Introduction to Programming (C/C++)

04: C Advanced

Huanchen Zhang

清華大學
Tsinghua University

交叉信息研究院
Institute for Interdisciplinary Information Sciences

Last Lecture

- **Array & Pointer**
  - Pointer arithmetic, pointer as function argument, dynamic array

- **Function**
  - Call stack, pass-by-value, variable scope

C String

- An array of characters terminated with the null character `'\0'`

"Tea Garden"

T	e	a	\20	G	a	r	d	e	n	\0

C String

```
char s[] = “Tea Garden”;
```

C String

```
char s[] = “Tea Garden”;
```

```
char s[] = {’T’, ‘e’, ‘a’, ‘ ’, ‘G’, ‘a’, ‘r’, ‘d’, ‘e’, ’n’, ‘\0’};
```

C String

```
char s[] = “Tea Garden”;
```

```
char s[] = {’T’, ‘e’, ‘a’, ‘ ’, ‘G’, ‘a’, ‘r’, ‘d’, ‘e’, ’n’, ‘\0’};
```

```
char s[11] = “Tea Garden”;
```

C String

```
char s[] = “Tea Garden”;
```

```
char s[] = {’T’, ‘e’, ‘a’, ‘ ’, ‘G’, ‘a’, ‘r’, ‘d’, ‘e’, ’n’, ‘\0’};
```

```
char s[11] = “Tea Garden”;
```

Make sure enough space is allocated

C String

```
char s[] = “Tea Garden”;
```

```
char *s = “Tea Garden”;
```

C String

```
char s[] = “Tea Garden”;
```

```
char *s = “Tea Garden”;
```

Allocates an 11-byte array on stack

C String

```
char s[] = “Tea Garden”;
```

Allocates an 11-byte array on stack

```
char *s = “Tea Garden”;
```

Allocates a pointer on stack

C String

```
char s[] = “Tea Garden”;
```

Allocates an 11-byte array on stack

Compact way to initialize the array

```
char *s = “Tea Garden”;
```

Allocates a pointer on stack

C String

```
char s[] = “Tea Garden”;
```

Allocates an 11-byte array on stack

Compact way to initialize the array

```
char *s = “Tea Garden”;
```

Allocates a pointer on stack

String Literal (**read-only**)

C String

```
char s[] = “Tea Garden”;
```

Allocates an 11-byte array on stack

Compact way to initialize the array

```
char *s = “Tea Garden”;
```

Allocates a pointer on stack

String Literal (**read-only**)

```
s[0] = 'S';
```

```
s[0] = 'S';
```

Undefined Behavior

C String

```
const char *s = “Tea Garden”;
s[0] = ’S';
```

C String

```
const char *s = “Tea Garden”;

s[0] = 'S';
```

Error will be caught by the **compiler**

C String

```
char s[] = “Tea Garden”;
```

s	T	e	a	\20	G	a	r	d	e	n	\0

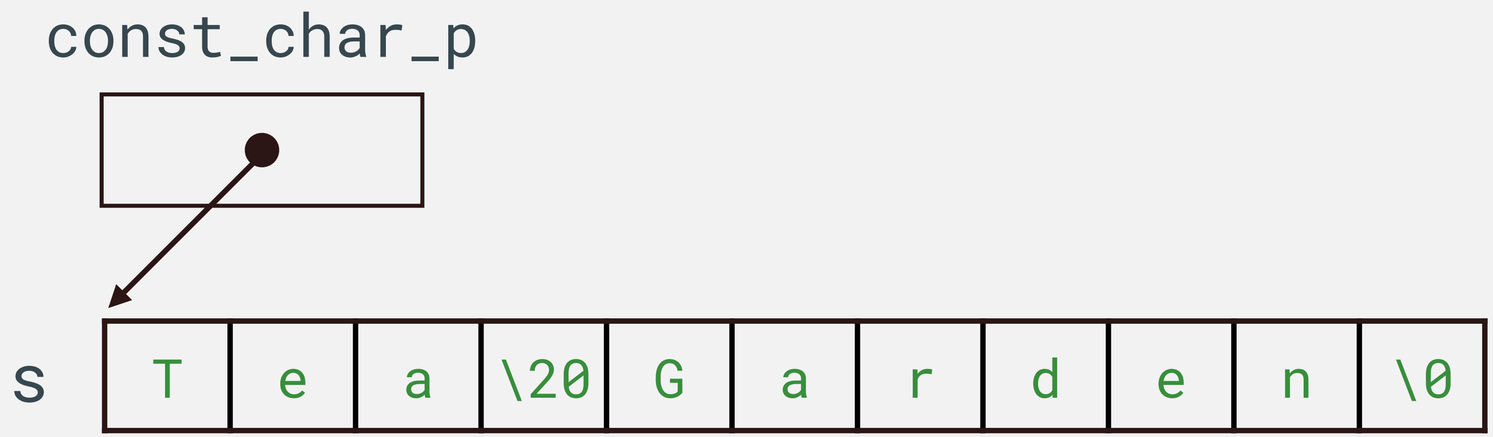
C String

```
char s[] = “Tea Garden”;
const char *const_char_p = s;
```

s	T	e	a	\20	G	a	r	d	e	n	\0

C String

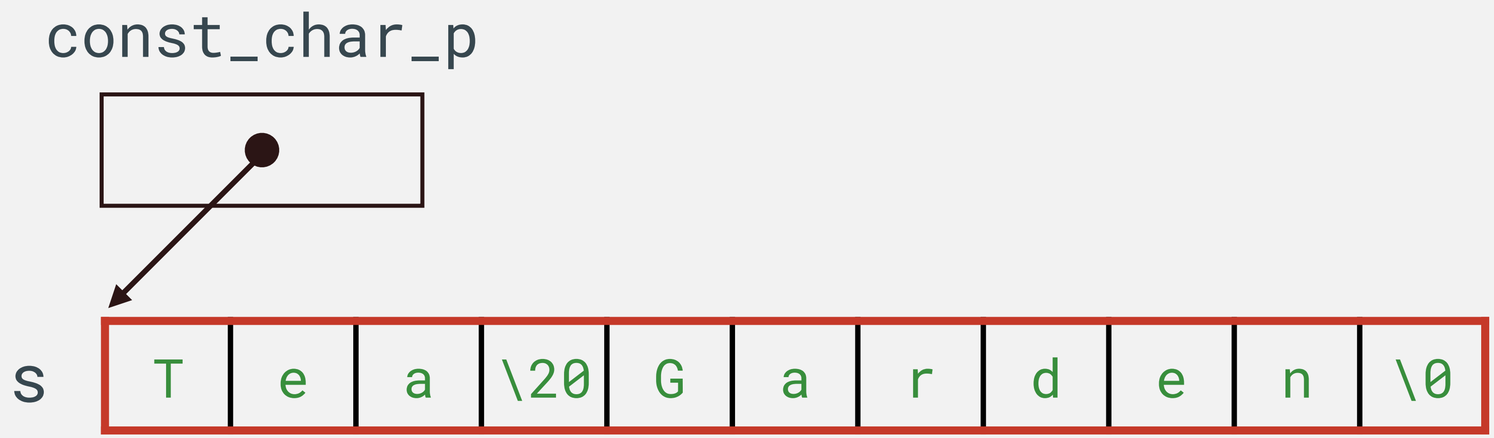
```
char s[] = “Tea Garden”;
const char *const_char_p = s;
```



C String

```
char s[] = “Tea Garden”;
const char *const_char_p = s;
```

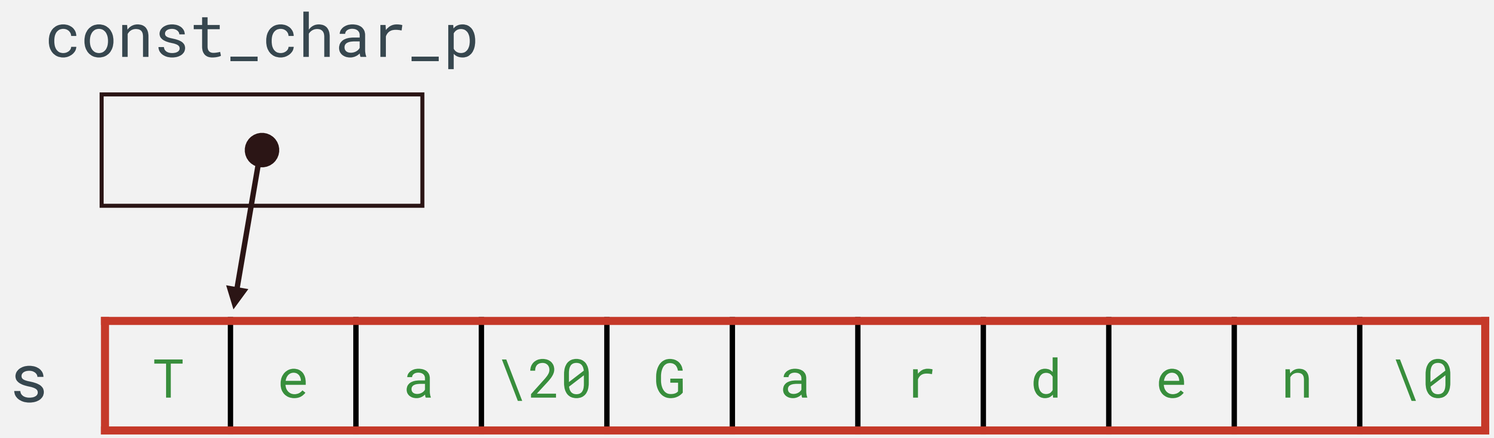
The **content** in the char array is immutable



C String

```
char s[] = “Tea Garden”;
const char *const_char_p = s;
```

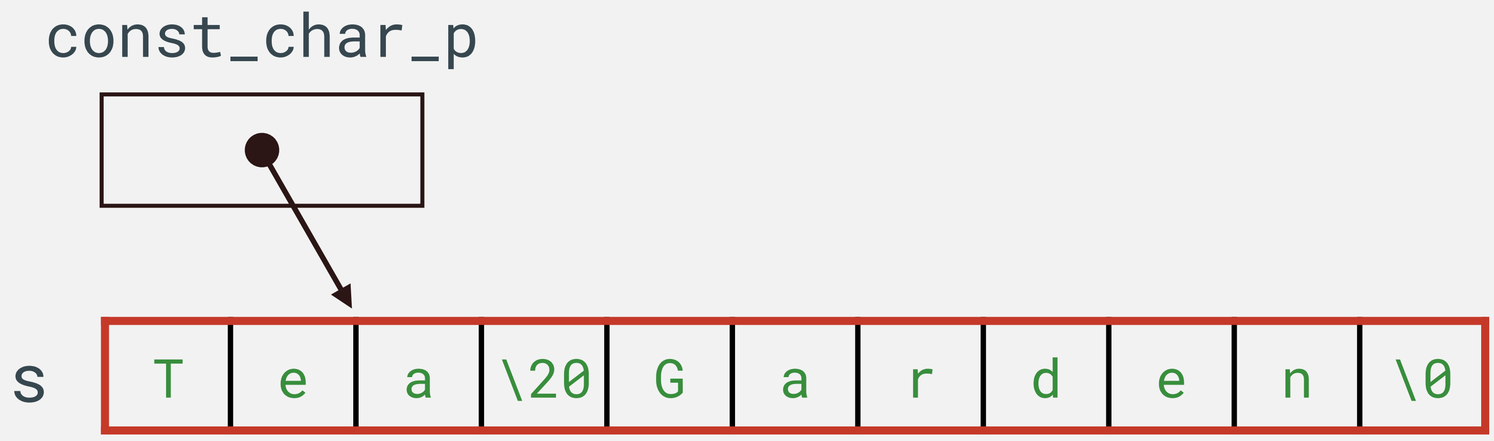
← The **content** in the char array is immutable



C String

```
char s[] = “Tea Garden”;
const char *const_char_p = s;
```

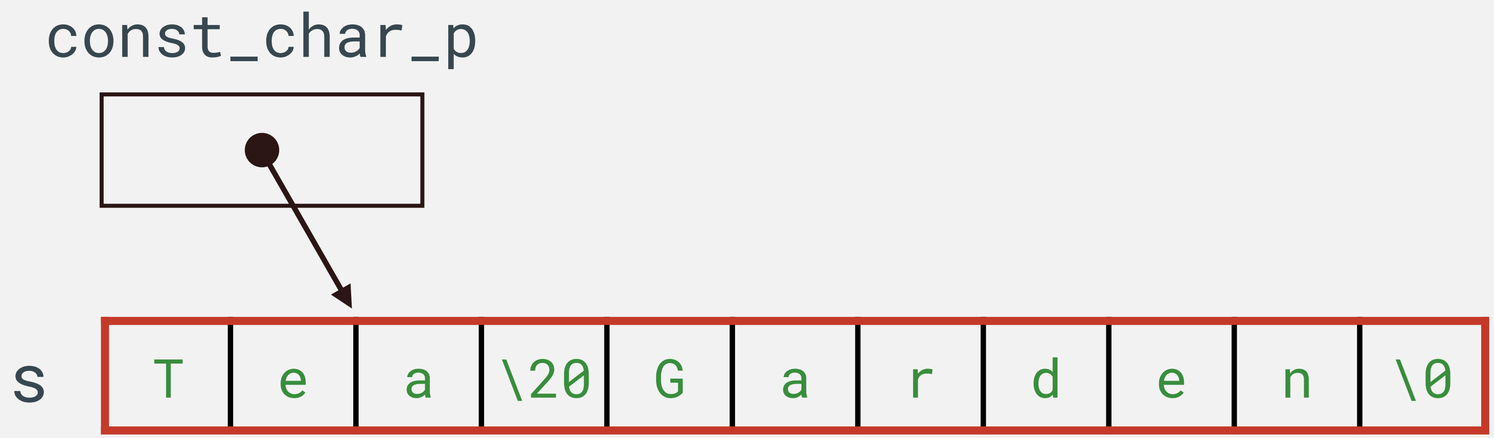
← The **content** in the char array is immutable



C String

```
char s[] = “Tea Garden”;
const char *const_char_p = s;
const_char_p++;
```

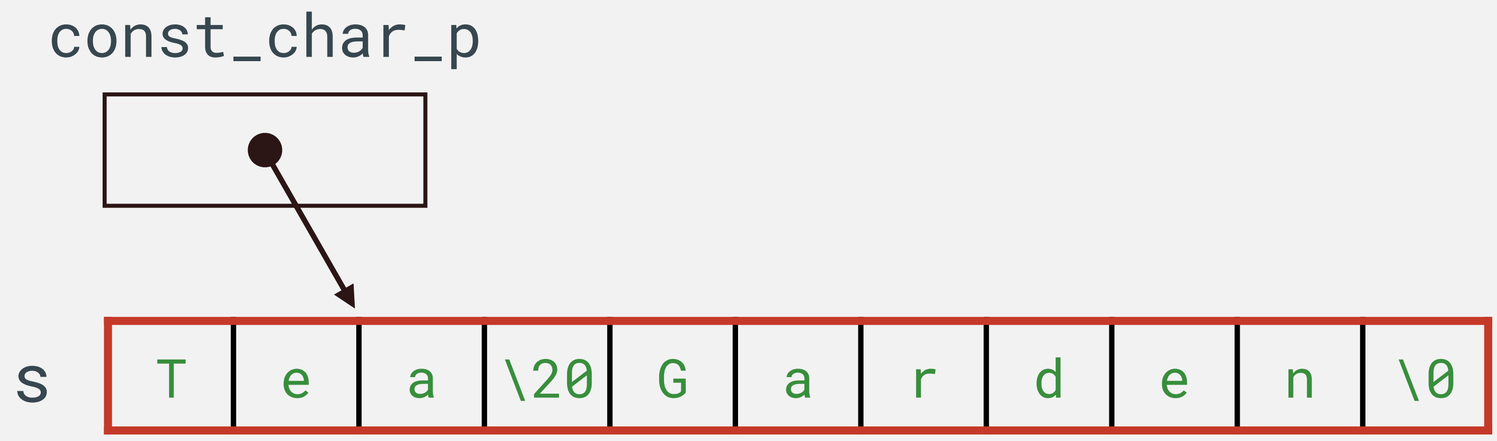
The **content** in the char array is immutable



C String

```
char s[] = “Tea Garden”;
const char *const_char_p = s;
const_char_p++;
const_char_p[2] = ’n’;
```

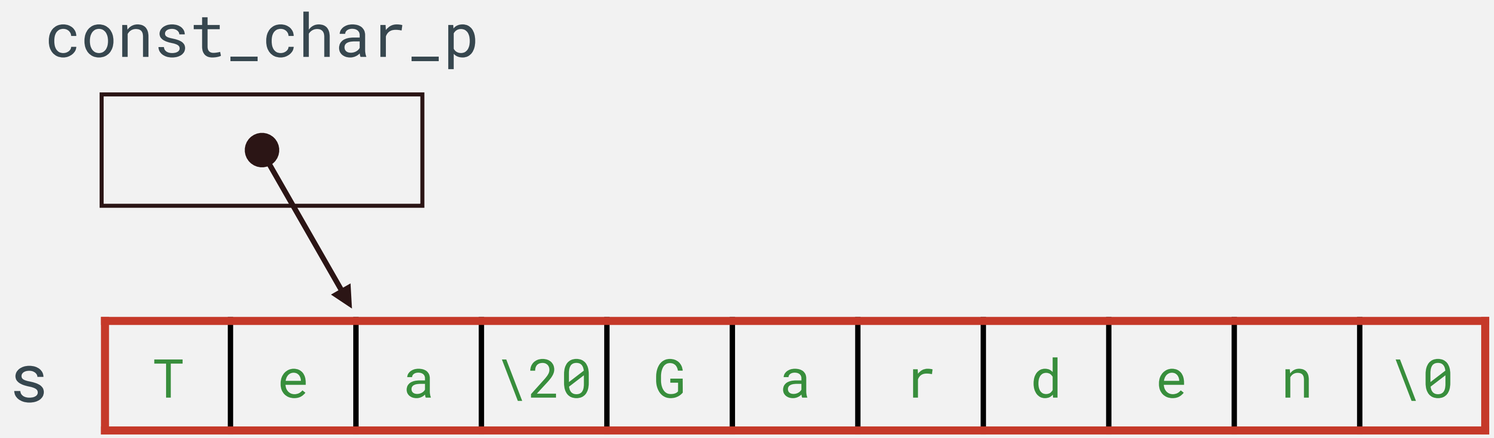
The **content** in the char array is immutable



C String

```
char s[] = “Tea Garden”;
const char *const_char_p = s;
const_char_p++;
const_char_p[2] = ’n’;
```

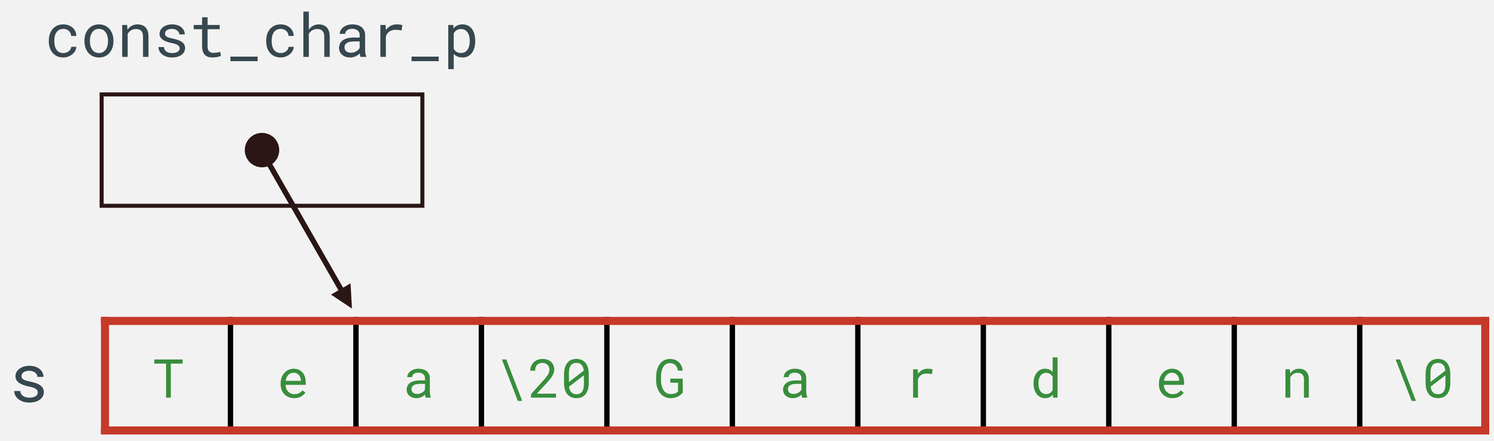
The **content** in the char array is immutable



C String

```
char s[] = “Tea Garden”;
const char *const_char_p = s;
```

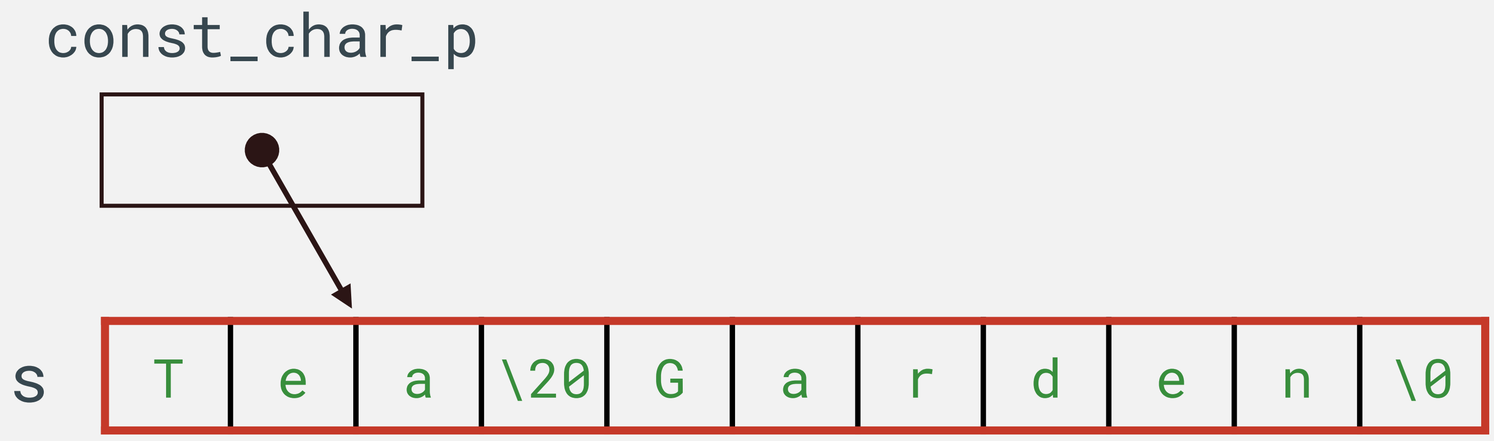
← The **content** in the char array is immutable



C String

```
char s[] = “Tea Garden”;
const char *const_char_p = s;
char const *char_const_p = s;
```

The **content** in the char array is immutable

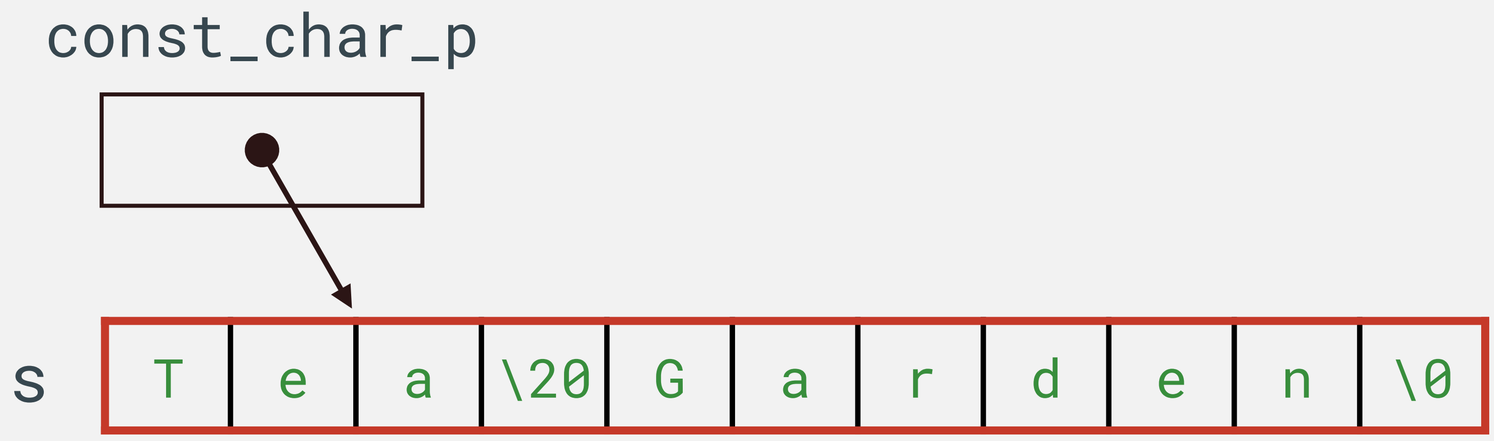


C String

```
char s[] = “Tea Garden”;
const char *const_char_p = s;
char const *char_const_p = s;
```

← The **content** in the char array is immutable

← Equivalent as above

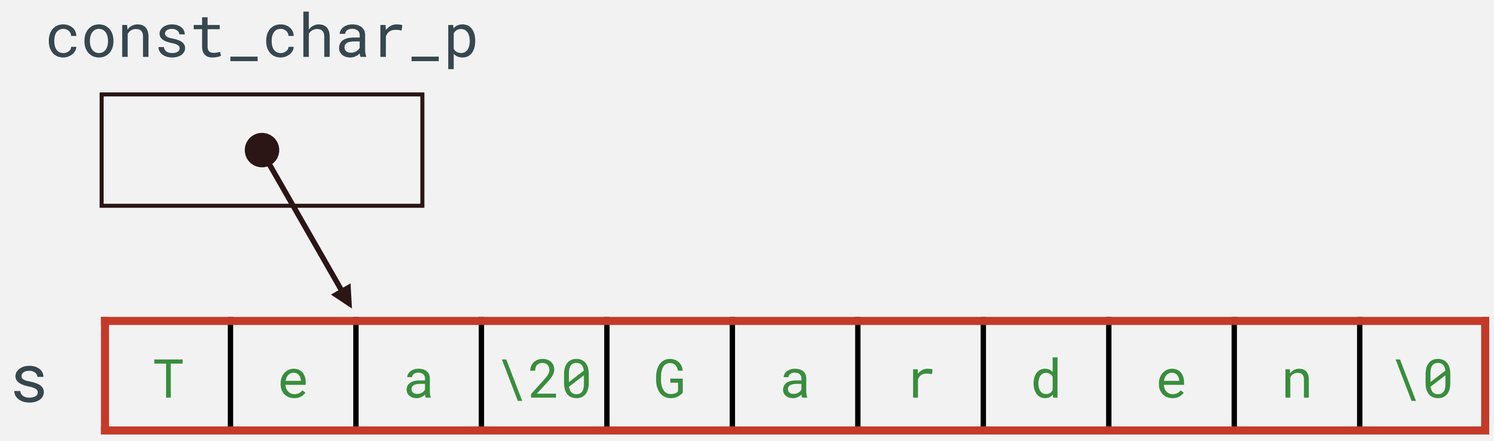


C String

```
char s[] = “Tea Garden”;
const char *const_char_p = s;
```

The **content** in the char array is immutable

~~`char const *char_const_p = s;`~~ ~~Equivalent as above~~



C String

```
char s[] = “Tea Garden”;
const char *const_char_p = s;
```

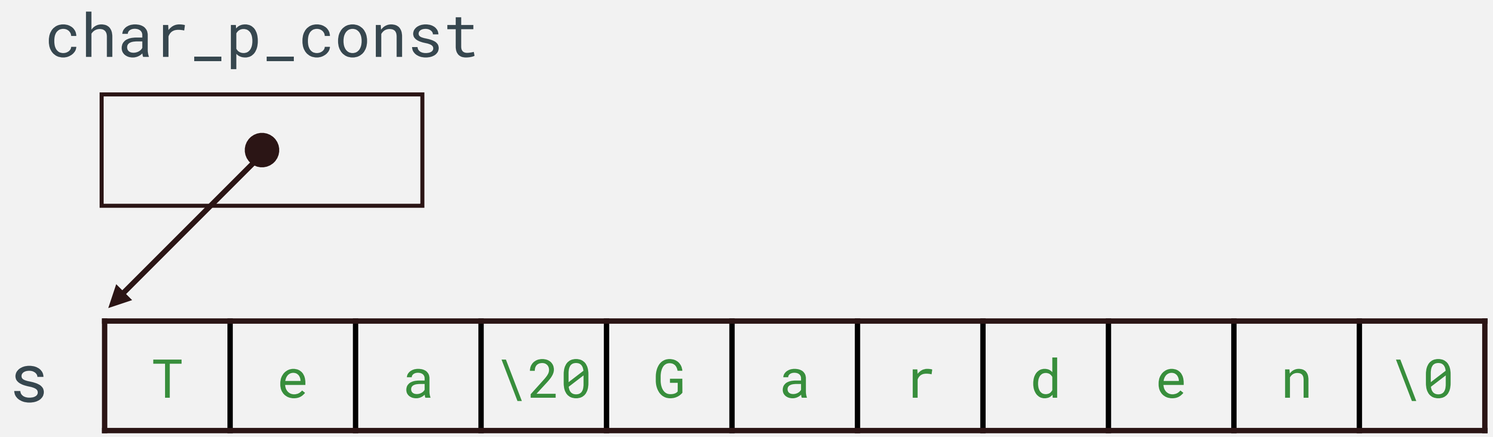
← The **content** in the char array is immutable

s	T	e	a	\20	G	a	r	d	e	n	\0

C String

```
char s[] = “Tea Garden”;
const char *const_char_p = s;
char * const char_p_const = s;
```

The **content** in the char array is immutable

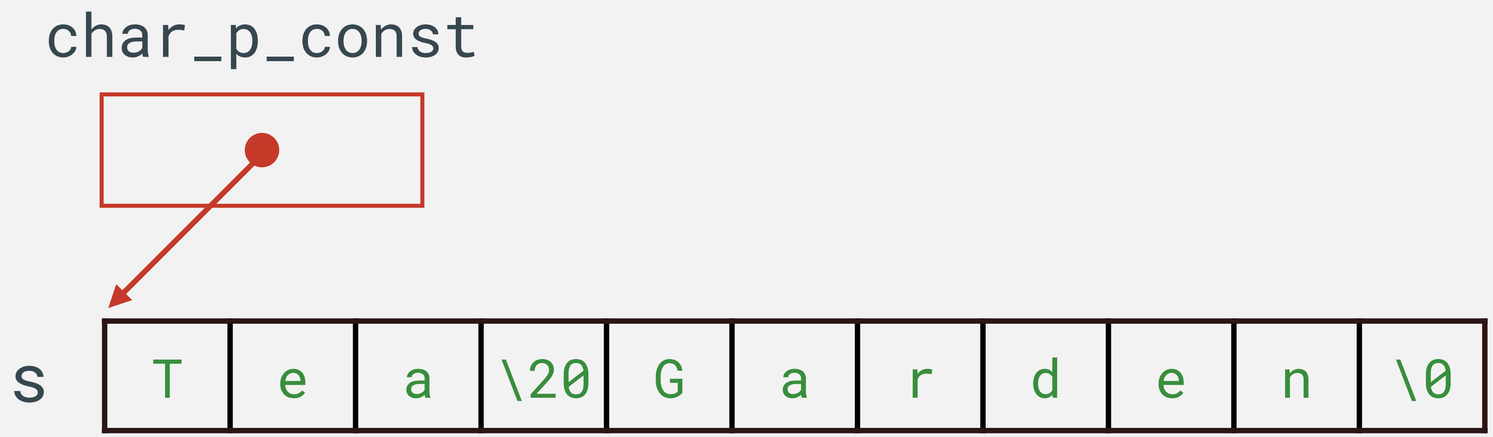


C String

```
char s[] = “Tea Garden”;
const char *const_char_p = s;
char * const char_p_const = s;
```

The **content** in the char array is immutable

The **pointer** itself is immutable

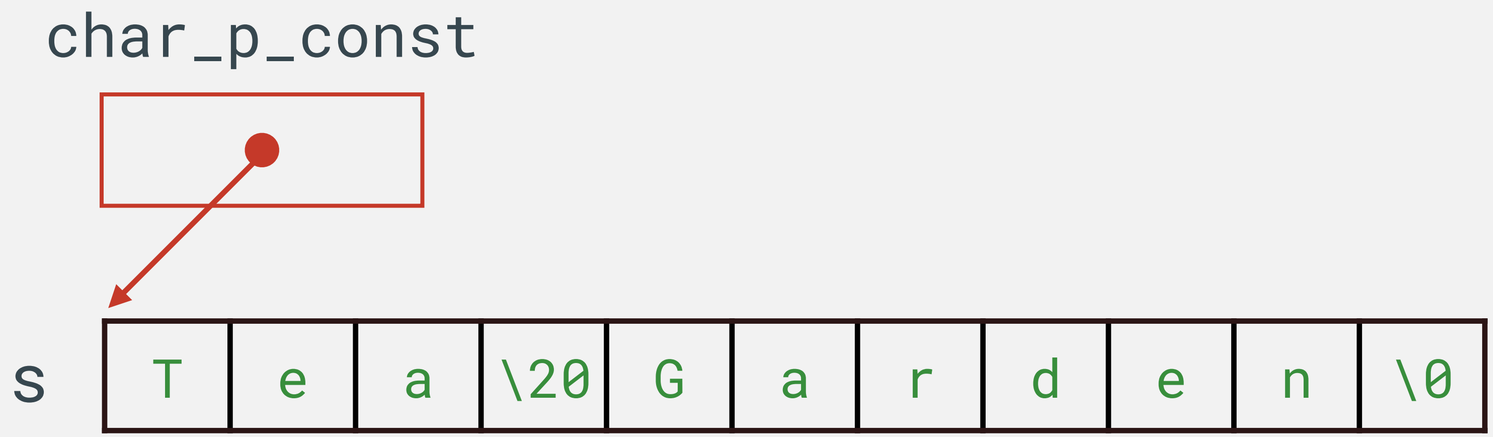


C String

```
char s[] = “Tea Garden”;
const char *const_char_p = s;
char * const char_p_const = s;
char_p_const++;
*(char_p_const + 2) = ’n’;
```

The **content** in the char array is immutable

The **pointer** itself is immutable

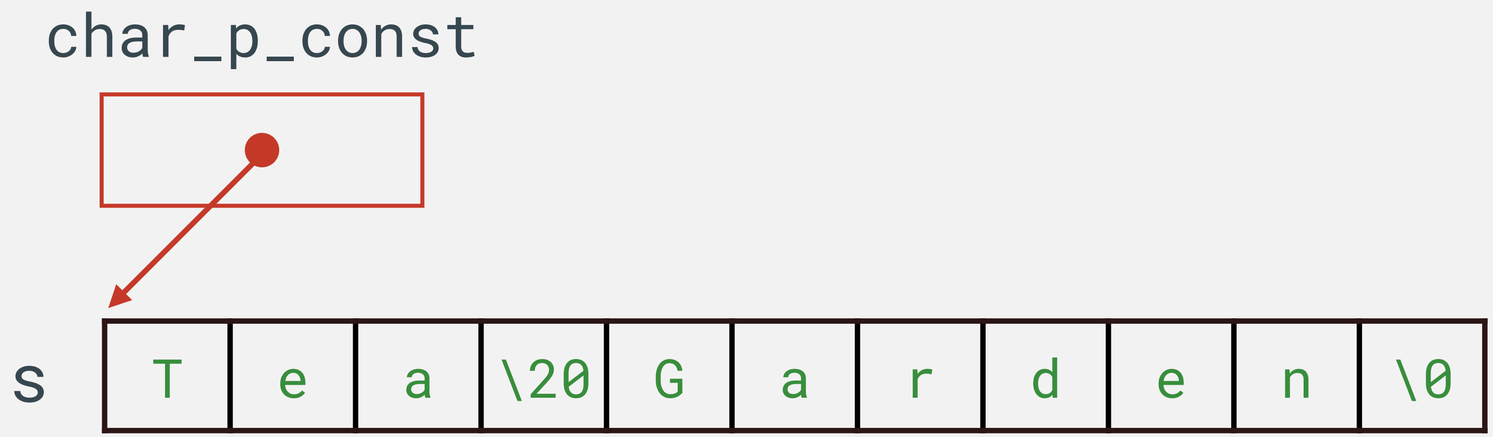


C String

```
char s[] = “Tea Garden”;
const char *const_char_p = s;
char * const char_p_const = s;
char_p_const++;
*(char_p_const + 2) = ’n’;
```

The **content** in the char array is immutable

The **pointer** itself is immutable

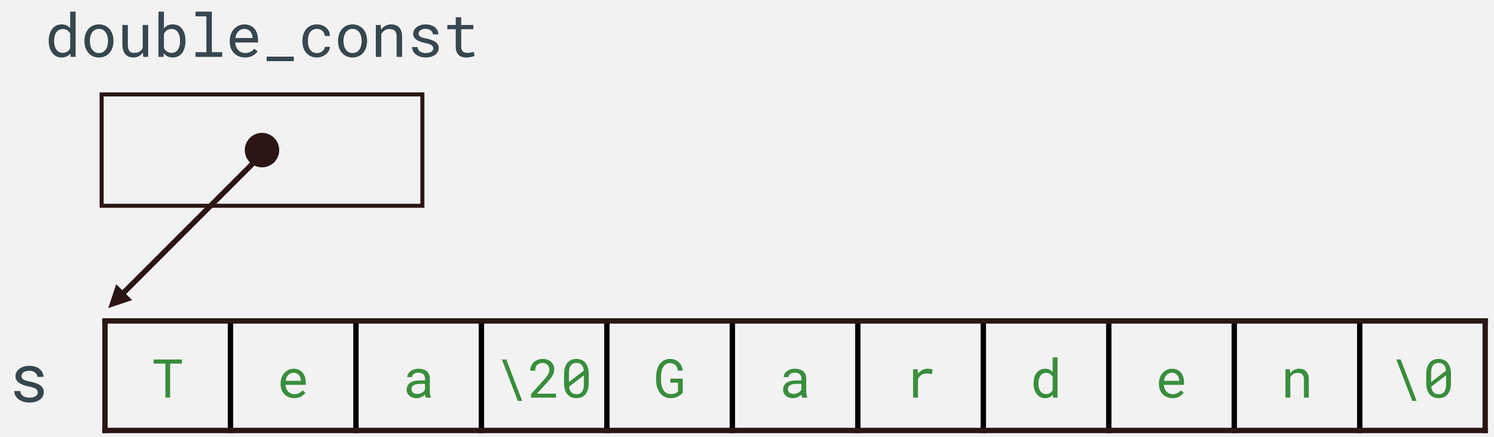


C String

```
char s[] = “Tea Garden”;
const char *const_char_p = s;
char * const char_p_const = s;
const char * const double_const = s;
```

The **content** in the char array is immutable

The **pointer** itself is immutable

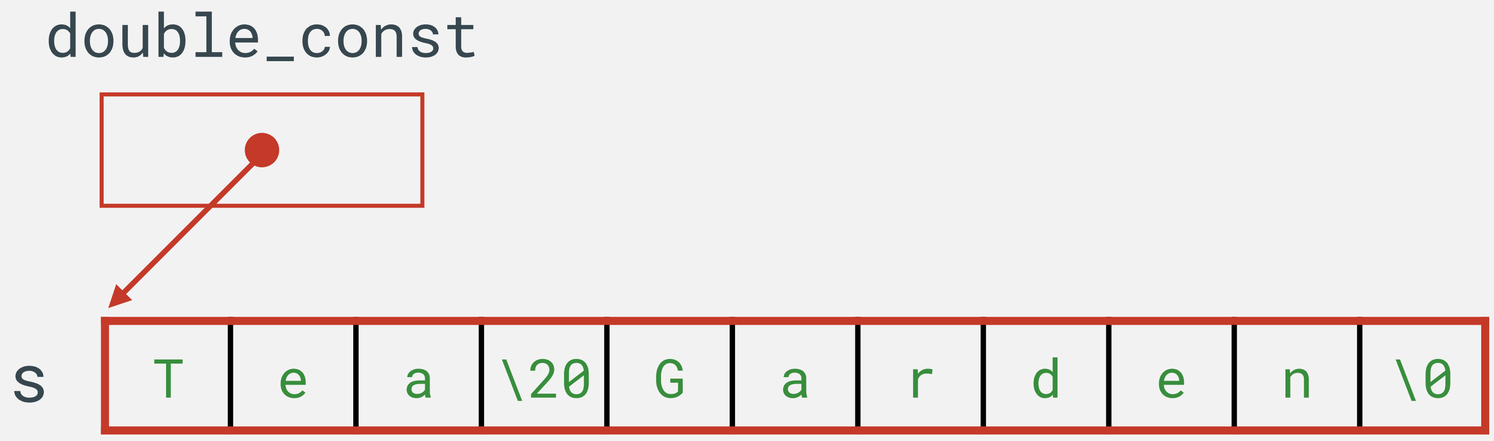


C String

```
char s[] = “Tea Garden”;
const char *const_char_p = s;
char * const char_p_const = s;
const char * const double_const = s;
```

The **content** in the char array is immutable

The **pointer** itself is immutable



String Copy

```
char s1[20] = “Tea Garden”;
char s2[20];
```

String Copy

```
char s1[20] = “Tea Garden”;
char s2[20];
s2 = s1;
```

String Copy

```
char s1[20] = “Tea Garden”;
char s2[20];
s2 = s1;
```

← Copies the pointer only

String Copy

```
char s1[20] = “Tea Garden”;
char s2[20];
s2 = s1;
*s2 = *s1;
```

← Copies the pointer only

String Copy

```
char s1[20] = “Tea Garden”;
char s2[20];
s2 = s1;
*s2 = *s1;
```

← Copies the pointer only

← Copies the first element only

String Copy

```
char s1[20] = “Tea Garden”;
char s2[20];
s2 = s1;
*s2 = *s1;
```

← Copies the pointer only

← Copies the first element only

```
for (int i = 0; i < 20; i++)
  s2[i] = s1[i];
```

String Copy

```
char s1[20] = “Tea Garden”;
char s2[20];
s2 = s1;
*s2 = *s1;
```

← Copies the pointer only

← Copies the first element only

```
int i = 0;
while (s1[i] != ‘\0’) {
  s2[i] = s1[i];
  i++;
}
```

String Copy

```
char s1[20] = “Tea Garden”;
char s2[20];
s2 = s1;
*s2 = *s1;
```

← Copies the pointer only

← Copies the first element only

```
int i = 0;
while (s1[i] != ‘\0’) {
  s2[i] = s1[i];
  i++;
}
```

```
while (*s2_copy++ = *s1_copy++)
  ;
```

String Copy

```
char s1[20] = “Tea Garden”;
char s2[20];
s2 = s1;
*s2 = *s1;
```

← Copies the pointer only

← Copies the first element only

```
strcpy(s2, s1); // #include <string.h>
```

String Copy

```
char s1[20] = “Tea Garden”;
char s2[20];
s2 = s1;
*s2 = *s1;
```

← Copies the pointer only

← Copies the first element only

```
strcpy(s2, s1); // #include <string.h>
```

Make sure s2 has enough space

String Compare

```
char s1[] = “Tea Garden”;
char s2[] = {’T’, ‘e’, ‘a’, ‘ ’, ‘G’, ‘a’, ‘r’, ‘d’, ‘e’, ’n’};


if (s1 == s2)
  printf(“Same String\n”);
```

String Compare

```
char s1[] = “Tea Garden”;
char s2[] = {’T’, ‘e’, ‘a’, ‘ ’, ‘G’, ‘a’, ‘r’, ‘d’, ‘e’, ’n’};


if (s1 == s2)
  printf(“Same String\n”);
```

← Compares the pointer (array address) only

String Compare

```
char s1[] = “Tea Garden”;
char s2[] = {’T’, ‘e’, ‘a’, ‘ ’, ‘G’, ‘a’, ‘r’, ‘d’, ‘e’, ’n’};

// #include <string.h>
if (strcmp(s1, s2) == 0)
  printf(“Same String\n”);
```

String Compare

```
char s1[] = “Tea Garden”;
char s2[] = {’T’, ‘e’, ‘a’, ‘ ’, ‘G’, ‘a’, ‘r’, ‘d’, ‘e’, ’n’};

// #include <string.h>
if (strcmp(s1, s2) == 0)
  printf(“Same String\n”);
else if (strcmp(s1, s2) < 0)
  printf(“s1 is smaller than s2\n”);
else
  printf(“s1 is greater than s2\n”);
```

String Length

```
char s1[] = “Tea Garden”;
printf(“s1 length = %u”, strlen(s1));
```

String Length

```
char s1[] = “Tea Garden”;
printf(“s1 length = %u”, strlen(s1));
```

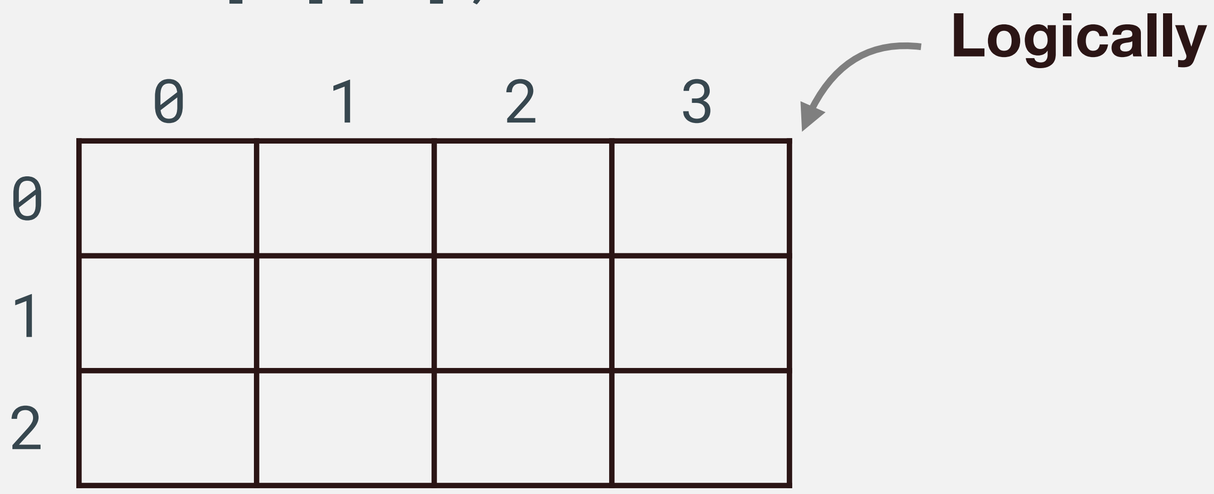
‘\0’ **excluded**

Multi-Dimensional Array

```
int a[3][4];
```

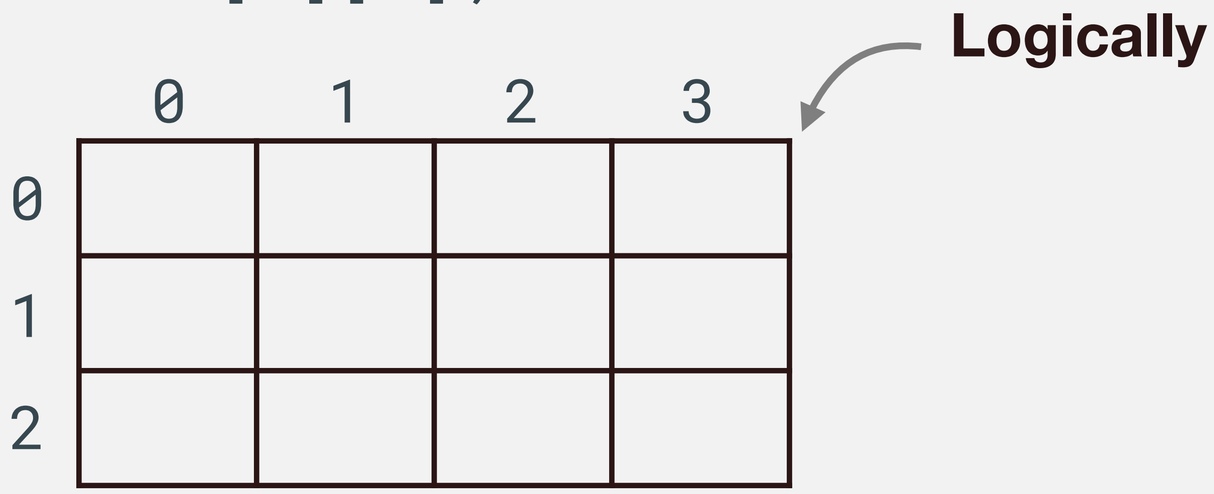
Multi-Dimensional Array

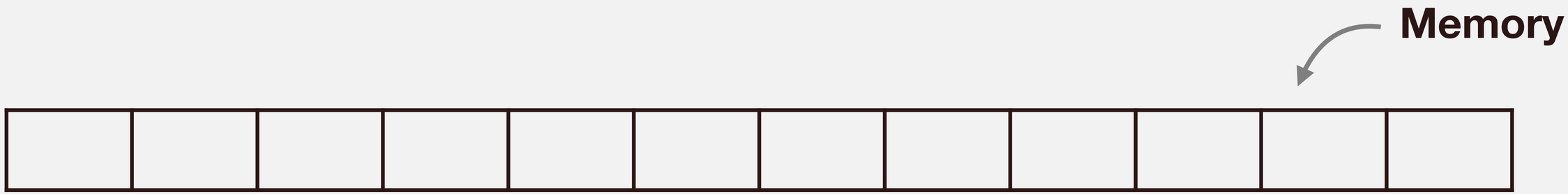
```
int a[3][4];
```



Multi-Dimensional Array

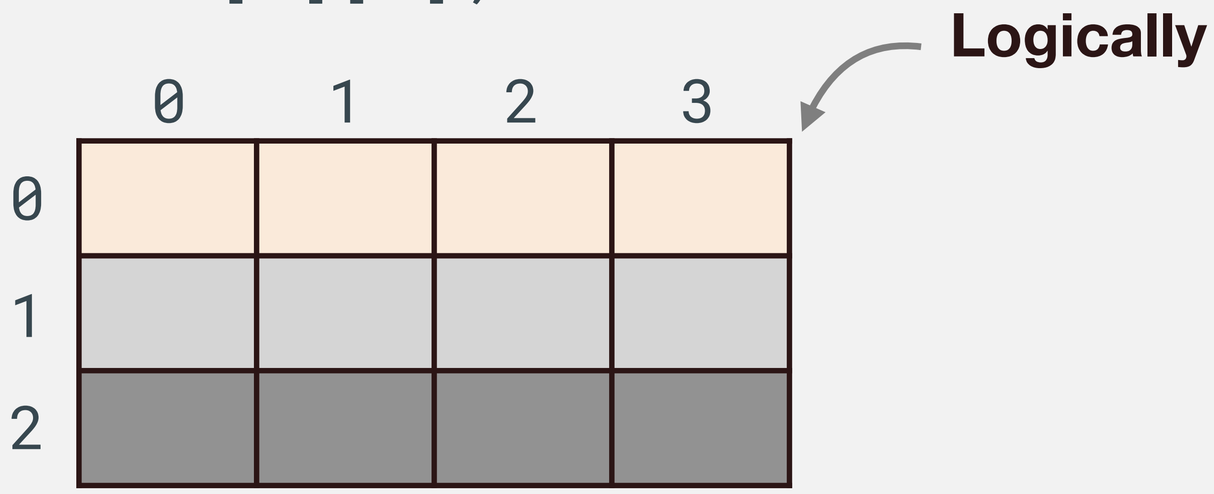
```
int a[3][4];
```

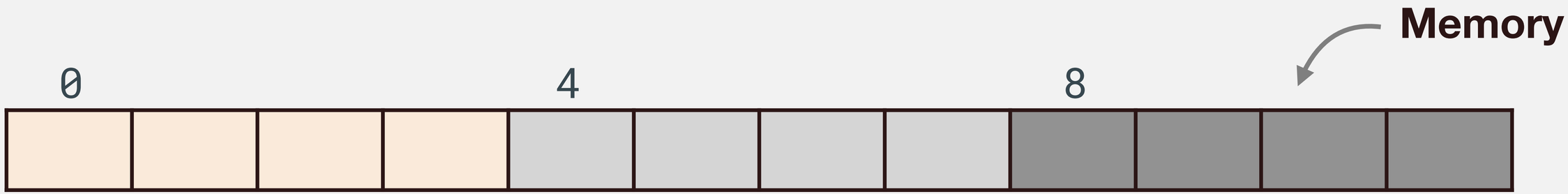




Multi-Dimensional Array

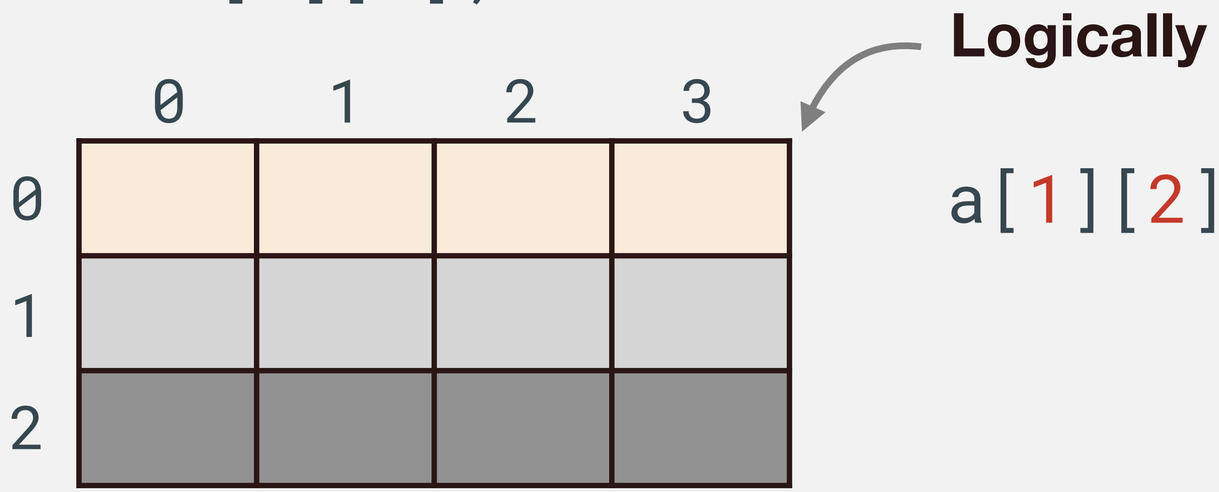
```
int a[3][4];
```



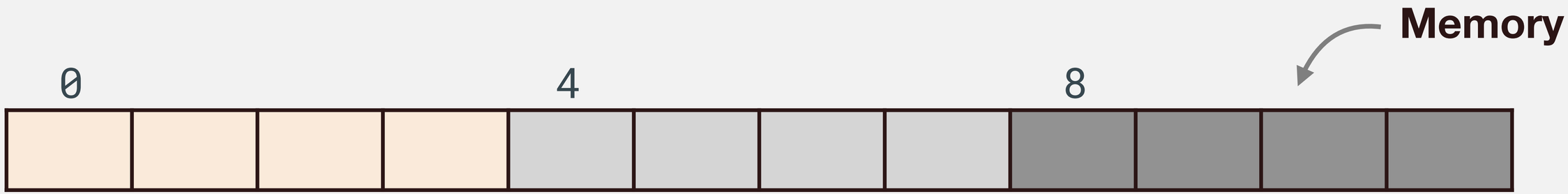


Multi-Dimensional Array

```
int a[3][4];
```

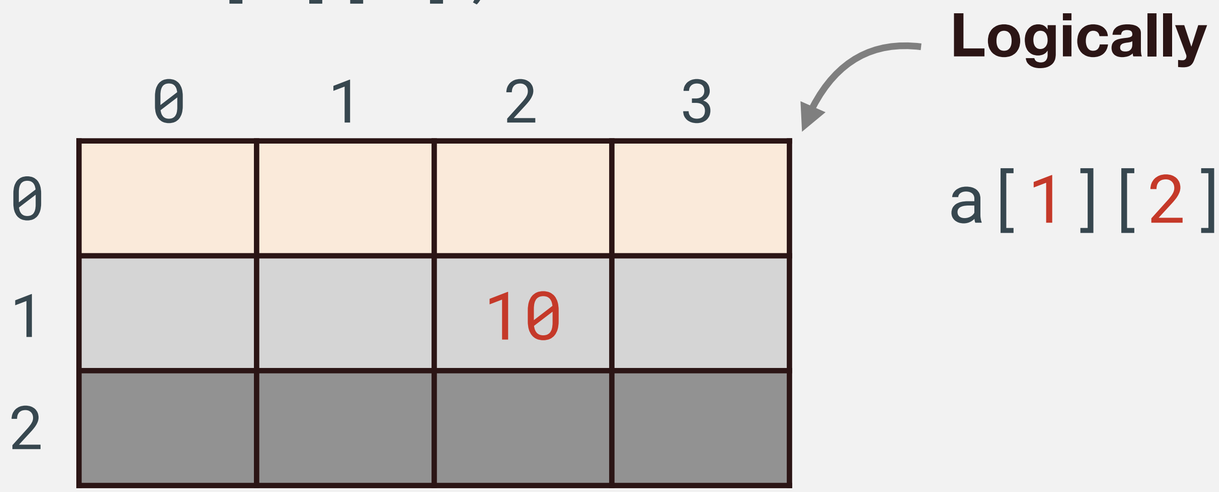



```
a[1][2] = 10;
```

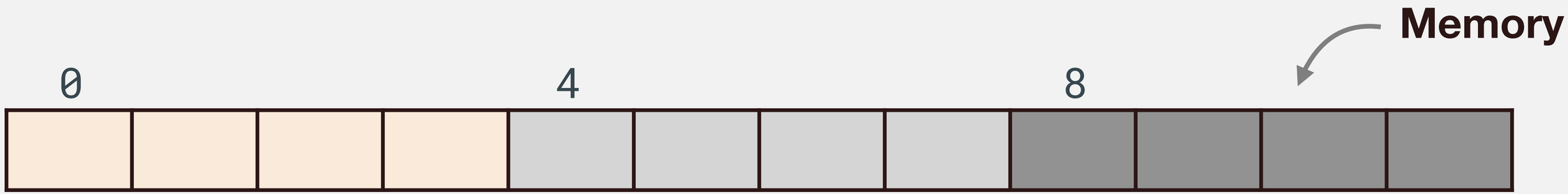


Multi-Dimensional Array

```
int a[3][4];
```

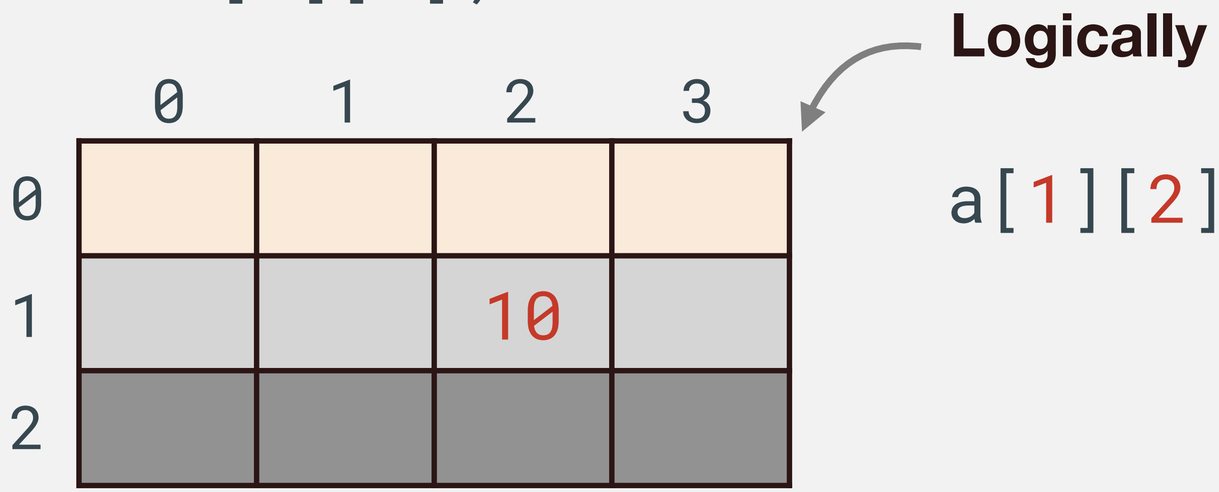



```
a[1][2] = 10;
```



Multi-Dimensional Array

```
int a[3][4];
```




```
a[1][2] = 10;
```

Multi-Dimensional Array

```
int a[3][4];
```

Multi-Dimensional Array

```
int a[3][4];
a[1]
```

Multi-Dimensional Array

```
int a[3][4];
a[1]
```

Multi-Dimensional Array

```
int a[3][4];
a[1] ◄---------- int *
```

Multi-Dimensional Array

```
int a[3][4];
a[1] ◄---------- int *     a[1][2] = 10;
```

Multi-Dimensional Array

```
int a[3][4];
a[1] ◄---------- int *      a[1][2] = 10; ≡ *(*a + 6) = 10;
```

Multi-Dimensional Array

```
int a[3][4];
a[1] ◂---------- int *
a
```

Multi-Dimensional Array

```
int a[3][4];
a[1] ◄---------- int *
a
```

Multi-Dimensional Array

```
int a[3][4];
a[1] ←---------- int *
a ←--------------- int (*)[4]
```

Multi-Dimensional Array

```
int a[3][4];
a[1] <---------- int *
a <---------------- int (*)[4]
```

```
int (*a)[4]; vs int *a[4];
```

Multi-Dimensional Array

```
int a[3][4];
a[1] ←---------- int *
a ←--------------- int (*)[4]
```

A pointer to a 4-int array

```
int (*a)[4]; vs int *a[4];
```

Multi-Dimensional Array

```
int a[3][4];
a[1] ←---------- int *
a ←--------------- int (*)[4]
```

A pointer to a 4-int array

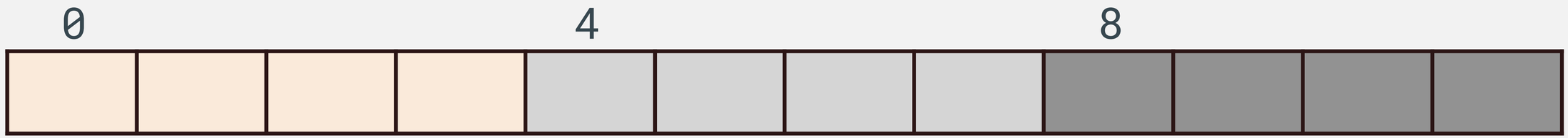
```
int (*a)[4]; vs int *a[4];
```

An array of 4 int pointers

Passing 2D-Array to Function

```c
#define R 3
#define C 4

int a[R][C];
...
int sum = ElementSum(a);
```

```c
int ElementSum(???) {
  int sum = 0;
  for (int i = 0; i < R; i++) {
    for (int j = 0; j < C; j++)
      sum += a[i][j];
  }
  return sum;
}
```



Passing 2D-Array to Function

```
int ElementSum(???) {
  int sum = 0;
  for (int i = 0; i < R; i++) {
    for (int j = 0; j < C; j++)
      sum += a[i][j];
  }
  return sum;
}
```

Passing 2D-Array to Function

```
int ElementSum(int a[R][C]) {
  int sum = 0;
  for (int i = 0; i < R; i++) {
    for (int j = 0; j < C; j++)
      sum += a[i][j];
  }
  return sum;
}
```

Passing 2D-Array to Function

```
int ElementSum(int a[R][C]) {
  int sum = 0;
  for (int i = 0; i < R; i++) {
    for (int j = 0; j < C; j++)
      sum += a[i][j];
  }
  return sum;
}
```

Passing 2D-Array to Function

```
int ElementSum(int a[R][C]) {
  int sum = 0;
  for (int i = 0; i < R; i++) {
    for (int j = 0; j < C; j++)
      sum += a[i][j];
  }
  return sum;
}
```

Only the pointer is copied, not the array!

Passing 2D-Array to Function

```
int ElementSum(int a[][C]) {
  int sum = 0;
  for (int i = 0; i < R; i++) {
    for (int j = 0; j < C; j++)
      sum += a[i][j];
  }
  return sum;
}
```

Passing 2D-Array to Function

```
int ElementSum(int a[][C]) {
  int sum = 0;
  for (int i = 0; i < R; i++) {
    for (int j = 0; j < C; j++)
      sum += a[i][j];
  }
  return sum;
}
```

Passing 2D-Array to Function

```
int ElementSum(int a[][C]) {
  int sum = 0;
  for (int i = 0; i < R; i++) {
    for (int j = 0; j < C; j++)
      sum += a[i][j];
  }
  return sum;
}
```

Passing 2D-Array to Function

```
int ElementSum(int a[][C]) {
  int sum = 0;
  for (int i = 0; i < R; i++) {
    for (int j = 0; j < C; j++)
      sum += a[i][j];
  }
  return sum;
}
```

Read Address

0x3E50 + 4 * (i * C + j)

0x3E50

Passing 2D-Array to Function

```
int ElementSum(int (*a)[C]) {
  int sum = 0;
  for (int i = 0; i < R; i++) {
    for (int j = 0; j < C; j++)
      sum += a[i][j];
  }
  return sum;
}
```

Passing 2D-Array to Function

```
int ElementSum(int (*a)[C]) {
  int sum = 0;
  for (int i = 0; i < R; i++) {
    for (int j = 0; j < C; j++)
      sum += a[i][j];
  }
  return sum;
}
```

Passing 2D-Array to Function

```
int ElementSum(int *a) {
  int sum = 0;
  for (int i = 0; i < R; i++) {
    for (int j = 0; j < C; j++)
      sum += a[i][j];
  }
  return sum;
}
```

Passing 2D-Array to Function

```
int ElementSum(int *a) {
  int sum = 0;
  for (int i = 0; i < R; i++) {
    for (int j = 0; j < C; j++)
      sum += a[i][j];
  }
  return sum;
}
```

Passing 2D-Array to Function

```
int ElementSum(int *a) {
  int sum = 0;
  for (int i = 0; i < R; i++) {
    for (int j = 0; j < C; j++)
      sum += a[i * C + j];
  }
  return sum;
}
```

Passing 2D-Array to Function

```
int ElementSum(int *a) {
  int sum = 0;
  for (int i = 0; i < R; i++) {
    for (int j = 0; j < C; j++)
      sum += a[i * C + j];
  }
  return sum;
}
```

Passing 2D-Array to Function

```
int ElementSum(int *a) {
  int sum = 0;
  for (int i = 0; i < R; i++) {
    for (int j = 0; j < C; j++)
      sum += a[i * C + j];
  }
  return sum;
}
```

```
int a[R][C];
...
int sum = ElementSum((int *)a);
```

Passing 2D-Array to Function

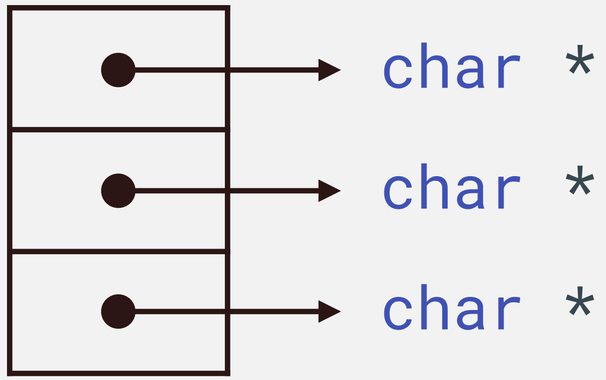
```
int ElementSum(int **a) {
  int sum = 0;
  for (int i = 0; i < R; i++) {
    for (int j = 0; j < C; j++)
      sum += a[i][j];
  }
  return sum;
}
```

Dynamically-Allocated 2D-Array

```
char *a[3];
```

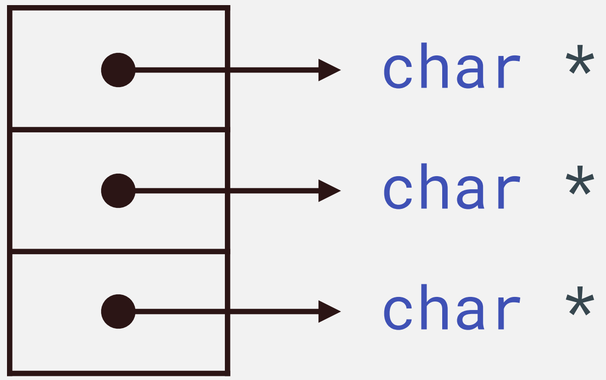
Dynamically-Allocated 2D-Array

```
char *a[3];
```



Dynamically-Allocated 2D-Array

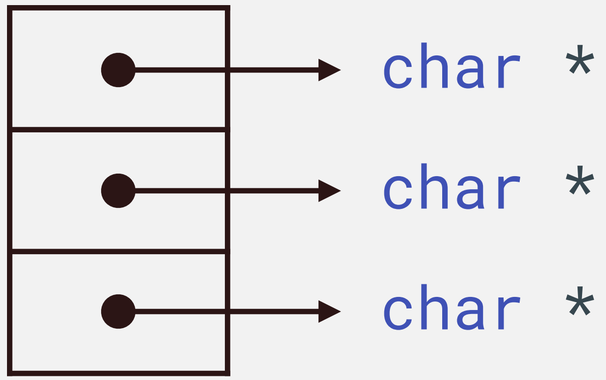
```
char *a[3]; // a has type (char **)
```



Dynamically-Allocated 2D-Array

```
char *a[3]; // a has type (char **)
```

a is a pointer pointing to (`char *`)

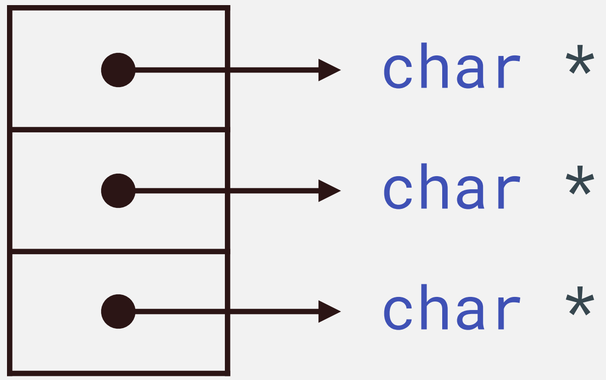


Dynamically-Allocated 2D-Array

```
char *a[3]; // a has type (char **)
```

a is a pointer pointing to (`char *`)

pointer array

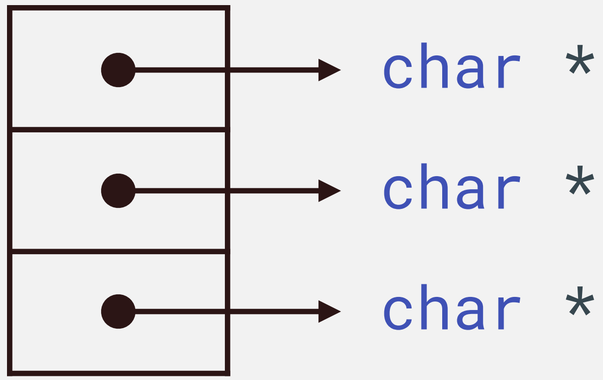


Dynamically-Allocated 2D-Array

```
char *a[3]; // a has type (char **)
```

a is a pointer pointing to (char *)

pointer array




```
char **a = (char **)malloc(3 * sizeof(char *));
```

Multi-Dimensional Array vs. Pointer Array

```
char a[3][10] = {“Shui”, “zai”, “juan???”};
```

```
char *a[3] = {“Shui”, “zai”, “juan???”};
```

Multi-Dimensional Array vs. Pointer Array

```
char a[3][10] = {“Shui”, “zai”, “juan???”};
```

```
char *a[3] = {“Shui”, “zai”, “juan???”};
```

Multi-Dimensional Array vs. Pointer Array

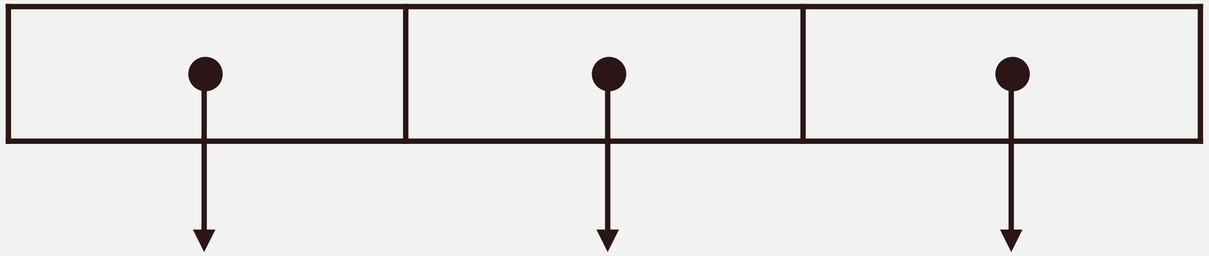
```
char a[3][10] = {“Shui”, “zai”, “juan???”};
```


```
char *a[3] = {“Shui”, “zai”, “juan???”};
```

Multi-Dimensional Array vs. Pointer Array

```
char a[3][10] = {“Shui”, “zai”, “juan???”};
```

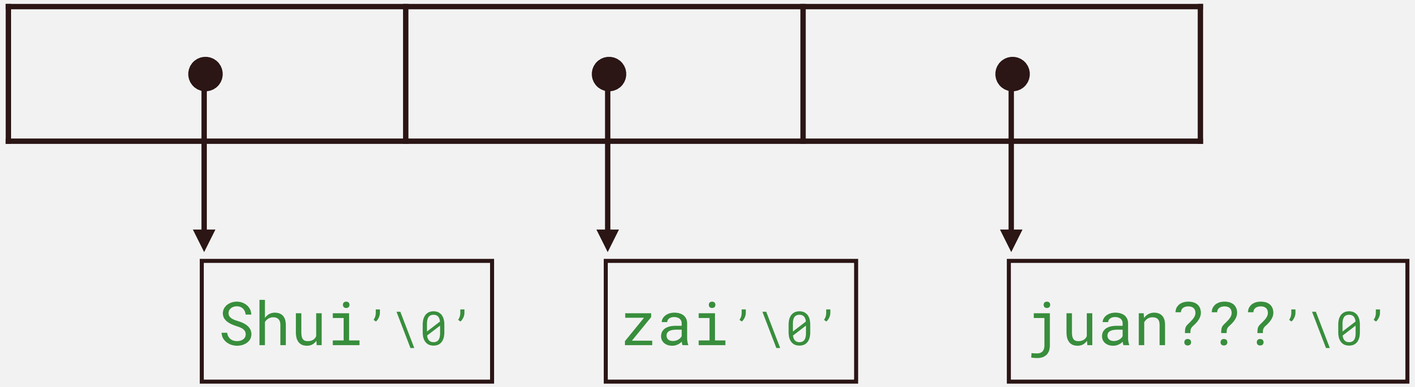

```
char *a[3] = {“Shui”, “zai”, “juan???”};
```



Multi-Dimensional Array vs. Pointer Array

```
char a[3][10] = {“Shui”, “zai”, “juan???”};
```


```
char *a[3] = {“Shui”, “zai”, “juan???”};
```



Passing 2D-Array to Function — Revisit

```c
#define R 3
#define C 4

int a[R][C];
...
int sum = ElementSum((int **)a);
```

```c
int ElementSum(int **a) {
  int sum = 0;
  for (int i = 0; i < R; i++) {
    for (int j = 0; j < C; j++)
      sum += a[i][j];
  }
  return sum;
}
```

Passing 2D-Array to Function — Revisit

```c
#define R 3
#define C 4

int a[R][C];
...
int sum = ElementSum((int **)a);
```

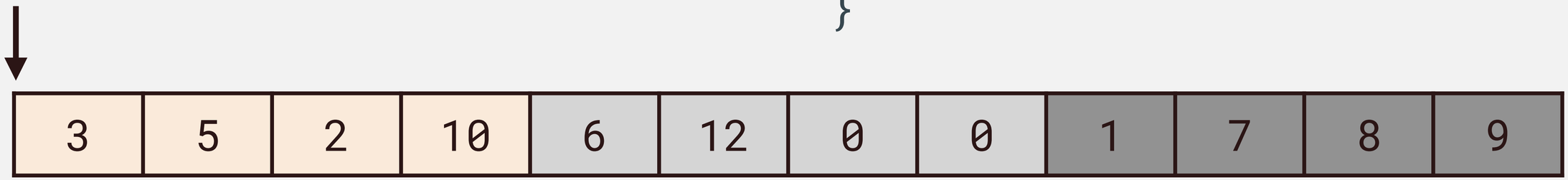
```c
int ElementSum(int **a) {
  int sum = 0;
  for (int i = 0; i < R; i++) {
    for (int j = 0; j < C; j++)
      sum += a[i][j];
  }
  return sum;
}
```

Passing 2D-Array to Function — Revisit

```c
#define R 3
#define C 4

int a[R][C];
...
int sum = ElementSum((int **)a);
```

```c
int ElementSum(int **a) {
  int sum = 0;
  for (int i = 0; i < R; i++) {
    for (int j = 0; j < C; j++)
      sum += a[i][j];
  }
  return sum;
}
```

(int **)a



Passing 2D-Array to Function — Revisit

```c
#define R 3
#define C 4

int a[R][C];
...
int sum = ElementSum((int **)a);
```

```c
int ElementSum(int **a) {
  int sum = 0;
  for (int i = 0; i < R; i++) {
    for (int j = 0; j < C; j++)
      sum += a[i][j];
  }
  return sum;
}
```

(int **)a

Passing 2D-Array to Function — Revisit

```c
#define R 3
#define C 4

int a[R][C];
...
int sum = ElementSum((int **)a);
```

```c
int ElementSum(int **a) {
  int sum = 0;
  for (int i = 0; i < R; i++) {
    for (int j = 0; j < C; j++)
      sum += a[i][j];
  }
  return sum;
}
```

(int **)a

Passing 2D-Array to Function — Revisit

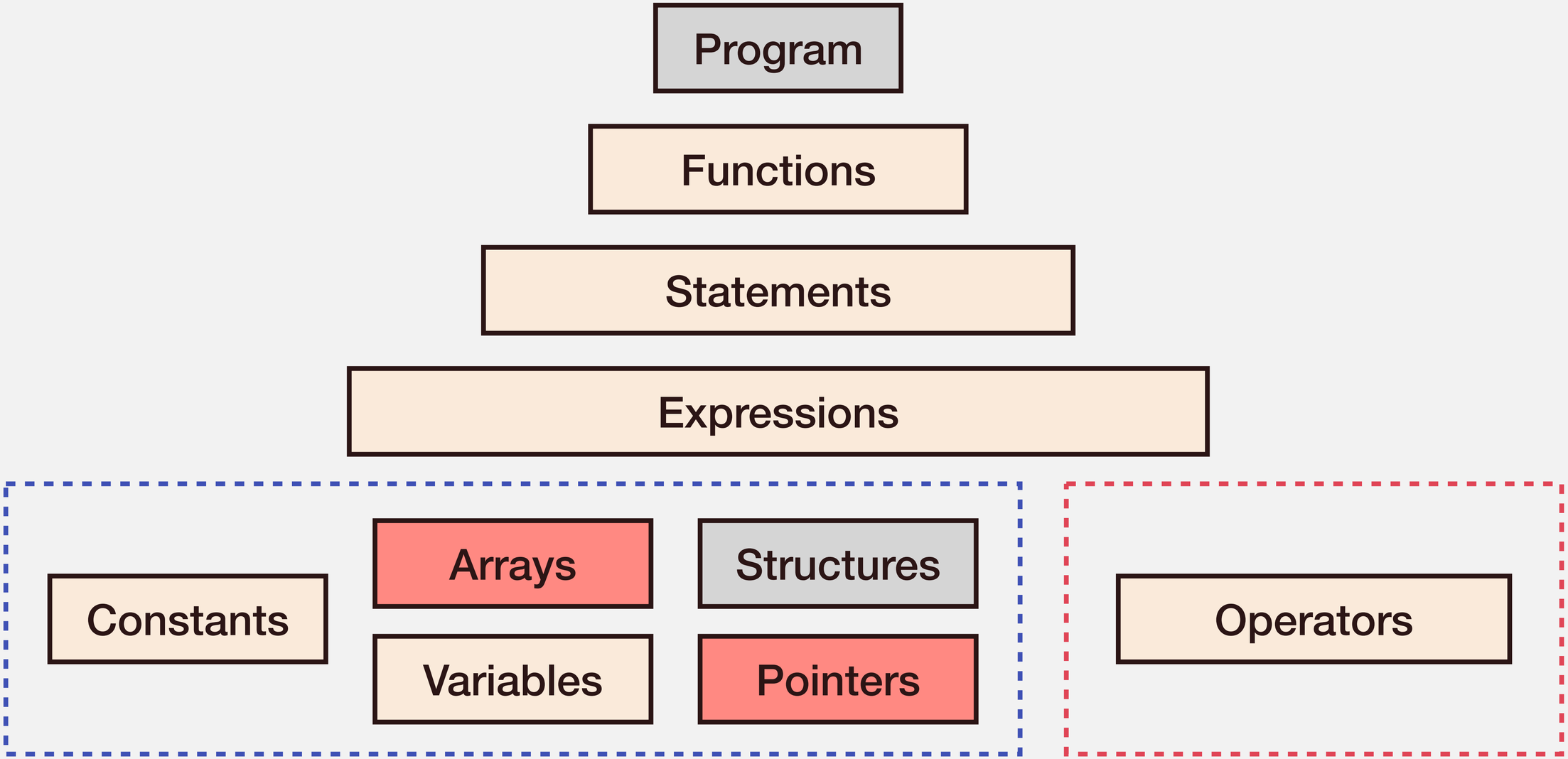
```c
#define R 3
#define C 4

int a[R][C];
...
int sum = ElementSum((int **)a);
```

```c
int ElementSum(int **a) {
  int sum = 0;
  for (int i = 0; i < R; i++) {
    for (int j = 0; j < C; j++)
      sum += a[i][j];
  }
  return sum;
}
```

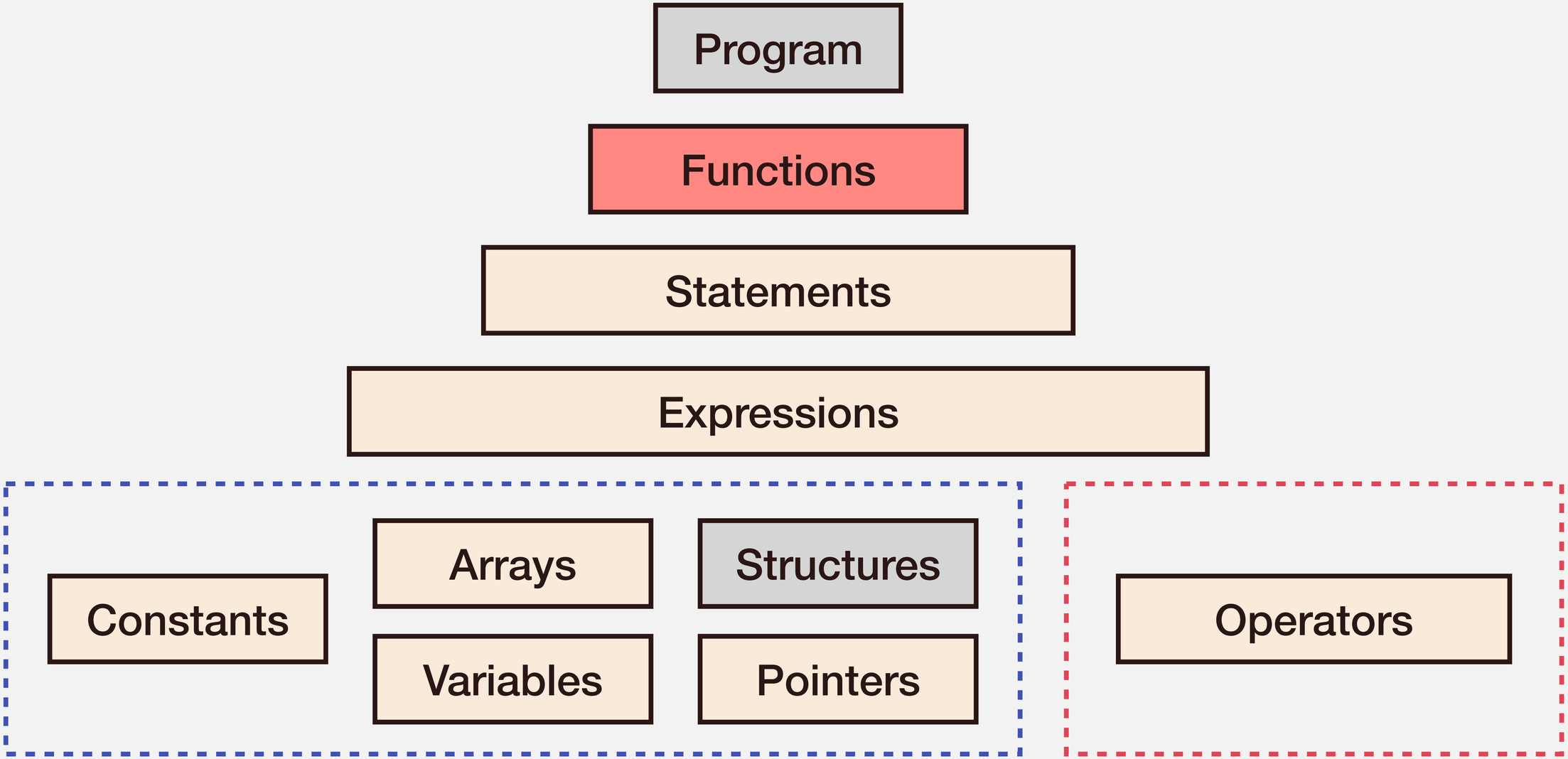
SegFault!

(int **)a

Road Map


Program
Functions
Statements
Expressions
Constants
Arrays
Variables
Structures
Pointers
Operators

Road Map


Program
Functions
Statements
Expressions
Constants
Arrays
Variables
Structures
Pointers
Operators

Recursion

- A function calls **itself**

Recursion

➜ A function calls **itself**

```
void func(...) {
  ...
  func(...)
}

int main() {
  func(...);
}
```

Recursion

➜ A function calls **itself**

```
void func(...) {
  ...
  func(...)
}

int main() {
  func(...);
}
```

Recursion

➜ A function calls **itself**

```
void func(...) {
  ...
  func(...)
}

int main() {
  func(...);
}
```

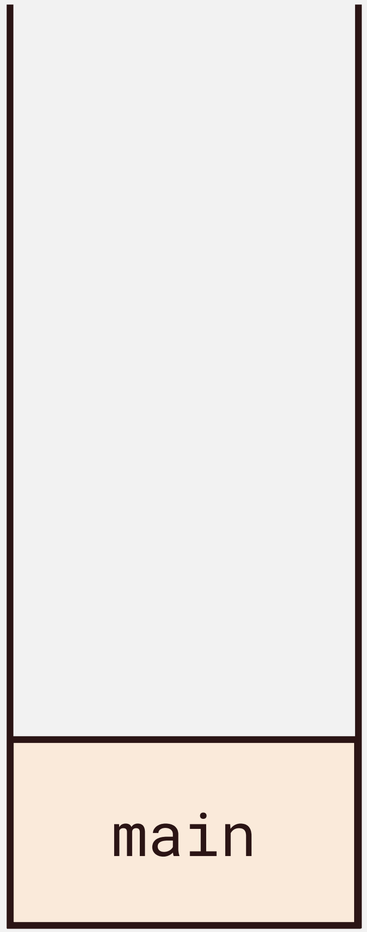
recursively

Recursion

➜ A function calls **itself**

```
void func(...) {
  ...
  func(...)
}

int main() {
  func(...);
}
```

recursively

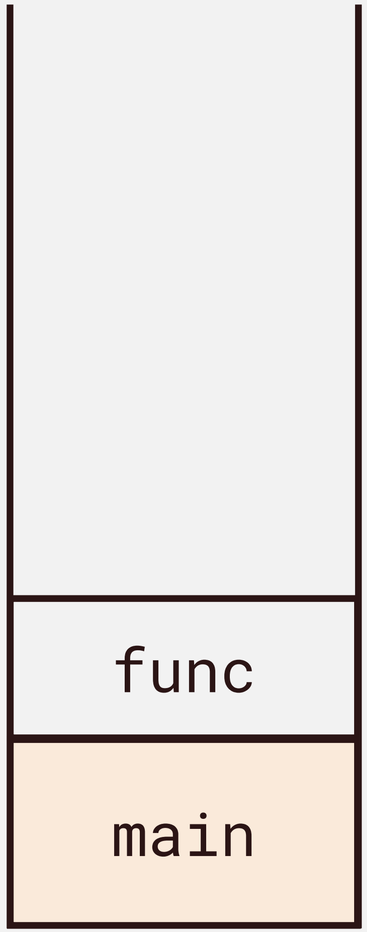


Recursion

➜ A function calls **itself**

```
void func(...) {
  ...
  func(...)
}

int main() {
  func(...);
}
```

recursively



Recursion

➜ A function calls **itself**

```
void func(...) {
  ...
  func(...)
}

int main() {
  func(...);
}
```

recursively

Recursion

➜ A function calls **itself**

```
void func(...) {
  ...
  func(...)
}

int main() {
  func(...);
}
```

recursively

Recursion

➜ A function calls **itself**

```
void func(...) {
  ...
  func(...)
}

int main() {
  func(...);
}
```

recursively

Recursion

➜ A function calls **itself**

```
void func(...) {
  ...
  func(...)
}

int main() {
  func(...);
}
```

recursively

Recursion

➜ A function calls **itself**

```
void func(...) {
  ...
  if (e)
    return;
  func(...)
}

int main() {
  func(...);
}
```

Recursion

➜ A function calls **itself**

```
void func(...) {
  ...
  if (e)
    return;
  func(...)
}

int main() {
  func(...);
}
```

Recursion

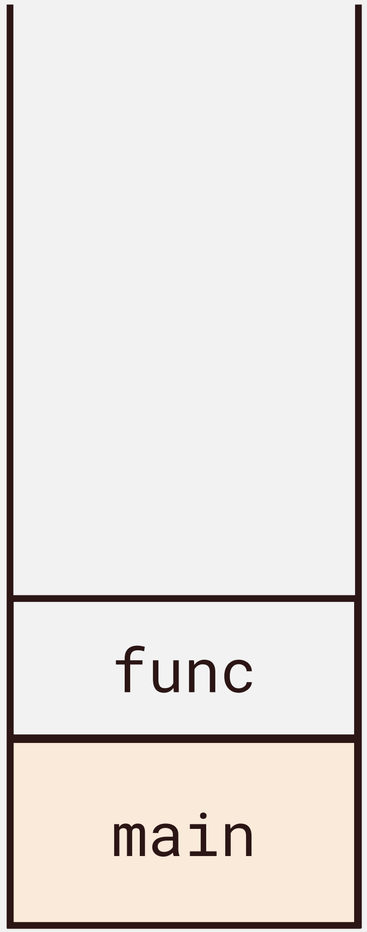
➜ A function calls **itself**

```
void func(...) {
  ...
  if (e)
    return;
  func(...)
}

int main() {
  func(...);
}
```

Recursion

➜ A function calls **itself**

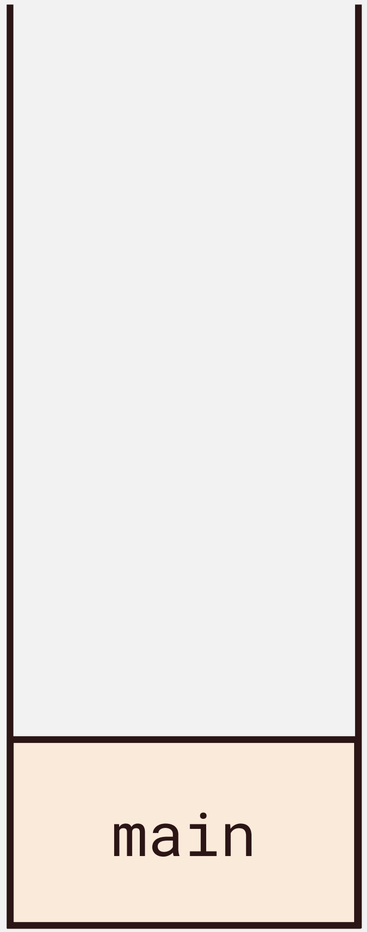
```
void func(...) {
  ...
  if (e)
    return;
  func(...)
}

int main() {
  func(...);
}
```



Recursion

➜ A function calls **itself**

```
void func(...) {
  ...
  if (e)
    return;
  func(...)
}

int main() {
  func(...);
}
```

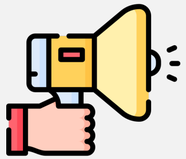


Divide and Conquer

- Break problem into (multiple) sub-problems of the **same type** until the sub-problems are easy to solve

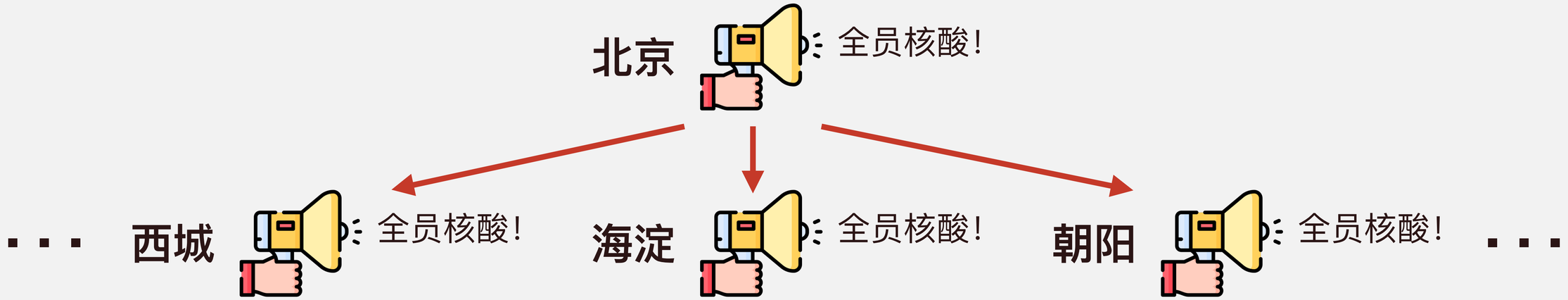
Divide and Conquer

北京

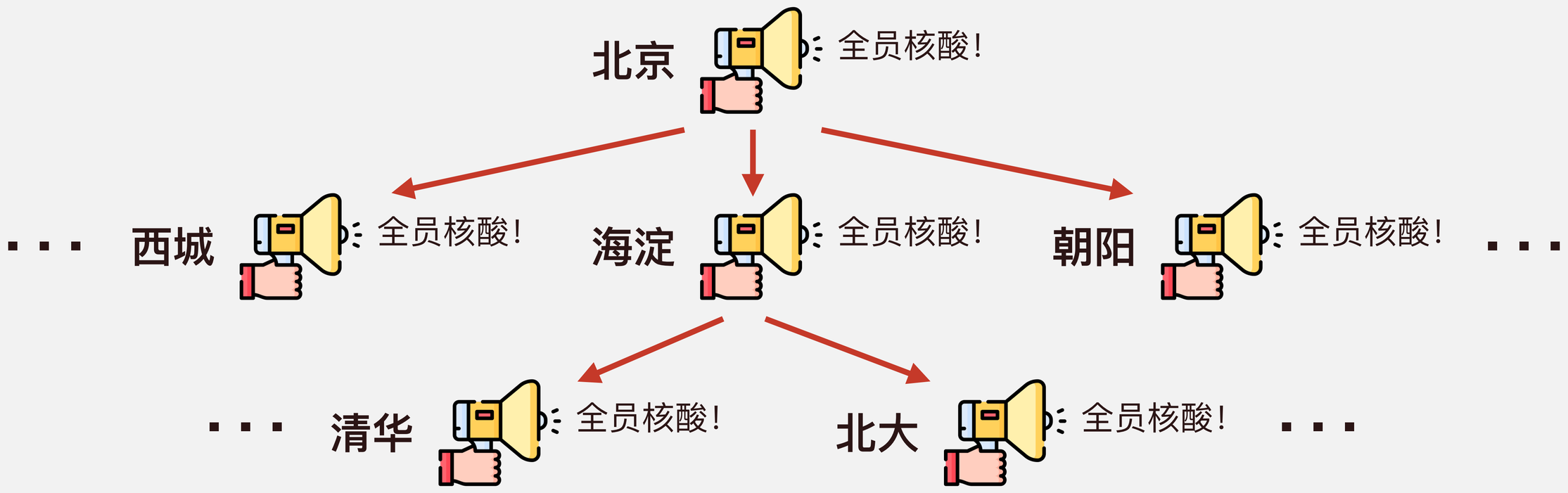


全员核酸！

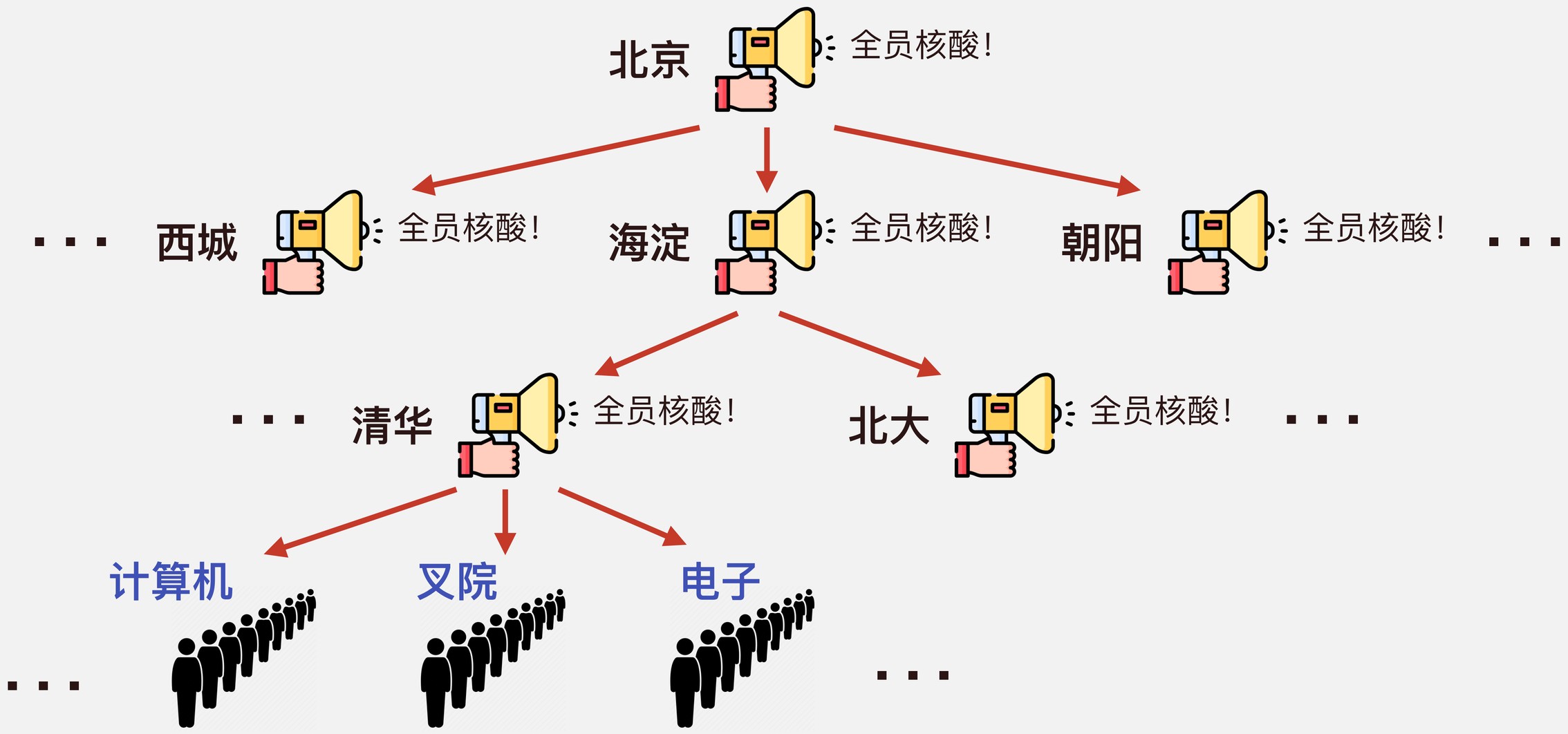
Divide and Conquer



Divide and Conquer



Divide and Conquer



Quicksort

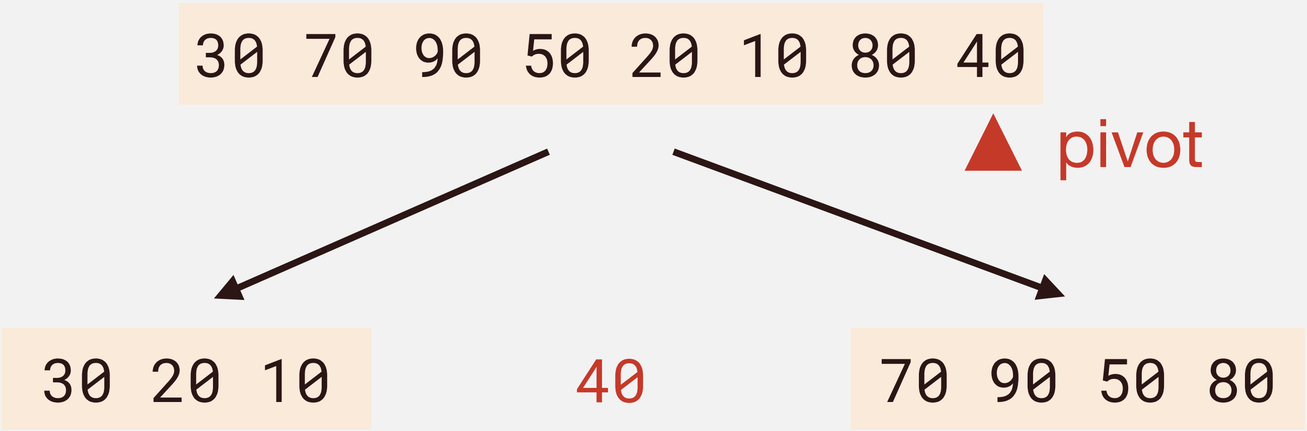
```
30 70 90 50 20 10 80 40
```

Quicksort

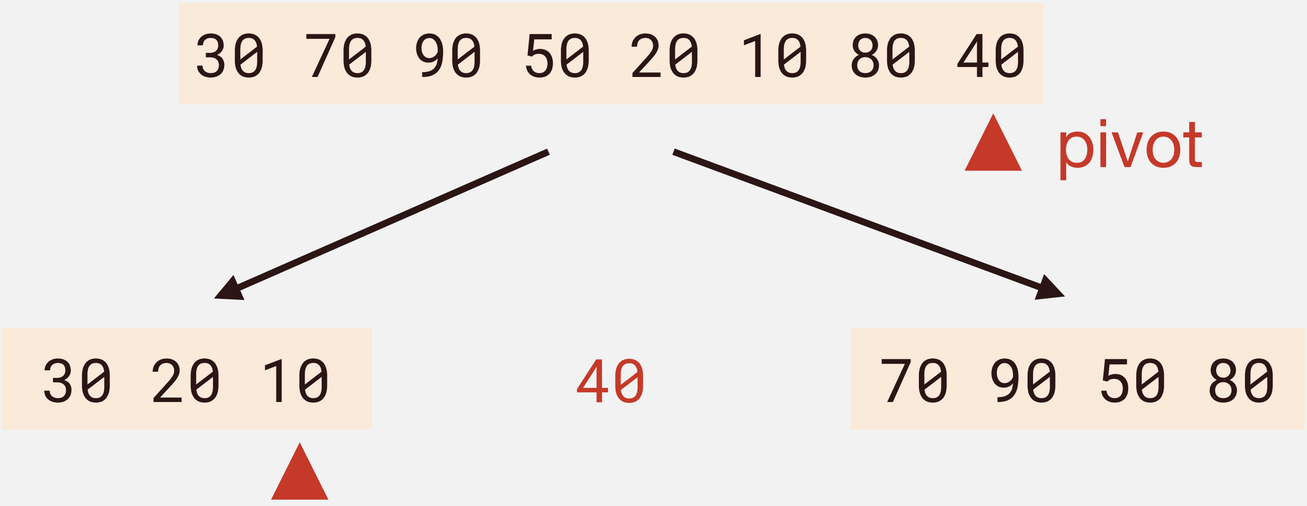
```
30 70 90 50 20 10 80 40
```

▲ pivot

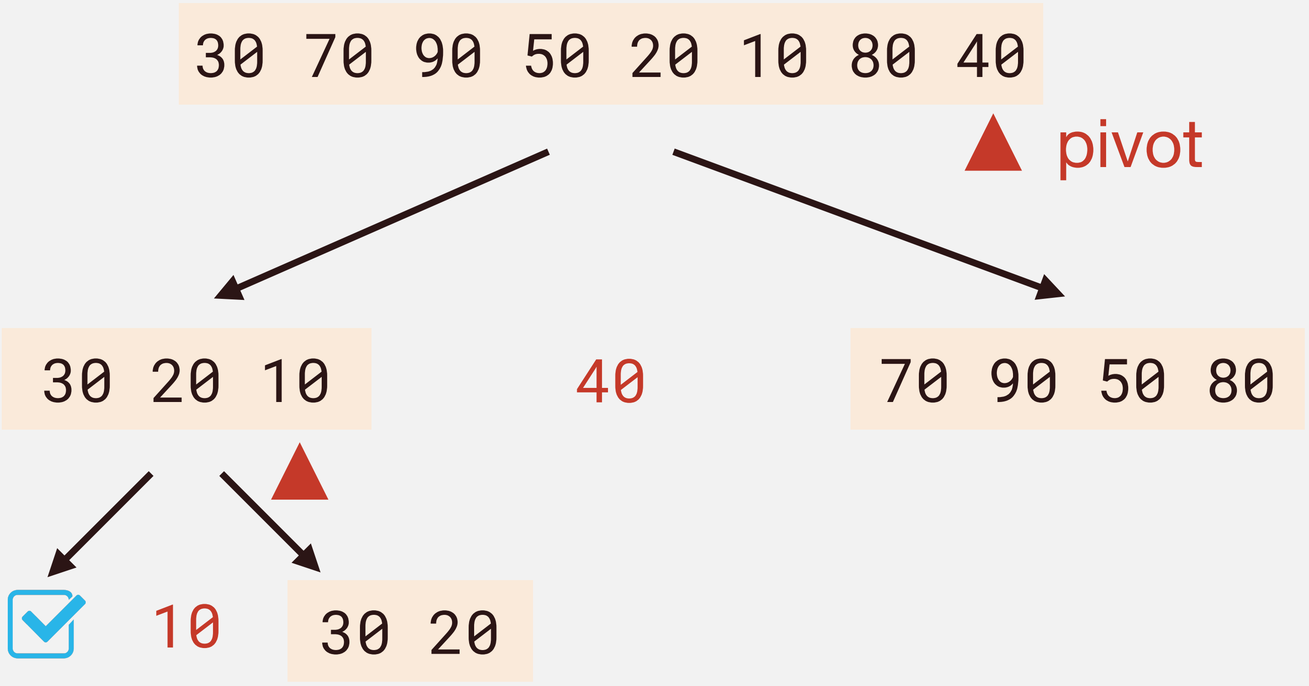
Quicksort



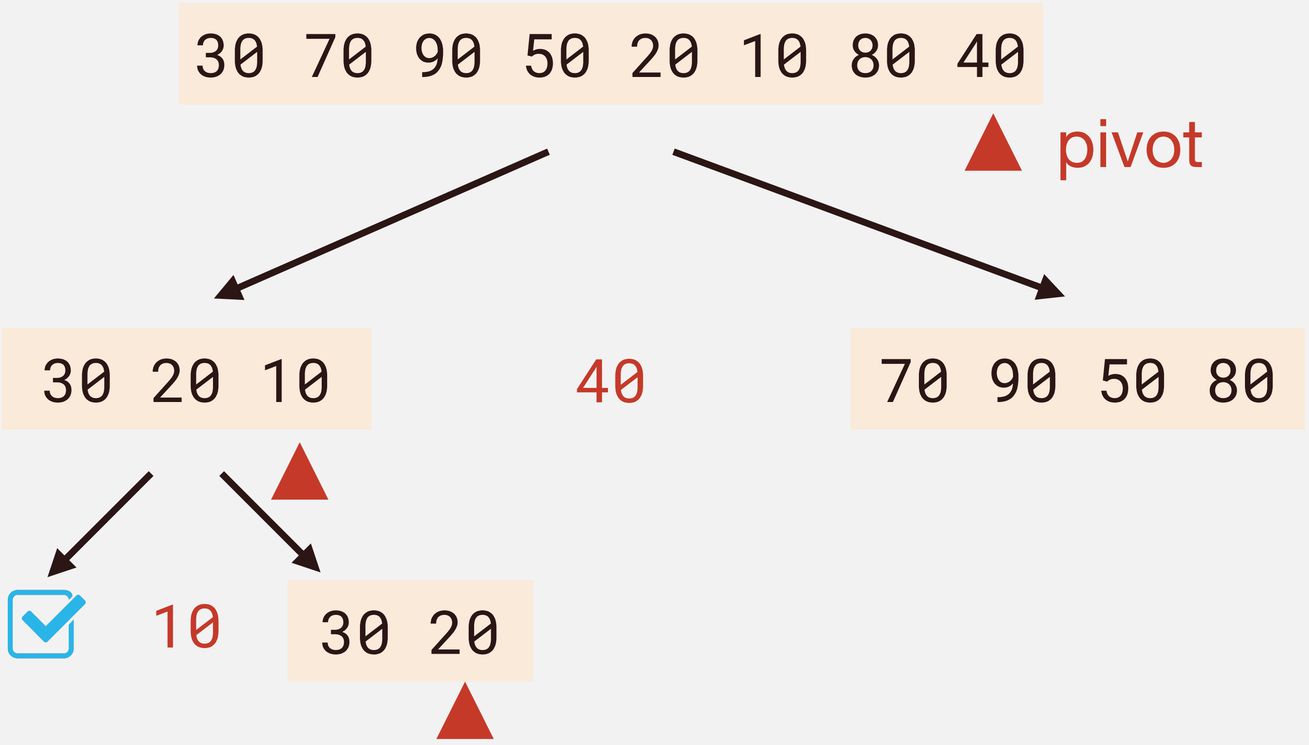
Quicksort



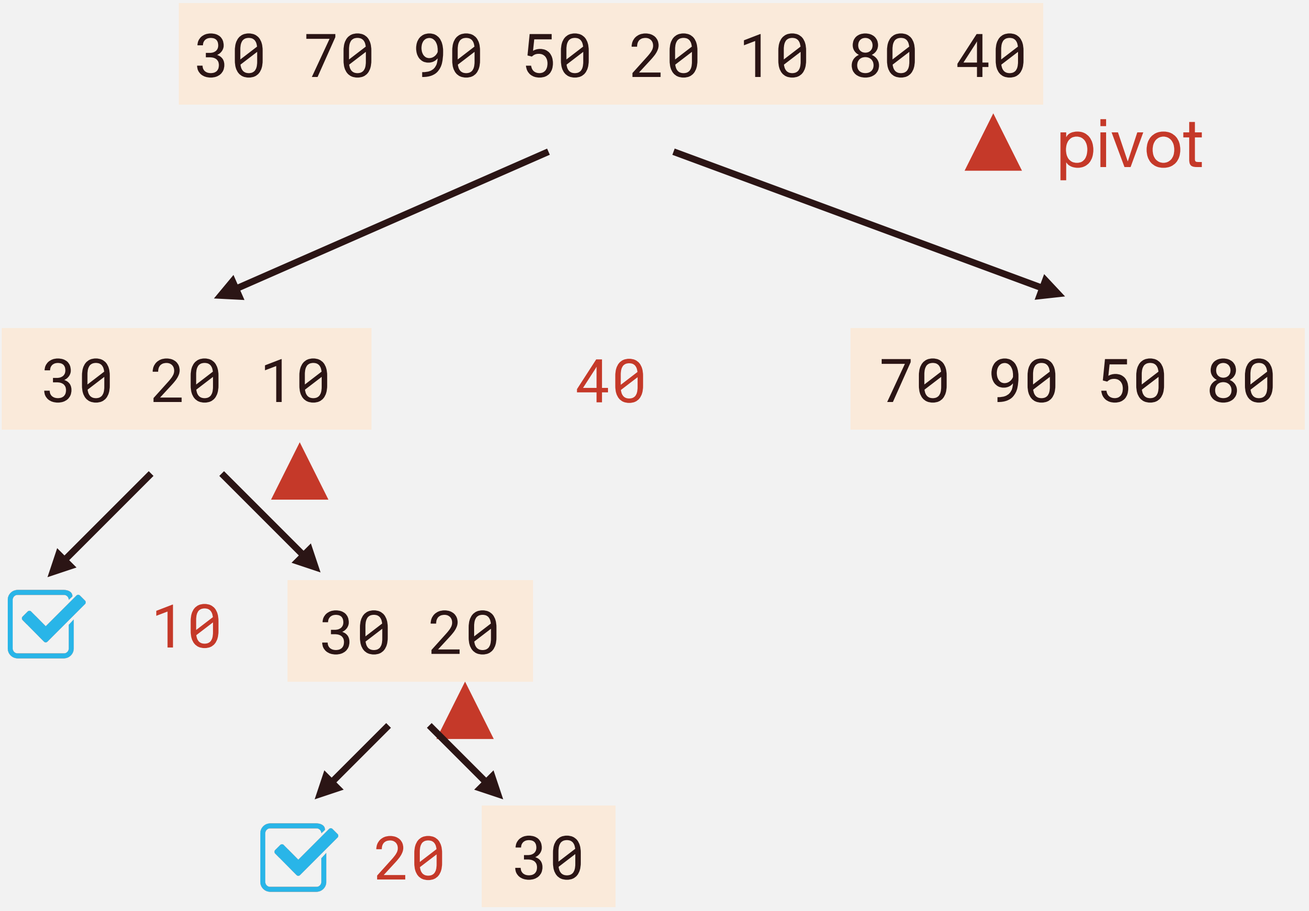
Quicksort



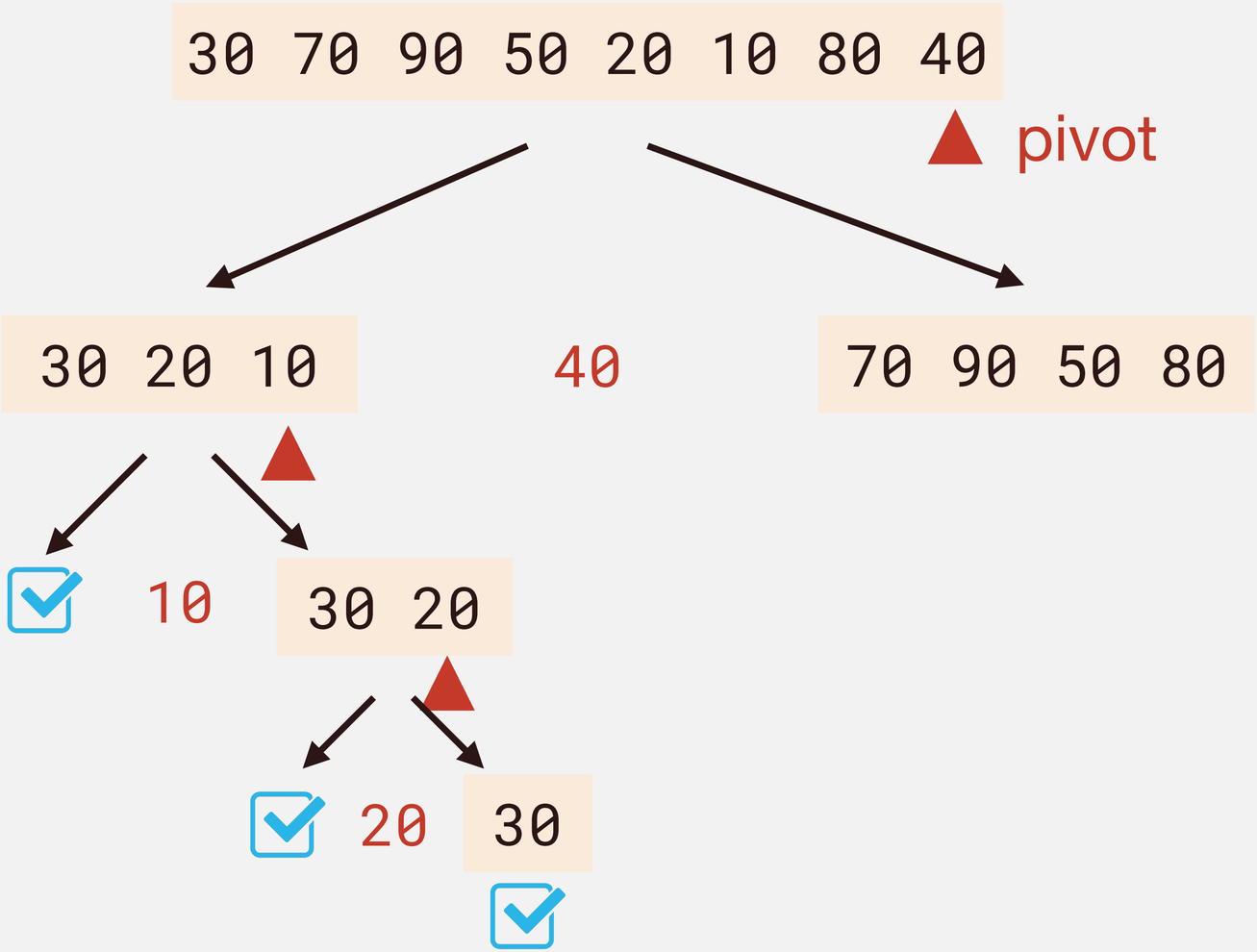
Quicksort



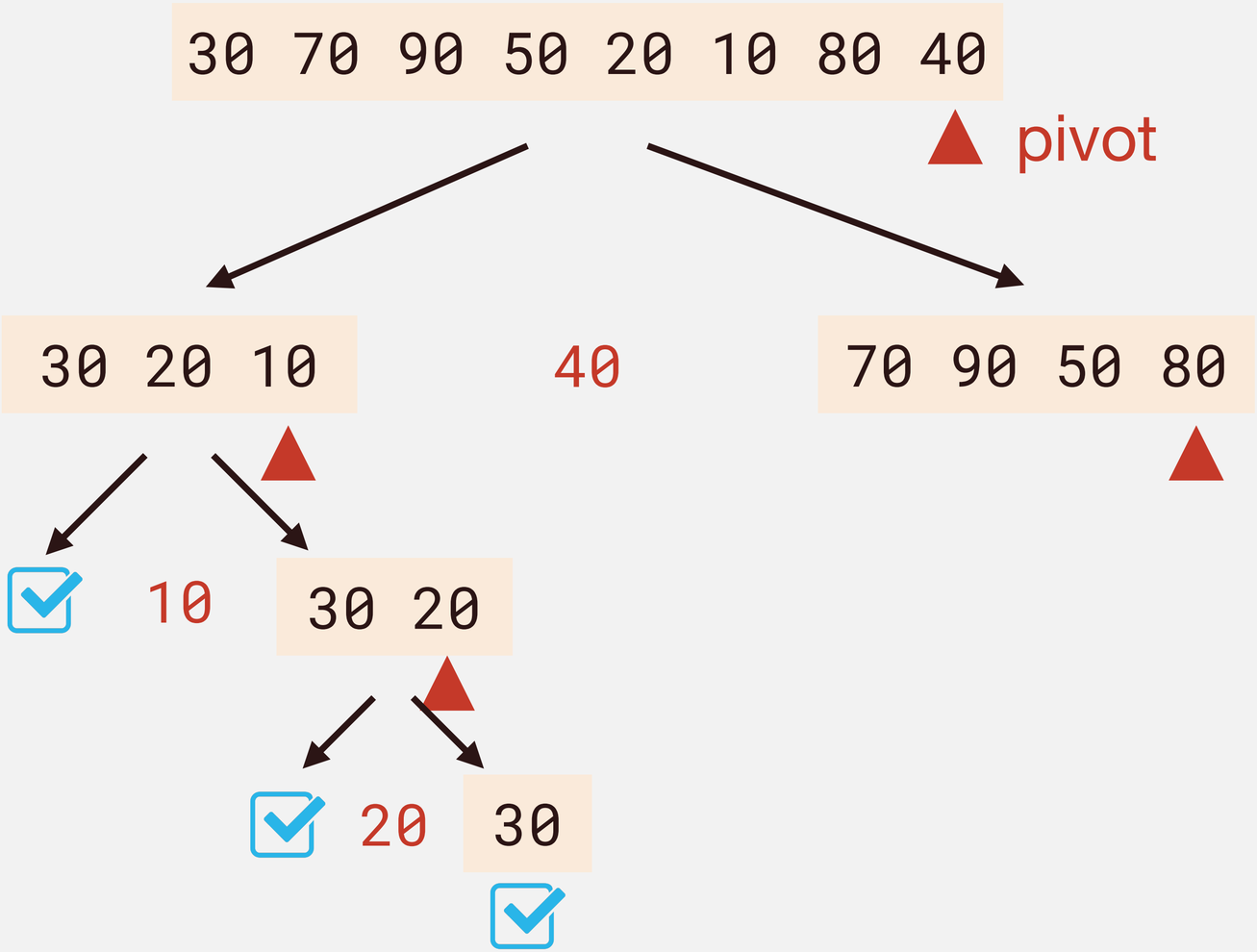
Quicksort



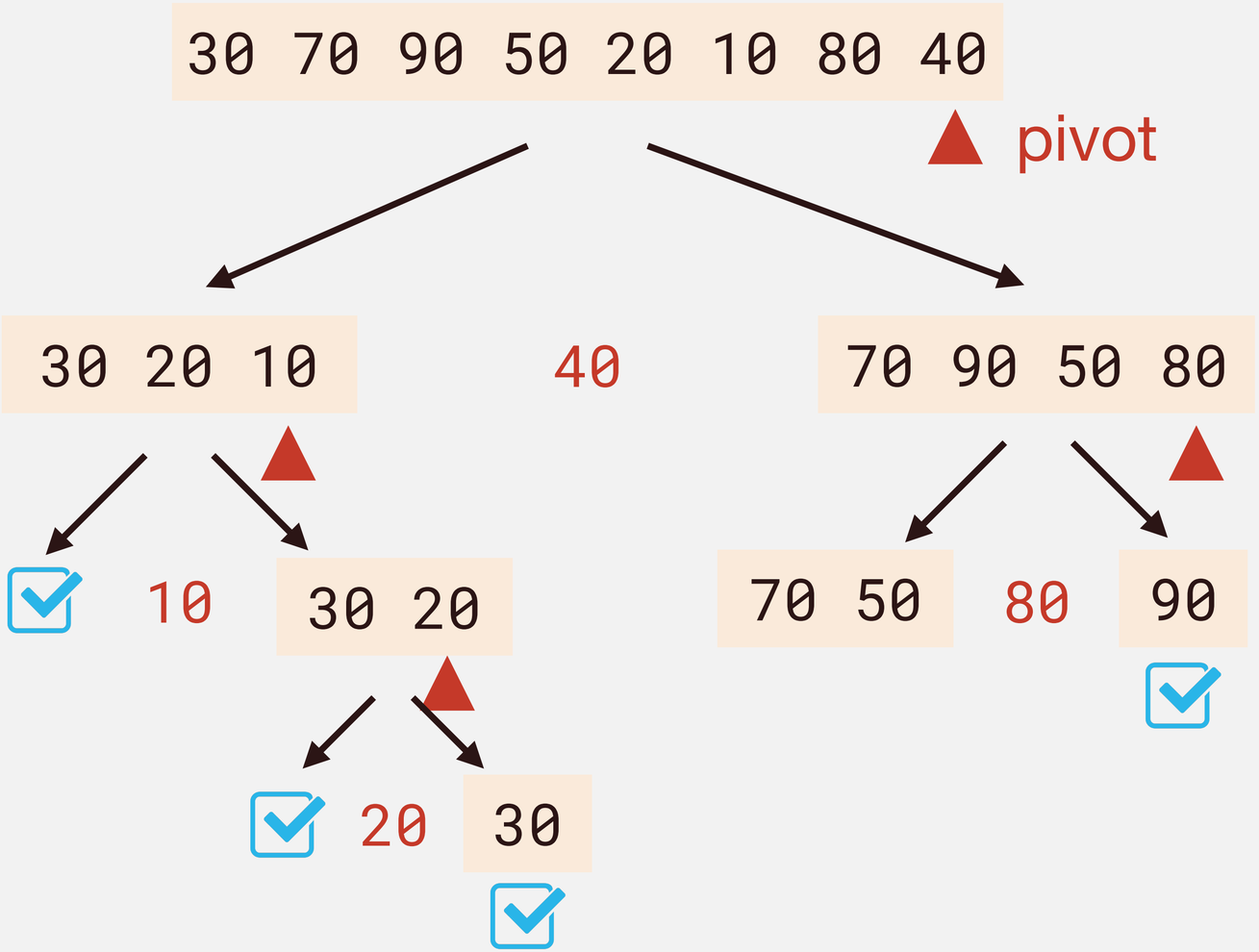
Quicksort



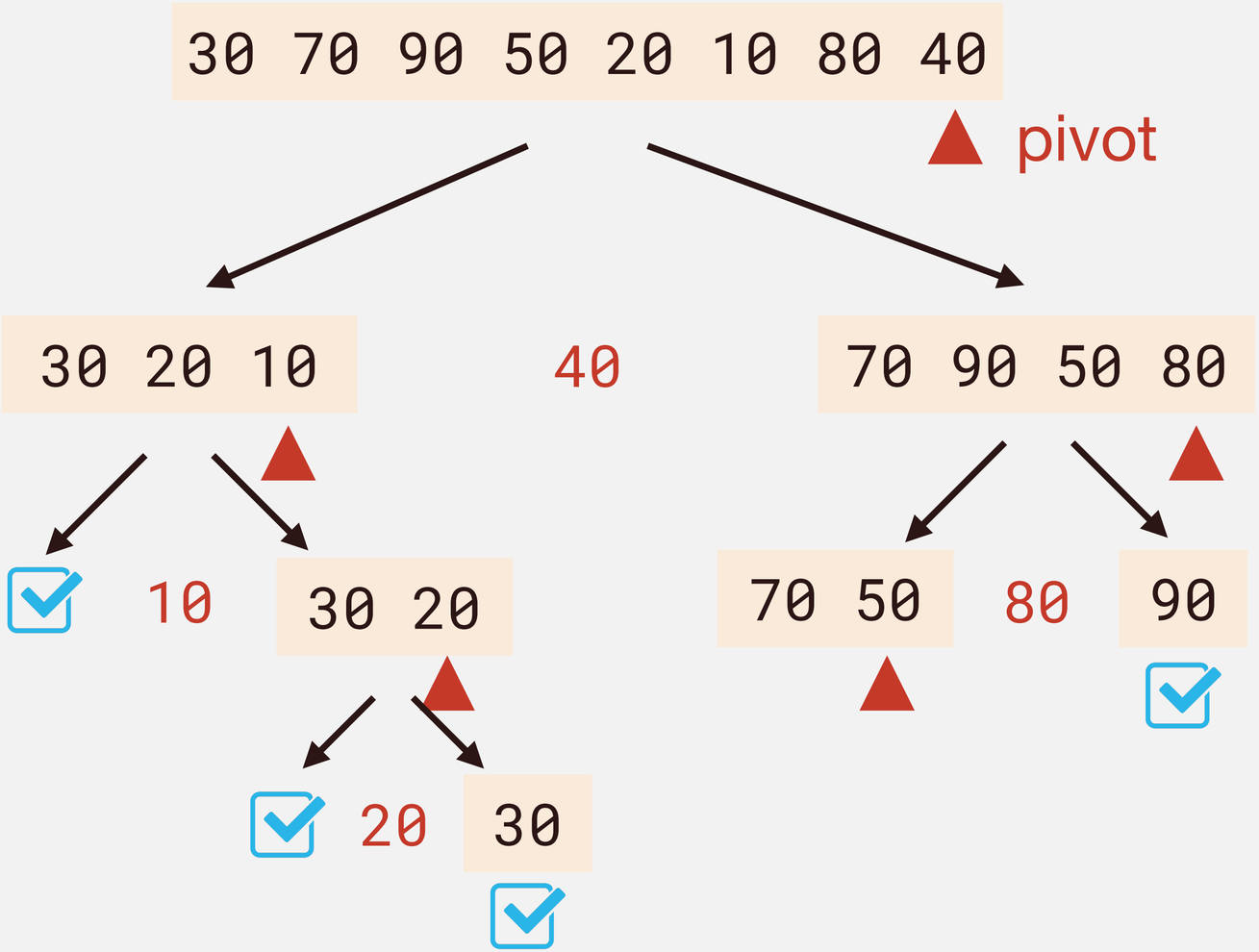
Quicksort



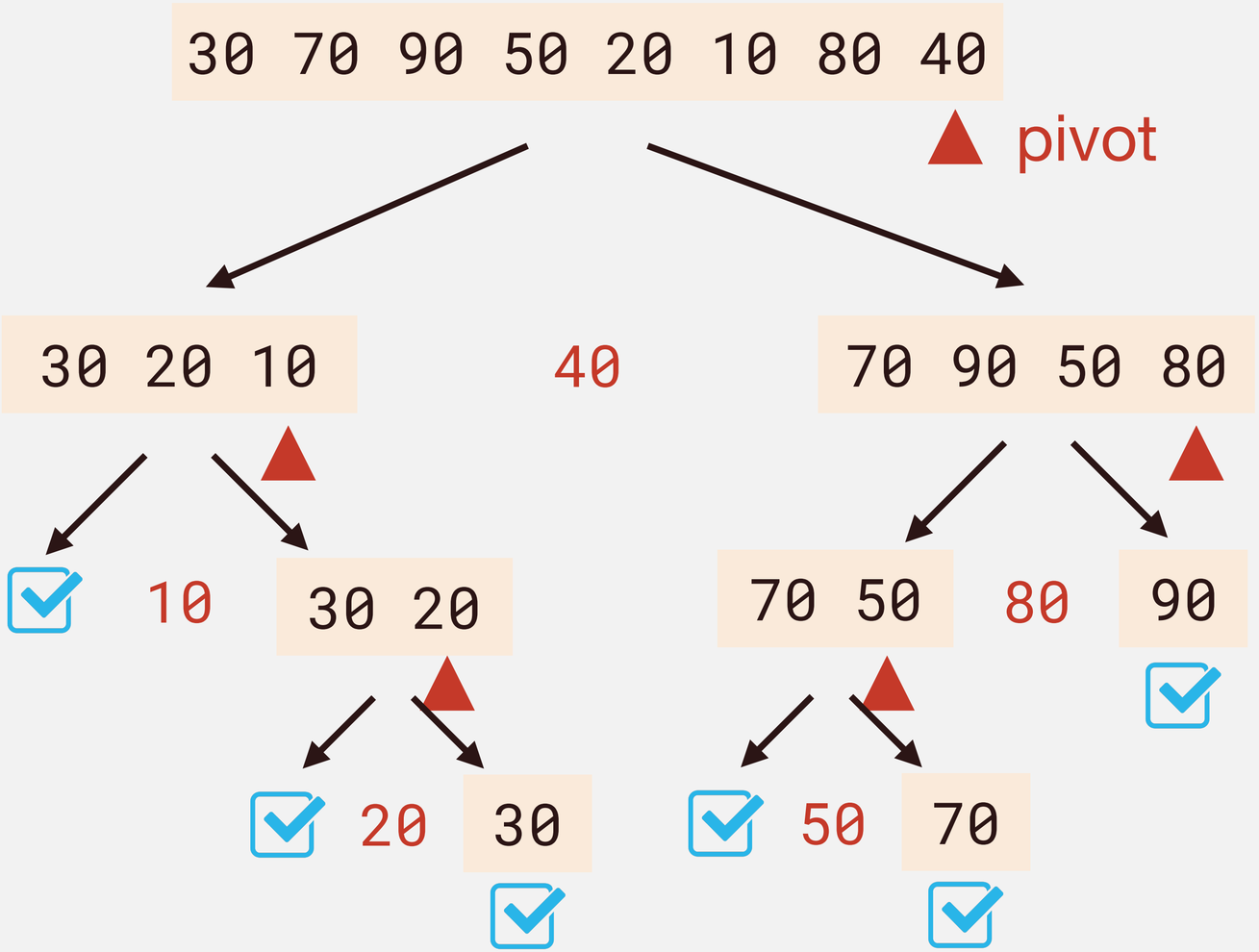
Quicksort


30 70 90 50 20 10 80 40
pivot
30 20 10
40
70 90 50 80
10
30 20
70 50
80
90
20
30

Quicksort


30 70 90 50 20 10 80 40
pivot
30 20 10
40
70 90 50 80
10
30 20
70 50
80
90
20
30

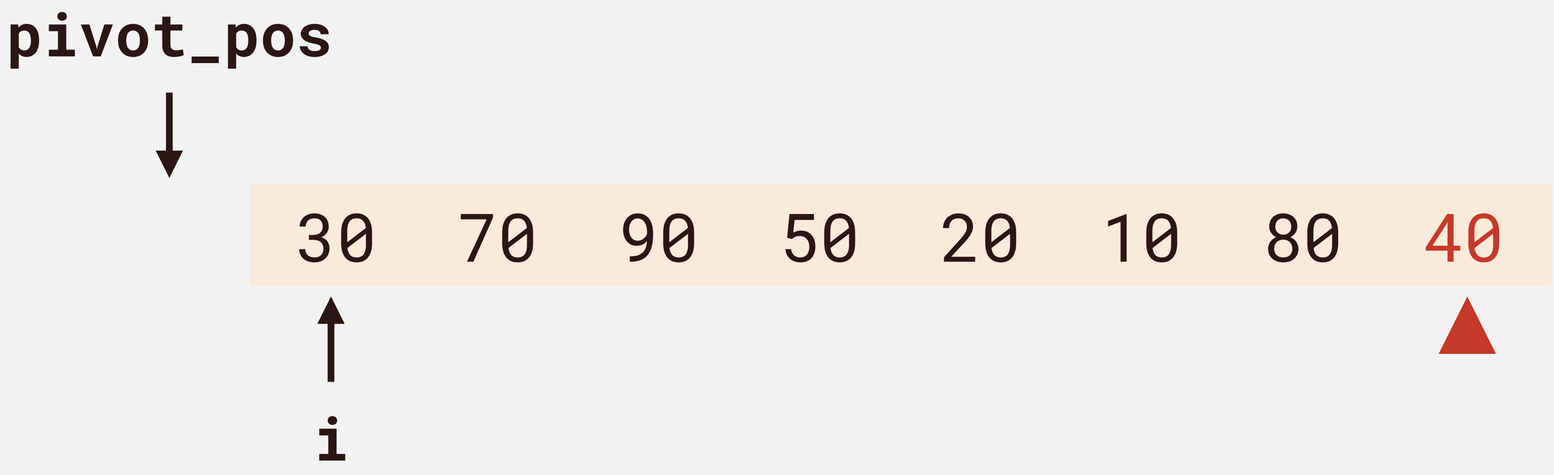
Quicksort


30 70 90 50 20 10 80 40
pivot
30 20 10
40
70 90 50 80
10
30 20
70 50
80
90
20
30
50
70

Quicksort

```
Quicksort(full_range) {
  Pick a pivot
  Rearrange around pivot
  Quicksort(before pivot)
  Quicksort(after pivot)
}
```

Rearrange Around Pivot



Rearrange Around Pivot

pivot_pos

30	70	90	50	20	10	80	40

i

30 < 40

Rearrange Around Pivot

```
pivot_pos
    ↓
 30  70  90  50  20  10  80  40
    ↑                       ▲
    i
```

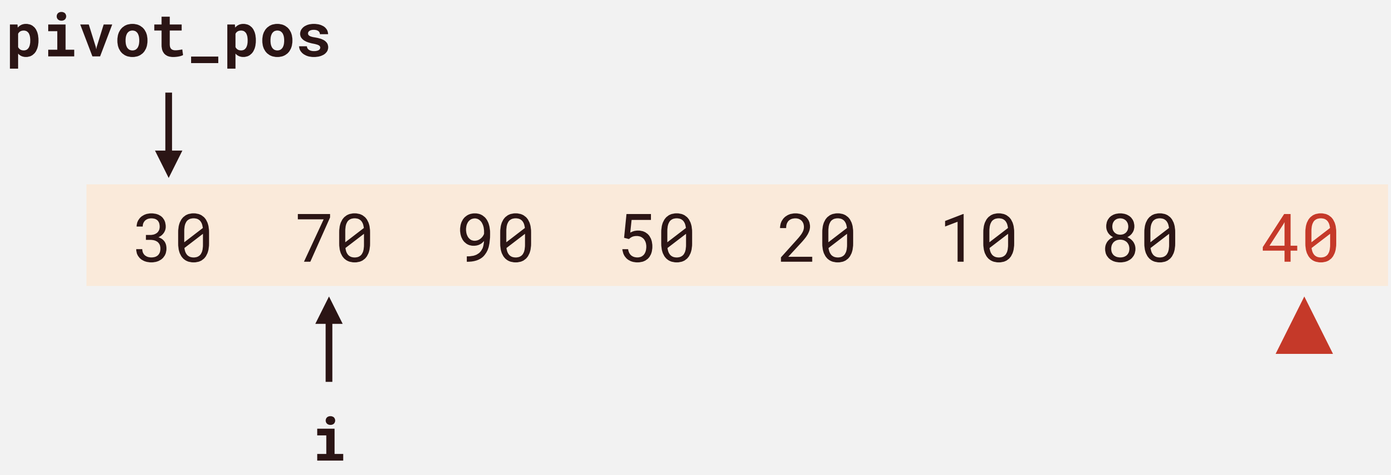
```
  30 < 40
++pivot_pos
```

Rearrange Around Pivot

```
pivot_pos
    ↓
 30  70  90  50  20  10  80  40
    ↑                       ▲
    i
```

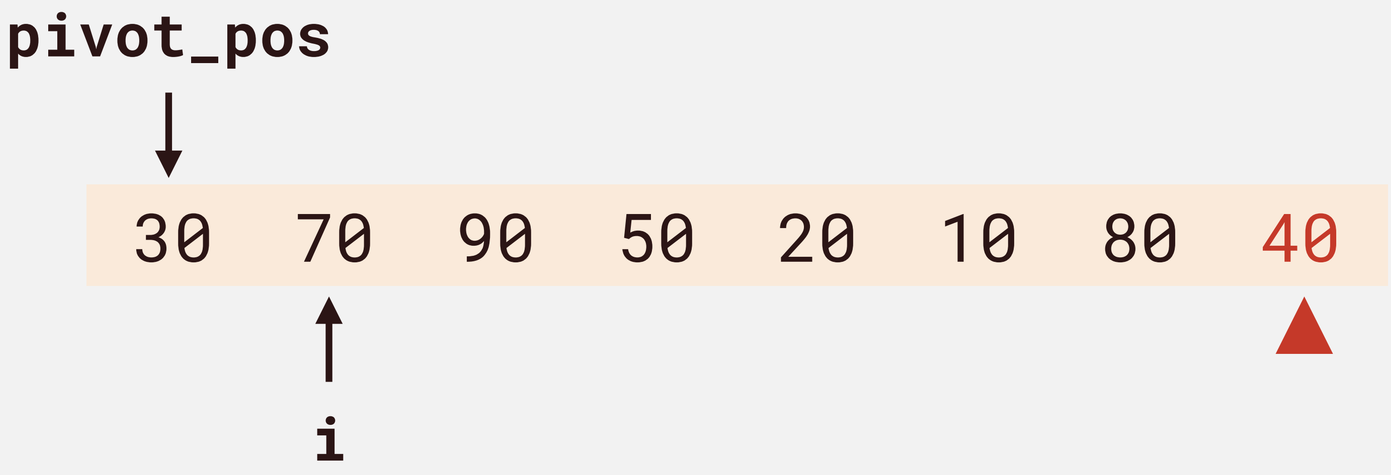
```
  30 < 40
++pivot_pos
    swap
```

Rearrange Around Pivot



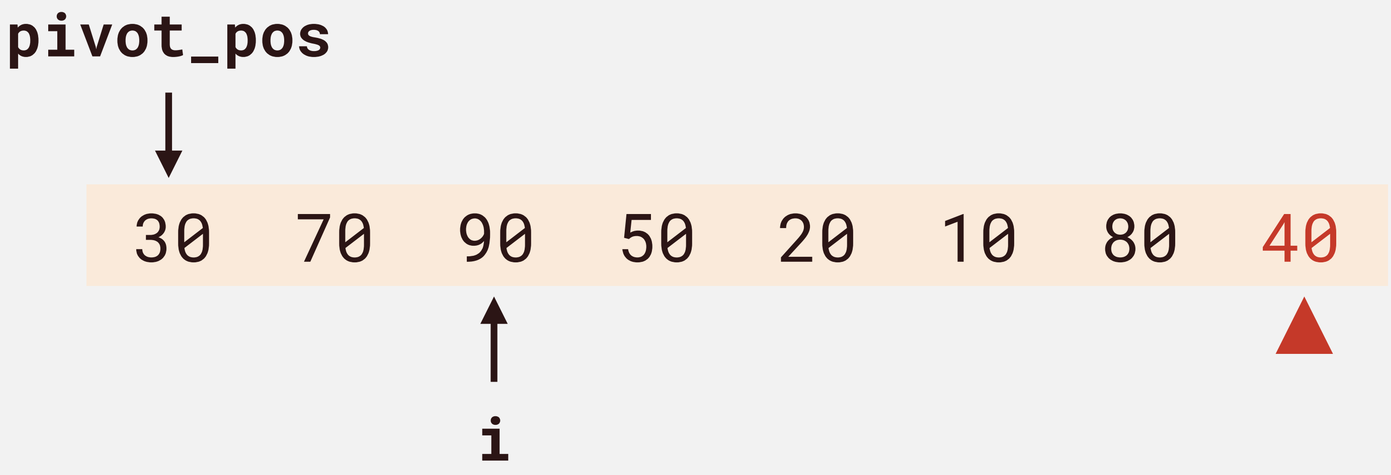

```
  30 < 40
++pivot_pos
    swap
```

Rearrange Around Pivot




```
  30 < 40   70 >= 40
++pivot_pos
    swap
```

Rearrange Around Pivot




```
  30 < 40   70 >= 40
++pivot_pos
    swap
```

Rearrange Around Pivot

```
pivot_pos
    ↓
 30   70   90   50   20   10   80   40
              ↑                     ▲
              i
```

```
  30 < 40   70 >= 40   90 >= 40
++pivot_pos
    swap
```

Rearrange Around Pivot

```
pivot_pos
    ↓
 30   70   90   50   20   10   80   40
                  ↑                  ▲
                  i
```

```
  30 < 40   70 >= 40   90 >= 40
++pivot_pos
    swap
```

Rearrange Around Pivot

```
pivot_pos
    ↓
 30   70   90   50   20   10   80   40
                  ↑                  ▲
                  i
```

```
  30 < 40   70 >= 40   90 >= 40   50 >= 40
++pivot_pos
    swap
```

Rearrange Around Pivot

```
pivot_pos
    ↓
 30   70   90   50   20   10   80   40
                      ↑            ▲
                      i
```

```
  30 < 40   70 >= 40   90 >= 40   50 >= 40
++pivot_pos
    swap
```

Rearrange Around Pivot

```
pivot_pos
    ↓
 30   70   90   50   20   10   80   40
                      ↑             ▲
                      i
```

```
  30 < 40   70 >= 40   90 >= 40   50 >= 40   20 < 40
++pivot_pos
    swap
```

Rearrange Around Pivot

pivot_pos

30	70	90	50	20	10	80	40

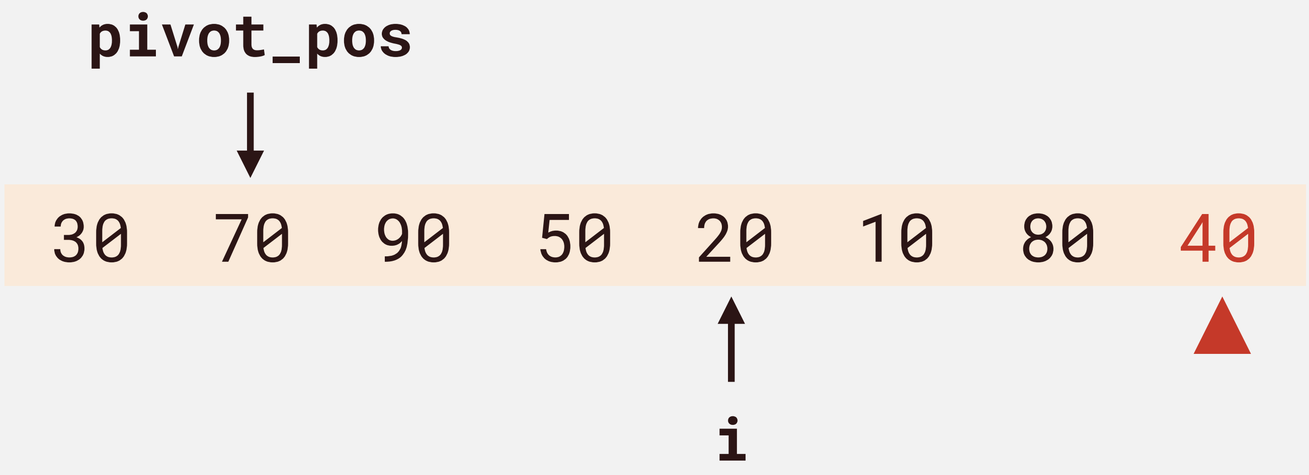
i

30 < 40 70 >= 40 90 >= 40 50 >= 40 20 < 40

++pivot_pos ++pivot_pos

swap

Rearrange Around Pivot



30 < 40	70 >= 40	90 >= 40	50 >= 40	20 < 40
++pivot_pos				++pivot_pos
swap				swap

Rearrange Around Pivot

pivot_pos

30	20	90	50	70	10	80	40

i

30 < 40 ++pivot_pos swap

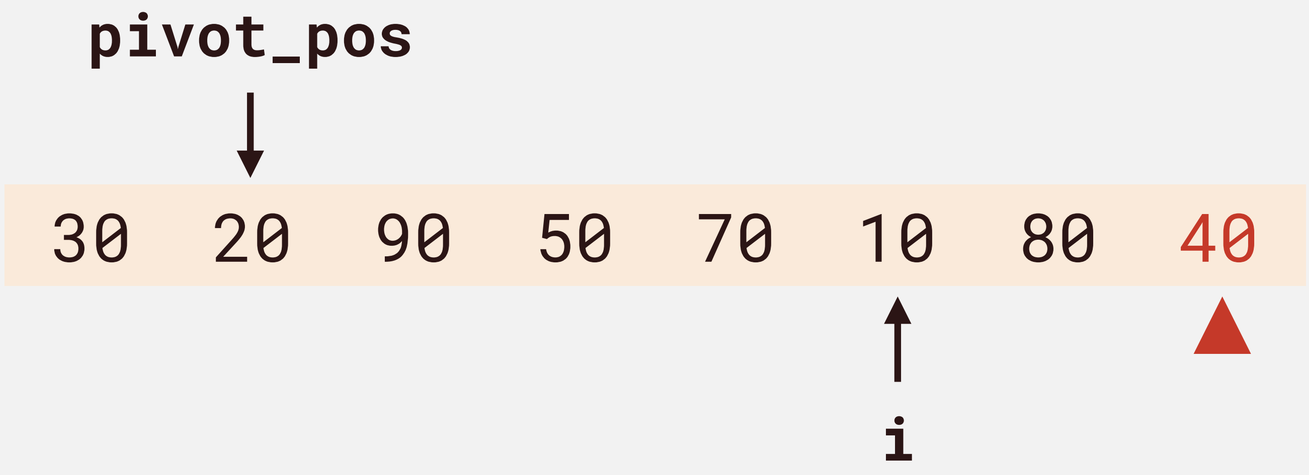
70 >= 40

90 >= 40

50 >= 40

20 < 40 ++pivot_pos swap

Rearrange Around Pivot



30 < 40 ++pivot_pos swap

70 >= 40

90 >= 40

50 >= 40

20 < 40 ++pivot_pos swap

Rearrange Around Pivot

pivot_pos

30	20	90	50	70	10	80	40

i

30 < 40 ++pivot_pos swap

70 >= 40

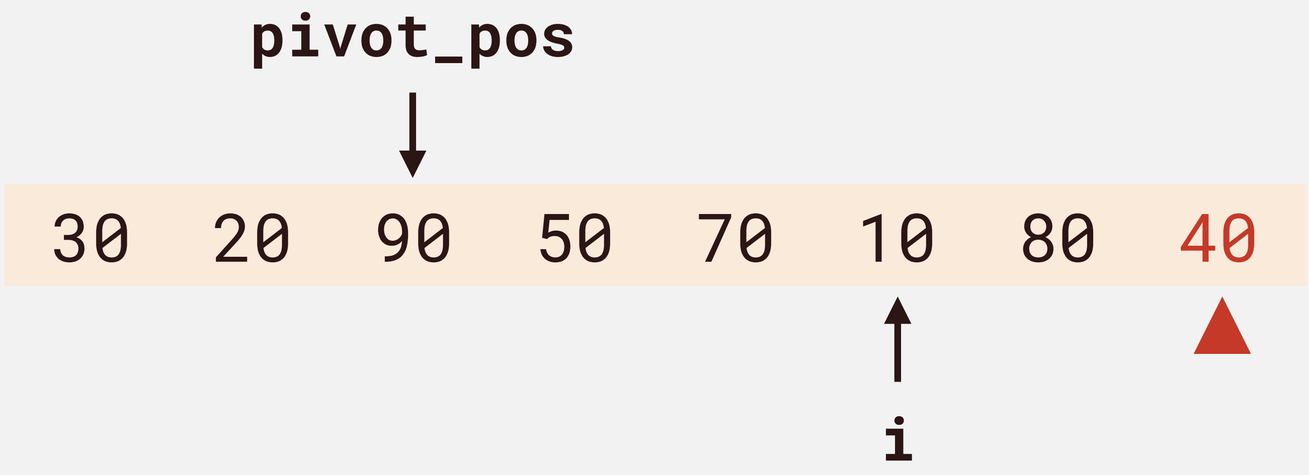
90 >= 40

50 >= 40

20 < 40 ++pivot_pos swap

10 < 40

Rearrange Around Pivot



30 < 40 ++pivot_pos swap

70 >= 40

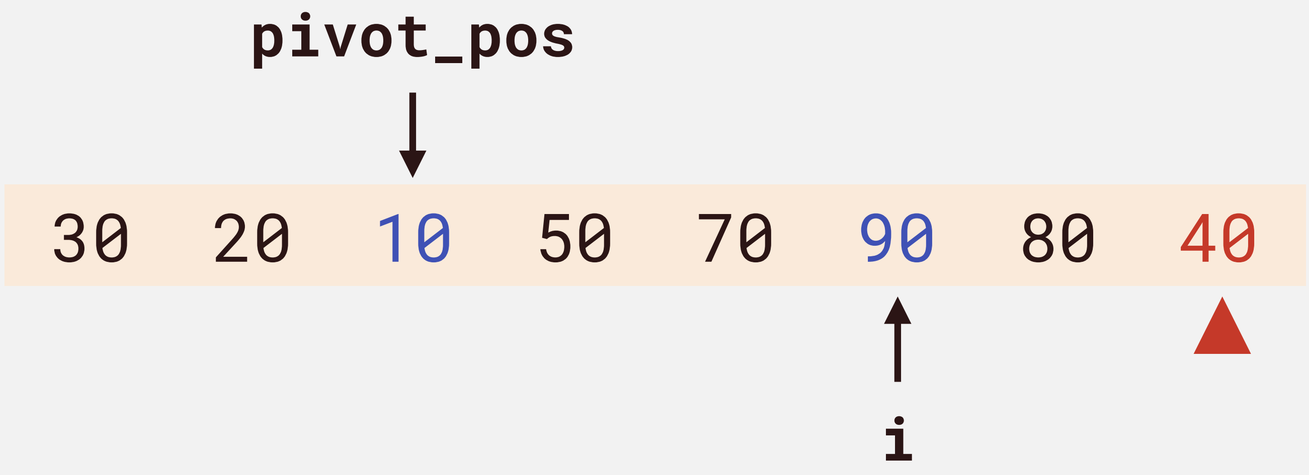
90 >= 40

50 >= 40

20 < 40 ++pivot_pos swap

10 < 40 ++pivot_pos

Rearrange Around Pivot



30 < 40 ++pivot_pos swap

70 >= 40

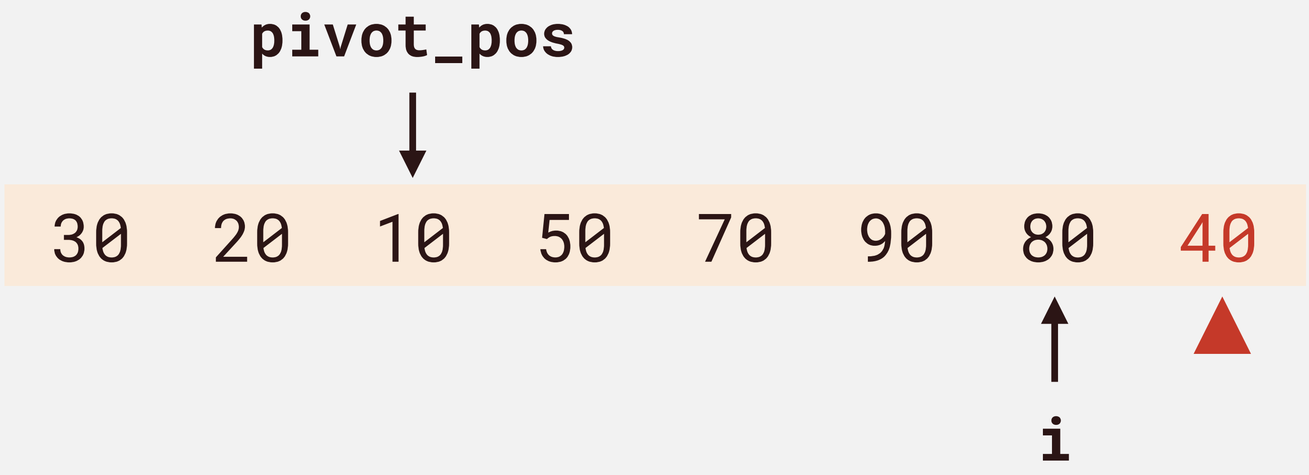
90 >= 40

50 >= 40

20 < 40 ++pivot_pos swap

10 < 40 ++pivot_pos swap

Rearrange Around Pivot



30 < 40	70 >= 40	90 >= 40	50 >= 40	20 < 40	10 < 40
++pivot_pos				++pivot_pos	++pivot_pos
swap				swap	swap

Rearrange Around Pivot

pivot_pos

30	20	10	50	70	90	80	40

i

30 < 40 ++pivot_pos swap

70 >= 40

90 >= 40

50 >= 40

20 < 40 ++pivot_pos swap

10 < 40 ++pivot_pos swap

final swap

Rearrange Around Pivot

pivot_pos

30 20 10 50 70 90 80 40

i

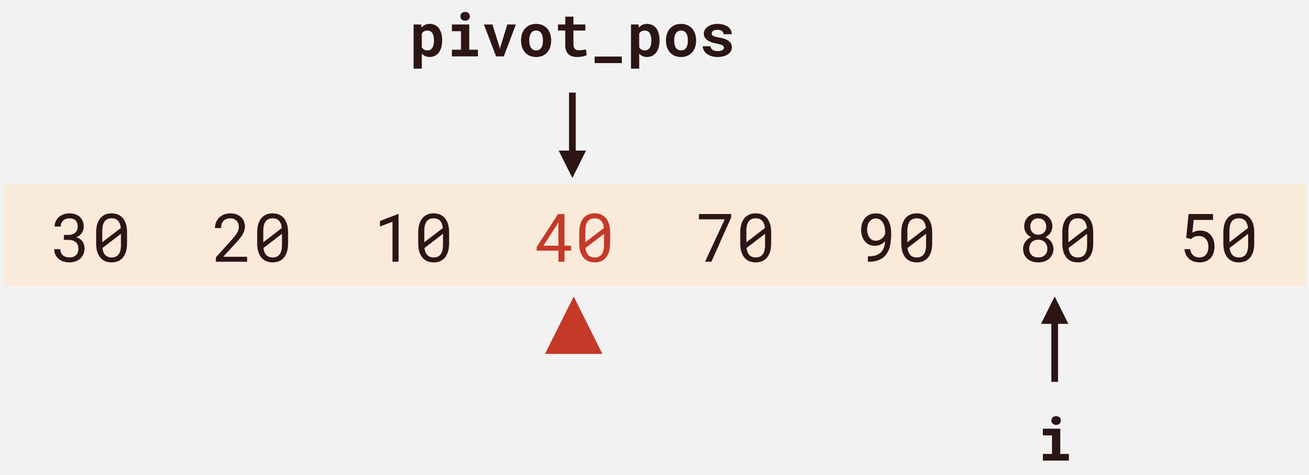
30 < 40 70 >= 40 90 >= 40 50 >= 40 20 < 40 10 < 40

++pivot_pos ++pivot_pos ++pivot_pos

swap swap swap

final swap

Rearrange Around Pivot



30 < 40 ++pivot_pos swap

70 >= 40

90 >= 40

50 >= 40

20 < 40 ++pivot_pos swap

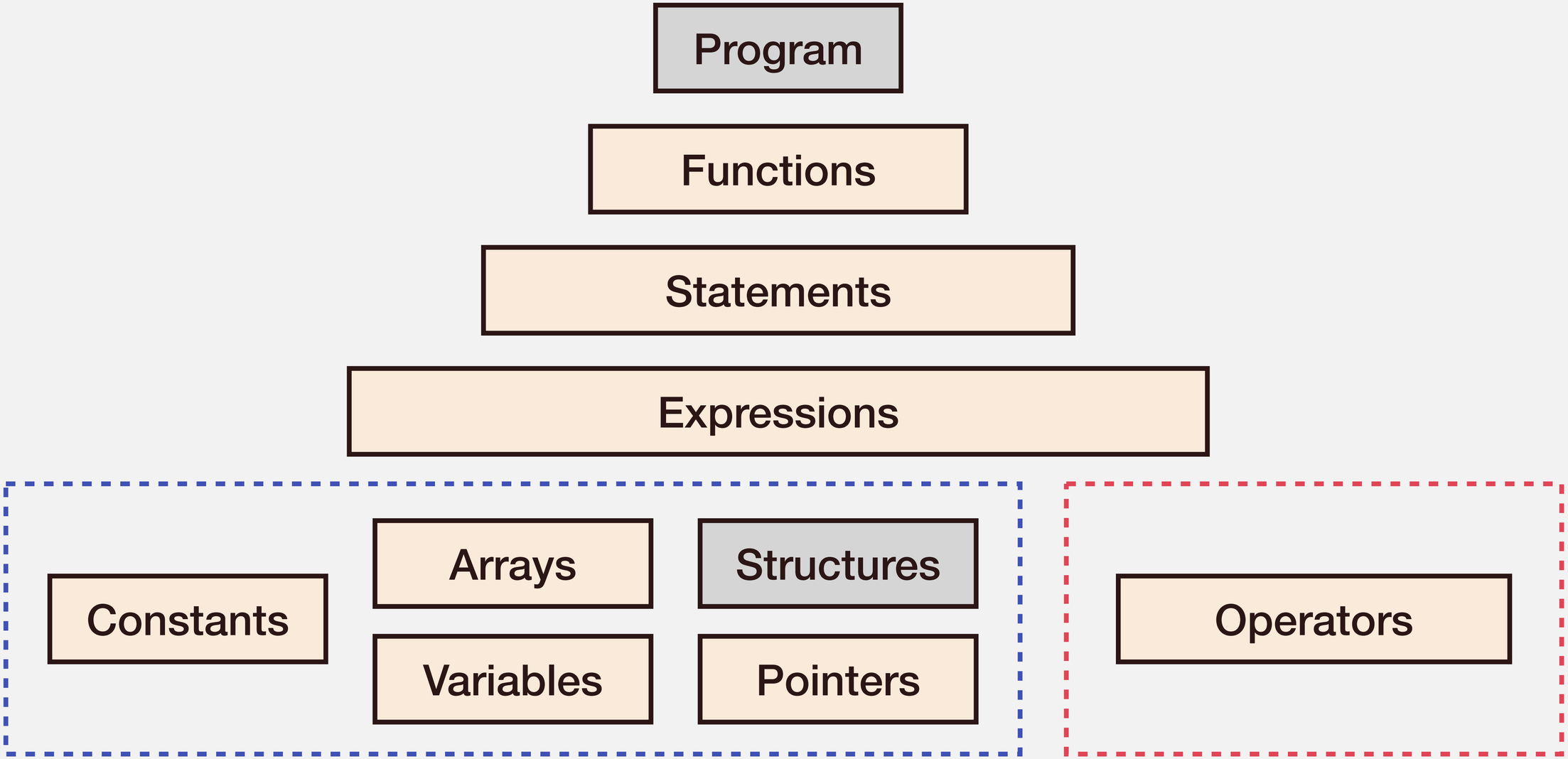
10 < 40 ++pivot_pos swap

final swap

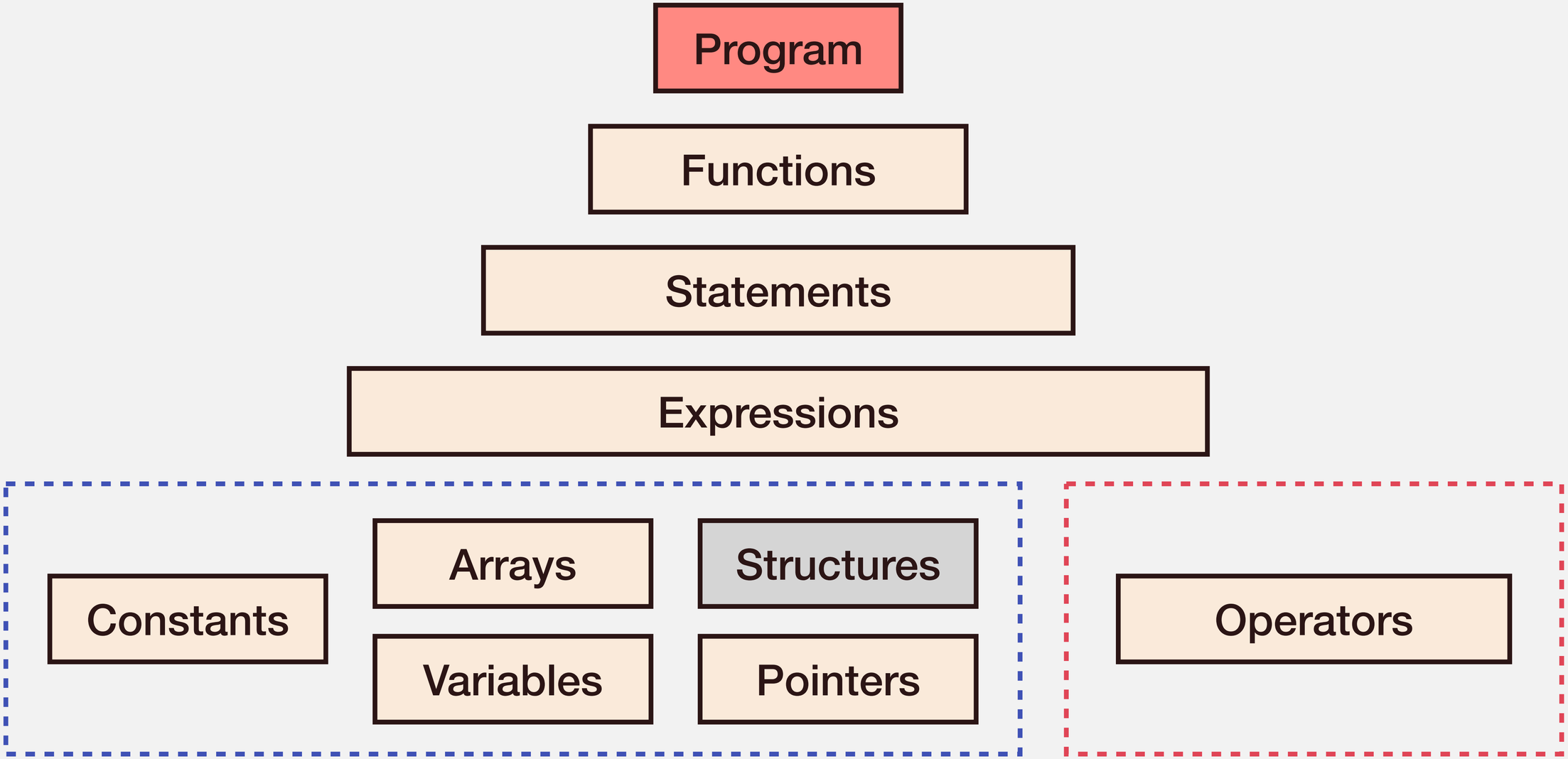
Quicksort

```
Quicksort(full_range) {
  Pick a pivot
  Rearrange around pivot
  Quicksort(before pivot)
  Quicksort(after pivot)
}
```

Road Map


Program
Functions
Statements
Expressions
Constants
Arrays
Variables
Structures
Pointers
Operators

Road Map


Program
Functions
Statements
Expressions
Constants
Arrays
Variables
Structures
Pointers
Operators

Single-File C Program

```c
#include <stdlib.h>
...
#define ERR_MSG “Wan le, bbq le”
...
int evil_global_var = 0;
...
void Quicksort(int *a, int l, int r);
...
int main() {
  ...
}
void Quicksort(int *a, int l, int r) {
  ...
}
```

1. Include header files
2. Define constants and macros
3. Global variables (**evil**)
4. Function prototypes
5. main
6. Function definition

Global Variables

- Declared and defined outside any function in a program

Global Variables

- Declared and defined outside any function in a program
- Accessible anywhere throughout the lifetime of the program

Global Variables

- Declared and defined outside any function in a program
- Accessible anywhere throughout the lifetime of the program
- Non-const global variables are **EVIL** 

Global Variables

- Declared and defined outside any function in a program
- Accessible anywhere throughout the lifetime of the program
- Non-const global variables are **EVIL** 
  - Can be modified by any part of the code; hard to reason about every possible use

Global Variables

- Declared and defined outside any function in a program
- Accessible anywhere throughout the lifetime of the program
- Non-const global variables are **EVIL** 
  - Can be modified by any part of the code; hard to reason about every possible use
  - Implicit coupling breaks modularity

Global Variables

- Declared and defined outside any function in a program
- Accessible anywhere throughout the lifetime of the program
- Non-const global variables are **EVIL** 
  - Can be modified by any part of the code; hard to reason about every possible use
  - Implicit coupling breaks modularity
  - No access control

Global Variables

- Declared and defined outside any function in a program
- Accessible anywhere throughout the lifetime of the program
- Non-const global variables are **EVIL** 
  - Can be modified by any part of the code; hard to reason about every possible use
  - Implicit coupling breaks modularity
  - No access control
  - Namespace pollution

Single-File C Program

```c
#include <stdlib.h>
...
#define ERR_MSG “Wan le, bbq le”
...

void Quicksort(int *a, int l, int r);
...

int main() {
  ...
}

void Quicksort(int *a, int l, int r) {
  ...
}
```

1. Include header files
2. Define constants and macros
3. Function prototypes
4. main
5. Function definition

Single-File C Program

```c
#include <stdlib.h>
...
#define ERR_MSG “Wan le, bbq le”
...
void Quicksort(int *a, int l, int r);
...
int main() {
  ...
}
void Quicksort(int *a, int l, int r) {
  ...
}
```

1. Include header files
2. Define constants and macros
3. Function prototypes
4. main
5. Function definition

Header Files (.h)

Source Files (.c)

Header File

sort_util.h

```
#ifndef __SORT_UTIL_H__
#define __SORT_UTIL_H__

#include <stdlib.h>

#define DEBUG_LEVEL 1

extern int evil_global_var;

void Quicksort(int *a, int l, int r);
void Mergesort(int *a, int l, int r);

#endif
```

Header File

sort_util.h

```
#ifndef __SORT_UTIL_H__
#define __SORT_UTIL_H__

#include <stdlib.h>

#define DEBUG_LEVEL 1

extern int evil_global_var;

void Quicksort(int *a, int l, int r);
void Mergesort(int *a, int l, int r);

#endif
```

Header File

sort_util.h

```
#ifndef __SORT_UTIL_H__
#define __SORT_UTIL_H__

#include <stdlib.h>

#define DEBUG_LEVEL 1

extern int evil_global_var;

void Quicksort(int *a, int l, int r);
void Mergesort(int *a, int l, int r);

#endif
```

Prevent variables/functions from being declared multiple times

Header File

sort_util.h

```
#ifndef __SORT_UTIL_H__
#define __SORT_UTIL_H__

#include <stdlib.h>

#define DEBUG_LEVEL 1

extern int evil_global_var;

void Quicksort(int *a, int l, int r);
void Mergesort(int *a, int l, int r);

#endif
```

Header File

sort_util.h

```
#ifndef __SORT_UTIL_H__
#define __SORT_UTIL_H__

#include <stdlib.h>

#define DEBUG_LEVEL 1

extern int evil_global_var;

void Quicksort(int *a, int l, int r);
void Mergesort(int *a, int l, int r);

#endif
```

← **Declares** that the global variable is defined somewhere else

Source File

sort_util.c

```c
#include “sort_util.h”

#include <stdlib.h>

static void Swap(int *a, int i, int j) {
  ...
}

void Quicksort(int *a, int l, int r) {
  ...
}

void Mergesort(int *a, int l, int r) {
  ...
}
```

Source File

sort_util.c

```
#include “sort_util.h”
```

← The corresponding header file

```
#include <stdlib.h>

static void Swap(int *a, int i, int j) {
  ...
}

void Quicksort(int *a, int l, int r) {
  ...
}

void Mergesort(int *a, int l, int r) {
  ...
}
```

Source File

sort_util.c

```
#include “sort_util.h”
```

← The corresponding header file

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

← Other function declarations needed for the implementation

```
static void Swap(int *a, int i, int j) {
  ...
}

void Quicksort(int *a, int l, int r) {
  ...
}

void Mergesort(int *a, int l, int r) {
  ...
}
```

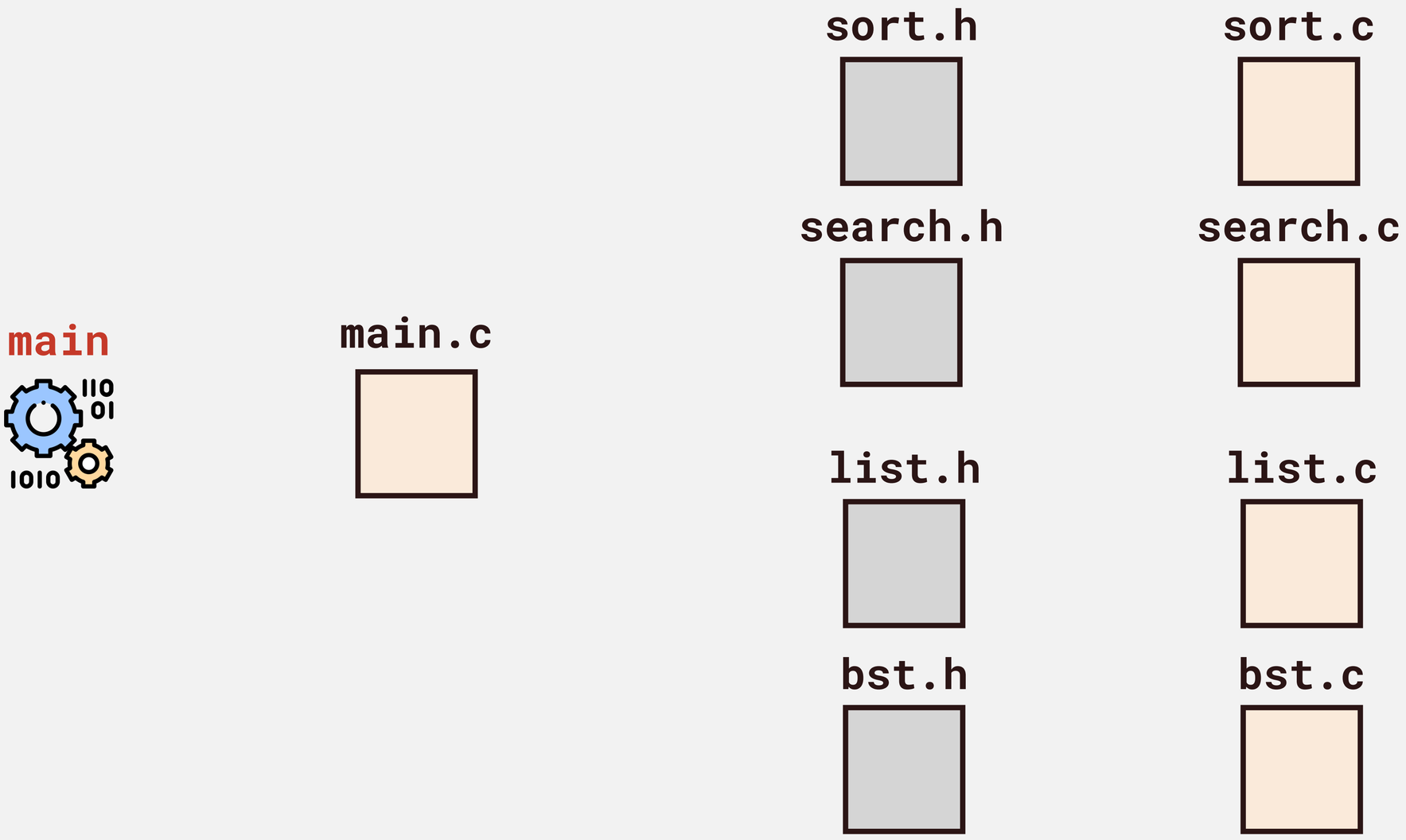
Source File

sort_util.c

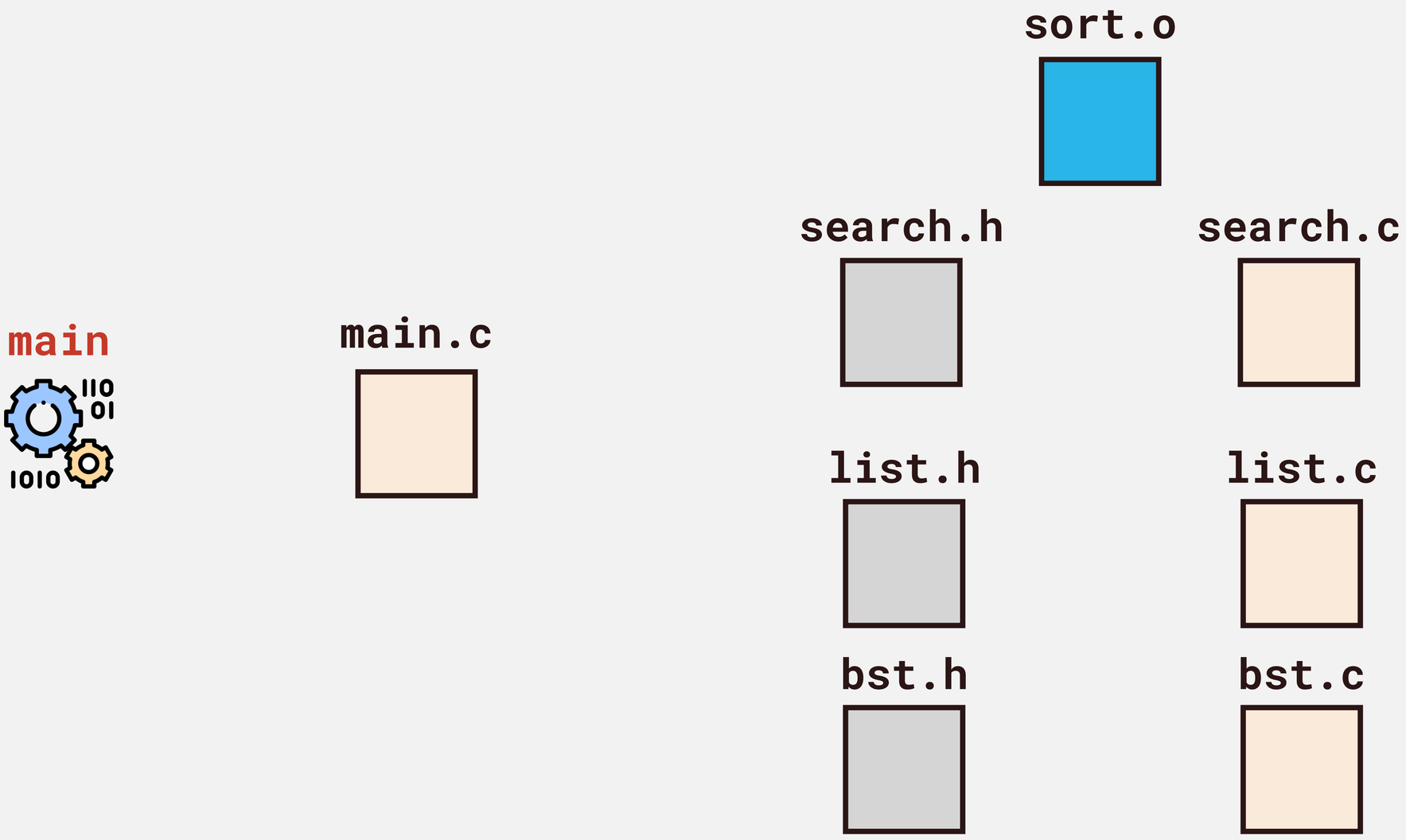
```
#include “sort_util.h”

#include <stdlib.h>

static void Swap(int *a, int i, int j) {
  ...
}

void Quicksort(int *a, int l, int r) {
  ...
}

void Mergesort(int *a, int l, int r) {
  ...
}
```

← Function is invisible outside the file

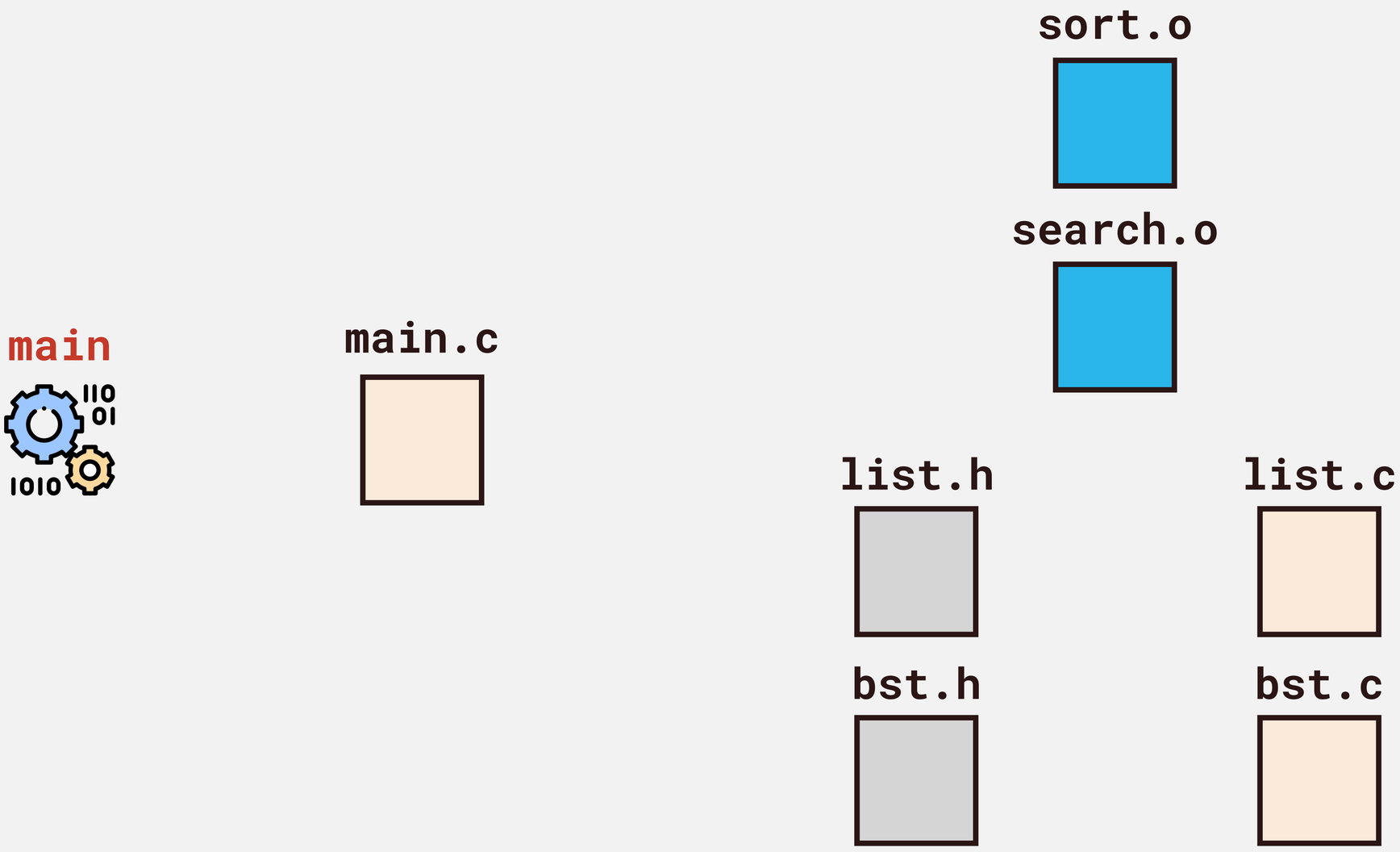
Multi-File C Program



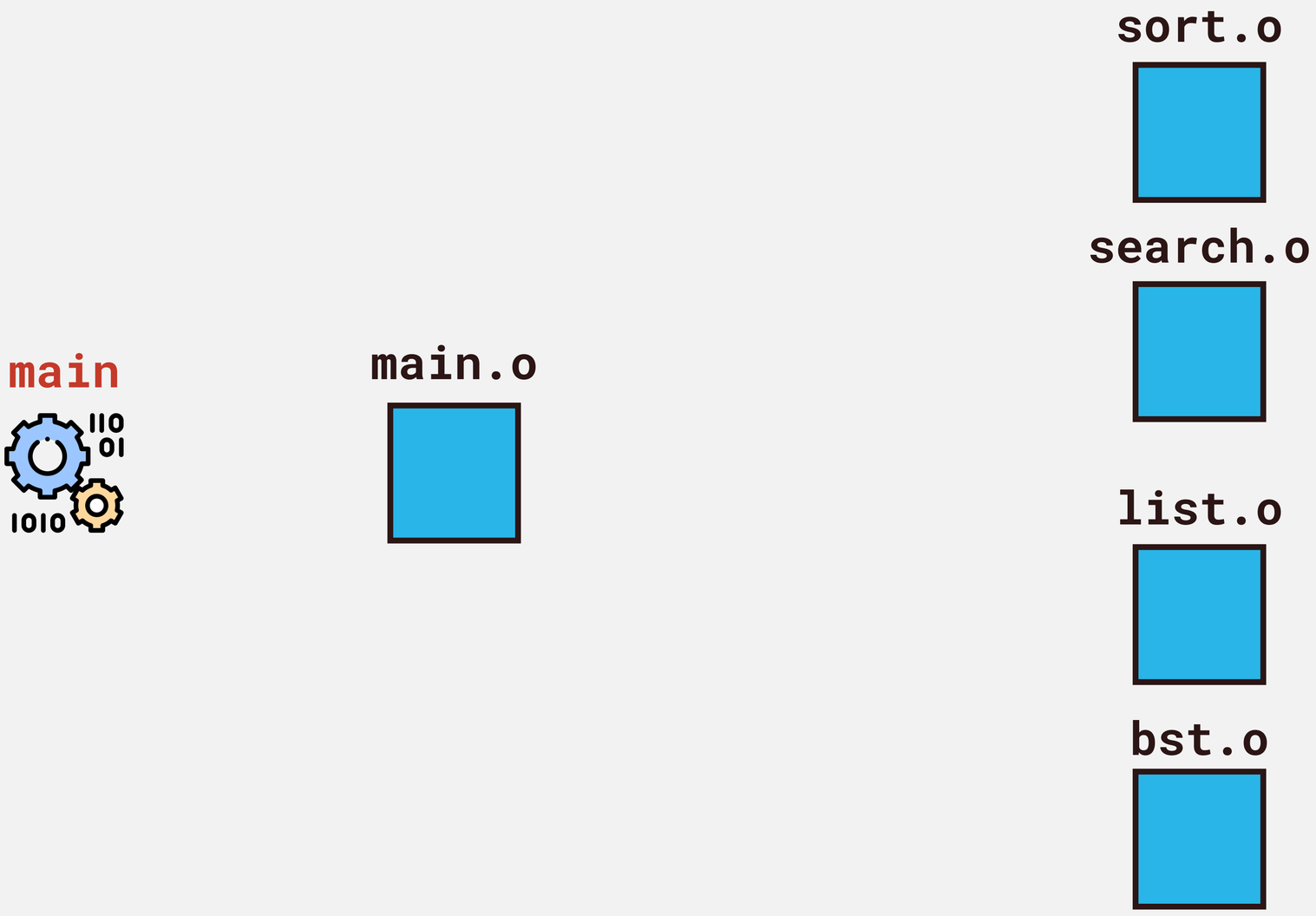
Multi-File C Program



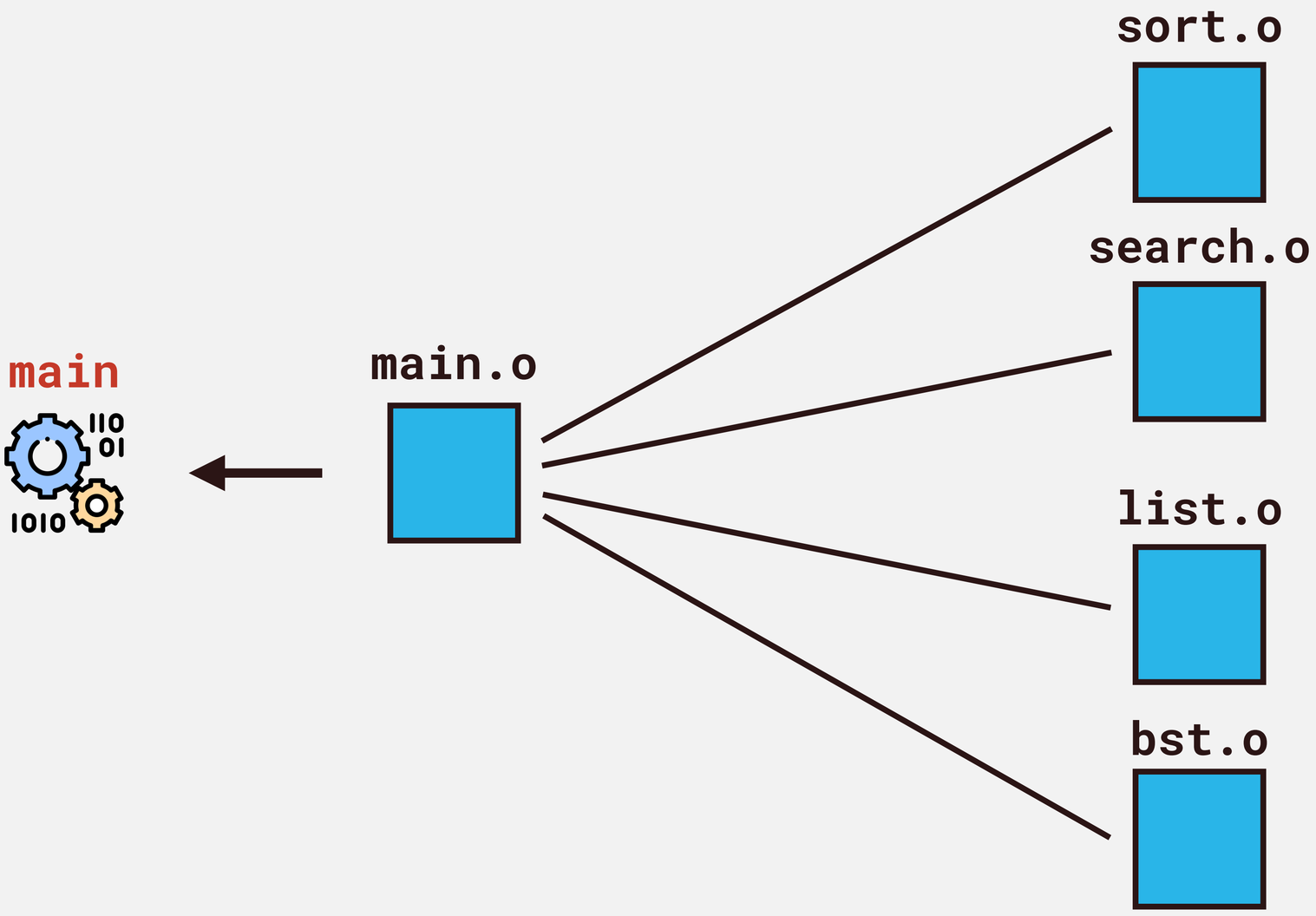
Multi-File C Program



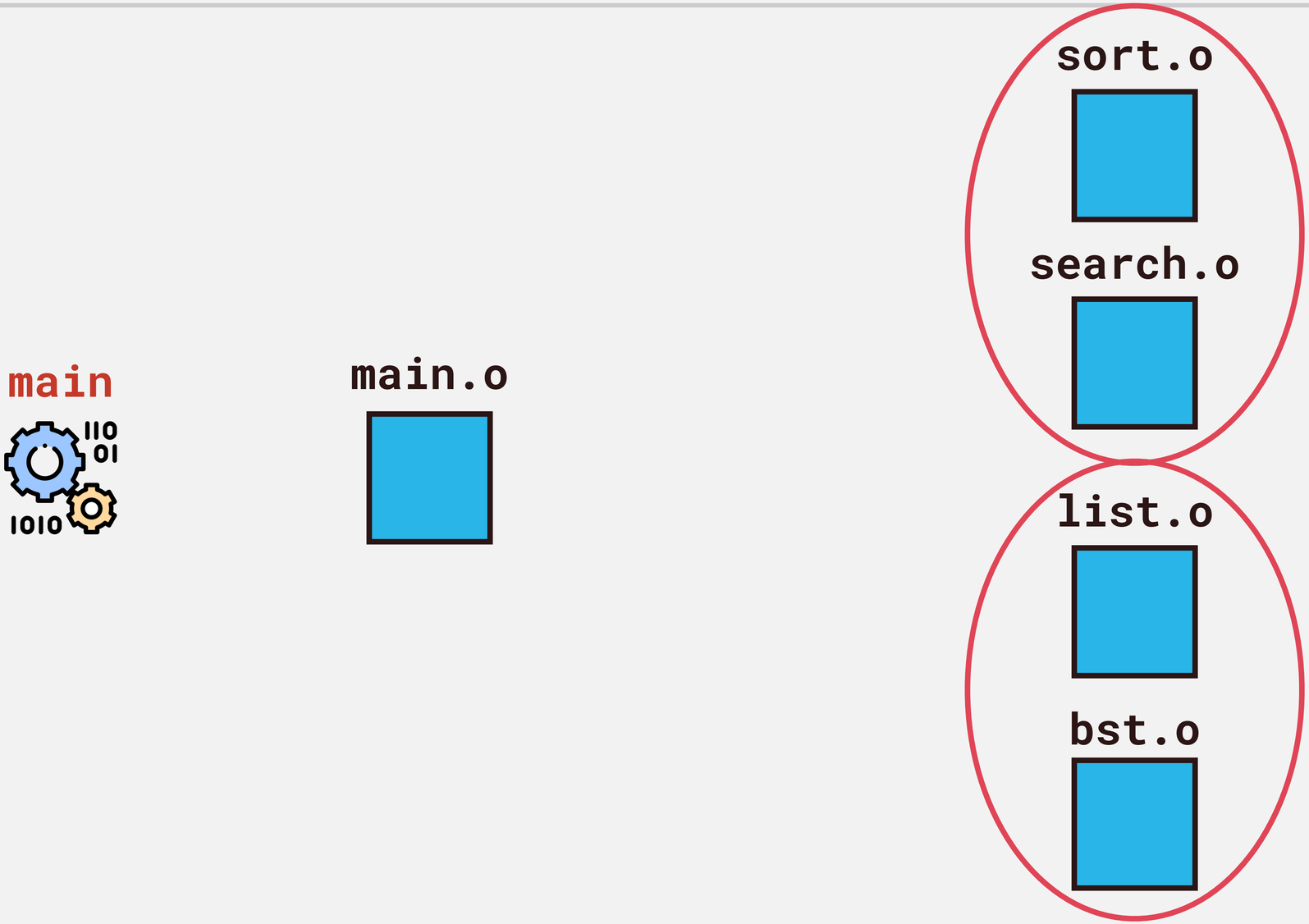
Multi-File C Program



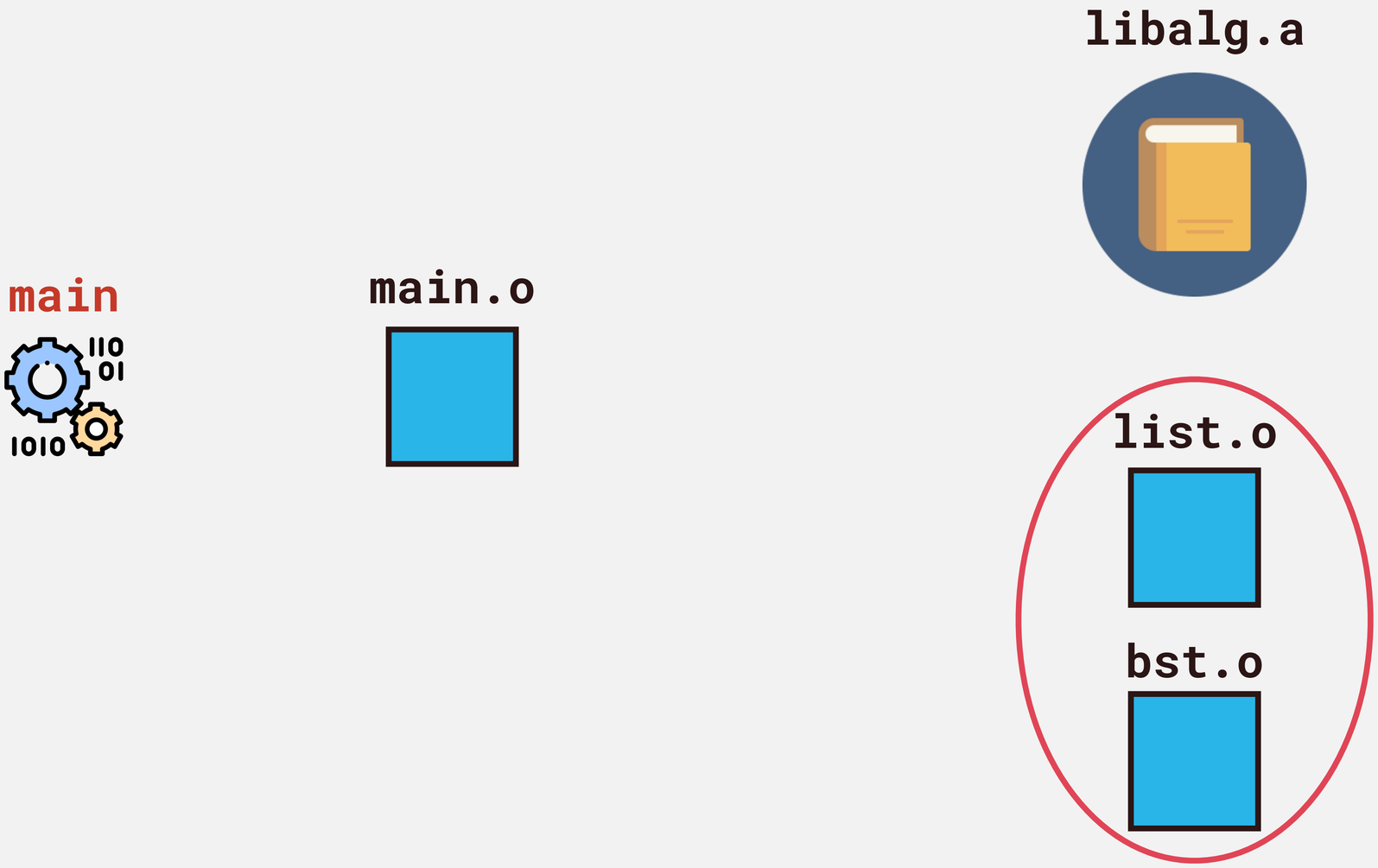
Multi-File C Program



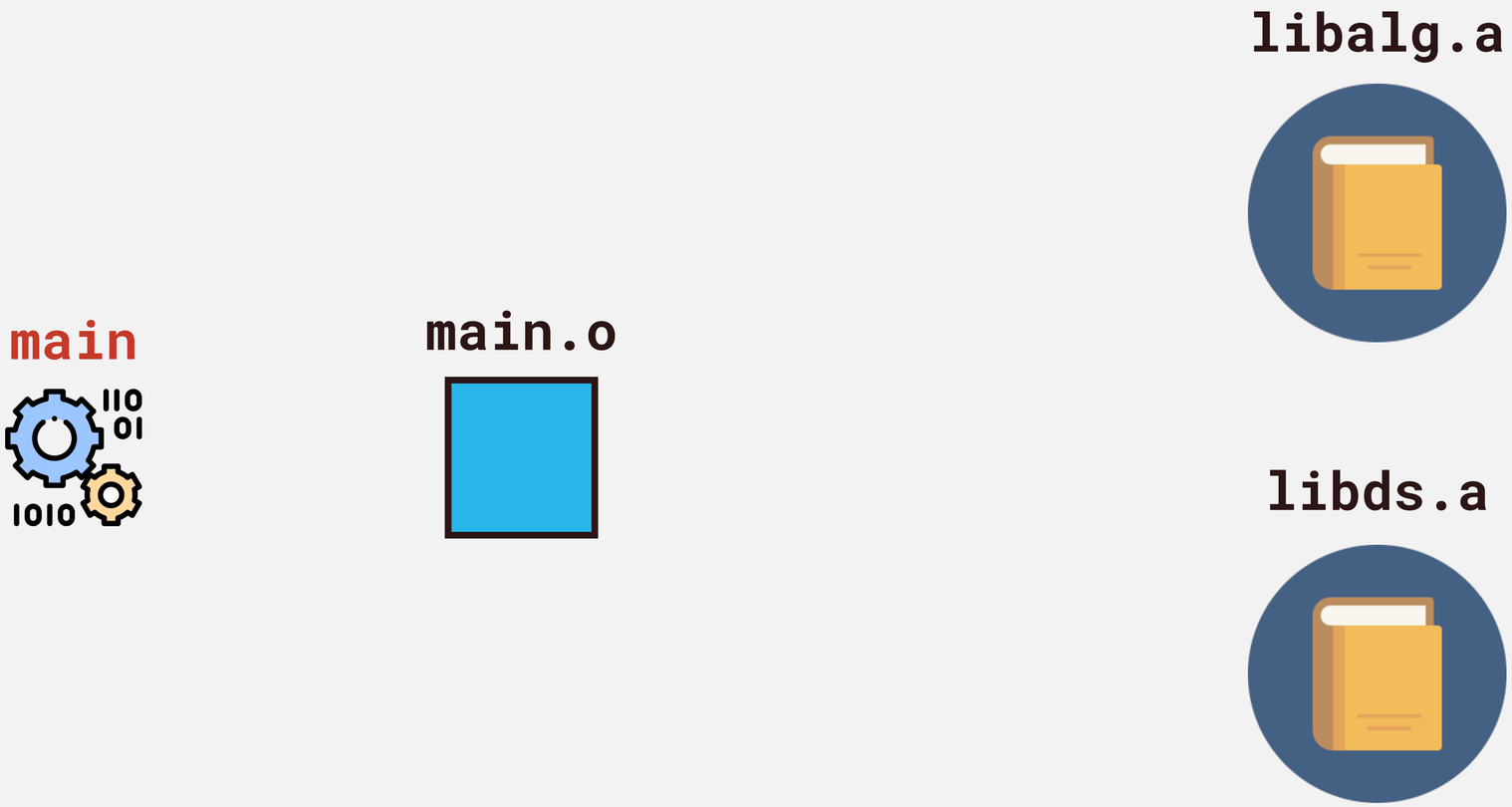
Libraries



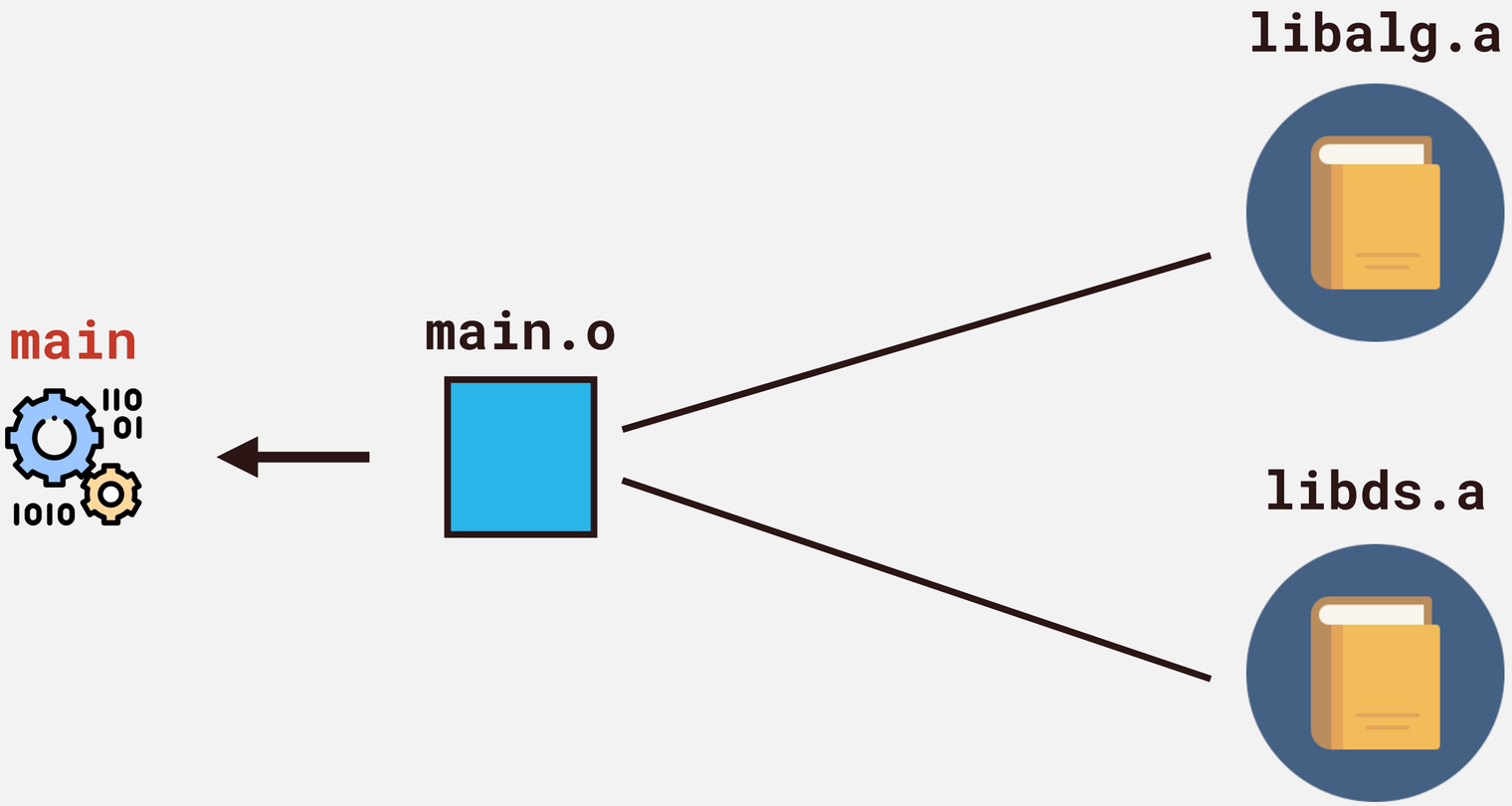
Libraries



Libraries



Libraries



Why Libraries?

prog1.c

prog2.c

sort.h

sort.c

search.h

search.c

list.h

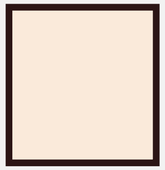
list.c

bst.h

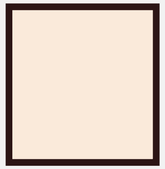
bst.c

Why Libraries?

prog1.c



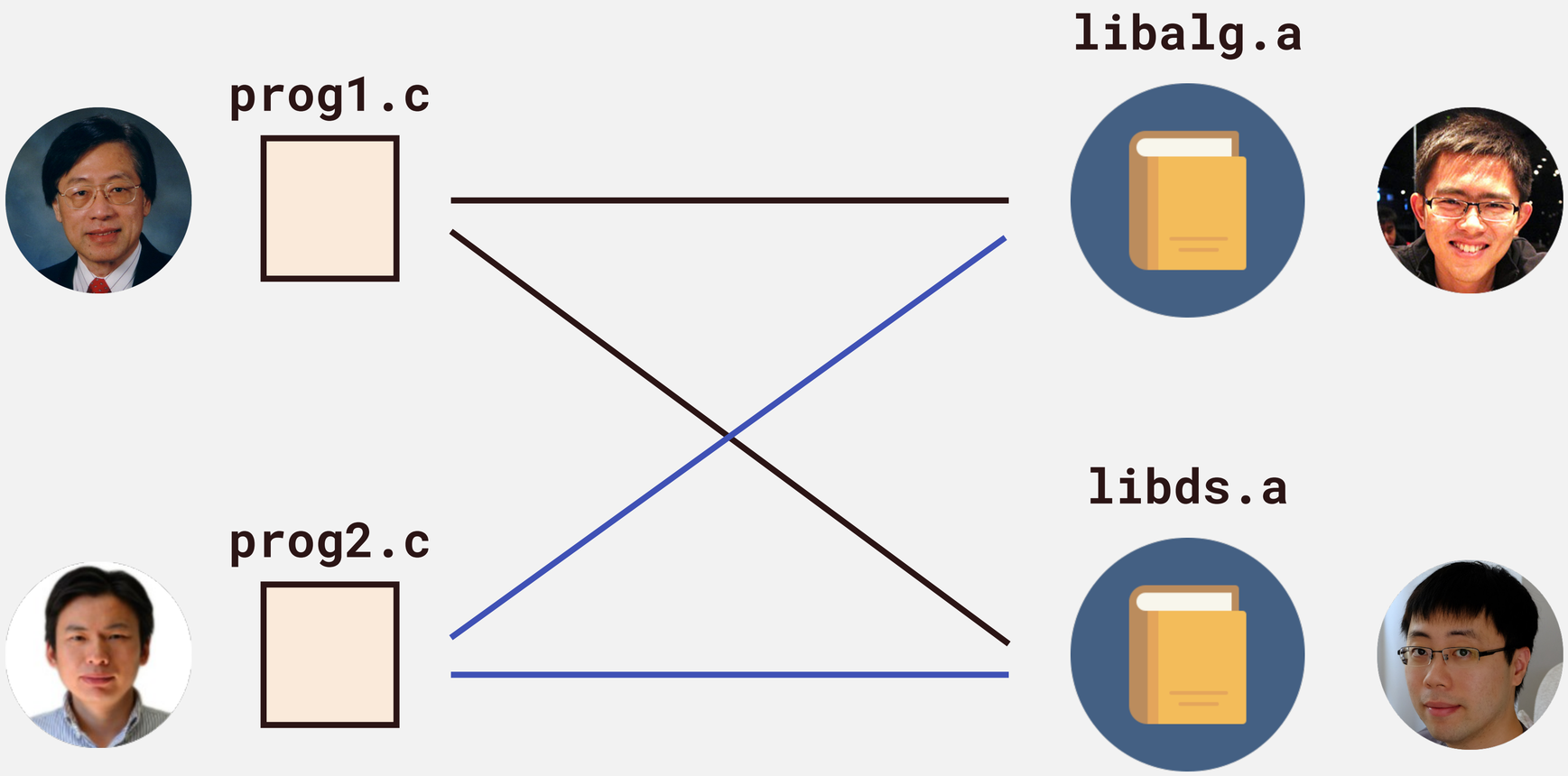
prog2.c



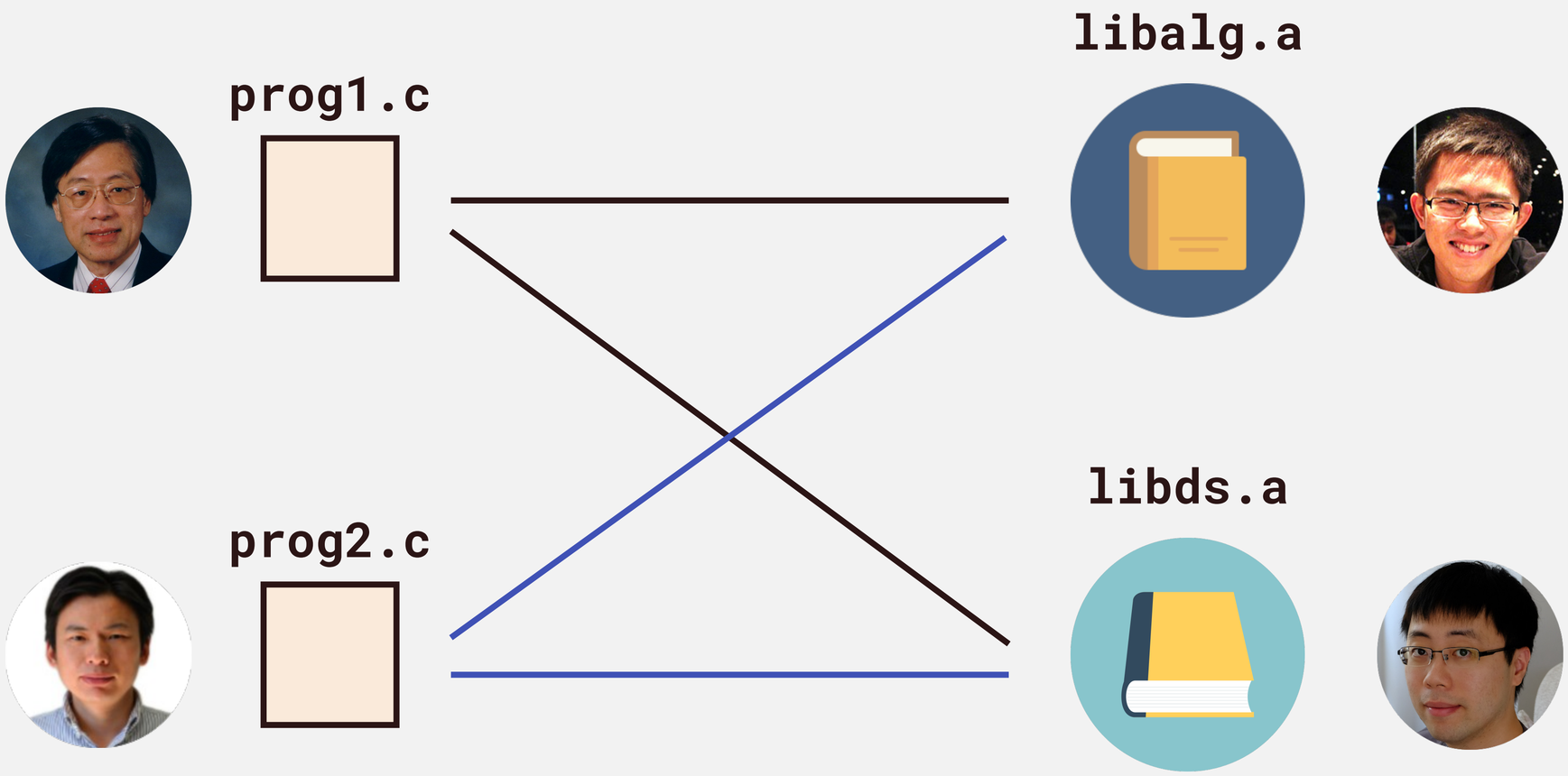
libalg.a

libds.a

Why Libraries?



Why Libraries?



Why Libraries?

- ➔ Speeds up compilation
- ➔ Facilitates collaboration and code reuse

Static Library

libfoo.a

```
void f() {
  ...
}
```

prog1.o

```
int main() {
  ...
  f();
}
```

prog2.o

```
int main() {
  f();
  …
}
```

Static Library

prog1

```
int main() {
  ...
  f();
}
```

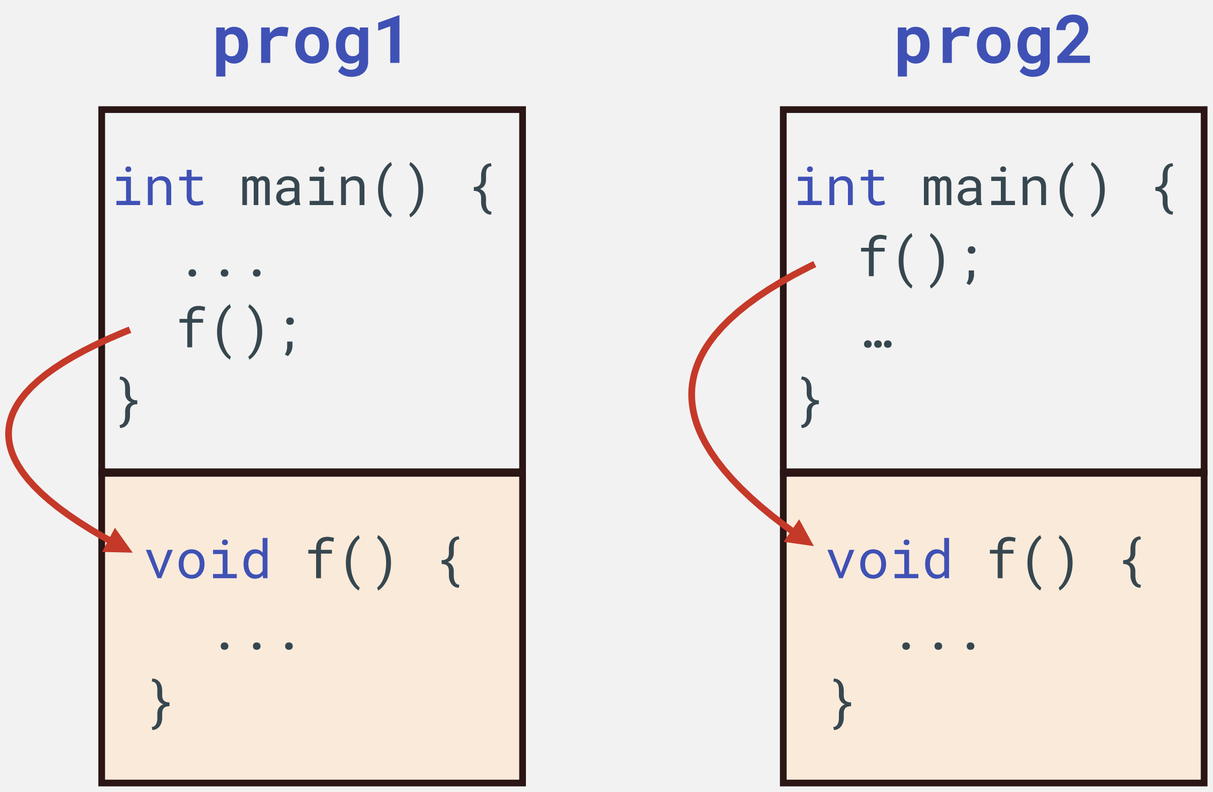
```
void f() {
  ...
}
```

prog2

```
int main() {
  f();
  …
}
```

```
void f() {
  ...
}
```

Static Library



Shared Library

libfoo.so

```
void f() {
  ...
}
```

prog1.o

```
int main() {
  ...
  f();
}
```

prog2.o

```
int main() {
  f();
  …
}
```

Shared Library

libfoo.so

```
void f() {
   ...
}
```

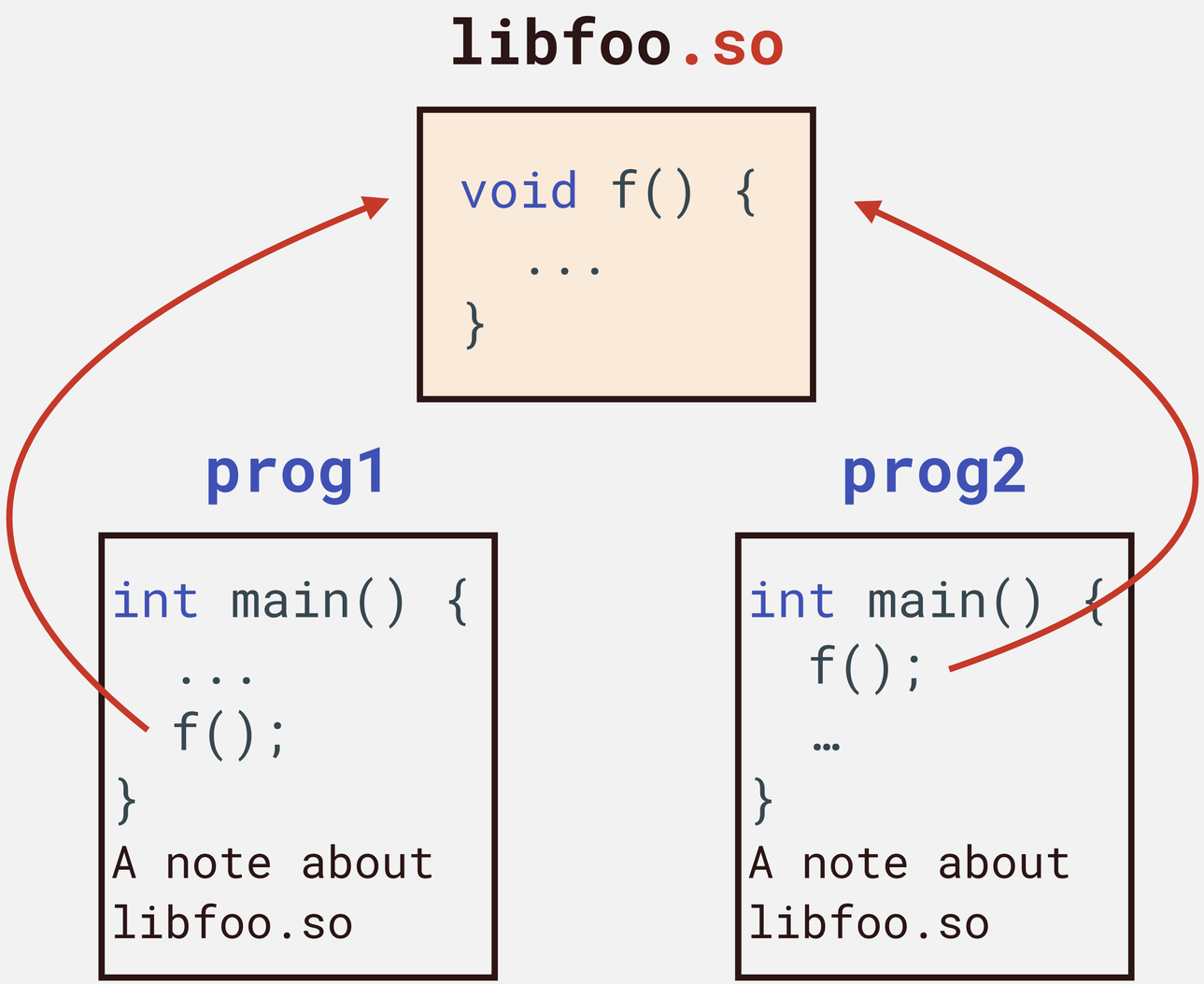
prog1

```
int main() {
   ...
   f();
}
A note about
libfoo.so
```

prog2

```
int main() {
   f();
   …
}
A note about
libfoo.so
```

Shared Library

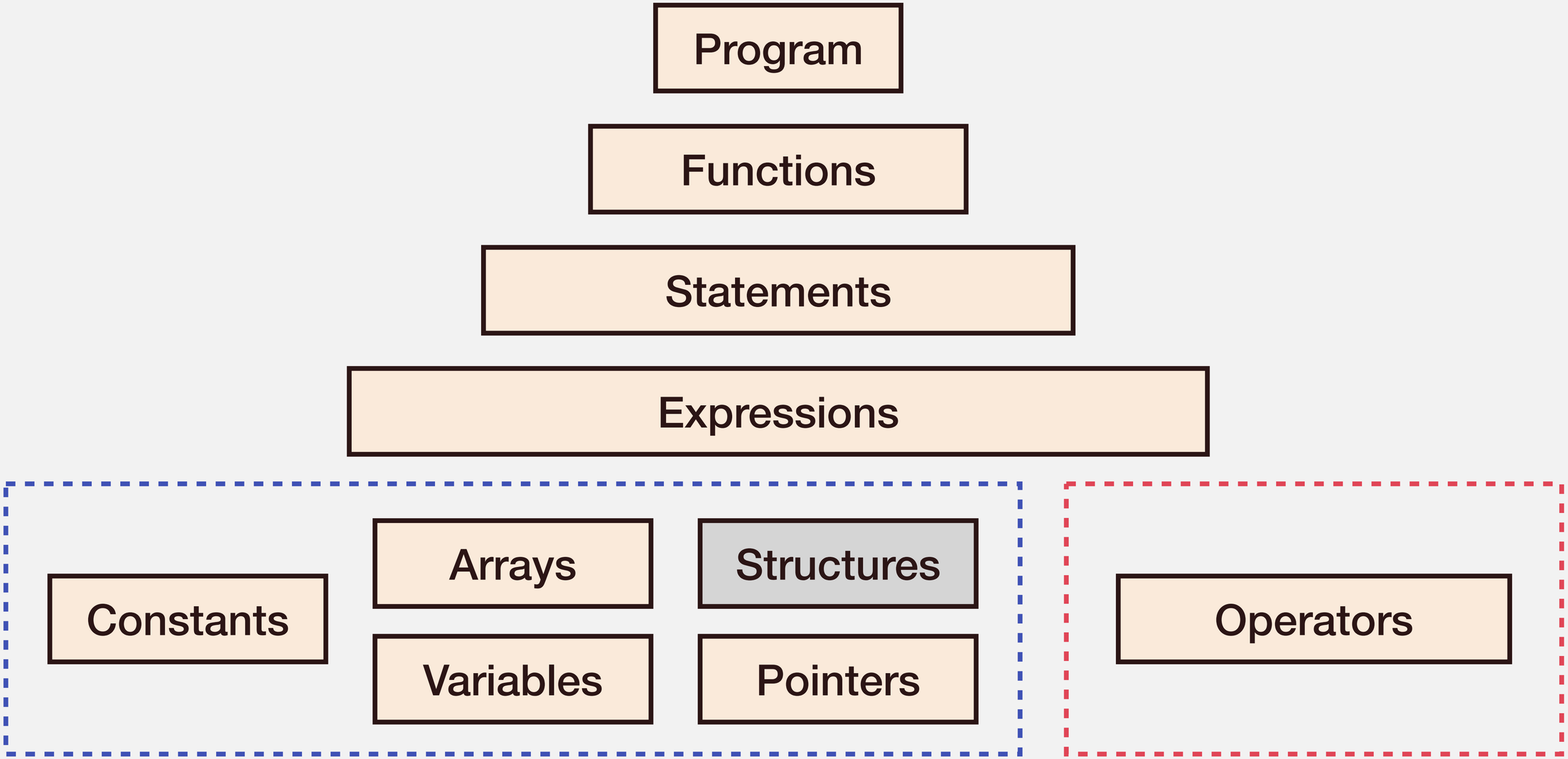


Use CMake

Summary

- **C String**
  - `char *`
- **Multi-Dimensional Array & Pointer Array**
  - `int a[M][N], int a[][N], int *a[M], int **a`
- **Recursion**
  - Divide and Conquer, quicksort
- **Multi-File C Program**
  - Header file, source file
  - Static/Shared library
  - Learn to use **CMake**

Road Map


Program
Functions
Statements
Expressions
Constants
Arrays
Variables
Structures
Pointers
Operators