CSE251: System Programming

6. Assembly (3)

Seongil Wi

Department of Computer Science and Engineering

Notification: HW1

- Print the input data by using the various data types

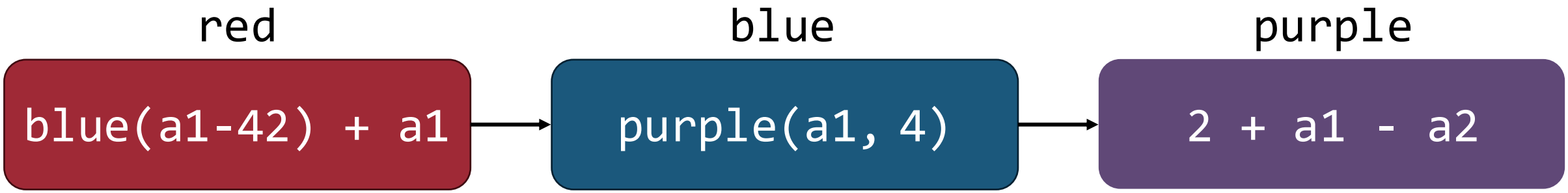
- Due March 24, 2026, 11:59 pm
- To access your assigned server, please read the notice “Guide to SSH Access and Linux Basics” on Blackboard.

- Responsible TA: Donggyu Kim (dgkim@unist.ac.kr)
- If you have any requests or questions, ask the TAs via Blackboard.
- Late submission will be assessed a penalty of 10% per day
  - We will only accept late submissions of up to 3 days

Notification: HW1

- There may be occasional connectivity issues after the due date as adjustments will be made for the HW2 setup

Recap: Questions



- How do we pass function parameters?
- When a function returns, how do we restore the register values of the caller
- Where do we store local variables?

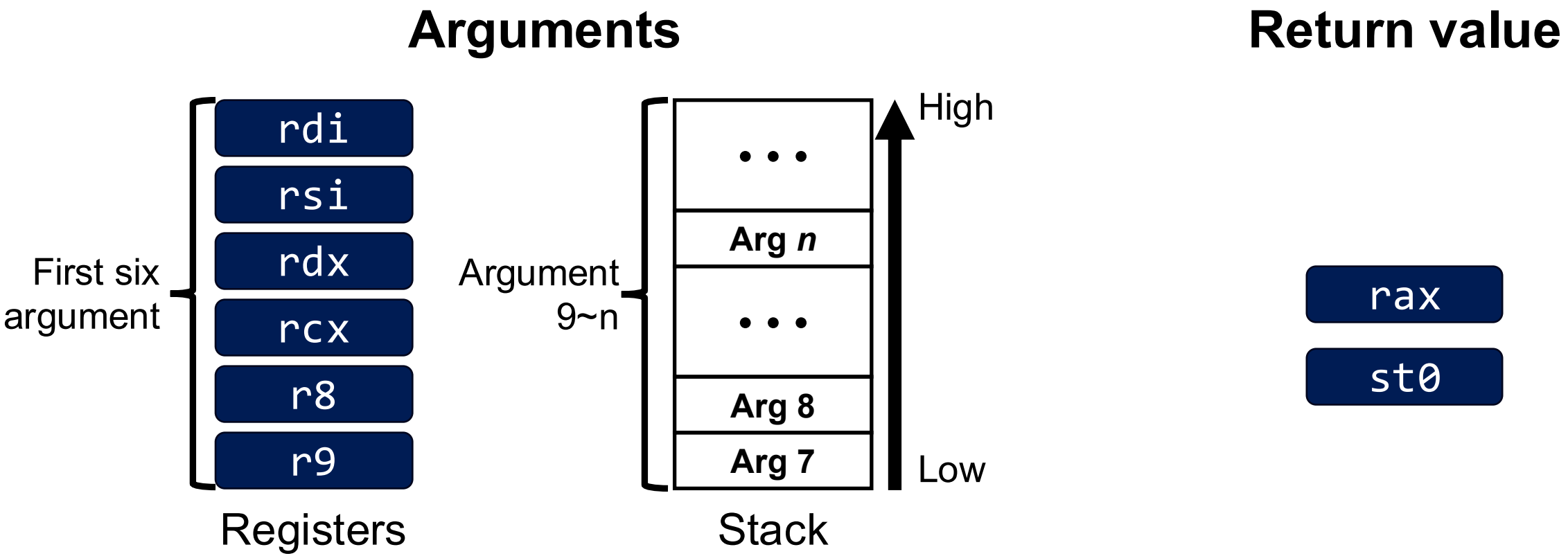
X86-64 Calling Conventions

Calling Conventions

- Dictates how a function call is implemented at the assembly level
  - How parameters are passed (i.e., through registers or stack), in which order
  - Which registers are preserved across function calls

x86-64 Parameter Passing

- First six arguments are passed through `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`
- The seventh and subsequent arguments are passed through the stack
- The return value is passed through `rax` (or `st0` for a FP value)



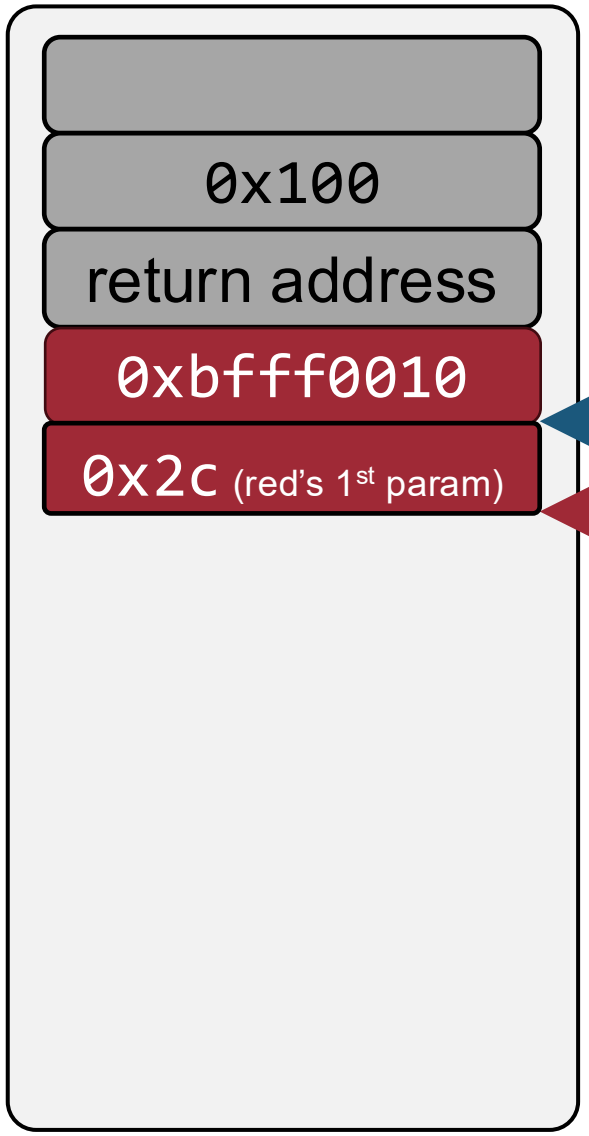
x86-64 Parameter Passing (Example)

- First six arguments are passed through `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`
- The seventh and subsequent arguments are passed through the stack
- The return value is passed through `rax` (or `st0` for a FP value)

```
foo(1, 2, 3, 4, 5, 6, 7);
```

```
push 7
mov r9, 6
mov r8, 5
mov rcx, 4
mov rdx, 3
mov rsi, 2
mov rdi, 1
call foo
```

Recap: Execution Example



0xbfff0010

0xbffefff8

0xbffefff0

```
long red(long a1) {
    return blue(a1 - 42) + a1;
}
```

Prepare rdi (the first parameter) for blue( )

```
rip: 0x117d
rbp: 0xbffefff8
rsp: 0xbffefff0
rdi: 0x2
rax: 0x2
```

Execution context

```
<red>:
  116
  11                                  0x8
  11                                  RD PTR [rbp-0x8], rdi
  1172: mov   rax, QWORD PTR [rbp-0x8]
  1176: sub   rax, 0x2a
  117a: mov   rdi, rax
  117d: call  blue
  117e: mov   rdx, QWORD PTR [rbp-0x8]
  1182: add   rax, rdx
  1185: leave
  1186: ret

<blue>:
  1143: push  rbp
  1144: mov   rbp, rsp
  ...
  1160: leave
  1161: ret
```

Calling Conventions

- Dictates how a function call is implemented at the assembly level
  - How parameters are passed (i.e., through registers or stack), in which order
  - Which registers are preserved across function calls

Calling Conventions

- Dictates how a function call is implemented at the assembly level
  - How parameters are passed (i.e., through registers or stack), in which order
  - Which registers are preserved across function calls

Motivation

- When the function `red` calls `blue`:
  - `red` is the **caller**
  - `blue` it the **callee**

Q. What would happen if there were no register saving convention?

```
red:
    ...
    mov rdx, 15213
    call blue
    add rax, rdx
    ...
    ret
```

```
blue:
    ...
    ...
    mov rdx, 18213
    ...
    ...
    ret
```

Motivation

- When the function `red` calls `blue`:
  - `red` is the **caller**
  - `blue` it the **callee**

Q. What would happen if there were no register saving convention?

```
red:
    ...
    mov rdx, 15213
    call blue
    add rax, rdx
    ...
    ret
```

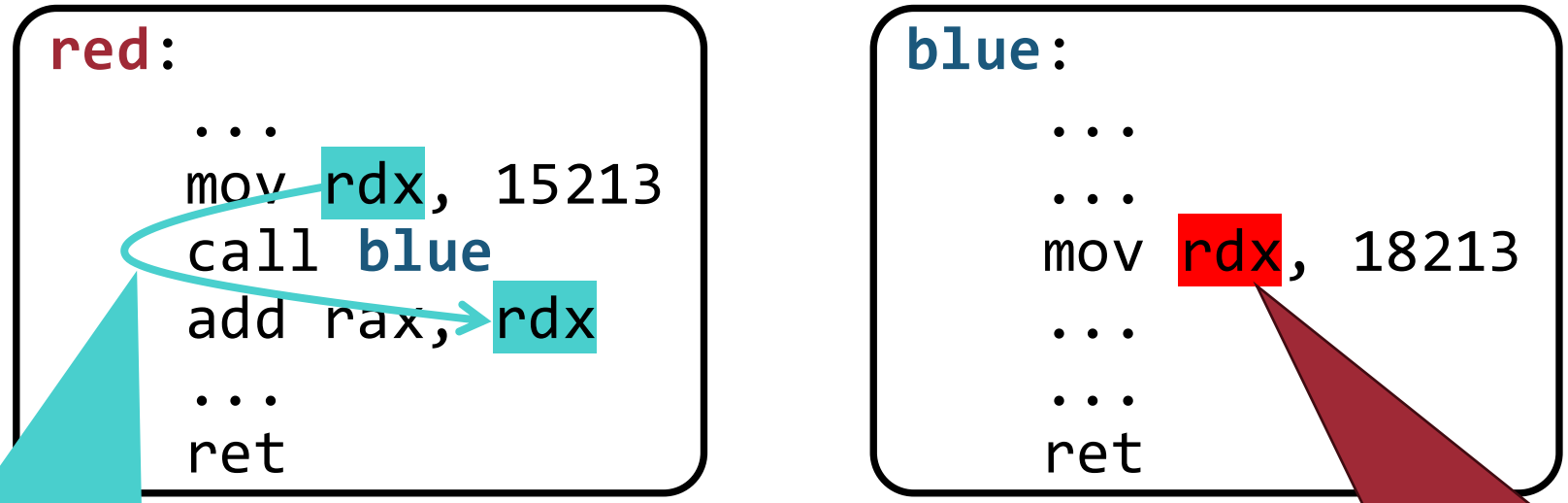
```
blue:
    ...
    ...
    mov rdx, 18213
    ...
    ...
    ret
```

Intention: add 15213 to `rax`

Motivation

- When the function `red` calls `blue`:
  - `red` is the **caller**
  - `blue` it the **callee**

Q. What would happen if there were no register saving convention?



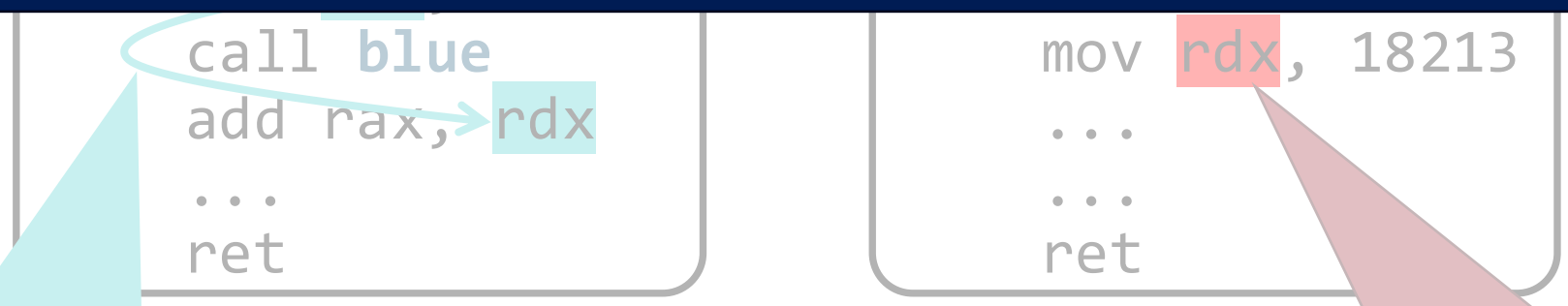
Intention: add 15213 to `rax`

Contents of `rdx` overwritten by `blue`

Motivation

- When the function `red` calls `blue`:
  - `red` is the **caller**
  - `blue` it the **callee**

This could be trouble.
We need some coordination (convention)



Intention: add 15213 to `rax`

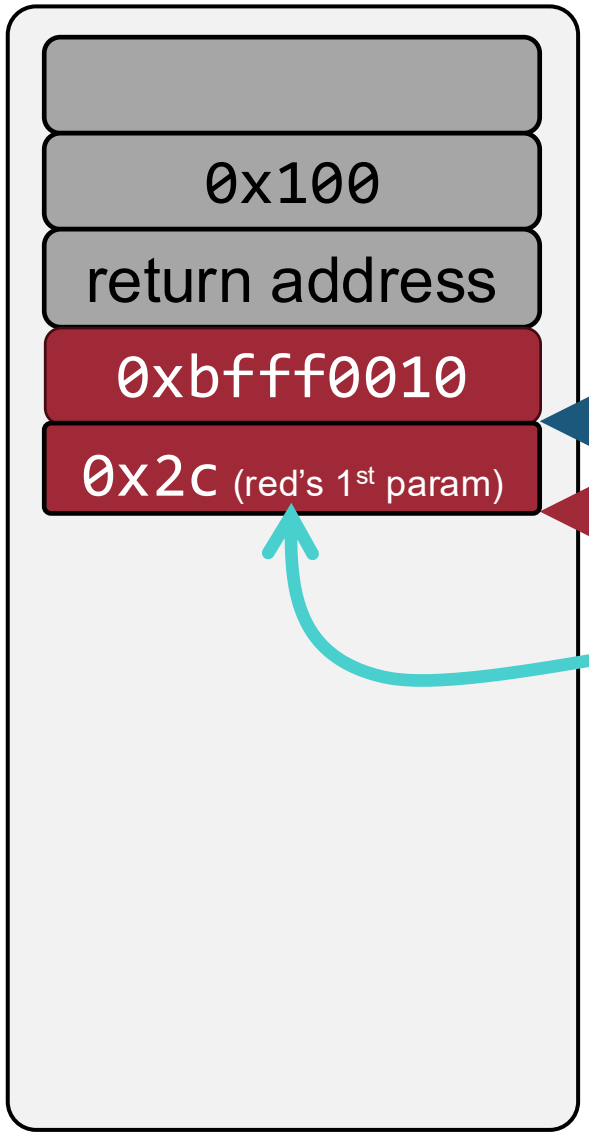
Contents of `rdx` overwritten by `blue`

Register Saving Conventions

- "***Caller-saved***" registers
  - *The caller* saves the values in its stack frame before the call (if it is necessary)
    - *The callee* is free to overwrite these registers
  - E.g., `rdi`

- "***Callee-saved***" registers

Caller (red) Saves rdi in its Stack Frame



0xbfff0010

```
long red(long a1) {
    return blue(a1 - 42) + a;
}
```

0xbffefff8

0xbffefff0

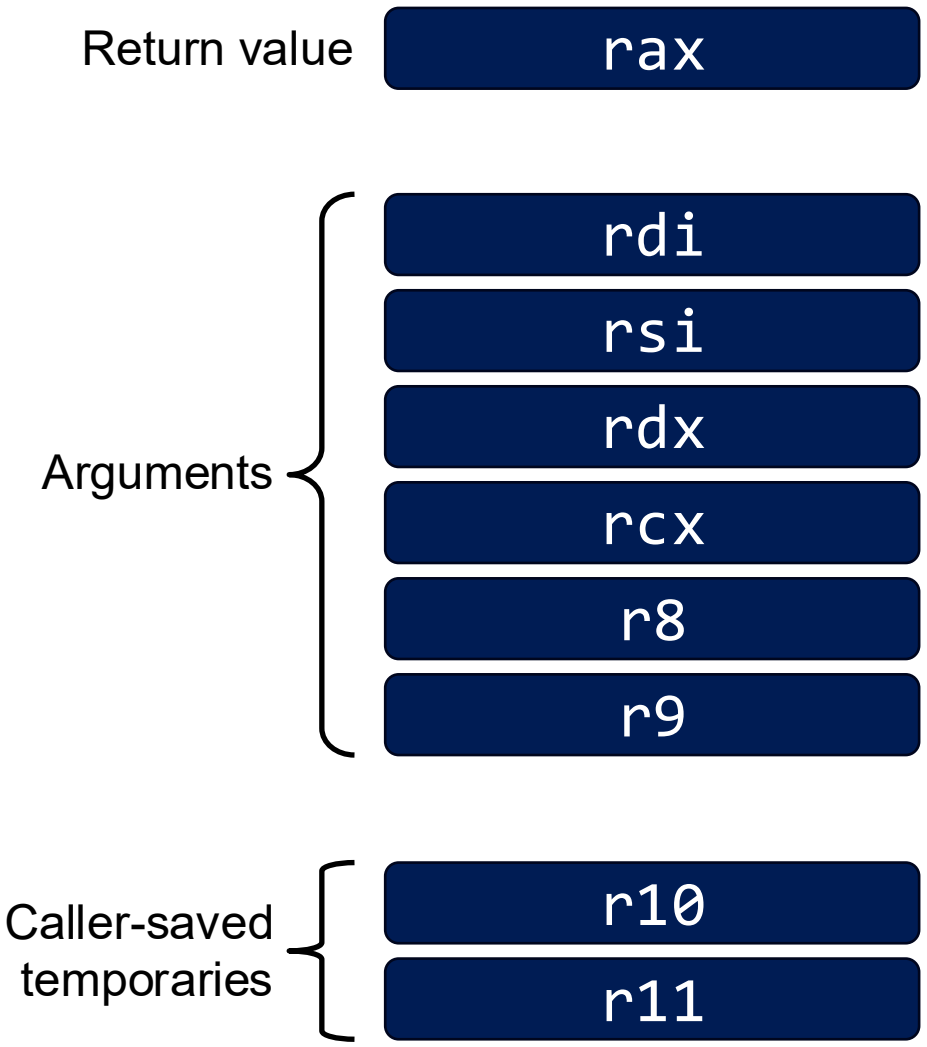
```
rip: 0x1172
rbp: 0xbffefff8
rsp: 0xbffefff0
rdi: 0x2c
```

Execution context

```
<red>:
                                  rsp
                                  0x8
  116e: mov   QWORD PTR [rbp-0x8], rdi
  1172: mov   rax, QWORD PTR [rbp-0x8]
  1176: sub   rax, 0x2a
  117a: mov   rdi, rax
  117d: call  blue
  117e: mov   rdx, QWORD PTR [rbp-0x8]
  1182: add   rax, rdx
  1185: leave
  1186: ret

<blue>:
  1143: push   rbp
  1144: mov    rbp, rsp
  ...
  1160: leave
  1161: ret
```

x86-64 Caller-saved Registers

- `rax`
  - Return value
  - Also caller-saved
  - Can be modified by procedure
- `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`
  - Arguments
  - Also caller-saved
  - Can be modified by procedure
- `r10`, `r11`
  - Caller-saved
  - Can be modified by procedure

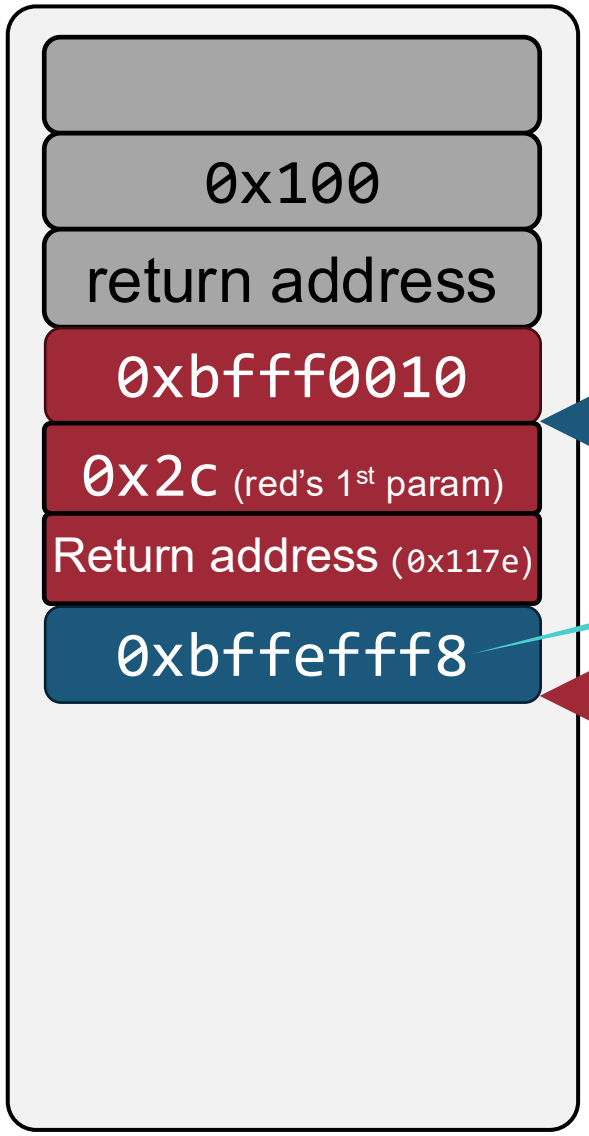


Register Saving Conventions

- "***Caller-saved***" registers
  - *The caller* saves the values in its stack frame before the call (if it is necessary)
    - *The callee* is free to overwrite these registers
  - E.g., `rdi`

- "***Callee-saved***" registers
  - *The callee* is responsible for preserving the value
    - *The callee* saves temporary values in its frame before using
    - *The callee* restores them before returning to caller
  - E.g., `rbp`

Callee (blue) Saves rbp in its Stack Frame



0xbfff0010

0xbffefff8

0xbffefff0

0xbffeffe8

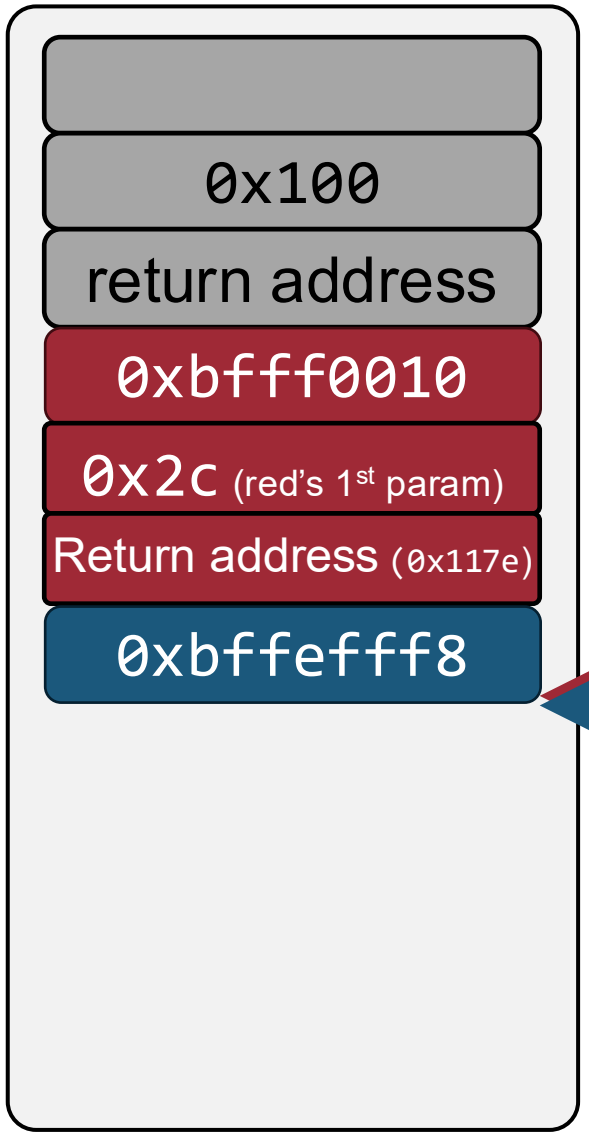
rbp's value

```
rip: 0x1144
rbp: 0xbffefff8
rsp: 0xbffeffe0
rdi: 0x2
rax: 0x2
```

Execution context

```
<red>:
   1166: push rbp
   1167: mov  rbp, rsp
   116a: sub  rsp, 0x8
   116e: mov  QWORD PTR [rbp-0x8], rdi
   1172: mov  rax, QWORD PTR [rbp-0x8]
   1176: sub  rax, 0x2a
   1179: mov  rdi, rax
   117c: call blue
   117e: mov  rdx, QWORD PTR [rbp-0x8]
   1182: add  rax, rdx
   1185: leave
   1186: ret

<blue>:
   1143: push   rbp
   1144: mov    rbp, rsp
   ...
   1160: leave
   1161: ret
```

Now, rbp can be Overwritten by blue



Virtual memory

0xbfff0010

0xbffefff8

0xbffefff0

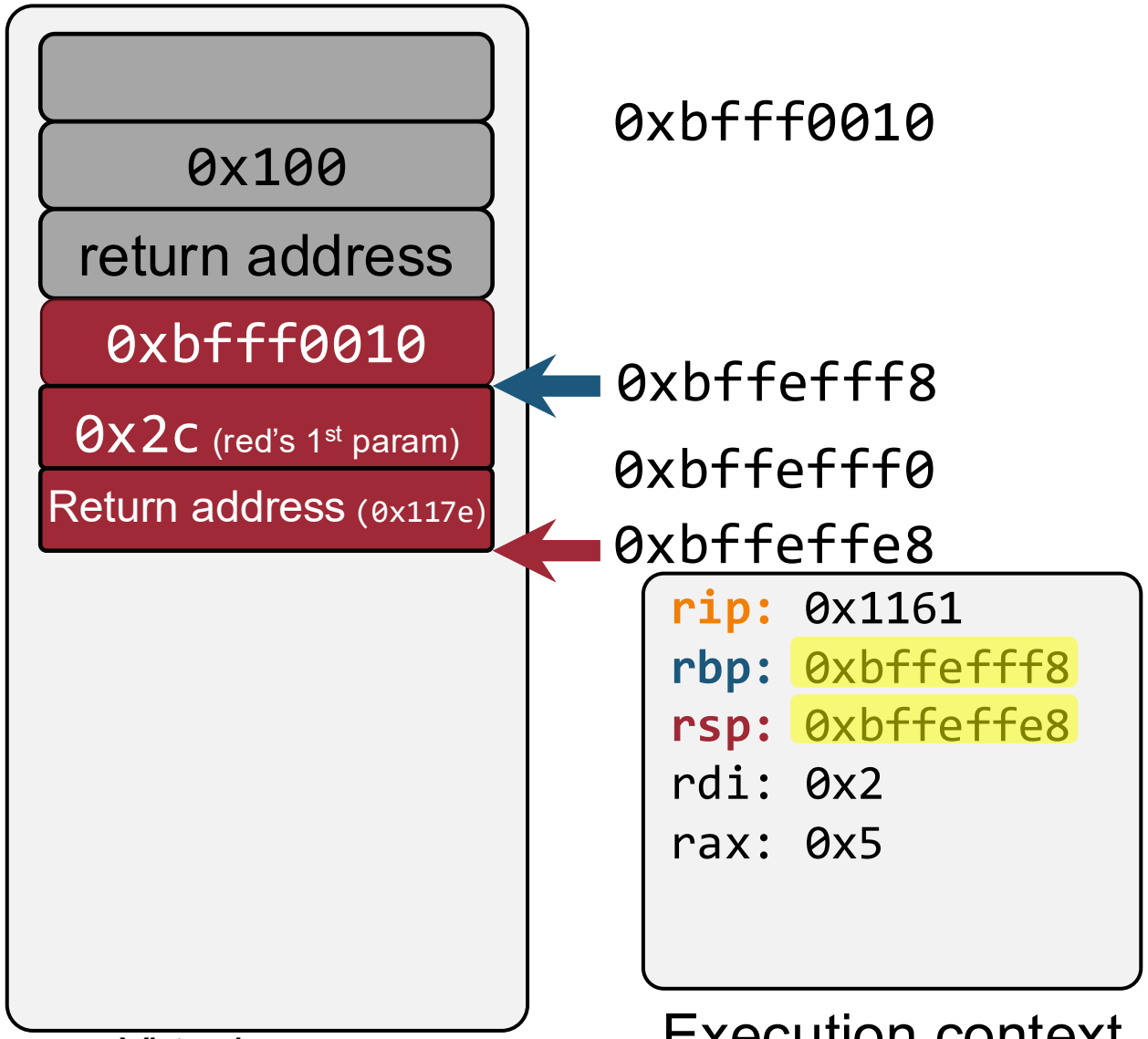
0xbffeffe8

```
rip: 0x1147
rbp: 0xbffeffe0
rsp: 0xbffeffe0
rdi: 0x2
rax: 0x2
```

Execution context

```
<red>:
  1166: push rbp
  1167: mov  rbp, rsp
  116a: sub  rsp, 0x8
  116e: mov  QWORD PTR [rbp-0x8], rdi
  1172: mov  rax, QWORD PTR [rbp-0x8]
  1176: sub  rax, 0x2a
  117a: mov  rdi, rax
  117d: call blue
  117e: mov  rdx, QWORD PTR [rbp-0x8]
  1182: add  rax, rdx
  1185: leave
  1186: ret

<blue>:
  1143: push   rbp
  1144: mov    rbp, rsp
  ...
  1160: leave
  1161: ret
```

Callee (blue) Restores rbp value

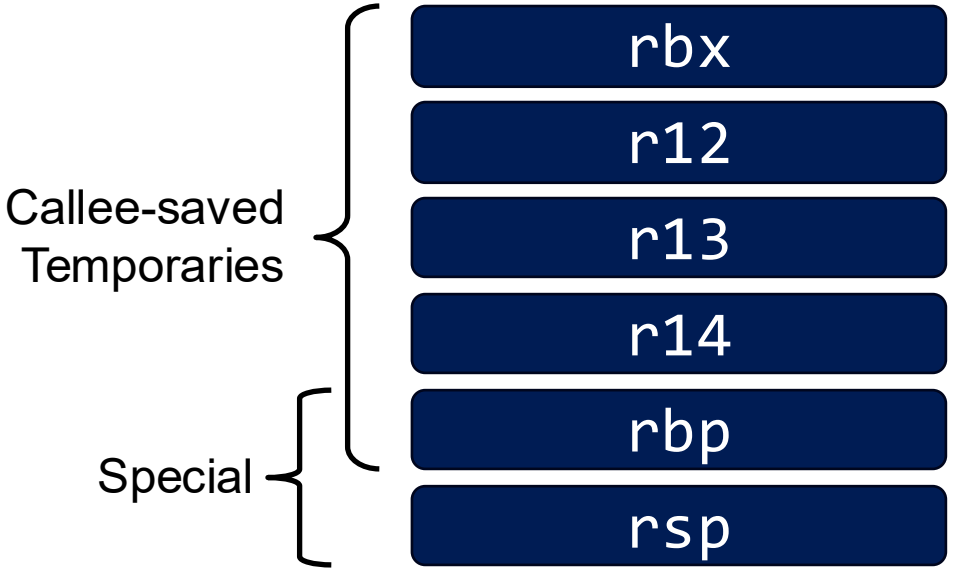



```
<red>:
   1166: push rbp
   1167: mov  rbp, rsp
   116a: sub  rsp, 0x8
   116e: mov  QWORD PTR [rbp-0x8], rdi
   1172: mov  rax, QWORD PTR [rbp-0x8]
   1176: sub  rax, 0x2a
   117a: mov  rdi, rax
   117d: call blue
   117e: mov  rdx, QWORD PTR [rbp-0x8]
   1182: add  rax, rdx
   1185: leave
   1186: ret

<blue>:
   1143: push   rbp
   1144: mov    rbp, rsp
   ...
   1160: leave
   1161: ret
```

mov rsp, rbp
pop rbp

x86-64 Callee-saved Registers

- `rbx`, `r12`, `r13`, `r14`
  - Callee-saved
  - Callee must save & restore
- `rbp`
  - Callee-saved
  - Callee must save & restore
  - May be used as frame pointer
  - Can mix & match
- `rsp`
  - Special form of callee save
  - Restored to original value upon exit from procedure



Register Saving Conventions

- "***Caller-saved***" registers
  - *The caller* saves the values in its stack frame before the call (if it is necessary)
    - *The callee* is free to overwrite these registers
  - E.g., `rdi`

- "***Callee-saved***" registers
  - *The callee* is responsible for preserving the value
    - *The callee* saves temporary values in its frame before using
    - *The callee* restores them before returning to caller
  - E.g., `rbp`

Note *A function can be a caller to some functions and a callee to others*

Structures

Structures (`struct`) in C

- Mechanism for grouping related data of different types

- Suppose we want to keep track of the data for the 100 students
  - For each day, we want the following data:

```
int id;
int birth_year;
int birth_month;
char letter_grade;
```

We can use a `struct` to group these data

Declaring a `struct`

- We name the `struct` and declare “members”

This is declaration so no memory is actually allocated!

```
struct student {
    int id;
    int birth_year;
    int birth_month;
    char letter_grade;
};
```

Defining and Using a `struct`

- We define a variable using our new data type

```
struct student alice = {20251234, 2006, 11, 'A'};
```

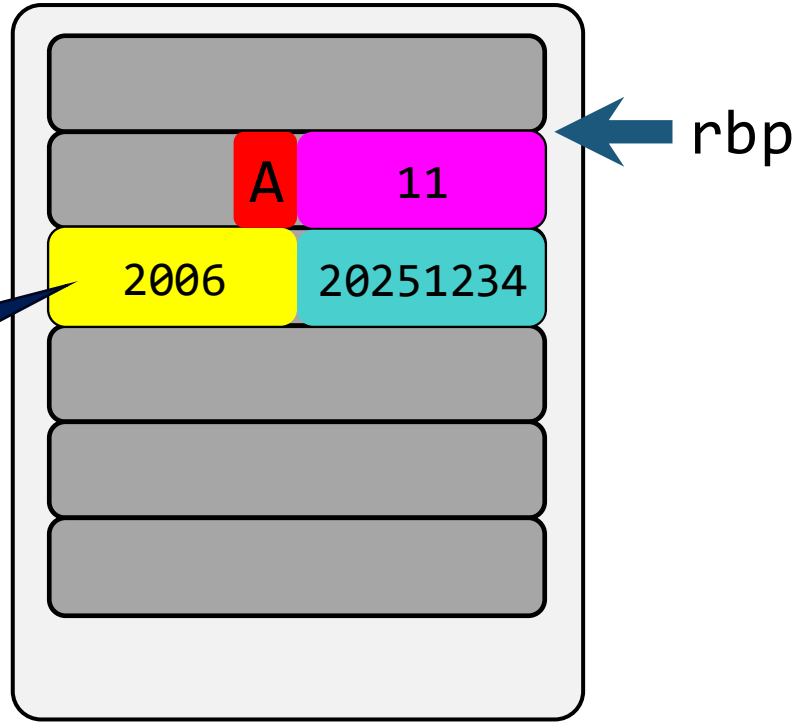
How is a structure represented in assembly and memory?

Structure Representation

- We define a variable using our new data type

```
struct student alice = {20251234, 2006, 11, 'A'};
```

```
mov DWORD PTR [rbp-0x10],0x1350262
mov DWORD PTR [rbp-0xc],0x7d6
mov DWORD PTR [rbp-0x8],0xb
mov BYTE PTR [rbp-0x4],0x41
```



- Structure represented as block of memory
- Fields ordered according to declaration

Defining and Using a `struct`


```
struct student alice = {20251234, 2006, 11, 'A’};
alice.id = 20241234;
```

We can access individual fields via
. (dot) operator

Pointers to Struct

- We can define and create a pointer to a `struct`

```
struct *student stuPTR;
stuPtr = &alice;
```

- To access a member of the `struct` addressed by `stuPtr`

```
(*stuPTR).id = 20241234;
```

*. (dot) operator has higher precedence than **

- Special syntax (`->`) for this common access pattern

```
stuPTR -> id = 20241234;
```

`(*stuPTR).id` = `stuPTR -> id`

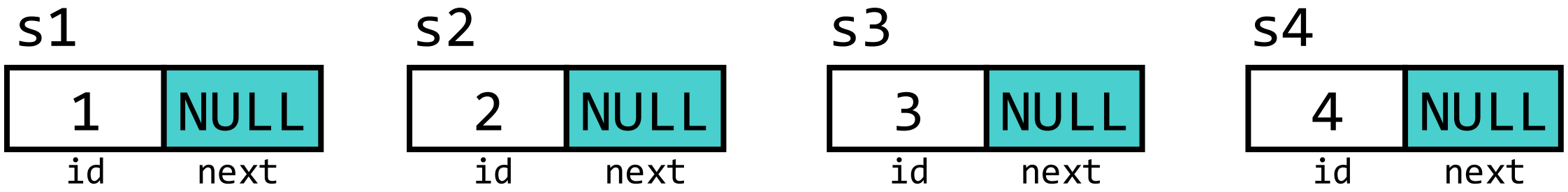
Linked List

- A linear collection of data elements
  - Similar with array
  - However, it's order is not given by their physical placement in memory
  - Instead, each element points to the next

Linked List

```
typedef struct student {
    int id;
    struct student *next;
} Student;
```

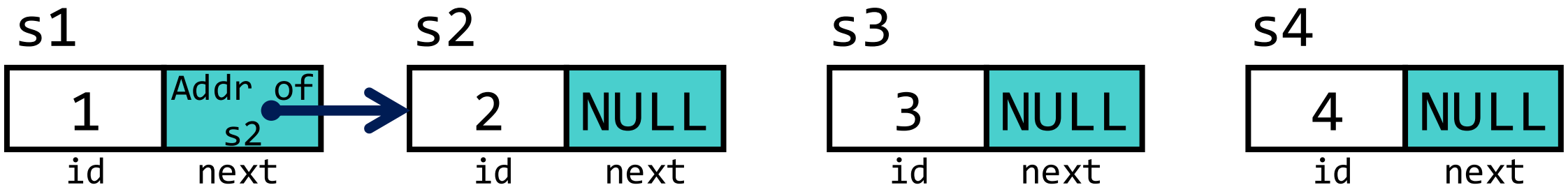
```
Student s1 = {1, NULL};
Student s2 = {2, NULL};
Student s3 = {3, NULL};
Student s4 = {4, NULL};
```



Linked List

```
typedef struct student {
    int id;
    struct student *next;
} Student;
```

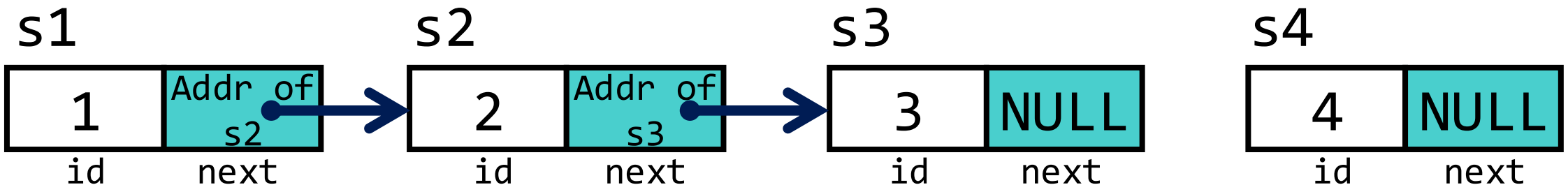
```
Student s1 = {1, NULL};
Student s2 = {2, NULL};
Student s3 = {3, NULL};
Student s4 = {4, NULL};

s1.next = &s2;
```



Linked List

```
typedef struct student {
    int id;
    struct student *next;
} Student;
```

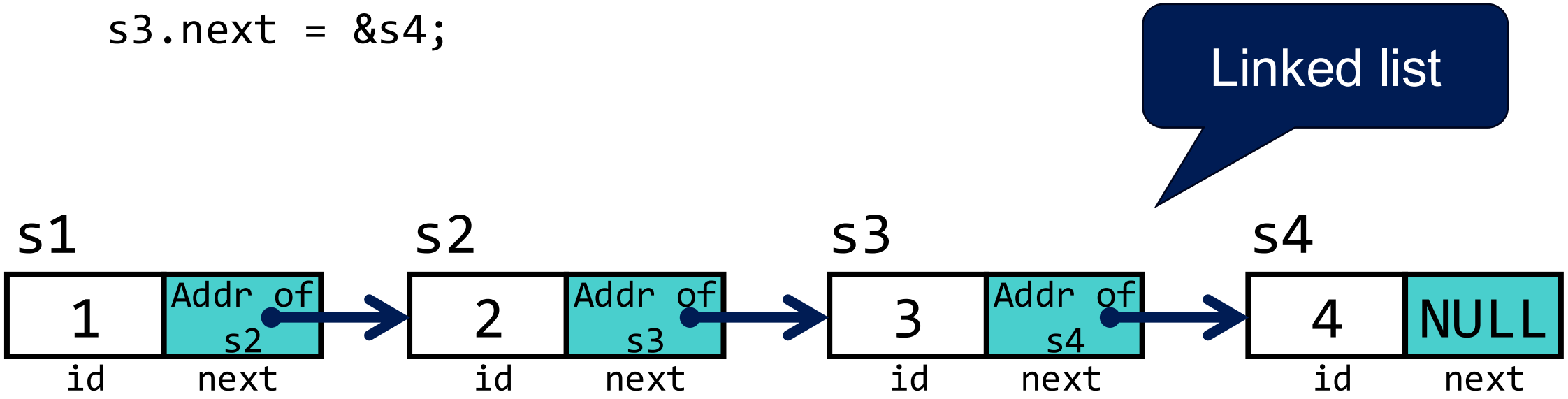
```
Student s1 = {1, NULL};
Student s2 = {2, NULL};
Student s3 = {3, NULL};
Student s4 = {4, NULL};

s1.next = &s2;
s2.next = &s3;
```



Linked List

```
typedef struct student {
    int id;
    struct student *next;
} Student;
```

```
Student s1 = {1, NULL};
Student s2 = {2, NULL};
Student s3 = {3, NULL};
Student s4 = {4, NULL};

s1.next = &s2;
s2.next = &s3;
s3.next = &s4;
```



Linked List

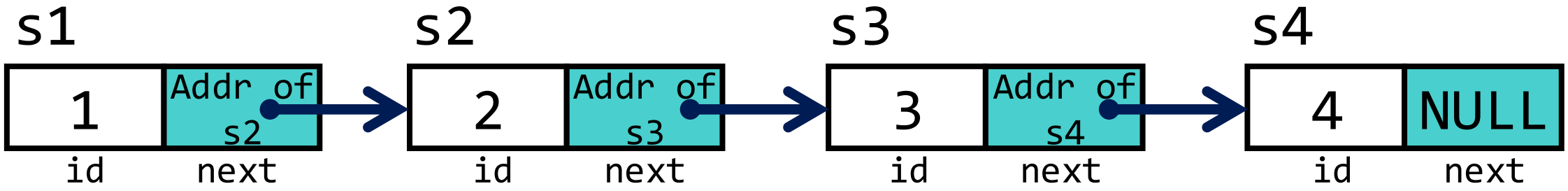
- A linear collection of data elements
  - Similar with array
  - However, it's order is not given by their physical placement in memory
  - Instead, each element points to the next
- It mainly allows *efficient insertion and deletion* operations compared to arrays
- It will be covered in more detail in CSE221 (Data Structures)

Linked List (Traverse)

```c
typedef struct student {
    int id;
    struct student *next;
} Student;
```

```c
Student s1 = {1, NULL};
Student s2 = {2, NULL};
Student s3 = {3, NULL};
Student s4 = {4, NULL};

s1.next = &s2;
s2.next = &s3;
s3.next = &s4;
```

```c
// Print the elements of the list

Student *cur = &s1;

while (cur != NULL) {
    printf("%d ", cur->id);
    cur = cur->next;
}
```

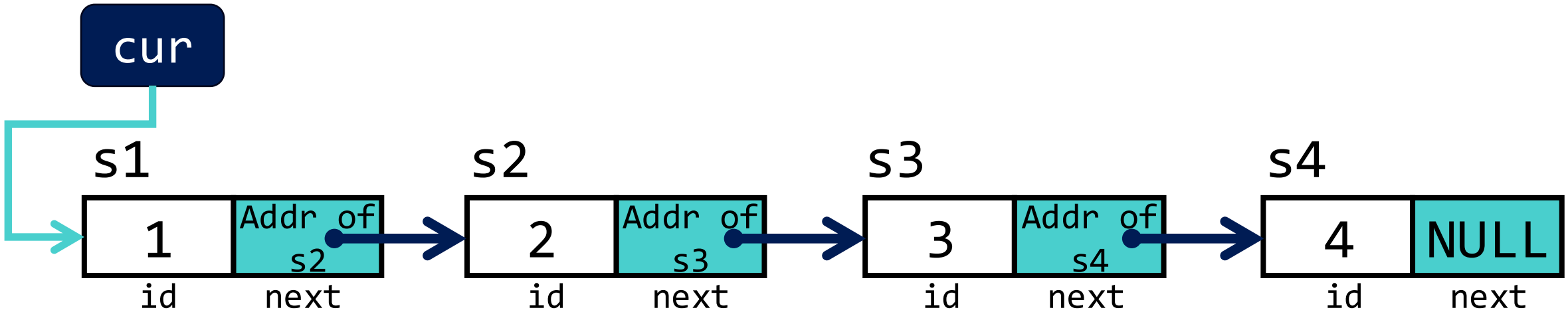


Linked List (Traverse)

```
typedef struct student {
    int id;
    struct student *next;
} Student;
```

```
Student s1 = {1, NULL};
Student s2 = {2, NULL};
Student s3 = {3, NULL};
Student s4 = {4, NULL};

s1.next = &s2;
s2.next = &s3;
s3.next = &s4;
```

```
// Print the elements of the list

Student *cur = &s1;

while (cur != NULL) {
    printf("%d ", cur->id);
    cur = cur->next;
}
```



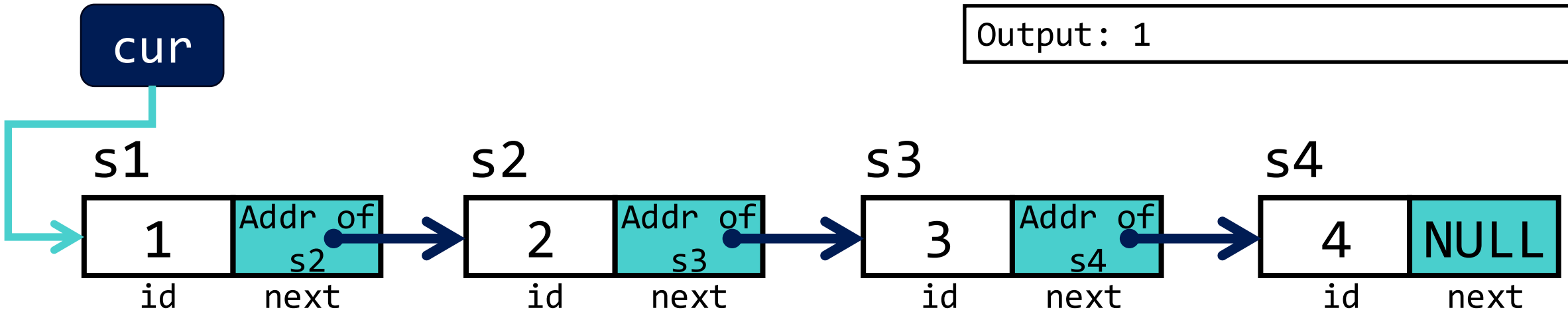
Linked List (Traverse)

```
typedef struct student {
    int id;
    struct student *next;
} Student;
```

```
Student s1 = {1, NULL};
Student s2 = {2, NULL};
Student s3 = {3, NULL};
Student s4 = {4, NULL};

s1.next = &s2;
s2.next = &s3;
s3.next = &s4;
```

```
// Print the elements of the list

Student *cur = &s1;

while (cur != NULL) {
    printf("%d ", cur->id);
    cur = cur->next;
}
```

```
Output: 1
```



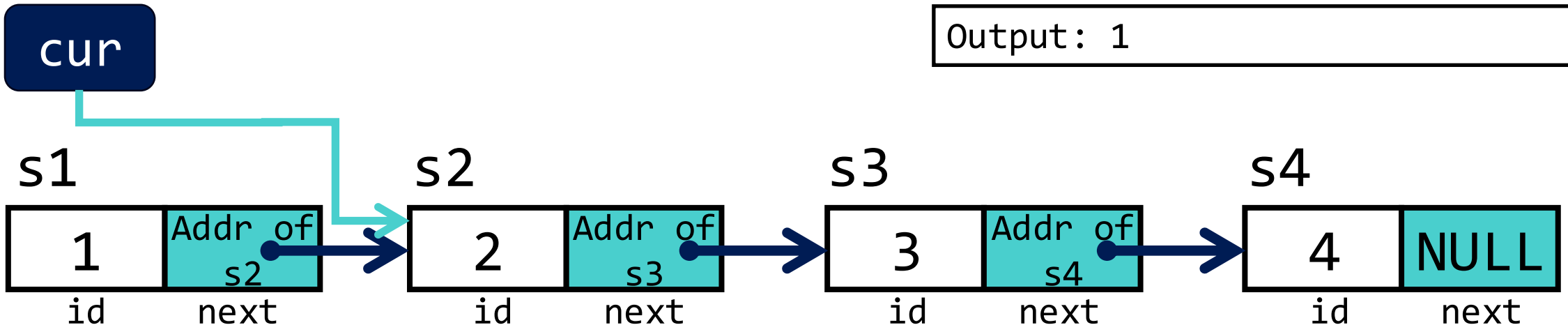
Linked List (Traverse)

```
typedef struct student {
    int id;
    struct student *next;
} Student;
```

```
Student s1 = {1, NULL};
Student s2 = {2, NULL};
Student s3 = {3, NULL};
Student s4 = {4, NULL};

s1.next = &s2;
s2.next = &s3;
s3.next = &s4;
```

```
// Print the elements of the list

Student *cur = &s1;

while (cur != NULL) {
    printf("%d ", cur->id);
    cur = cur->next;
}
```

```
Output: 1
```



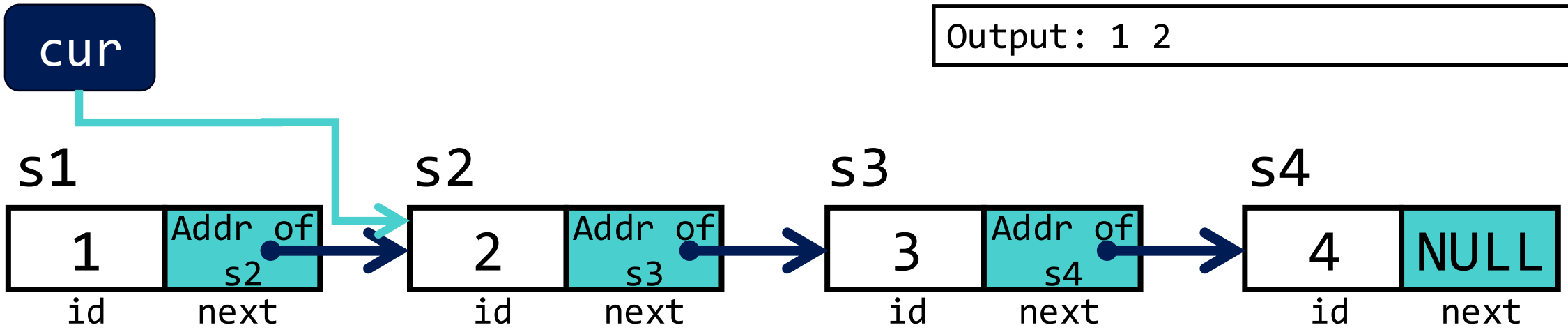
Linked List (Traverse)

```
typedef struct student {
    int id;
    struct student *next;
} Student;
```

```
Student s1 = {1, NULL};
Student s2 = {2, NULL};
Student s3 = {3, NULL};
Student s4 = {4, NULL};

s1.next = &s2;
s2.next = &s3;
s3.next = &s4;
```

```
// Print the elements of the list

Student *cur = &s1;

while (cur != NULL) {
    printf("%d ", cur->id);
    cur = cur->next;
}
```

```
Output: 1 2
```



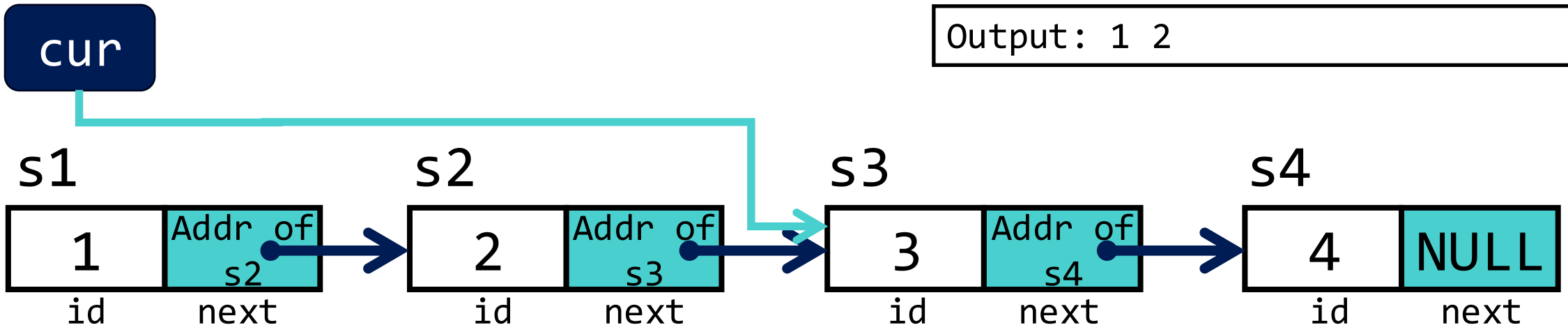
Linked List (Traverse)

```
typedef struct student {
    int id;
    struct student *next;
} Student;
```

```
Student s1 = {1, NULL};
Student s2 = {2, NULL};
Student s3 = {3, NULL};
Student s4 = {4, NULL};

s1.next = &s2;
s2.next = &s3;
s3.next = &s4;
```

```
// Print the elements of the list

Student *cur = &s1;

while (cur != NULL) {
    printf("%d ", cur->id);
    cur = cur->next;
}
```

```
Output: 1 2
```



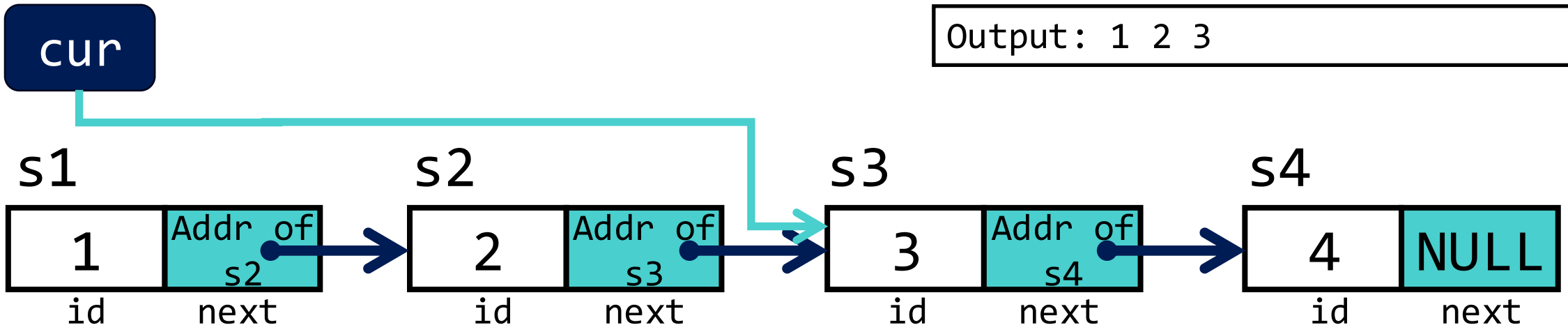
Linked List (Traverse)

```
typedef struct student {
    int id;
    struct student *next;
} Student;
```

```
Student s1 = {1, NULL};
Student s2 = {2, NULL};
Student s3 = {3, NULL};
Student s4 = {4, NULL};

s1.next = &s2;
s2.next = &s3;
s3.next = &s4;
```

```
// Print the elements of the list

Student *cur = &s1;

while (cur != NULL) {
    printf("%d ", cur->id);
    cur = cur->next;
}
```

```
Output: 1 2 3
```



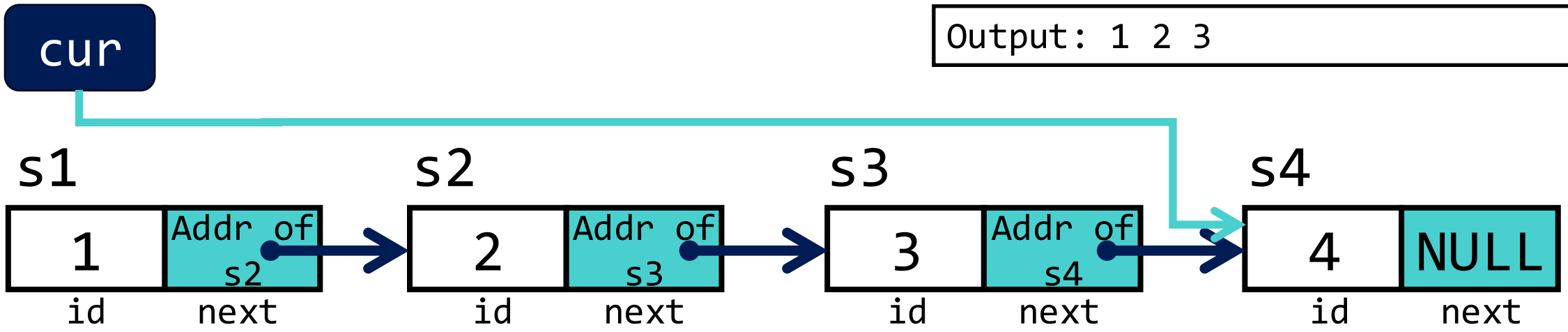
Linked List (Traverse)

```
typedef struct student {
    int id;
    struct student *next;
} Student;
```

```
Student s1 = {1, NULL};
Student s2 = {2, NULL};
Student s3 = {3, NULL};
Student s4 = {4, NULL};

s1.next = &s2;
s2.next = &s3;
s3.next = &s4;
```

```
// Print the elements of the list

Student *cur = &s1;

while (cur != NULL) {
    printf("%d ", cur->id);
    cur = cur->next;
}
```

```
Output: 1 2 3
```



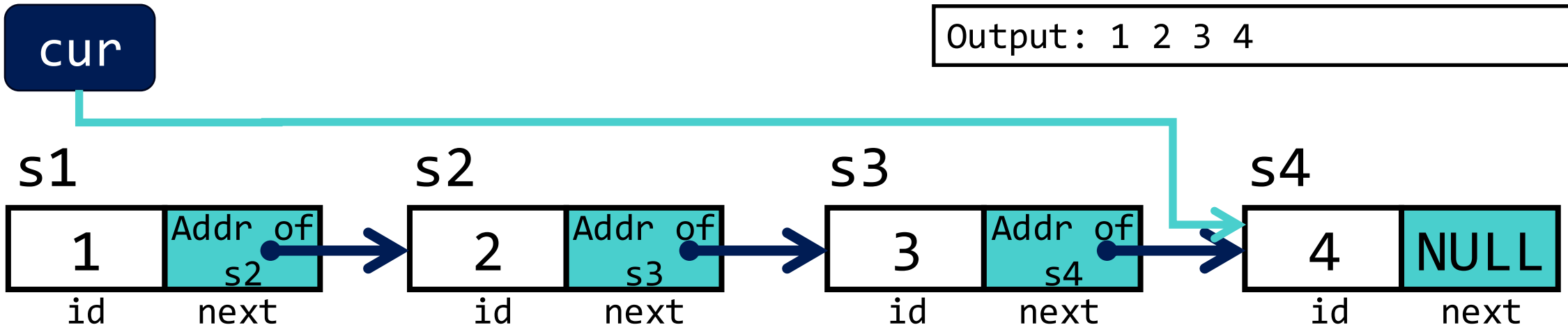
Linked List (Traverse)

```
typedef struct student {
    int id;
    struct student *next;
} Student;
```

```
Student s1 = {1, NULL};
Student s2 = {2, NULL};
Student s3 = {3, NULL};
Student s4 = {4, NULL};

s1.next = &s2;
s2.next = &s3;
s3.next = &s4;
```

```
// Print the elements of the list

Student *cur = &s1;

while (cur != NULL) {
    printf("%d ", cur->id);
    cur = cur->next;
}
```

```
Output: 1 2 3 4
```



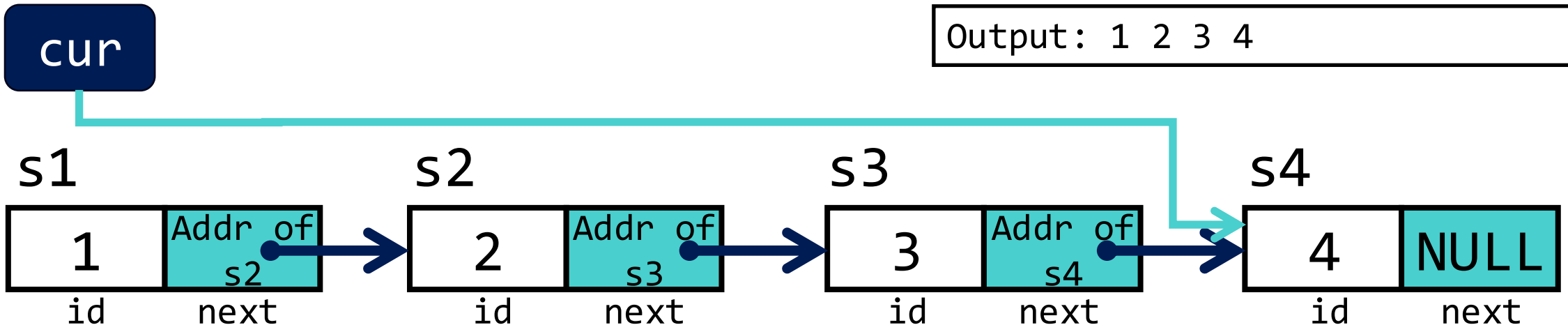
Linked List (Traverse)

```
typedef struct student {
    int id;
    struct student *next;
} Student;
```

```
Student s1 = {1, NULL};
Student s2 = {2, NULL};
Student s3 = {3, NULL};
Student s4 = {4, NULL};

s1.next = &s2;
s2.next = &s3;
s3.next = &s4;
```

```
// Print the elements of the list

Student *cur = &s1;

while (cur != NULL) {
    printf("%d ", cur->id);
    cur = cur->next;
}
```

```
Output: 1 2 3 4
```



Linked List

How is a linked list represented in assembly and memory?

By combining what you have learned so far, you can infer it.
Analyze it yourself through HW2!

Question?