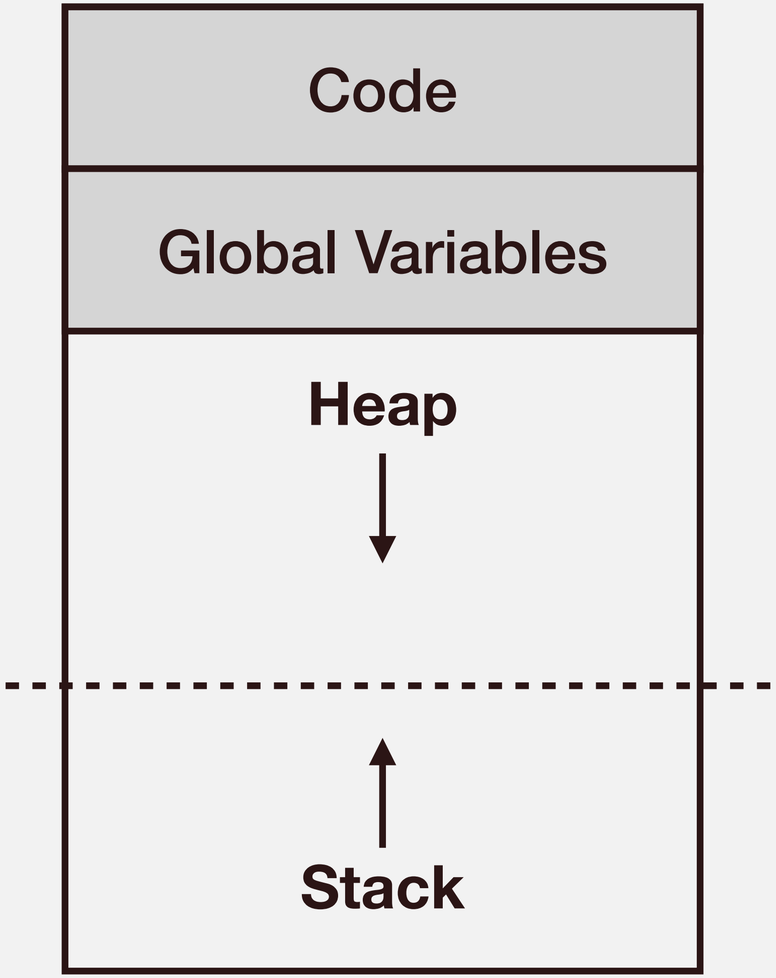
`free(pointer)` → Reclaim the memory for later use

Memory Layout



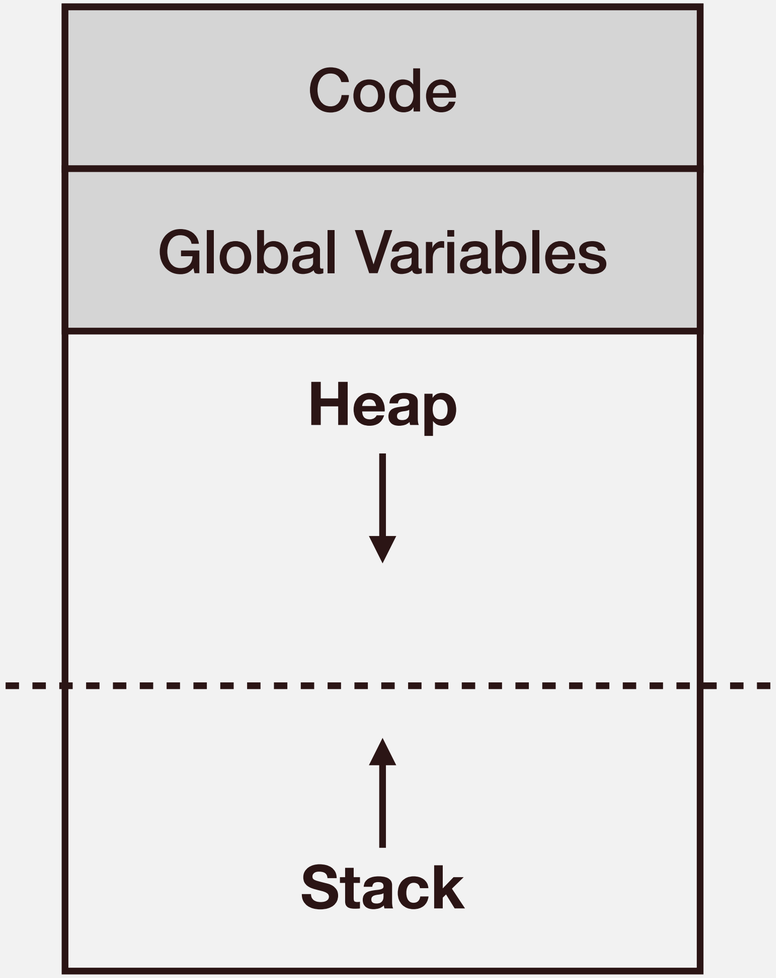
	Stack	Heap
Scope		
Size Limit		
GC		
Resize?		
Usage		

Memory Layout



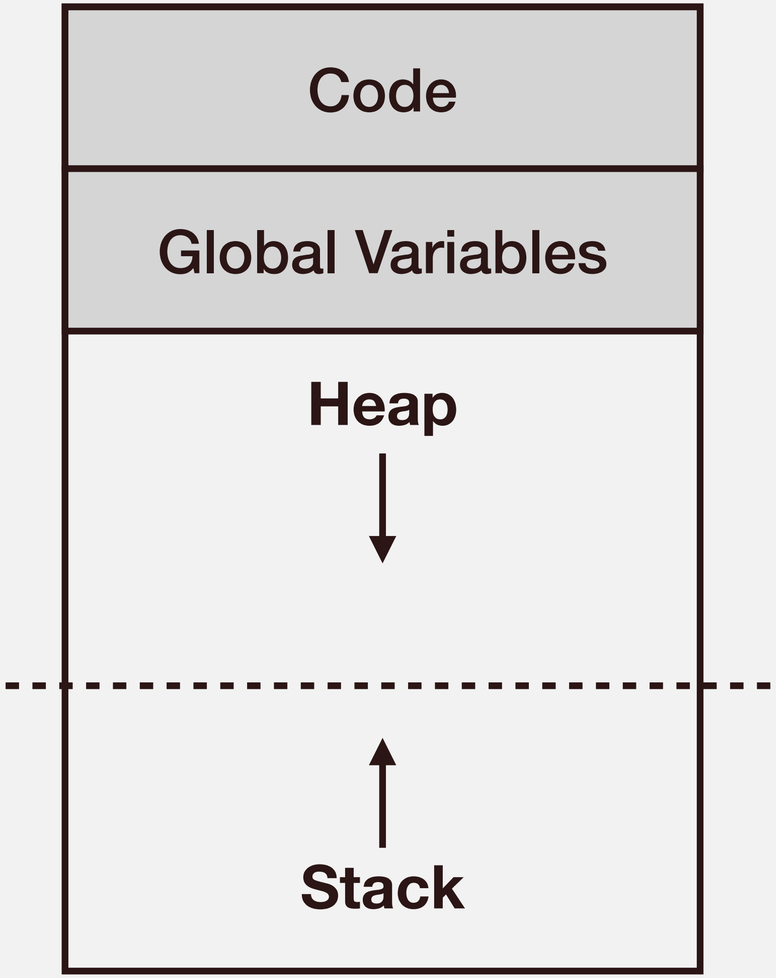
	Stack	Heap
Scope	Local	Global
Size Limit		
GC		
Resize?		
Usage		

Memory Layout



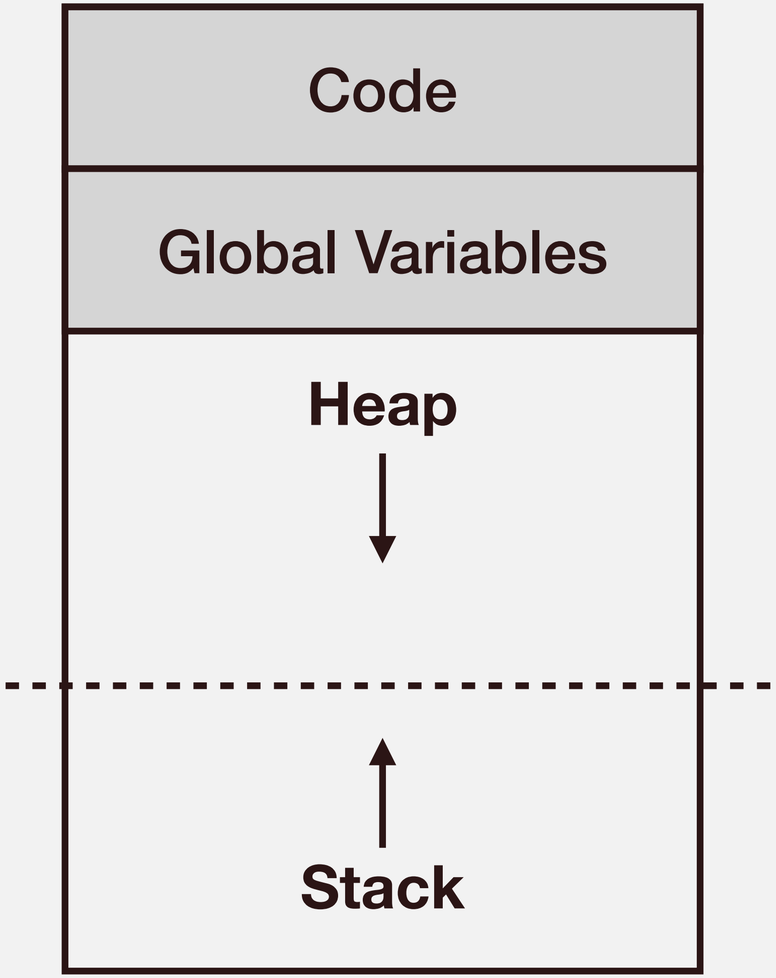
	Stack	Heap
Scope	Local	Global
Size Limit	By OS	Physical Mem
GC		
Resize?		
Usage		

Memory Layout



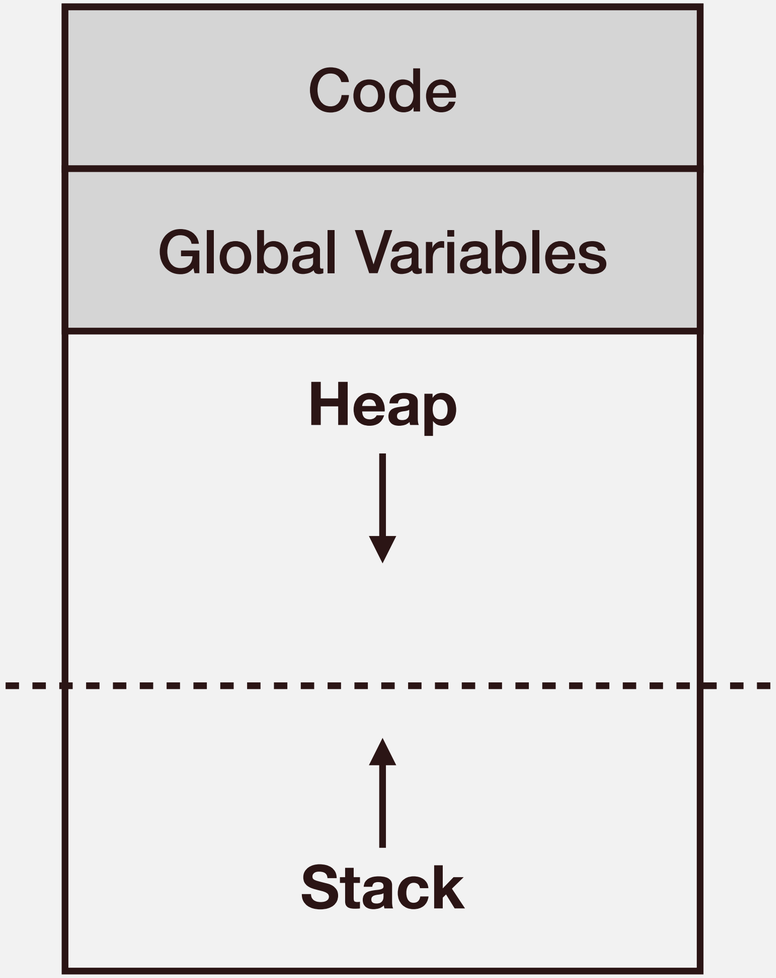
	Stack	Heap
Scope	Local	Global
Size Limit	By OS	Physical Mem
GC	Automatic	Manual
Resize?		
Usage		

Memory Layout



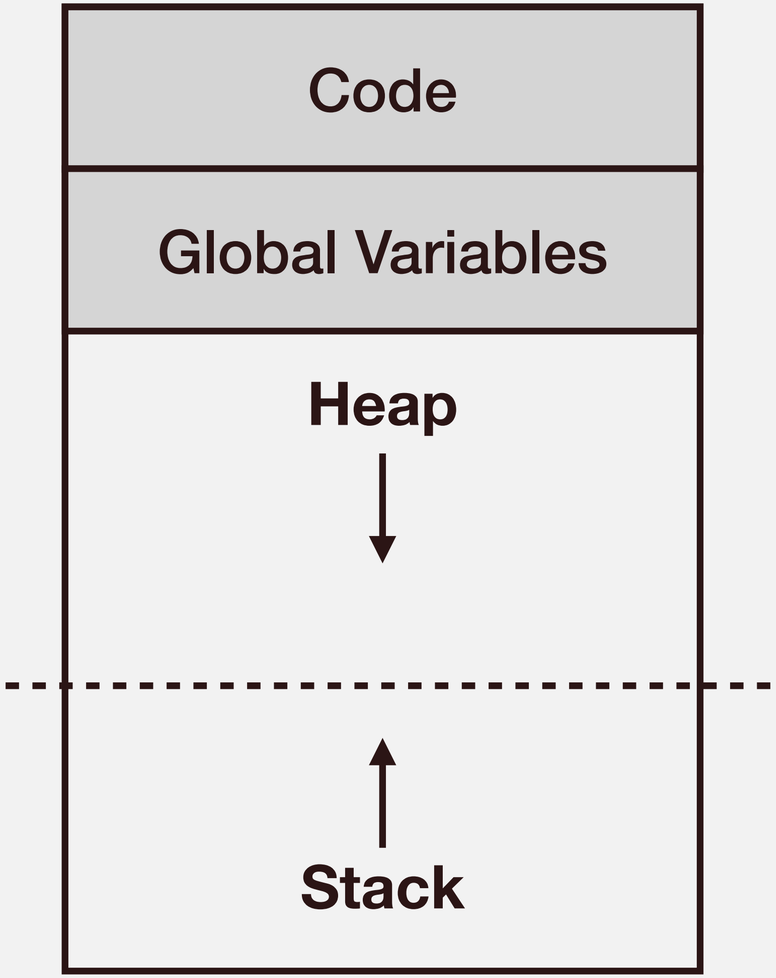
	Stack	Heap
Scope	Local	Global
Size Limit	By OS	Physical Mem
GC	Automatic	Manual
Resize?	No	Yes
Usage		

Memory Layout



	Stack	Heap
Scope	Local	Global
Size Limit	By OS	Physical Mem
GC	Automatic	Manual
Resize?	No	Yes
Usage	Local var, function	

Memory Layout



	Stack	Heap
Scope	Local	Global
Size Limit	By OS	Physical Mem
GC	Automatic	Manual
Resize?	No	Yes
Usage	Local var, function	Global/Large var

Segmentation Fault

- Attempt to access memory locations you are not allowed to visit

Segmentation Fault

- Attempt to access memory locations you are not allowed to visit
- **Common Reasons**
  1. Dereferencing NULL or uninitialized pointers

Segmentation Fault

- Attempt to access memory locations you are not allowed to visit
- **Common Reasons**
  1. Dereferencing NULL or uninitialized pointers
  2. Access freed memory

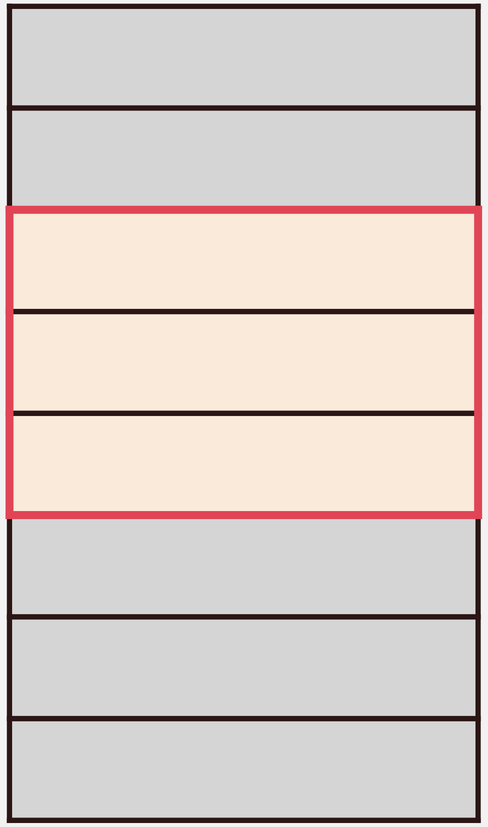
Segmentation Fault

- Attempt to access memory locations you are not allowed to visit

- **Common Reasons**
  1. Dereferencing NULL or uninitialized pointers
  2. Access freed memory
  3. Array index out of bound

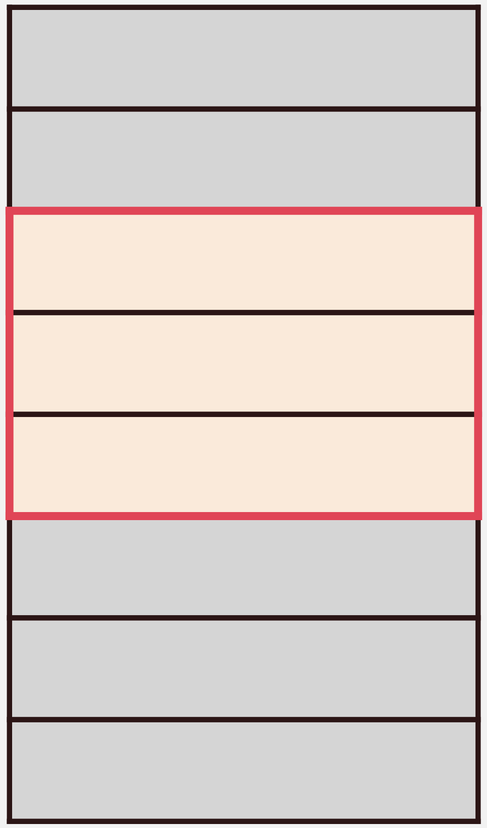
Index Out-of-Bound

Memory



```
array[-1]
array[0]
array[1]
array[2]
array[3]
```

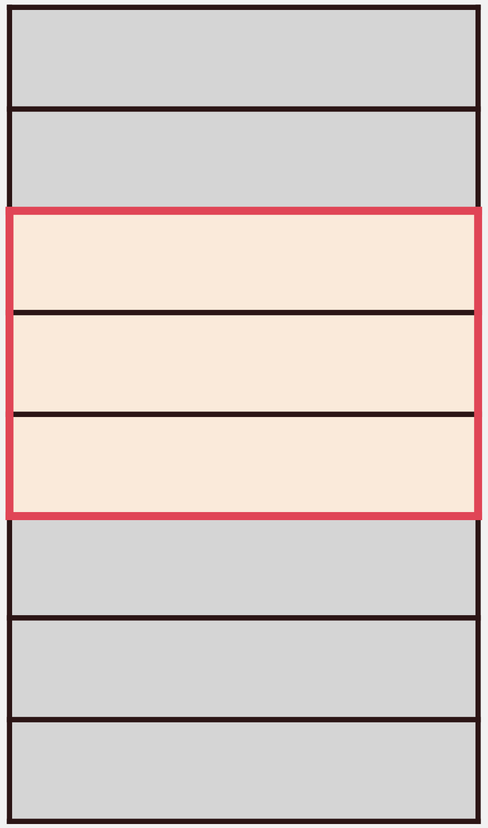
Index Out-of-Bound




```
array[-1]
array[0]
array[1]
array[2]
array[3]
```

Undefined behavior, super dangerous

Index Out-of-Bound




```
array[-1]
array[0]
array[1]
array[2]
array[3]
```

Undefined behavior, super dangerous

Don’t do that!

Segmentation Fault

- Attempt to access memory locations you are not allowed to visit

- **Common Reasons**
  1. Dereferencing NULL or uninitialized pointers
  2. Access freed memory
  3. Array index out of bound
  4. Stack overflow

Segmentation Fault

- Attempt to access memory locations you are not allowed to visit

- **Common Reasons**
  1. Dereferencing NULL or uninitialized pointers
  2. Access freed memory
  3. Array index out of bound
  4. Stack overflow

- How to debug?
  - `gdb` is your friend!

Common GDB Commands

`gdb prgm`	Enters GDB with program prgm loaded
`Ctrl-x Ctrl-a`	Enters the TUI mode
`[r]un <args>`	Runs the loaded program with command line arguments
`[b]ack[t]race`	Prints the call stack trace
`[b]reak <func_name or file_name:line#>`	Sets a breakpoint
`[d]elete #`	Deletes breakpoint #
`[s]tep`	Steps through a single line of code: INTO function calls
`[n]ext`	Steps through a single line of code: OVER function calls
`[p]rint <expr>`	Prints the current value of the expression
`[k]ill`	Kills the current debugging session
`[q]uit`	Quits GDB

Memory Leak

- Fail to release the memory you no longer need
  - Often unnoticed for short tasks
  - Problematic for long-running services

Memory Leak

- Fail to release the memory you no longer need
  - Often unnoticed for short tasks
  - Problematic for long-running services
- There **must be** a `free` for **each** `malloc`

Memory Leak

- Fail to release the memory you no longer need
  - Often unnoticed for short tasks
  - Problematic for long-running services
- There **must be** a `free` for **each** `malloc`
- How to debug?
  - Use `valgrind` (or leaks for MacOS)!

Summary

- **Control Flow**
- **Function Basics**
  - Call stack, pass-by-value, variable scope
- **Array & Pointer**
  - Pointer arithmetic, pointer as function argument, dynamic array
- **Debugging**
  - Segfault, memory leak
  - `gdb, valgrind`