Pointers

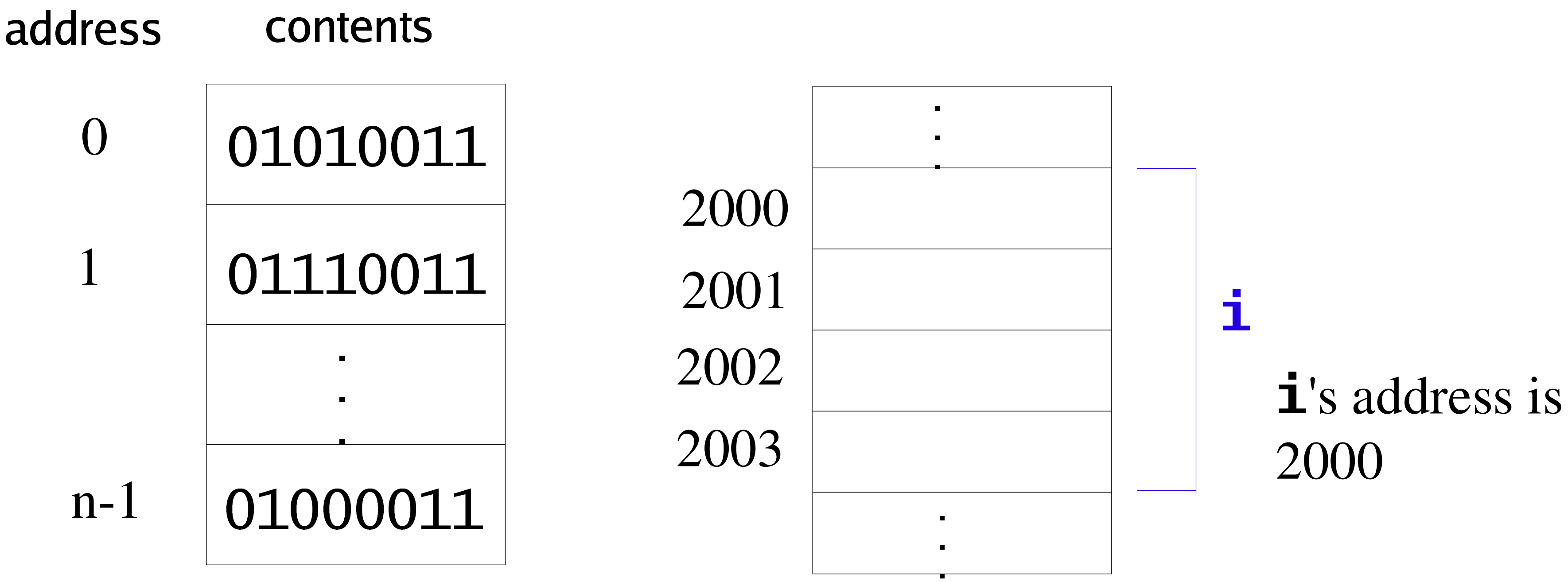
CS2023 Winter 2004

Outcomes: Introduction to Pointers

- “C for Java Programmers”, Chapter 8, sections 8.1-8.8
- Other textbooks on C on reserve
- After the conclusion of this section you should be able to
  - Describe the two separate areas of memory: heap and stack
  - Declare and initialize pointers of the appropriate type
  - Dereference and copy pointers
  - Describe generic pointers and the NULL macro (we'll use these soon)

Addresses

- Each variable occupies one or more bytes of memory
- Address of first byte is address of variable



address	contents
0	01010011
1	01110011
	⋮
n-1	01000011

i's address is 2000

Memory Management

- Compilers manage memory in three main parts:

  1) Global and static variables

```
#include <stdio.h>
int f(int j);
int k = 0;
int main ()
{
   int i;
   ...
   i = f(k);
   ...
}
int f(int j){
   static int counter;
   int m;
...}
```

Memory Management

- Compilers manage memory in three main parts:
  1) Global and static variables: *never cease to exist!*
    - Addresses compiled into code
    - Allocated at compile time
    - limited to fixed-size objects

Memory Management

- Compilers manage memory in three main parts:

  2) Automatic variables

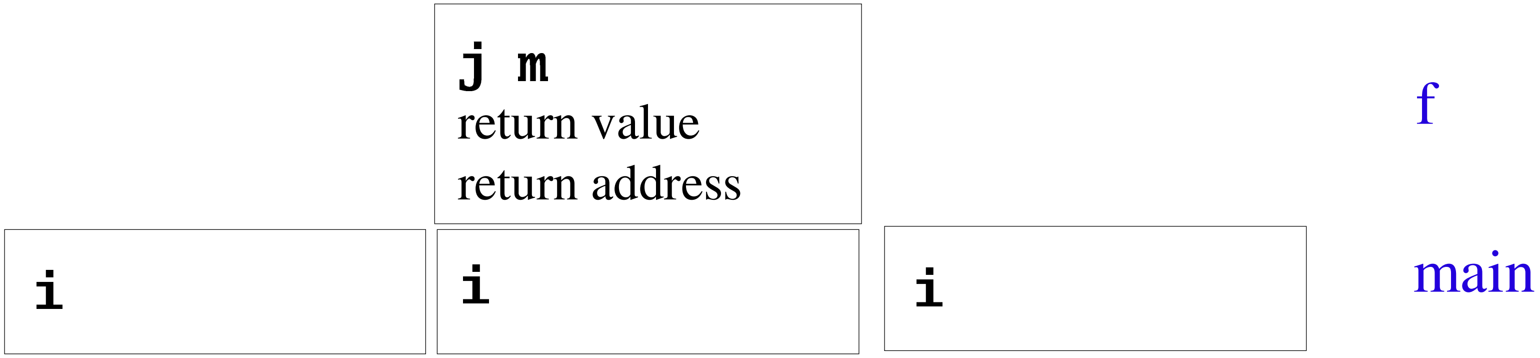
```
#include <stdio.h>
int f(int j);
int k = 0;
int main ()
{
   int i;
   ...
   i = f(k);
   ...
}
int f(int j){
   static int counter;
   int m;
...}
```

Memory Management

- Compilers manage memory in three main parts:

  2) Automatic variables: *also called stack variables*

    - Size known at compile time
    - Stored on a stack (run-time stack)



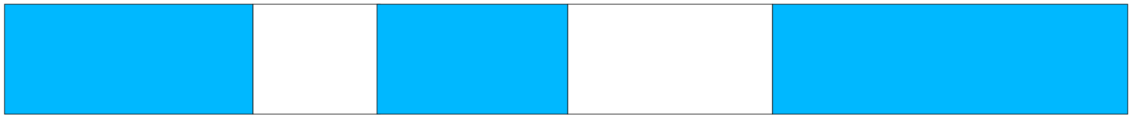
main calls f

Memory Management

- Compilers manage memory in three main parts:
  2) Automatic variables: *also called stack variables*
    - Function that terminates always the last called
    - Never a hole in the stack!

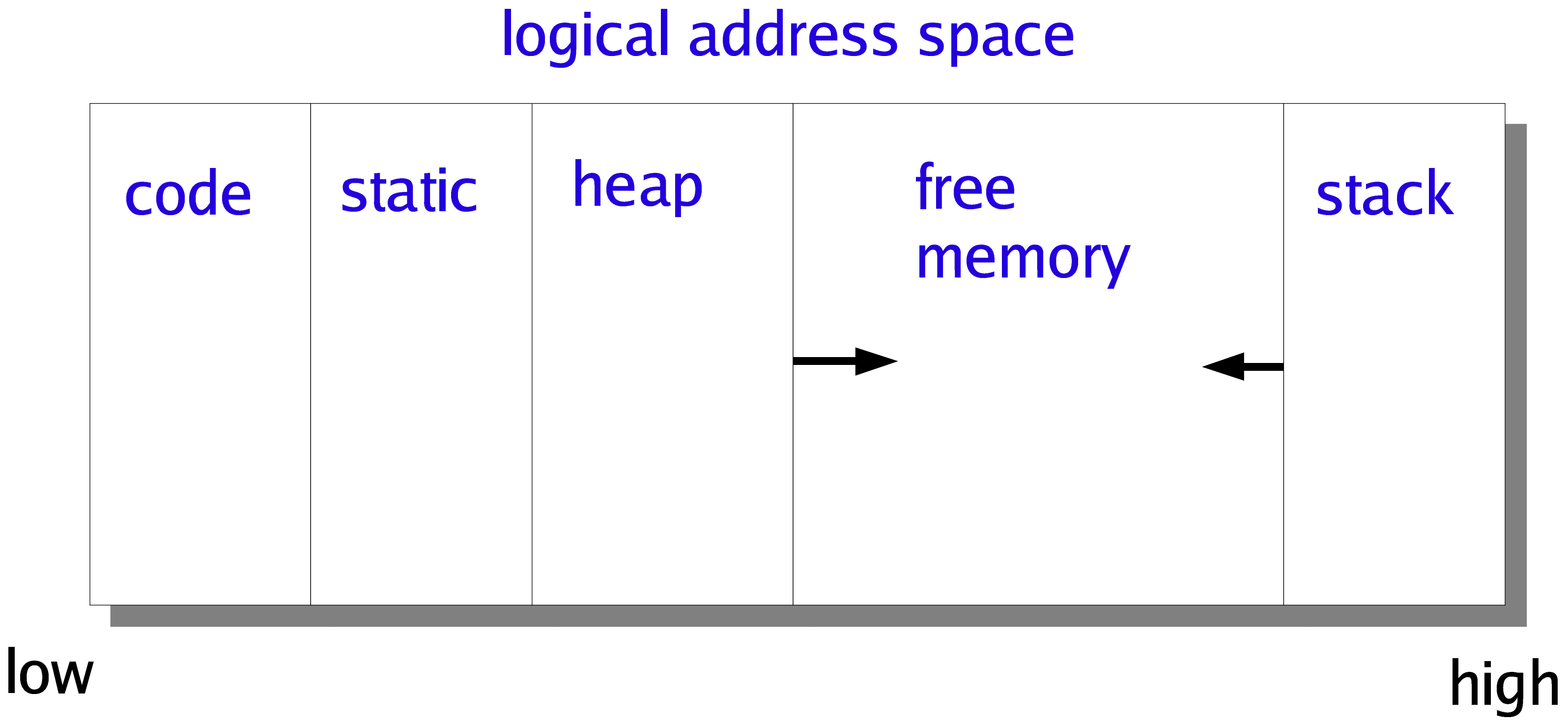
Memory Management

- Compilers manage memory in three main parts:
  3) Dynamically allocated variables
    - Memory allocated and destroyed at run time, under control of programmer!
    - No guarantee that first variable to be destroyed is last created
    - This area of memory can have holes and is called the **heap**



Memory Management

- Usually heap and stack begin at opposite ends of the program's memory, and grow towards each other



Memory Management

Stack-based memory: *implicitly* managed by function calls and function returns.

Advantage: you do not have to be concerned with deallocation.

Disadvantage: may lead to programming errors, e.g.

dangling reference problem

a variable referencing a memory block
whose lifetime has expired

Memory Management

Heap-based memory: *explicitly* managed by the programmer.

May result in heap fragmentation

C programmers are *responsible* for memory management

Improper memory management may lead to **memory leakage**

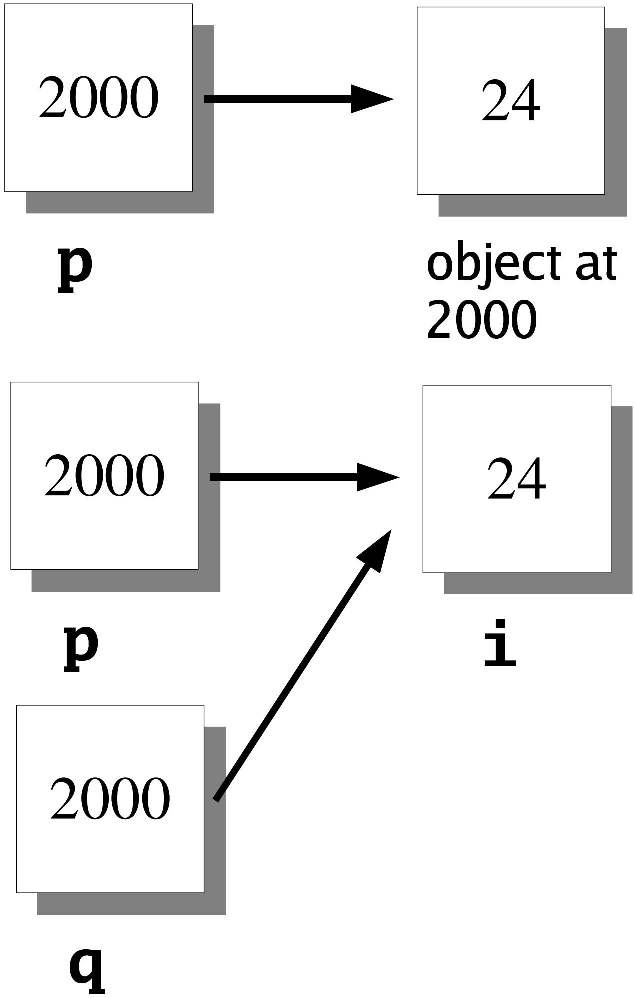
Memory is a resource, and has to be managed like any other resource (e.g. file handles for open files)

Pointers

- A **pointer** is a variable whose value is a memory address representing the location of the chunk of memory on either the run-time stack or on the heap.
- Pointers have names, values and types.

- Value of p versus value pointed to by p

Pointers



object at memory location 2000

Declaring Pointers

For any C data type T, you can define a variable of type "pointer to T":

`int *p;`	pointer to **`int`**, or **`int`** pointer
`char *q;`	pointer to **`char`**, or **`char`** pointer
`double **w;`	pointer to pointer to **`double`**

Here:

`p` may point to a block of **`sizeof(int)`** bytes

`q` may point to a block of **`sizeof(char)`** bytes

`w` may point to a block of **`sizeof(double*)`** bytes

Declaring Pointers

- The placement of the whitespace around the asterisk in a pointer declaration is a convention

```
int* p;
int * p;
int *p;
```

We will use the third convention

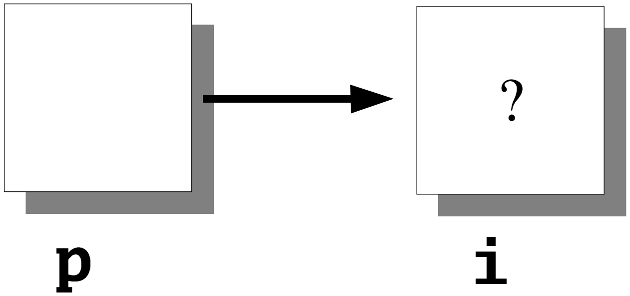
Address Operator

- Declaring a pointer variable sets aside space for a pointer but doesn't make it point to an object

```
int *p;
```

- Need to initialize **p**

```
int i, *p;
p = &i;
```



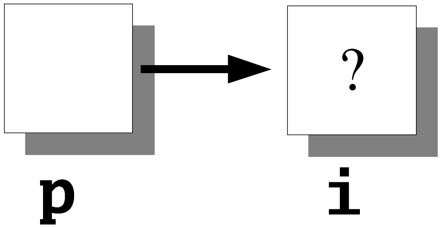
`i` is an `int` variable

`&i` is like an `int` pointer, pointing to the variable `i`

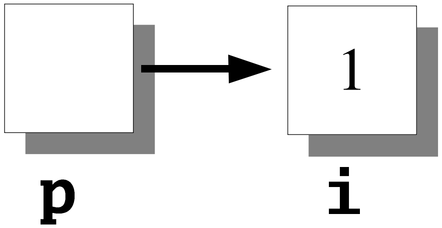
(but you must not assign to it)

Dereferencing: Indirection Operator

```
int i, *p;
p = &i;
```



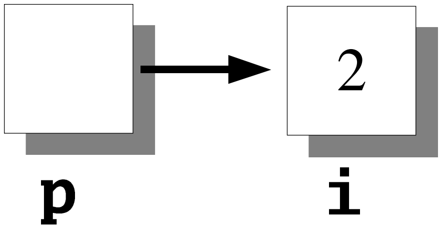
```
i = 1;
```



```
printf("%d\n", i);  /* prints 1/
printf("%d\n", *p); /* prints 1/
```

Dereferencing: Indirection Operator

```
*p = 2;
```



```
printf("%d\n", i);  /* prints 2/
printf("%d\n", *p); /* prints 2/
```

Can change variable `i` without actually using `i`

- use this to implement function call by reference

Dereferencing: Indirection Operator

- Never dereference an unitialized pointer!

```
int *p;
printf("%d", p); /* prints garbage */
```

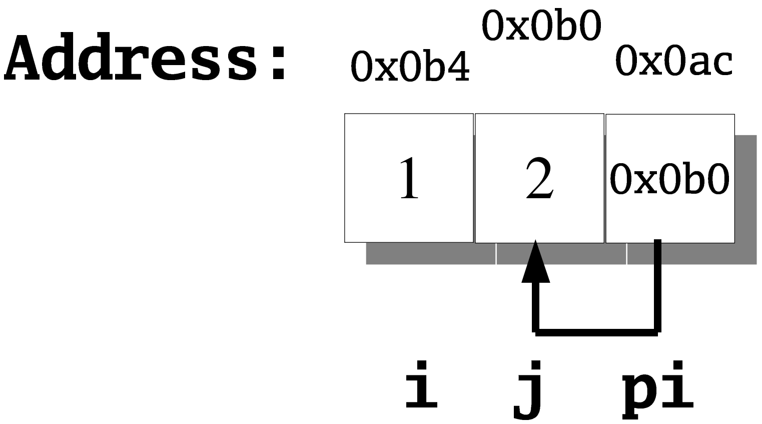
- Assigning a value to **`*p`** much worse!

```
int *p;
*p = 1;
```

  - Where does **`p`** point to? It might point to memory belonging to the program, causing it to behave erratically, or to memory belonging to another process, causing a segmentation fault.

Adresses and Values of variables

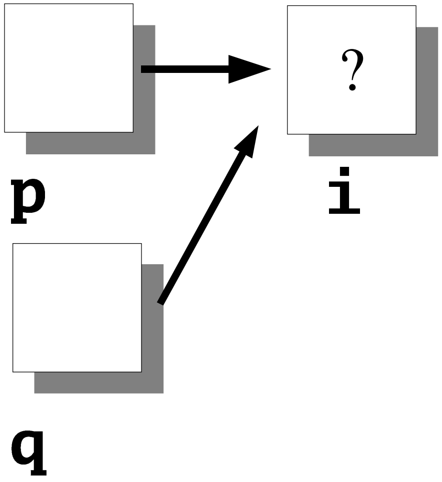
```
int i = 1, j = 2, *pi = &j;
printf("&i=%p, &j=%p, &pi=%p\n", &i, &j, &pi);
printf("pi=%p, *pi=%d, i=%d, j=%d\n", pi, *pi, i, j);
return 0;
```



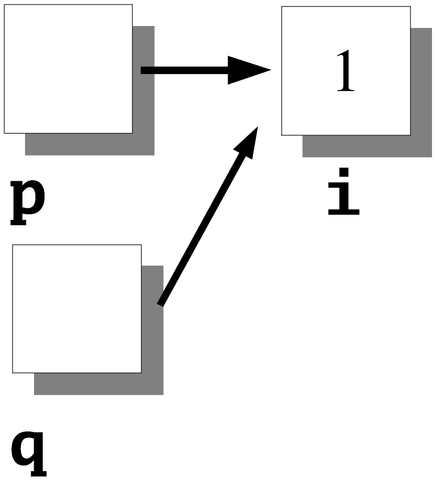
Pointer Assignment

- Can copy pointers of the same type

```
int i, j, *p, *q;
p = &i;
q = p;
```

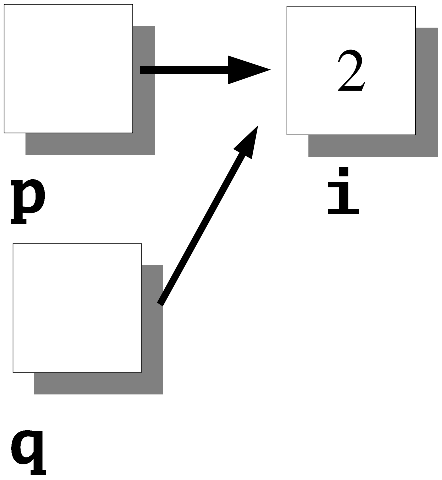


```
*p = 1;
```



Pointer Assignment

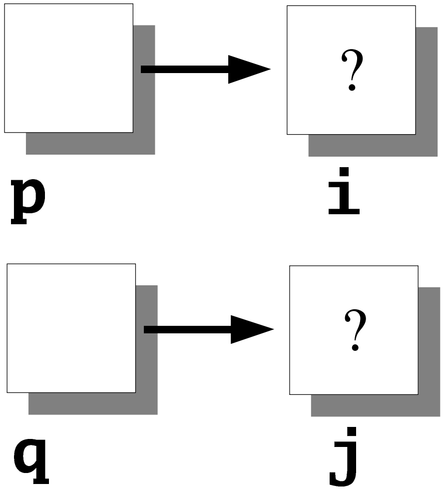
`*q = 2;`

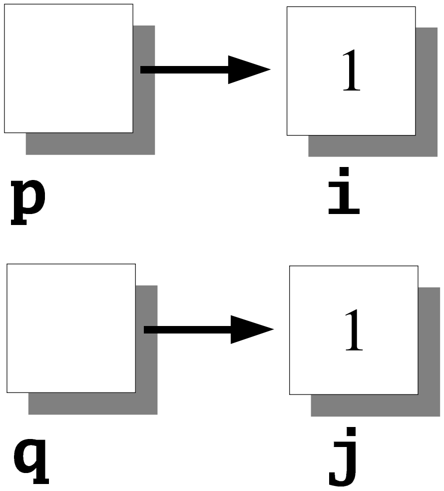


Pointer Assignment

- Don't confuse q=p with *q=*p

```
int i, j, *p, *q;
p = &i;
q = &j;
i = 1;
```





```
*q = *p;
```

Using pointers: example 1

```
int i, j, *pi;

scanf("%d%d", &i, &j);
pi = i > j ? &i: &j;
printf("%d\n", *pi);
```

Using pointers: example 2

```
int i, j;
int *pi = &i;
int *pj = &j;

scanf("%d%d", pi, pj);
printf("%d\n", *pi > *pj ? *pi : *pj);
```

Don't need the & operator in scanf, because pi and pj are already pointers

Using pointers: example 3

Why bother using i and j?

```
int *pi;
int *pj;

scanf("%d%d", pi, pj);
```

What will go wrong?

const pointers and pointers to const

`const int *p;` pointer to a constant integer, the value of **`p`** may change, but the value of **`*p`** can not

`int *const p;` constant pointer to integer; the value of **`*p`** can change, but the value of **`p`** can not

`const int *const p;` constant pointer to constant integer.

Generic Pointers

- A reference to "any" kind of object
  use a variable of type **`void*`**

```
void *p;
```

defines a *generic*, or *typeless* pointer **`p`**. Often cast to **`(T*)p`**

Generic pointers cannot be deferenced. Must cast.

Generic Pointers

Data stored in a memory object can be recovered as a value of a specific data type. For example, for a generic pointer

```
void *p
```

which *points* to an object containing a **`double`** value, you can retrieve this value using the following syntax:

```
*(double*)p
```

Generic Pointers

```
void *p;
char c = 'c';
char *cp = &c;
```

Can assign to p:

```
p = cp;
```

but not dereference it:

```
putchar(*p);
```

Have to use cast:

```
putchar(*(char*)p);
```

NULL macro

- Special "zero" value that is useful to initialize pointers, and then to compare pointer's value:
  - if(p == NULL) ...
- NULL defined in six headers, including **`stdio.h`** and **`stdlib.h`**