

II - Data

ICS 26Fall

Junqi Xu 2026.9.16

Topics

- Information Structure
- Encodings

Information Structure - Intro

What is a **pointer**?

```
int num = 42;  
int* ptr = &num;
```

Information Structure - Intro

What is a **pointer**?

Address - Address of what? Of where?

Reference - Reference is a kind of pointer.

Alias - Only reference could be called alias.

```
int num = 42;  
int* ptr = &num;
```

Information Structure - Stack & Heap

Stack stores:

Function-owned information
(a.k.a. Stack Frame)

Heap stores:

Program-owned information
(Process-owned, TBP)

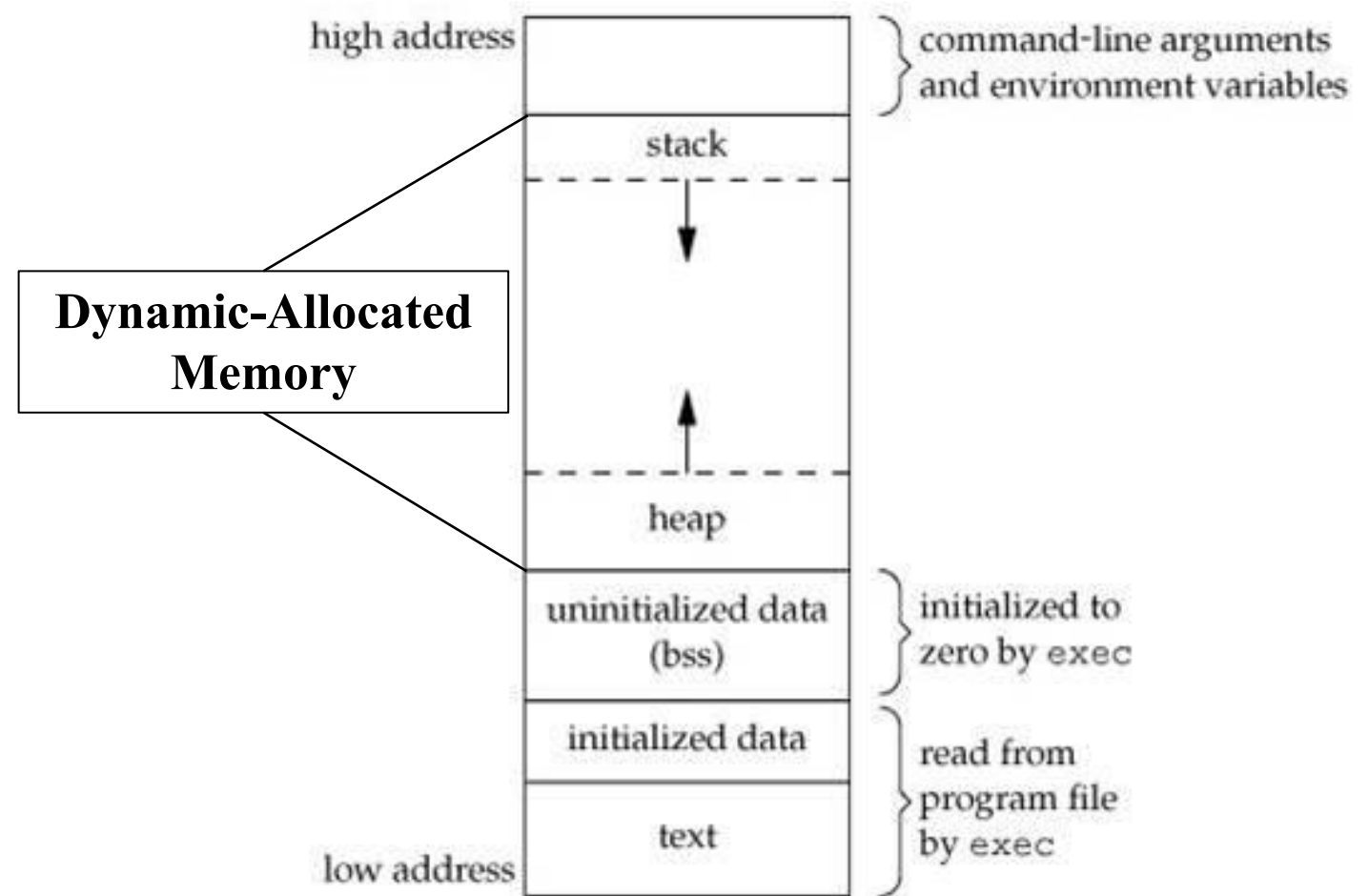


Fig.1^[1]

[1] STEVENS W R, RAGO S A. Advanced programming in the UNIX environment[M]. 3rd ed. Upper Saddle River, NJ: Addison-Wesley, 2013.

Information Structure - Stack & Heap - Lifetime

Lifetimes are a key concept in Rust,
but this concept is not limited to Rust.
It is a **general abstraction** of **program execution**.

Let's begin with a simple C program.

```
int double_(int num) {  
    void* p = malloc(60);  
    return num * 2;  
}
```

```
int main(){  
    void* p = malloc(60);  
    int x = 42;  
    int y = double_(x);  
    free(p);  
    return y;  
}
```

Information Structure - Stack & Heap - Lifetime

→ The program execution begins.

```
int double_(int num) {  
    void* p = malloc(60);  
    return num * 2;  
}
```

```
int main(){  
    void* p = malloc(60);  
    int x = 42;  
    int y = double_(x);  
    free(p);  
    return y;  
}
```

Information Structure - Stack & Heap - Lifetime

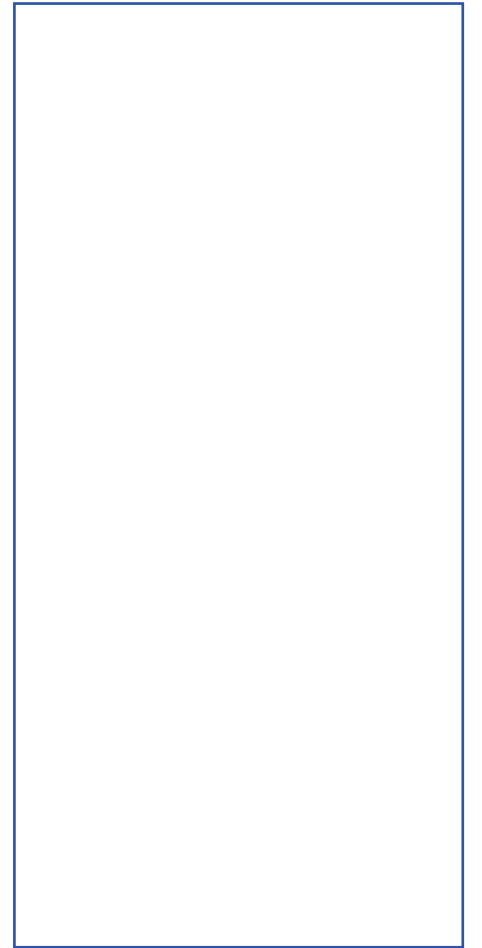
→ The process is created.

```
int double_(int num) {  
    void* p = malloc(60);  
    return num * 2;  
}
```

```
int main(){  
    void* p = malloc(60);  
    int x = 42;  
    int y = double_(x);  
    free(p);  
    return y;  
}
```



Stack



Heap

Information Structure - Stack & Heap - Lifetime

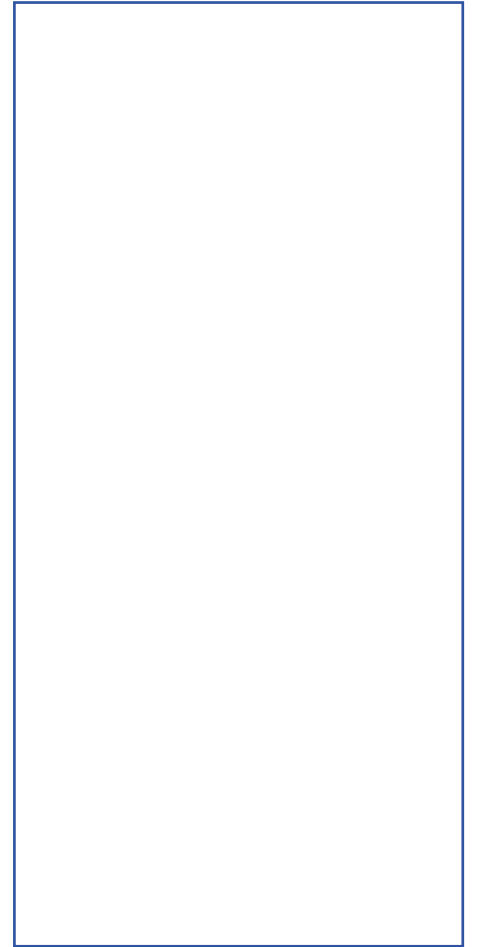
➡ Preparing to call main().

```
int double_(int num) {  
    void* p = malloc(60);  
    return num * 2;  
}
```

```
int main(){  
    void* p = malloc(60);  
    int x = 42;  
    int y = double_(x);  
    free(p);  
    return y;  
}
```



Stack

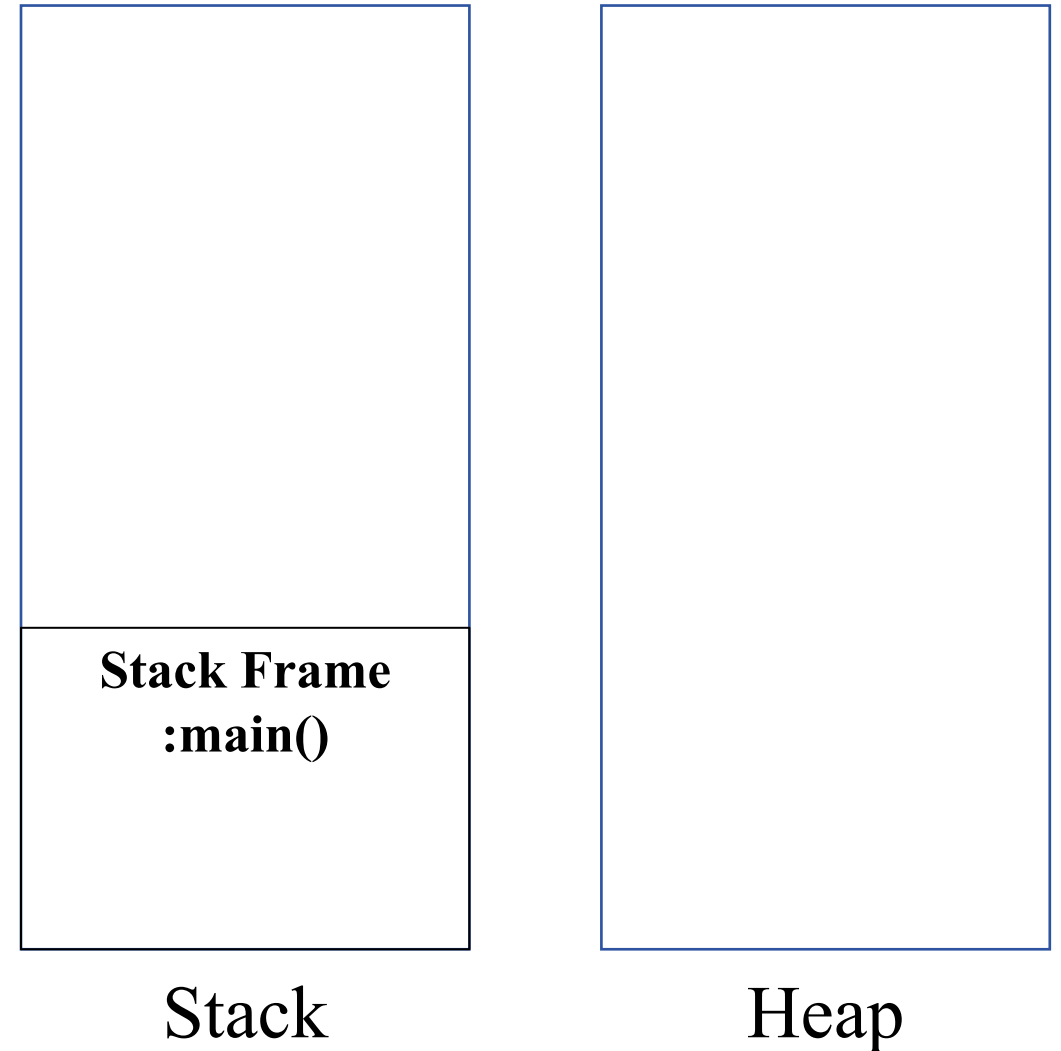


Heap

Information Structure - Stack & Heap - Lifetime

```
int double_(int num) {  
    void* p = malloc(60);  
    return num * 2;  
}
```

```
→ int main(){  
    void* p = malloc(60);  
    int x = 42;  
    int y = double_(x);  
    free(p);  
    return y;  
}
```

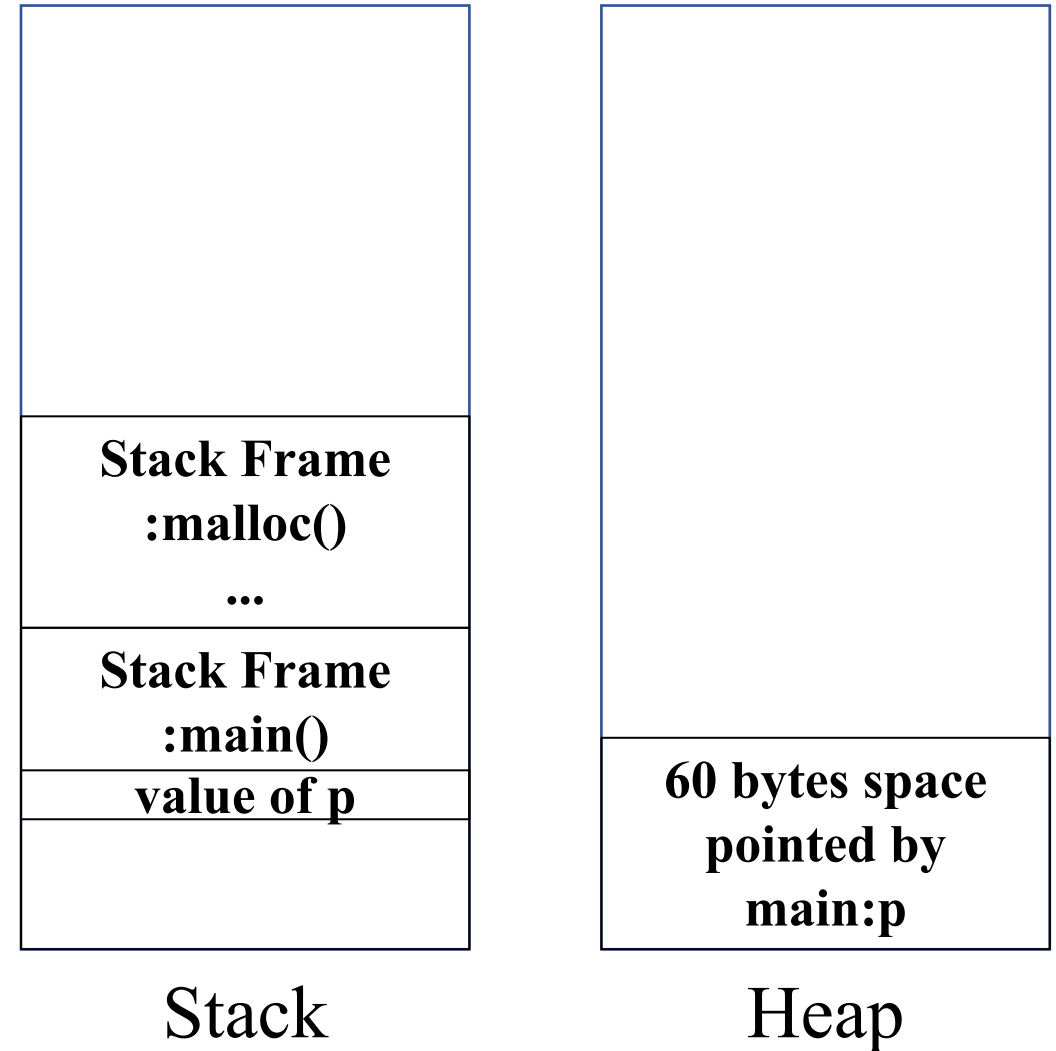


Information Structure - Stack & Heap - Lifetime

```
int double_(int num) {  
    void* p = malloc(60);  
    return num * 2;  
}
```

→

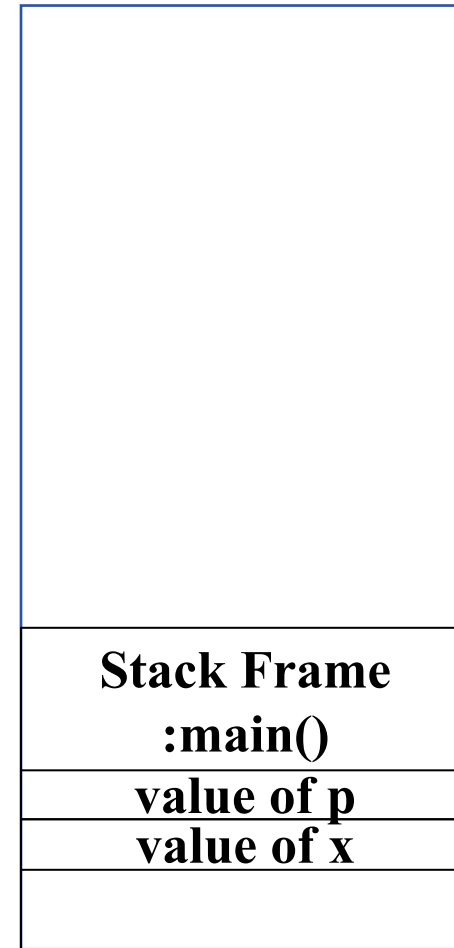
```
int main(){  
    void* p = malloc(60);  
    int x = 42;  
    int y = double_(x);  
    free(p);  
    return y;  
}
```



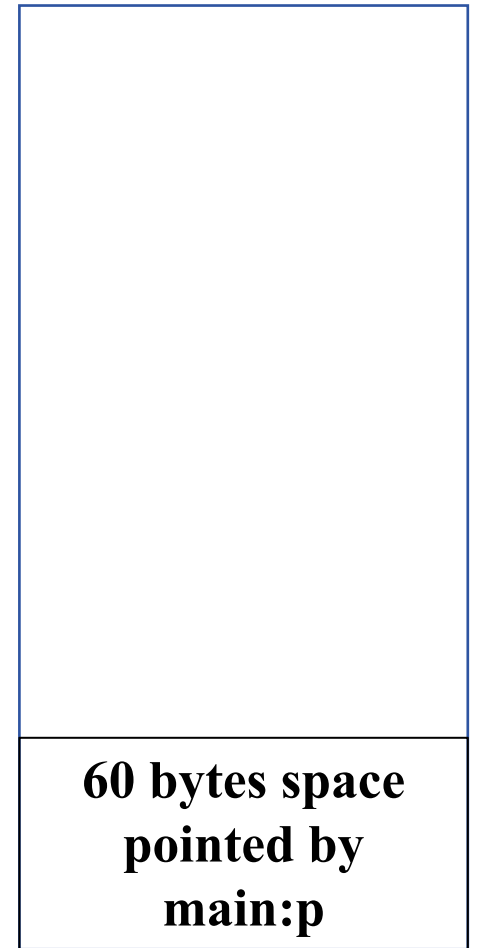
Information Structure - Stack & Heap - Lifetime

```
int double_(int num) {  
    void* p = malloc(60);  
    return num * 2;  
}
```

```
int main(){  
    void* p = malloc(60);  
    int x = 42;  
    int y = double_(x);  
    free(p);  
    return y;  
}
```



Stack

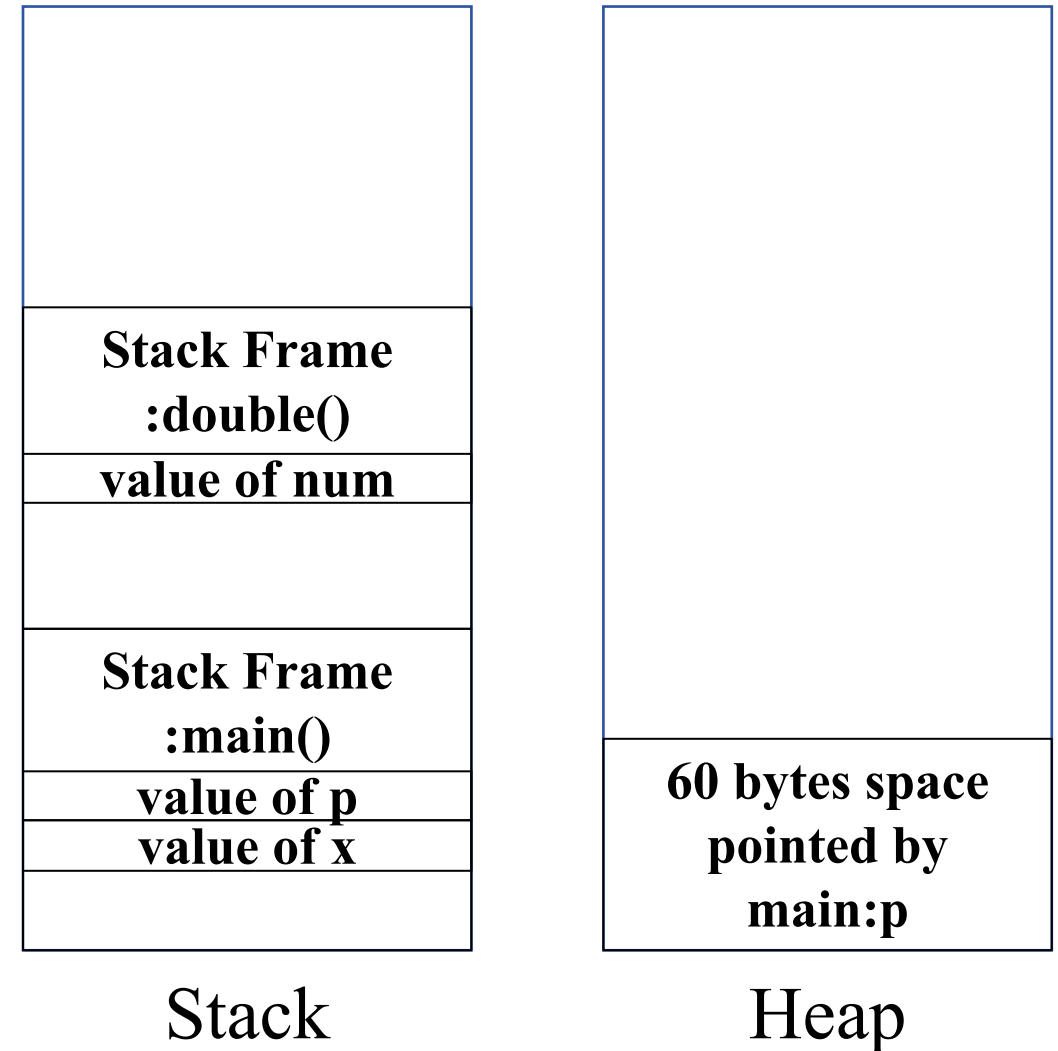


Heap

Information Structure - Stack & Heap - Lifetime

```
int double_(int num) {  
    void* p = malloc(60);  
    return num * 2;  
}
```

```
int main(){
    void* p = malloc(60);
    int x = 42;
    int y = double_(x);
    free(p);
    return y;
}
```

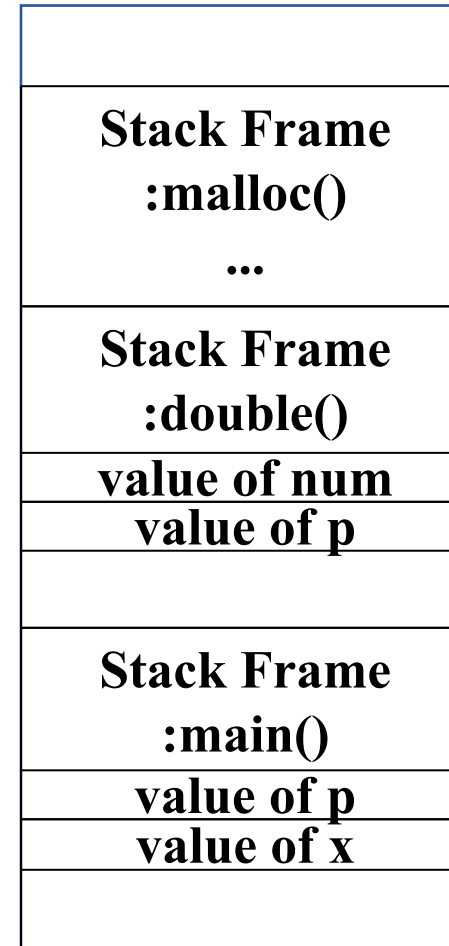


Information Structure - Stack & Heap - Lifetime

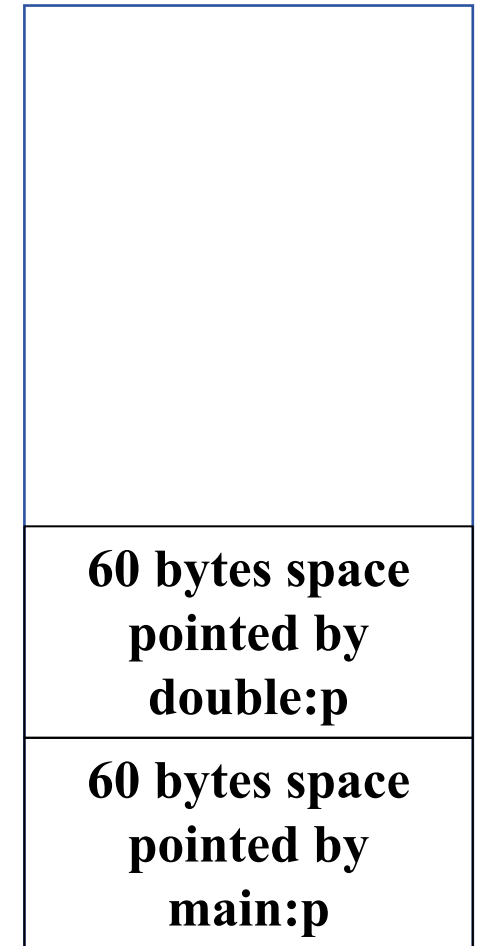
→

```
int double_(int num) {  
    void* p = malloc(60);  
    return num * 2;  
}
```

```
int main(){  
    void* p = malloc(60);  
    int x = 42;  
    int y = double_(x);  
    free(p);  
    return y;  
}
```



Stack

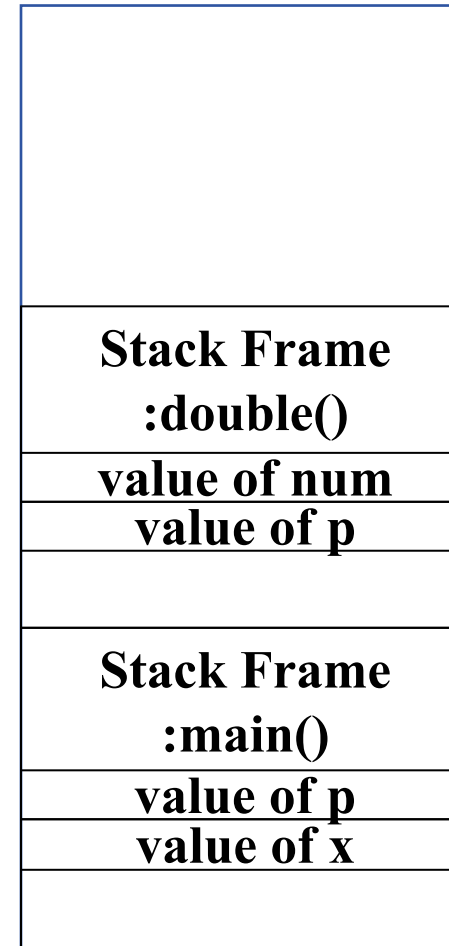


Heap

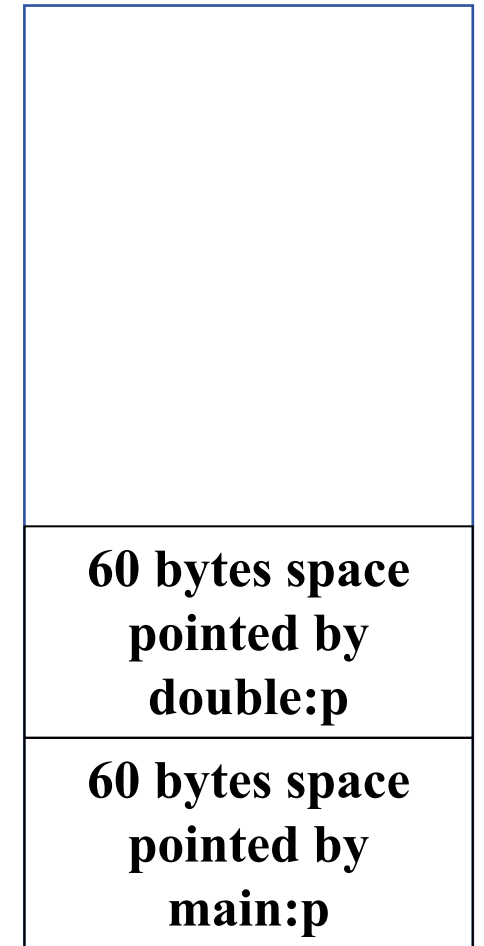
Information Structure - Stack & Heap - Lifetime

```
int double_(int num) {  
    void* p = malloc(60);  
    return num * 2;  
}
```

```
int main(){  
    void* p = malloc(60);  
    int x = 42;  
    int y = double_(x);  
    free(p);  
    return y;  
}
```



Stack

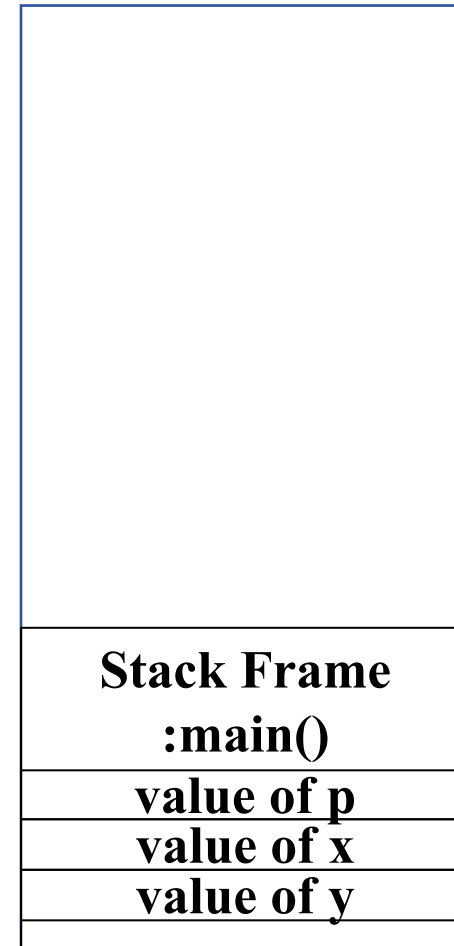


Heap

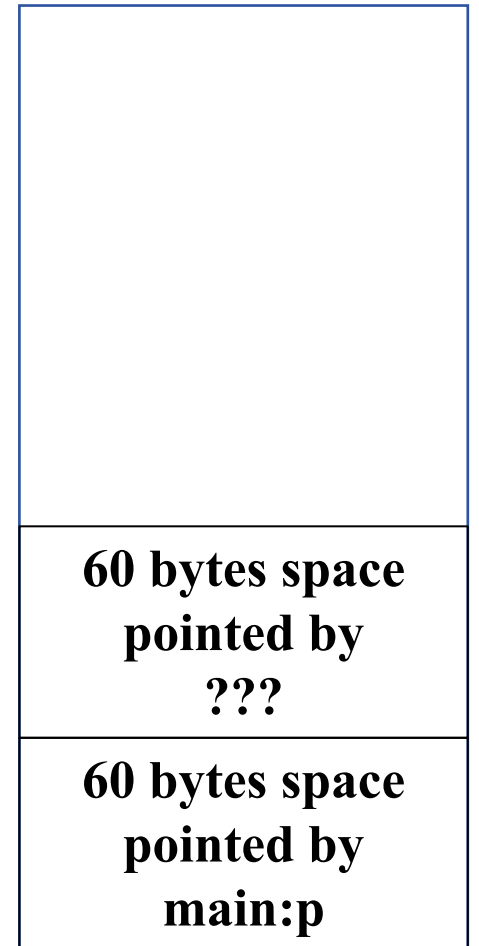
Information Structure - Stack & Heap - Lifetime

```
int double_(int num) {  
    void* p = malloc(60);  
    return num * 2;  
}
```

```
int main(){  
    void* p = malloc(60);  
    int x = 42;  
    int y = double_(x);  
    free(p);  
    return y;  
}
```



Stack

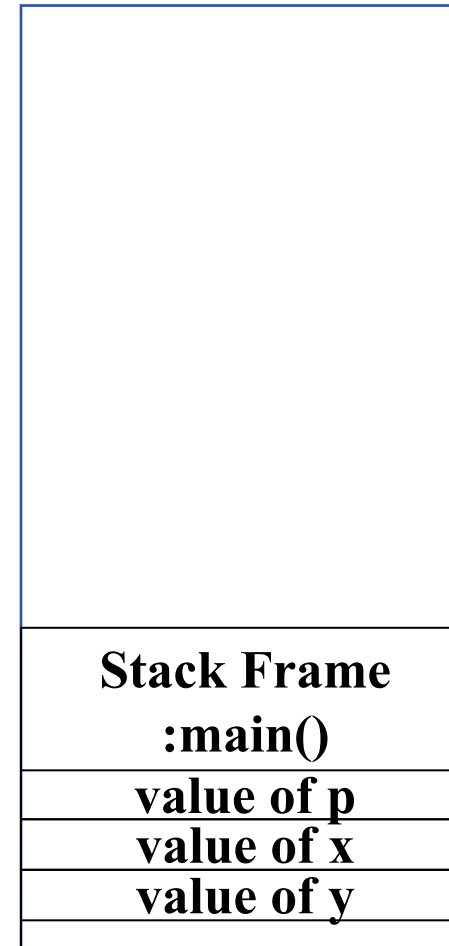


Heap

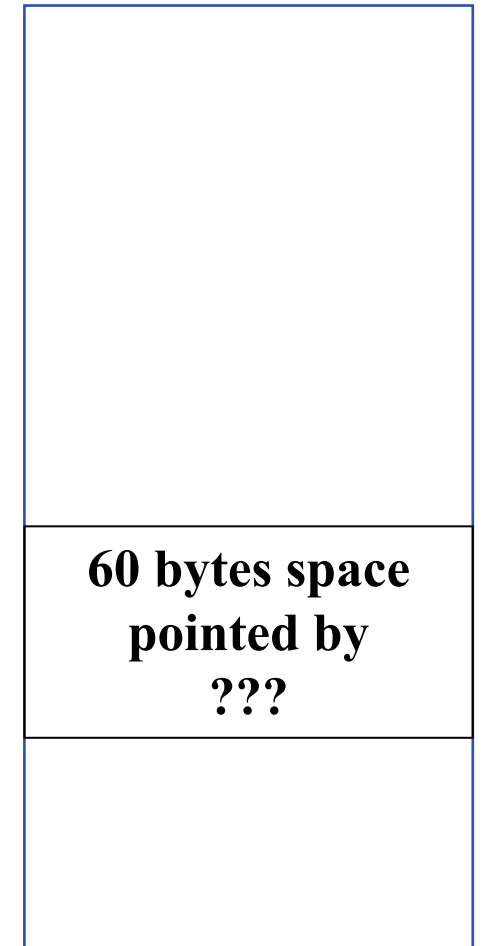
Information Structure - Stack & Heap - Lifetime

```
int double_(int num) {  
    void* p = malloc(60);  
    return num * 2;  
}
```

```
int main(){  
    void* p = malloc(60);  
    int x = 42;  
    int y = double_(x);  
    free(p);  
    return y;  
}
```



Stack

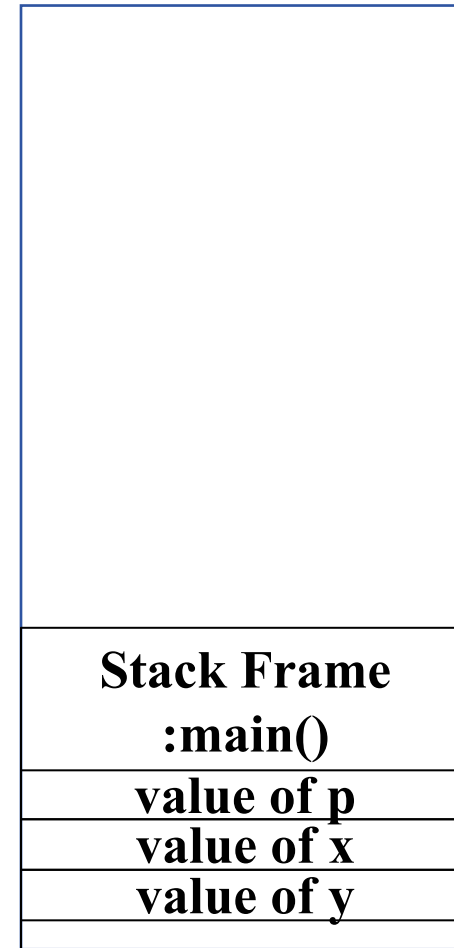


Heap

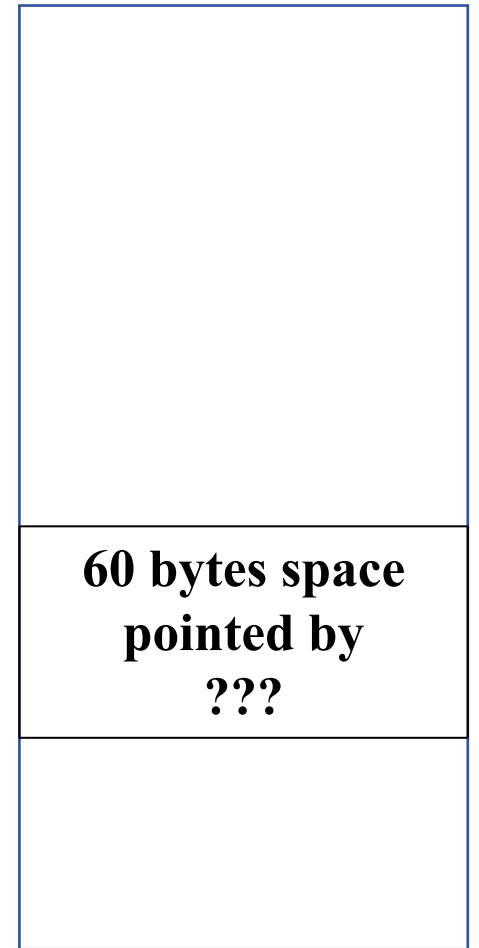
Information Structure - Stack & Heap - Lifetime

```
int double_(int num) {  
    void* p = malloc(60);  
    return num * 2;  
}
```

```
int main(){  
    void* p = malloc(60);  
    int x = 42;  
    int y = double_(x);  
    free(p);  
    return y;  
}
```



Stack



Heap

Information Structure - Stack & Heap - Lifetime

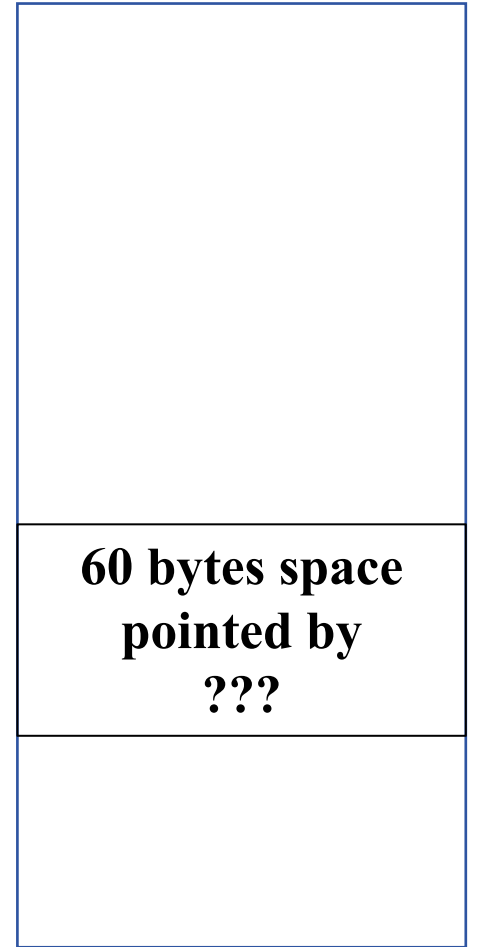
→ Doing cleanup.

```
int double_(int num) {  
    void* p = malloc(60);  
    return num * 2;  
}
```

```
int main(){  
    void* p = malloc(60);  
    int x = 42;  
    int y = double_(x);  
    free(p);  
    return y;  
}
```



Stack



Heap

Information Structure - Stack & Heap - Lifetime

→ The process is terminated.

```
int double_(int num) {  
    void* p = malloc(60);  
    return num * 2;  
}
```

```
int main(){  
    void* p = malloc(60);  
    int x = 42;  
    int y = double_(x);  
    free(p);  
    return y;  
}
```

Information Structure - Stack & Heap

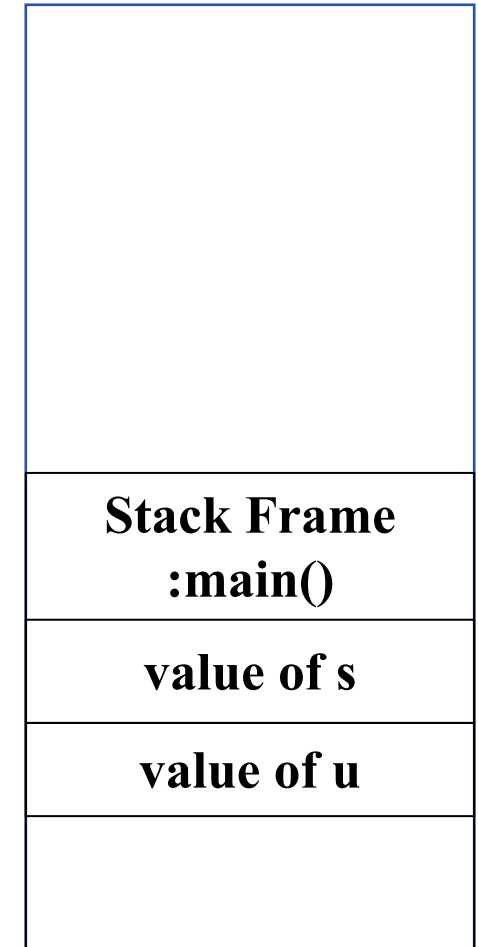
Let's practice with a more complicated C program.

```
typedef struct Struct {  
    /* Some fields */  
} Struct;  
  
typedef union Union {  
    /* Some fields */  
} Union;  
  
int main() {  
    Struct s = { /* ... */ };  
    Union u = { /* ... */ };  
}
```

Information Structure - Stack & Heap

Let's practice with a more complicated C program.

```
typedef struct Struct {  
    /* Some fields */  
} Struct;  
  
typedef union Union {  
    /* Some fields */  
} Union;  
  
int main() {  
    Struct s = { /* ... */ };  
    Union u = { /* ... */ };  
}
```



Stack

Information Structure - Stack & Heap

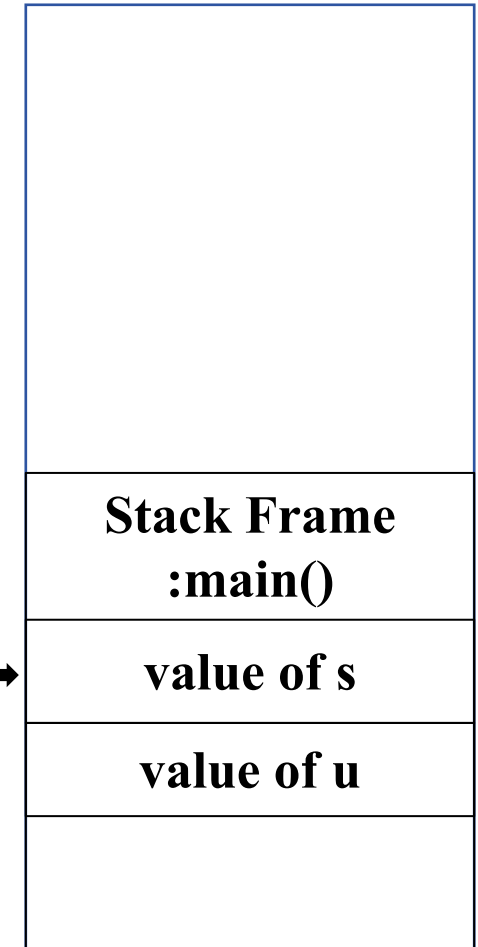
Let's practice with a more complicated C program.

```
typedef struct Struct {  
    /* Some fields */  
} Struct;
```

```
typedef union Union {  
    /* Some fields */  
} Union;
```

```
int main() {  
    Struct s = { /* ... */ };  
    Union u = { /* ... */ };  
}
```

What is exactly the value of s? ➡



Stack

Information Structure - Type Layout

Type Layout: How an instance of a type is laid out in the memory.

Three components:

1. Size
2. Alignment
3. Fields Order

Exercise:

Size and Alignment of these types in C?

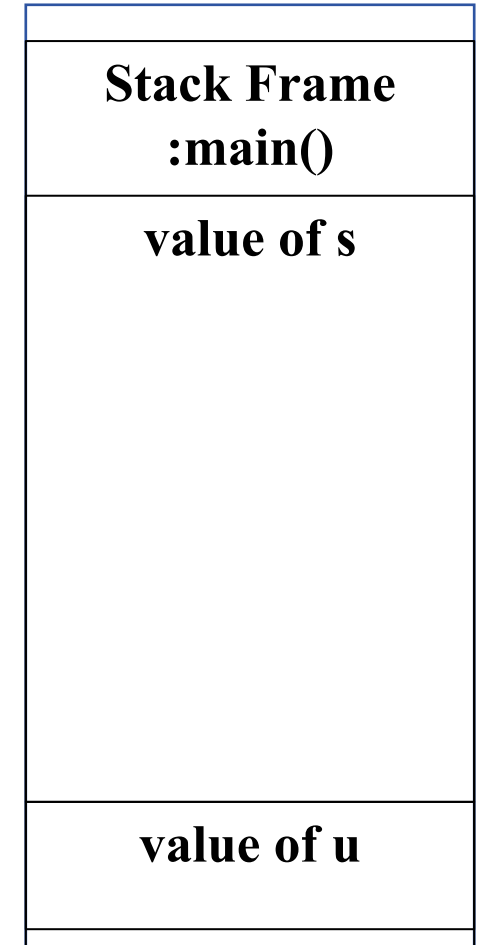
`char short int long double float[5] char[19] int*`

Information Structure - Type Layout - Structure

```
typedef struct Struct {  
    char c;  
    int i;  
    short s;  
    double d;  
    char bytes[3];  
    char tail;  
} Struct;
```

Size: ?

Alignment: ?



Stack

Information Structure - Type Layout - Structure

```
typedef struct Struct {  
    char c;  
    int i;  
    short s;  
    double d;  
    char bytes[3];  
    char tail;  
} Struct;
```

Size: 32 Bytes

Alignment: 8 Bytes

Stack Frame :main()
value of s
1 Bytes char
3 Bytes Padding
4 Bytes int
2 Bytes short
6 Bytes Padding
8 Bytes double
3 Bytes char[3]
1 Bytes char
4 Bytes Padding
value of u

Stack

Information Structure - Type Layout - Structure

```
typedef struct Struct {  
    char c;  
    int i;  
    short s;  
    double d;  
    char bytes[3];  
    char tail;  
} Struct;
```

Size: 32 Bytes

Alignment: 8 Bytes

Exercise: Reorder Struct to achieve minimum size.

Stack Frame :main()
value of s
1 Bytes char
3 Bytes Padding
4 Bytes int
2 Bytes short
6 Bytes Padding
8 Bytes double
3 Bytes char[3]
1 Bytes char
4 Bytes Padding
value of u

Stack

Information Structure - Type Layout - Union

```
typedef union Union {  
    char c;  
    int i;  
    double d;  
    char bytes[11];  
} Union;
```

Size: ?

Alignment: ?

Stack Frame :main()
value of s
1 Bytes char
3 Bytes Padding
4 Bytes int
2 Bytes short
6 Bytes Padding
8 Bytes double
3 Bytes char[3]
1 Bytes char
4 Bytes Padding
value of u

Stack

Information Structure - Type Layout - Union

```
typedef union Union {  
    char c;  
    int i;  
    double d;  
    char bytes[11];  
} Union;
```

Size: 16

Alignment: 8

Stack Frame :main()
value of s
1 Bytes char
3 Bytes Padding
4 Bytes int
2 Bytes short
6 Bytes Padding
8 Bytes double
3 Bytes char[3]
1 Bytes char
4 Bytes Padding
value of u
16 Bytes

Stack

Information Structure - Bytes

```
int i = 0x12345678;
```

4 bytes:

0x12 0x34 0x56 0x78

Order of bytes? →

Stack Frame :main()
value of s
1 Bytes char
3 Bytes Padding
4 Bytes int
2 Bytes short
6 Bytes Padding
8 Bytes double
3 Bytes char[3]
1 Bytes char
4 Bytes Padding
value of u
16 Bytes

Stack

Information Structure - Bytes

```
int i = 0x12345678;
```

4 bytes:

0x12 0x34 0x56 0x78

2 byte orders:

Big Endianness: 0x12 0x34 0x56 0x78

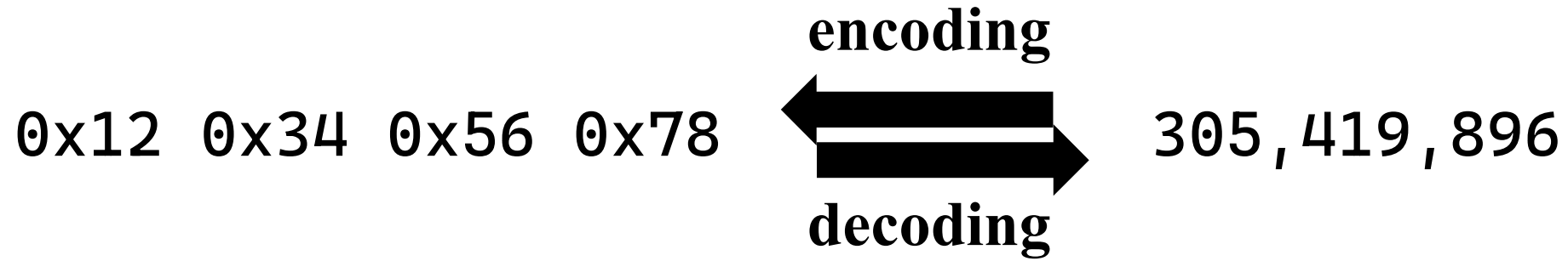
Small Endianness: 0x78 0x56 0x34 0x12

Why small endianness? Just to stand out?

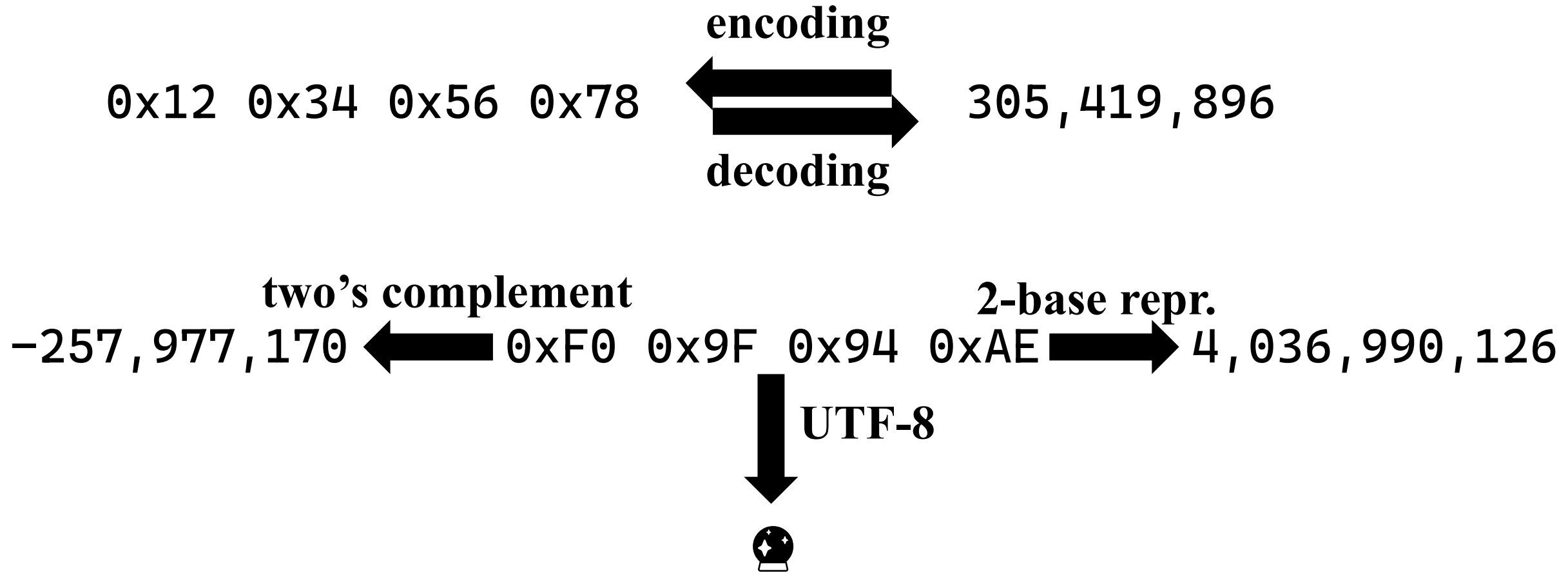
Stack Frame :main()
value of s
1 Bytes char
3 Bytes Padding
4 Bytes int
2 Bytes short
6 Bytes Padding
8 Bytes double
3 Bytes char[3]
1 Bytes char
4 Bytes Padding
value of u
16 Bytes

Stack

Information Structure - Encodings



Information Structure - Encodings



Information Structure - Encodings - Integer

Unsigned: **2-Base Representation**

Signed: **Two's Complement**

Relatives of Two's Complement:

Sign-Magnitude

One's Complement

Information Structure - Encodings - Real Number

IEEE754 Floating-Point Encoding

Sign	Exponent	Significand
------	----------	-------------

$$\text{val} = (-1)^{\text{Sign}} * \text{Significand} * 2^{\text{Exponent} - \text{Bias}}$$

Exercise:

C types: `float`, `double`

FP concepts: `Inf`, `NaN`, `Subnormal`

Information Structure - Encodings - Character

ASCII: 128 character encodings. (1 byte)

Unicode: More than 155, 000 character encodings, even growing. (3 bytes)

UTF-8: All Unicode character encodings. (variable-width, 1-4 bytes)

UTF-8 Prefixes: 0, 10, 110, 1110, 11110

Takeaway

Concepts:

Stack, Heap, Stack Frame, Type Layout, Alignment, Bytes Ordering.

Encodings, Decodings, 2's Complement.

IEEE754 Floating-Point, Bias, Inf, NaN, Subnormal.

ASCII, Unicode, UTF-8.