ECE 264 Spring 2023

Advanced C Programming

Aravind Machiry
Purdue University

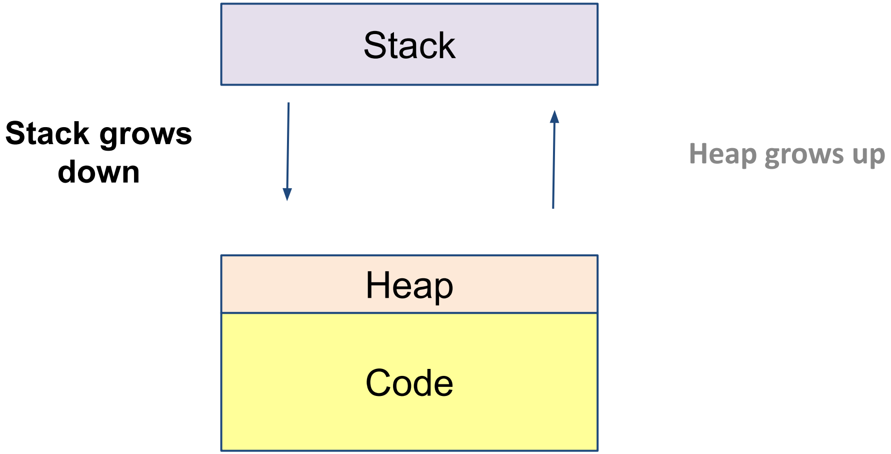
Types of Program Memory

Stack Memory (Stack Segment)	Allocated on Demand (When a function starts).
Heap Memory (Data Segment)	Allocated on Request.
Program Memory (Code Segment)	Allocated at the Beginning.

Dynamic memory allocation

- Memory allocated dynamically based on program usage.

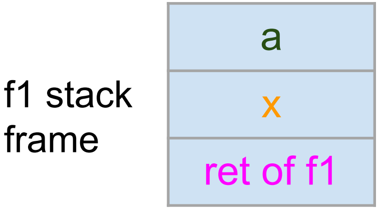
- Why don’t these segments grow in the same direction?



Contents of a Stack Frame

- What do we need to store for each active function?
    - Arguments.
    - Local Variables.
    - Return Address.

How stack frames are created?



f1 stack frame

a

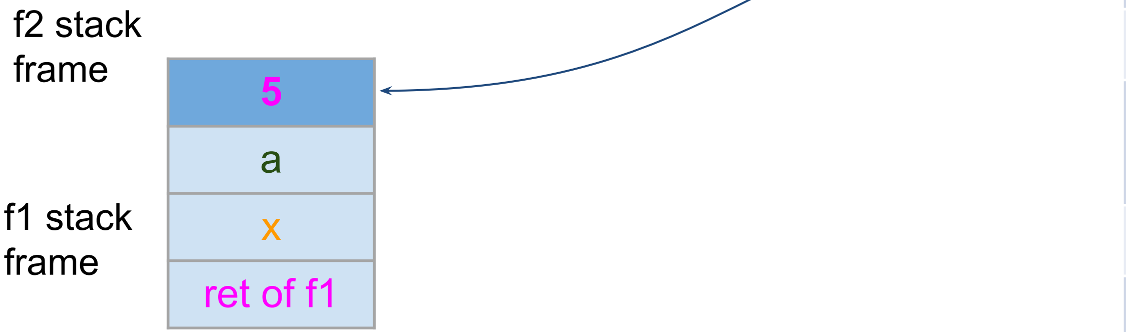
x

ret of f1

```
void f1(int x)
{
    int a;
    a = f3(7, 3.2);
    x = a + 5;
    ...
}
int f3(int y,
          double z)

{
     int m=4;
```

How stack frames are created?

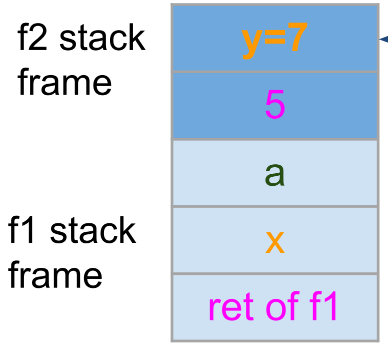
Push return address



```
1   void f1(int x)
2   {
3       int a;
4       a = f3(7, 3.2);
5       x = a + 5;
6       ...
7   }
8   int f3(int y,
            double z)
    {
10      int m=4;
```

How stack frames are created?

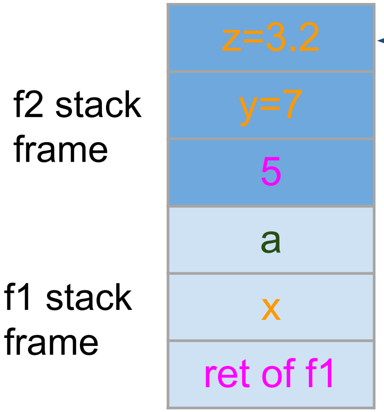
Push argument y



```
void f1(int x)
{
    int a;
    a = f3(7, 3.2);
    x = a + 5;
    ...
}
int f3(int y,
         double z)
{
     int m=4;
```

How stack frames are created?

Push argument z



z=3.2

y=7

5

a

x

ret of f1

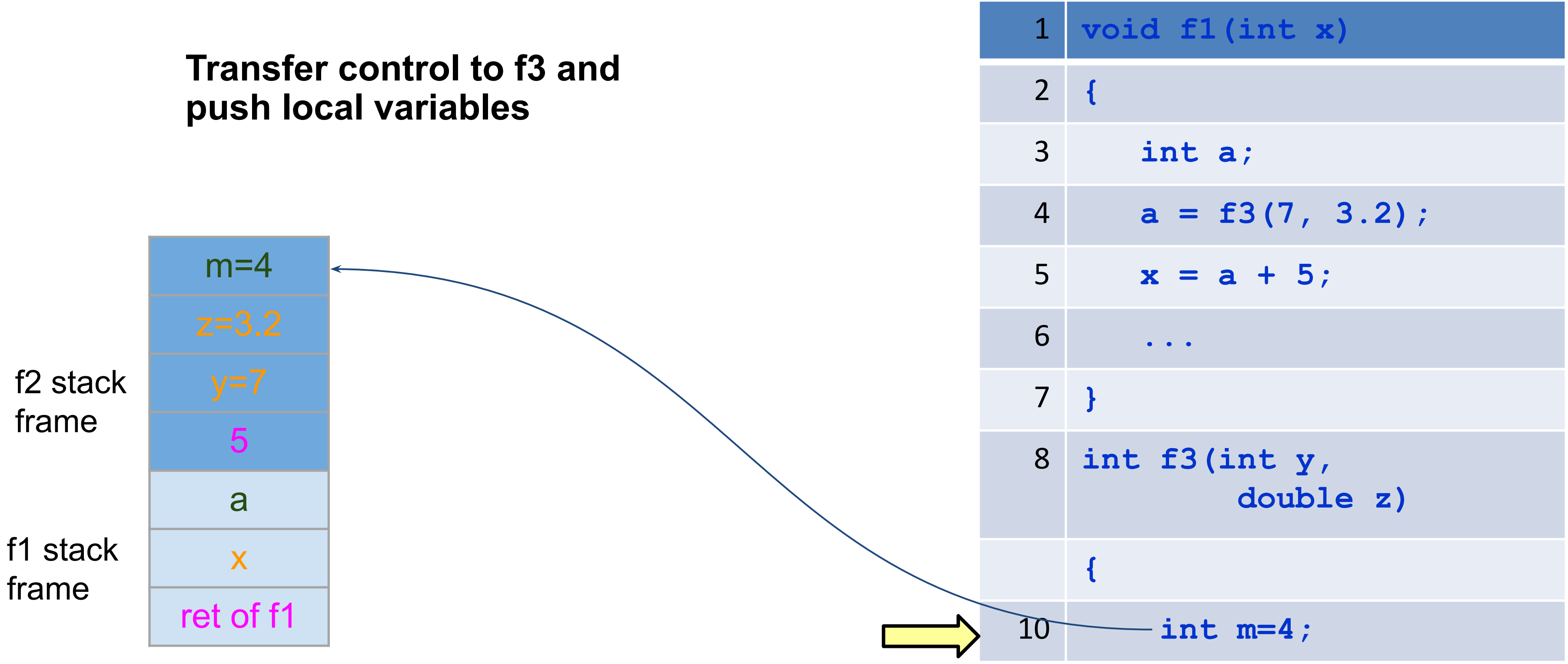
f2 stack frame

f1 stack frame

```
void f1(int x)
{
    int a;
    a = f3(7, 3.2);
    x = a + 5;
    ...
}
int f3(int y,
          double z)
{
     int m=4;
```

How stack frames are created?

Transfer control to f3 and push local variables

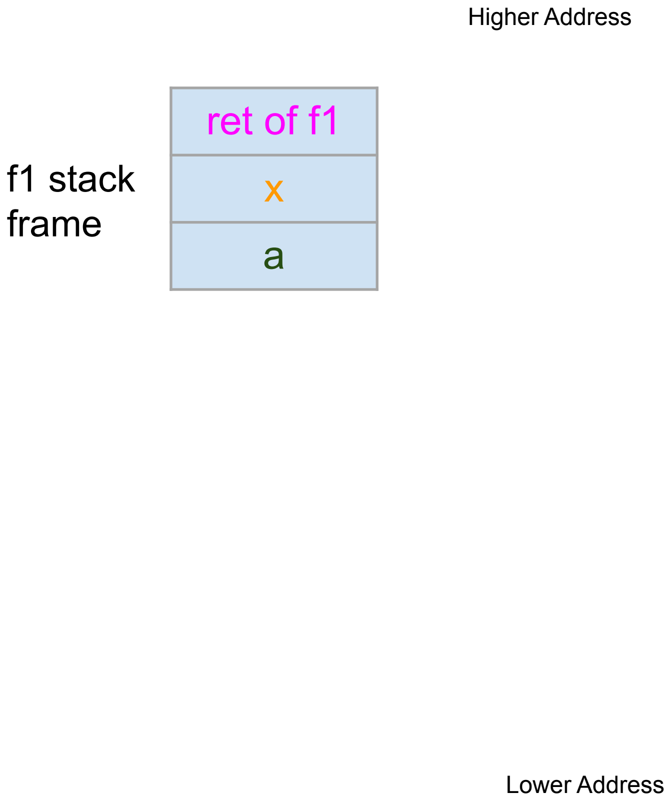


```
void f1(int x)
{
    int a;
    a = f3(7, 3.2);
    x = a + 5;
    ...
}
int f3(int y,
           double z)
{
    int m=4;
```

Stack Growth

- On real machine stack grows downward.
  - Imagine a bottom facing book stack.
  - New stack frames gets allocated at lower addresses.

How stack frames are created (for real)?



Higher Address

f1 stack frame

ret of f1

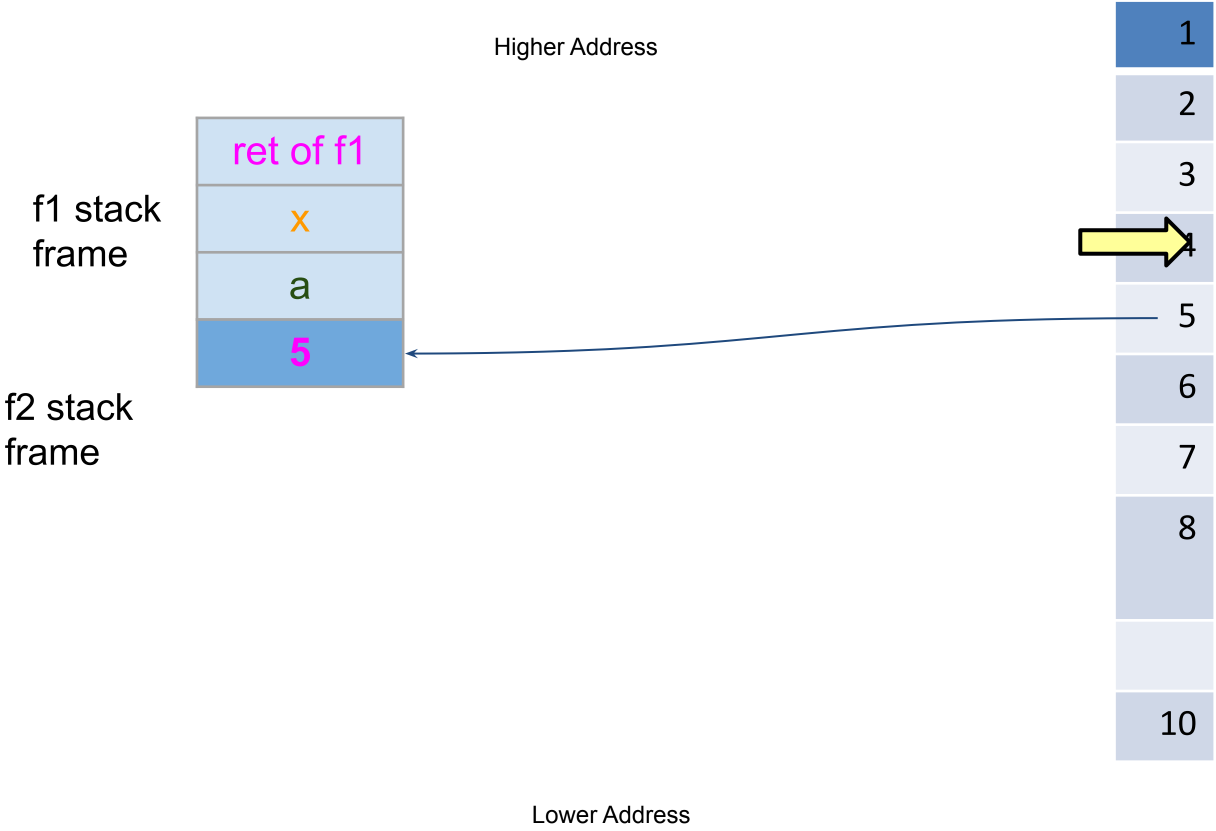
x

a

Lower Address

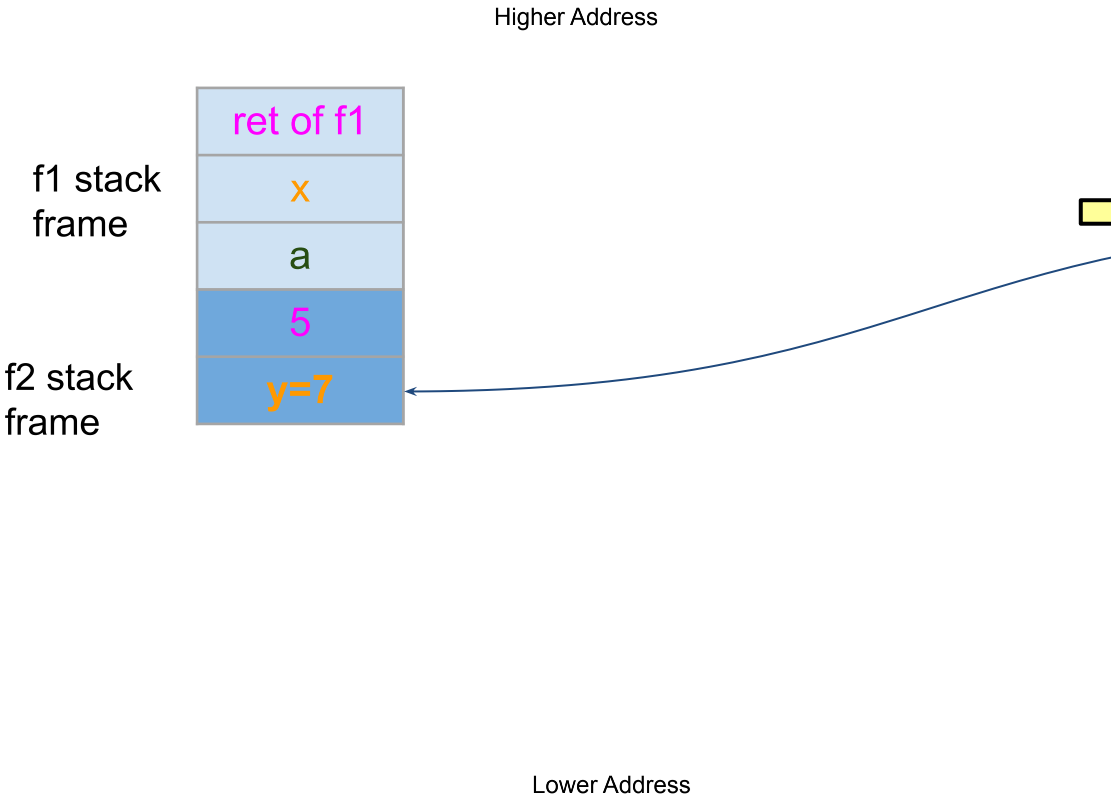
```
void f1(int x)
{
    int a;
    a = f3(7, 3.2);
    x = a + 5;
    ...
}
int f3(int y,
        double z)
{
     int m=4;
```

How stack frames are created (for real)?



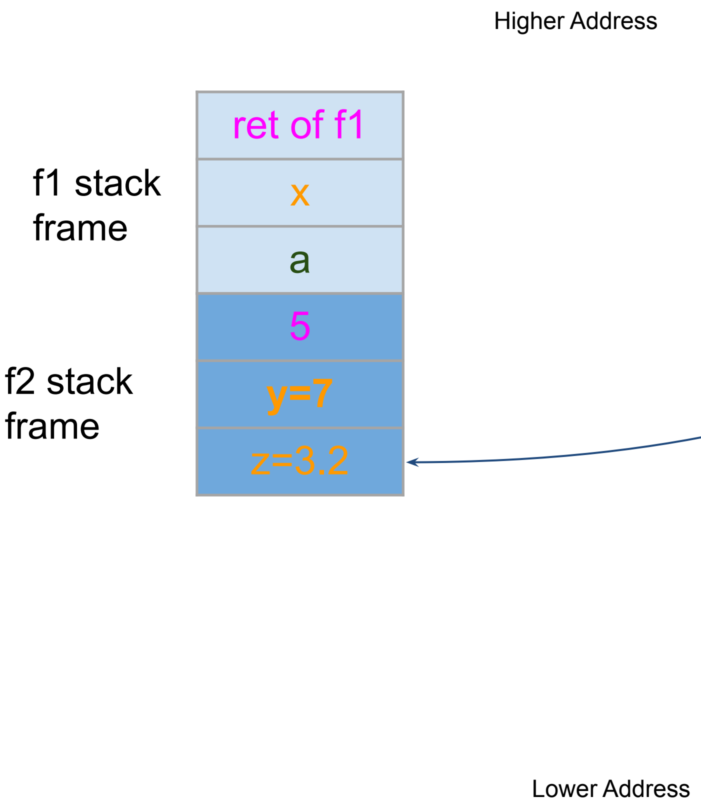
```
void f1(int x)
{
    int a;
    a = f3(7, 3.2);
    x = a + 5;
    ...
}
int f3(int y,
         double z)
{
     int m=4;
```

How stack frames are created (for real)?



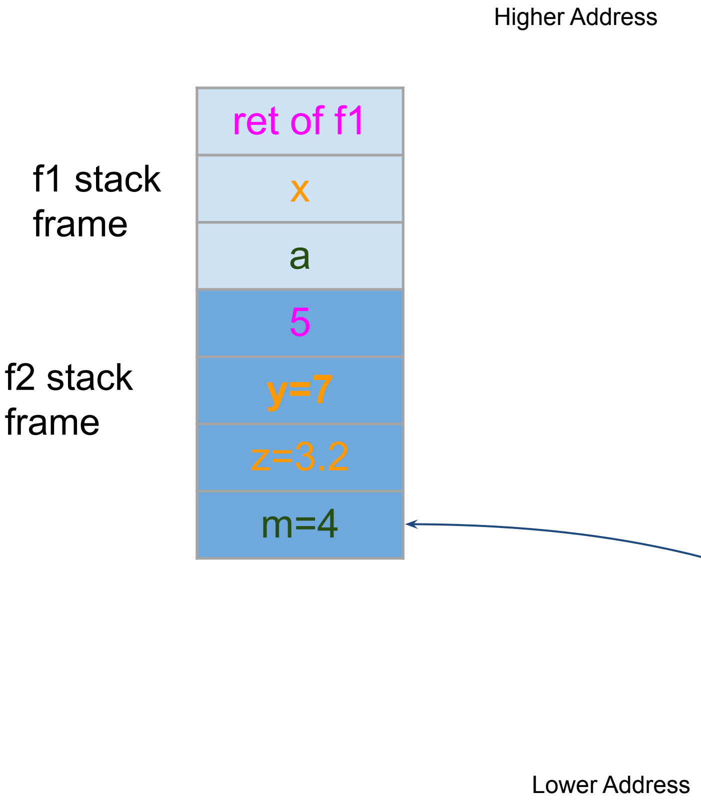
```
void f1(int x)
{
    int a;
    a = f3(7, 3.2);
    x = a + 5;
    ...
}
int f3(int y,
        double z)
{
     int m=4;
```

How stack frames are created (for real)?



```
void f1(int x)
{
    int a;
    a = f3(7, 3.2);
    x = a + 5;
    ...
}
int f3(int y,
            double z)
{
     int m=4;
```

How stack frames are created (for real)?



Higher Address

f1 stack frame

ret of f1

x

a

f2 stack frame

5

y=7

z=3.2

m=4

Lower Address

```
void f1(int x)
{
    int a;
    a = f3(7, 3.2);
    x = a + 5;
    ...
}
int f3(int y,
        double z)
{
    int m=4;
```

Stack frame memory

- Computer access memory using its address.
- Memory has address : n-bit value

  - Stack frame has address
    - All elements in stack frame also has addresses

Stack frame details

Frame	Symbol	Address	Value
Frame of f3	m	106	4
	z	105	3.2
	y	104	7
	RL	103	line 5
Frame of f1	a	102	a =
	x	101	x =
	RL	100	line ?

	1	`void f1(int x)`
	2	`{`
	3	`    int a;`
	4	`    a = f3(7, 3.2);`
	5	`    x = a + 5;`
	6	`    ...`
	7	`}`

	8	`int f3(int y,`
		`          double z)`

		`{`
⇨	10	`    int m=4;`

Stack frame details

For Humans

Frame	Symbol	Address	Value
Frame of f3	m	106	4
	z	105	3.2
	y	104	7
	RL	103	line 5
Frame of f1	a	102	a =
	x	101	x =
	RL	100	line ?

```
 1  void f1(int x)
 2  {
 3      int a;
 4      a = f3(7, 3.2);
 5      x = a + 5;
 6      ...
 7  }
 8  int f3(int y,
            double z)
    {
10      int m=4;
```

The need for pointers

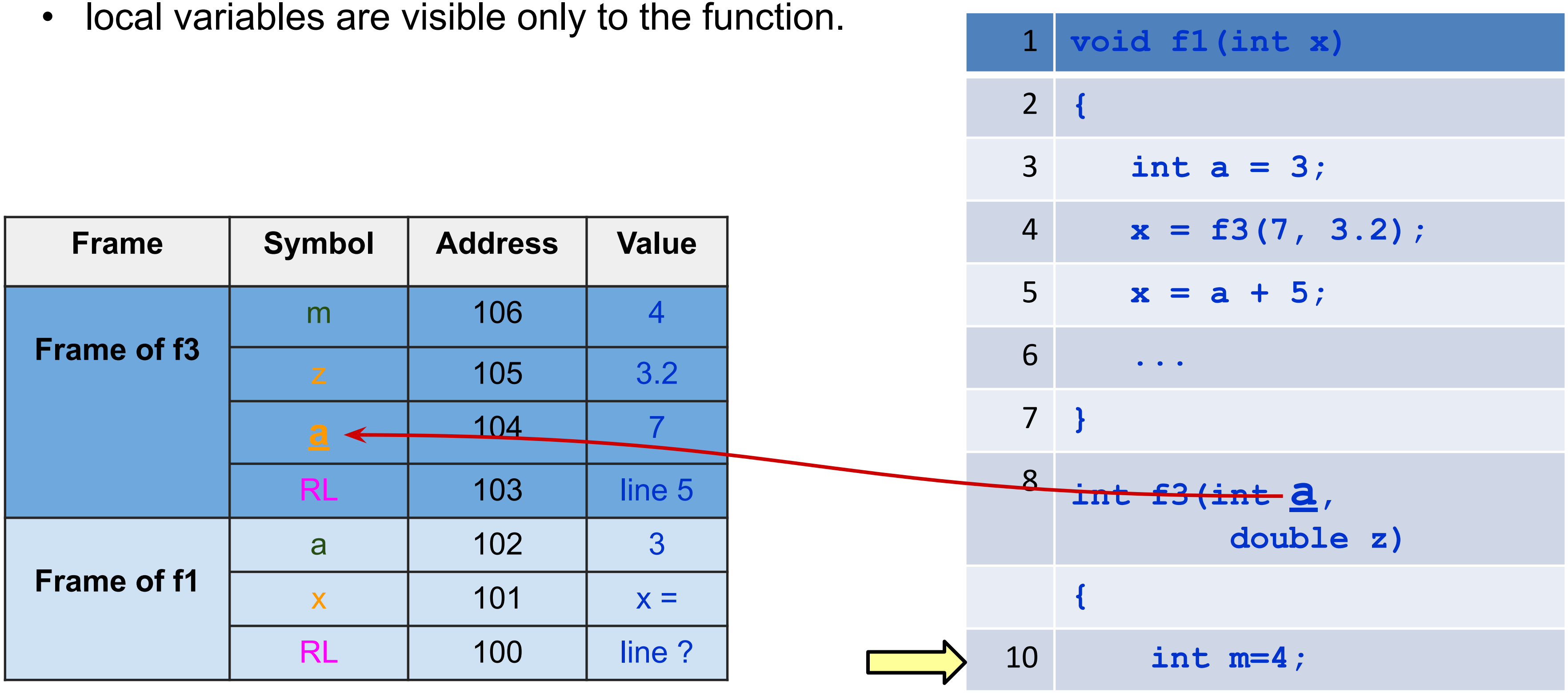
- local variables are visible only to the function.

Frame	Symbol	Address	Value
Frame of f3	m	106	4
	z	105	3.2
	y	104	7
	RL	103	line 5
Frame of f1	a	102	a =
	x	101	x =
	RL	100	line ?

```
1   void f1(int x)
2   {
3       int a;
4       a = f3(7, 3.2);
5       x = a + 5;
6       ...
7   }
8   int f3(int y,
            double z)
    {
10       int m=4;
```

The need for pointers

- local variables are visible only to the function.



Frame	Symbol	Address	Value
Frame of f3	m	106	4
	z	105	3.2
	a	104	7
	RL	103	line 5
Frame of f1	a	102	3
	x	101	x =
	RL	100	line ?

```
1   void f1(int x)
2   {
3       int a = 3;
4       x = f3(7, 3.2);
5       x = a + 5;
6       ...
7   }
8   int f3(int a,
               double z)
    {
10      int m=4;
```

The need for pointers

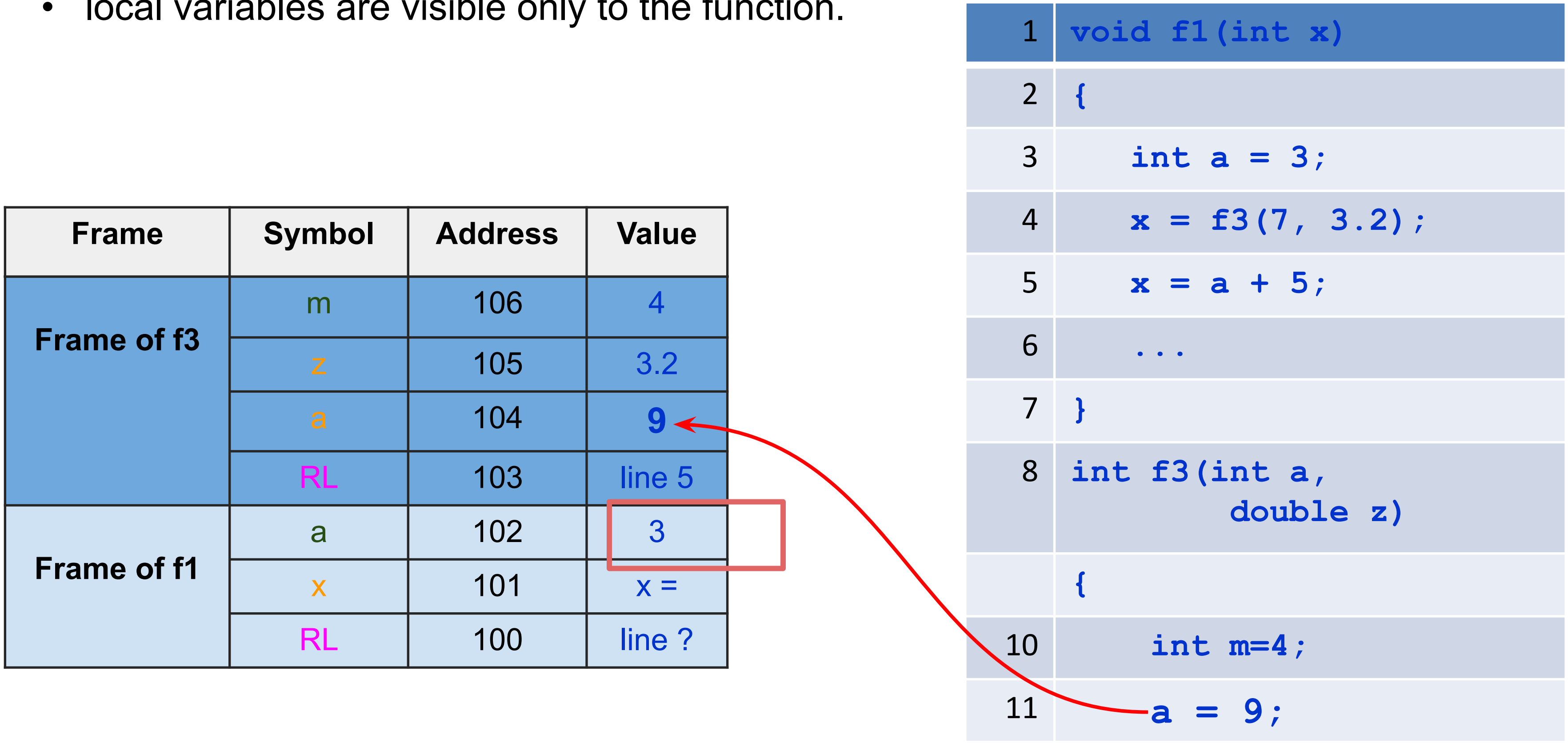
- local variables are visible only to the function.

Frame	Symbol	Address	Value
Frame of f3	m	106	4
	z	105	3.2
	a	104	**8**
	RL	103	line 5
Frame of f1	a	102	3
	x	101	x =
	RL	100	line ?

```
 1  void f1(int x)
 2  {
 3      int a = 3;
 4      x = f3(7, 3.2);
 5      x = a + 5;
 6      ...
 7  }
 8  int f3(int a,
              double z)
    {
10       int m=4;
11     a = 8;
```

The need for pointers

- local variables are visible only to the function.



Frame	Symbol	Address	Value
Frame of f3	m	106	4
	z	105	3.2
	a	104	**9**
	RL	103	line 5
Frame of f1	a	102	3
	x	101	x =
	RL	100	line ?

```
 1  void f1(int x)
 2  {
 3      int a = 3;
 4      x = f3(7, 3.2);
 5      x = a + 5;
 6      ...
 7  }
 8  int f3(int a,
            double z)
    {
10      int m=4;
11      a = 9;
```

The need for pointers

- We may need other functions modify the local variables.
- Function to swap 2 numbers.

swap function

```
int a = 5;
int b = 7;
swap(a, b);
// a should be 7 here
// b should be 5 here
```

swap function

```
int a = 5;
int b = 7;
swap(a, b);
// a should be 7 here
// b should be 5 here
```

Attempt 1 (wrong)

```
int a = 5;
int b = 7;
a = swap(a, b);
// both a and b are 7
int swap (int x, int y)
{
    return y;
}
```

swap function

```
int a = 5;
int b = 7;
swap(a, b);
// a should be 7 here
// b should be 5 here
```

Attempt 2 (wrong)

```
int a = 5;
int b = 7;
swap(a, b);
// a and b unchanged
void swap (int a, int b)
{
   int t = a;
   a = b;
   b = t;
}
```

Attempt 2 (wrong)

Frame	Symbol	Address	Value
	a	101	5
	b	100	7

```
int a = 5;
int b = 7;
swap(a, b);
// a and b unchanged
…
void swap (int a, int b)
{
int t = a;
a = b;
b = t;
}
```

Frame	Symbol	Address	Value
swap	t	106	**5**
	b	105	7
	a	104	5
	RL	103	line 4
	a	101	5
	b	100	7

Attempt 2 (wrong)

```
1.  int a = 5;
2.  int b = 7;
3.  swap(a, b);
4.  // a and b unchanged
…
a.  void swap (int a, int b)
b.  {
c.  int t = a;
d.  a = b;
e.  b = t;
f.  }
```

Attempt 2 (wrong)

Frame	Symbol	Address	Value
swap	t	106	5
	b	105	7
	a	104	5 → 7
	RL	103	line 4
	a	101	5
	b	100	7

```
1.  int a = 5;
2.  int b = 7;
3.  swap(a, b);
4.  // a and b unchanged
…
a.  void swap (int a, int b)
b.  {
c.   int t = a;
d.   a = b;
e.   b = t;
f.  }
```

Attempt 2 (wrong)

Frame	Symbol	Address	Value
swap	t	106	5
	b	105	7 → 5
	a	104	7
	RL	103	line 4
	a	101	5
	b	100	7

```
1.  int a = 5;
2.  int b = 7;
3.  swap(a, b);
4.  // a and b unchanged
…
a.  void swap (int a, int b)
b.  {
c.   int t = a;
d.   a = b;
e.   b = t;
f.  }
```

Frame	Symbol	Address	Value
swap	t	106	5
	b	105	5
	a	104	7
	RL	103	line 4
	a	101	5
	b	100	7

Attempt 2 (wrong)

```
1.  int a = 5;
2.  int b = 7;
3.  swap(a, b);
4.  // a and b unchanged
…
a.  void swap (int a, int b)
b.  {
c.   int t = a;
d.   a = b;
e.   b = t;
f.  }
```

Attempt 2 (wrong)

Frame	Symbol	Address	Value
~~swap~~	~~t~~	~~106~~	~~5~~
	~~b~~	~~105~~	~~5~~
	~~a~~	~~104~~	~~7~~
	~~RL~~	~~103~~	~~line 4~~
	a	101	5
	b	100	7

```
1.  int a = 5;
2.  int b = 7;
3.  swap(a, b);
4.  // a and b unchanged
…
a.  void swap (int a, int b)
b.  {
c.   int t = a;
d.   a = b;
e.   b = t;
f.  }
```

Frame	Symbol	Address	Value
swap	t	106	5
	b	105	5
	a	104	7
	RL	103	line 4
	a	101	5
	b	100	7

Attempt 2 (wrong)

```
1.  int a = 5;
2.  int b = 7;
3.  swap(a, b);
4.  // a and b unchanged
…
a.  void swap (int a, int b)
b.  {
c.   int t = a;
d.   a = b;
e.   b = t;
f.  }
```

Frame	Symbol	Address	Value
swap	t	106	5
	y	105	5
	x	104	7
	RL	103	line 4
	a	101	5
	b	100	7

Attempt 2 (wrong)

```
1.  int a = 5;
2.  int b = 7;
3.  swap(a, b);
4.  // a and b unchanged
…
a.  void swap (int x, int y)
b.  {
c.  int t = x;
d.  x = y;
e.  y = t;
f.  }
```

pointer: a variable whose value is an address

pointer: variable storing address

```
int t = 5;
int * p =& t;
```

same

```
int t = 5;
int * p;
p =& t;
```

***** `p` means `p` is a pointer (`p`’s value is an address)

`int` `* p` means the address stores an integer

& `t` gets the address of `t`

Symbol	Address	Value
p	101	A100
t	100	5

How to use pointers?

```
int t = 5;
int * p;      // create a pointer
p = & t;
* p = -6;     // left-hand-side (LHS) of =
int s = * p;     // right-hand-side (RHS)of =
```

LHS	1. take p’s value as an address 2. go to that address	3. modify the value at that address
RHS		3. read the value at that address

How to use pointers?

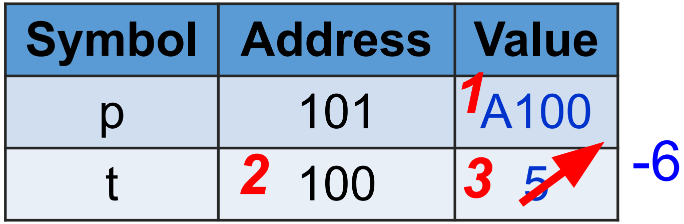
```
a.  int t = 5;
b.  int * p;
c.  p = & t;
d.  * p = -6;    // LHS
e.  int s = * p;    // RHS
```

Symbol	Address	Value
p	101	A100
t	100	5

LHS	1. take p’s value as an address 2. go to that address	3. modify the value at that address
RHS		3. read the value at that address

How to use pointers?

```
a.  int t = 5;
b.  int * p;
c.  p = & t;
d.  * p = -6;    // LHS
e.  int s = * p;    // RHS
```



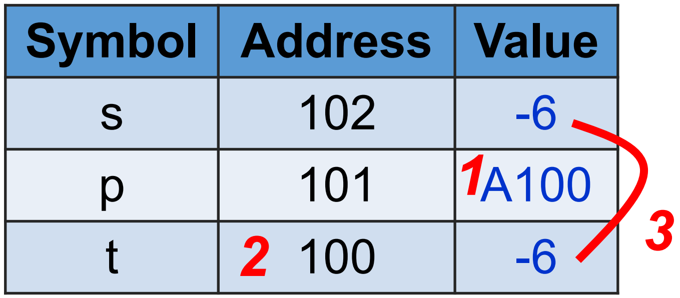
Symbol	Address	Value
p	101	1 A100
t	2 100	3 5

-6

LHS	1. take p's value as an address 2. go to that address	3. modify the value at that address
RHS		3. read the value at that address

How to use pointers?

```
a.  int t = 5;
b.  int * p;
c.  p = & t;
d.  * p = -6;    // LHS
e.  int s = * p;    // RHS
```



Symbol	Address	Value
s	102	-6
p	101	A100
t	100	-6

LHS	1. take p’s value as an address 2. go to that address	3. modify the value at that address
RHS		3. read the value at that address

Confusion of *

three ways of using *:

type can be int, double, char …

```
1.  type * p;    // create a pointer
2.  * p =        // left hand side of =
3.     = * p;    // right hand side of =
```

Confusion of *

~~three~~ four ways of using *:

type can be int, double, char …

```
1.  type * p;    // create a pointer
2.  * p =        // left hand side of =
3.     = * p;    // right hand side of =
4.  int t = 5 * 9; // multiplication; t is 45
```

correct swap function

```
int a = 5;
int b = 7;
swap(& a, & b);
// a is 7 and b is 5
```

…

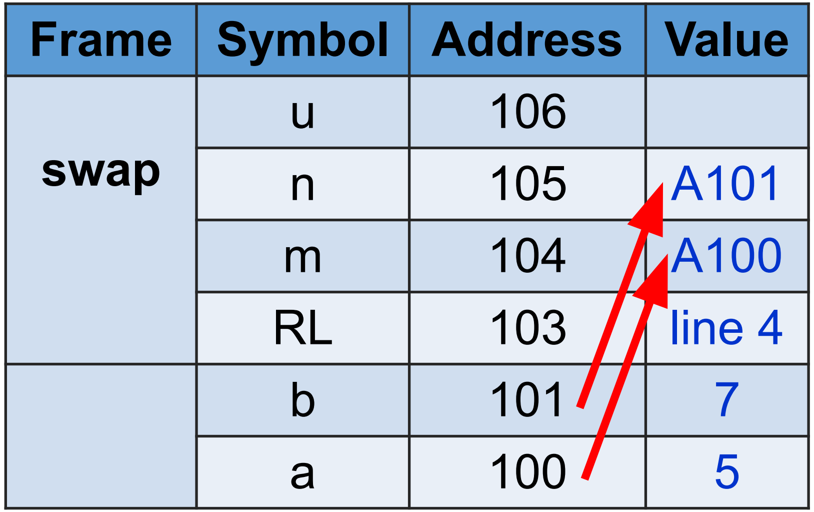
```
void swap (int * m, int * n)
{
int u = * m;
*m = *n;
*n = u;
}
```

Frame	Symbol	Address	Value
	b	101	7
	a	100	5

correct swap function

```
1.  int a = 5;
2.  int b = 7;
3.  swap(& a, & b);
4.  // a is 7 and b is 5
…
a.  void swap (int * m, int * n)
b.  {
c.  int u = * m;
d.  *m = *n;
e.  *n = u;
f.  }
```

addresses of a and b



Frame	Symbol	Address	Value
swap	u	106	
	n	105	A101
	m	104	A100
	RL	103	line 4
	b	101	7
	a	100	5

correct swap function

```
1.  int a = 5;
2.  int b = 7;
3.  swap(& a, & b);
4.  // a is 7 and b is 5
…
a.  void swap (int * m, int * n)
b.  {
c.  int u = * m; // RHS
d.  *m = *n;
e.  *n = u;
f.  }
```

addresses of a and b

Frame	Symbol	Address	Value
swap	u	106	5 3
	n	105	A101
	m	104	1 A100
	RL	103	line 4
	b	101	7
	a	2 100	5

correct swap function

```
1.  int a = 5;
2.  int b = 7;
3.  swap(& a, & b);
4.  // a is 7 and b is 5
…
a.  void swap (int * m, int * n)
b.  {
c.   int u = *
d.   *m = *n; // RHS ⇒ LHS
e.   *n = u;
f.  }
```

addresses of a and b

```
int t = *n;
* m = t;
```

Frame	Symbol	Address	Value
swap	t	107	7 3
	u	106	5
	n	105	1 A101
	m	104	A100
	RL	103	line 4
	b	2 101	7
	a	100	5

correct swap function

```
1.  int a = 5;
2.  int b = 7;
3.  swap(& a, & b);
4.  // a is 7 and b is 5
…
a.   void swap (int * m, int * n)
b.   {
c.    int u = *
d.    *m = *n; // RHS ⇒ LHS
e.    *n = u;
f.   }
```

addresses of a and b

```
int t = *n;
* m = t;
```

Frame	Symbol	Address	Value
swap	t	107	7 3
	u	106	5
	n	105	A101
	m	104	1 A100
	RL	103	line 4
	b	101	7
	a	2 100	5

7

correct swap function

```
1.  int a = 5;
2.  int b = 7;
3.  swap(& a, & b);
4.  // a is 7 and b is 5
…
a.  void swap (int * m, int * n)
b.  {
c.   int u = * m; // RHS
d.   *m = *n; // RHS ⇒ LHS
e.   *n = u;  // LHS
f.  }
```

addresses of
a and b

Frame	Symbol	Address	Value
swap	t	107	7 3
	u	106	5
	n	105	1 A101
	m	104	A100
	RL	103	line 4
	b	2 101	~~7~~ 5
	a	100	7

correct swap function

```
1.  int a = 5;
2.  int b = 7;
3.  swap(& a, & b);
4.  // a is 7 and b is 5
…
a.  void swap (int * m, int * n)
b.  {
c.   int u = * m; // RHS
d.   *m = *n; // RHS ⇒ LHS
e.   *n = u;  // LHS
f.   }
```

addresses of a and b

Frame	Symbol	Address	Value
swap	t	107	7
	u	106	5
	n	105	A101
	m	104	A100
	RL	103	line 4
	b	101	5
	a	100	7

correct swap function

```
1.  int a = 5;
2.  int b = 7;
3.  swap(& a, & b);
4.  // a is 7 and b is 5
…
a.  void swap (int * m, int * n)
b.  {
c.   int u = * m; // RHS
d.   *m = *n; // RHS ⇒ LHS
e.   *n = u;  // LHS
f.  }
```

addresses of a and b

Frame	Symbol	Address	Value
~~swap~~	~~t~~	~~107~~	~~7~~
	~~u~~	~~106~~	~~5~~
	~~n~~	~~105~~	~~A101~~
	~~m~~	~~104~~	~~A100~~
	~~RL~~	~~103~~	~~line 4~~
	b	101	5
	a	100	7

RHS rules without =

```
int a = 2020;
int * p = & a;
printf("%d\n", * p); // RHS
f(* p); // RHS
…
void f(int t) // t is 2020
{
    …
}
```

Symbol	Address	Value
p	101	1 A100
a	2 100	3 2020

Data Types

- Data types specify the information and operations.
- Data types specify the amount of information for each entity.
- `int, char, double` are data types
- Programmers can create new data types, such as car, desk, phone, light bulb.

Type	Information	Operation
Car	engine size, number of seats, fuel tank	accelerate, decelerate
Desk	width, height, length	write on
Phone	screen size, amount of storage	call, text message, map

- Do not mix data types.

Type Rules

creation	**value**	**type**
`t1 x;`	`x`’s value is `t1`	`&` `x` is `t1 *` (address of x)
`t2 * y;`	`y`’s value is `t2 *`	`*` `y` is `t2` (LHS or RHS)
`t1` and `t2` are types, such as int, char, double, car, phone, desk …		

Understand Syntax About Pointers

```
int a = 5;
int b = 7;
int * p;
p = & a;
p = & b;
p = a; // error
int * q;
q = p;
```

```
int * p = & a;
```

Symbol	Address	Value
q	103	
p	102	A100
b	101	5
a	100	7

Understand Syntax About Pointers

```
1.  int a = 5;
2.  int b = 7;
3.  int * p;
4.  p = & a;
5.  p = & b;  ⇦
6.  p = a;  // error: p is int *, a is int
7.  int * q;
8.  q = p;
```

Symbol	Address	Value
q	103	
p	102	A101
b	101	5
a	100	7

Understand Syntax About Pointers

1. `int a = 5;`
2. `int b = 7;`
3. `int * p;`
4. `p = & a;`
5. `p = & b;`
6. `p = a; // error`
7. `int * q;`
8. `q = p;`

Symbol	Address	Value
q	103	A101
p	102	A101
b	101	5
a	100	7

```
int a = 5;
int b = 7;
int * p = & b;
int * q = p;
* p = -264;
int c = * q;
q = & c;
c = q; // error
c = & a; // error
```

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

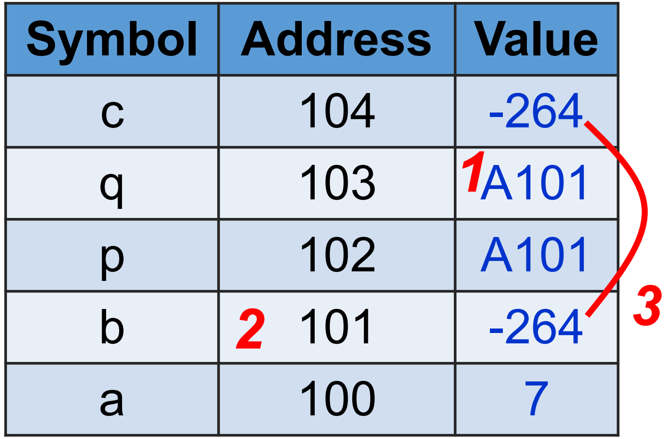
Symbol	Address	Value
q	103	A101
p	102	A101
b	101	5
a	100	7

```
int a = 5;
int b = 7;
int * p = & b;
int * q = p;
* p = -264;
int c = * q;
q = & c;
c = q; // error
c = & a; // error
```

```
b = -264;
```

Symbol	Address	Value
q	103	A101
p	102	1 A101
b	2 101	3 ~~5~~ → -264
a	100	7

```
int a = 5;
int b = 7;
int * p = & b;
int * q = p;
* p = -264;
int c = * q;
q = & c;
c = q; // error
c = & a; // error
```



Symbol	Address	Value
c	104	-264
q	103	A101
p	102	A101
b	101	-264
a	100	7

```
1. int a = 5;
2. int b = 7;
3. int * p = & b;
4. int * q = p;
5. * p = -264;
6. int c = * q;
7. q = & c;
8. c = q; // error
9. c = & a; // error
```

Symbol	Address	Value
c	104	-264
q	103	A104
p	102	A101
b	101	-264
a	100	7

```
int a = 5;
int b = 7;
int * p = & b;
int * q = p;
* p = -264;
int c = * q;
q = & c;
c = q;  // error, int and int *
c = & a;  // error
& a = …;  // error, cannot change address
p = 2020; // error
```

Symbol	Address	Value
c	104	-264
q	103	A104
p	102	A101
b	101	-264
a	100	7

Pointer Rules

- You can never change anything’s address.
- You can change only values.
- You must not mix pointers with non-pointers.

```
1. int b = 7;
2. int * p = & b;
3. int * q = p;
4. p = -264;  // error, -264 is int, not pointer
5. int c = q; // error, c is int, not pointer
6. b = p;     // error, b is int, not pointer
```

```
* p = -264 // ok
```

Type Mismatch

- Mixing types is common mistakes.
- Programs will behave in surprising ways.
- Most of time, gcc can detect type mismatch.
- If gcc gives warnings or errors, you must correct them.

Student: My program does not work. I have not slept for two days.

Teaching Assistant: Do you notice this gcc warning about types?

Student: I will worry about that after making my program work.

Teaching Assistant: This is your problem. You need to add * in front of a pointer.

Student: It works now. I spent 30 **hours** on finding this problem.

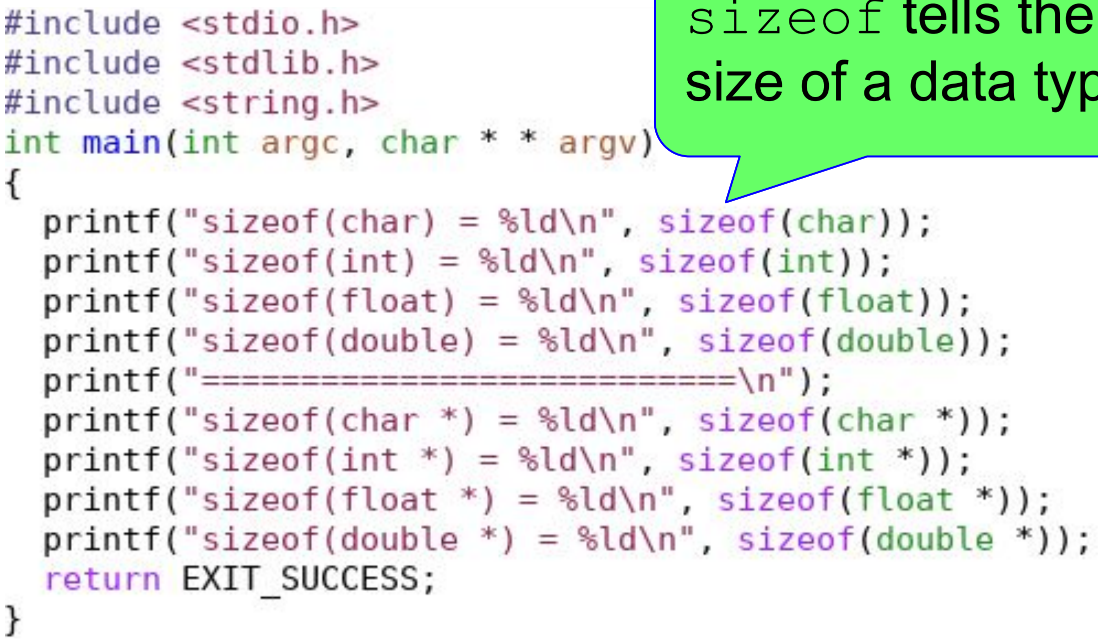
Teaching Assistant: It took me 30 **seconds** because gcc told me the problem.

Student: Thank you.

```
int a = 5;
int b = 7;
int * p = & b;
int * q = p;
* p = -264;
int c = * q;
q = & c;
```

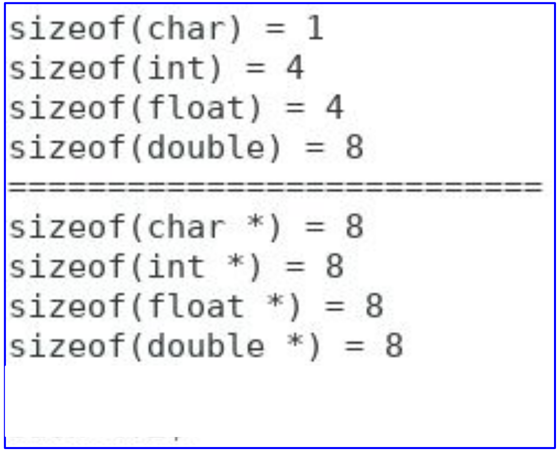
Symbol	Address	Value
c	104	
q	103	
p	102	
b	101	
a	100	

Understand sizes of types



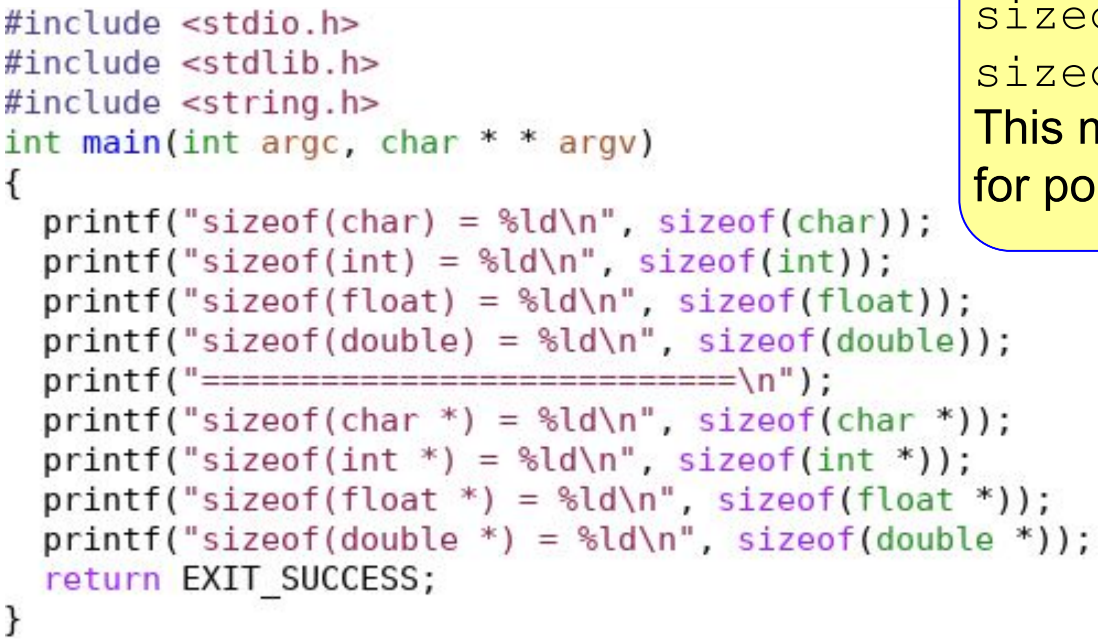
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char * * argv)
{
  printf("sizeof(char) = %ld\n", sizeof(char));
  printf("sizeof(int) = %ld\n", sizeof(int));
  printf("sizeof(float) = %ld\n", sizeof(float));
  printf("sizeof(double) = %ld\n", sizeof(double));
  printf("===========================\n");
  printf("sizeof(char *) = %ld\n", sizeof(char *));
  printf("sizeof(int *) = %ld\n", sizeof(int *));
  printf("sizeof(float *) = %ld\n", sizeof(float *));
  printf("sizeof(double *) = %ld\n", sizeof(double *));
  return EXIT_SUCCESS;
}
```

`sizeof` tells the size of a data type



```
sizeof(char) = 1
sizeof(int) = 4
sizeof(float) = 4
sizeof(double) = 8
===========================
sizeof(char *) = 8
sizeof(int *) = 8
sizeof(float *) = 8
sizeof(double *) = 8
```

Understand sizes of types

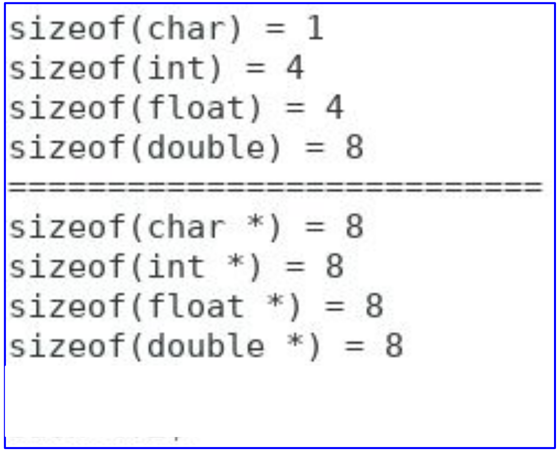


```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char * * argv)
{
  printf("sizeof(char) = %ld\n", sizeof(char));
  printf("sizeof(int) = %ld\n", sizeof(int));
  printf("sizeof(float) = %ld\n", sizeof(float));
  printf("sizeof(double) = %ld\n", sizeof(double));
  printf("===========================\n");
  printf("sizeof(char *) = %ld\n", sizeof(char *));
  printf("sizeof(int *) = %ld\n", sizeof(int *));
  printf("sizeof(float *) = %ld\n", sizeof(float *));
  printf("sizeof(double *) = %ld\n", sizeof(double *));
  return EXIT_SUCCESS;
}
```

`sizeof (char)` is 1

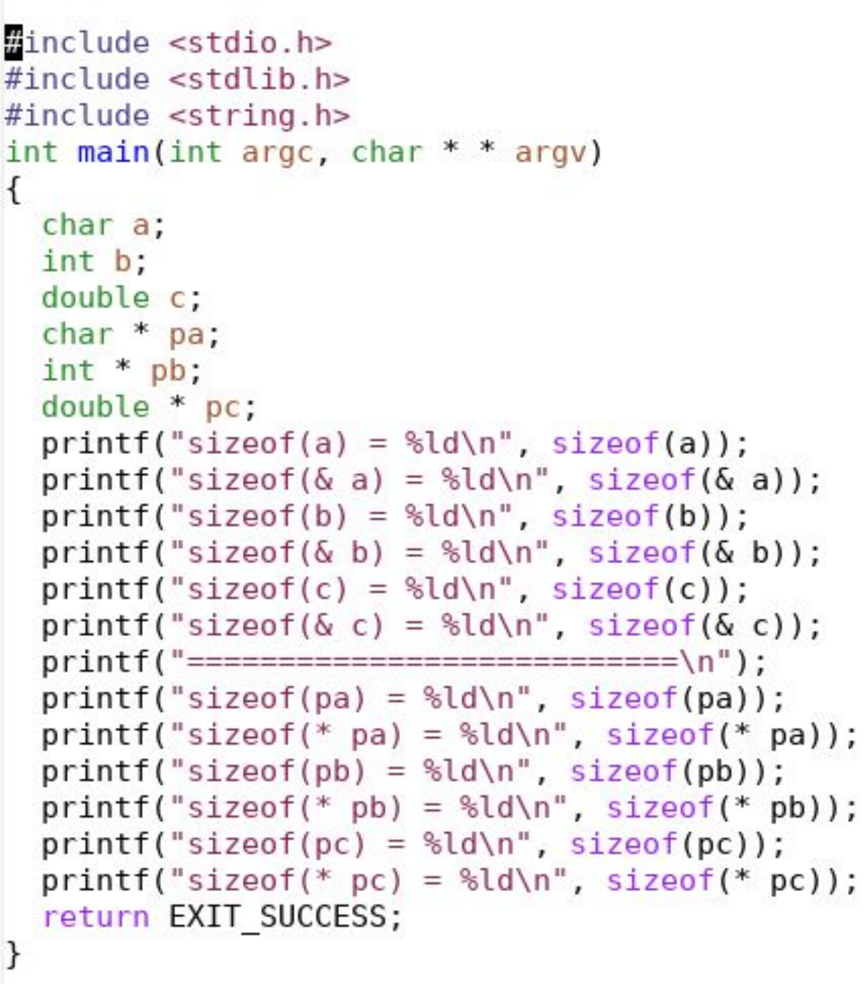
`sizeof(int)` depends on machine

This machine uses 64 bits (8 bytes) for pointers

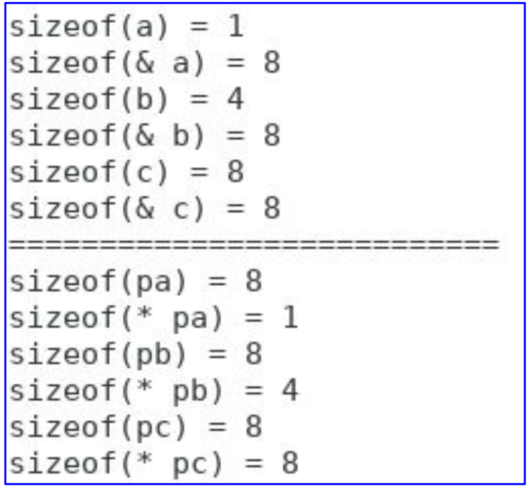


```
sizeof(char) = 1
sizeof(int) = 4
sizeof(float) = 4
sizeof(double) = 8
===========================
sizeof(char *) = 8
sizeof(int *) = 8
sizeof(float *) = 8
sizeof(double *) = 8
```

`sizeof` can also be used for variables



```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char * * argv)
{
  char a;
  int b;
  double c;
  char * pa;
  int * pb;
  double * pc;
  printf("sizeof(a) = %ld\n", sizeof(a));
  printf("sizeof(& a) = %ld\n", sizeof(& a));
  printf("sizeof(b) = %ld\n", sizeof(b));
  printf("sizeof(& b) = %ld\n", sizeof(& b));
  printf("sizeof(c) = %ld\n", sizeof(c));
  printf("sizeof(& c) = %ld\n", sizeof(& c));
  printf("===========================\n");
  printf("sizeof(pa) = %ld\n", sizeof(pa));
  printf("sizeof(* pa) = %ld\n", sizeof(* pa));
  printf("sizeof(pb) = %ld\n", sizeof(pb));
  printf("sizeof(* pb) = %ld\n", sizeof(* pb));
  printf("sizeof(pc) = %ld\n", sizeof(pc));
  printf("sizeof(* pc) = %ld\n", sizeof(* pc));
  return EXIT_SUCCESS;
}
```



```
sizeof(a) = 1
sizeof(& a) = 8
sizeof(b) = 4
sizeof(& b) = 8
sizeof(c) = 8
sizeof(& c) = 8
===========================
sizeof(pa) = 8
sizeof(* pa) = 1
sizeof(pb) = 8
sizeof(* pb) = 4
sizeof(pc) = 8
sizeof(* pc) = 8
```

Do not mix types

```
int a = 123;
char * p;      // sizeof (*p) = 1
p = & a;    // error: sizeof(a)= 4
* p = 2020;    // 2020 is bigger than one byte
int b = 264;
double * q;    // sizeof(*q) = 8
q = & b;    // error: sizeof (b) = 4
double c = * q;
```

Match Types

```
1. int a = 123;
2. int * p;
3. p = & a;
4. * p = 2020;
5. double b = 26.4;
6. double * q;
7. q = & b;
8. double c = * q;
```

int and int *

double and double *

Review: correct swap function

```
1.  int a = 5;
2.  int b = 7;
3.  swap(& a, & b);
4.  // a is 7 and b is 5
…
a.  void swap (int * m, int * n)
b.  {
c.   int u = * m; // RHS
d.   *m = *n; // RHS ⇒ LHS
e.   *n = u;  // LHS
f.  }
```

addresses of a and b

Frame	Symbol	Address	Value
swap	t	107	7
	u	106	5
	n	105	A101
	m	104	A100
	RL	103	line 4
	b	101	5
	a	100	7

Types of Program Memory

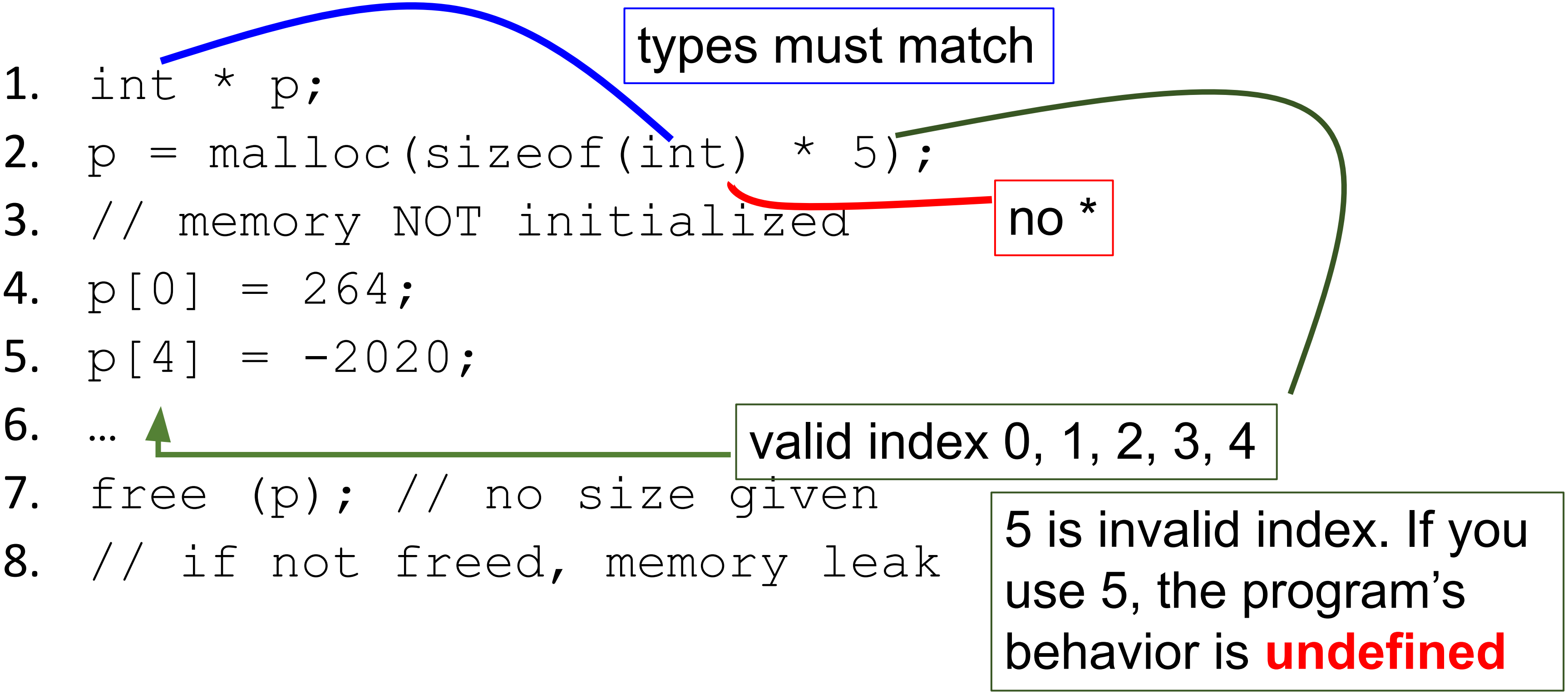
Stack Memory (Stack Segment)	Allocated on Demand (When a function starts).
Heap Memory (Data Segment)	Allocated on Request.
Program Memory (Code Segment)	Allocated at the Beginning.

Stack vs. Heap Memory

	Stack Memory	**Heap Memory**
Rules	Last-in, First-out	Flexible
Size	Determined at compilation time	Run time
Responsibility	Compiler	**Programmer**
Visibility	Current frame or lower frames (use pointers)	All functions, by pointers
Pointer	Not necessary	**Must**
Computing Model*	Push-Down Automata	Turing Machine
Capability	Limited	General
Relationship	Proper subset of heap memory	Superset of stack memory

* Please find a book on the topic of Computation or Automata Theory

Heap Memory: array of five integers



```
int * p;
p = malloc(sizeof(int) * 5);
// memory NOT initialized
p[0] = 264;
p[4] = -2020;
…
free (p); // no size given
// if not freed, memory leak
```

types must match

no *

valid index 0, 1, 2, 3, 4

5 is invalid index. If you use 5, the program’s behavior is **undefined**

Heap Memory

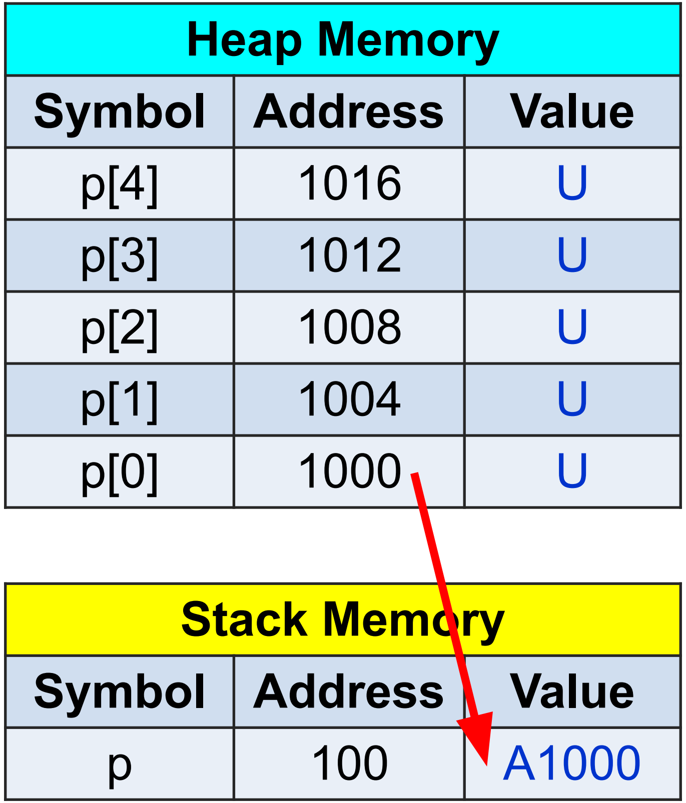
It may not be zero

1. `int * p; // p's value is unknown (U)`

Stack Memory		
Symbol	**Address**	**Value**
p	100	U

Heap Memory

```
int * p;
p = malloc(sizeof(int) * 5);
// memory NOT initialized
// malloc decides the address
```



Heap Memory		
Symbol	**Address**	**Value**
p[4]	1016	U
p[3]	1012	U
p[2]	1008	U
p[1]	1004	U
p[0]	1000	U

Stack Memory		
Symbol	**Address**	**Value**
p	100	A1000

Heap Memory

```
int * p;
p = malloc(sizeof(int) * 5);
// memory NOT initialized
// malloc decides the address
```

&p[k] = &p[0] + k · size of one element

Heap Memory		
Symbol	**Address**	**Value**
p[4]	1016	U
p[3]	1012	U
p[2]	1008	U
p[1]	1004	U
p[0]	1000	U

Stack Memory		
Symbol	**Address**	**Value**
p	100	A1000

Heap Memory

```
int * p;
p = malloc(sizeof(int) * 5);
// memory NOT initialized
p[0] = 264;
```

Heap Memory		
Symbol	Address	Value
p[4]	1016	U
p[3]	1012	U
p[2]	1008	U
p[1]	1004	U
p[0]	1000	264

Stack Memory		
Symbol	Address	Value
p	100	A1000

Heap Memory

```
int * p;
p = malloc(sizeof(int) * 5);
// memory NOT initialized
p[0] = 264;
p[4] = -2020;
```

Heap Memory		
Symbol	Address	Value
p[4]	1016	-2020
p[3]	1012	U
p[2]	1008	U
p[1]	1004	U
p[0]	1000	264

Stack Memory		
Symbol	Address	Value
p	100	A1000

Heap Memory

```
int * p;
p = malloc(sizeof(int) * 5);
// memory NOT initialized
p[0] = 264;
p[4] = -2020;
…
free (p); // no size given
// if not freed, memory leak
```

Heap Memory

Symbol	Address	Value
p[4]	1016	U
p[3]	1012	U
p[2]	1008	U
p[1]	1004	U
p[0]	1000	U

Stack Memory

Symbol	Address	Value
p	100	U

Heap Memory

```
int * p;
p = malloc(sizeof(int) * 5);
// memory NOT initialized
p[0] = 264;
p[4] = -2020;
…
free (p); // no size given
// if not freed, memory leak
```

Heap Memory		
Symbol	**Address**	**Value**
p[4]	1016	-2020
p[3]	1012	U
p[2]	1008	U
p[1]	1004	U
p[0]	1000	264

Stack Memory		
Symbol	**Address**	**Value**
p	100	A1000

The value is unchanged ⬆

Heap Memory

- `malloc` and `free` always go together, no exception
- Allocated memory is ***not*** initialized
- Valid indexes start at 0, last = size – 1, no exception
- Using `p[size]` is ***wrong***. program behavior **undefined**
- `free(p)` does ***not*** change p’s value. `p` is ***not*** `NULL`.
- Leaking memory is not allowed

Undefined Program Behavior

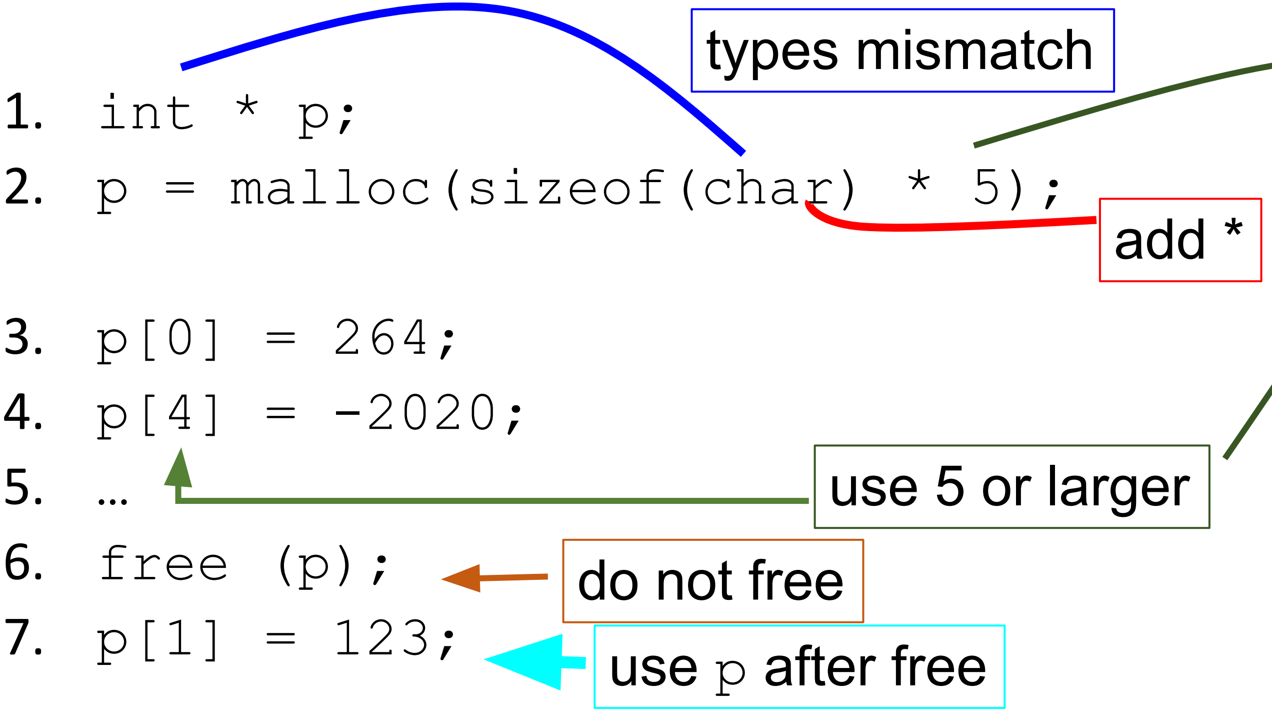
- Sometimes, the program may “work”.
- Sometimes, the program may not “work”.
- Usually, the program “works” when students test.
- The program does not “work” in grading.

- If an array has n elements, valid indexes are 0, 1, 2, … n – 1; n is an ***invalid*** index.

If an array has n elements,
n is an ***invalid*** index.

Common Mistakes

```
sizeof(int)  = 4
sizeof(char)  = 1
```



```
1.  int * p;
2.  p = malloc(sizeof(char) * 5);

3.  p[0] = 264;
4.  p[4] = -2020;
5.  …
6.  free (p);
7.  p[1] = 123;
```

types mismatch

add *

use 5 or larger

do not free

use `p` after free

Memory leak is wrong

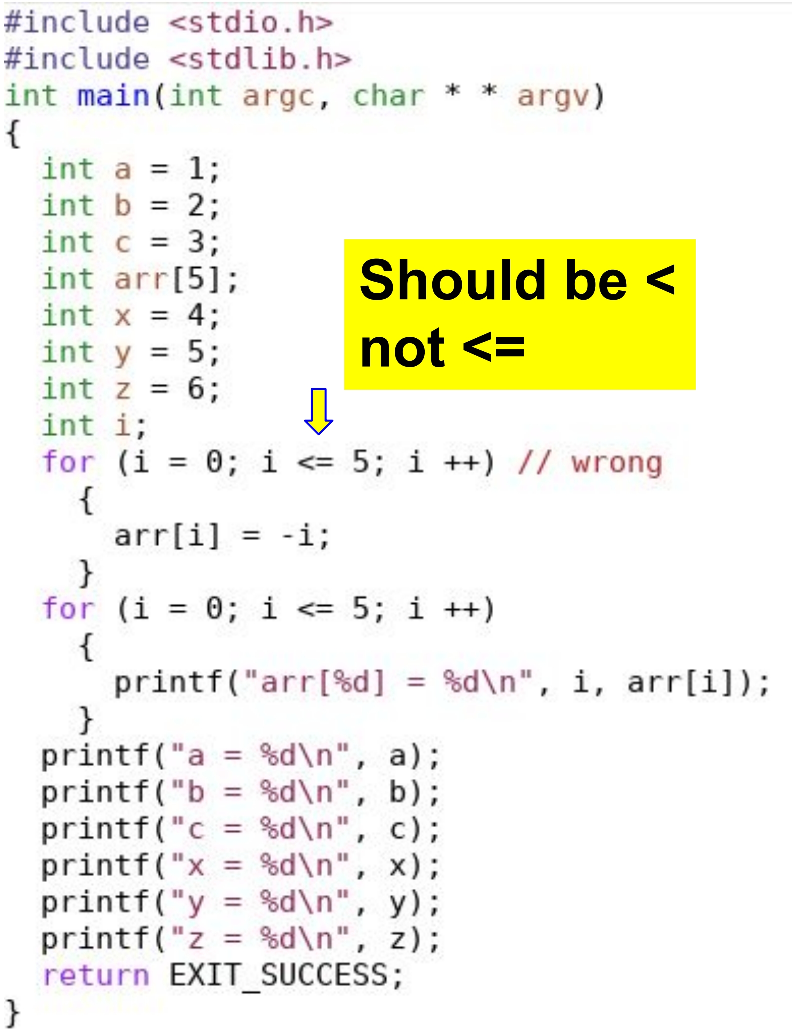
- Memory leak does not immediately stop programs but the programs will eventually run out of memory.
- Leaking memory is unacceptable, in the same way as an airplane leaks fuel.

Use heap memory carefully

- Heap memory is flexible. Freedom comes with responsibility.
- Usually, malloc and free are called in the same functions:

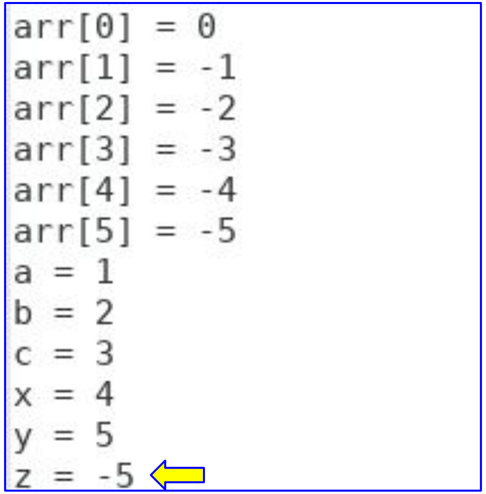
```
p = malloc (…);
… // processing data
free(p);
```

- Before calling `malloc`, think about where to call `free`.



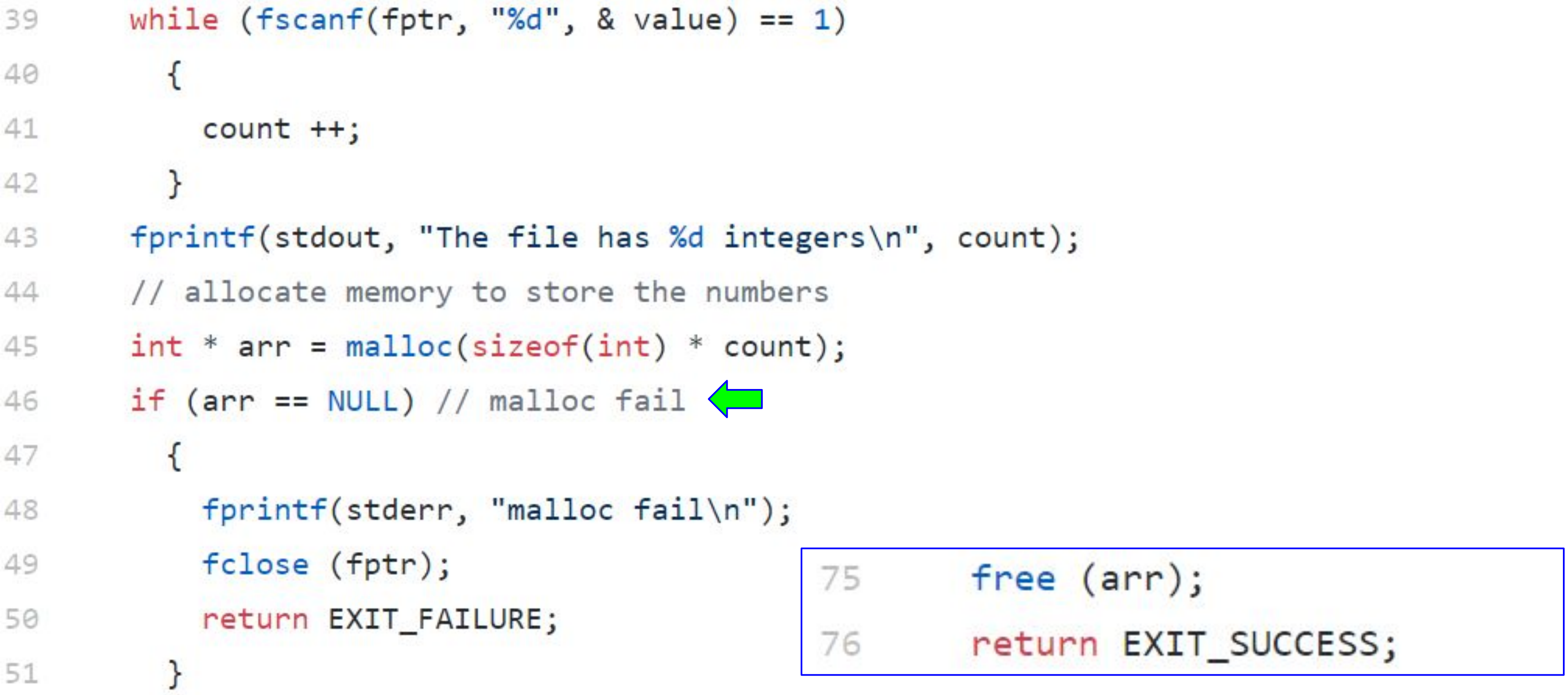
```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * * argv)
{
  int a = 1;
  int b = 2;
  int c = 3;
  int arr[5];
  int x = 4;
  int y = 5;
  int z = 6;
  int i;
  for (i = 0; i <= 5; i ++) // wrong
    {
      arr[i] = -i;
    }
  for (i = 0; i <= 5; i ++)
    {
      printf("arr[%d] = %d\n", i, arr[i]);
    }
  printf("a = %d\n", a);
  printf("b = %d\n", b);
  printf("c = %d\n", c);
  printf("x = %d\n", x);
  printf("y = %d\n", y);
  printf("z = %d\n", z);
  return EXIT_SUCCESS;
}
```

Should be < not <=



```
arr[0] = 0
arr[1] = -1
arr[2] = -2
arr[3] = -3
arr[4] = -4
arr[5] = -5
a = 1
b = 2
c = 3
x = 4
y = 5
z = -5
```

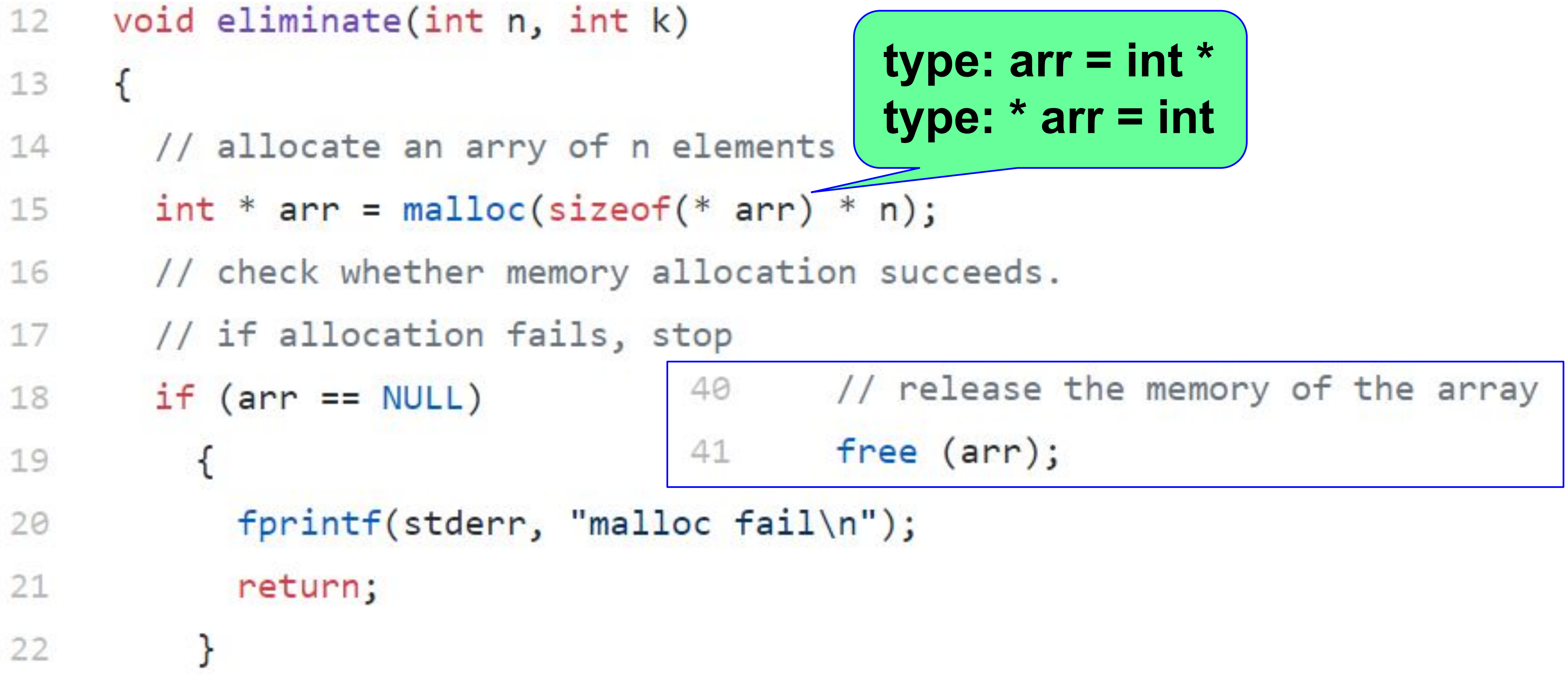
Heap Memory in HW 01



```
    while (fscanf(fptr, "%d", & value) == 1)
      {
        count ++;
      }
    fprintf(stdout, "The file has %d integers\n", count);
    // allocate memory to store the numbers
    int * arr = malloc(sizeof(int) * count);
    if (arr == NULL) // malloc fail
      {
        fprintf(stderr, "malloc fail\n");
        fclose (fptr);
        return EXIT_FAILURE;
      }
```

```
    free (arr);
    return EXIT_SUCCESS;
```

Heap Memory in HW 03



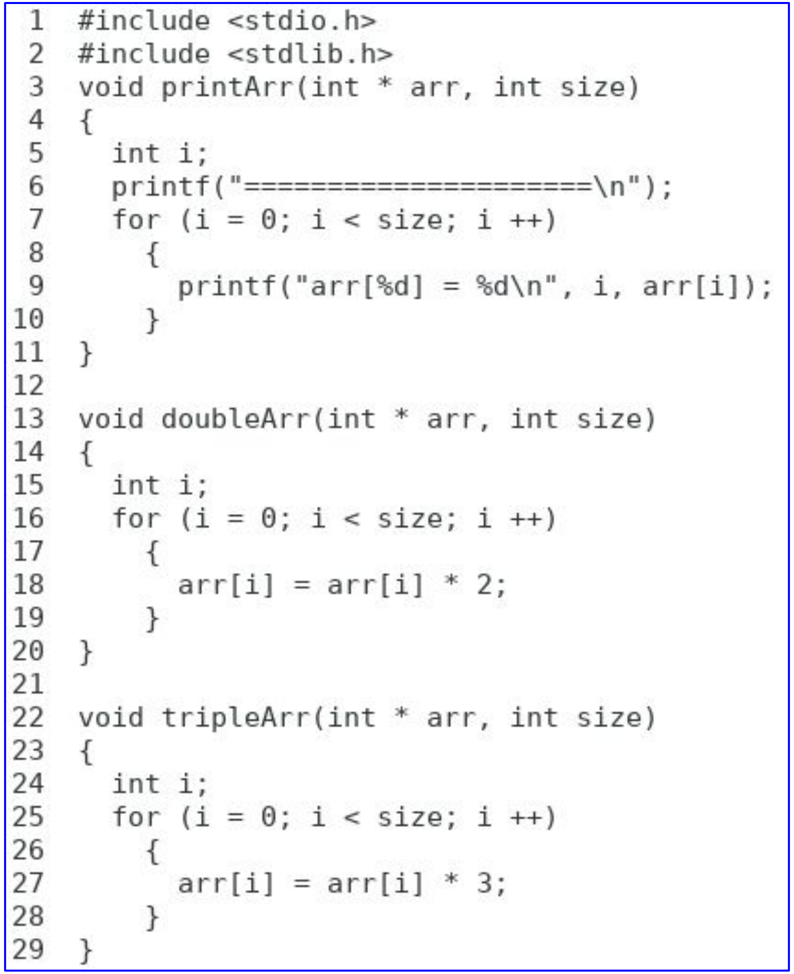
```c
void eliminate(int n, int k)
{
  // allocate an arry of n elements
  int * arr = malloc(sizeof(* arr) * n);
  // check whether memory allocation succeeds.
  // if allocation fails, stop
  if (arr == NULL)
    {
      fprintf(stderr, "malloc fail\n");
      return;
    }
```

type: arr = int *

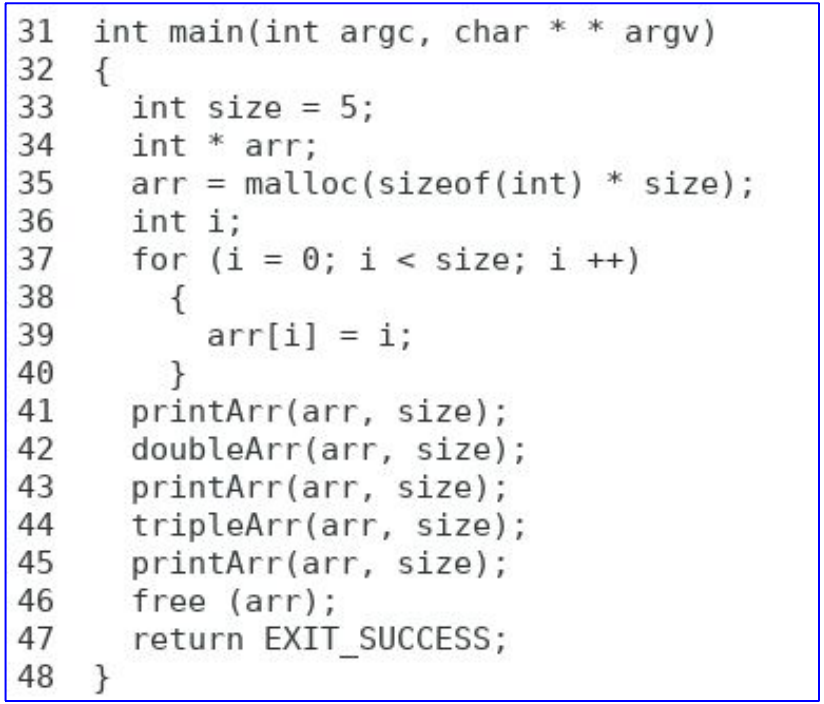
type: * arr = int

```c
  // release the memory of the array
  free (arr);
```

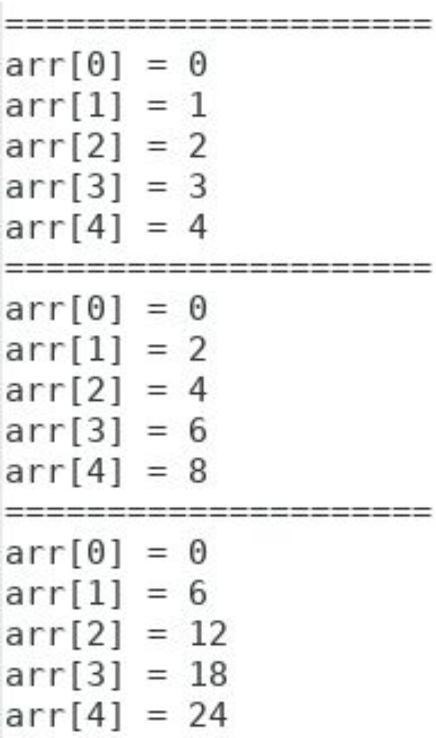
Pass Heap Memory in Functions



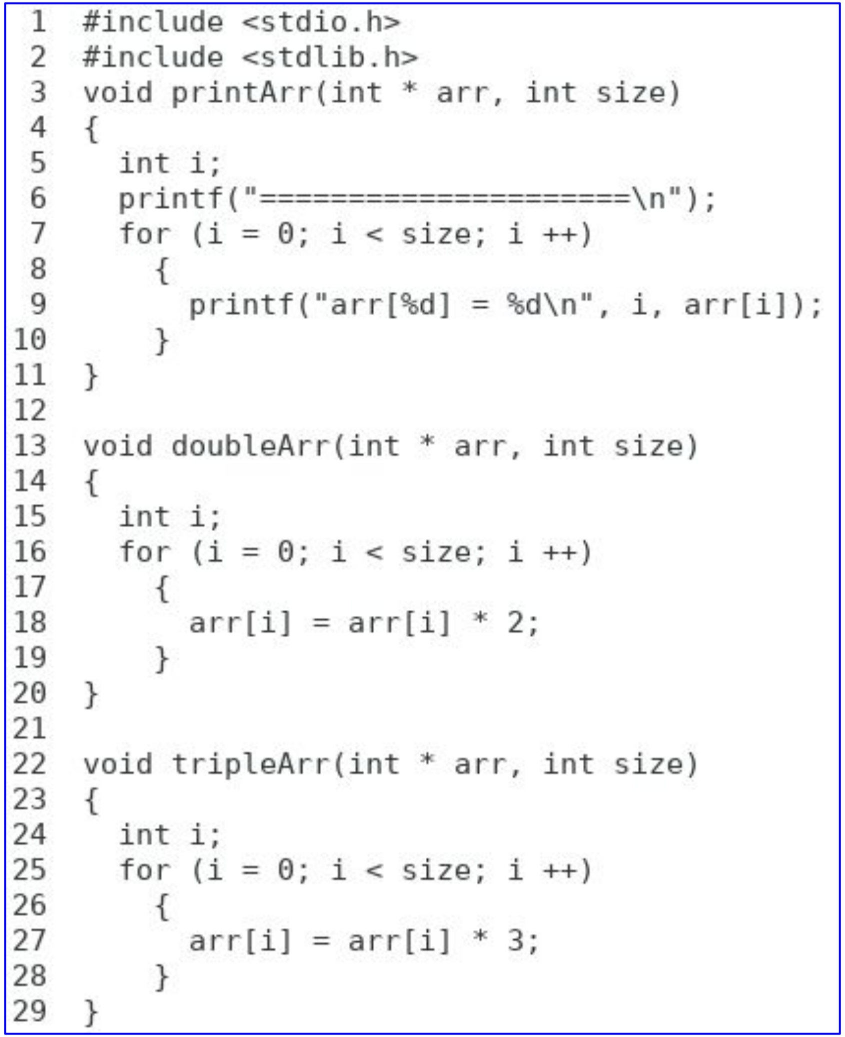
```c
#include <stdio.h>
#include <stdlib.h>
void printArr(int * arr, int size)
{
  int i;
  printf("=====================\n");
  for (i = 0; i < size; i ++)
    {
      printf("arr[%d] = %d\n", i, arr[i]);
    }
}

void doubleArr(int * arr, int size)
{
  int i;
  for (i = 0; i < size; i ++)
    {
      arr[i] = arr[i] * 2;
    }
}

void tripleArr(int * arr, int size)
{
  int i;
  for (i = 0; i < size; i ++)
    {
      arr[i] = arr[i] * 3;
    }
}
```



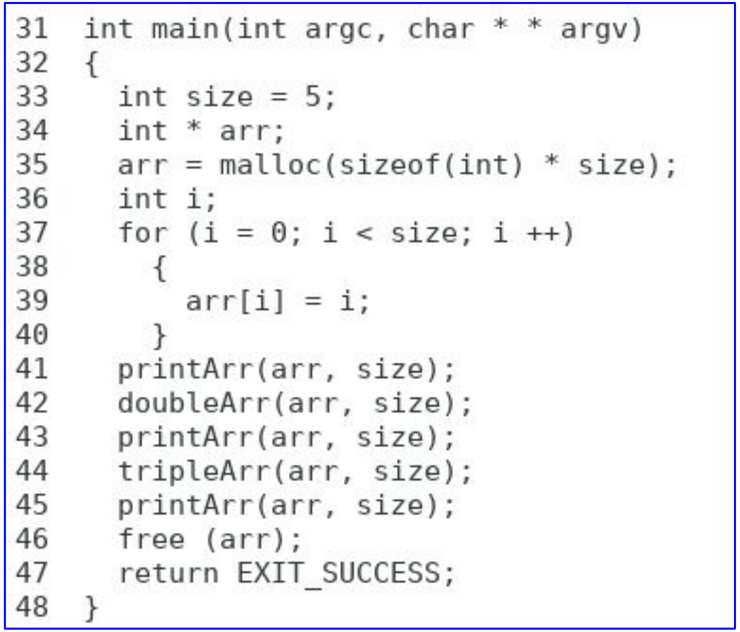
```c
int main(int argc, char * * argv)
{
  int size = 5;
  int * arr;
  arr = malloc(sizeof(int) * size);
  int i;
  for (i = 0; i < size; i ++)
    {
      arr[i] = i;
    }
  printArr(arr, size);
  doubleArr(arr, size);
  printArr(arr, size);
  tripleArr(arr, size);
  printArr(arr, size);
  free (arr);
  return EXIT_SUCCESS;
}
```



```
=======================
arr[0] = 0
arr[1] = 1
arr[2] = 2
arr[3] = 3
arr[4] = 4
=======================
arr[0] = 0
arr[1] = 2
arr[2] = 4
arr[3] = 6
arr[4] = 8
=======================
arr[0] = 0
arr[1] = 6
arr[2] = 12
arr[3] = 18
arr[4] = 24
```

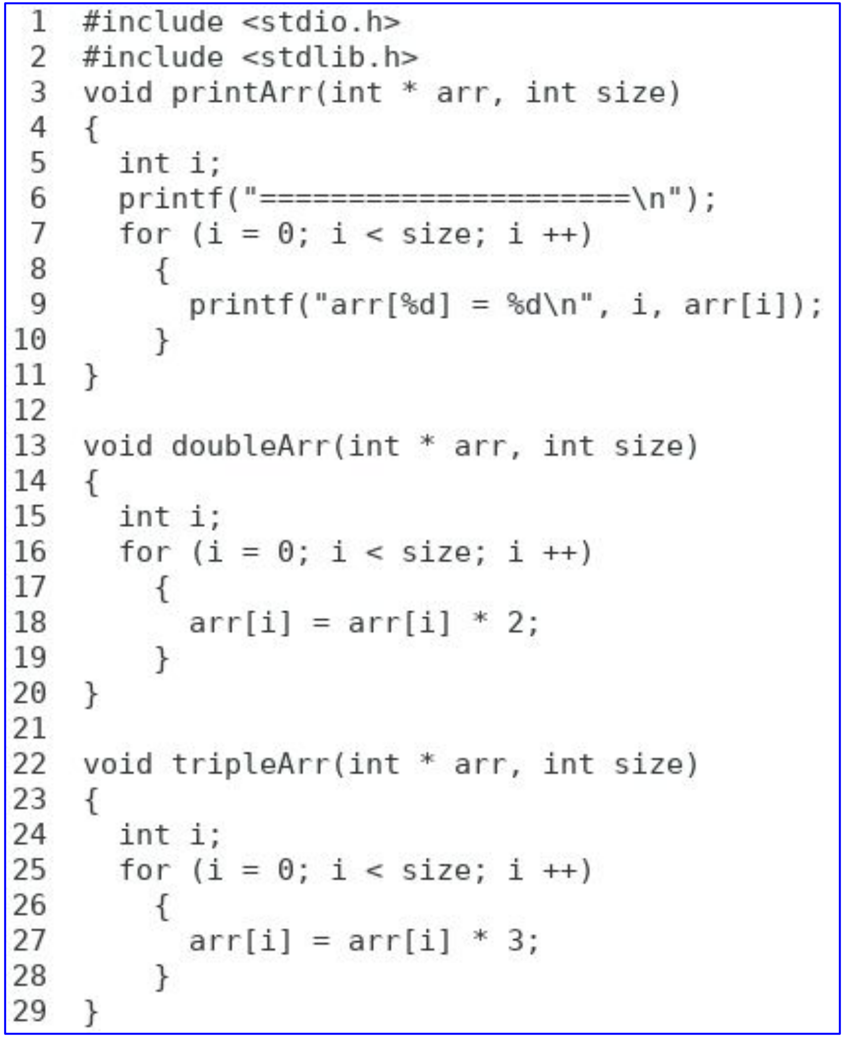


```c
#include <stdio.h>
#include <stdlib.h>
void printArr(int * arr, int size)
{
  int i;
  printf("=====================\n");
  for (i = 0; i < size; i ++)
    {
      printf("arr[%d] = %d\n", i, arr[i]);
    }
}

void doubleArr(int * arr, int size)
{
  int i;
  for (i = 0; i < size; i ++)
    {
      arr[i] = arr[i] * 2;
    }
}

void tripleArr(int * arr, int size)
{
  int i;
  for (i = 0; i < size; i ++)
    {
      arr[i] = arr[i] * 3;
    }
}
```

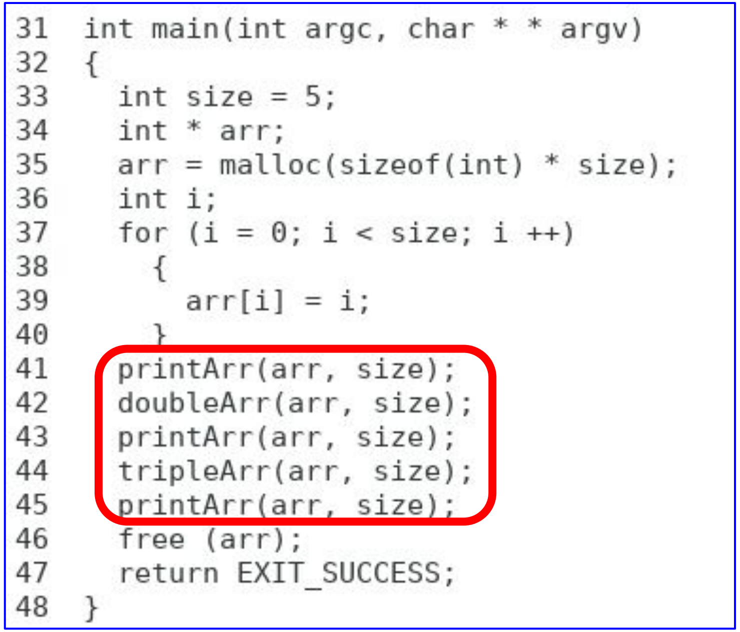


```c
int main(int argc, char * * argv)
{
  int size = 5;
  int * arr;
  arr = malloc(sizeof(int) * size);
  int i;
  for (i = 0; i < size; i ++)
    {
      arr[i] = i;
    }
  printArr(arr, size);
  doubleArr(arr, size);
  printArr(arr, size);
  tripleArr(arr, size);
  printArr(arr, size);
  free (arr);
  return EXIT_SUCCESS;
}
```

malloc and free in the same function

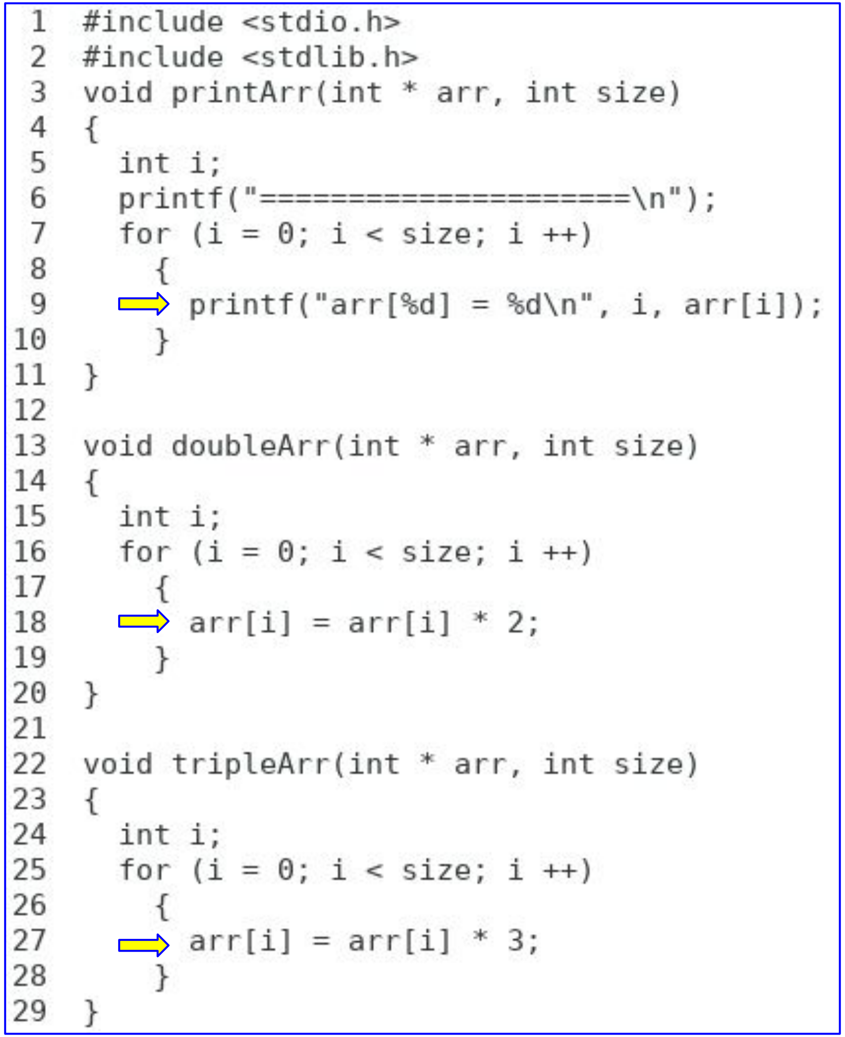


```c
#include <stdio.h>
#include <stdlib.h>
void printArr(int * arr, int size)
{
  int i;
  printf("=====================\n");
  for (i = 0; i < size; i ++)
    {
      printf("arr[%d] = %d\n", i, arr[i]);
    }
}

void doubleArr(int * arr, int size)
{
  int i;
  for (i = 0; i < size; i ++)
    {
      arr[i] = arr[i] * 2;
    }
}

void tripleArr(int * arr, int size)
{
  int i;
  for (i = 0; i < size; i ++)
    {
      arr[i] = arr[i] * 3;
    }
}
```

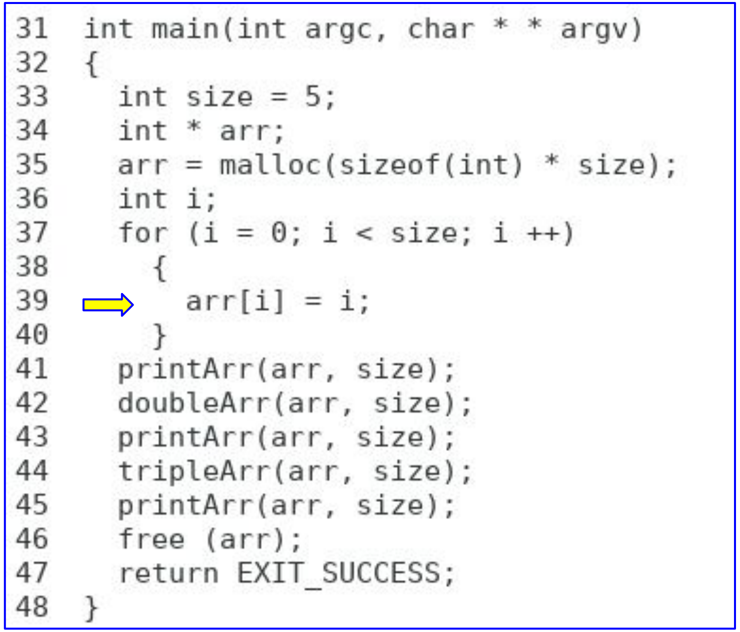


```c
int main(int argc, char * * argv)
{
  int size = 5;
  int * arr;
  arr = malloc(sizeof(int) * size);
  int i;
  for (i = 0; i < size; i ++)
    {
      arr[i] = i;
    }
  printArr(arr, size);
  doubleArr(arr, size);
  printArr(arr, size);
  tripleArr(arr, size);
  printArr(arr, size);
  free (arr);
  return EXIT_SUCCESS;
}
```

calling functions using heap memory



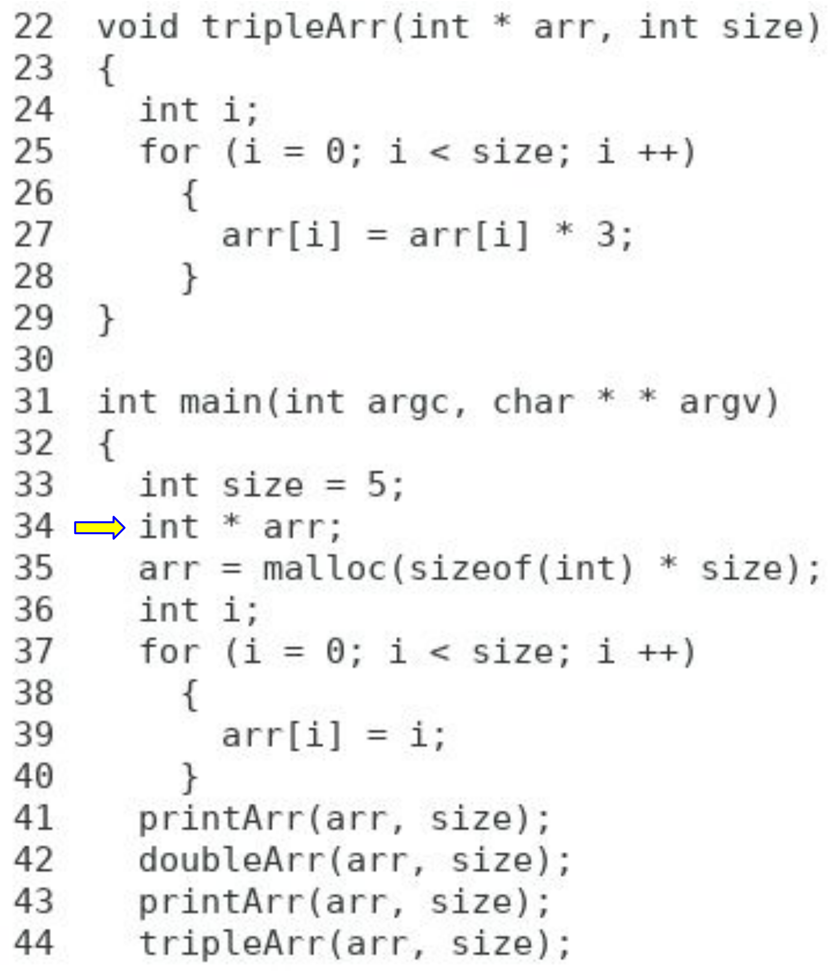
```
#include <stdio.h>
#include <stdlib.h>
void printArr(int * arr, int size)
{
  int i;
  printf("=====================\n");
  for (i = 0; i < size; i ++)
    {
      printf("arr[%d] = %d\n", i, arr[i]);
    }
}

void doubleArr(int * arr, int size)
{
  int i;
  for (i = 0; i < size; i ++)
    {
      arr[i] = arr[i] * 2;
    }
}

void tripleArr(int * arr, int size)
{
  int i;
  for (i = 0; i < size; i ++)
    {
      arr[i] = arr[i] * 3;
    }
}
```



```
int main(int argc, char * * argv)
{
  int size = 5;
  int * arr;
  arr = malloc(sizeof(int) * size);
  int i;
  for (i = 0; i < size; i ++)
    {
      arr[i] = i;
    }
  printArr(arr, size);
  doubleArr(arr, size);
  printArr(arr, size);
  tripleArr(arr, size);
  printArr(arr, size);
  free (arr);
  return EXIT_SUCCESS;
}
```

heap memory treated as an array

Stack and Heap Memory



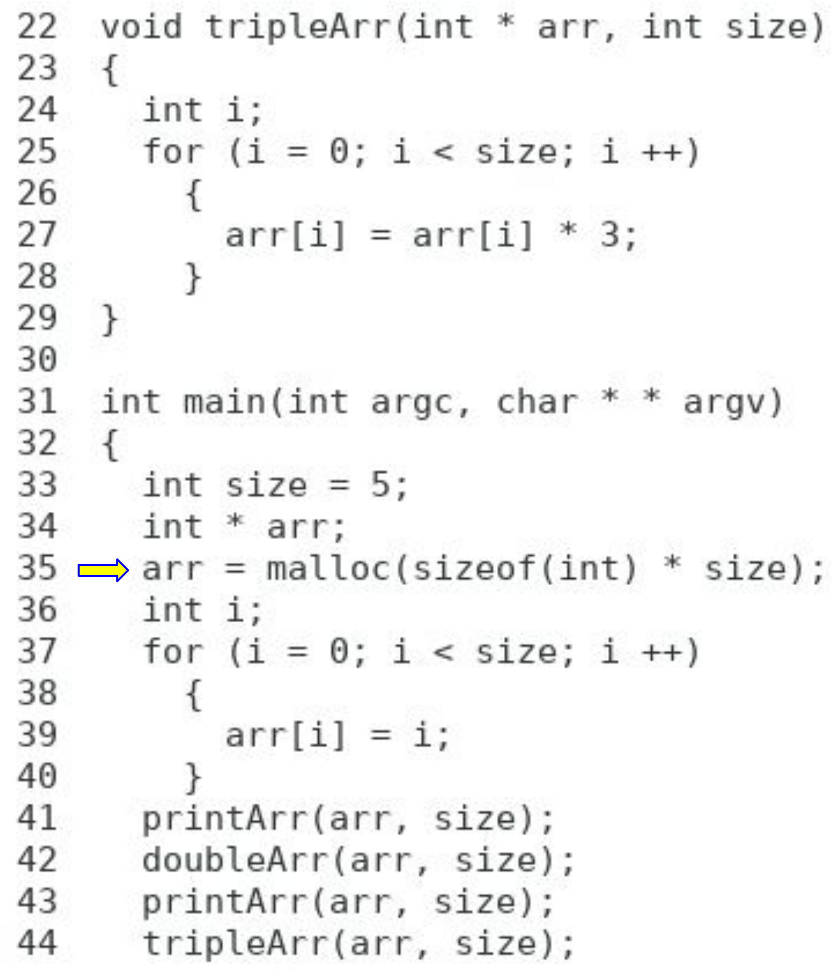
```c
void tripleArr(int * arr, int size)
{
  int i;
  for (i = 0; i < size; i ++)
    {
      arr[i] = arr[i] * 3;
    }
}

int main(int argc, char * * argv)
{
  int size = 5;
  int * arr;
  arr = malloc(sizeof(int) * size);
  int i;
  for (i = 0; i < size; i ++)
    {
      arr[i] = i;
    }
  printArr(arr, size);
  doubleArr(arr, size);
  printArr(arr, size);
  tripleArr(arr, size);
```

Heap Memory

Symbol	Address	Value

Stack Memory

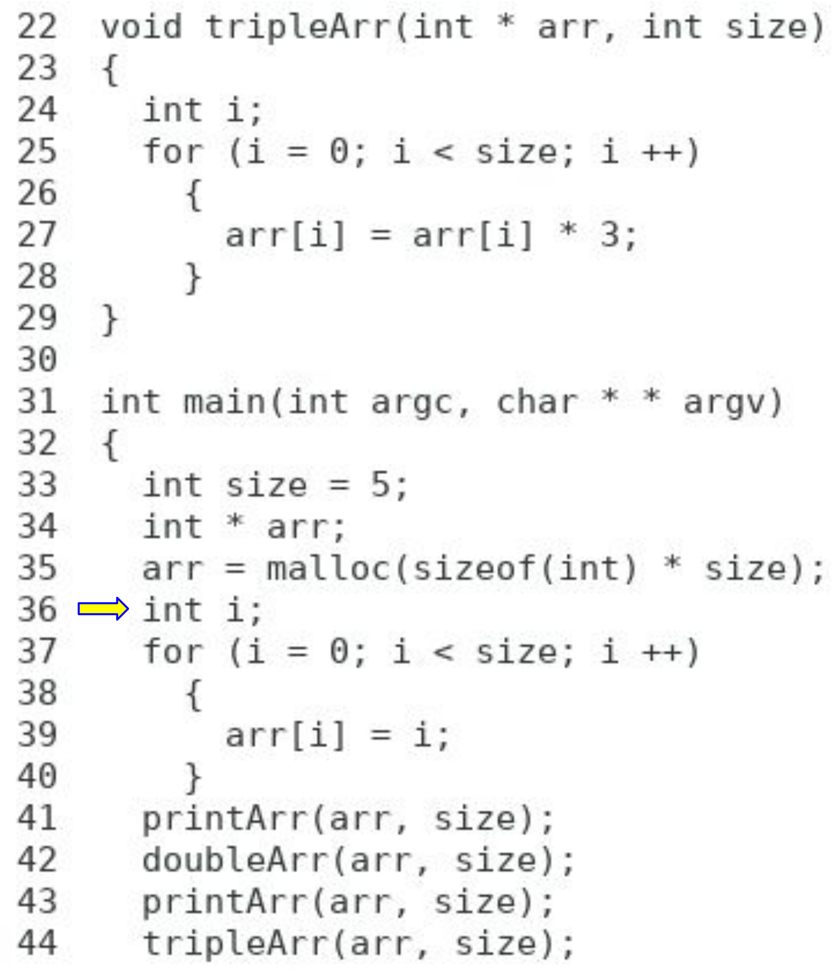
Symbol	Address	Value
arr	104	U
size	100	5



```
void tripleArr(int * arr, int size)
{
  int i;
  for (i = 0; i < size; i ++)
    {
      arr[i] = arr[i] * 3;
    }
}

int main(int argc, char * * argv)
{
  int size = 5;
  int * arr;
  arr = malloc(sizeof(int) * size);
  int i;
  for (i = 0; i < size; i ++)
    {
      arr[i] = i;
    }
  printArr(arr, size);
  doubleArr(arr, size);
  printArr(arr, size);
  tripleArr(arr, size);
```

Heap Memory		
Symbol	**Address**	**Value**
arr[4]	2016	U
arr[3]	2012	U
arr[2]	2008	U
arr[1]	2004	U
arr[0]	2000	U

Stack Memory		
Symbol	**Address**	**Value**
arr	104	A2000
size	100	5



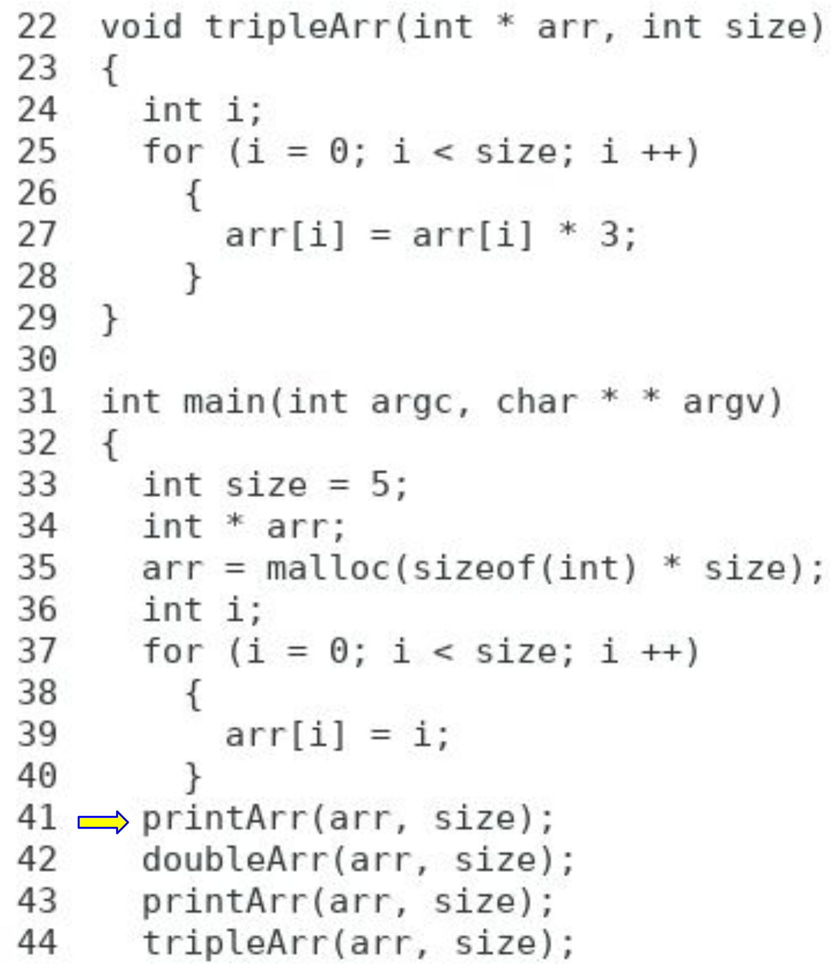
```
void tripleArr(int * arr, int size)
{
  int i;
  for (i = 0; i < size; i ++)
    {
      arr[i] = arr[i] * 3;
    }
}

int main(int argc, char * * argv)
{
  int size = 5;
  int * arr;
  arr = malloc(sizeof(int) * size);
⇨ int i;
  for (i = 0; i < size; i ++)
    {
      arr[i] = i;
    }
  printArr(arr, size);
  doubleArr(arr, size);
  printArr(arr, size);
  tripleArr(arr, size);
```

Heap Memory

Symbol	Address	Value
arr[4]	2016	U
arr[3]	2012	U
arr[2]	2008	U
arr[1]	2004	U
arr[0]	2000	U

Stack Memory

Symbol	Address	Value
i	112	U
arr	104	A2000
size	100	5



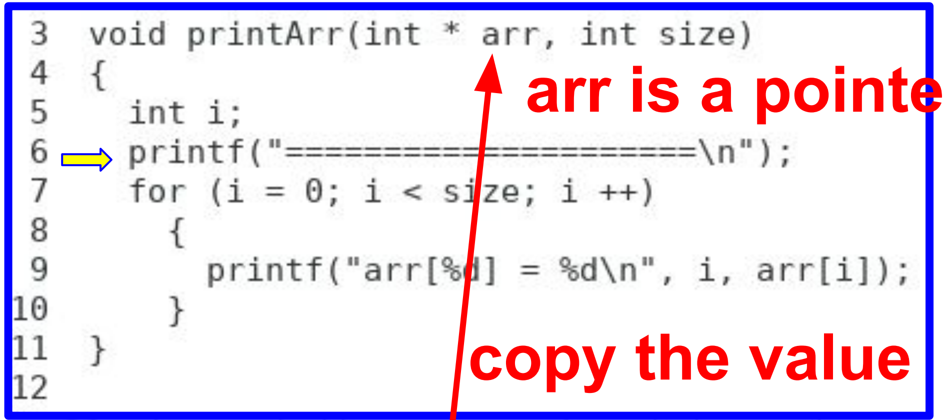
```
void tripleArr(int * arr, int size)
{
  int i;
  for (i = 0; i < size; i ++)
    {
      arr[i] = arr[i] * 3;
    }
}

int main(int argc, char * * argv)
{
  int size = 5;
  int * arr;
  arr = malloc(sizeof(int) * size);
  int i;
  for (i = 0; i < size; i ++)
    {
      arr[i] = i;
    }
⇨ printArr(arr, size);
  doubleArr(arr, size);
  printArr(arr, size);
  tripleArr(arr, size);
```

Heap Memory

Symbol	Address	Value
arr[4]	2016	4
arr[3]	2012	3
arr[2]	2008	2
arr[1]	2004	1
arr[0]	2000	0

Stack Memory

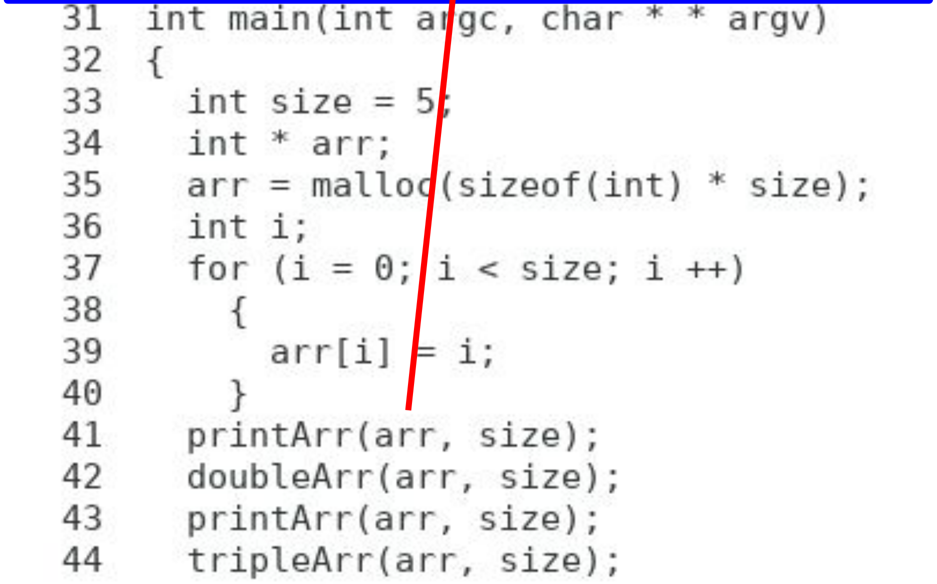
Symbol	Address	Value
i	112	U
arr	104	A2000
size	100	5



```
3  void printArr(int * arr, int size)
4  {
5    int i;
6    printf("====================\n");
7    for (i = 0; i < size; i ++)
8      {
9        printf("arr[%d] = %d\n", i, arr[i]);
10     }
11 }
12
```

arr is a pointer

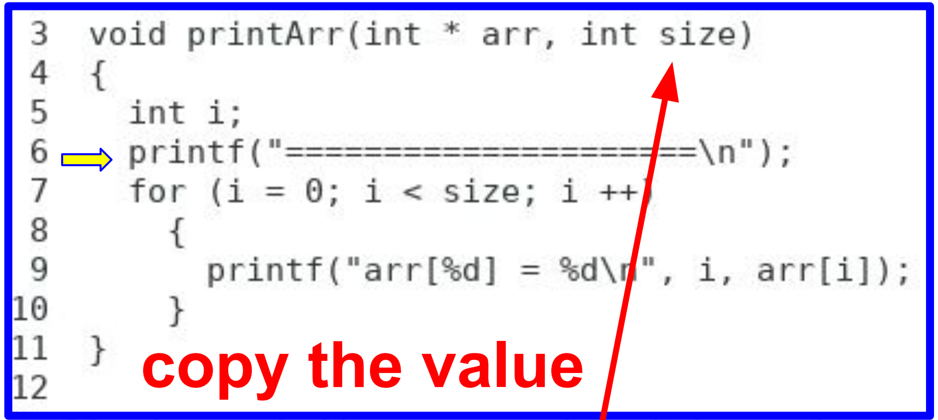
copy the value



```
31  int main(int argc, char * * argv)
32  {
33    int size = 5;
34    int * arr;
35    arr = malloc(sizeof(int) * size);
36    int i;
37    for (i = 0; i < size; i ++)
38      {
39        arr[i] = i;
40      }
41    printArr(arr, size);
42    doubleArr(arr, size);
43    printArr(arr, size);
44    tripleArr(arr, size);
```

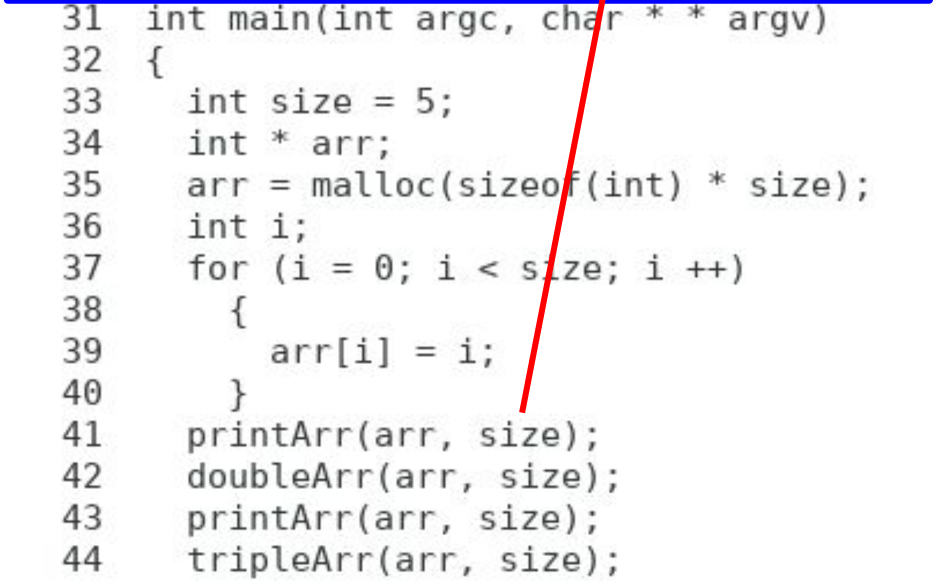
Heap Memory		
Symbol	**Address**	**Value**
arr[4]	2016	4
arr[3]	2012	3
arr[2]	2008	2
arr[1]	2004	1
arr[0]	2000	0

Stack Memory			
Frame	**Symbol**	**Address**	**Value**
printArr	i	212	U
	size	208	5
	arr	200	A2000
	Return Location line 42		
main	i	112	U
	arr	104	A2000
	size	100	5



```
3  void printArr(int * arr, int size)
4  {
5    int i;
6    printf("====================\n");
7    for (i = 0; i < size; i ++)
8      {
9        printf("arr[%d] = %d\n", i, arr[i]);
10     }
11 }
12
```

copy the value



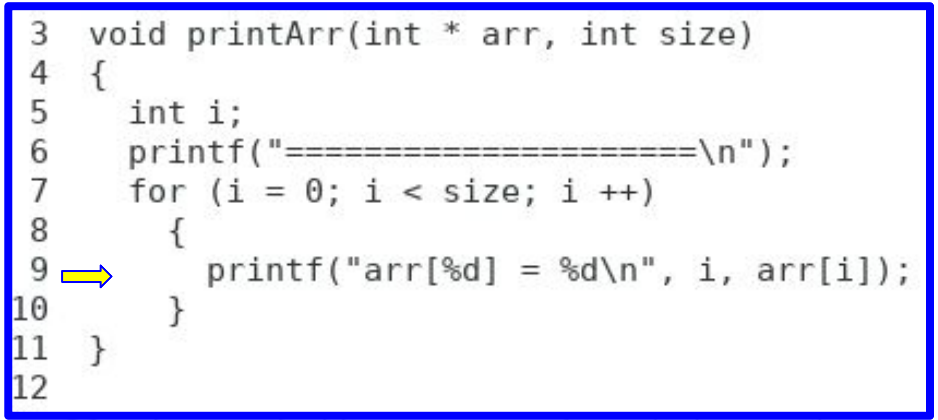
```
31  int main(int argc, char * * argv)
32  {
33    int size = 5;
34    int * arr;
35    arr = malloc(sizeof(int) * size);
36    int i;
37    for (i = 0; i < size; i ++)
38      {
39        arr[i] = i;
40      }
41    printArr(arr, size);
42    doubleArr(arr, size);
43    printArr(arr, size);
44    tripleArr(arr, size);
```

Heap Memory

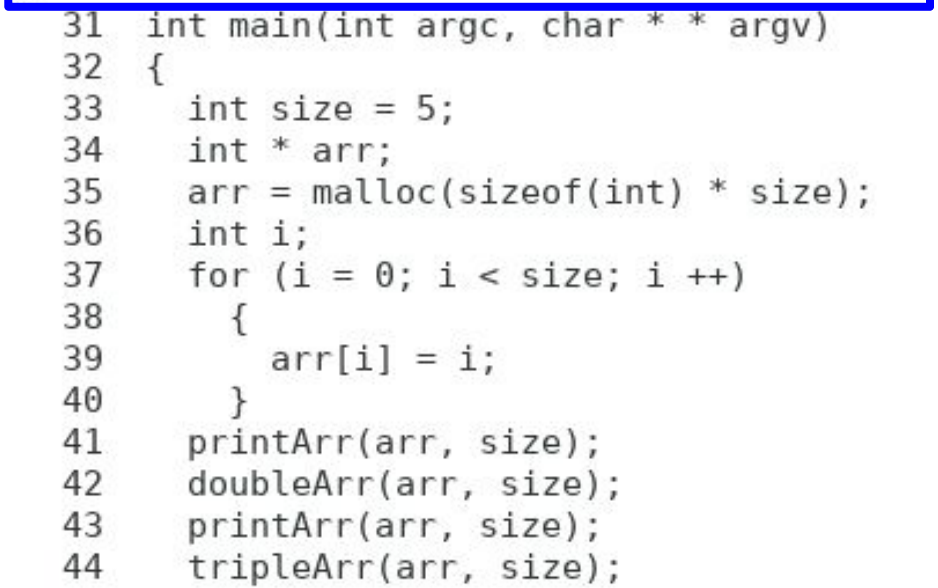
Symbol	Address	Value
arr[4]	2016	4
arr[3]	2012	3
arr[2]	2008	2
arr[1]	2004	1
arr[0]	2000	0

Stack Memory

Frame	Symbol	Address	Value
printArr	i	212	U
	size	208	5
	arr	200	A2000
	Return Location line 42		
main	i	112	U
	arr	104	A2000
	size	100	5



```
3  void printArr(int * arr, int size)
4  {
5    int i;
6    printf("=====================\n");
7    for (i = 0; i < size; i ++)
8      {
9        printf("arr[%d] = %d\n", i, arr[i]);
10     }
11 }
12
```



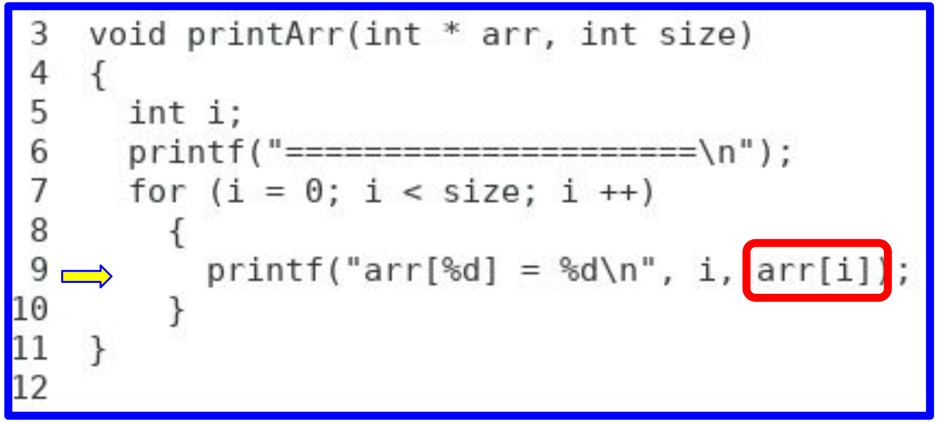
```
31  int main(int argc, char * * argv)
32  {
33    int size = 5;
34    int * arr;
35    arr = malloc(sizeof(int) * size);
36    int i;
37    for (i = 0; i < size; i ++)
38      {
39        arr[i] = i;
40      }
41    printArr(arr, size);
42    doubleArr(arr, size);
43    printArr(arr, size);
44    tripleArr(arr, size);
```

Heap Memory

Symbol	Address	Value
arr[4]	2016	4
arr[3]	2012	3
arr[2]	2008	2
arr[1]	2004	1
arr[0]	2000	0

Stack Memory

Frame	Symbol	Address	Value
printArr	i	212	0
	size	208	5
	arr	200	A2000
	Return Location line 42		
main	i	112	U
	arr	104	A2000
	size	100	5



```
void printArr(int * arr, int size)
{
  int i;
  printf("====================\n");
  for (i = 0; i < size; i ++)
    {
      printf("arr[%d] = %d\n", i, arr[i]);
    }
}
```

2. add i · size of one element
3. take the value as an address
4. go to that address
   - LHS: modify the value at the address
   - RHS: read the value at the address

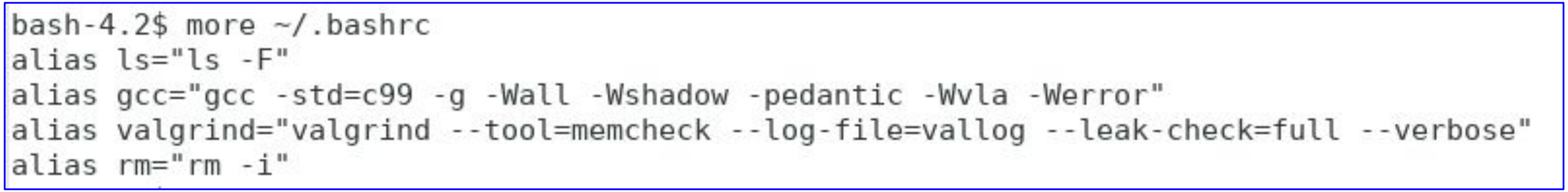
Heap Memory		
Symbol	**Address**	**Value**
arr[4]	2016	4
arr[3]	2012	3
arr[2]	2008	2
arr[1]	2004	1
arr[0]	2000	0

Stack Memory			
Frame	**Symbol**	**Address**	**Value**
printArr	i	212	0
	size	208	5
	arr	200	A2000
	Return Location line 42		
main	i	112	U
	arr	104	A2000
	size	100	5

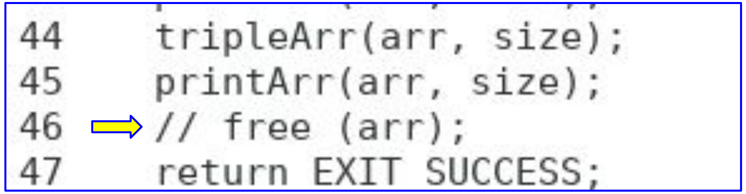
& arr[k] = & arr[0] + k · size of one element

The address of the element with index k is the address of the first element (index is 0) + k multiplied with the size of one element

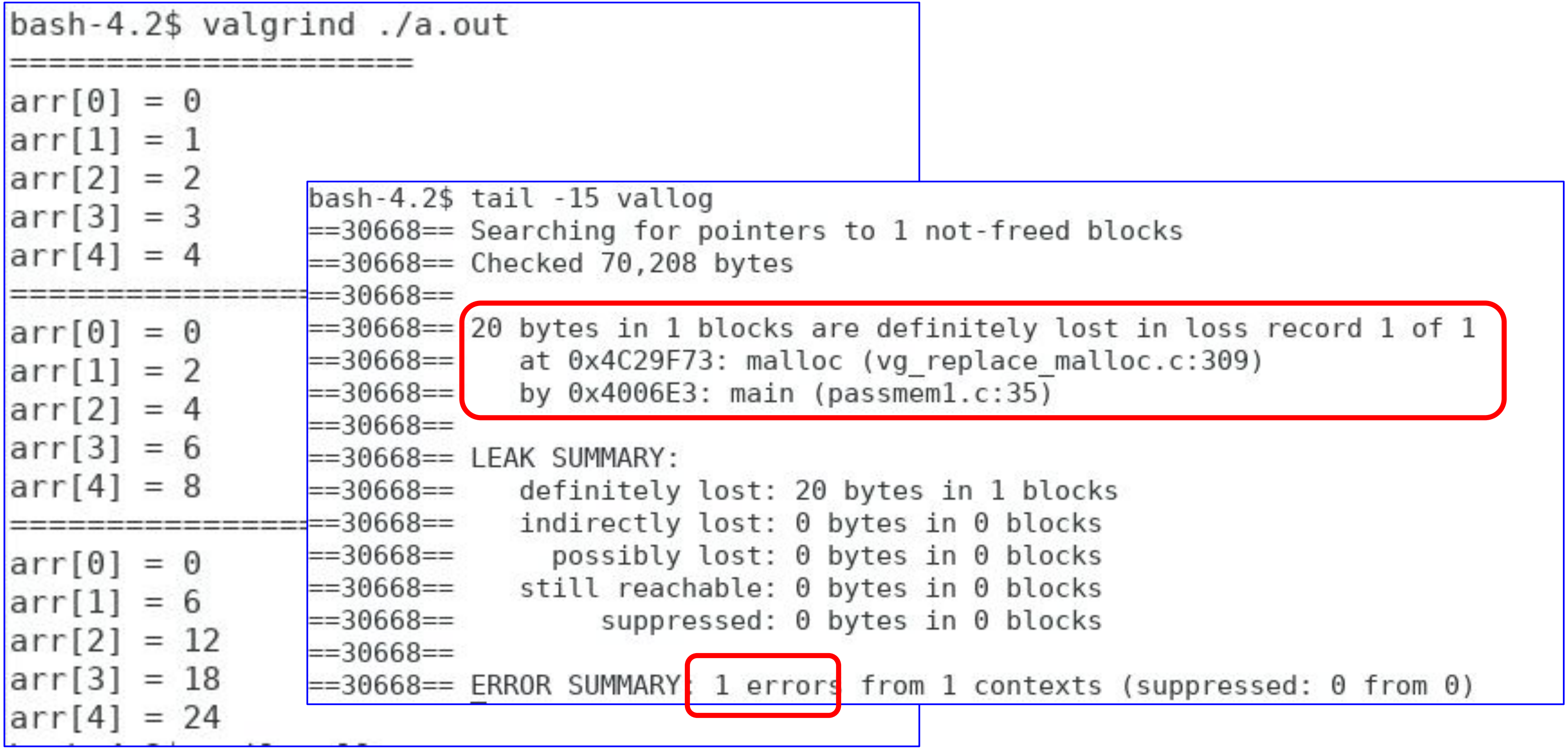
use `valgrind` to detect memory leak



```
bash-4.2$ more ~/.bashrc
alias ls="ls -F"
alias gcc="gcc -std=c99 -g -Wall -Wshadow -pedantic -Wvla -Werror"
alias valgrind="valgrind --tool=memcheck --log-file=vallog --leak-check=full --verbose"
alias rm="rm -i"
```



```
    tripleArr(arr, size);
    printArr(arr, size);
    // free (arr);
    return EXIT_SUCCESS;
```



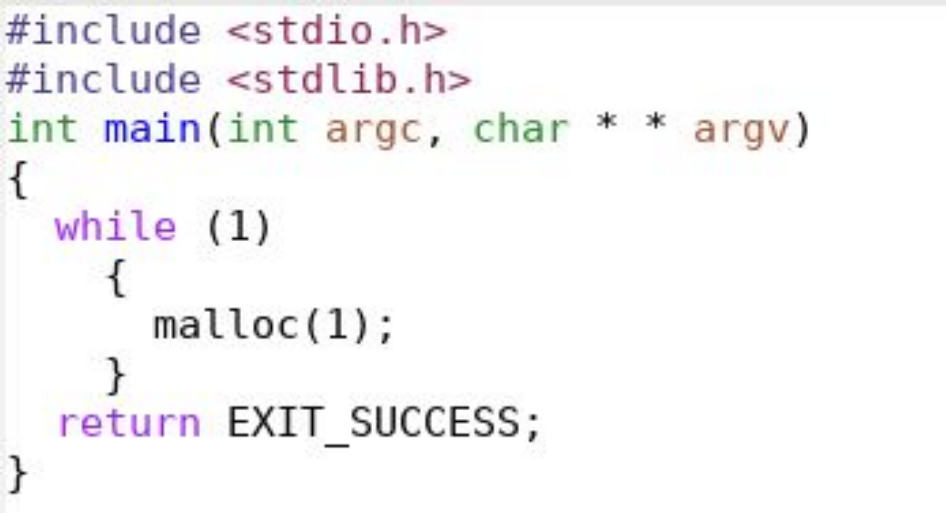
```
bash-4.2$ valgrind ./a.out
========================
arr[0] = 0
arr[1] = 1
arr[2] = 2
arr[3] = 3
arr[4] = 4
=================
arr[0] = 0
arr[1] = 2
arr[2] = 4
arr[3] = 6
arr[4] = 8
=================
arr[0] = 0
arr[1] = 6
arr[2] = 12
arr[3] = 18
arr[4] = 24
```

```
bash-4.2$ tail -15 vallog
==30668== Searching for pointers to 1 not-freed blocks
==30668== Checked 70,208 bytes
==30668==
==30668== 20 bytes in 1 blocks are definitely lost in loss record 1 of 1
==30668==    at 0x4C29F73: malloc (vg_replace_malloc.c:309)
==30668==    by 0x4006E3: main (passmem1.c:35)
==30668==
==30668== LEAK SUMMARY:
==30668==    definitely lost: 20 bytes in 1 blocks
==30668==    indirectly lost: 0 bytes in 0 blocks
==30668==      possibly lost: 0 bytes in 0 blocks
==30668==    still reachable: 0 bytes in 0 blocks
==30668==         suppressed: 0 bytes in 0 blocks
==30668==
==30668== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Memory leak is not acceptable.

Don’t do this

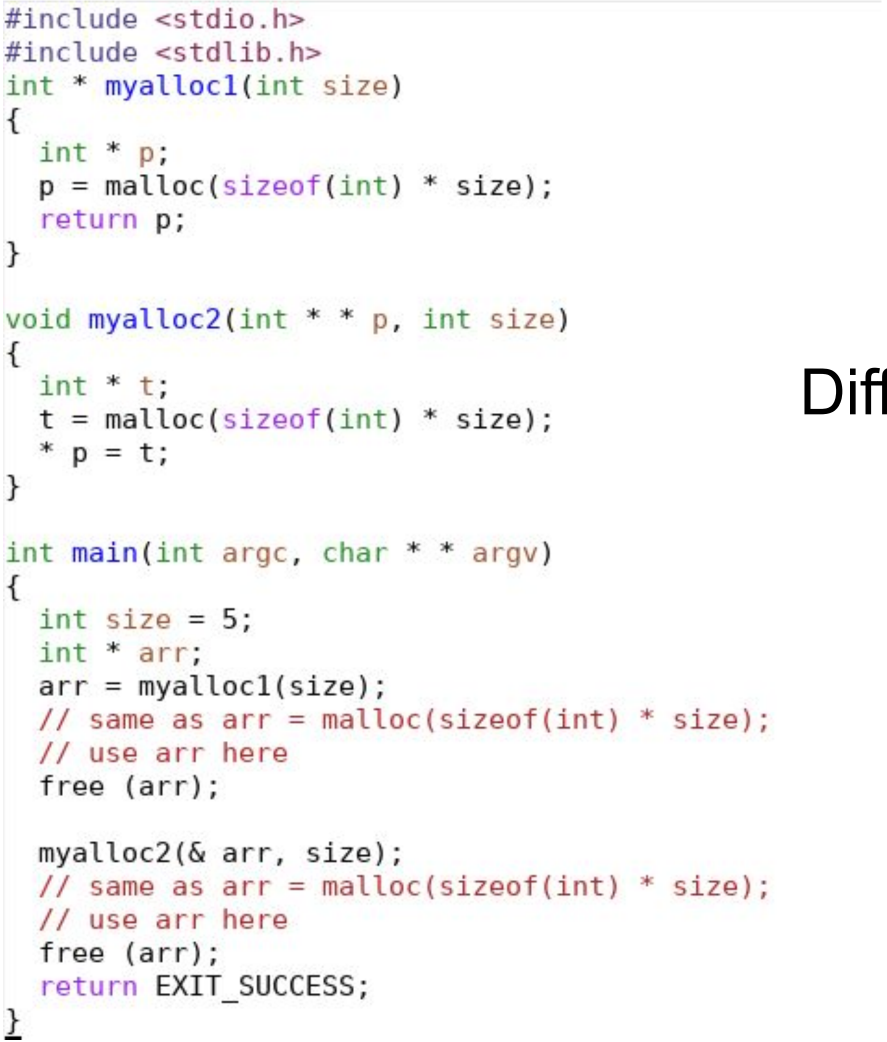
Do not do this on Purdue computers.



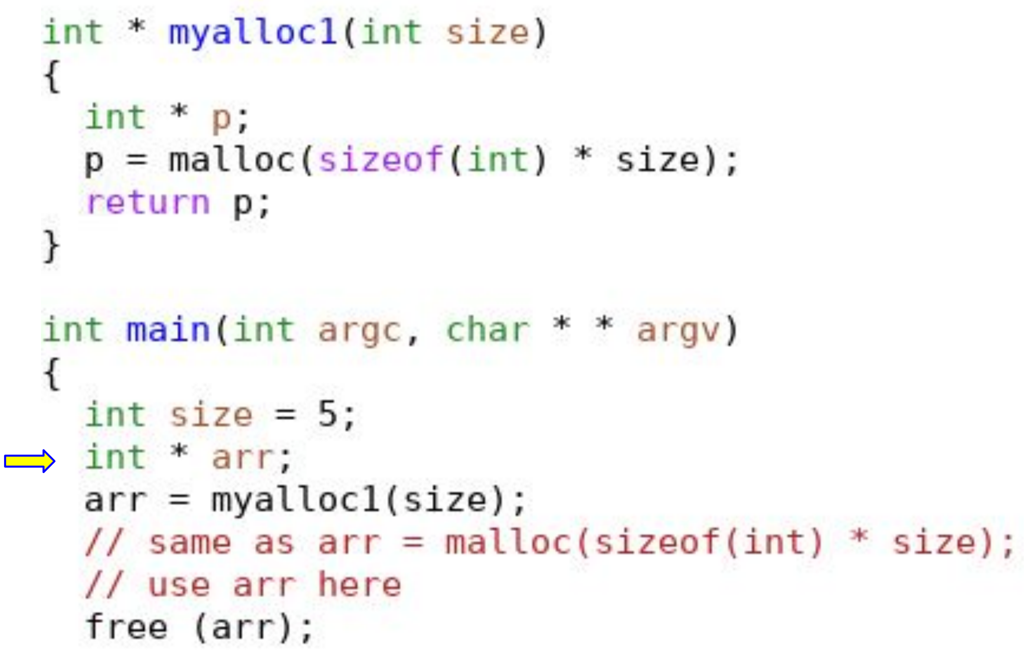
```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * * argv)
{
  while (1)
    {
      malloc(1);
    }
  return EXIT_SUCCESS;
}
```

If you do this on your own computer, your computer will become unusable.

Different ways to allocate memory

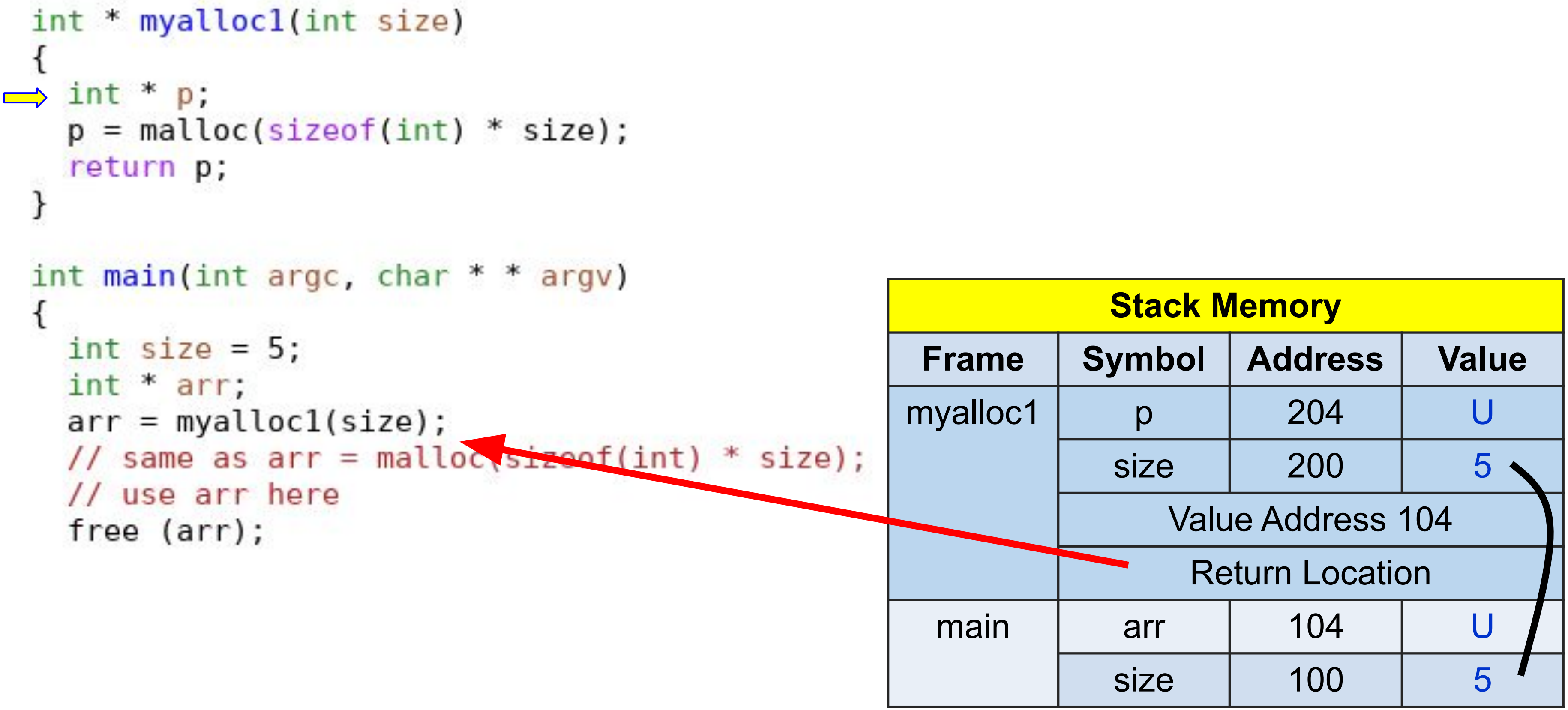


```
#include <stdio.h>
#include <stdlib.h>
int * myalloc1(int size)
{
  int * p;
  p = malloc(sizeof(int) * size);
  return p;
}

void myalloc2(int * * p, int size)
{
  int * t;
  t = malloc(sizeof(int) * size);
  * p = t;
}

int main(int argc, char * * argv)
{
  int size = 5;
  int * arr;
  arr = myalloc1(size);
  // same as arr = malloc(sizeof(int) * size);
  // use arr here
  free (arr);

  myalloc2(& arr, size);
  // same as arr = malloc(sizeof(int) * size);
  // use arr here
  free (arr);
  return EXIT_SUCCESS;
}
```



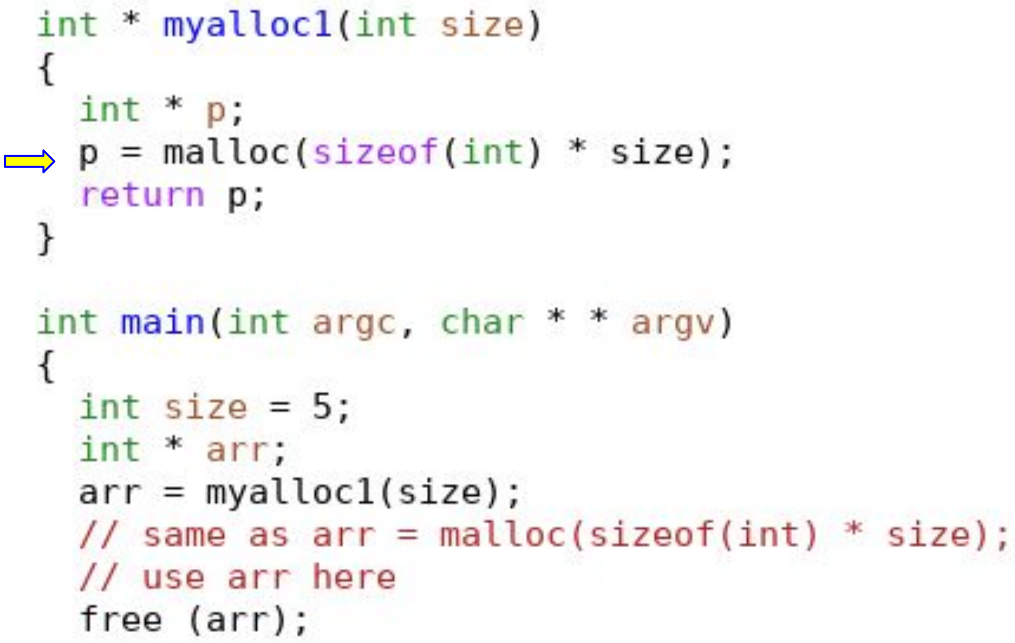
```c
int * myalloc1(int size)
{
  int * p;
  p = malloc(sizeof(int) * size);
  return p;
}

int main(int argc, char * * argv)
{
  int size = 5;
  int * arr;
  arr = myalloc1(size);
  // same as arr = malloc(sizeof(int) * size);
  // use arr here
  free (arr);
```

Stack Memory			
Frame	**Symbol**	**Address**	**Value**
main	arr	104	U
	size	100	5

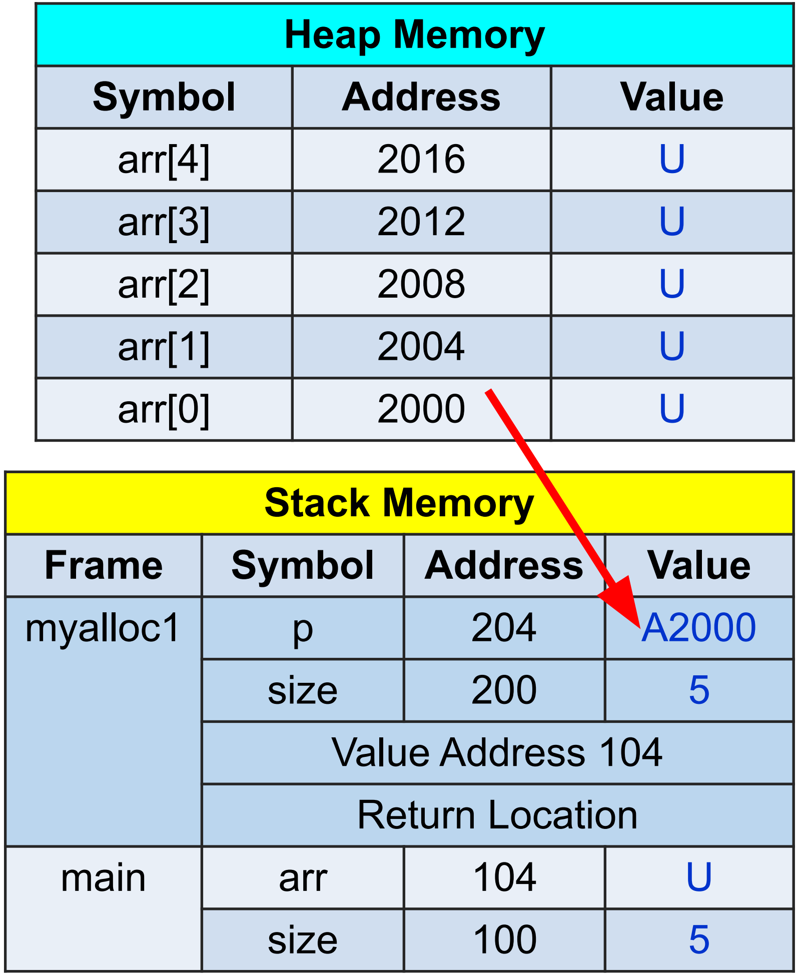


```
int * myalloc1(int size)
{
  int * p;
  p = malloc(sizeof(int) * size);
  return p;
}

int main(int argc, char * * argv)
{
  int size = 5;
  int * arr;
  arr = myalloc1(size);
  // same as arr = malloc(sizeof(int) * size);
  // use arr here
  free (arr);
```

Stack Memory			
Frame	**Symbol**	**Address**	**Value**
myalloc1	p	204	U
	size	200	5
	Value Address 104		
	Return Location		
main	arr	104	U
	size	100	5



```
int * myalloc1(int size)
{
  int * p;
  p = malloc(sizeof(int) * size);
  return p;
}

int main(int argc, char * * argv)
{
  int size = 5;
  int * arr;
  arr = myalloc1(size);
  // same as arr = malloc(sizeof(int) * size);
  // use arr here
  free (arr);
```

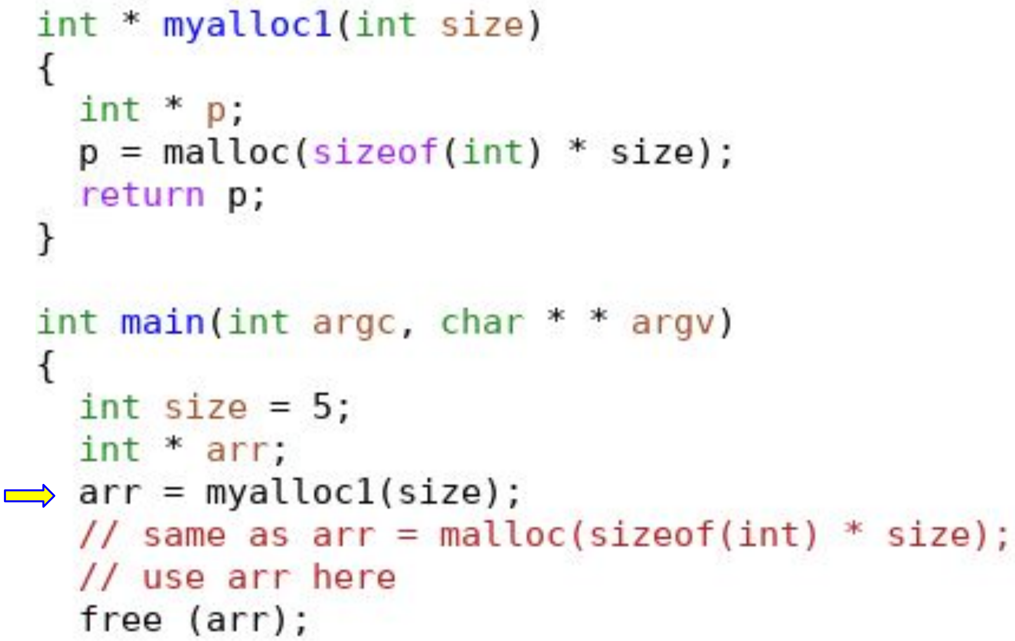


Heap Memory

Symbol	Address	Value
arr[4]	2016	U
arr[3]	2012	U
arr[2]	2008	U
arr[1]	2004	U
arr[0]	2000	U

Stack Memory

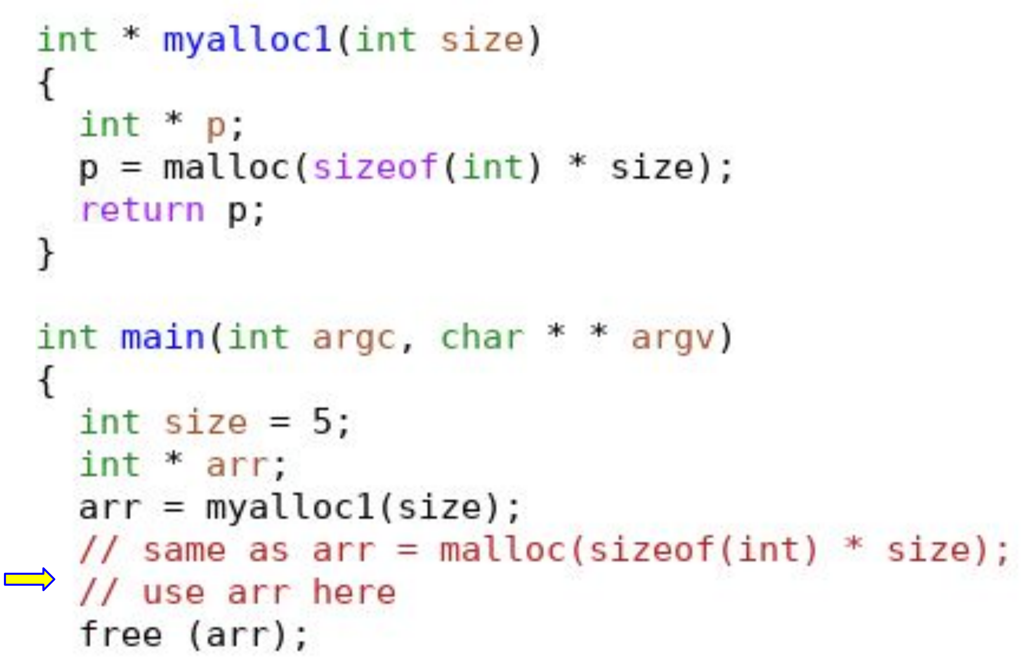
Frame	Symbol	Address	Value
myalloc1	p	204	A2000
	size	200	5
	Value Address 104		
	Return Location		
main	arr	104	U
	size	100	5



```
int * myalloc1(int size)
{
  int * p;
  p = malloc(sizeof(int) * size);
  return p;
}

int main(int argc, char * * argv)
{
  int size = 5;
  int * arr;
  arr = myalloc1(size);
  // same as arr = malloc(sizeof(int) * size);
  // use arr here
  free (arr);
```

Heap Memory		
Symbol	**Address**	**Value**
arr[4]	2016	U
arr[3]	2012	U
arr[2]	2008	U
arr[1]	2004	U
arr[0]	2000	U

Stack Memory			
Frame	**Symbol**	**Address**	**Value**
myalloc1	p	204	A2000
	size	200	5
	Value Address 104		
	Return Location		
main	arr	104	A2000
	size	100	5



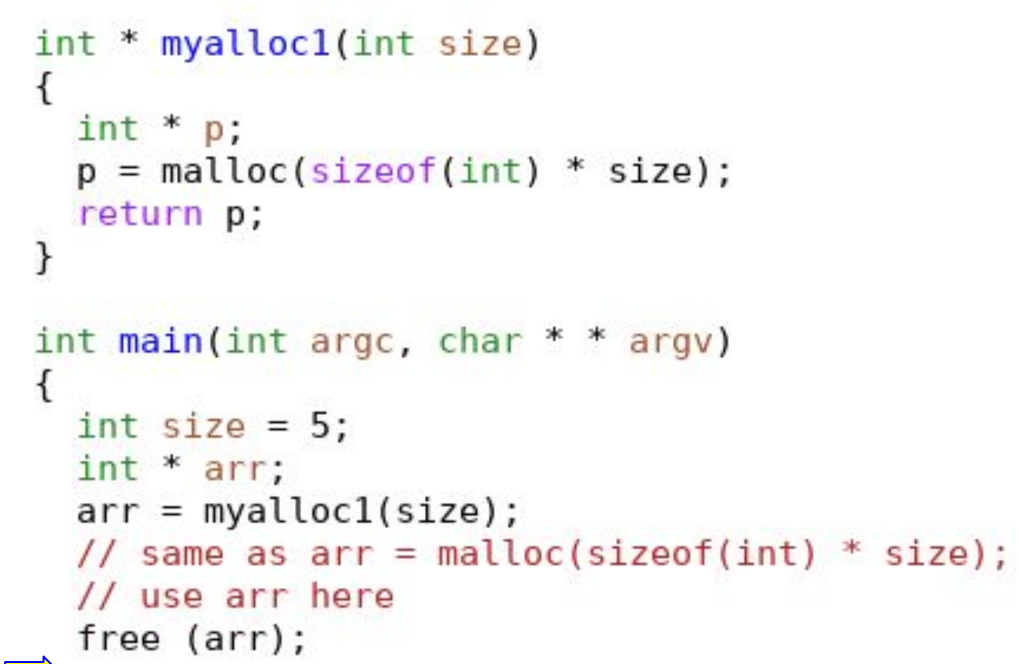
```
int * myalloc1(int size)
{
  int * p;
  p = malloc(sizeof(int) * size);
  return p;
}

int main(int argc, char * * argv)
{
  int size = 5;
  int * arr;
  arr = myalloc1(size);
  // same as arr = malloc(sizeof(int) * size);
  // use arr here
  free (arr);
```

Heap Memory

Symbol	Address	Value
arr[4]	2016	U
arr[3]	2012	U
arr[2]	2008	U
arr[1]	2004	U
arr[0]	2000	U

Stack Memory

Frame	Symbol	Address	Value
main	arr	104	A2000
	size	100	5



```
int * myalloc1(int size)
{
  int * p;
  p = malloc(sizeof(int) * size);
  return p;
}

int main(int argc, char * * argv)
{
  int size = 5;
  int * arr;
  arr = myalloc1(size);
  // same as arr = malloc(sizeof(int) * size);
  // use arr here
  free (arr);
```

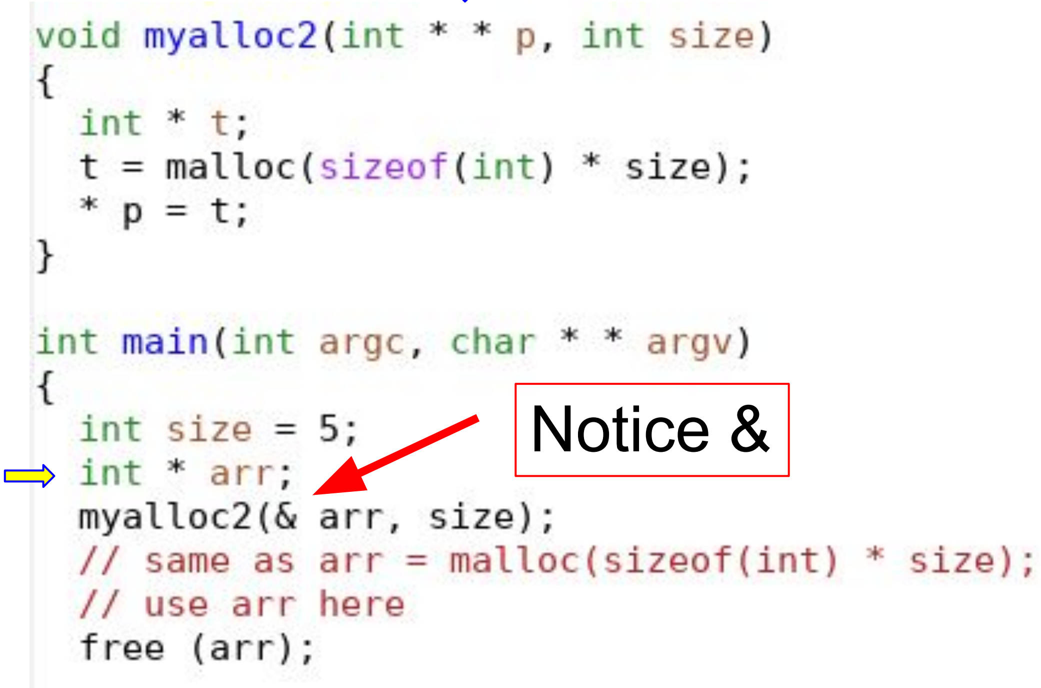
Heap Memory		
Symbol	**Address**	**Value**

Stack Memory			
Frame	**Symbol**	**Address**	**Value**
main	arr	104	A2000
	size	100	5

Remember to call free even though `malloc` is not called in the same function

To modify the value in another frame of the stack, pass the address.

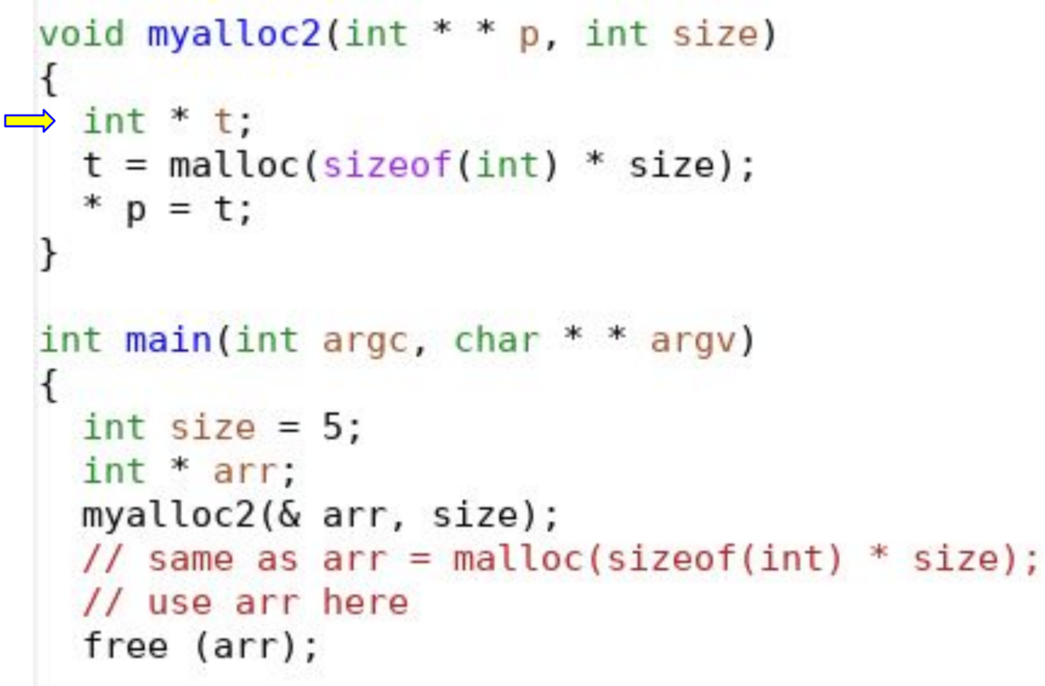
Notice * *



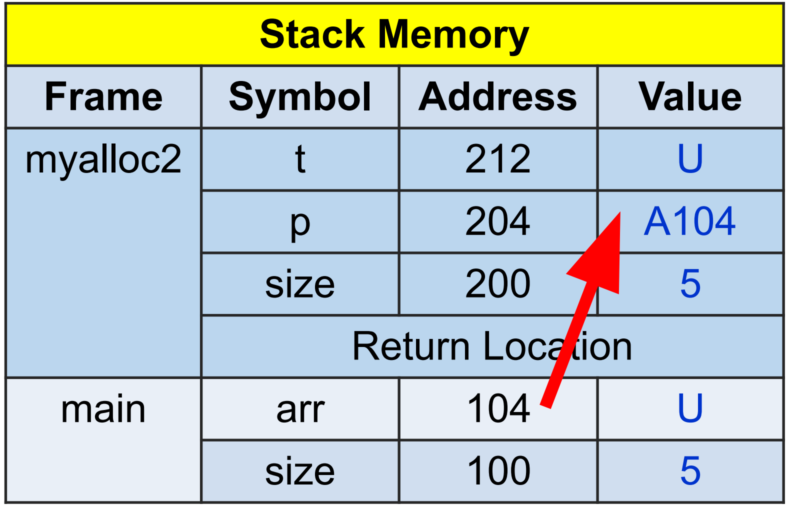
```
void myalloc2(int * * p, int size)
{
  int * t;
  t = malloc(sizeof(int) * size);
  * p = t;
}

int main(int argc, char * * argv)
{
  int size = 5;
  int * arr;
  myalloc2(& arr, size);
  // same as arr = malloc(sizeof(int) * size);
  // use arr here
  free (arr);
```

Notice &

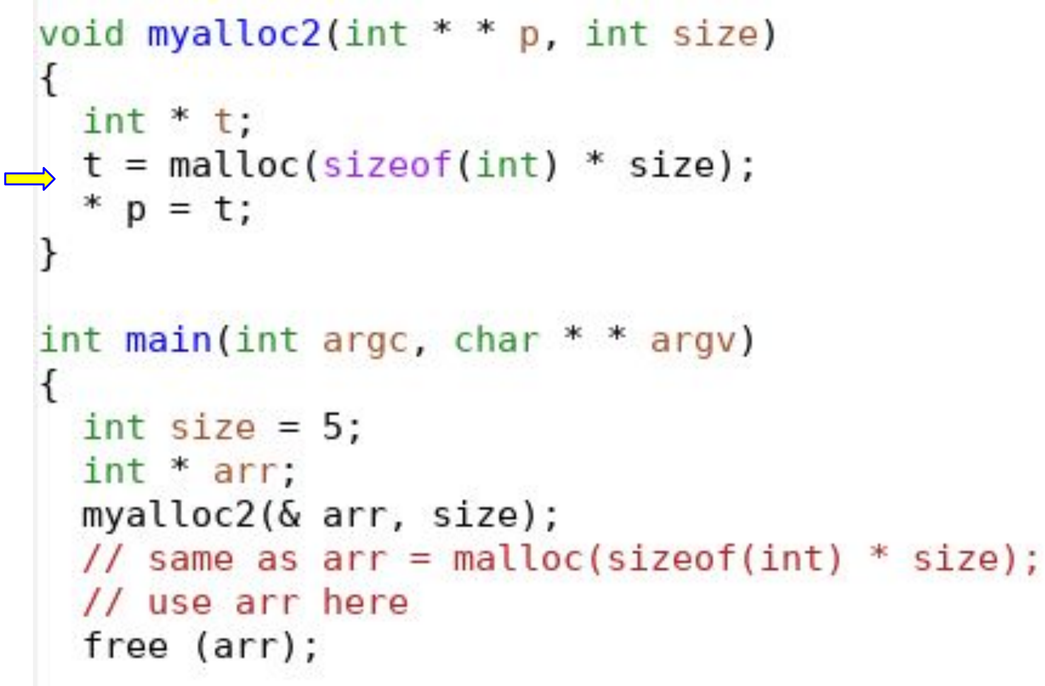
Stack Memory			
Frame	**Symbol**	**Address**	**Value**
main	arr	104	U
	size	100	5



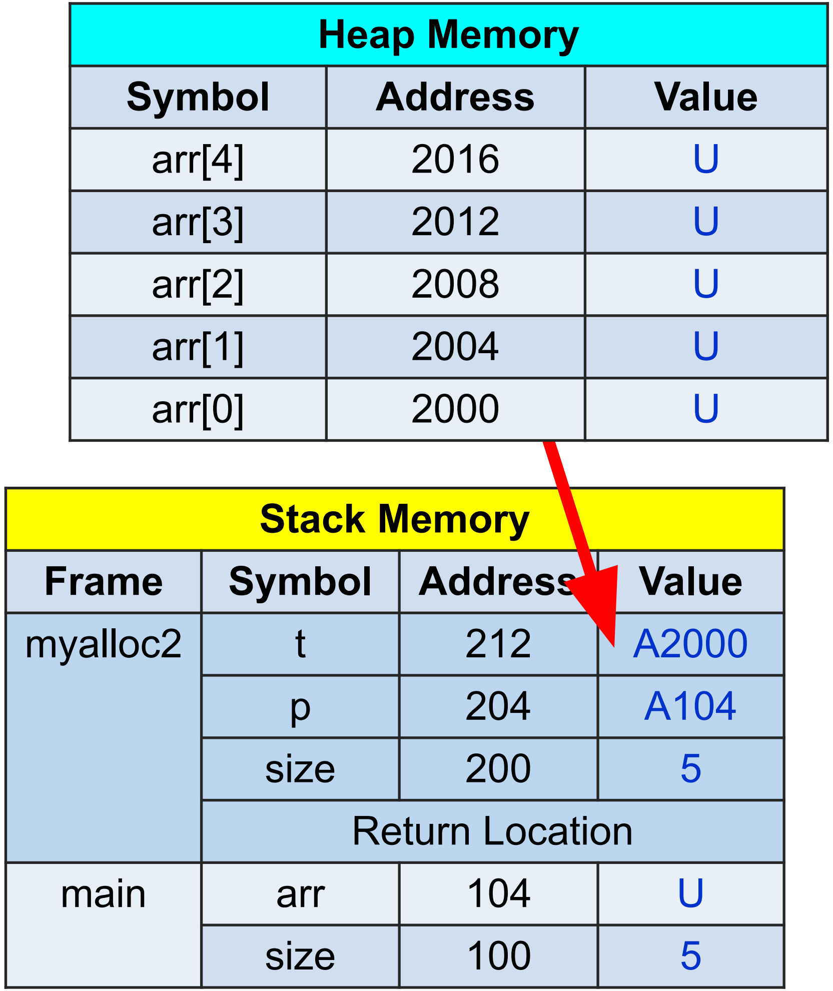
```
void myalloc2(int * * p, int size)
{
  int * t;
  t = malloc(sizeof(int) * size);
  * p = t;
}

int main(int argc, char * * argv)
{
  int size = 5;
  int * arr;
  myalloc2(& arr, size);
  // same as arr = malloc(sizeof(int) * size);
  // use arr here
  free (arr);
```



Stack Memory			
Frame	**Symbol**	**Address**	**Value**
myalloc2	t	212	U
	p	204	A104
	size	200	5
	Return Location		
main	arr	104	U
	size	100	5

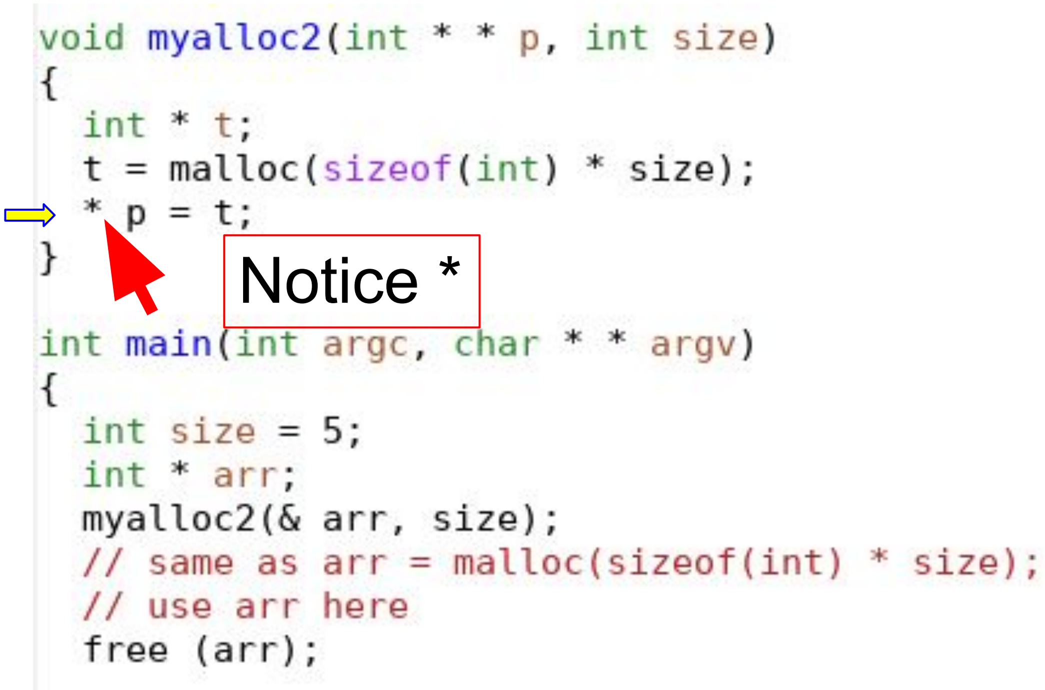


```
void myalloc2(int * * p, int size)
{
  int * t;
  t = malloc(sizeof(int) * size);
  * p = t;
}

int main(int argc, char * * argv)
{
  int size = 5;
  int * arr;
  myalloc2(& arr, size);
  // same as arr = malloc(sizeof(int) * size);
  // use arr here
  free (arr);
```



Heap Memory		
Symbol	**Address**	**Value**
arr[4]	2016	U
arr[3]	2012	U
arr[2]	2008	U
arr[1]	2004	U
arr[0]	2000	U

Stack Memory			
Frame	**Symbol**	**Address**	**Value**
myalloc2	t	212	A2000
	p	204	A104
	size	200	5
	Return Location		
main	arr	104	U
	size	100	5

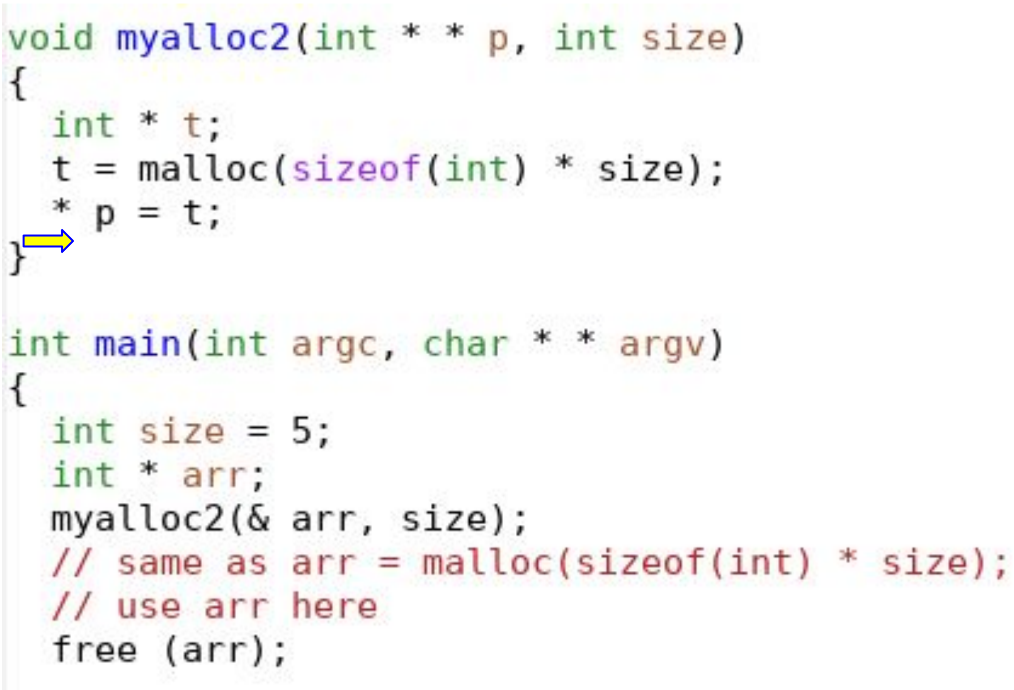


```
void myalloc2(int * * p, int size)
{
  int * t;
  t = malloc(sizeof(int) * size);
  * p = t;
}

int main(int argc, char * * argv)
{
  int size = 5;
  int * arr;
  myalloc2(& arr, size);
  // same as arr = malloc(sizeof(int) * size);
  // use arr here
  free (arr);
```

Notice *

Heap Memory		
Symbol	**Address**	**Value**
arr[4]	2016	U
arr[3]	2012	U
arr[2]	2008	U
arr[1]	2004	U
arr[0]	2000	U

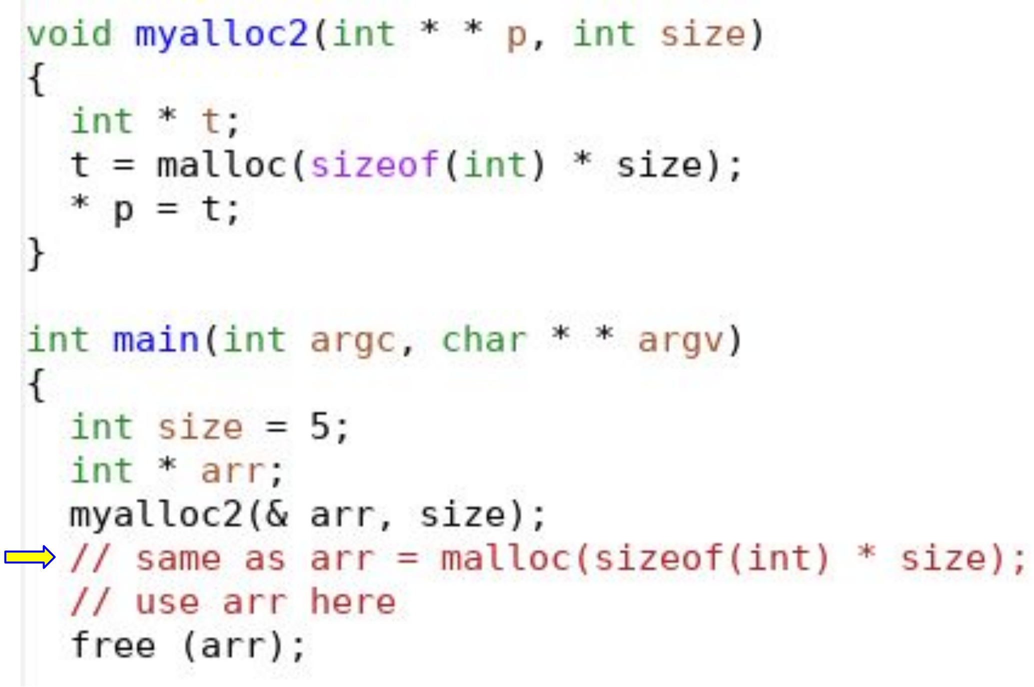
Stack Memory			
Frame	**Symbol**	**Address**	**Value**
myalloc2	t	212	A2000
	p	204	1 A104
	size	200	5
	Return Location		
main	arr	2 104	3 U
	size	100	5



```
void myalloc2(int * * p, int size)
{
  int * t;
  t = malloc(sizeof(int) * size);
  * p = t;
}

int main(int argc, char * * argv)
{
  int size = 5;
  int * arr;
  myalloc2(& arr, size);
  // same as arr = malloc(sizeof(int) * size);
  // use arr here
  free (arr);
```

Heap Memory		
Symbol	**Address**	**Value**
arr[4]	2016	U
arr[3]	2012	U
arr[2]	2008	U
arr[1]	2004	U
arr[0]	2000	U

Stack Memory			
Frame	**Symbol**	**Address**	**Value**
myalloc2	t	212	A2000
	p	204	A104
	size	200	5
	Return Location		
main	arr	104	A2000
	size	100	5



```
void myalloc2(int * * p, int size)
{
  int * t;
  t = malloc(sizeof(int) * size);
  * p = t;
}

int main(int argc, char * * argv)
{
  int size = 5;
  int * arr;
  myalloc2(& arr, size);
  // same as arr = malloc(sizeof(int) * size);
  // use arr here
  free (arr);
```

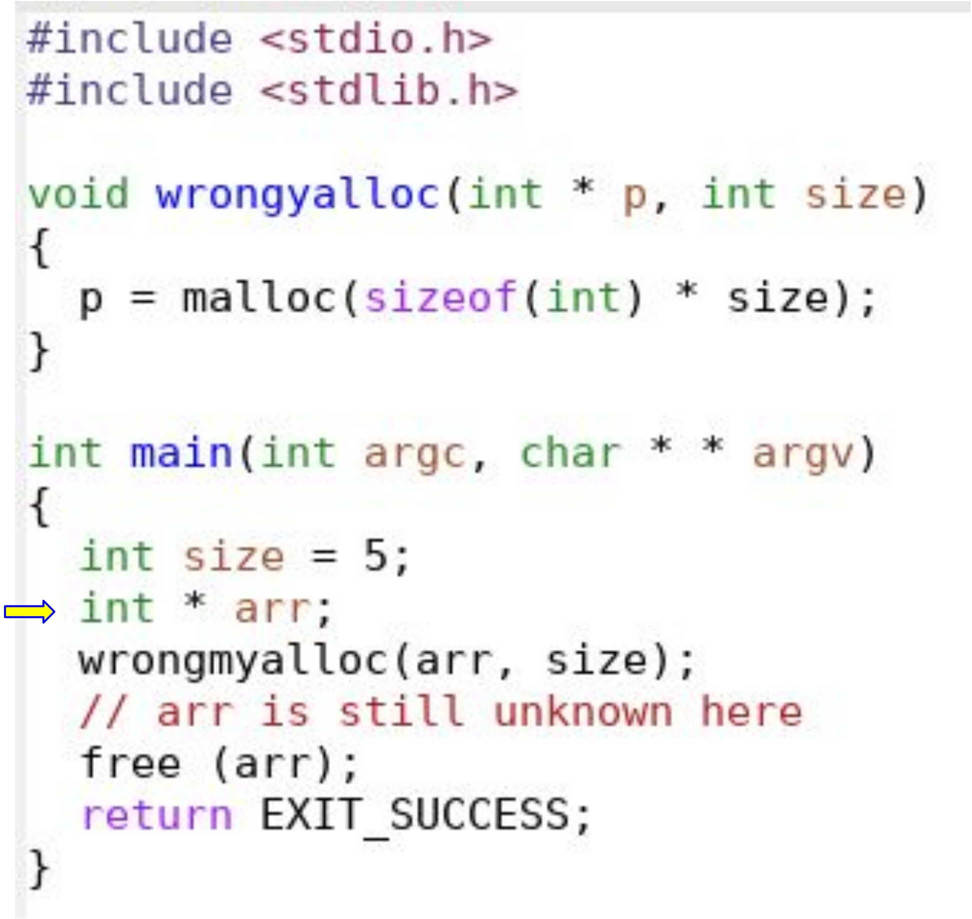
Heap Memory

Symbol	Address	Value
arr[4]	2016	U
arr[3]	2012	U
arr[2]	2008	U
arr[1]	2004	U
arr[0]	2000	U

Stack Memory

Frame	Symbol	Address	Value
main	arr	104	A2000
	size	100	5

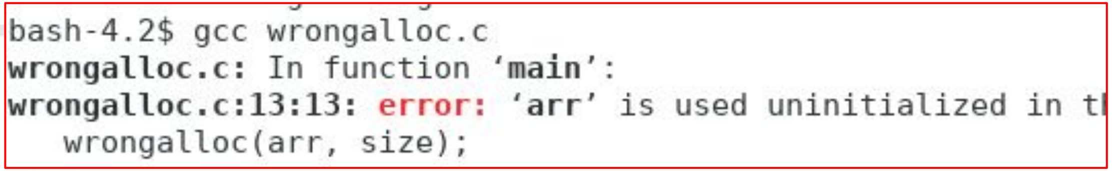
Common Mistake



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

void wrongyalloc(int * p, int size)
{
  p = malloc(sizeof(int) * size);
}

int main(int argc, char * * argv)
{
  int size = 5;
  int * arr;
  wrongmyalloc(arr, size);
  // arr is still unknown here
  free (arr);
  return EXIT_SUCCESS;
}
```

Stack Memory

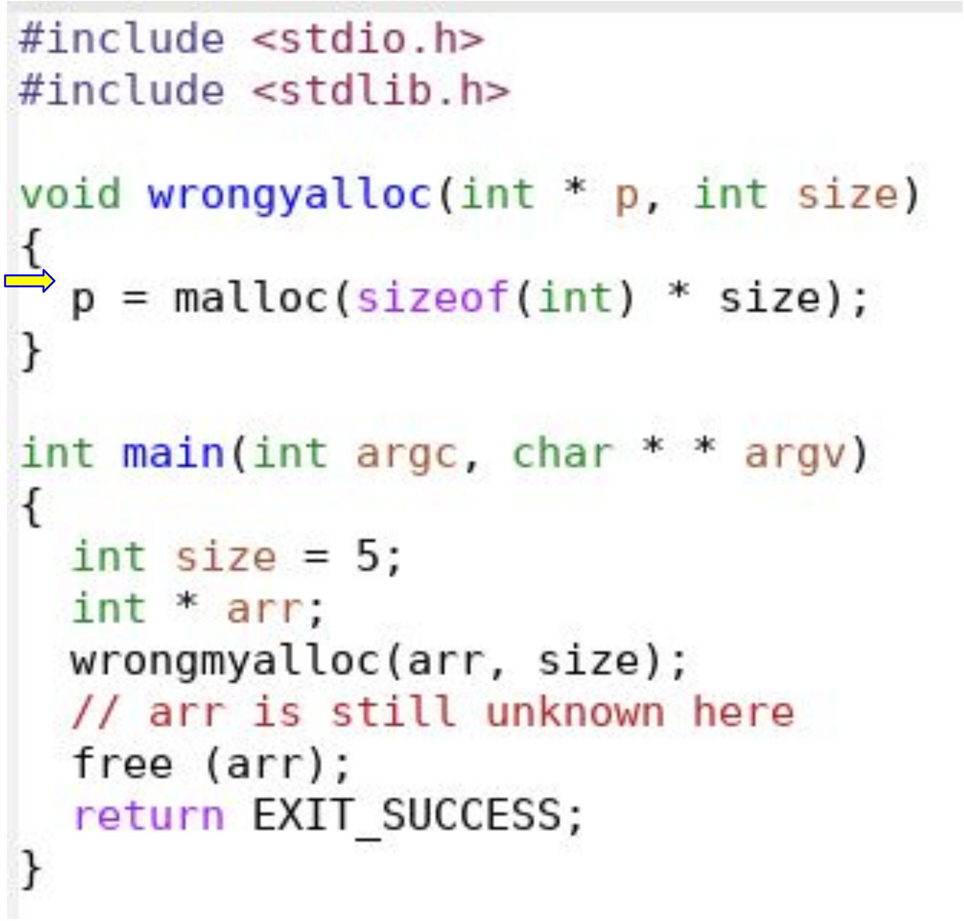
Frame	Symbol	Address	Value
main	arr	104	U
	size	100	5



```
bash-4.2$ gcc wrongalloc.c
wrongalloc.c: In function ‘main’:
wrongalloc.c:13:13: error: ‘arr’ is used uninitialized in t
   wrongalloc(arr, size);
```

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

void wrongyalloc(int * p, int size)
{
  p = malloc(sizeof(int) * size);
}

int main(int argc, char * * argv)
{
  int size = 5;
  int * arr;
  wrongmyalloc(arr, size);
  // arr is still unknown here
  free (arr);
  return EXIT_SUCCESS;
}
```

Stack Memory			
Frame	**Symbol**	**Address**	**Value**
main	arr	104	U
	size	100	5



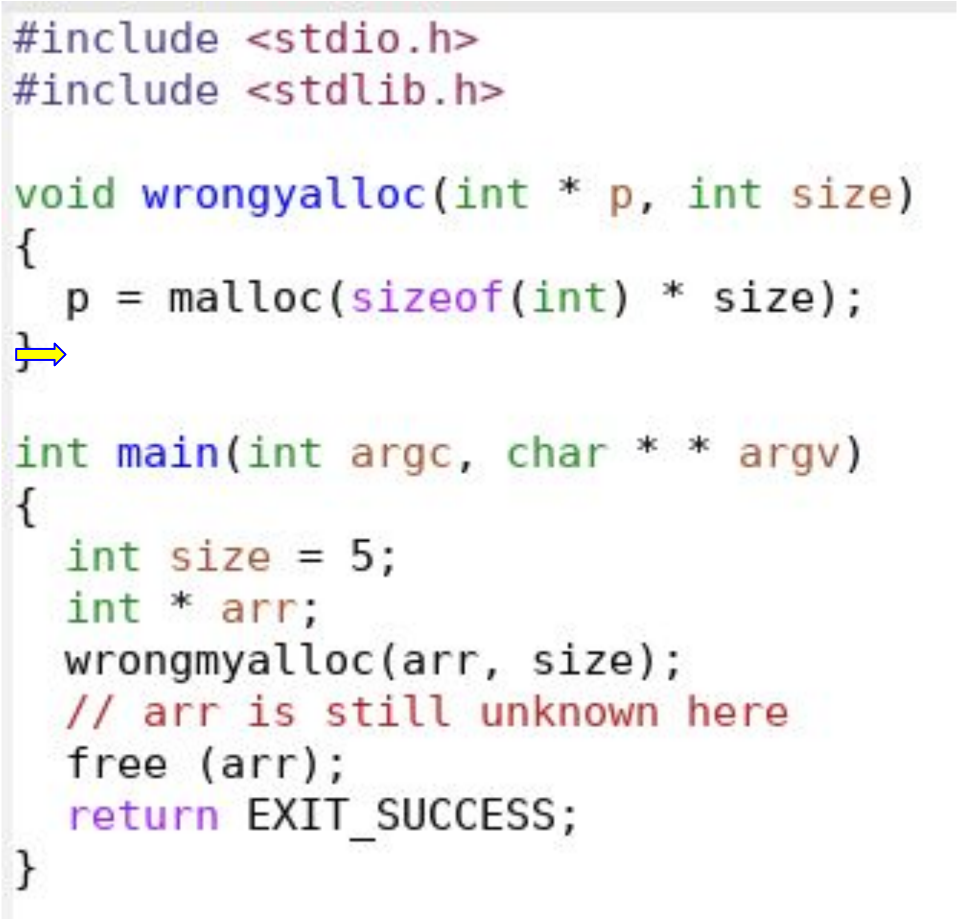
```
#include <stdio.h>
#include <stdlib.h>

void wrongyalloc(int * p, int size)
{
  p = malloc(sizeof(int) * size);
}

int main(int argc, char * * argv)
{
  int size = 5;
  int * arr;
  wrongmyalloc(arr, size);
  // arr is still unknown here
  free (arr);
  return EXIT_SUCCESS;
}
```

Heap Memory

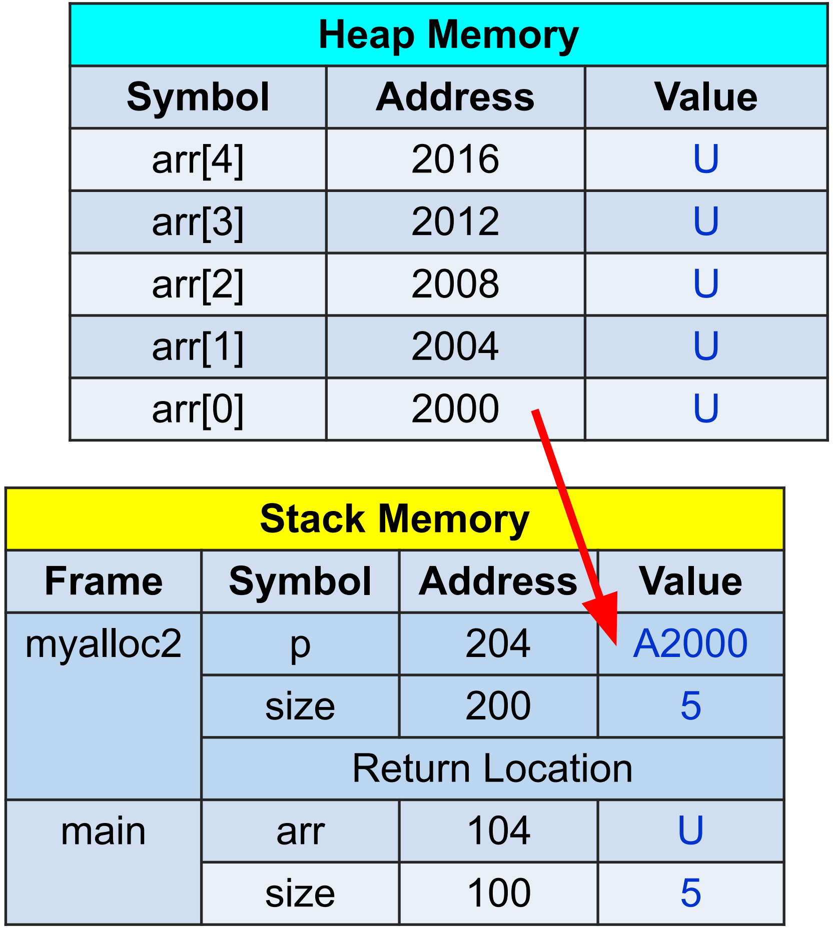
Symbol	Address	Value

Stack Memory

Frame	Symbol	Address	Value
myalloc2	p	204	U
	size	200	5
	Return Location		
main	arr	104	U
	size	100	5

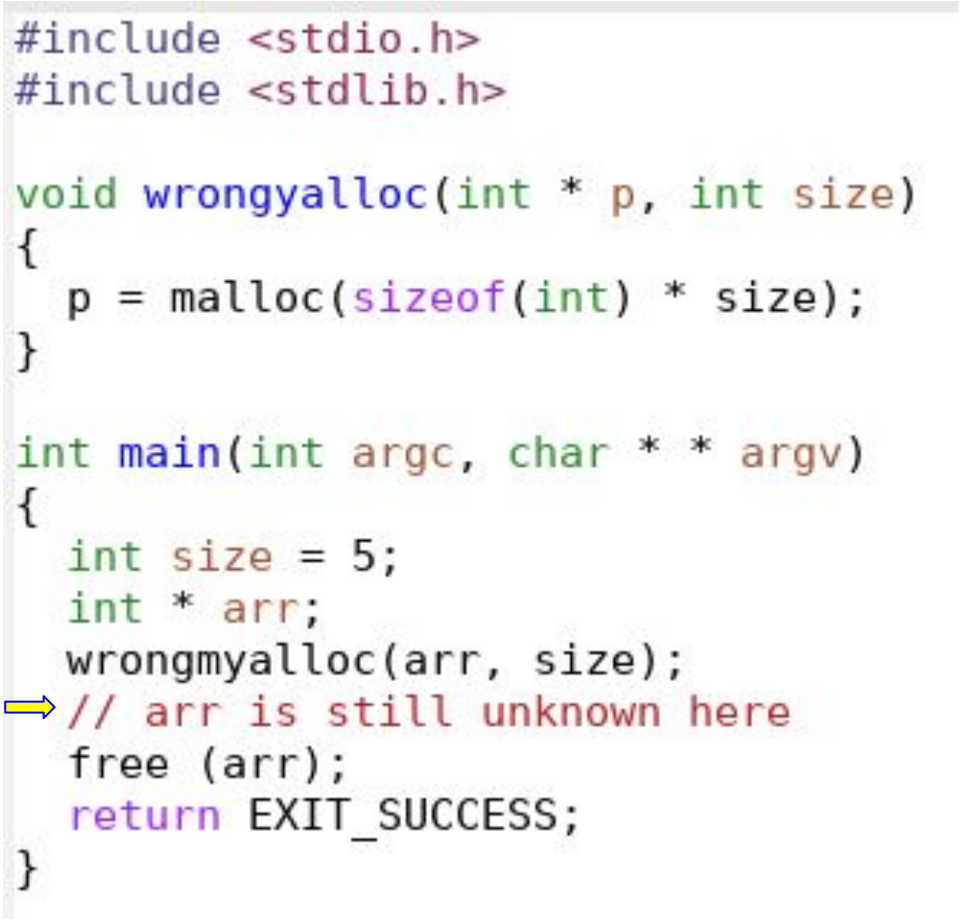


```
#include <stdio.h>
#include <stdlib.h>

void wrongyalloc(int * p, int size)
{
  p = malloc(sizeof(int) * size);
}

int main(int argc, char * * argv)
{
  int size = 5;
  int * arr;
  wrongmyalloc(arr, size);
  // arr is still unknown here
  free (arr);
  return EXIT_SUCCESS;
}
```



Heap Memory		
Symbol	**Address**	**Value**
arr[4]	2016	U
arr[3]	2012	U
arr[2]	2008	U
arr[1]	2004	U
arr[0]	2000	U

Stack Memory			
Frame	**Symbol**	**Address**	**Value**
myalloc2	p	204	A2000
	size	200	5
	Return Location		
main	arr	104	U
	size	100	5



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

void wrongyalloc(int * p, int size)
{
  p = malloc(sizeof(int) * size);
}

int main(int argc, char * * argv)
{
  int size = 5;
  int * arr;
  wrongmyalloc(arr, size);
  // arr is still unknown here
  free (arr);
  return EXIT_SUCCESS;
}
```

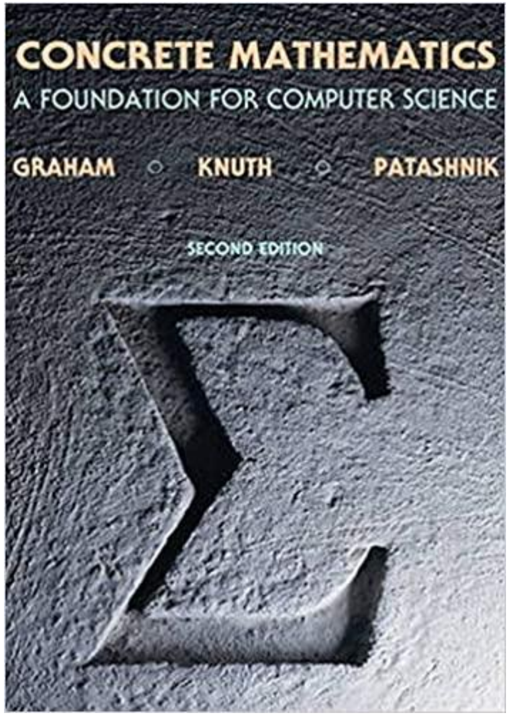
Heap Memory

Symbol	Address	Value
arr[4]	2016	U
arr[3]	2012	U
arr[2]	2008	U
arr[1]	2004	U
arr[0]	2000	U

Stack Memory

Frame	Symbol	Address	Value
main	arr	104	U
	size	100	5

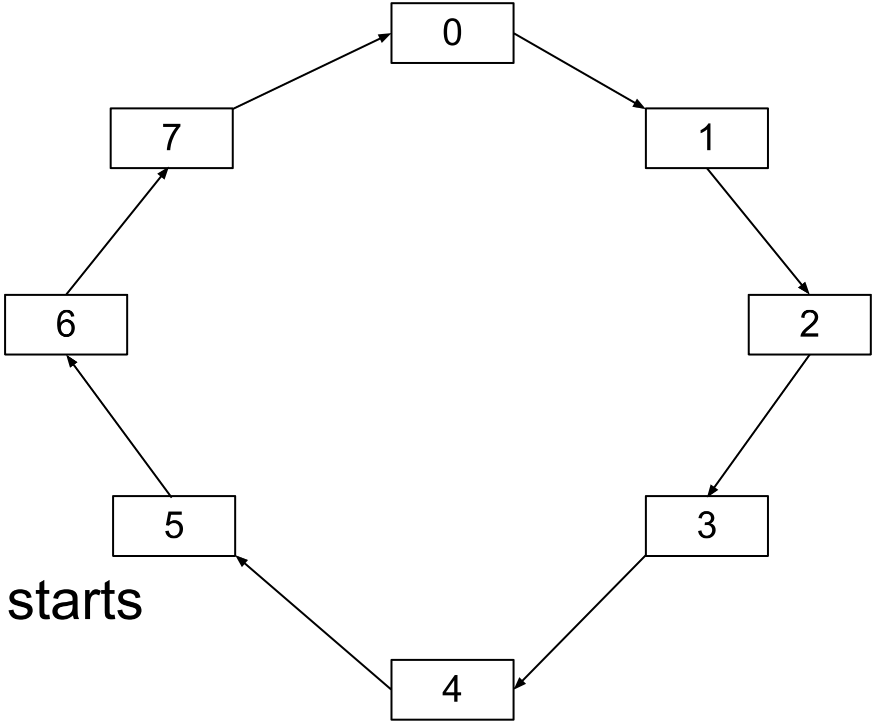
Homework 4
Who Gets the Cake?
(Inspired by 1.3 of Concrete Mathematics)



yunglu@purdue.

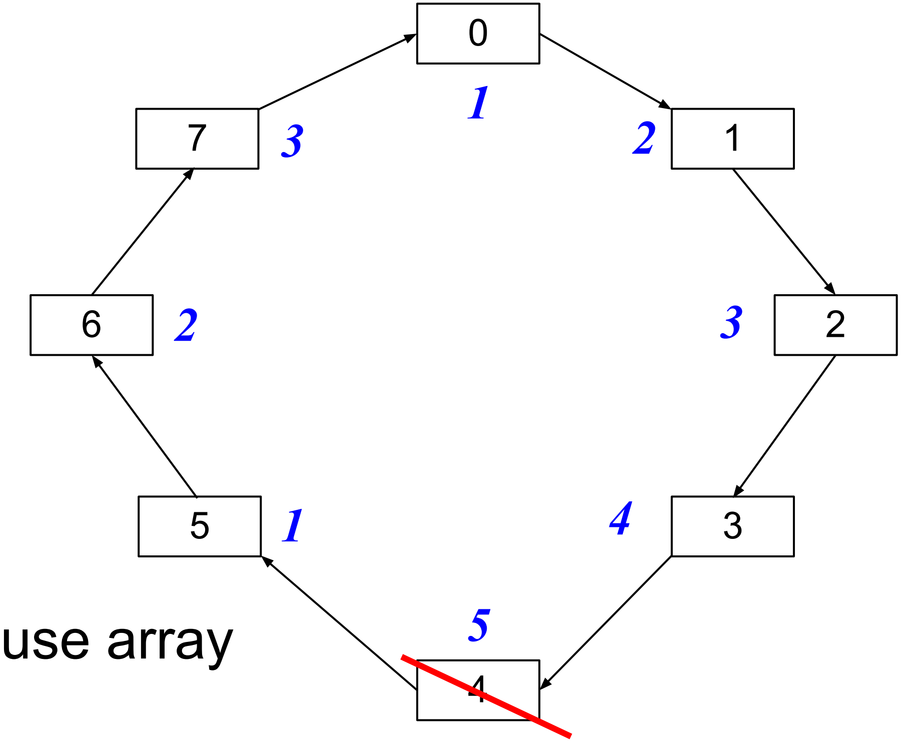
Let’s Play a Game

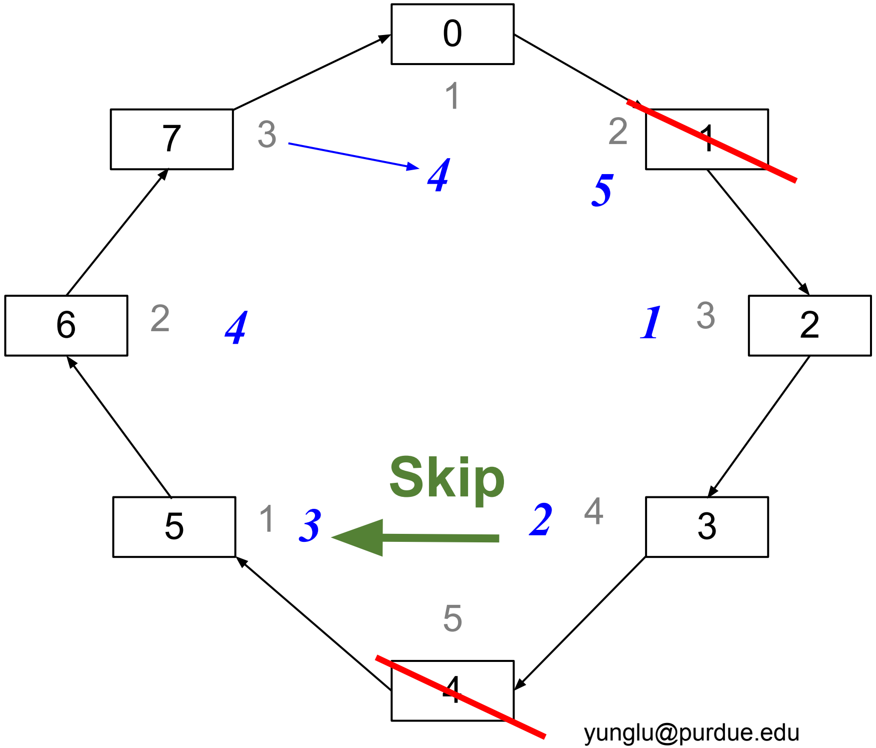
- A group of n people.
- There is only one slice of cake.
- Who gets the cake?
- The people form a circle.
- A number k (k > 1) is selected
- We use an array. In C, array index starts from zero (not one).



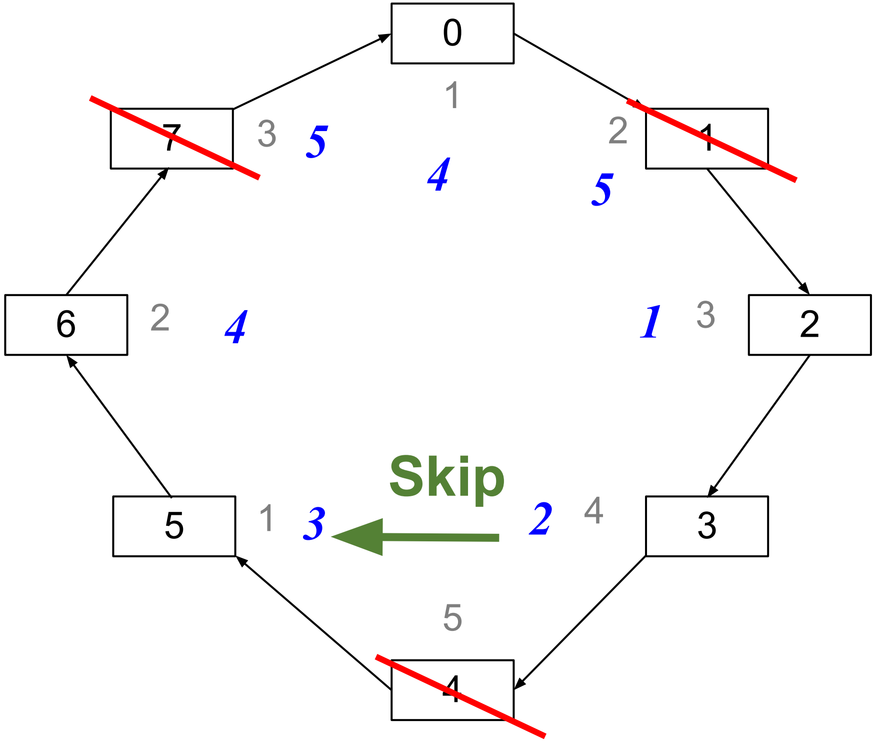
Count to k

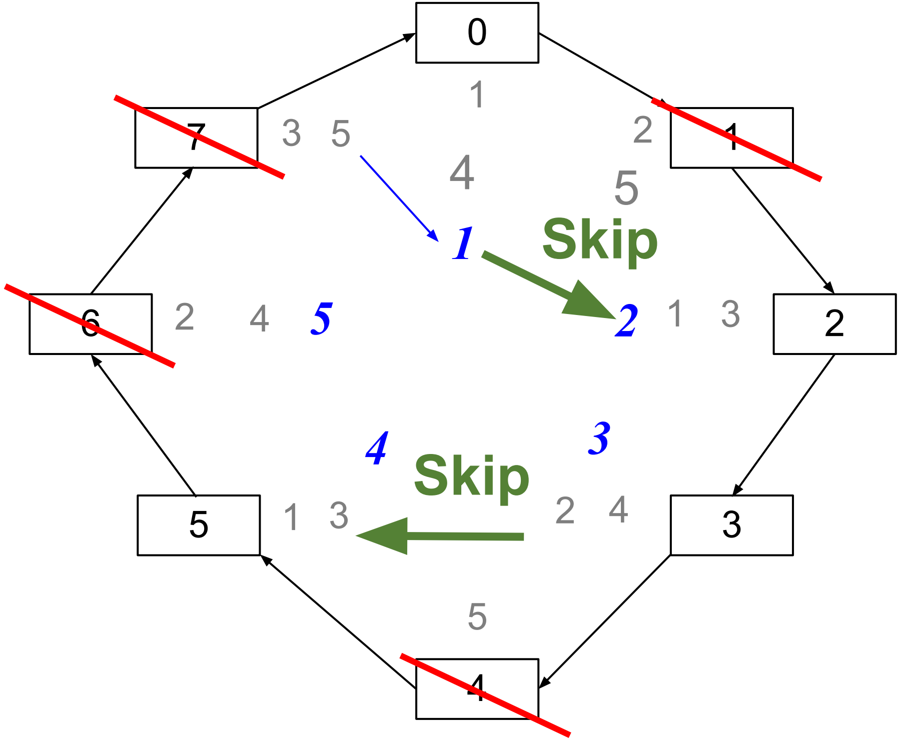
- The kth person is removed
- Keep counting
- Wrap around to the beginning
- n = 8 in this case
- choose k = 5
- This example uses 0, 1, 2… because array indexes start from zero.

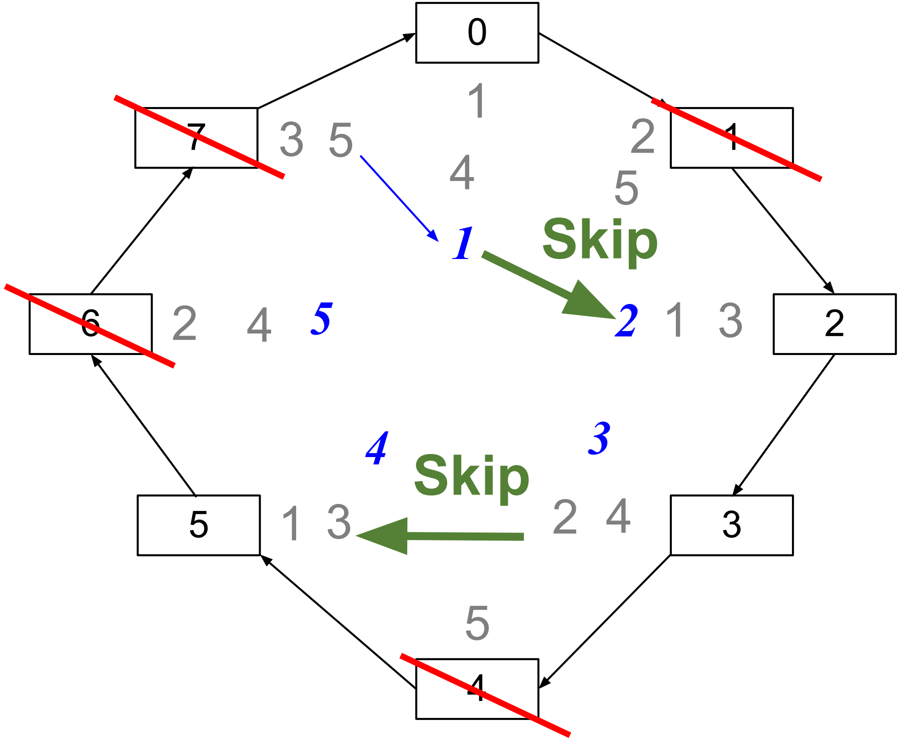




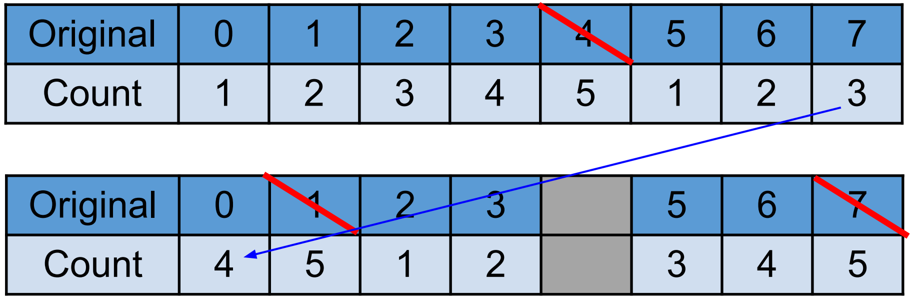
yunglu@purdue.edu




0
1
2
3
4
5
6
7
1
2
3 5
4
5
1
Skip
2 1 3
2 4 5
4
3
Skip
2 4
1 3
5

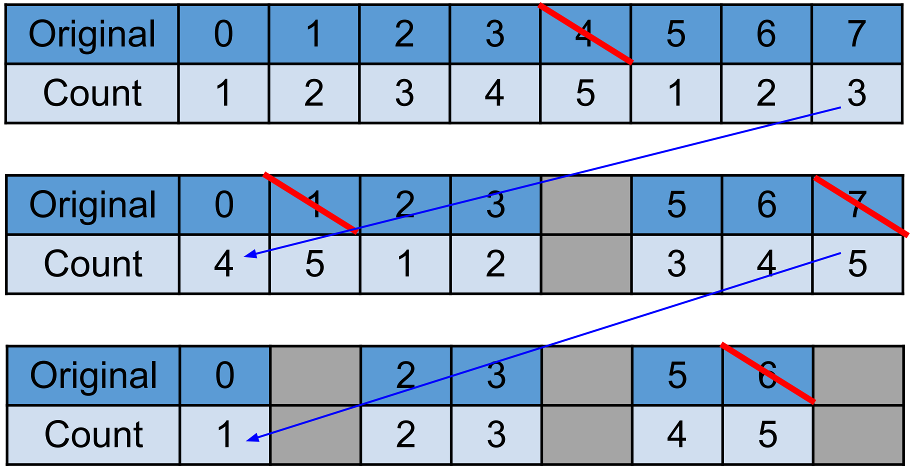


Original	0	1	2	3	~~4~~	5	6	7
Count	1	2	3	4	5	1	2	3



Original	0	1	2	3	~~4~~	5	6	7
Count	1	2	3	4	5	1	2	3

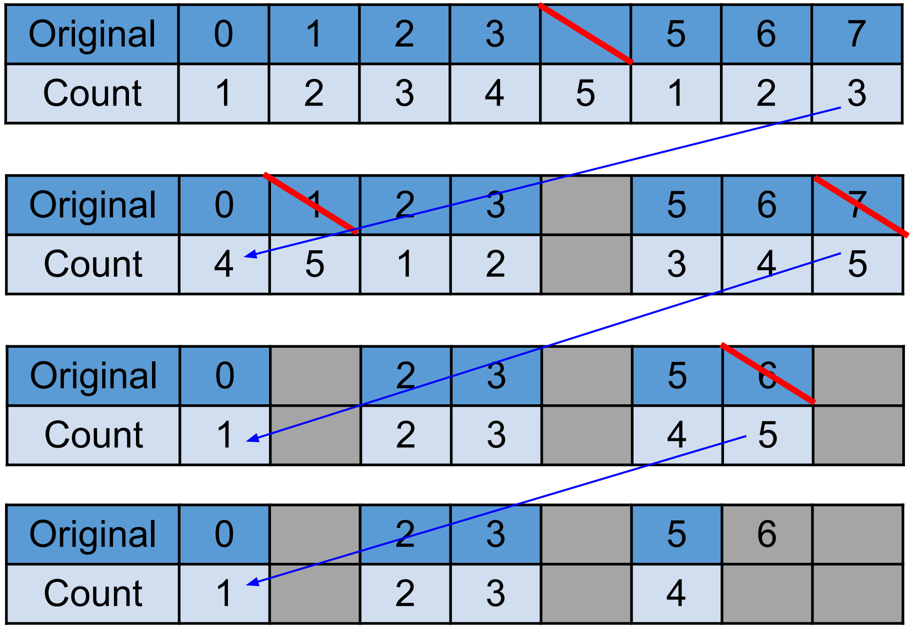
Original	0	~~1~~	2	3		5	6	~~7~~
Count	4	5	1	2		3	4	5



Original	0	1	2	3	~~4~~	5	6	7
Count	1	2	3	4	5	1	2	3

Original	0	~~1~~	2	3		5	6	~~7~~
Count	4	5	1	2		3	4	5

Original	0		2	3		5	~~6~~	
Count	1		2	3		4	5	

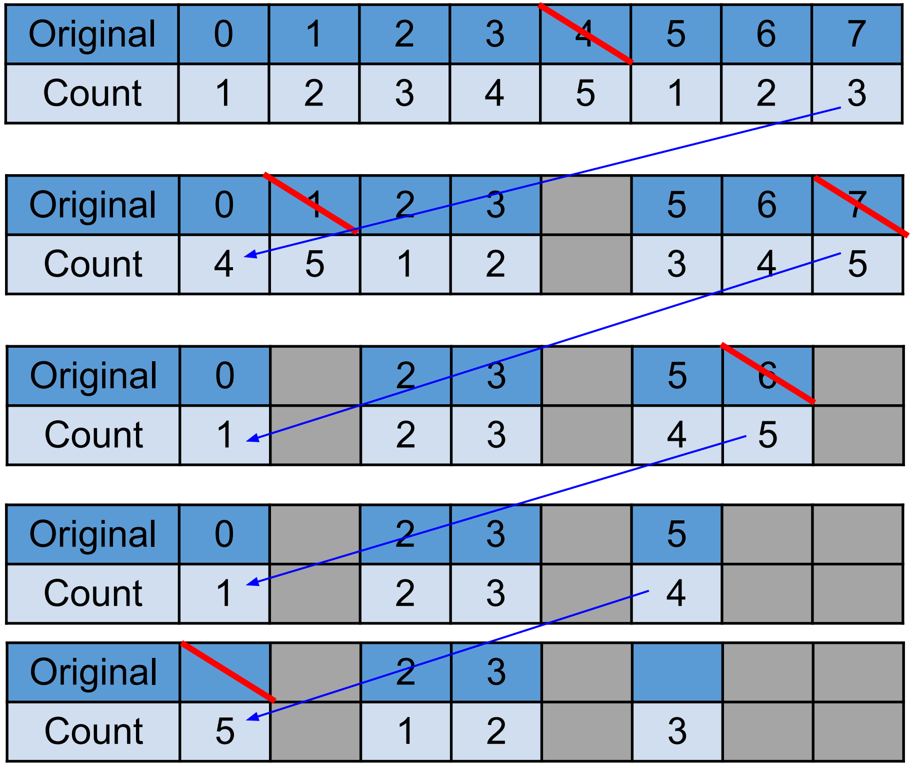


Original	0	1	2	3		5	6	7
Count	1	2	3	4	5	1	2	3

Original	0	~~1~~	2	3		5	6	~~7~~
Count	4	5	1	2		3	4	5

Original	0		2	3		5	~~6~~	
Count	1		2	3		4	5	

Original	0		2	3		5	6	
Count	1		2	3		4		



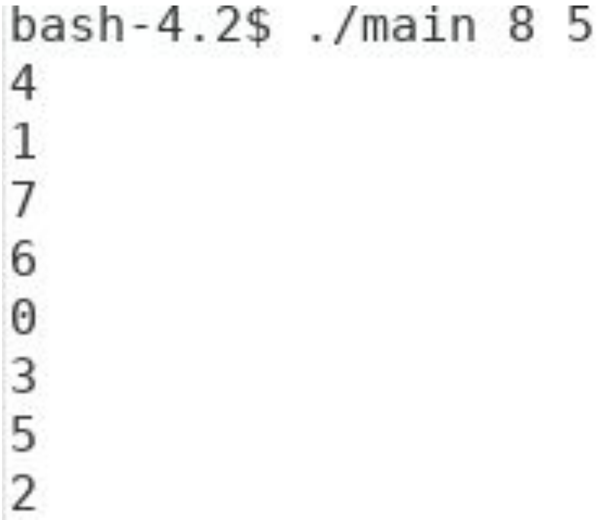
Original	0	1	2	3	~~4~~	5	6	7
Count	1	2	3	4	5	1	2	3

Original	0	~~1~~	2	3		5	6	~~7~~
Count	4	5	1	2		3	4	5

Original	0		2	3		5	~~6~~	
Count	1		2	3		4	5	

Original	0		2	3		5		
Count	1		2	3		4		

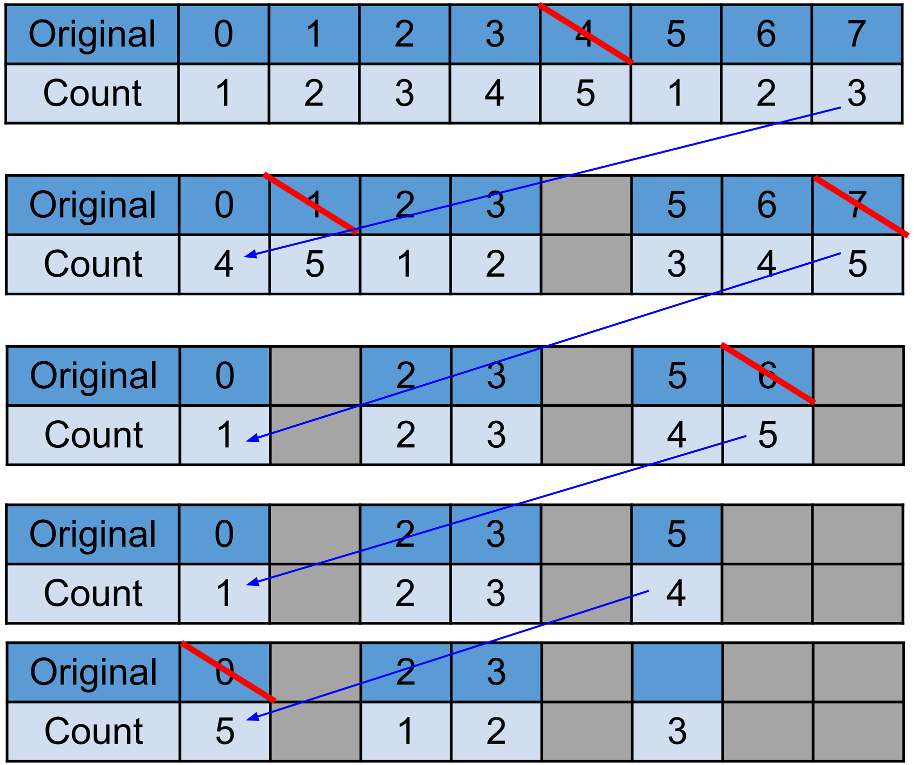
Original			2	3				
Count	5		1	2		3		



```
bash-4.2$ ./main 8 5
4
1
7
6
0
3
5
2
```

The program takes two numbers: n and k.

prints the order of removed people.



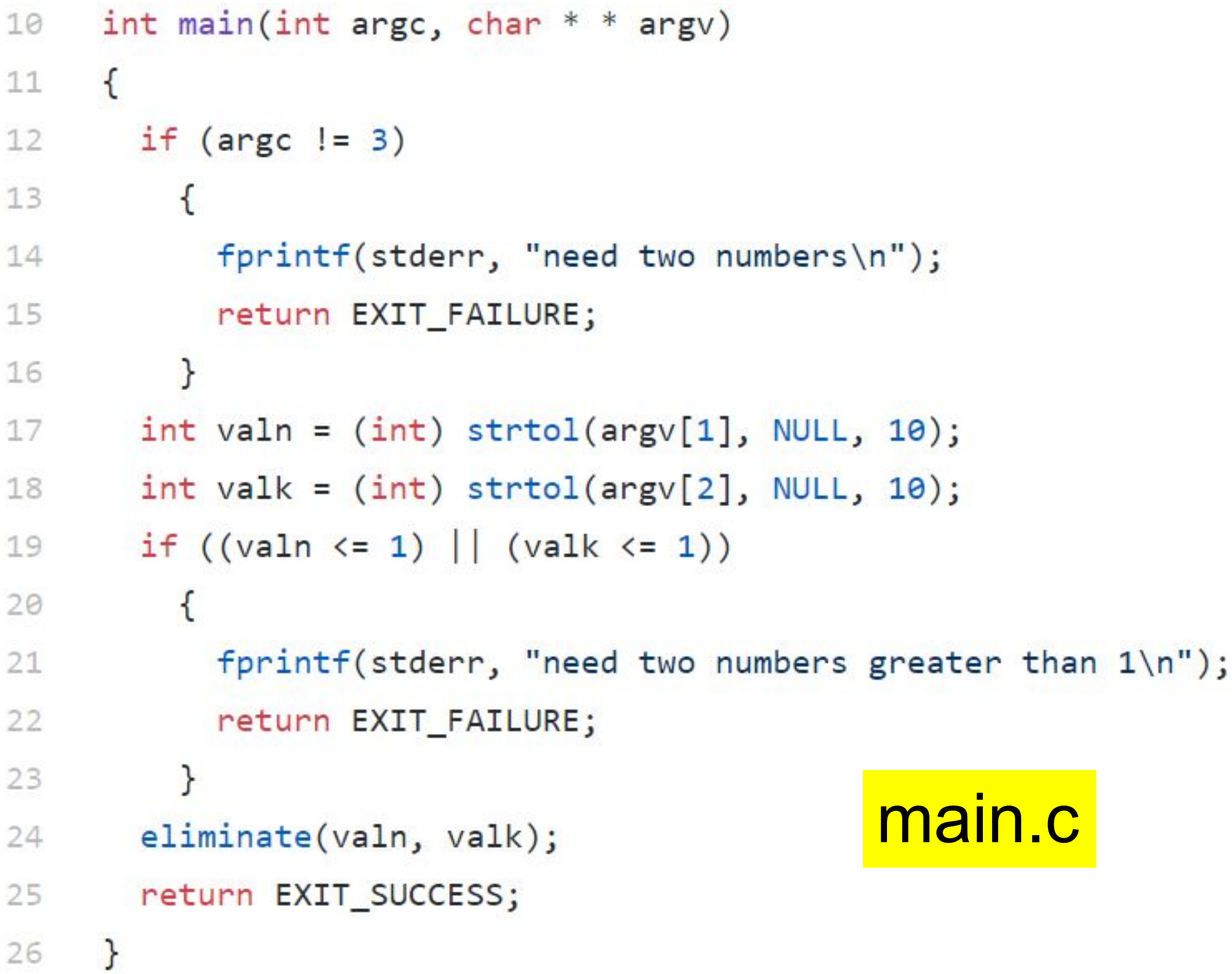
Original	0	1	2	3	~~4~~	5	6	7
Count	1	2	3	4	5	1	2	3

Original	0	~~1~~	2	3		5	6	~~7~~
Count	4	5	1	2		3	4	5

Original	0		2	3		5	~~6~~	
Count	1		2	3		4	5	

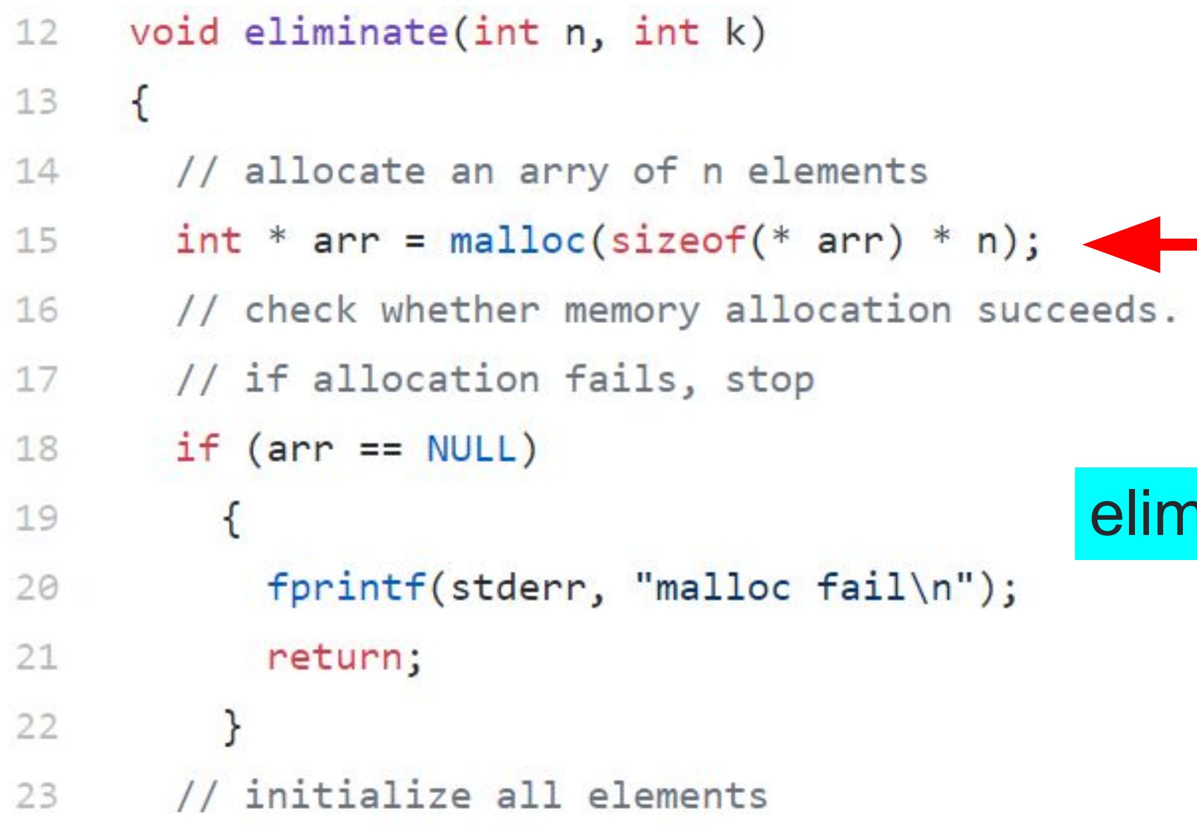
Original	0		2	3		5		
Count	1		2	3		4		

Original	~~0~~		2	3				
Count	5		1	2		3		



```c
int main(int argc, char * * argv)
{
  if (argc != 3)
    {
      fprintf(stderr, "need two numbers\n");
      return EXIT_FAILURE;
    }
  int valn = (int) strtol(argv[1], NULL, 10);
  int valk = (int) strtol(argv[2], NULL, 10);
  if ((valn <= 1) || (valk <= 1))
    {
      fprintf(stderr, "need two numbers greater than 1\n");
      return EXIT_FAILURE;
    }
  eliminate(valn, valk);
  return EXIT_SUCCESS;
}
```

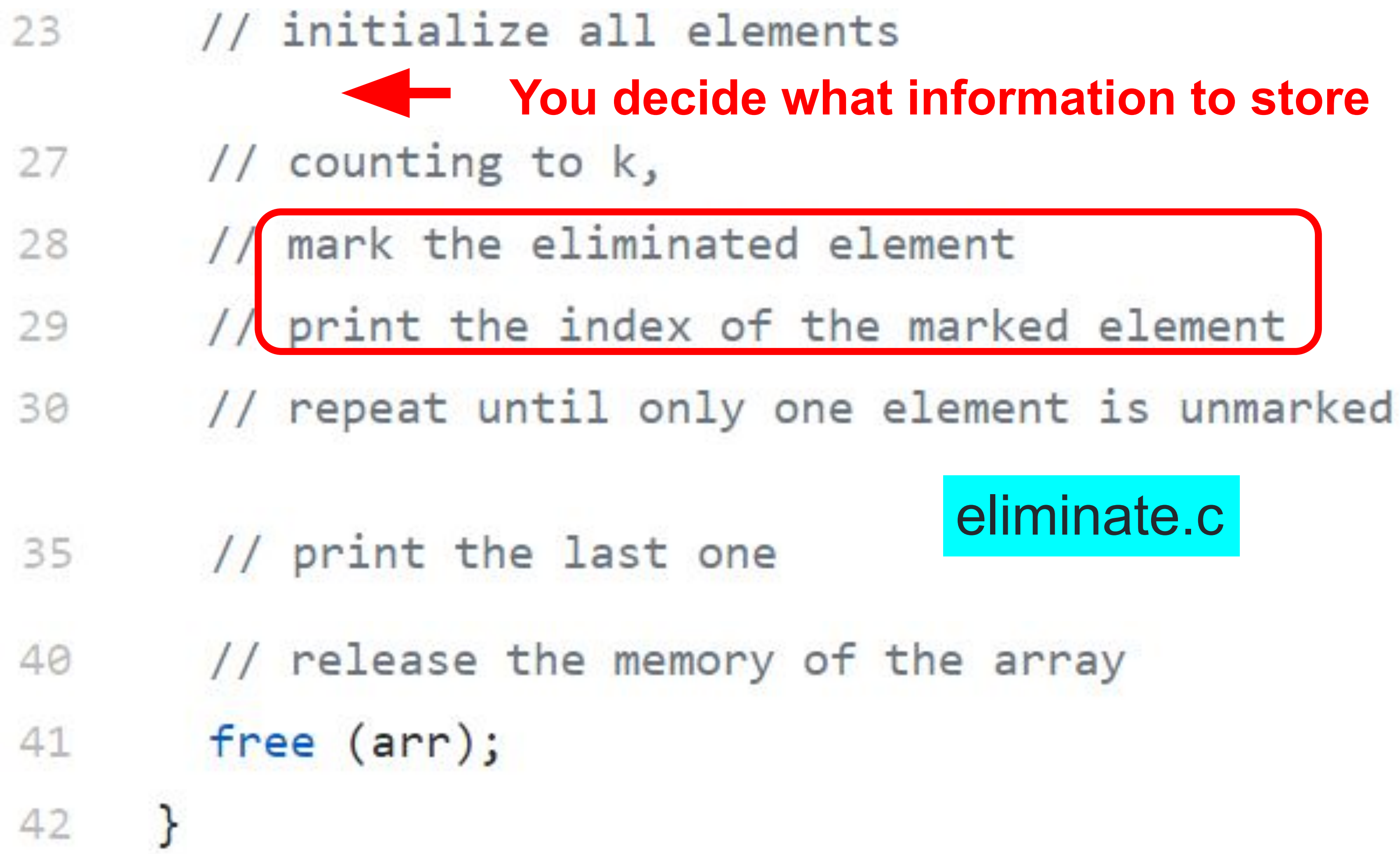
main.c



```c
void eliminate(int n, int k)
{
  // allocate an arry of n elements
  int * arr = malloc(sizeof(* arr) * n);
  // check whether memory allocation succeeds.
  // if allocation fails, stop
  if (arr == NULL)
    {
      fprintf(stderr, "malloc fail\n");
      return;
    }
  // initialize all elements
```

Each element is an int

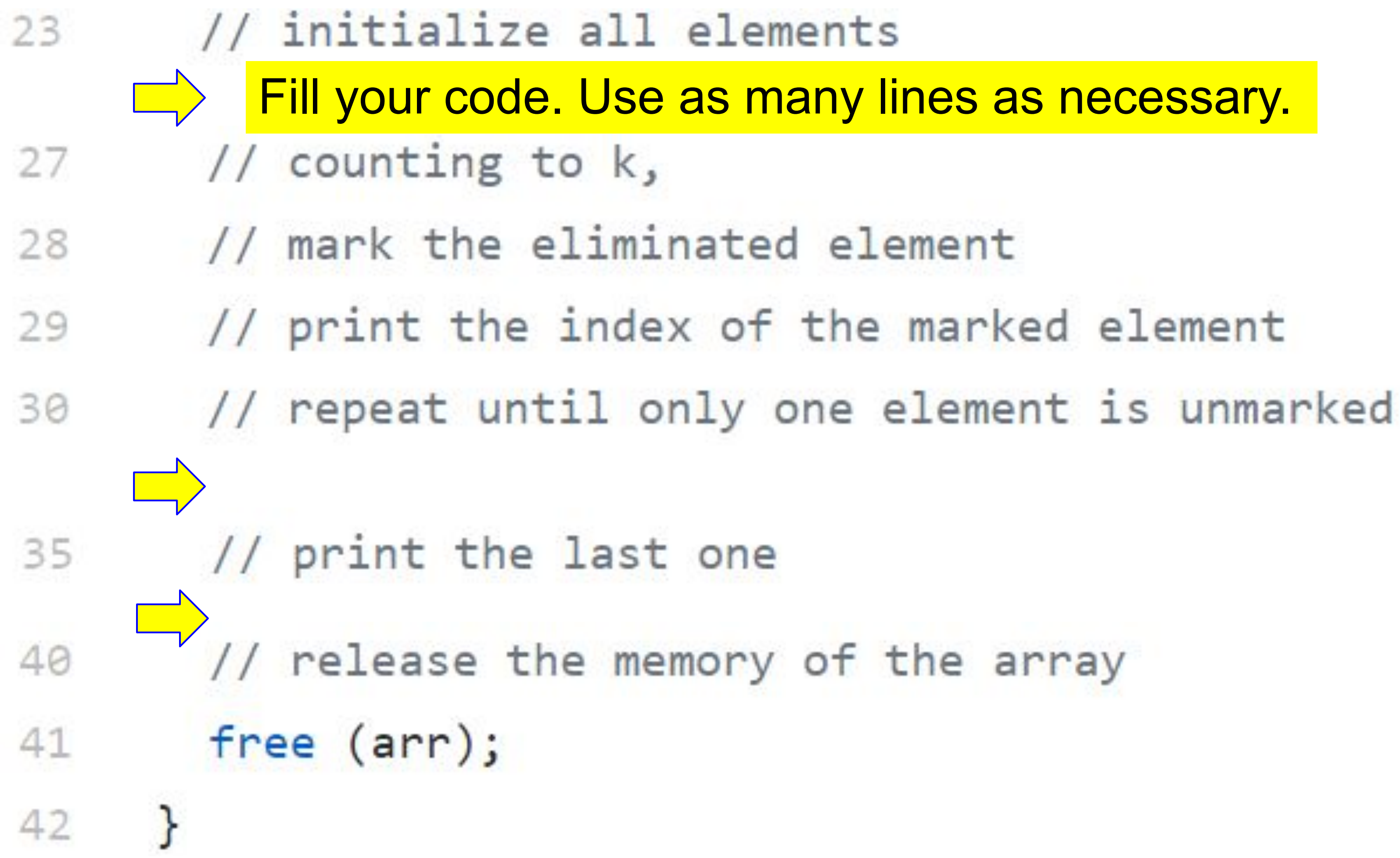
eliminate.c



```
    // initialize all elements

    // counting to k,
    // mark the eliminated element
    // print the index of the marked element
    // repeat until only one element is unmarked

    // print the last one

    // release the memory of the array
    free (arr);
}
```

You decide what information to store

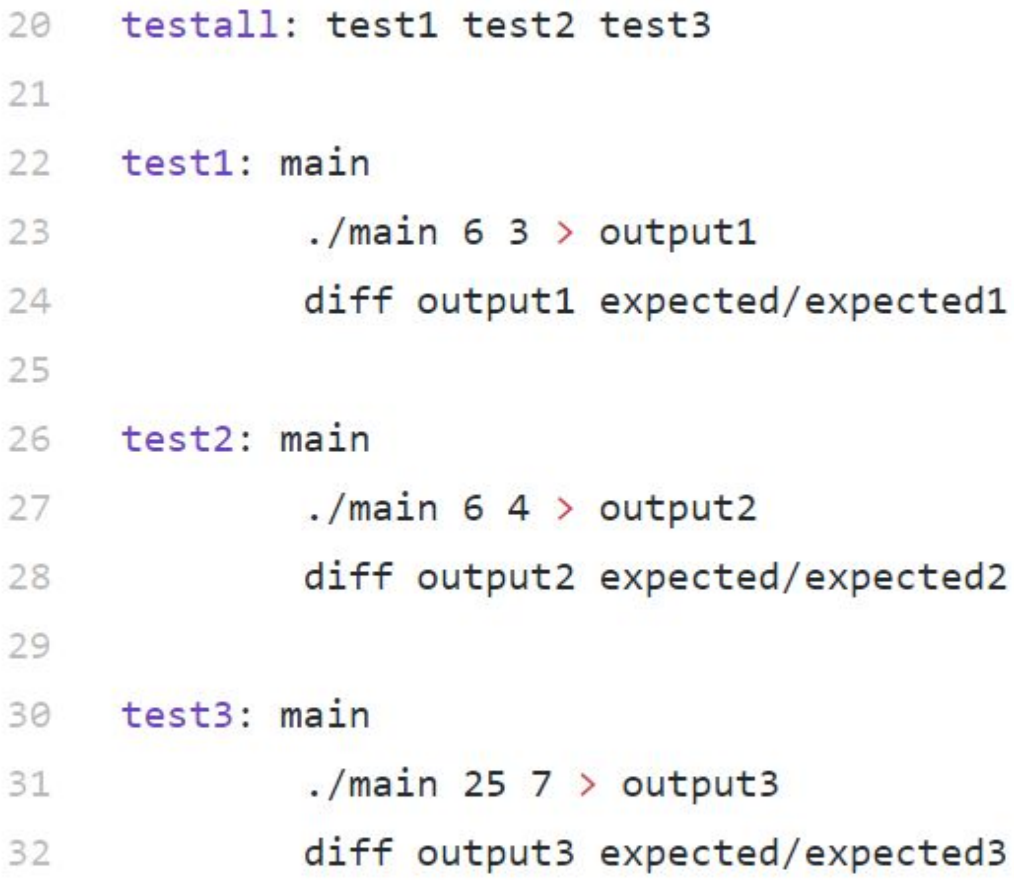
eliminate.c



```
    // initialize all elements
```

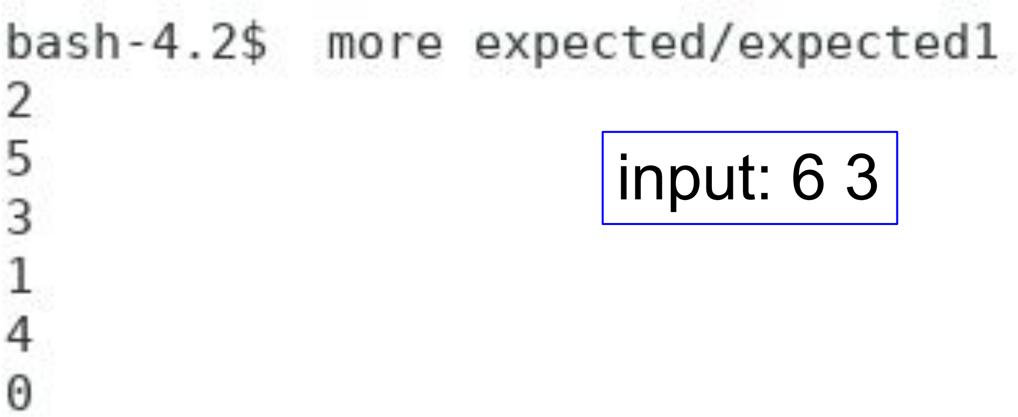
Fill your code. Use as many lines as necessary.

```
    // counting to k,
    // mark the eliminated element
    // print the index of the marked element
    // repeat until only one element is unmarked

    // print the last one

    // release the memory of the array
    free (arr);
}
```



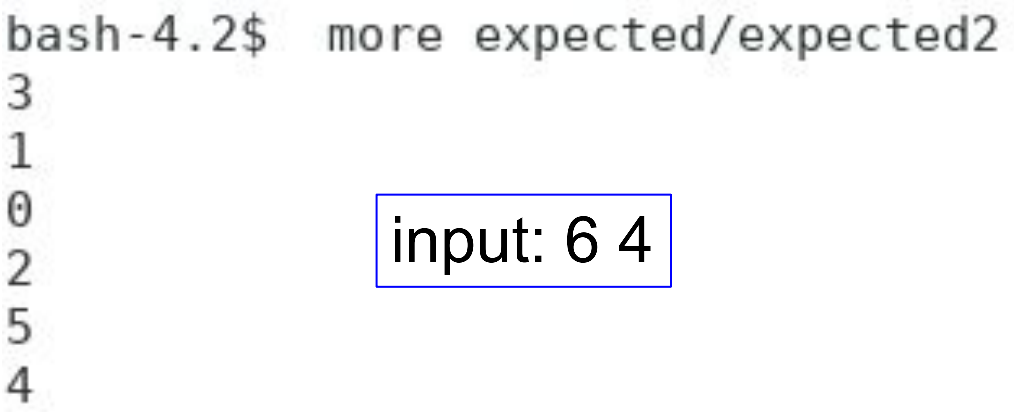
```makefile
testall: test1 test2 test3

test1: main
	./main 6 3 > output1
	diff output1 expected/expected1

test2: main
	./main 6 4 > output2
	diff output2 expected/expected2

test3: main
	./main 25 7 > output3
	diff output3 expected/expected3
```

Makefile



```
bash-4.2$  more expected/expected1
2
5
3
1
4
0
```

input: 6 3



```
bash-4.2$  more expected/expected2
3
1
0
2
5
4
```

input: 6 4

Homework 04

Count Letters

The Nobel Prize in Physics 2016 was divided, one half awarded to David J. Thouless, the other half jointly to F. Duncan M. Haldane and J. Michael Kosterlitz "for theoretical discoveries of topological phase transitions and topological phases of matter."

```
68, D, 2
70, F, 1
72, H, 1
74, J, 2
75, K, 1
77, M, 2
78, N, 1
80, P, 2
84, T, 2
97, a, 19
98, b, 1
99, c, 7
```

Count Letters

The Nobel Prize in Physics 2016 was divided, one half awarded to David J. Thouless, the other half jointly to F. Duncan **M**. Haldane and J. **M**ichael Kosterlitz "for theoretical discoveries of topological phase transitions and topological phases of matter."

68, D, 2

70, F, 1

72, H, 1

74, J, 2

75, K, 1

77, M, 2

78, N, 1

80, P, 2

84, T, 2

97, a, 19

98, b, 1

99, c, 7

Count Letters

The Nobel Prize in Physi**c**s 2016 was divided, one half awarded to David J. Thouless, the other half jointly to F. Dun**c**an M. Haldane and J. Mi**c**hael Kosterlitz "for theoreti**c**al dis**c**overies of topologi**c**al phase transitions and topologi**c**al phases of matter."

68, D, 2

70, F, 1

72, H, 1

74, J, 2

75, K, 1

77, M, 2

78, N, 1

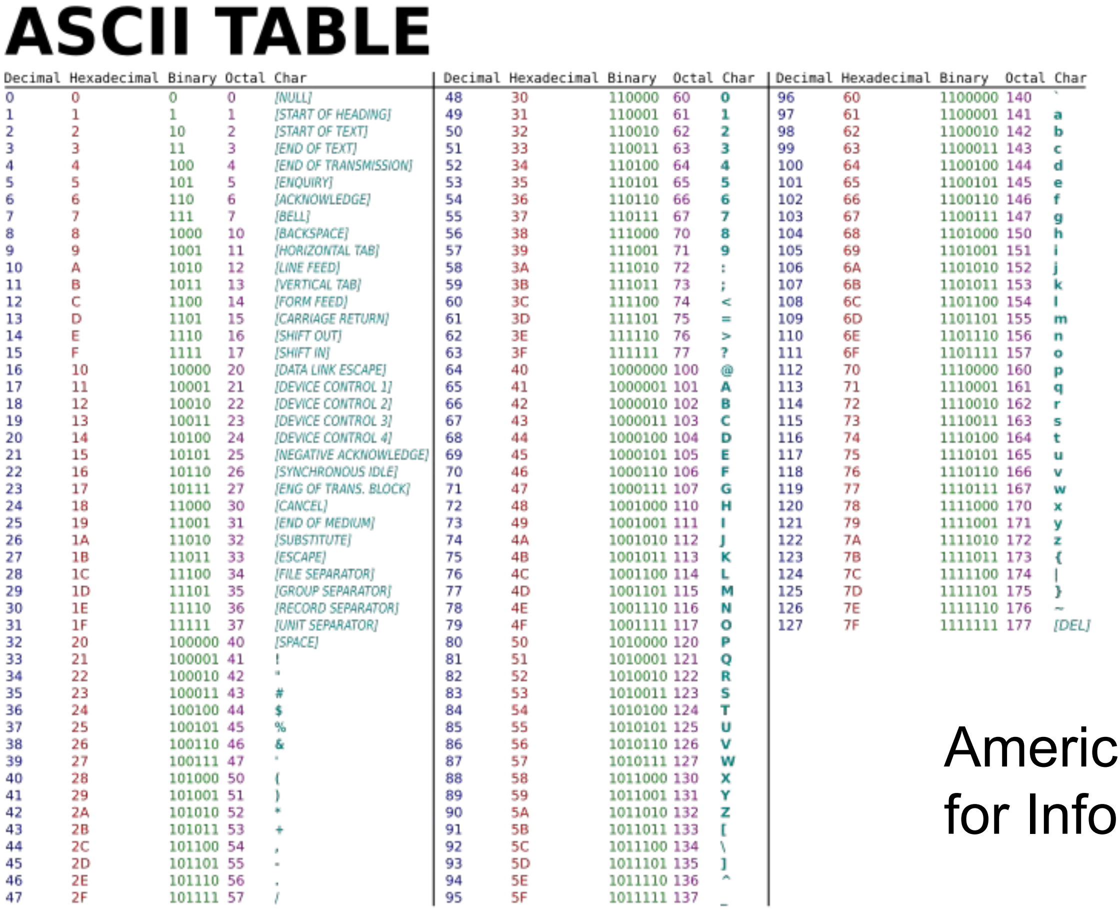
80, P, 2

84, T, 2

97, a, 19

98, b, 1

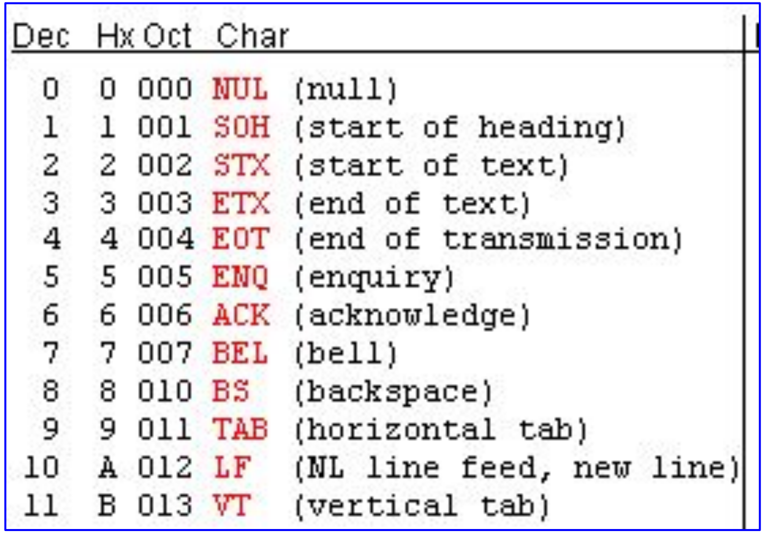
99, c, 7



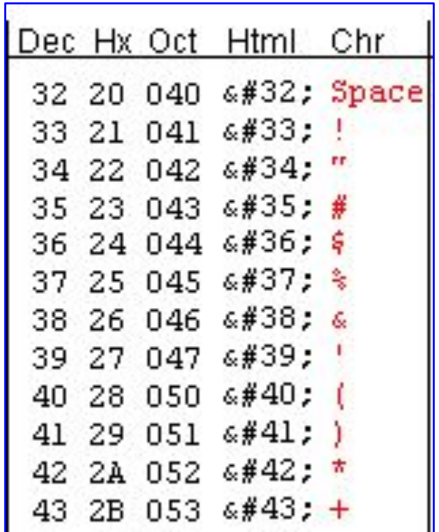
ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	110000	60	0	96	60	1100000	140	`
1	1	1	1	[START OF HEADING]	49	31	110001	61	1	97	61	1100001	141	a
2	2	10	2	[START OF TEXT]	50	32	110010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	110011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	110111	67	7	103	67	1100111	147	g
8	8	1000	10	[BACKSPACE]	56	38	111000	70	8	104	68	1101000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	111001	71	9	105	69	1101001	151	i
10	A	1010	12	[LINE FEED]	58	3A	111010	72	:	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	111011	73	;	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	111100	74	<	108	6C	1101100	154	l
13	D	1101	15	[CARRIAGE RETURN]	61	3D	111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1110000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	A	113	71	1110001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1110010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1110011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1110100	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1110101	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1110110	166	v
23	17	10111	27	[ENG OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1110111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111100	174	|
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111101	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111110	176	~
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					
33	21	100001	41	!	81	51	1010001	121	Q					
34	22	100010	42	"	82	52	1010010	122	R					
35	23	100011	43	#	83	53	1010011	123	S					
36	24	100100	44	$	84	54	1010100	124	T					
37	25	100101	45	%	85	55	1010101	125	U					
38	26	100110	46	&	86	56	1010110	126	V					
39	27	100111	47	'	87	57	1010111	127	W					
40	28	101000	50	(	88	58	1011000	130	X					
41	29	101001	51	)	89	59	1011001	131	Y					
42	2A	101010	52	*	90	5A	1011010	132	Z					
43	2B	101011	53	+	91	5B	1011011	133	[					
44	2C	101100	54	,	92	5C	1011100	134	\					
45	2D	101101	55	-	93	5D	1011101	135	]					
46	2E	101110	56	.	94	5E	1011110	136	^					
47	2F	101111	57	/	95	5F	1011111	137	_					

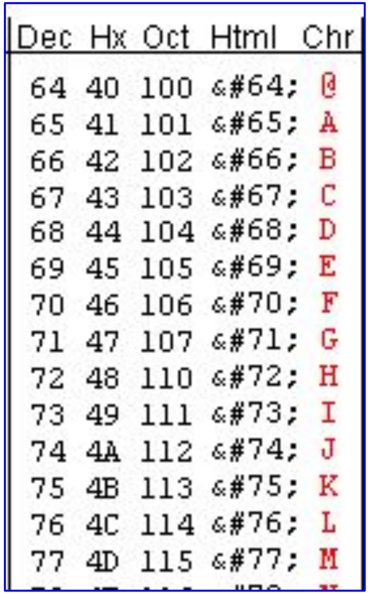
American Standard Code for Information Interchange



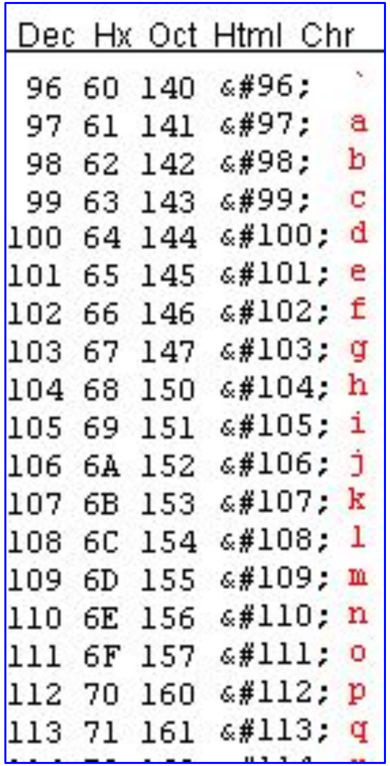
Dec	Hx	Oct	Char	
0	0	000	NUL	(null)
1	1	001	SOH	(start of heading)
2	2	002	STX	(start of text)
3	3	003	ETX	(end of text)
4	4	004	EOT	(end of transmission)
5	5	005	ENQ	(enquiry)
6	6	006	ACK	(acknowledge)
7	7	007	BEL	(bell)
8	8	010	BS	(backspace)
9	9	011	TAB	(horizontal tab)
10	A	012	LF	(NL line feed, new line)
11	B	013	VT	(vertical tab)



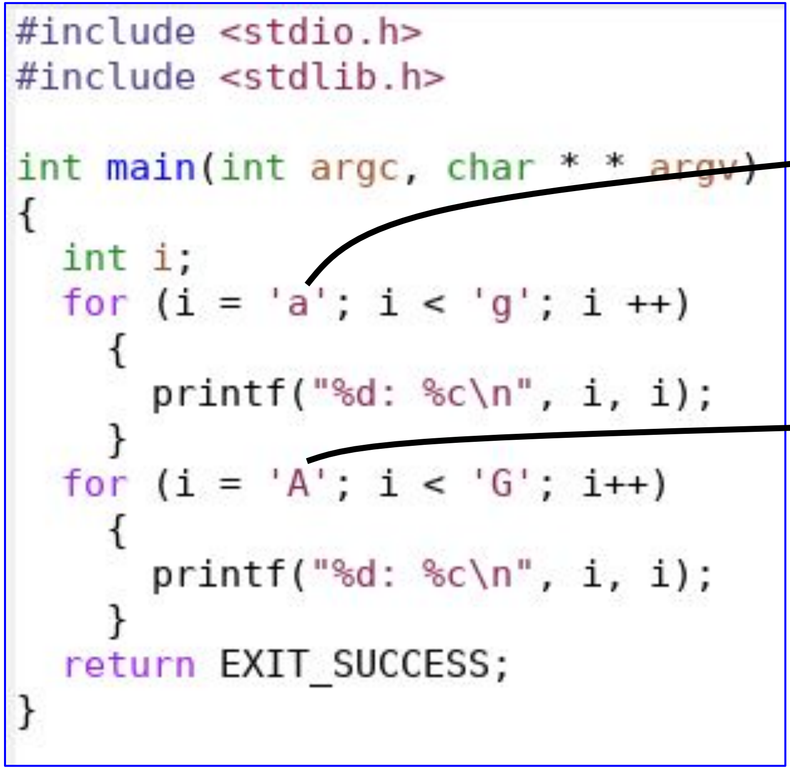
Dec	Hx	Oct	Html	Chr
32	20	040	&#32;	Space
33	21	041	&#33;	!
34	22	042	&#34;	"
35	23	043	&#35;	#
36	24	044	&#36;	$
37	25	045	&#37;	%
38	26	046	&#38;	&
39	27	047	&#39;	'
40	28	050	&#40;	(
41	29	051	&#41;	)
42	2A	052	&#42;	*
43	2B	053	&#43;	+



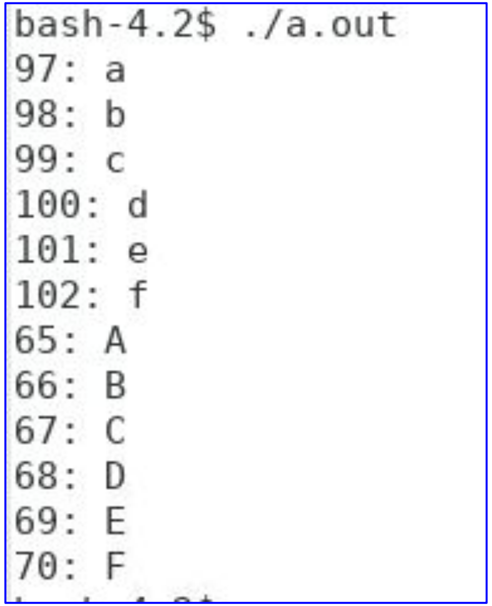
Dec	Hx	Oct	Html	Chr
64	40	100	&#64;	@
65	41	101	&#65;	A
66	42	102	&#66;	B
67	43	103	&#67;	C
68	44	104	&#68;	D
69	45	105	&#69;	E
70	46	106	&#70;	F
71	47	107	&#71;	G
72	48	110	&#72;	H
73	49	111	&#73;	I
74	4A	112	&#74;	J
75	4B	113	&#75;	K
76	4C	114	&#76;	L
77	4D	115	&#77;	M



Dec	Hx	Oct	Html	Chr
96	60	140	&#96;	`
97	61	141	&#97;	a
98	62	142	&#98;	b
99	63	143	&#99;	c
100	64	144	&#100;	d
101	65	145	&#101;	e
102	66	146	&#102;	f
103	67	147	&#103;	g
104	68	150	&#104;	h
105	69	151	&#105;	i
106	6A	152	&#106;	j
107	6B	153	&#107;	k
108	6C	154	&#108;	l
109	6D	155	&#109;	m
110	6E	156	&#110;	n
111	6F	157	&#111;	o
112	70	160	&#112;	p
113	71	161	&#113;	q



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

int main(int argc, char * * argv)
{
  int i;
  for (i = 'a'; i < 'g'; i ++)
    {
      printf("%d: %c\n", i, i);
    }
  for (i = 'A'; i < 'G'; i++)
    {
      printf("%d: %c\n", i, i);
    }
  return EXIT_SUCCESS;
}
```

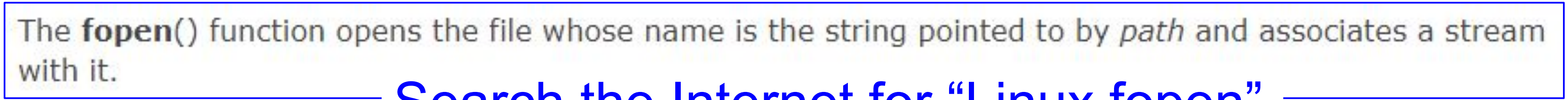


```
bash-4.2$ ./a.out
97: a
98: b
99: c
100: d
101: e
102: f
65: A
66: B
67: C
68: D
69: E
70: F
```

‘X’: a single letter, equivalent to a number (in ASCII)

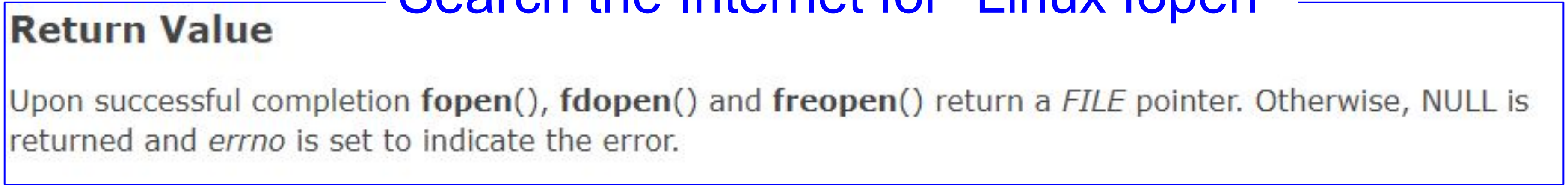
read characters from file

```
FILE * fptr = fopen(filename, “r”);
if (fptr == NULL)
{
   // fopen fail, handle error
   // Do NOT fclose
}
int ch = fgetc(fptr); // read one character
```



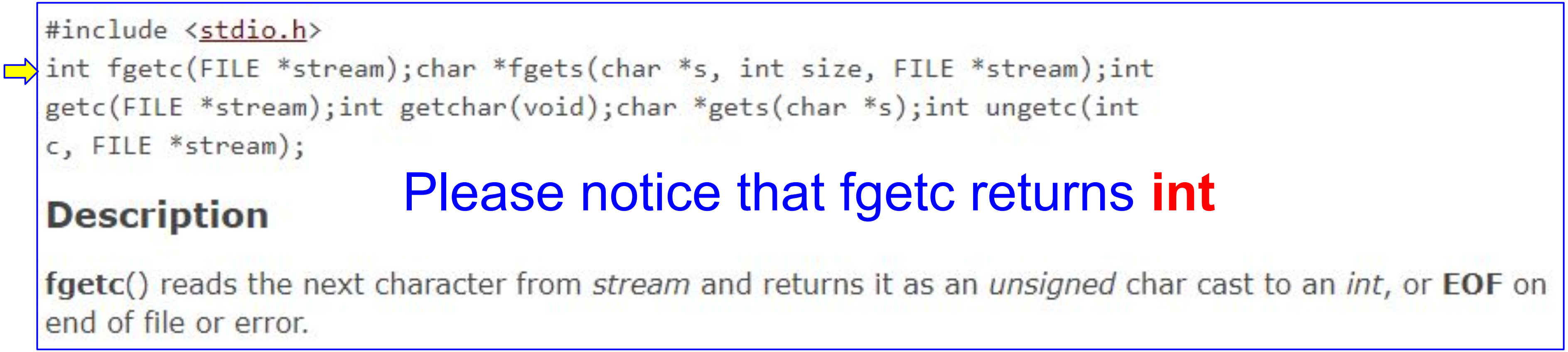
The **fopen**() function opens the file whose name is the string pointed to by *path* and associates a stream with it.

Search the Internet for “Linux fopen”



Return Value

Upon successful completion **fopen**(), **fdopen**() and **freopen**() return a *FILE* pointer. Otherwise, NULL is returned and *errno* is set to indicate the error.



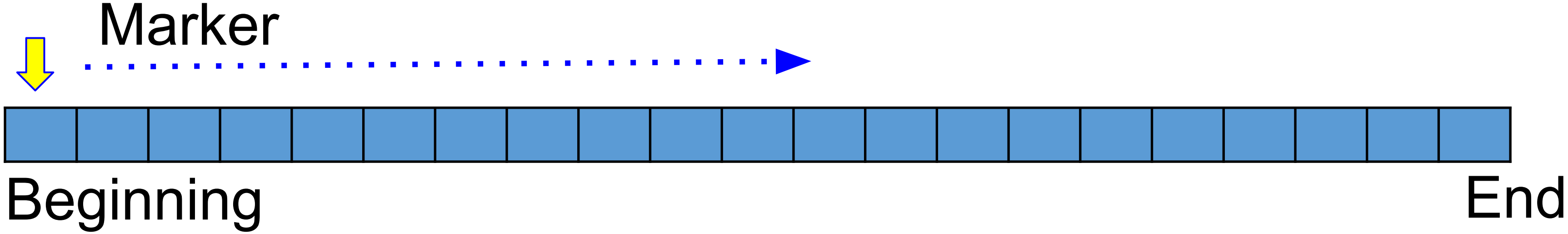
```
#include <stdio.h>
int fgetc(FILE *stream);char *fgets(char *s, int size, FILE *stream);int
getc(FILE *stream);int getchar(void);char *gets(char *s);int ungetc(int
c, FILE *stream);
```

Please notice that fgetc returns **int**

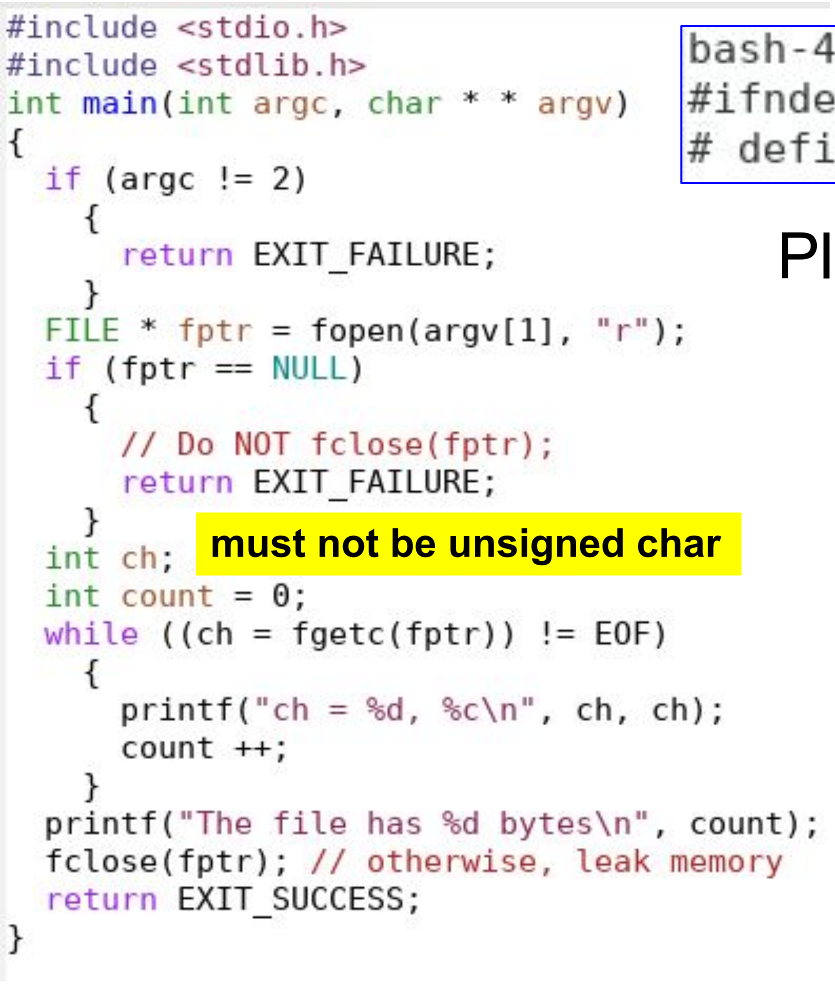
Description

fgetc() reads the next character from *stream* and returns it as an *unsigned* char cast to an *int*, or **EOF** on end of file or error.

file: “stream” in a C program

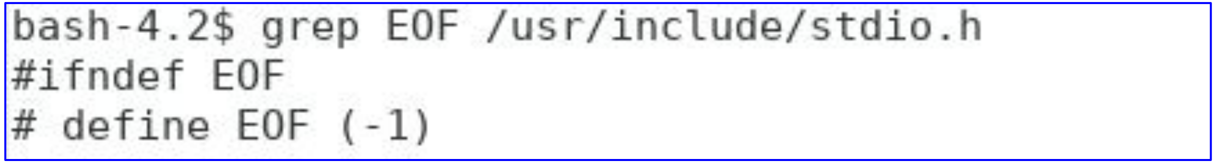


- Think of a file as a river (stream).
- A marker points to the current location.
- The marker is at the beginning after **`fopen`**.
- The marker moves toward the end after reading or writing data.
- **`ftell`** reports the current location. **`fseek`** sets the location.



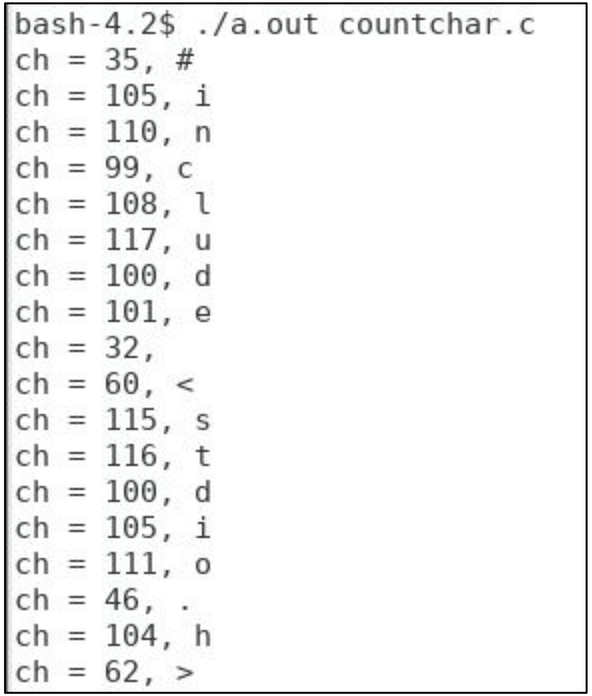
```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * * argv)
{
  if (argc != 2)
    {
      return EXIT_FAILURE;
    }
  FILE * fptr = fopen(argv[1], "r");
  if (fptr == NULL)
    {
      // Do NOT fclose(fptr);
      return EXIT_FAILURE;
    }
  int ch;
  int count = 0;
  while ((ch = fgetc(fptr)) != EOF)
    {
      printf("ch = %d, %c\n", ch, ch);
      count ++;
    }
  printf("The file has %d bytes\n", count);
  fclose(fptr); // otherwise, leak memory
  return EXIT_SUCCESS;
}
```

must not be unsigned char

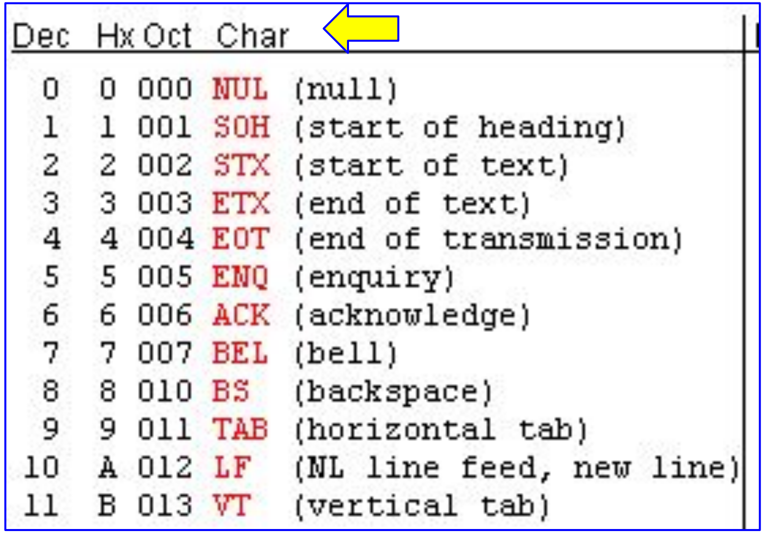


```
bash-4.2$ grep EOF /usr/include/stdio.h
#ifndef EOF
# define EOF (-1)
```

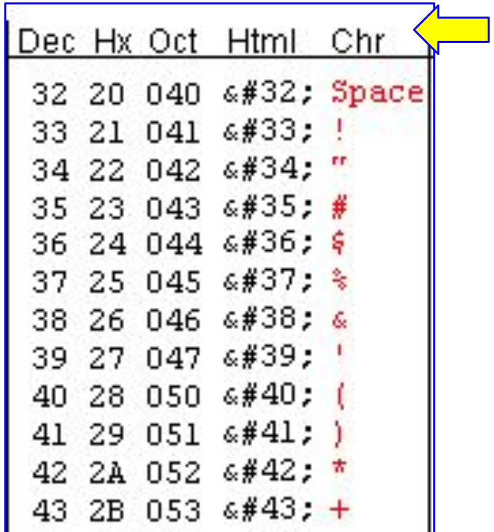
Please notice that **EOF** is -1, not 0



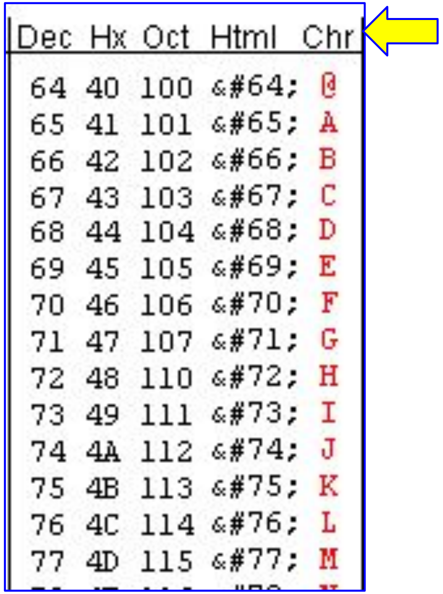
```
bash-4.2$ ./a.out countchar.c
ch = 35, #
ch = 105, i
ch = 110, n
ch = 99, c
ch = 108, l
ch = 117, u
ch = 100, d
ch = 101, e
ch = 32,  
ch = 60, <
ch = 115, s
ch = 116, t
ch = 100, d
ch = 105, i
ch = 111, o
ch = 46, .
ch = 104, h
ch = 62, >
```



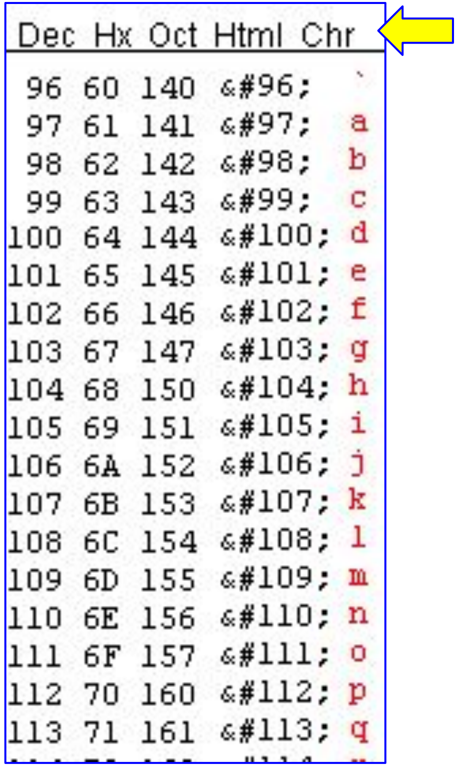
Dec	Hx	Oct	Char	
0	0	000	NUL	(null)
1	1	001	SOH	(start of heading)
2	2	002	STX	(start of text)
3	3	003	ETX	(end of text)
4	4	004	EOT	(end of transmission)
5	5	005	ENQ	(enquiry)
6	6	006	ACK	(acknowledge)
7	7	007	BEL	(bell)
8	8	010	BS	(backspace)
9	9	011	TAB	(horizontal tab)
10	A	012	LF	(NL line feed, new line)
11	B	013	VT	(vertical tab)



Dec	Hx	Oct	Html	Chr
32	20	040	&#32;	Space
33	21	041	&#33;	!
34	22	042	&#34;	"
35	23	043	&#35;	#
36	24	044	&#36;	$
37	25	045	&#37;	%
38	26	046	&#38;	&
39	27	047	&#39;	'
40	28	050	&#40;	(
41	29	051	&#41;	)
42	2A	052	&#42;	*
43	2B	053	&#43;	+



Dec	Hx	Oct	Html	Chr
64	40	100	&#64;	@
65	41	101	&#65;	A
66	42	102	&#66;	B
67	43	103	&#67;	C
68	44	104	&#68;	D
69	45	105	&#69;	E
70	46	106	&#70;	F
71	47	107	&#71;	G
72	48	110	&#72;	H
73	49	111	&#73;	I
74	4A	112	&#74;	J
75	4B	113	&#75;	K
76	4C	114	&#76;	L
77	4D	115	&#77;	M

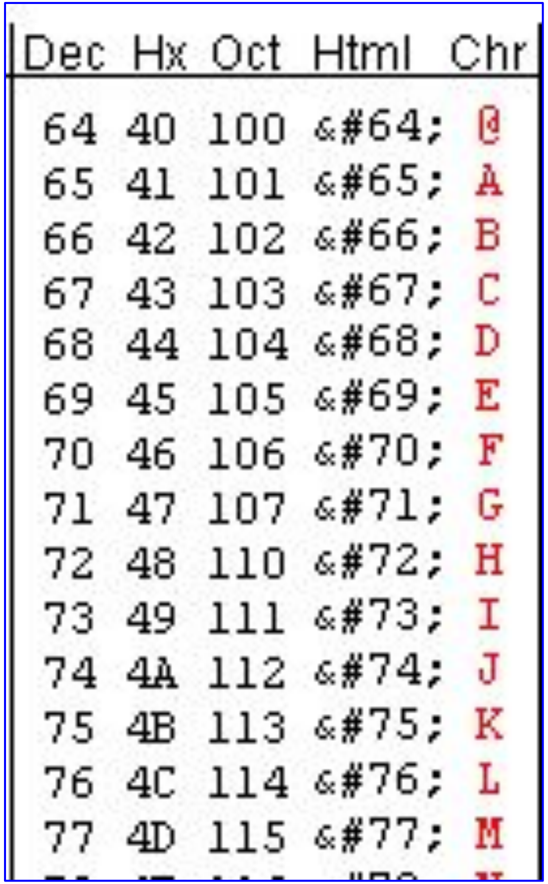


Dec	Hx	Oct	Html	Chr
96	60	140	&#96;	`
97	61	141	&#97;	a
98	62	142	&#98;	b
99	63	143	&#99;	c
100	64	144	&#100;	d
101	65	145	&#101;	e
102	66	146	&#102;	f
103	67	147	&#103;	g
104	68	150	&#104;	h
105	69	151	&#105;	i
106	6A	152	&#106;	j
107	6B	153	&#107;	k
108	6C	154	&#108;	l
109	6D	155	&#109;	m
110	6E	156	&#110;	n
111	6F	157	&#111;	o
112	70	160	&#112;	p
113	71	161	&#113;	q

Number Systems

- Decimal: base 10, 10 digits ⇒ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- Binary: base 2, 2 digits ⇒ 0, 1
- Hexadecimal: base 16, 16 digits ⇒ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
- Octal: base 8, 8 digits ⇒ 0, 1, 2, 3, 4, 5, 6, 7
- $1234_{(10)} = 1 \text{ x } 10^3 + 2 \text{ x } 10^2 + 3 \text{ x } 10^1 + 4 \text{ x } 10^0$
- $1011_{(2)} = 1 \text{ x } 2^3 + 0 \text{ x } 2^2 + 1 \text{ x } 2^1 + 1 \text{ x } 2^0$
- $B9C6_{(16)} = 11 \text{ x } 16^3 + 9 \text{ x } 16^2 + 12 \text{ x } 16^1 + 6 \text{ x } 16^0$

- $512.34_{(10)} = 5 \text{ x } 10^{2} + 1 \text{ x } 10^{1} + 2 \text{ x } 10^{0} + 3 \text{ x } 10^{-1} + 4 \text{ x } 10^{-2}$
- $110.11_{(2)} = 1 \text{ x } 2^{2} + 1 \text{ x } 2^{1} + 0 \text{ x } 2^{0} + 1 \text{ x } 2^{-1} + 1 \text{ x } 2^{-2}$
- $7B9.C6_{(16)} = 7 \text{ x } 16^{2} + 11 \text{ x } 16^{1} + 9 \text{ x } 16^{0} + 12 \text{ x } 16^{-1} + 6 \text{ x } 16^{-2}$
- $534_{(10)} = 512 + 22 = 2 \text{ x } 16^{2} = 1 \text{ x } 16^{1} + 6 \text{ x } 16^{0} = 216_{(16)}$
- $16_{(10)} = 2^{4} = 10000_{(2)}$
- $D_{(16)} = 13_{(10)} = 1 \text{ x } 2^{3} + 1 \text{ x } 2^{2} + 0 \text{ x } 2^{1} + 1 \text{ x } 2^{0} = 1101_{(2)}$



Dec	Hx	Oct	Html	Chr
64	40	100	&#64;	@
65	41	101	&#65;	A
66	42	102	&#66;	B
67	43	103	&#67;	C
68	44	104	&#68;	D
69	45	105	&#69;	E
70	46	106	&#70;	F
71	47	107	&#71;	G
72	48	110	&#72;	H
73	49	111	&#73;	I
74	4A	112	&#74;	J
75	4B	113	&#75;	K
76	4C	114	&#76;	L
77	4D	115	&#77;	M

$$64_{(10)} = 40_{(16)} = 100_{(8)}$$

$$71_{(10)} = 64_{(10)} + 7_{(10)} = 40_{(16)} + 7_{(16)} = 47_{(16)}$$

Homework 05

Add Numbers

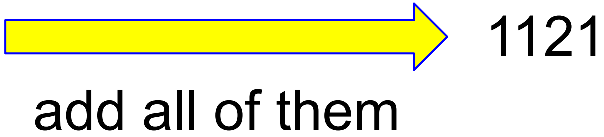
-598

322

202

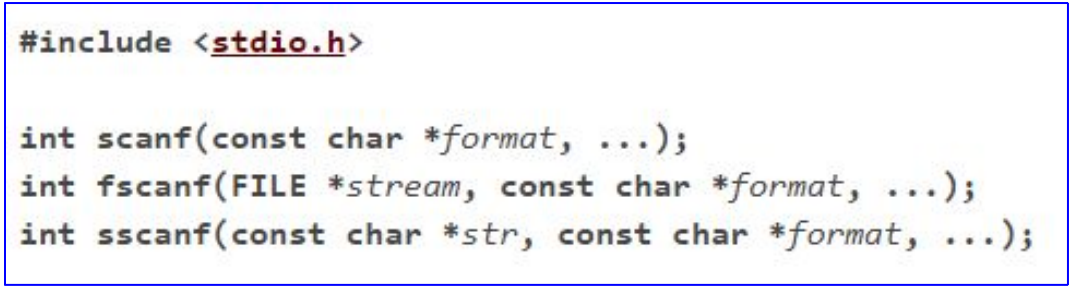
517

678

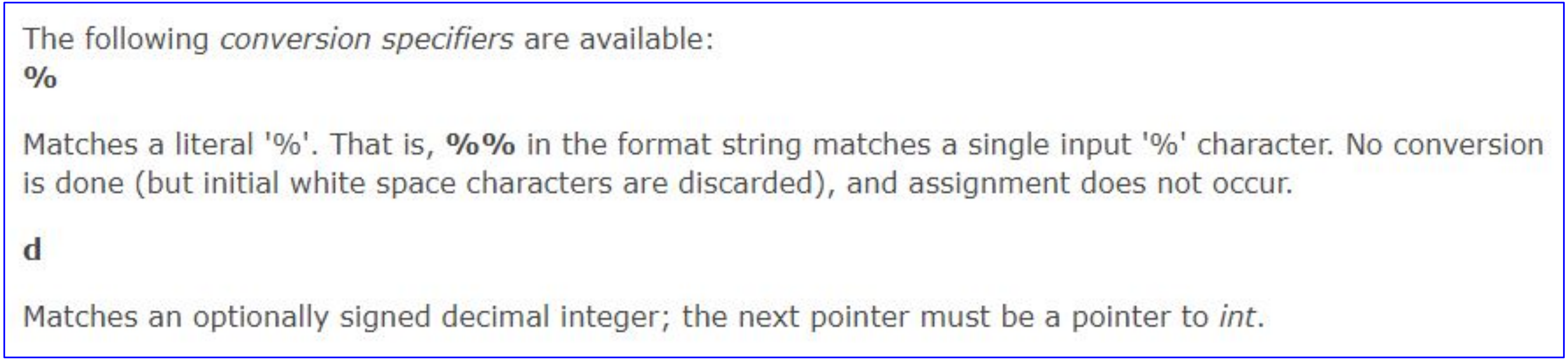


1121

add all of them



```
#include <stdio.h>

int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *str, const char *format, ...);
```



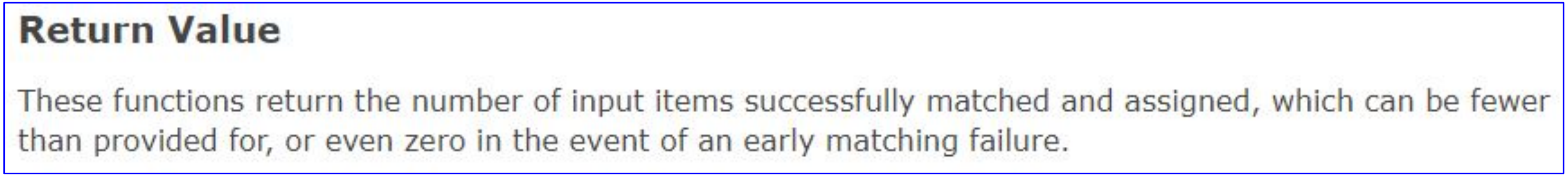
The following *conversion specifiers* are available:

%

Matches a literal '%'. That is, **%%** in the format string matches a single input '%' character. No conversion is done (but initial white space characters are discarded), and assignment does not occur.

d

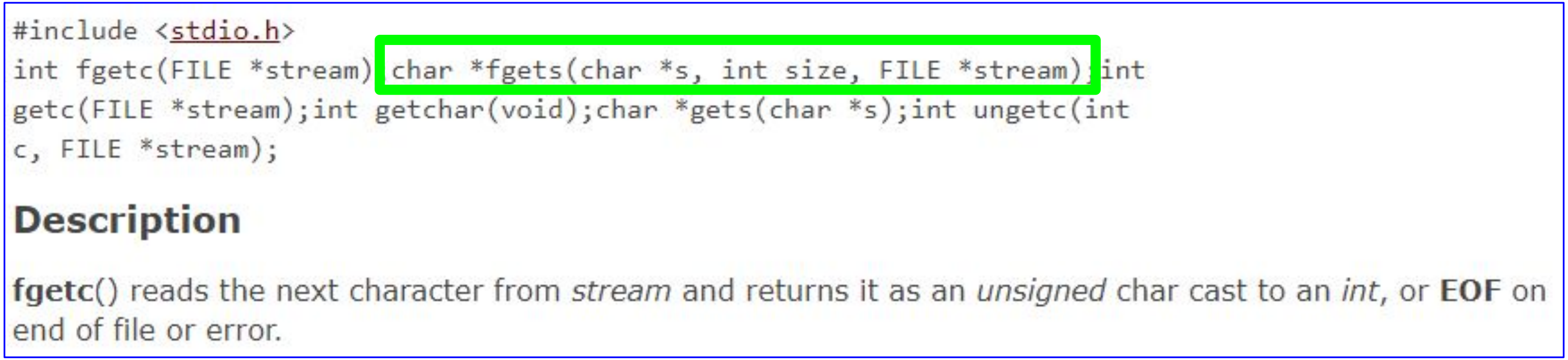
Matches an optionally signed decimal integer; the next pointer must be a pointer to *int*.



Return Value

These functions return the number of input items successfully matched and assigned, which can be fewer than provided for, or even zero in the event of an early matching failure.

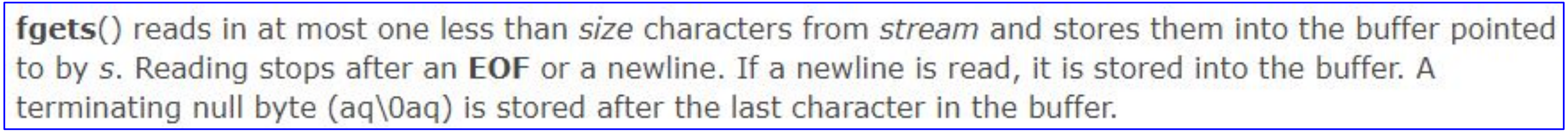
Different ways reading from a file



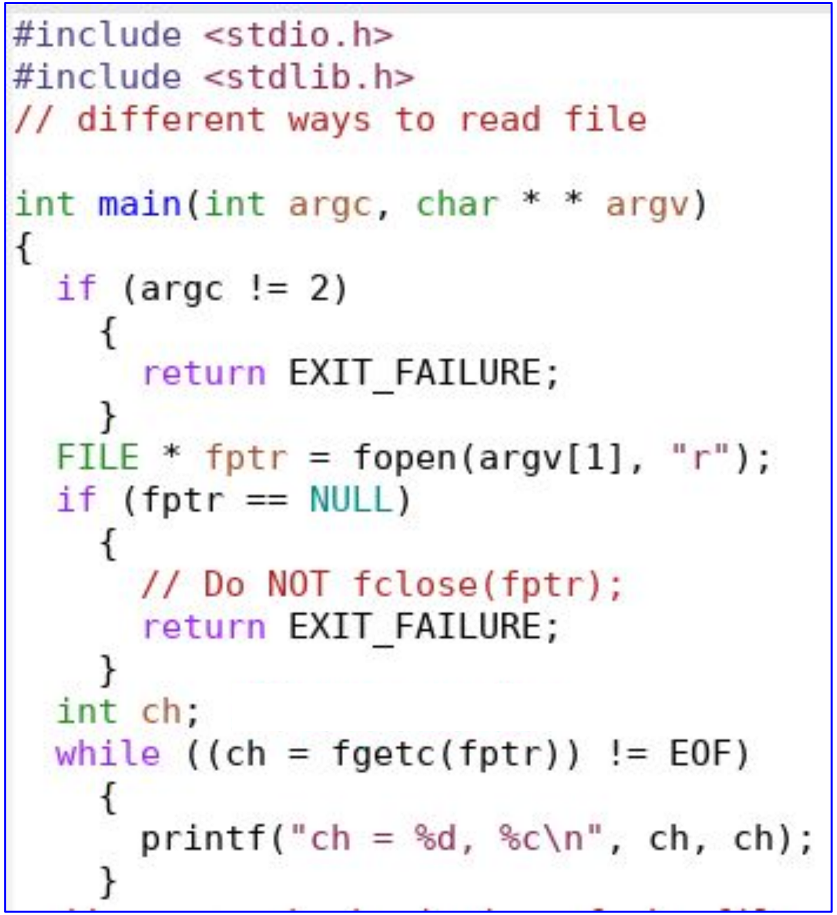
```
#include <stdio.h>
int fgetc(FILE *stream);char *fgets(char *s, int size, FILE *stream);int
getc(FILE *stream);int getchar(void);char *gets(char *s);int ungetc(int
c, FILE *stream);
```

Description

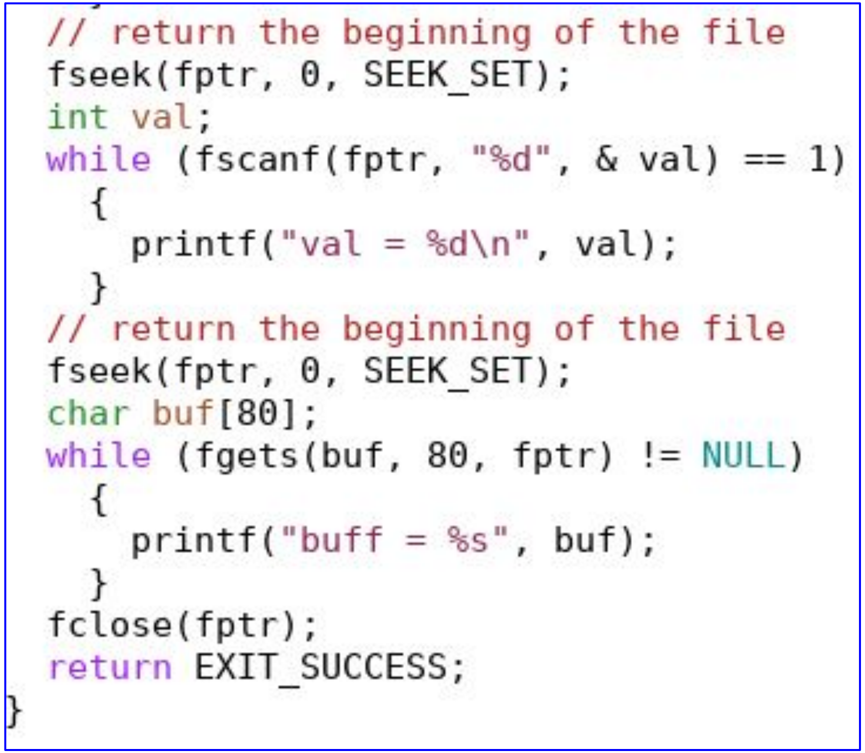
fgetc() reads the next character from *stream* and returns it as an *unsigned* char cast to an *int*, or **EOF** on end of file or error.



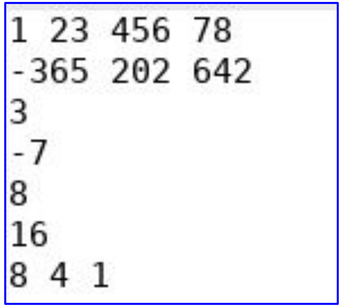
fgets() reads in at most one less than *size* characters from *stream* and stores them into the buffer pointed to by *s*. Reading stops after an **EOF** or a newline. If a newline is read, it is stored into the buffer. A terminating null byte (aq\0aq) is stored after the last character in the buffer.



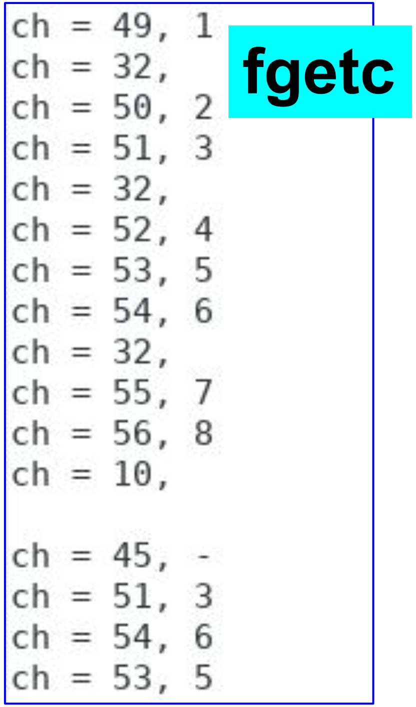
```c
#include <stdio.h>
#include <stdlib.h>
// different ways to read file

int main(int argc, char * * argv)
{
  if (argc != 2)
    {
      return EXIT_FAILURE;
    }
  FILE * fptr = fopen(argv[1], "r");
  if (fptr == NULL)
    {
      // Do NOT fclose(fptr);
      return EXIT_FAILURE;
    }
  int ch;
  while ((ch = fgetc(fptr)) != EOF)
    {
      printf("ch = %d, %c\n", ch, ch);
    }
```



```c
  // return the beginning of the file
  fseek(fptr, 0, SEEK_SET);
  int val;
  while (fscanf(fptr, "%d", & val) == 1)
    {
      printf("val = %d\n", val);
    }
  // return the beginning of the file
  fseek(fptr, 0, SEEK_SET);
  char buf[80];
  while (fgets(buf, 80, fptr) != NULL)
    {
      printf("buff = %s", buf);
    }
  fclose(fptr);
  return EXIT_SUCCESS;
}
```



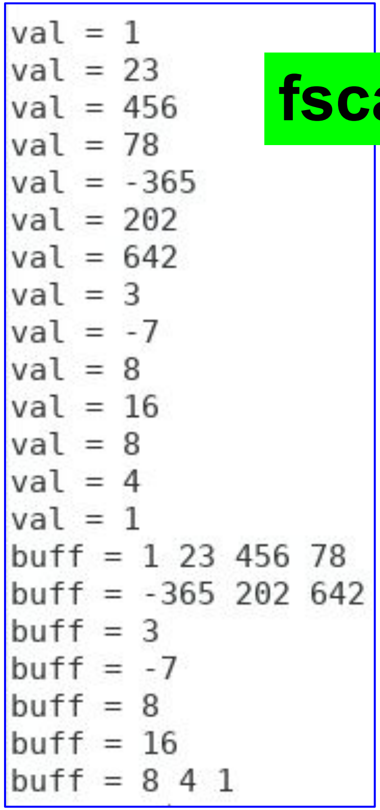
```
1 23 456 78
-365 202 642
3
-7
8
16
8 4 1
```



fgetc

```
ch = 49, 1
ch = 32,  
ch = 50, 2
ch = 51, 3
ch = 32,  
ch = 52, 4
ch = 53, 5
ch = 54, 6
ch = 32,  
ch = 55, 7
ch = 56, 8
ch = 10, 

ch = 45, -
ch = 51, 3
ch = 54, 6
ch = 53, 5
```

⋮

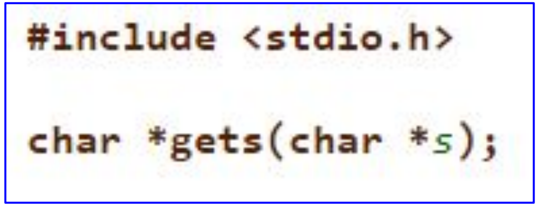


fscanf

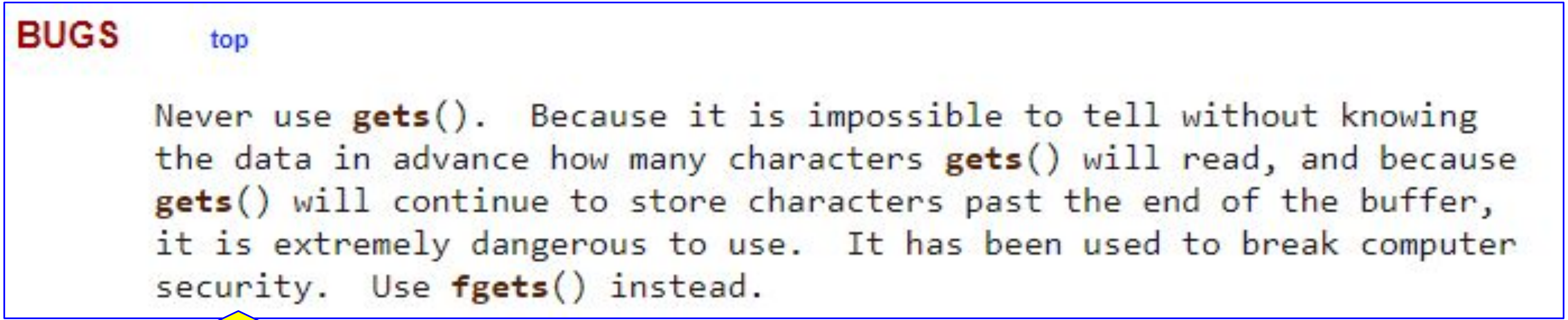
```
val = 1
val = 23
val = 456
val = 78
val = -365
val = 202
val = 642
val = 3
val = -7
val = 8
val = 16
val = 8
val = 4
val = 1
```

fgets

```
buff = 1 23 456 78
buff = -365 202 642
buff = 3
buff = -7
buff = 8
buff = 16
buff = 8 4 1
```



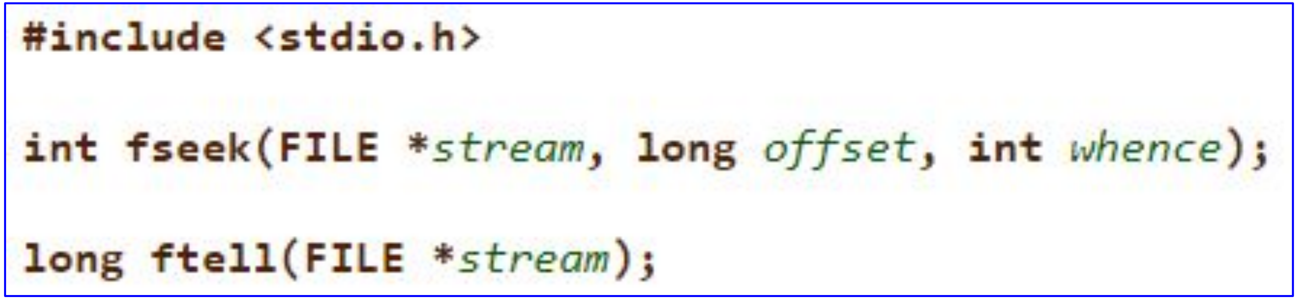
```
#include <stdio.h>

char *gets(char *s);
```



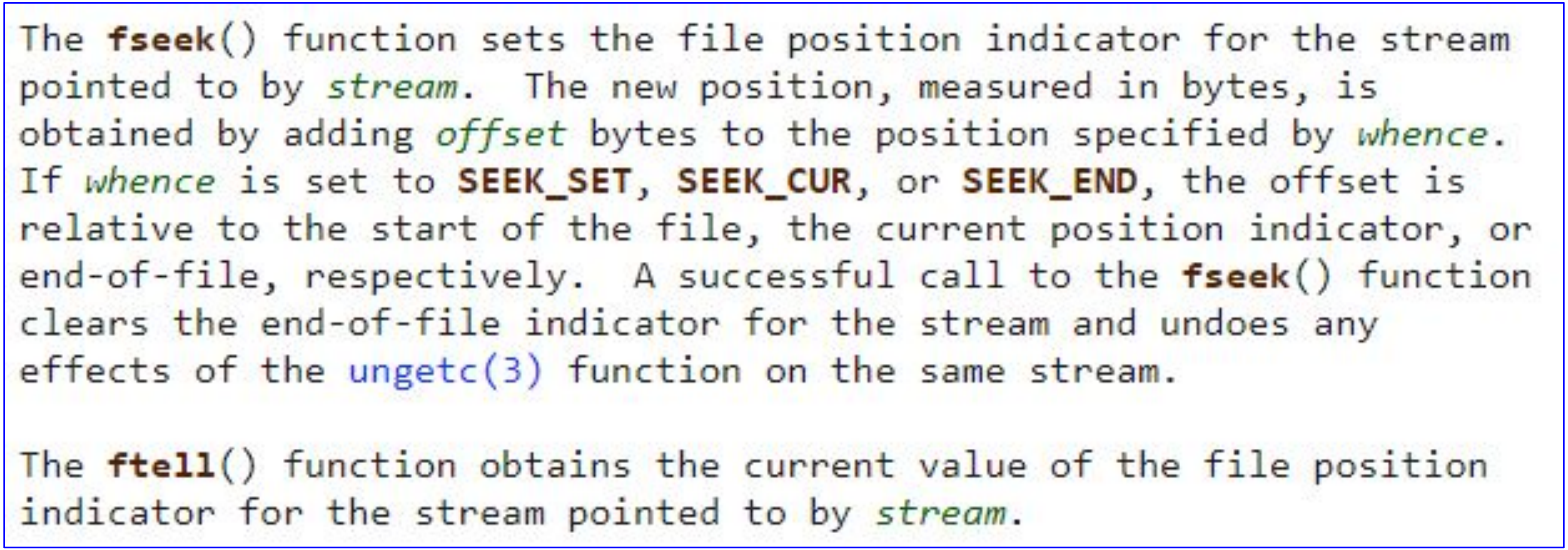
BUGS

top

Never use **gets**(). Because it is impossible to tell without knowing the data in advance how many characters **gets**() will read, and because **gets**() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use **fgets**() instead.

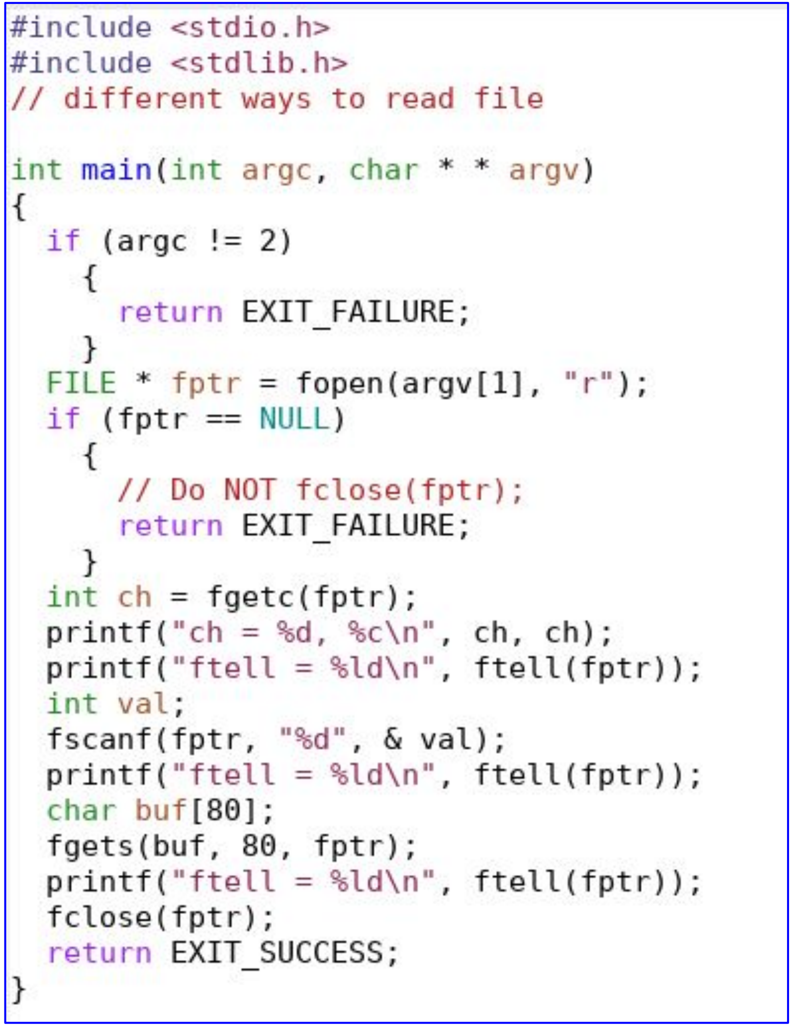


```c
#include <stdio.h>

int fseek(FILE *stream, long offset, int whence);

long ftell(FILE *stream);
```

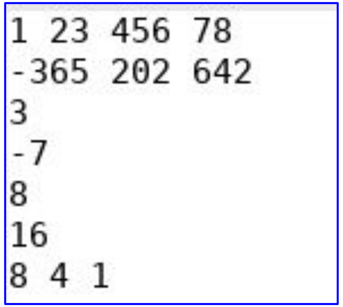


The **fseek**() function sets the file position indicator for the stream pointed to by *stream*. The new position, measured in bytes, is obtained by adding *offset* bytes to the position specified by *whence*. If *whence* is set to **SEEK_SET**, **SEEK_CUR**, or **SEEK_END**, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively. A successful call to the **fseek**() function clears the end-of-file indicator for the stream and undoes any effects of the ungetc(3) function on the same stream.

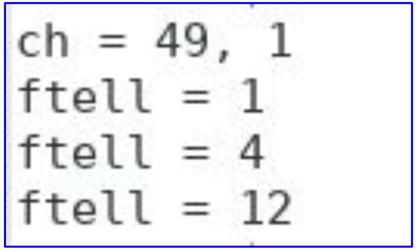
The **ftell**() function obtains the current value of the file position indicator for the stream pointed to by *stream*.



```
#include <stdio.h>
#include <stdlib.h>
// different ways to read file

int main(int argc, char * * argv)
{
  if (argc != 2)
    {
      return EXIT_FAILURE;
    }
  FILE * fptr = fopen(argv[1], "r");
  if (fptr == NULL)
    {
      // Do NOT fclose(fptr);
      return EXIT_FAILURE;
    }
  int ch = fgetc(fptr);
  printf("ch = %d, %c\n", ch, ch);
  printf("ftell = %ld\n", ftell(fptr));
  int val;
  fscanf(fptr, "%d", & val);
  printf("ftell = %ld\n", ftell(fptr));
  char buf[80];
  fgets(buf, 80, fptr);
  printf("ftell = %ld\n", ftell(fptr));
  fclose(fptr);
  return EXIT_SUCCESS;
}
```



```
1 23 456 78
-365 202 642
3
-7
8
16
8 4 1
```



```
ch = 49, 1
ftell = 1
ftell = 4
ftell = 12
```

write to a file

```
FILE * fptr = fopen(filename, “w”);
if (fptr == NULL)
{
   // fopen fail, handle error
   // Do NOT fclose
}
fprintf(fptr, “%d\n”, 264);
%c: character, %s: string, %f: floating-point
```