Computer Programming II

Ming-Feng Tsai (Victor Tsai)

Dept. of Computer Science
National Chengchi University

C Preprocessors

include files

include files

- **`#include`**

include files

- **#include**
  - allows the program to use source code from another file

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

    tells the preprocessor to take the file stdio.h (Standard I/O) and insert it in the program

include files

include files

- **`#include <stdio.h>`**

include files

- **`#include <stdio.h>`**
  - most **`#include`** directives come at the head of the program

include files

- **`#include <stdio.h>`**
  - most **`#include`** directives come at the head of the program
  - the angle brackets (<>) indicate that the file is a standard header file. On UNIX, these files are located in **`/usr/include`**

include files

include files

- Local include files

include files

- Local include files
  - can be specified by using double quotes (“”) around the file name, for example:

    #include “defs.h”
    or ***#include “../../data.h”***
    or ***#include “/root/include/const.h”***

include files

include files

- Notice that avoid defining a constant, a data structure, or union twice, for example:

  ```
  #include “data.h”
  #include “io.h”
  ```

  if they have common constant definitions, it may generate errors

include files

- Notice that avoid defining a constant, a data structure, or union twice, for example:

  ```
  #include “data.h”
  #include “io.h”
  ```

  if they have common constant definitions, it may generate errors

- Use **`#ifndef`** to avoid the problem

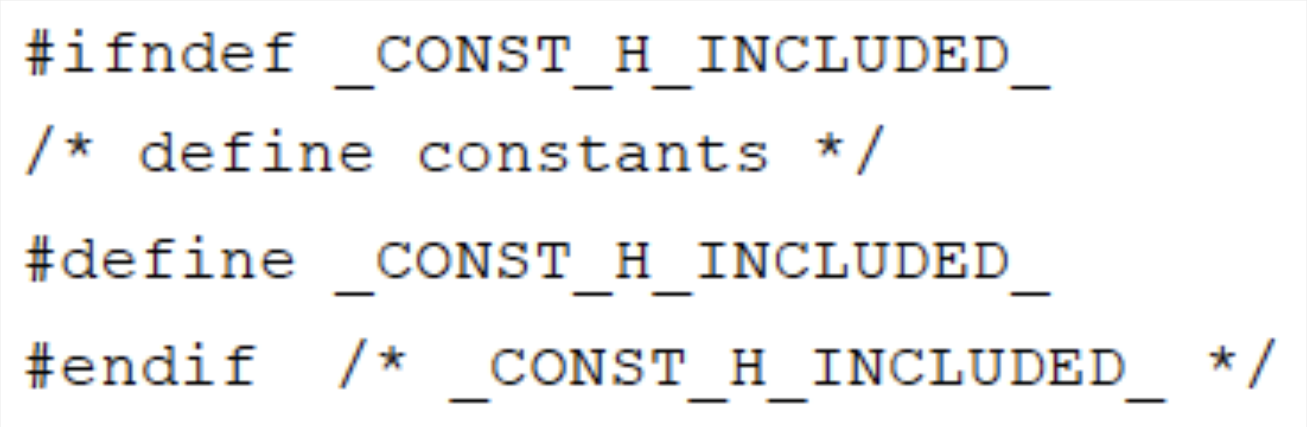
include files

- Notice that avoid defining a constant, a data structure, or union twice, for example:

  ```
  #include “data.h”
  #include “io.h”
  ```

  if they have common constant definitions, it may generate errors

- Use **`#ifndef`** to avoid the problem



```
#ifndef _CONST_H_INCLUDED_
/* define constants */
#define _CONST_H_INCLUDED_
#endif  /* _CONST_H_INCLUDED_ */
```

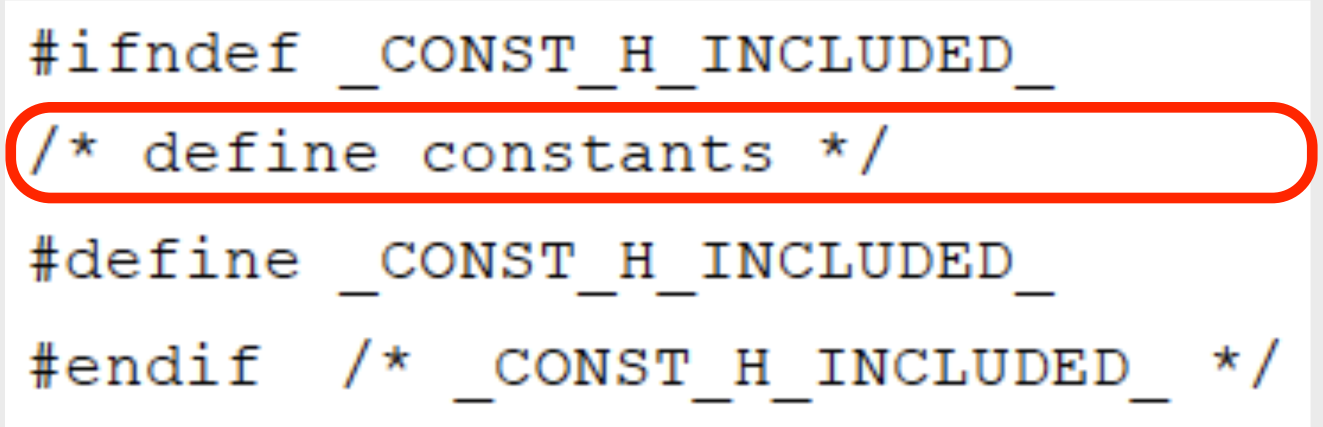
include files

- Notice that avoid defining a constant, a data structure, or union twice, for example:

  #include “data.h”
  #include “io.h”

  if they have common constant definitions, it may generate errors

- Use **#ifndef** to avoid the problem



```
#ifndef _CONST_H_INCLUDED_
/* define constants */
#define _CONST_H_INCLUDED_
#endif  /* _CONST_H_INCLUDED_ */
```

Parameterized Macros

- **`#define`** with parameters, for example:

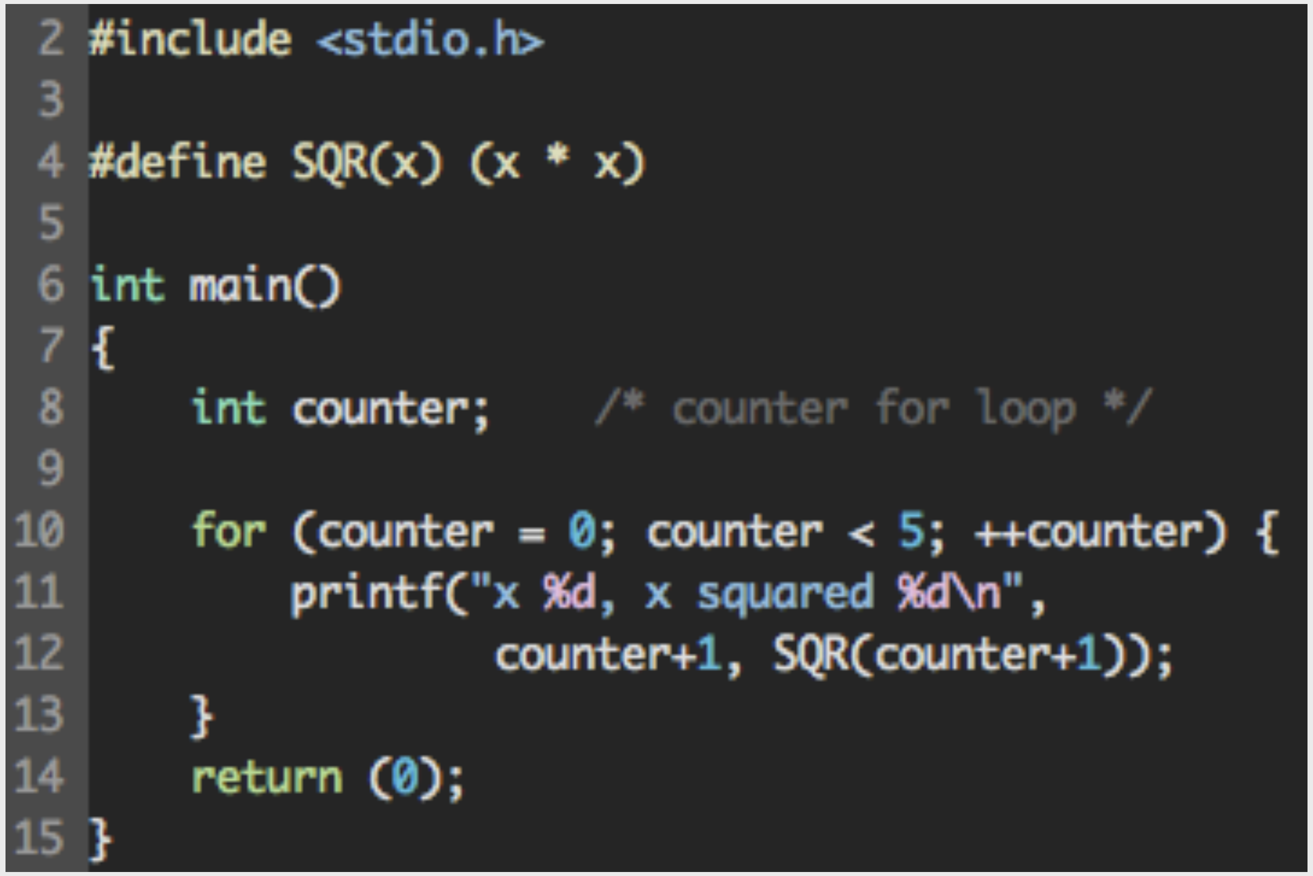
```
#define SQR(x)  ((x) * (x))
/* Square a number */
```

- the macro will replace x by the following

```
SQR(5)   expands to  ((5) * (5))
```

Parameterized Macros

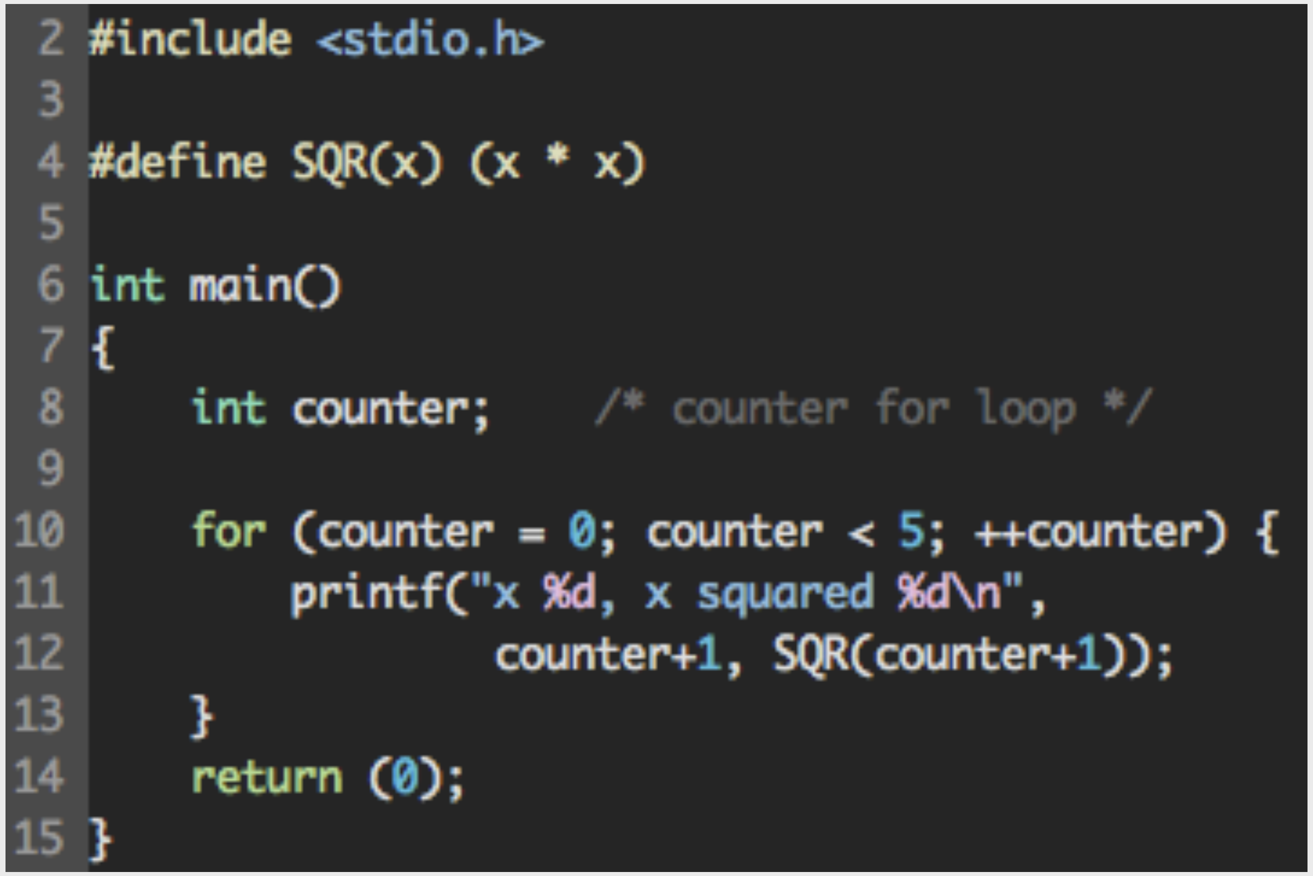
- Example: sqr.c



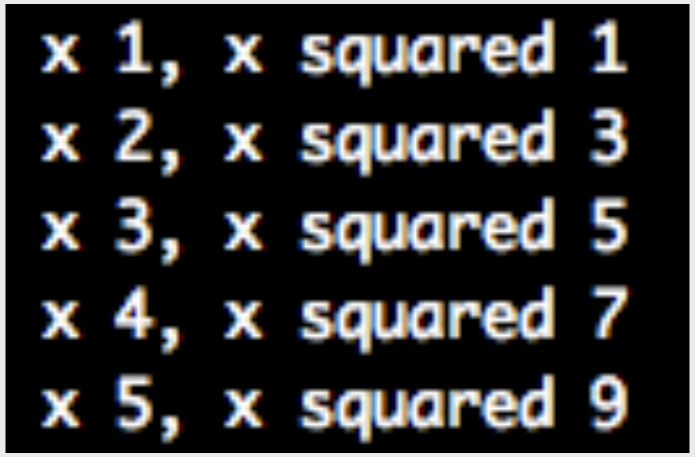
```c
#include <stdio.h>

#define SQR(x) (x * x)

int main()
{
    int counter;    /* counter for loop */

    for (counter = 0; counter < 5; ++counter) {
        printf("x %d, x squared %d\n",
                counter+1, SQR(counter+1));
    }
    return (0);
}
```

Parameterized Macros

- Example: sqr.c



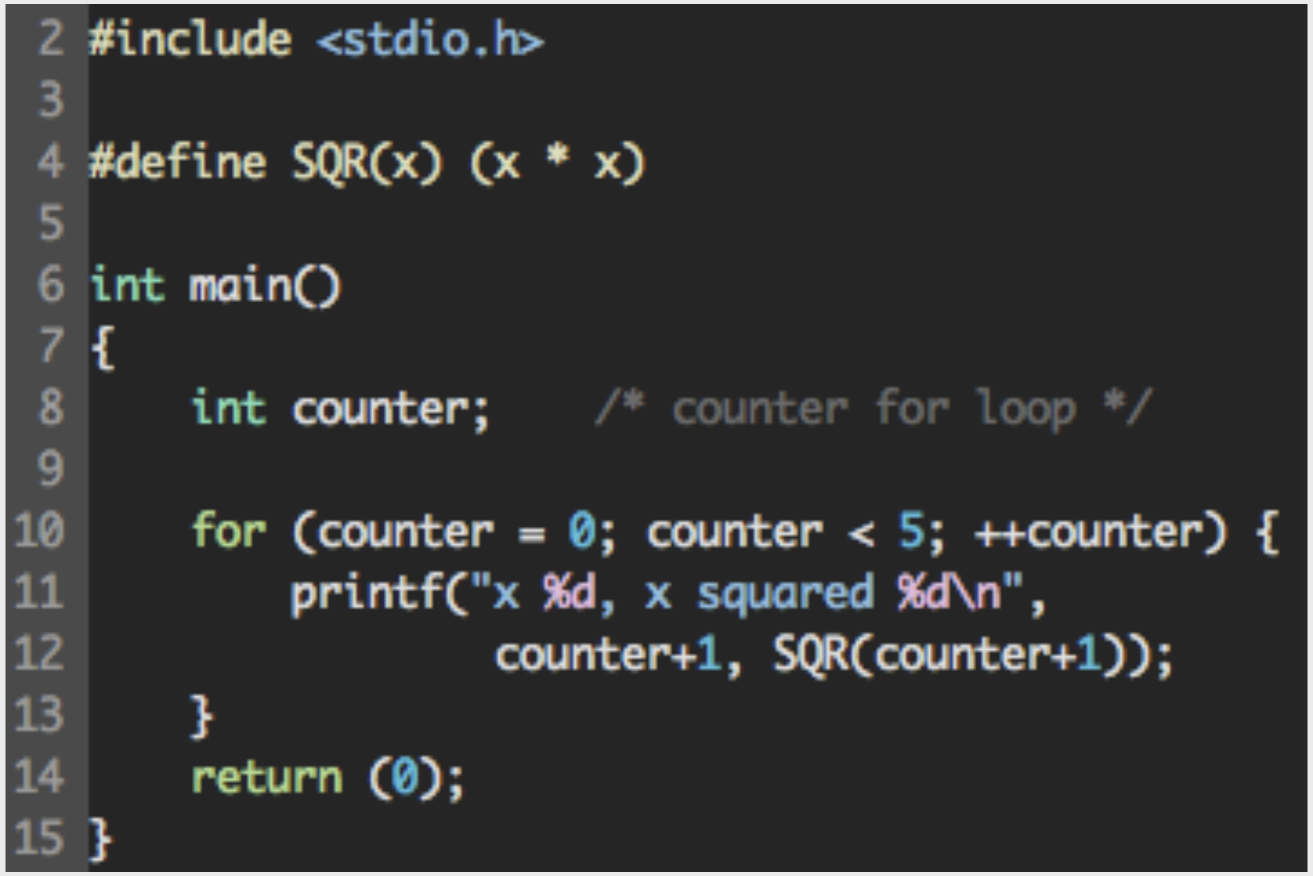
```c
#include <stdio.h>

#define SQR(x) (x * x)

int main()
{
    int counter;    /* counter for loop */

    for (counter = 0; counter < 5; ++counter) {
        printf("x %d, x squared %d\n",
                counter+1, SQR(counter+1));
    }
    return (0);
}
```



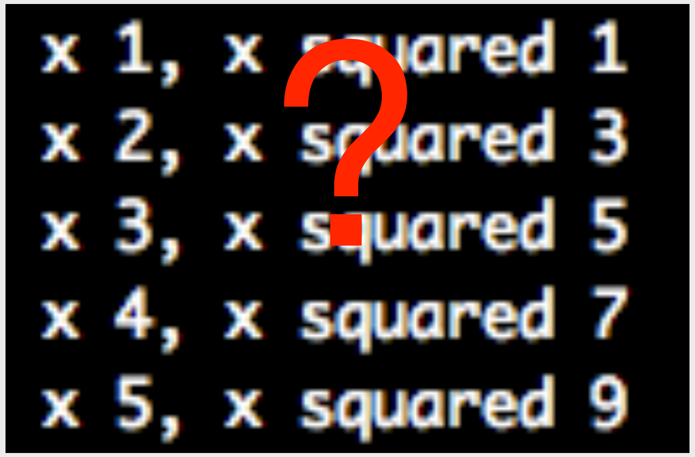
```
x 1, x squared 1
x 2, x squared 3
x 3, x squared 5
x 4, x squared 7
x 5, x squared 9
```

Parameterized Macros

- Example: sqr.c



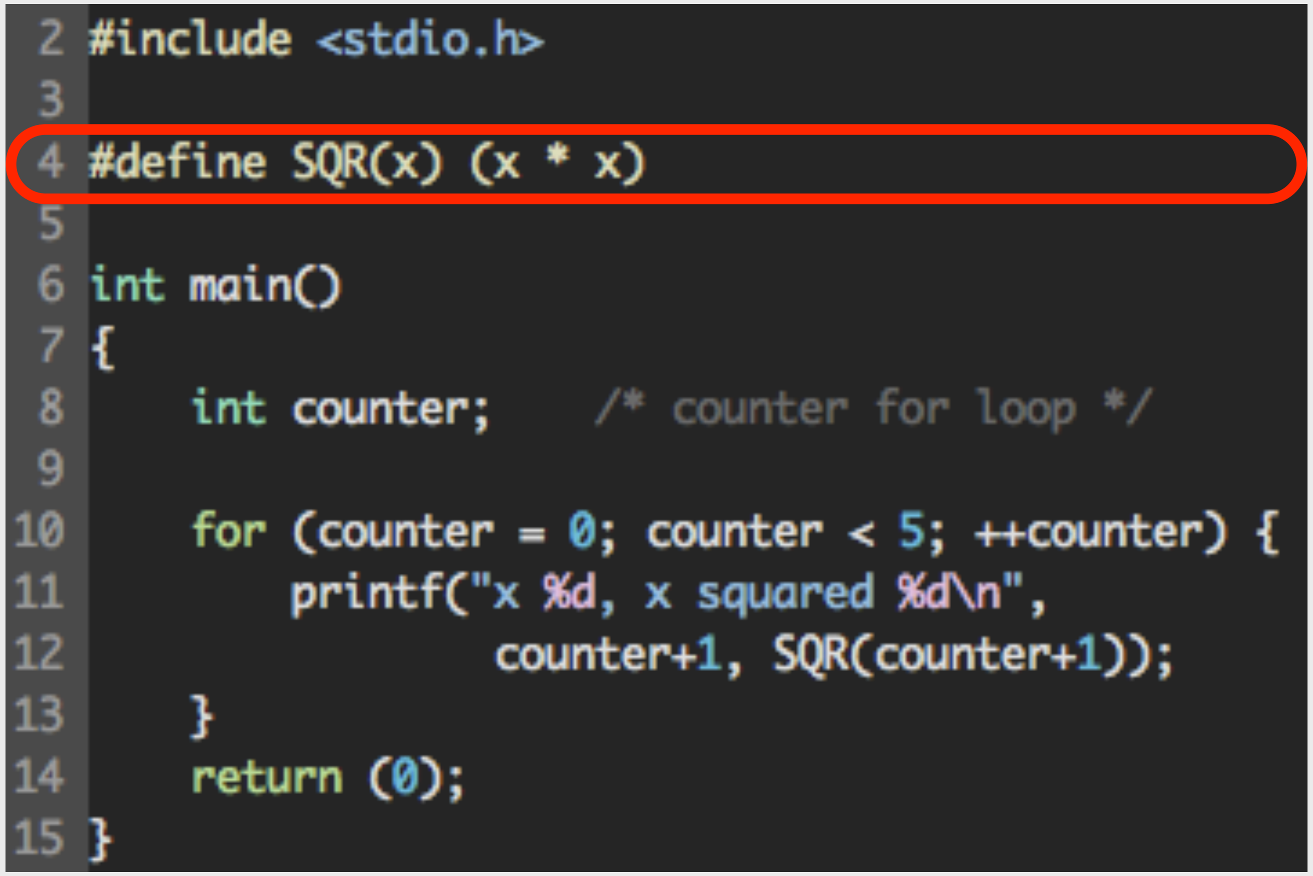
```c
#include <stdio.h>

#define SQR(x) (x * x)

int main()
{
    int counter;    /* counter for loop */

    for (counter = 0; counter < 5; ++counter) {
        printf("x %d, x squared %d\n",
                counter+1, SQR(counter+1));
    }
    return (0);
}
```



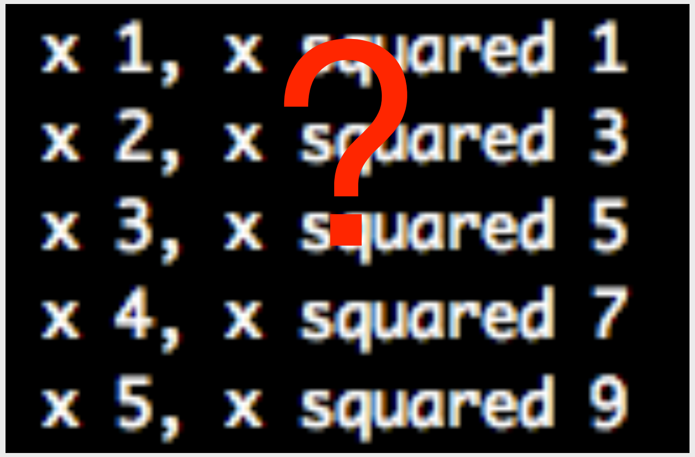
```
x 1, x squared 1
x 2, x squared 3
x 3, x squared 5
x 4, x squared 7
x 5, x squared 9
```

Parameterized Macros

- Example: sqr.c



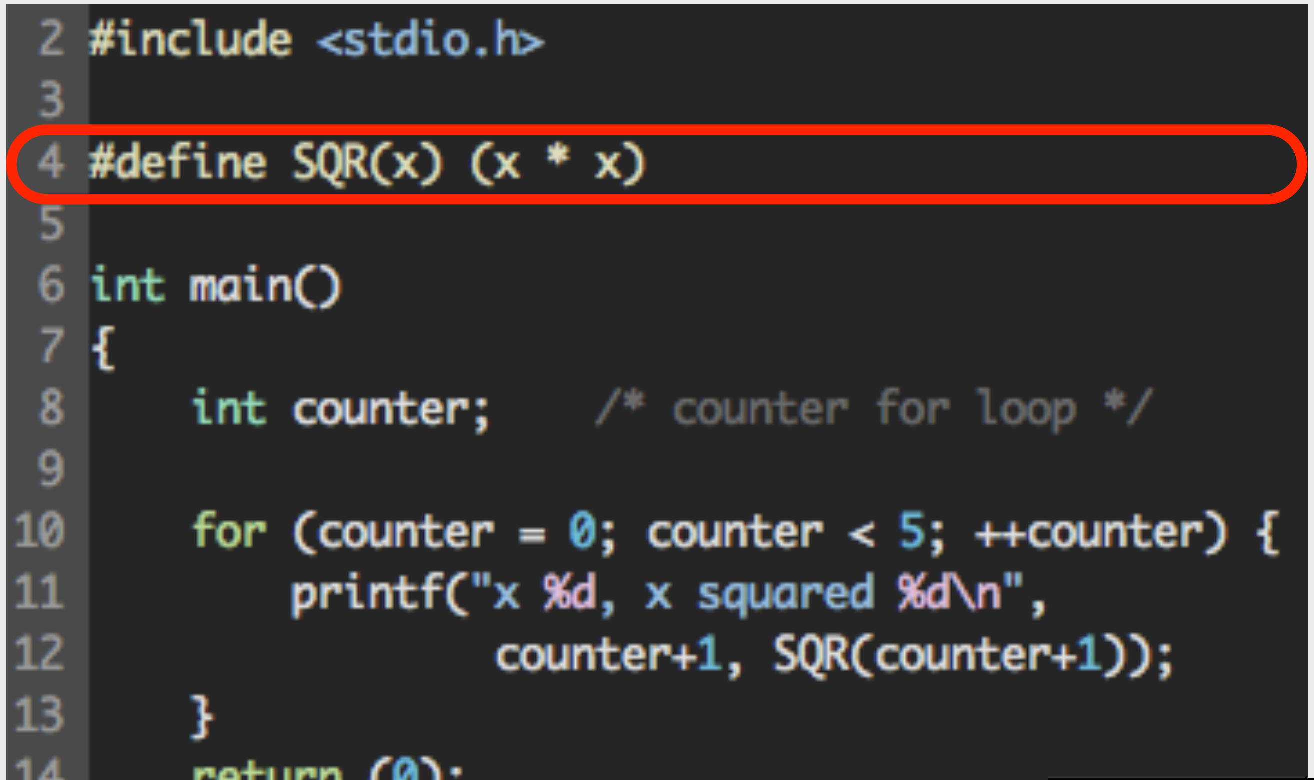
```c
#include <stdio.h>

#define SQR(x) (x * x)

int main()
{
    int counter;     /* counter for loop */

    for (counter = 0; counter < 5; ++counter) {
        printf("x %d, x squared %d\n",
                counter+1, SQR(counter+1));
    }
    return (0);
}
```



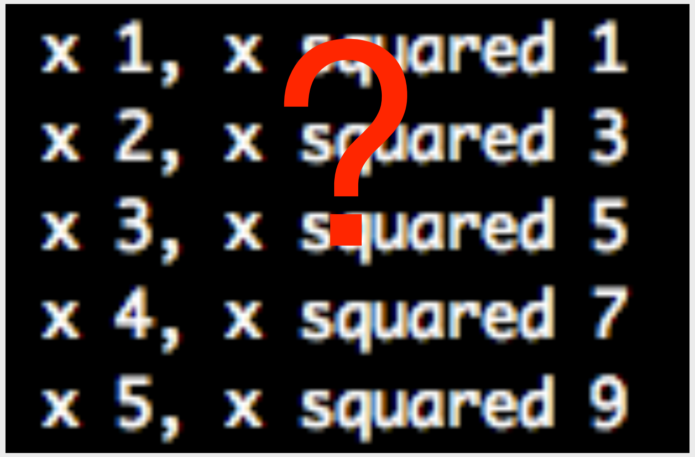
```
x 1, x squared 1
x 2, x squared 3
x 3, x squared 5
x 4, x squared 7
x 5, x squared 9
```

Parameterized Macros

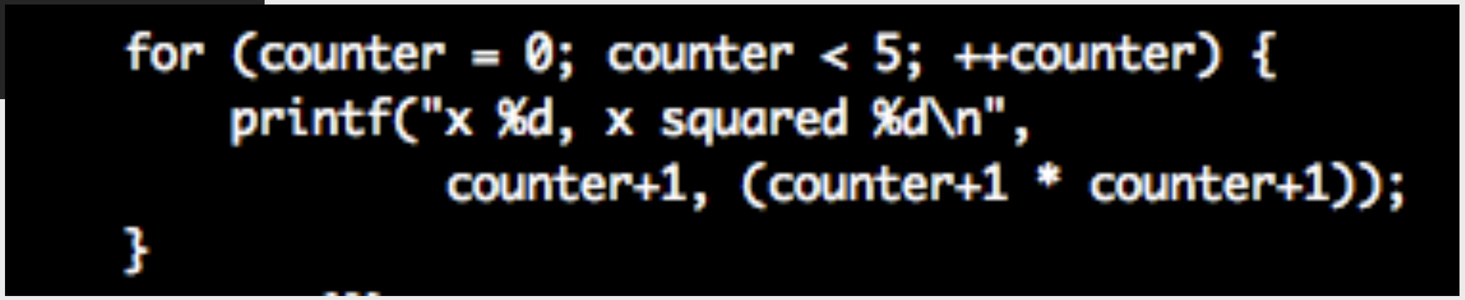
- Example: sqr.c



```
#include <stdio.h>

#define SQR(x) (x * x)

int main()
{
    int counter;    /* counter for loop */

    for (counter = 0; counter < 5; ++counter) {
        printf("x %d, x squared %d\n",
                counter+1, SQR(counter+1));
    }
    return (0);
}
```



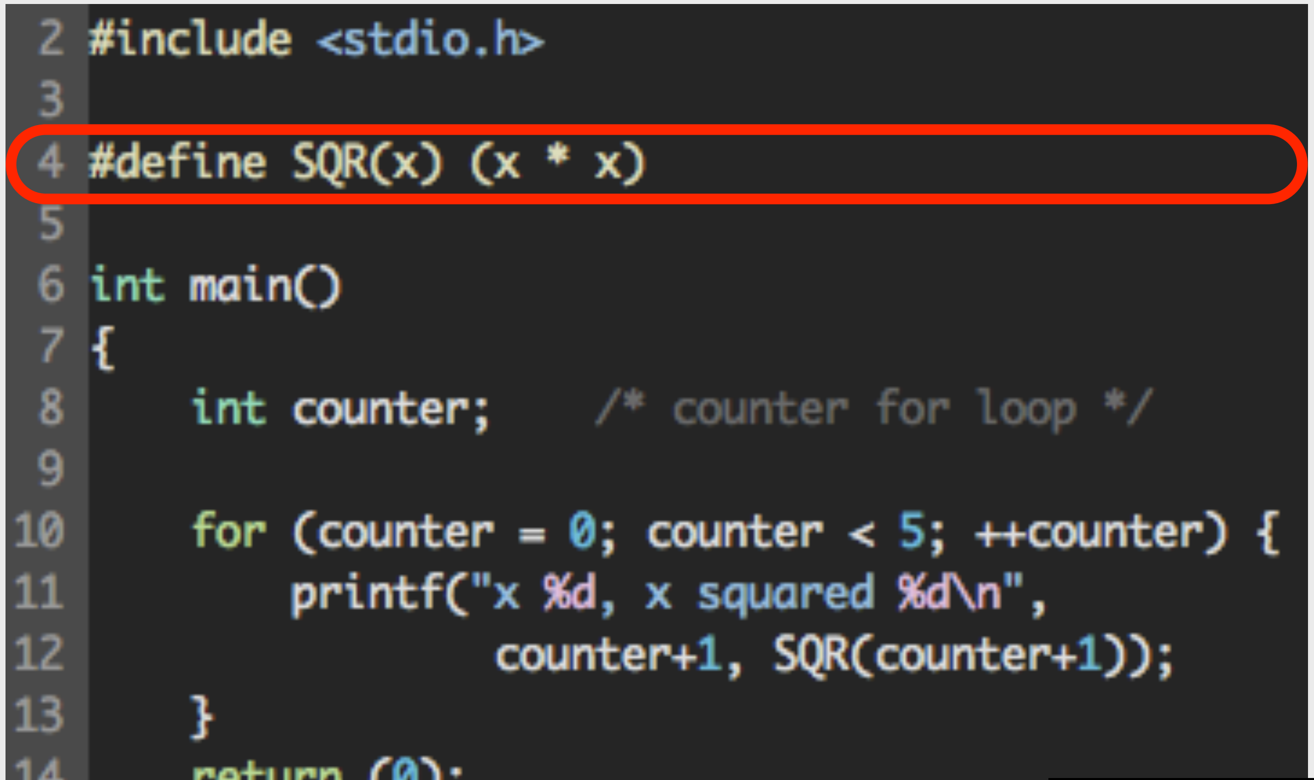
```
x 1, x squared 1
x 2, x squared 3
x 3, x squared 5
x 4, x squared 7
x 5, x squared 9
```



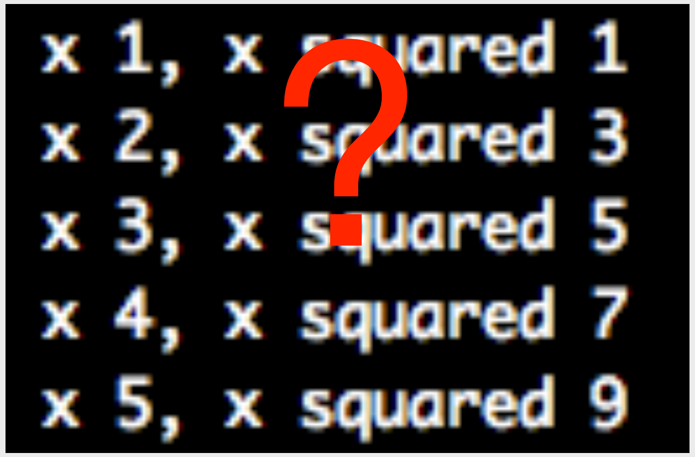
```
for (counter = 0; counter < 5; ++counter) {
    printf("x %d, x squared %d\n",
            counter+1, (counter+1 * counter+1));
}
```

Parameterized Macros

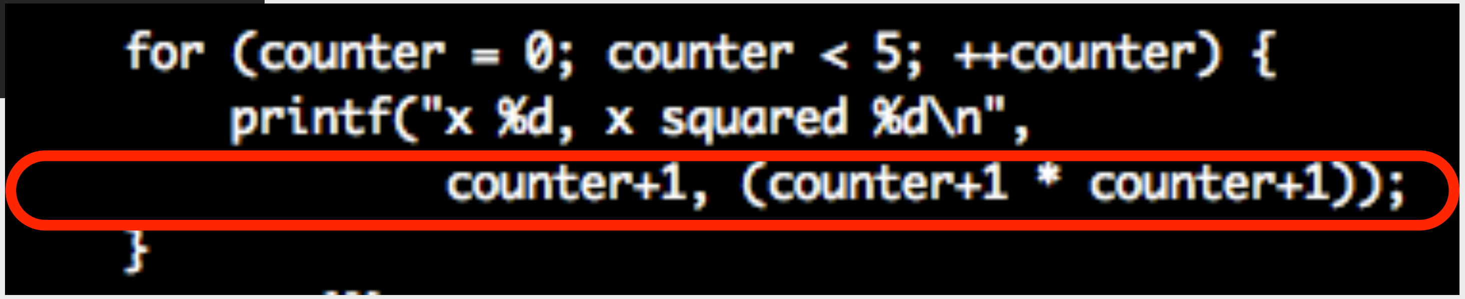
- Example: sqr.c



```c
#include <stdio.h>

#define SQR(x) (x * x)

int main()
{
    int counter;     /* counter for loop */

    for (counter = 0; counter < 5; ++counter) {
        printf("x %d, x squared %d\n",
               counter+1, SQR(counter+1));
    }
    return (0);
}
```



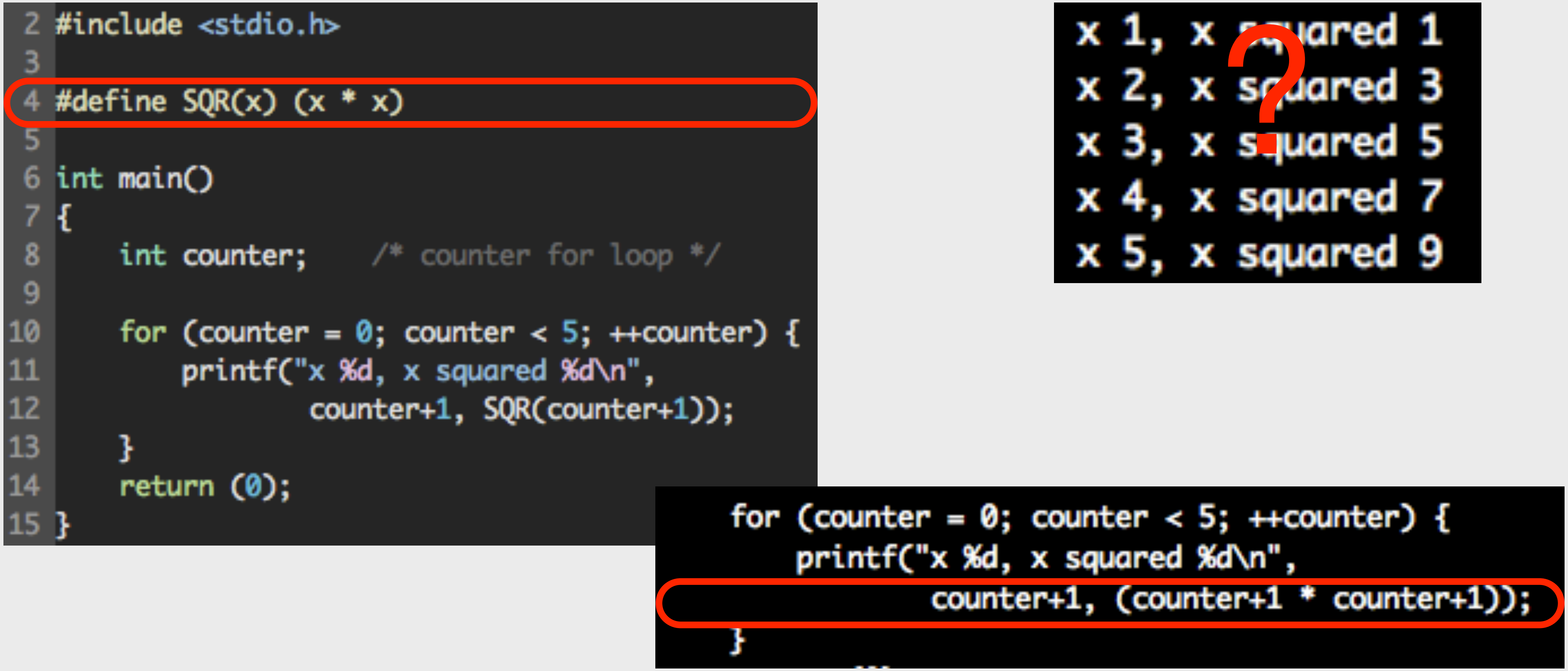
```
x 1, x squared 1
x 2, x squared 3
x 3, x squared 5
x 4, x squared 7
x 5, x squared 9
```



```c
    for (counter = 0; counter < 5; ++counter) {
        printf("x %d, x squared %d\n",
               counter+1, (counter+1 * counter+1));
    }
```

Parameterized Macros

- Example: sqr.c



```c
#include <stdio.h>

#define SQR(x) (x * x)

int main()
{
    int counter;    /* counter for loop */

    for (counter = 0; counter < 5; ++counter) {
        printf("x %d, x squared %d\n",
               counter+1, SQR(counter+1));
    }
    return (0);
}
```

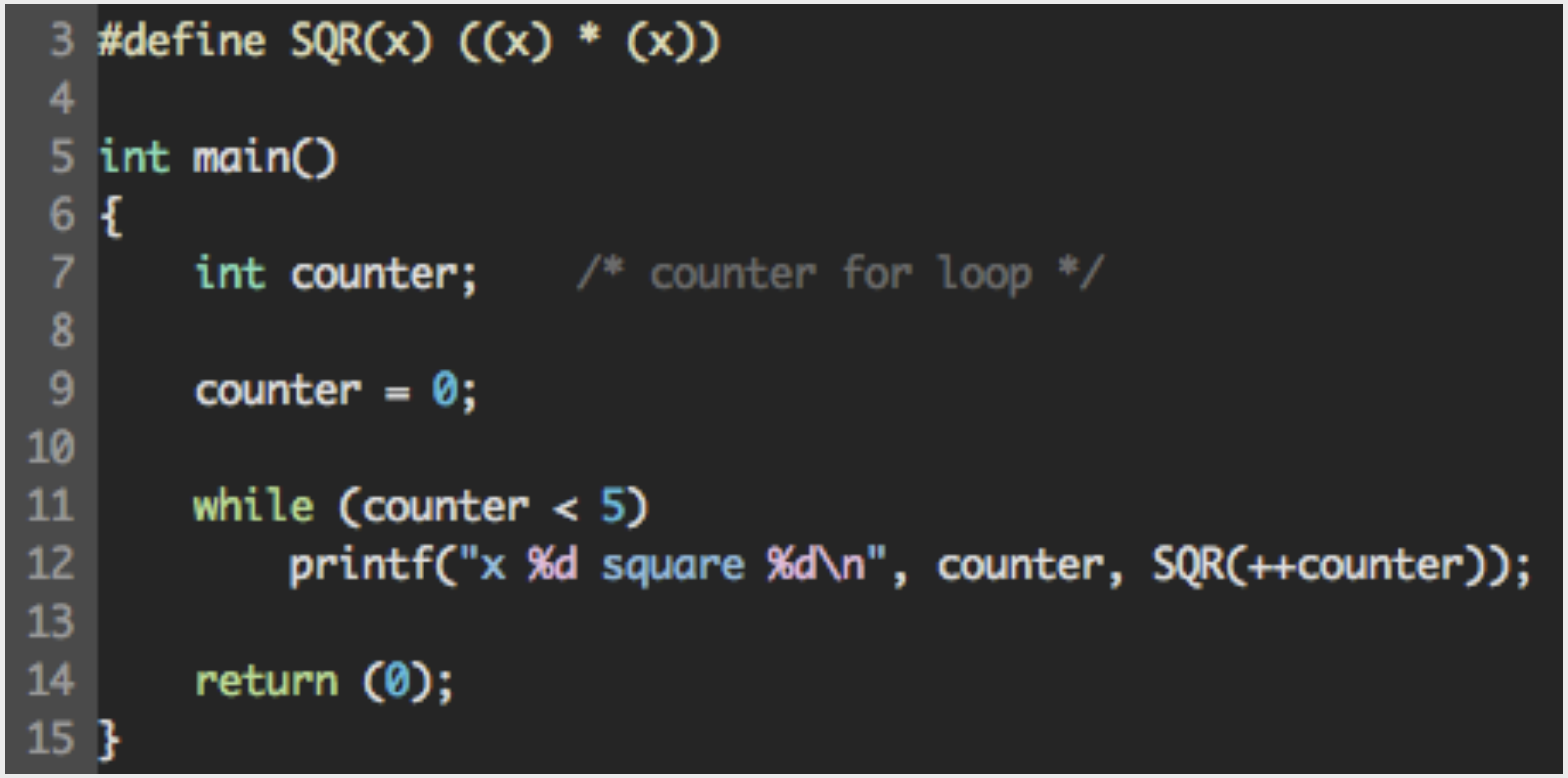
```
x 1, x squared 1
x 2, x squared 3
x 3, x squared 5
x 4, x squared 7
x 5, x squared 9
```

```c
    for (counter = 0; counter < 5; ++counter) {
        printf("x %d, x squared %d\n",
               counter+1, (counter+1 * counter+1));
    }
```

Always put parentheses, (), around the parameters of a macro

Parameterized Macros

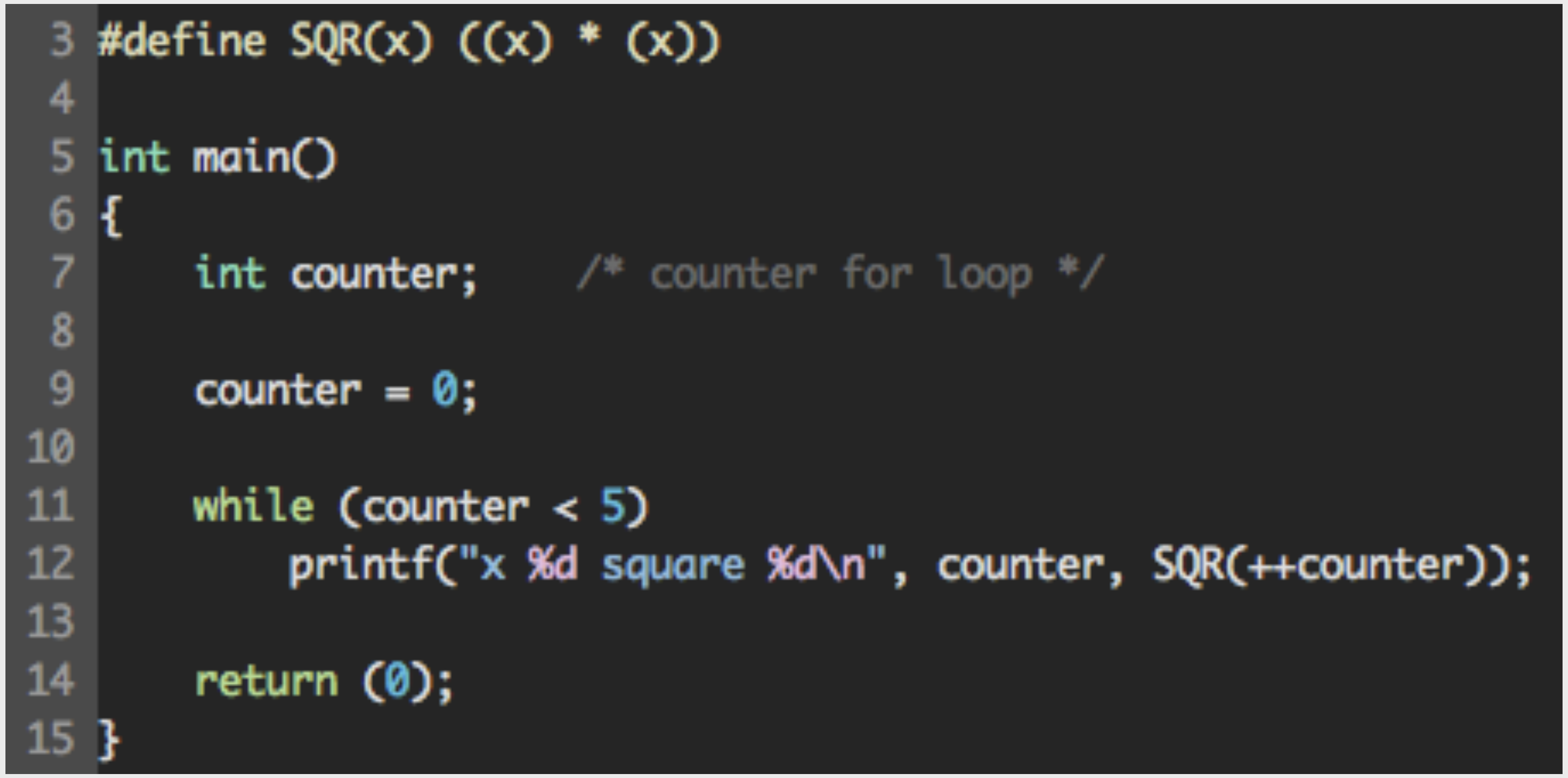
- Example: sqr-i.c



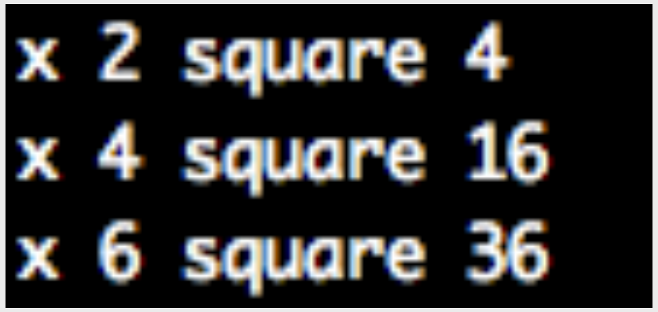
```c
#define SQR(x) ((x) * (x))

int main()
{
    int counter;    /* counter for loop */

    counter = 0;

    while (counter < 5)
        printf("x %d square %d\n", counter, SQR(++counter));

    return (0);
}
```

Parameterized Macros

- Example: sqr-i.c



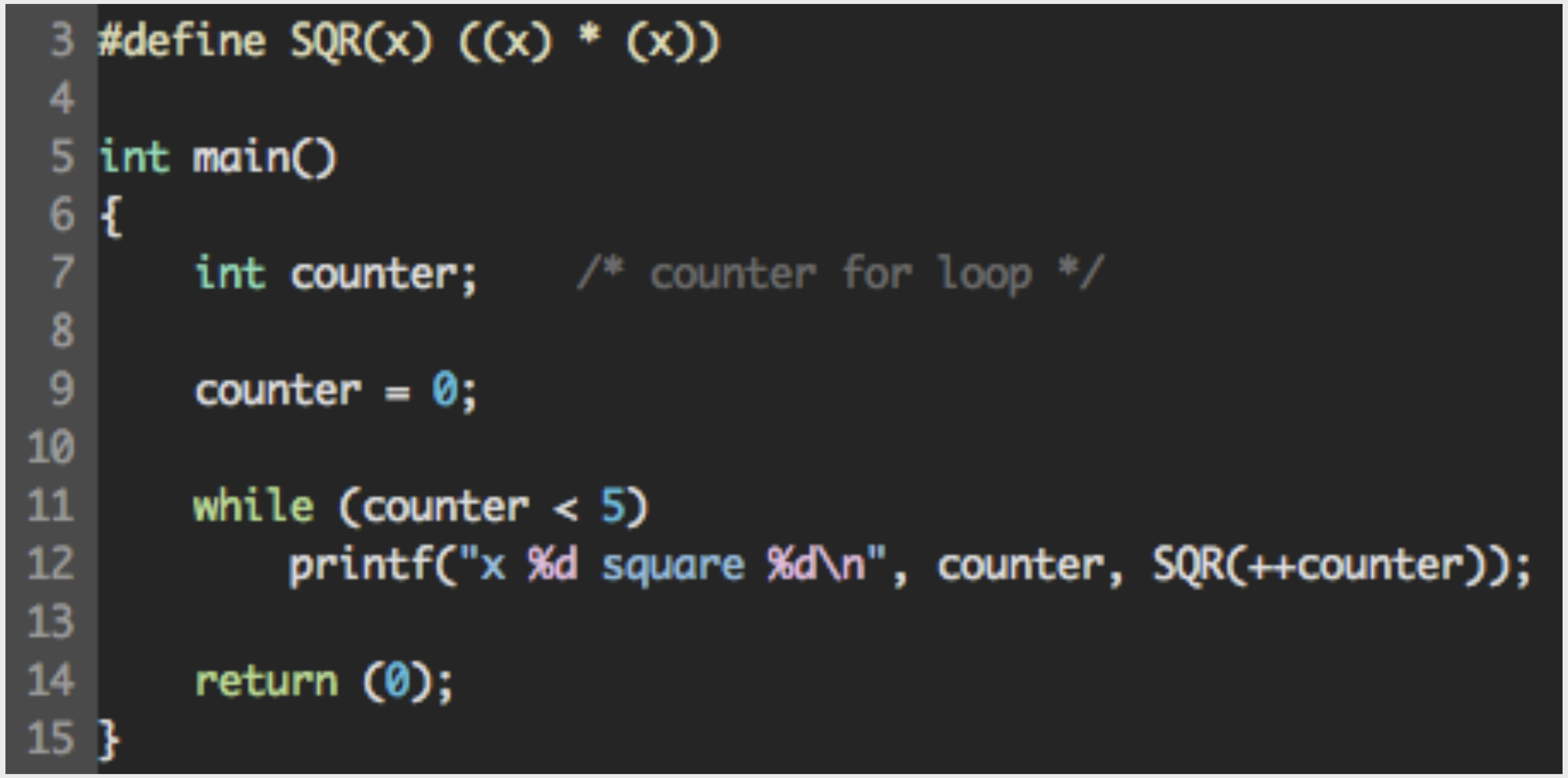
```c
#define SQR(x) ((x) * (x))

int main()
{
    int counter;    /* counter for loop */

    counter = 0;

    while (counter < 5)
        printf("x %d square %d\n", counter, SQR(++counter));

    return (0);
}
```



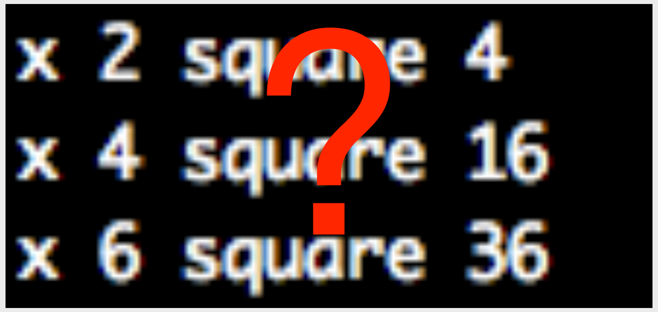
```
x 2 square 4
x 4 square 16
x 6 square 36
```

Parameterized Macros

- Example: sqr-i.c



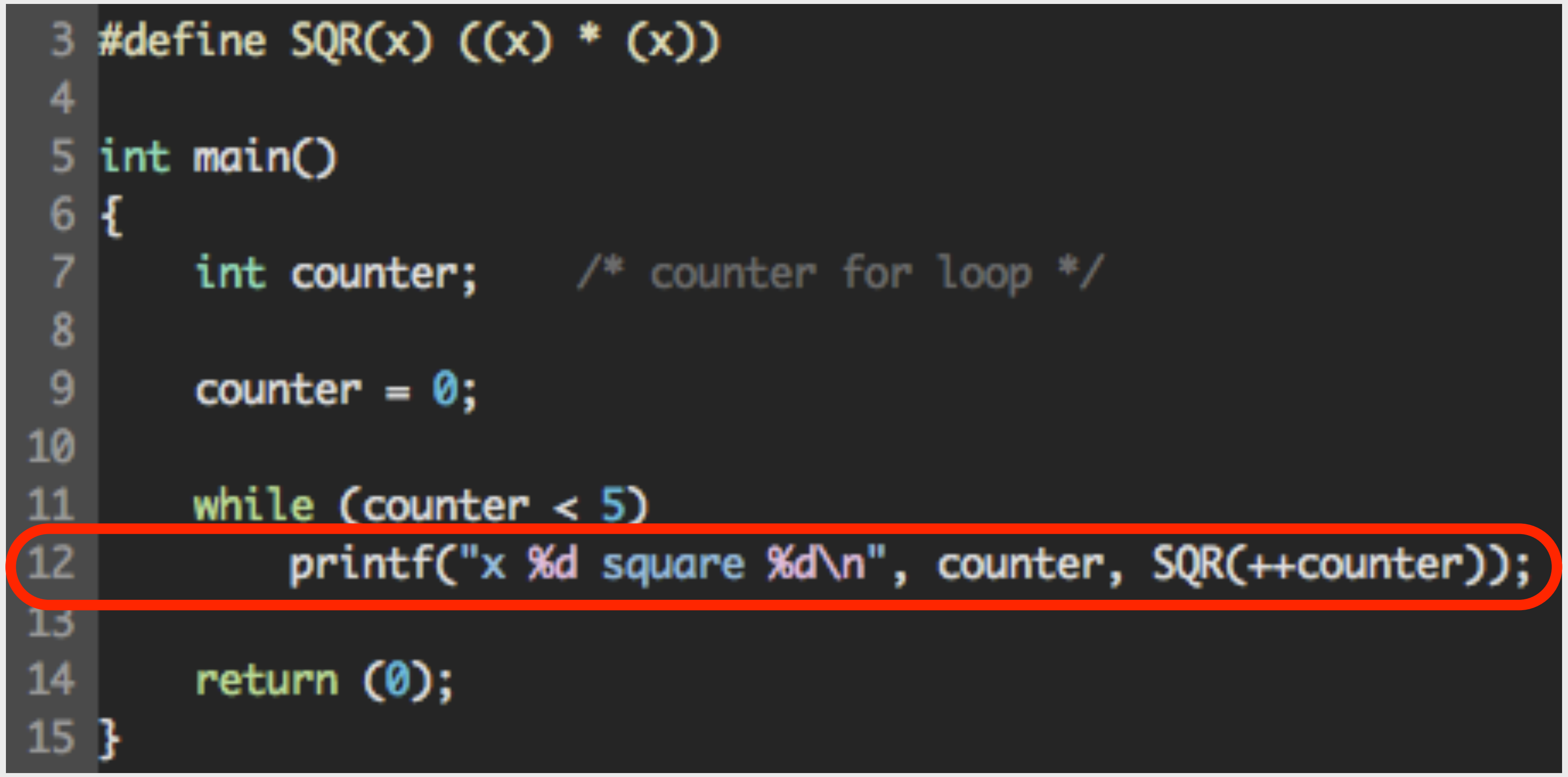
```c
#define SQR(x) ((x) * (x))

int main()
{
    int counter;     /* counter for loop */

    counter = 0;

    while (counter < 5)
        printf("x %d square %d\n", counter, SQR(++counter));

    return (0);
}
```



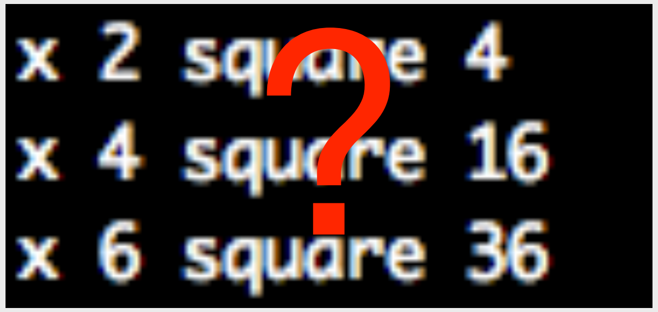
```
x 2 square 4
x 4 square 16
x 6 square 36
```

Parameterized Macros

- Example: sqr-i.c



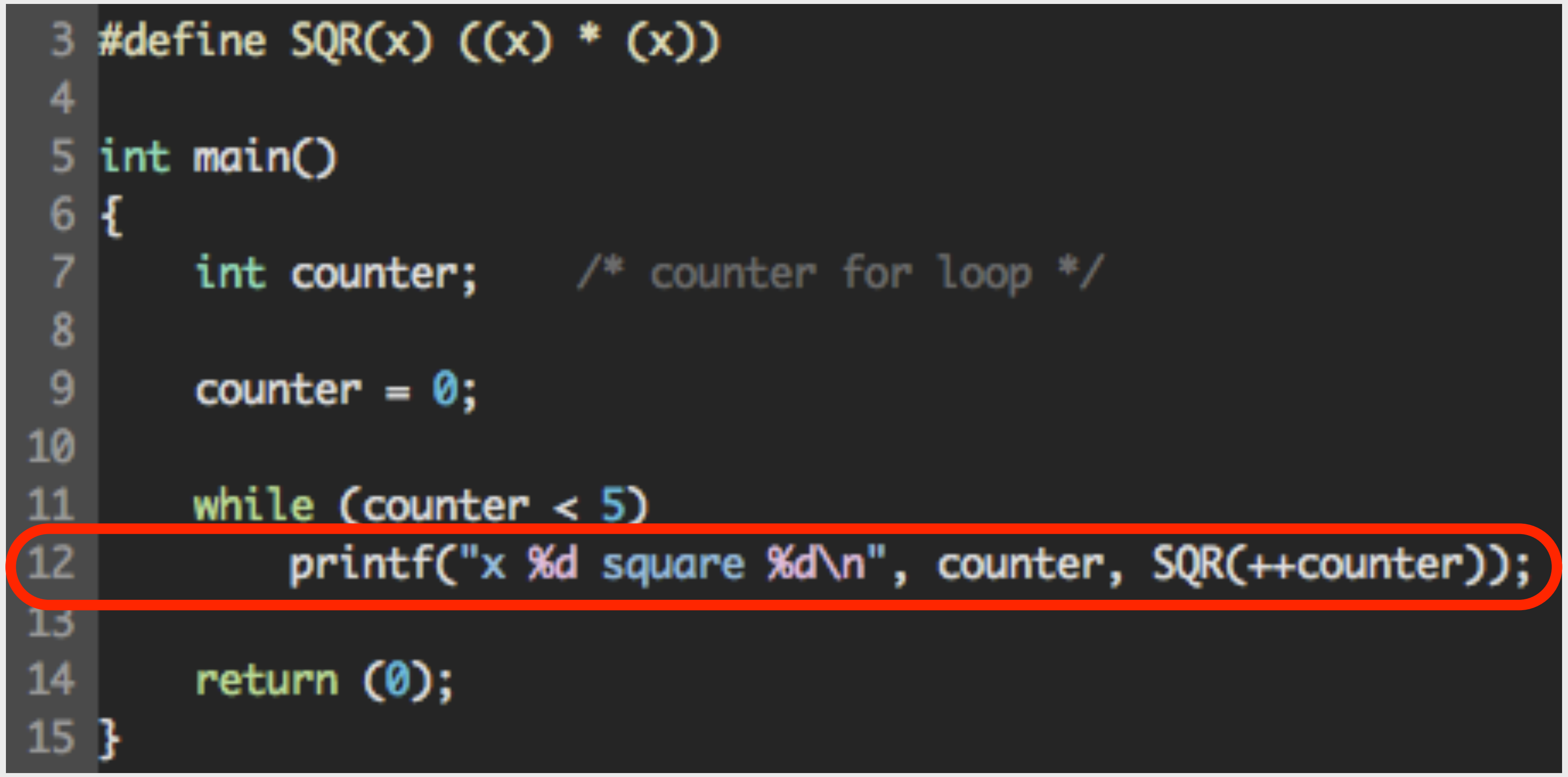
```c
#define SQR(x) ((x) * (x))

int main()
{
    int counter;     /* counter for loop */

    counter = 0;

    while (counter < 5)
        printf("x %d square %d\n", counter, SQR(++counter));

    return (0);
}
```



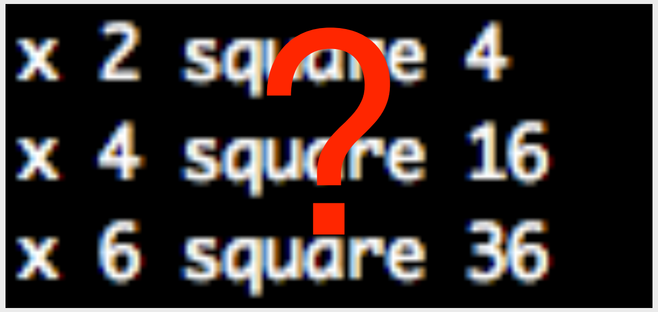
```
x 2 square 4
x 4 square 16
x 6 square 36
```

Parameterized Macros

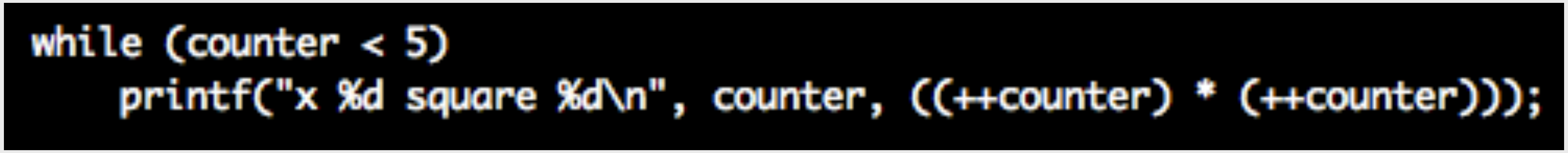
- Example: sqr-i.c



```
#define SQR(x) ((x) * (x))

int main()
{
    int counter;     /* counter for loop */

    counter = 0;

    while (counter < 5)
        printf("x %d square %d\n", counter, SQR(++counter));

    return (0);
}
```



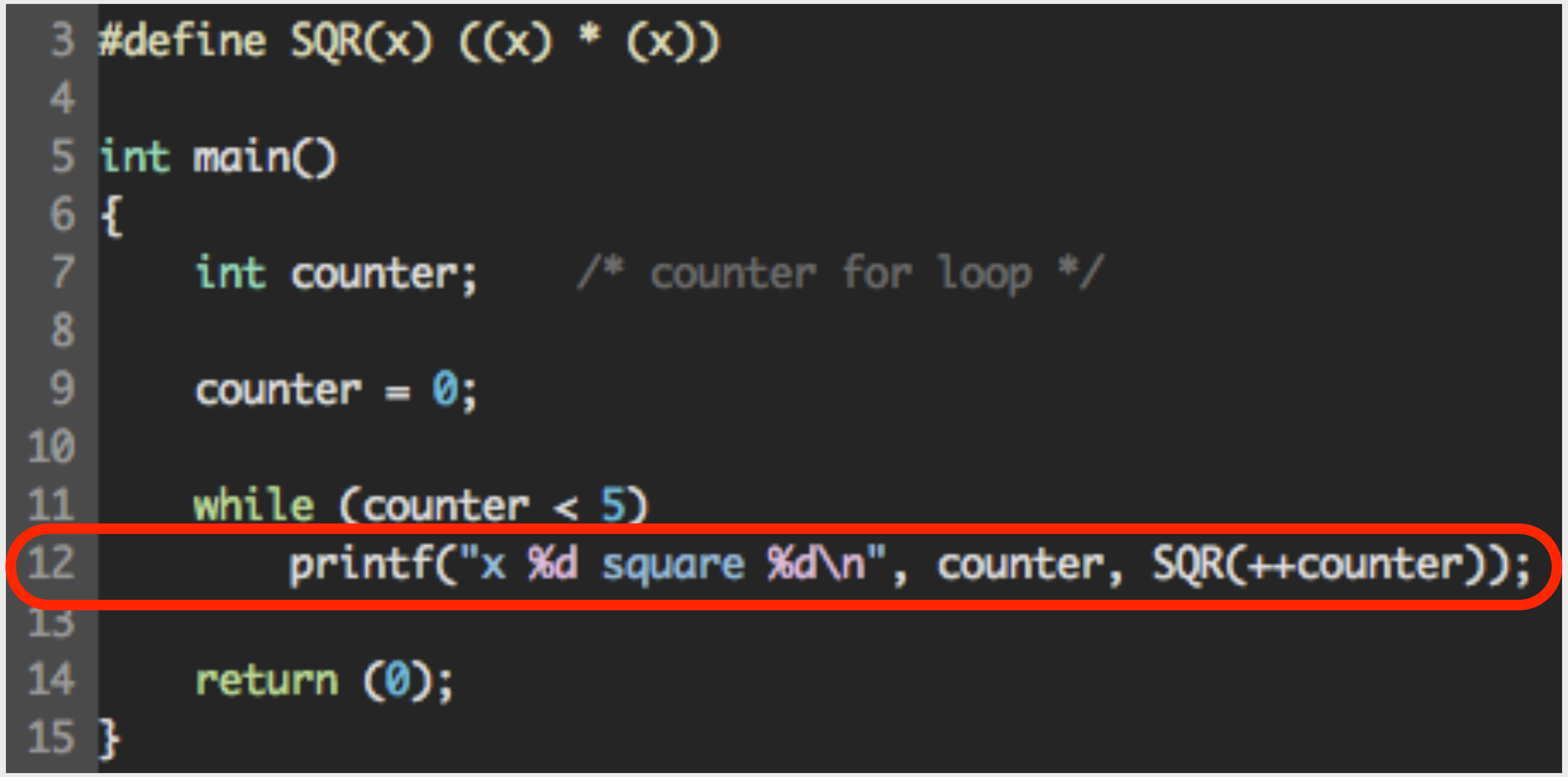
```
x 2 square 4
x 4 square 16
x 6 square 36
```



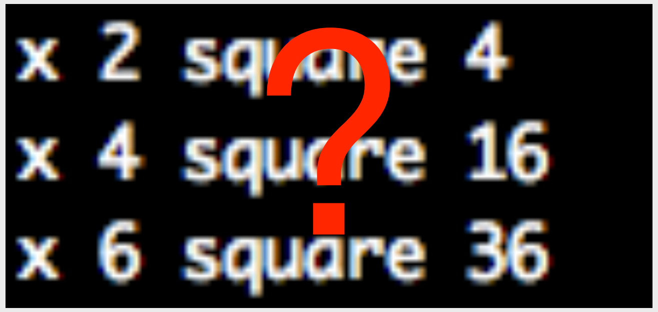
```
while (counter < 5)
    printf("x %d square %d\n", counter, ((++counter) * (++counter)));
```

Parameterized Macros

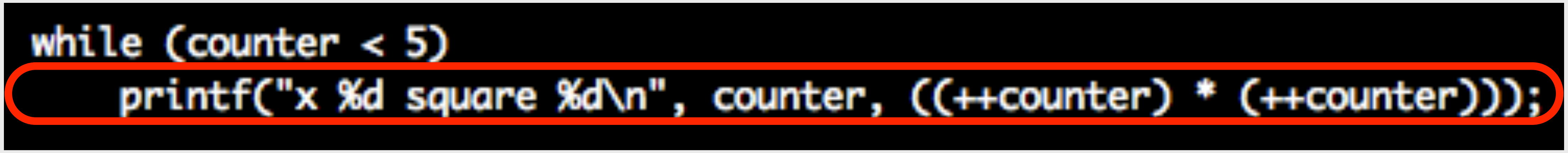
- Example: sqr-i.c



```
#define SQR(x) ((x) * (x))

int main()
{
    int counter;    /* counter for loop */

    counter = 0;

    while (counter < 5)
        printf("x %d square %d\n", counter, SQR(++counter));

    return (0);
}
```



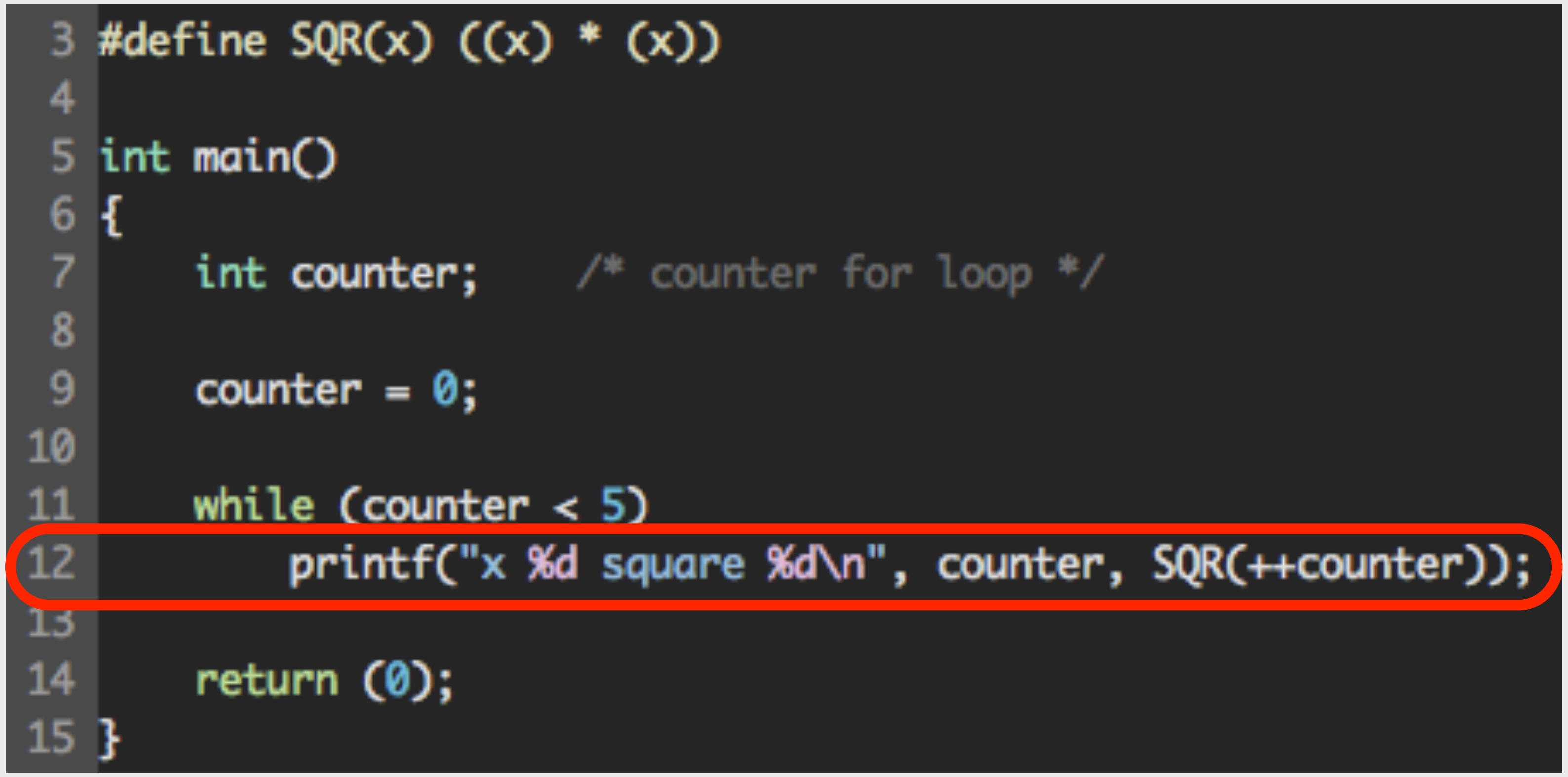
```
x 2 square 4
x 4 square 16
x 6 square 36
```



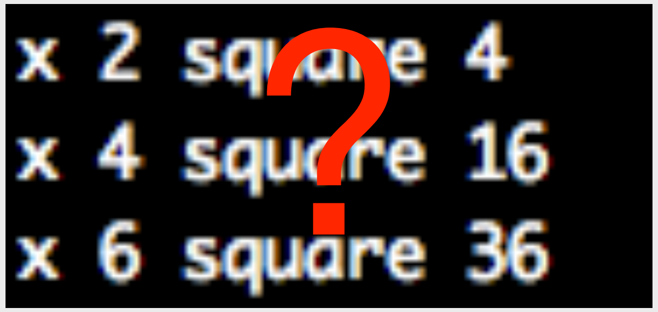
```
while (counter < 5)
    printf("x %d square %d\n", counter, ((++counter) * (++counter)));
```

Parameterized Macros

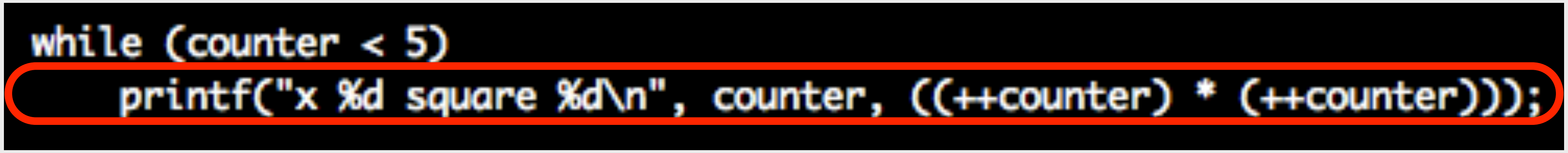
- Example: sqr-i.c



```
#define SQR(x) ((x) * (x))

int main()
{
    int counter;     /* counter for loop */

    counter = 0;

    while (counter < 5)
        printf("x %d square %d\n", counter, SQR(++counter));

    return (0);
}
```



```
x 2 square 4
x 4 square 16
x 6 square 36
```

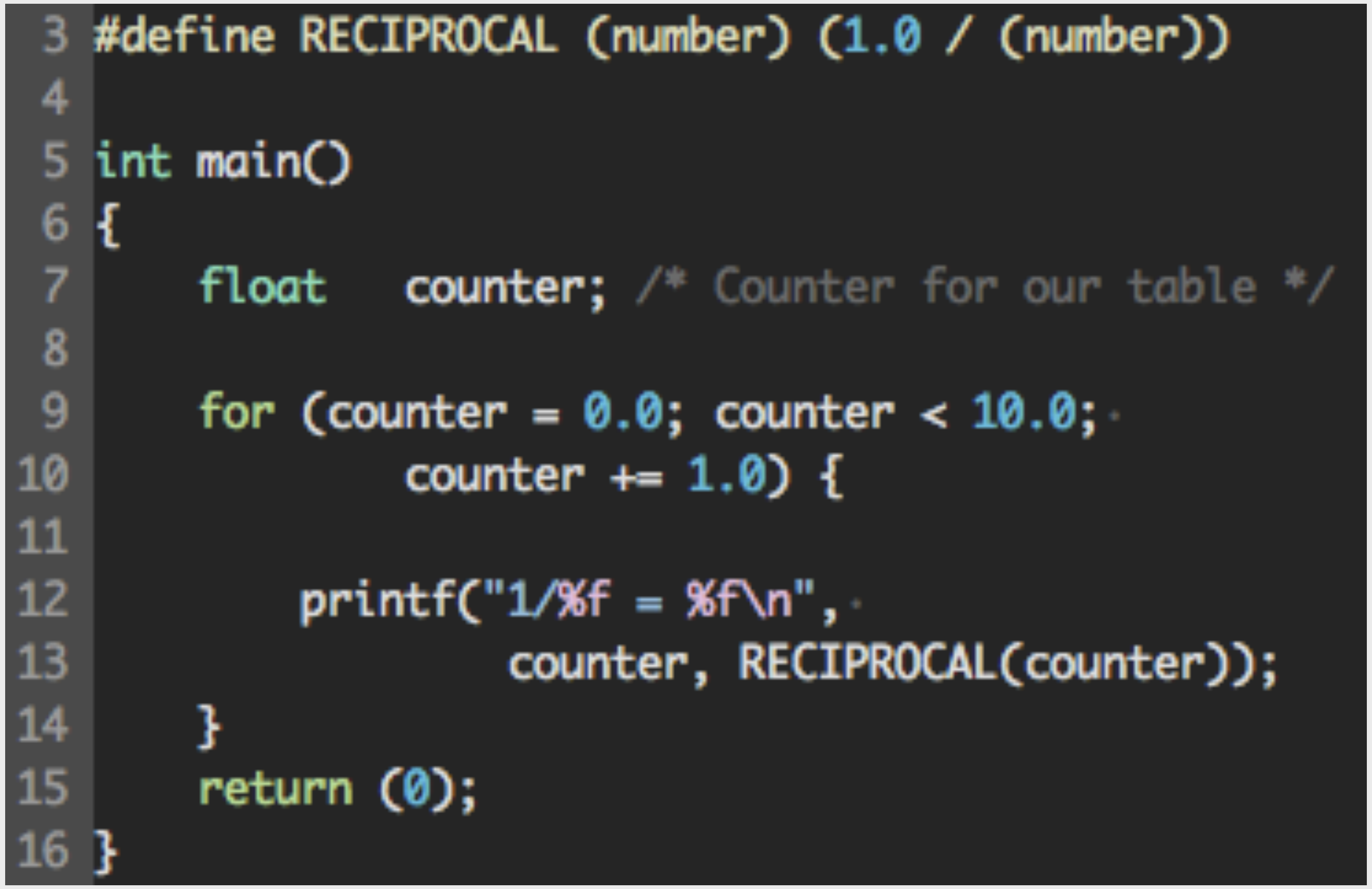


```
while (counter < 5)
    printf("x %d square %d\n", counter, ((++counter) * (++counter)));
```

Avoid using the increment (++) or decrement (--) more than once on a single line!!

Parameterized Macros

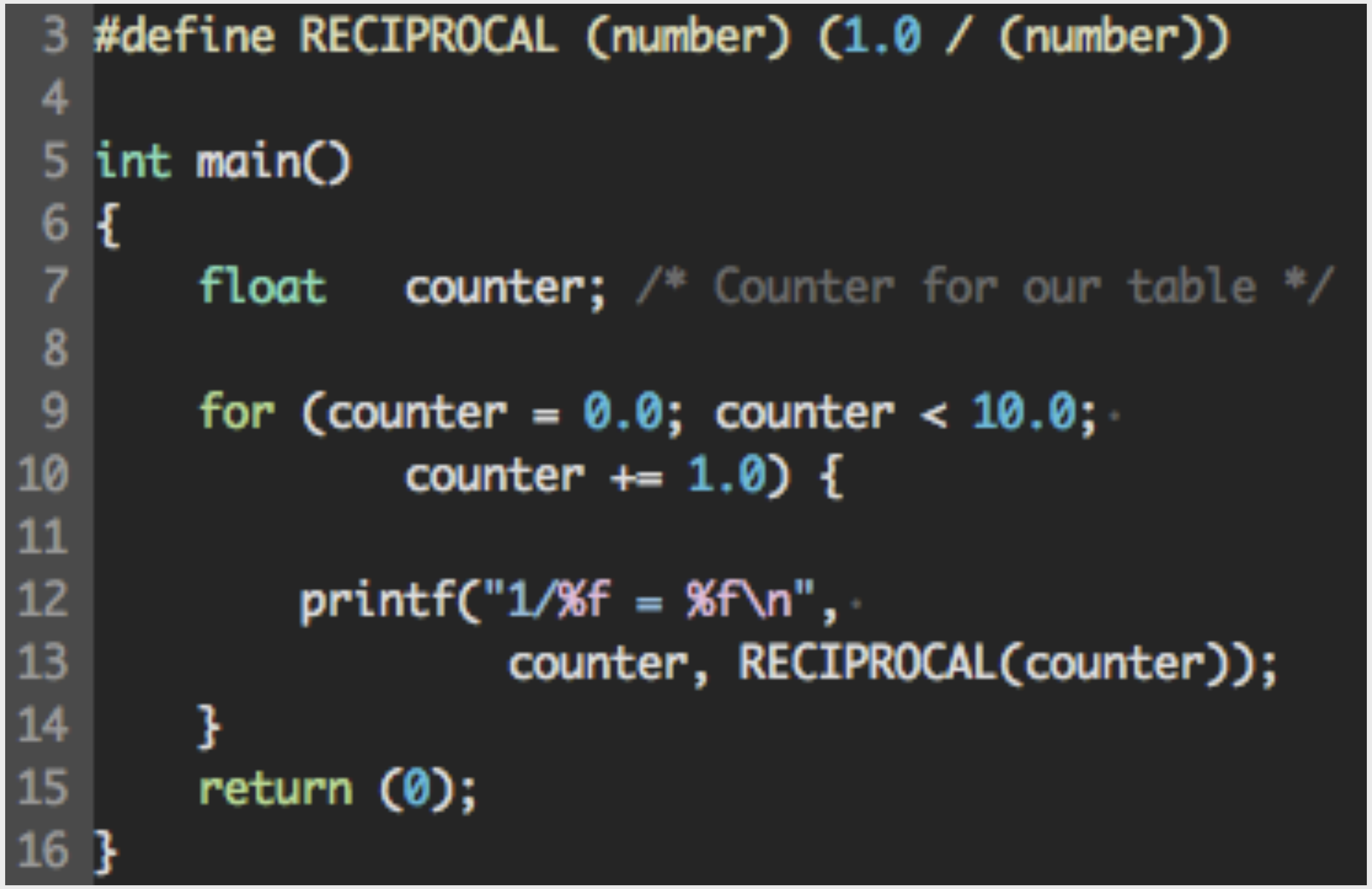
- Example: rec.c



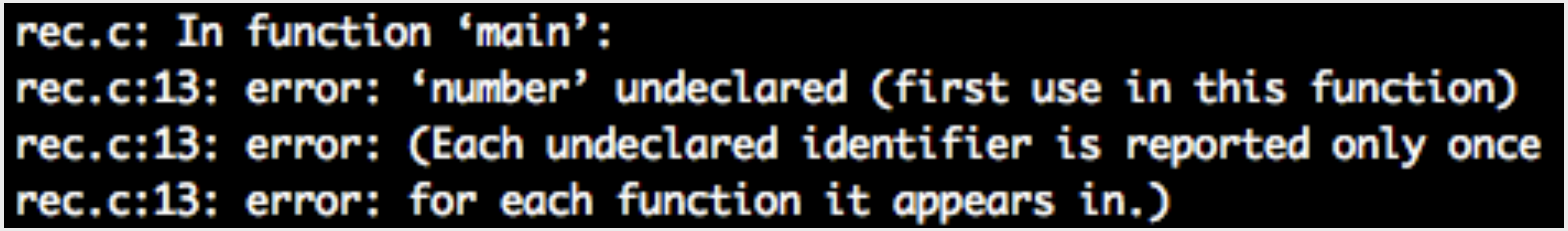
```
#define RECIPROCAL (number) (1.0 / (number))

int main()
{
    float   counter; /* Counter for our table */

    for (counter = 0.0; counter < 10.0;
            counter += 1.0) {

        printf("1/%f = %f\n",
                counter, RECIPROCAL(counter));
    }
    return (0);
}
```

Parameterized Macros

- Example: rec.c



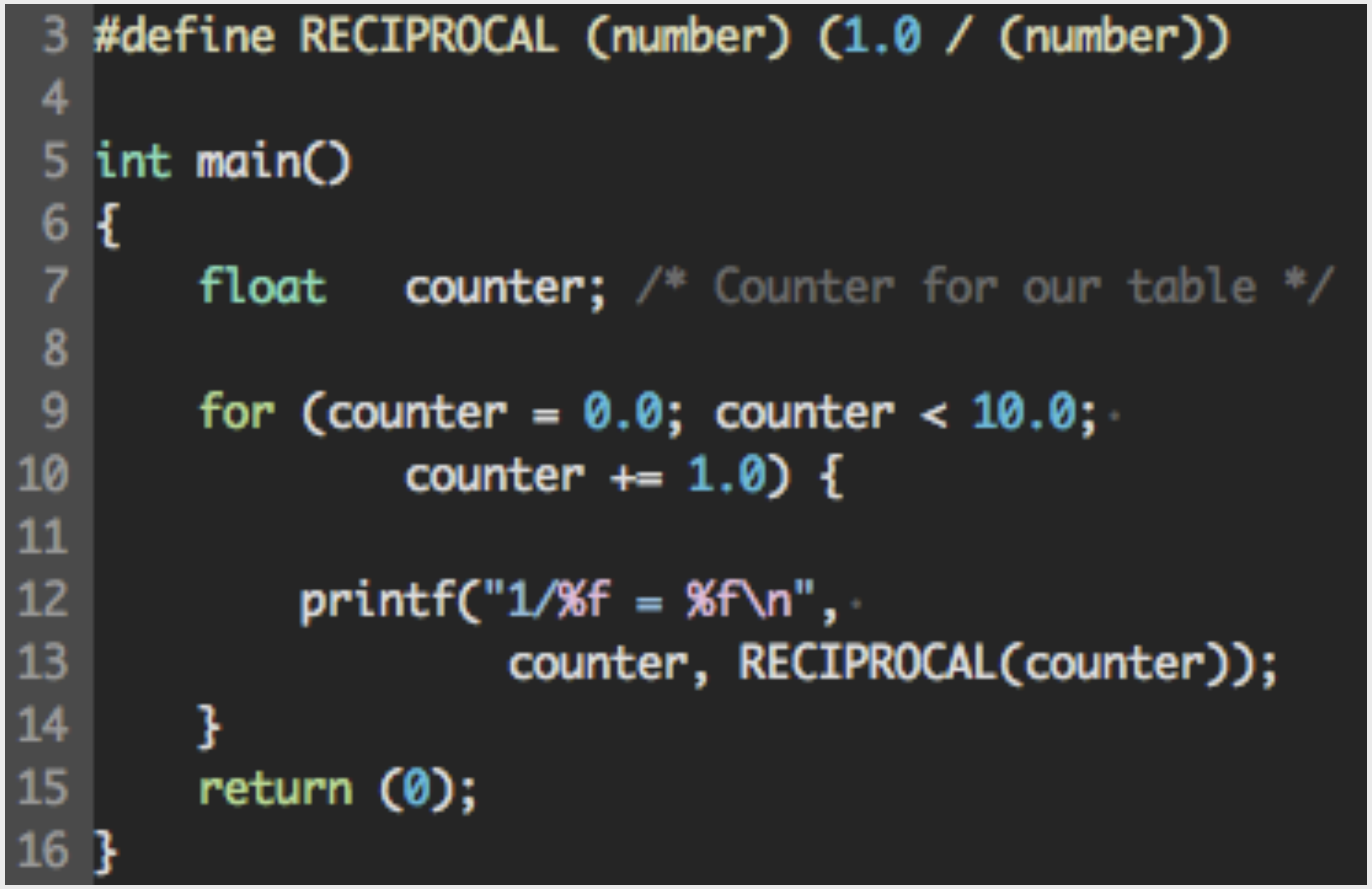
```
 3 #define RECIPROCAL (number) (1.0 / (number))
 4
 5 int main()
 6 {
 7     float   counter; /* Counter for our table */
 8
 9     for (counter = 0.0; counter < 10.0;
10             counter += 1.0) {
11
12         printf("1/%f = %f\n",
13                 counter, RECIPROCAL(counter));
14     }
15     return (0);
16 }
```



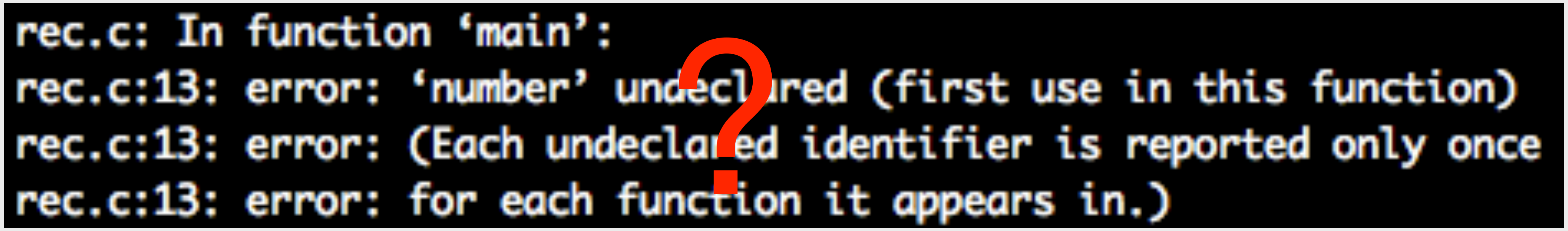
```
rec.c: In function ‘main’:
rec.c:13: error: ‘number’ undeclared (first use in this function)
rec.c:13: error: (Each undeclared identifier is reported only once
rec.c:13: error: for each function it appears in.)
```

Parameterized Macros

- Example: rec.c



```
 3 #define RECIPROCAL (number) (1.0 / (number))
 4 
 5 int main()
 6 {
 7     float   counter; /* Counter for our table */
 8 
 9     for (counter = 0.0; counter < 10.0;
10             counter += 1.0) {
11 
12         printf("1/%f = %f\n",
13                 counter, RECIPROCAL(counter));
14     }
15     return (0);
16 }
```

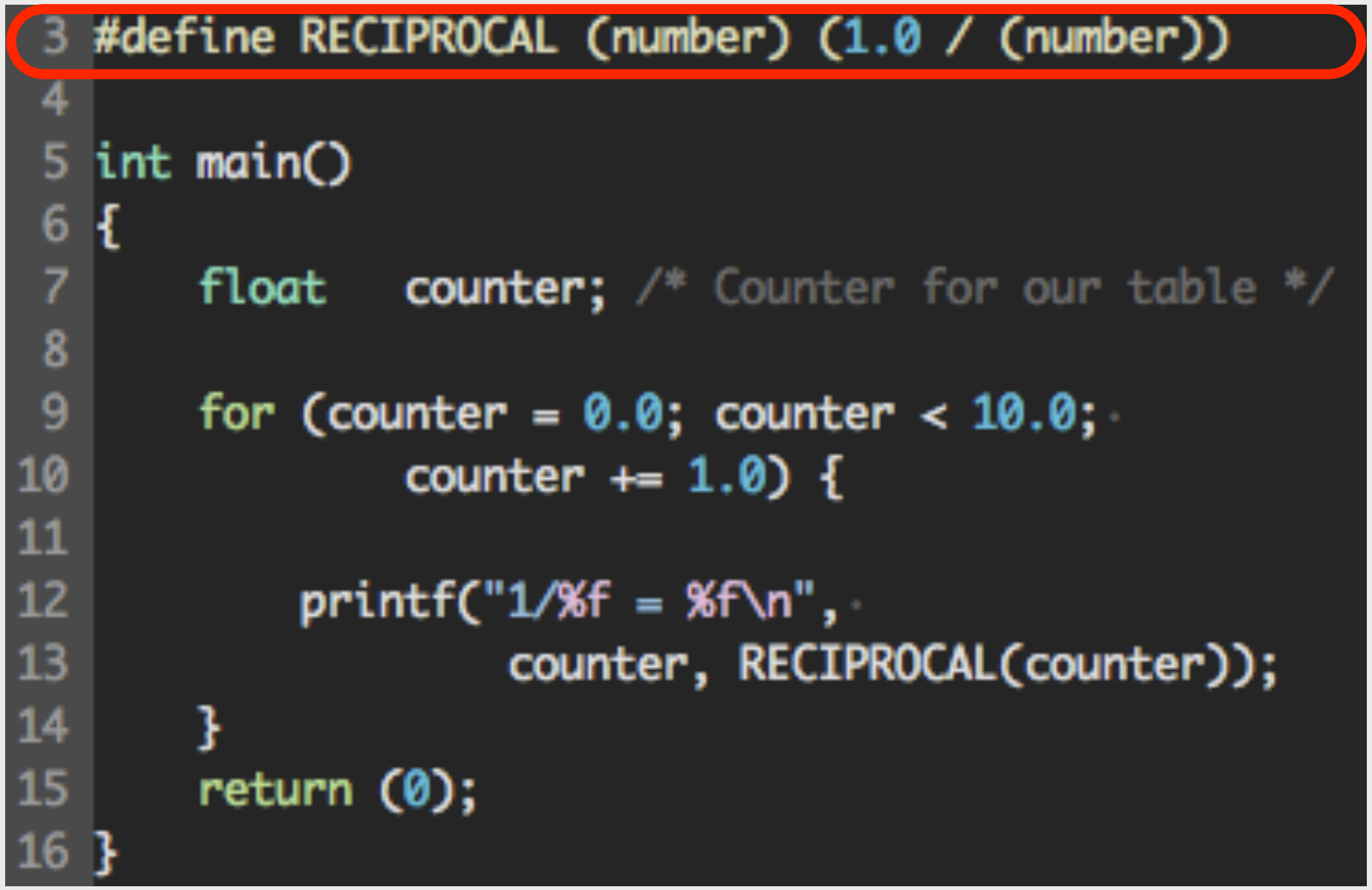


```
rec.c: In function ‘main’:
rec.c:13: error: ‘number’ undeclared (first use in this function)
rec.c:13: error: (Each undeclared identifier is reported only once
rec.c:13: error: for each function it appears in.)
```

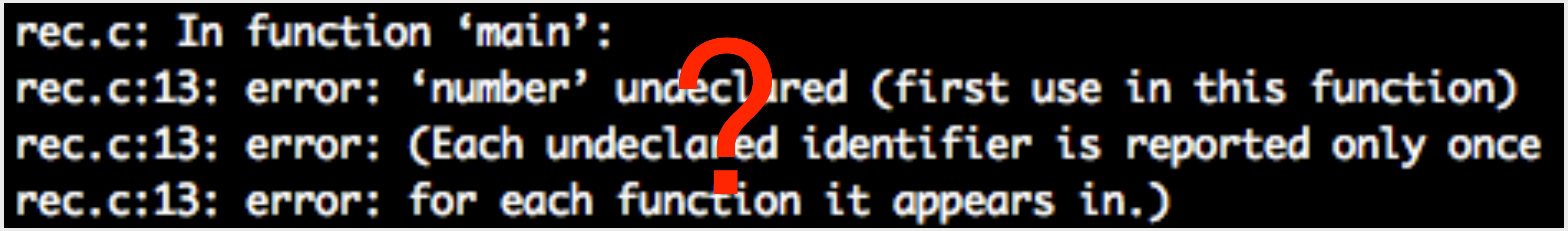
?

Parameterized Macros

- Example: rec.c



```
 3 #define RECIPROCAL (number) (1.0 / (number))
 4 
 5 int main()
 6 {
 7     float   counter; /* Counter for our table */
 8 
 9     for (counter = 0.0; counter < 10.0;
10             counter += 1.0) {
11 
12         printf("1/%f = %f\n",
13                 counter, RECIPROCAL(counter));
14     }
15     return (0);
16 }
```

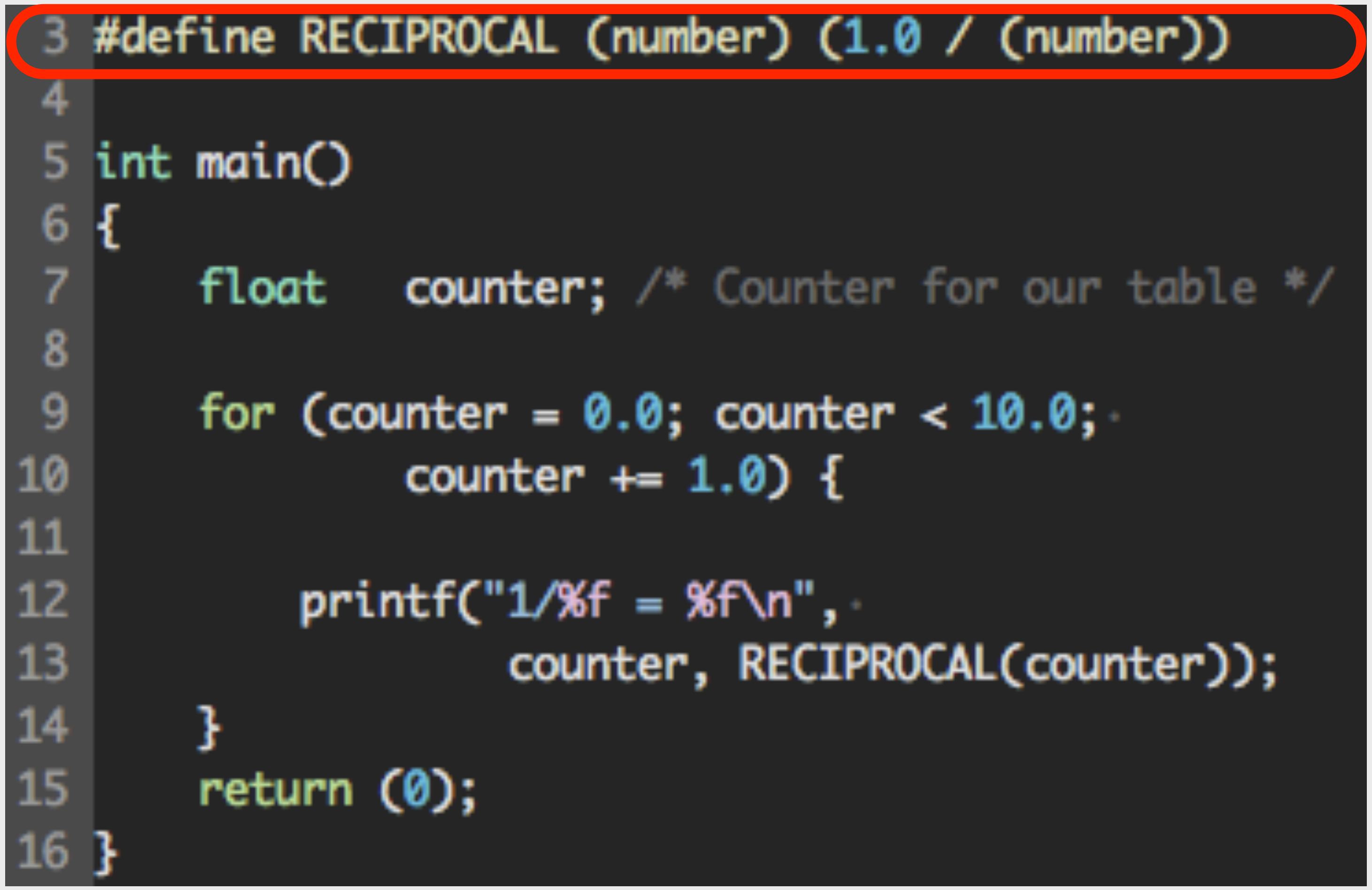


```
rec.c: In function ‘main’:
rec.c:13: error: ‘number’ undeclared (first use in this function)
rec.c:13: error: (Each undeclared identifier is reported only once
rec.c:13: error: for each function it appears in.)
```

?

Parameterized Macros

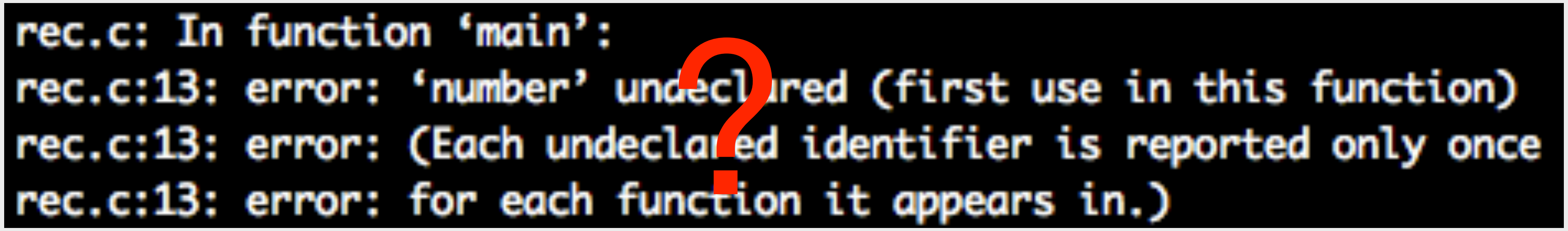
- Example: rec.c



```c
#define RECIPROCAL (number) (1.0 / (number))

int main()
{
    float   counter; /* Counter for our table */

    for (counter = 0.0; counter < 10.0;
            counter += 1.0) {

        printf("1/%f = %f\n",
                counter, RECIPROCAL(counter));
    }
    return (0);
}
```

Remove the space between RECIPROCAL and (number)!!

** Watch out the space when using macros!!



```
rec.c: In function ‘main’:
rec.c:13: error: ‘number’ undeclared (first use in this function)
rec.c:13: error: (Each undeclared identifier is reported only once
rec.c:13: error: for each function it appears in.)
```

Summary

Summary

- C preprocessor is a useful part of the C language

Summary

- C preprocessor is a useful part of the C language
- Simple rules to avoid problems

Summary

- C preprocessor is a useful part of the C language
- Simple rules to avoid problems
  - Put parentheses ( ) around everything

Summary

- C preprocessor is a useful part of the C language
- Simple rules to avoid problems
  - Put parentheses **( )** around everything
  - When defining a macro with more than one statement, enclose the code in curly braces **{ }**

Summary

- C preprocessor is a useful part of the C language
- Simple rules to avoid problems
  - Put parentheses **( )** around everything
  - When defining a macro with more than one statement, enclose the code in curly braces **{ }**
  - The preprocessor is not C. **Don’t use = and ;**

Advanced Data Types

Structures

- General form of a structure definition:

Structures

- General form of a structure definition:

```
struct structure-name {
    field-type field-name; /* comment */
    field-type field-name; /* comment */
    ....
} variable-name;
```

Structure

- Example

Structure

- Example

```
struct bin {
   char name[30];
   int quantity;
   int cost;
} printer_cable_bin;
```

Structure

- Example

```
struct bin {
    char name[30];
    int quantity;
    int cost;
} printer_cable_bin;
```

```
printer_cable_bin.cost = 1295;
```

Structure

- Example

```
        struct bin {
            char name[30];
            int quantity;
            int cost;
        } printer_cable_bin;
```

```
printer_cable_bin.cost = 1295;

struct bin terminal_cable_box;
```

Structure

- Example

Structure

- Example

```
struct {
   char name[30];
   int quantity;
   int cost;
} printer_cable_bin;
```

Structure

- Example

```
struct {
   char name[30];
   int quantity;
   int cost;
} printer_cable_bin;
```

```
struct bin {
   char name[30];
   int quantity;
   int cost;
};
```

Structure Initialization

- Example: structure.c

Structure Initialization

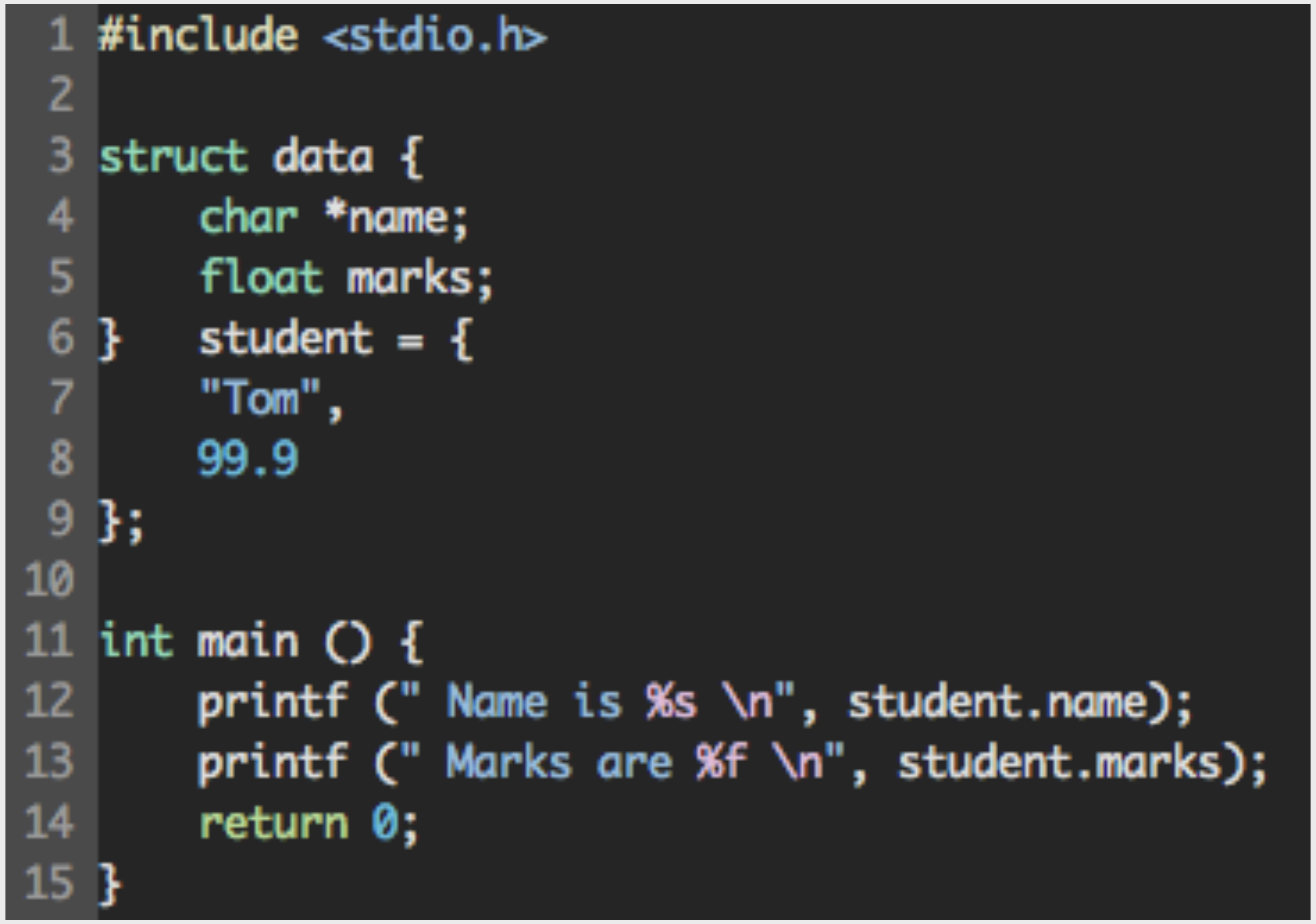
- Example: structure.c

```
struct bin {
    char name[30];
    int quantity;
    int cost;
} printer_cable_bin = {
  “Printer Cables”,
  0,
  1295
};
```

Structure Initialization

- Example: structure.c

```
struct bin {
    char name[30];
    int quantity;
    int cost;
} printer_cable_bin = {
  “Printer Cables”,
  0,
  1295
};
```



```
#include <stdio.h>

struct data {
    char *name;
    float marks;
}   student = {
    "Tom",
    99.9
};

int main () {
    printf (" Name is %s \n", student.name);
    printf (" Marks are %f \n", student.marks);
    return 0;
}
```

Unions

- Structure
  - Define a data type with several fields. Each field takes up a separate storage location
- Union
  - Define a single location that can be given many different field names

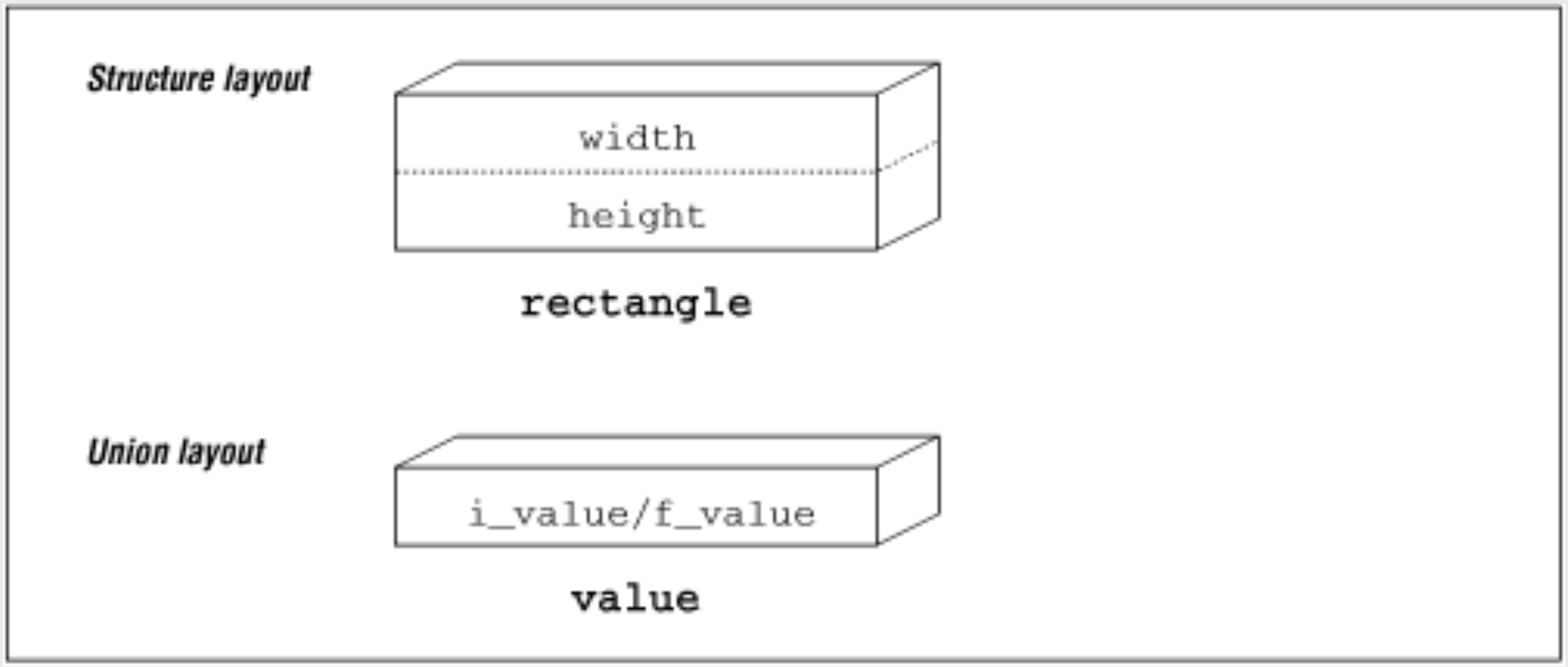
Unions

- Structure
  - Define a data type with several fields. Each field takes up a separate storage location
- Union
  - Define a single location that can be given many different field names

```
union value {
    long int i_value;
    float f_value;
};
```

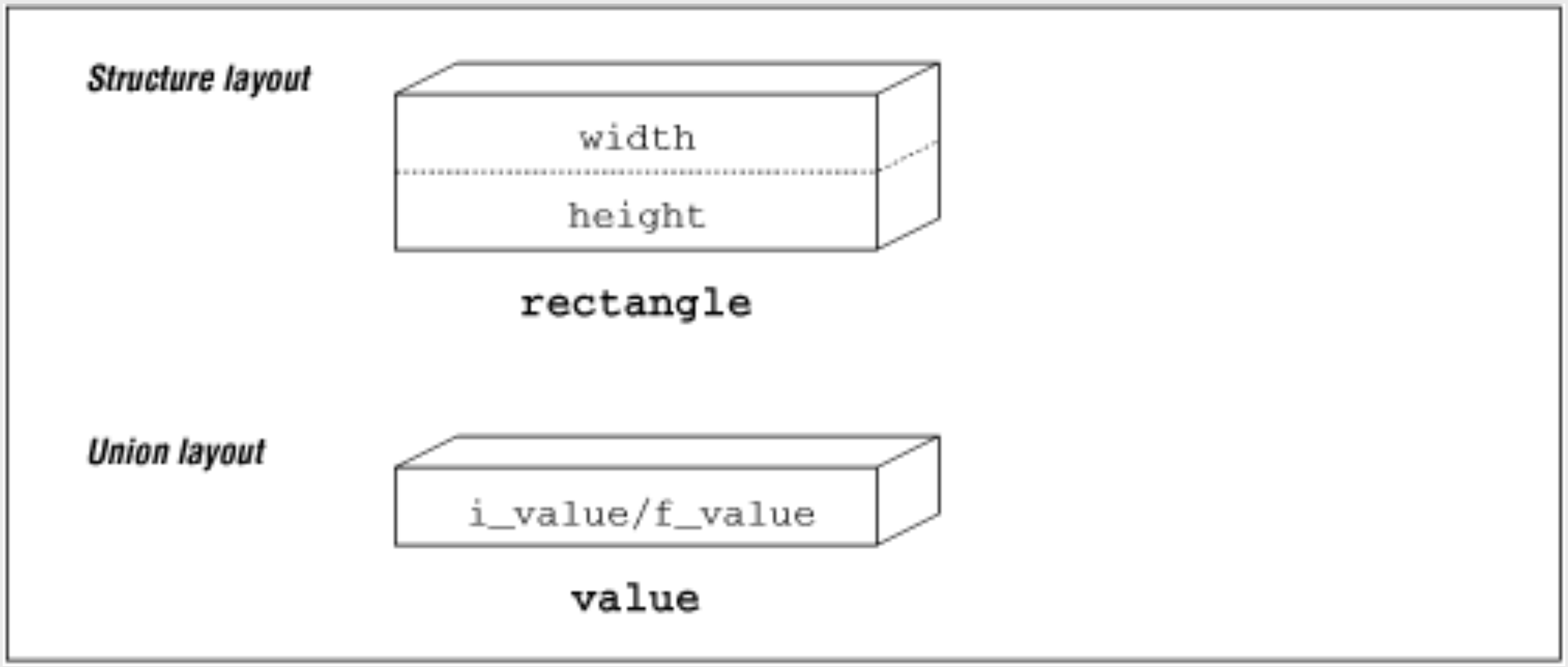
Unions

- Layout of structure and union



Unions

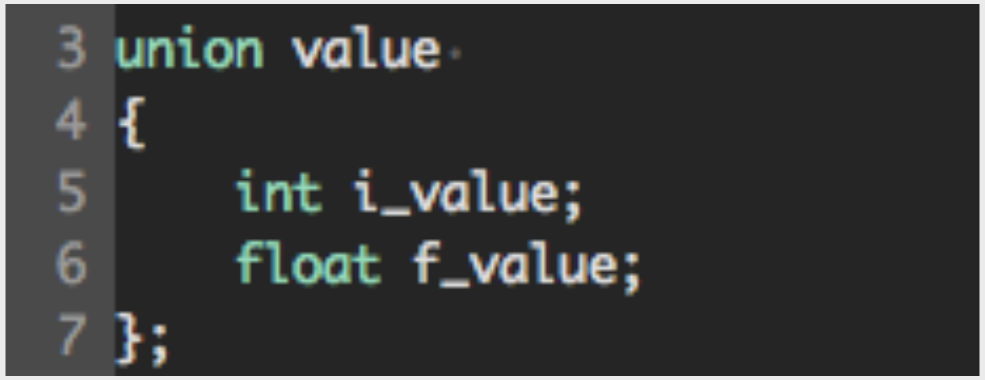
- Layout of structure and union



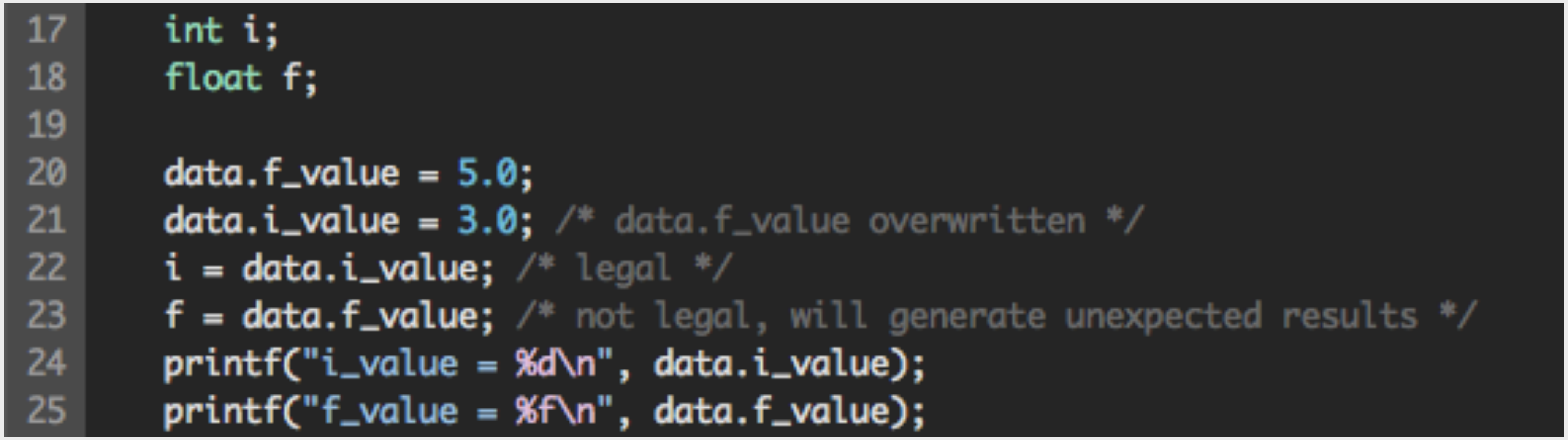
In a union, all fields occupy the same space, so only one may be active at a time.

Unions

- Example: unions.c



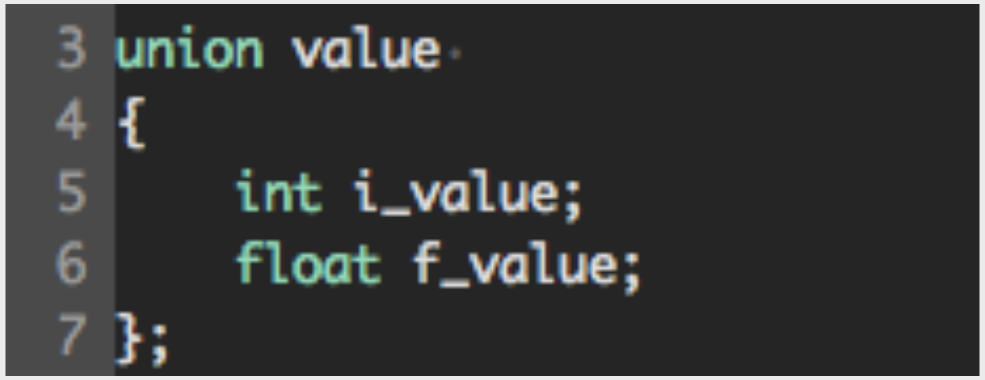
```c
union value
{
    int i_value;
    float f_value;
};
```



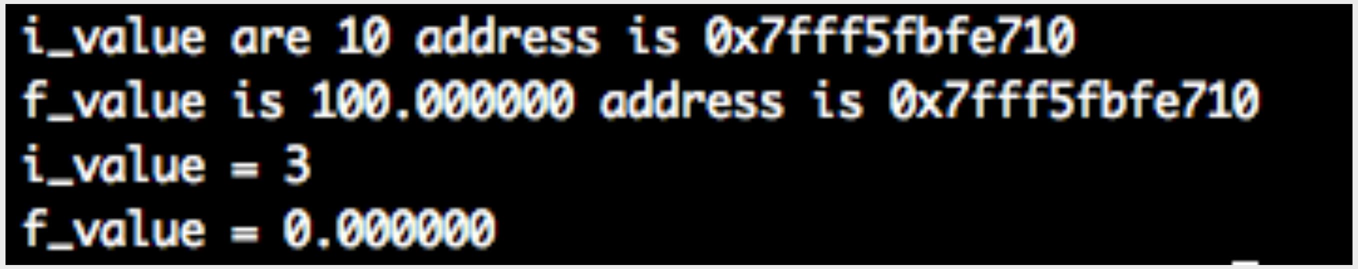
```c
    int i;
    float f;

    data.f_value = 5.0;
    data.i_value = 3.0; /* data.f_value overwritten */
    i = data.i_value; /* legal */
    f = data.f_value; /* not legal, will generate unexpected results */
    printf("i_value = %d\n", data.i_value);
    printf("f_value = %f\n", data.f_value);
```

Unions

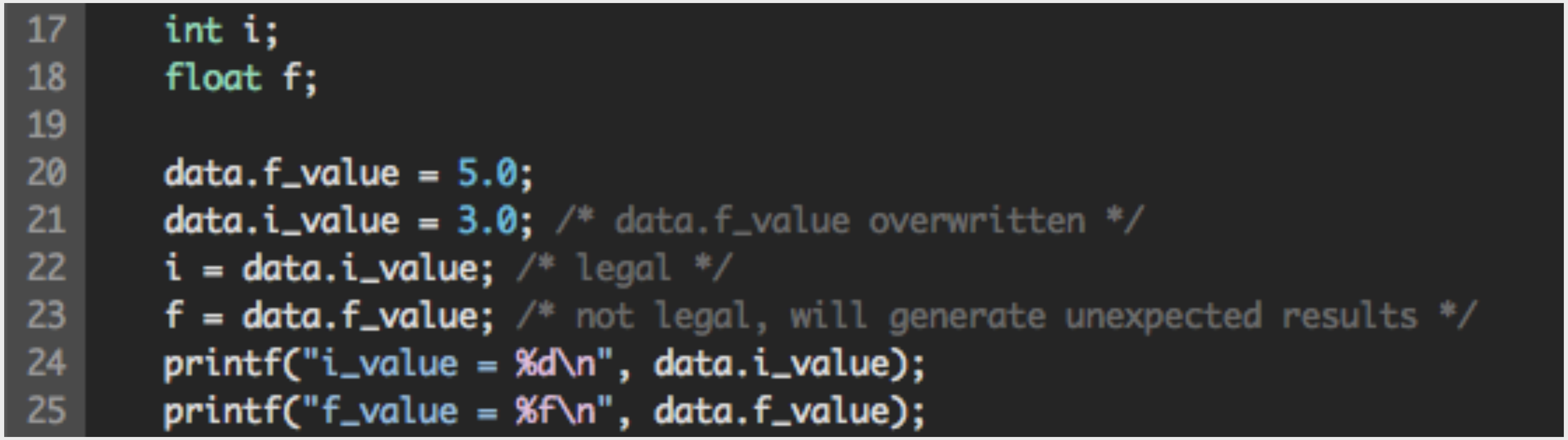
- Example: unions.c



```c
union value
{
    int i_value;
    float f_value;
};
```



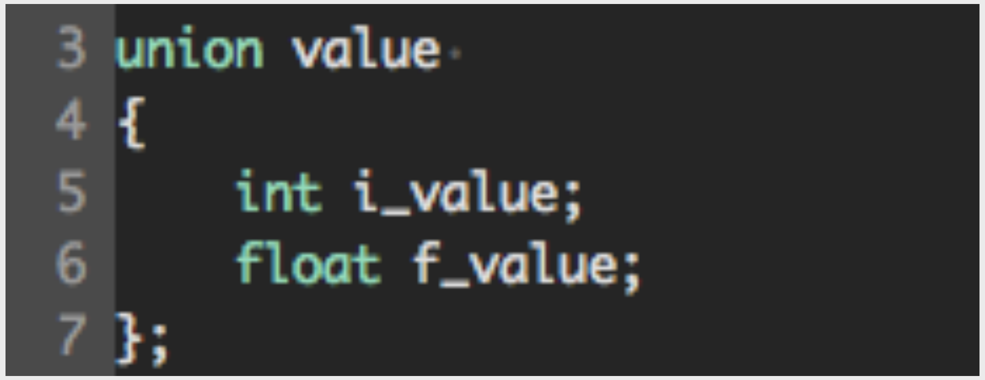
```
i_value are 10 address is 0x7fff5fbfe710
f_value is 100.000000 address is 0x7fff5fbfe710
i_value = 3
f_value = 0.000000
```



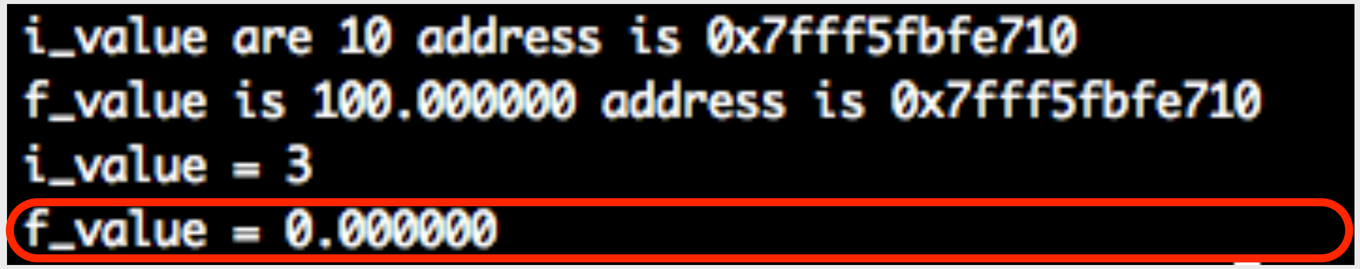
```c
    int i;
    float f;

    data.f_value = 5.0;
    data.i_value = 3.0; /* data.f_value overwritten */
    i = data.i_value; /* legal */
    f = data.f_value; /* not legal, will generate unexpected results */
    printf("i_value = %d\n", data.i_value);
    printf("f_value = %f\n", data.f_value);
```

Unions

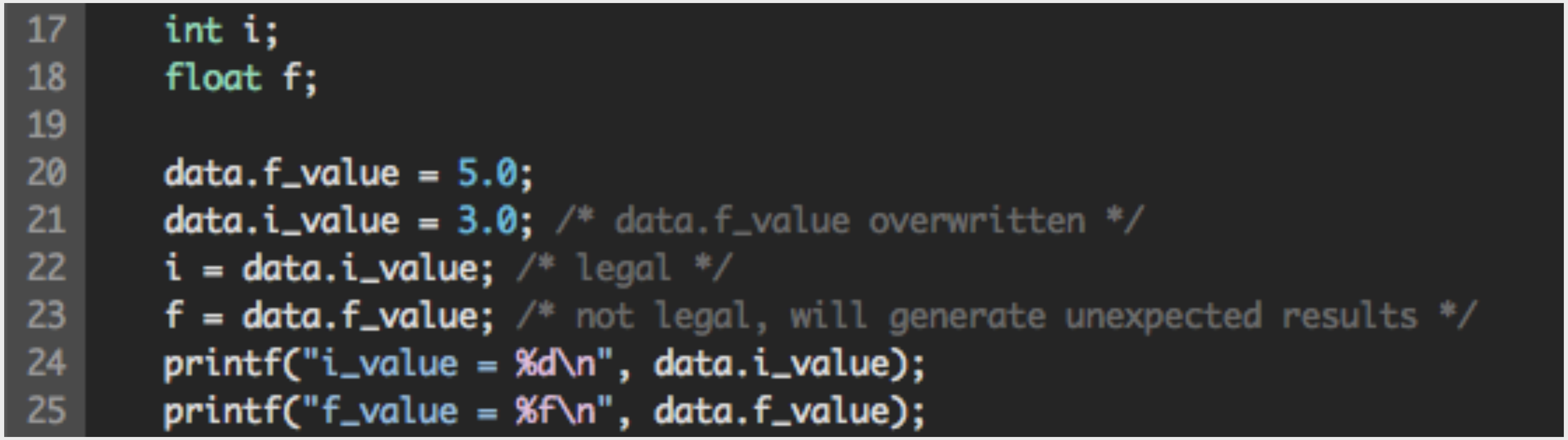
- Example: unions.c



```
union value
{
    int i_value;
    float f_value;
};
```



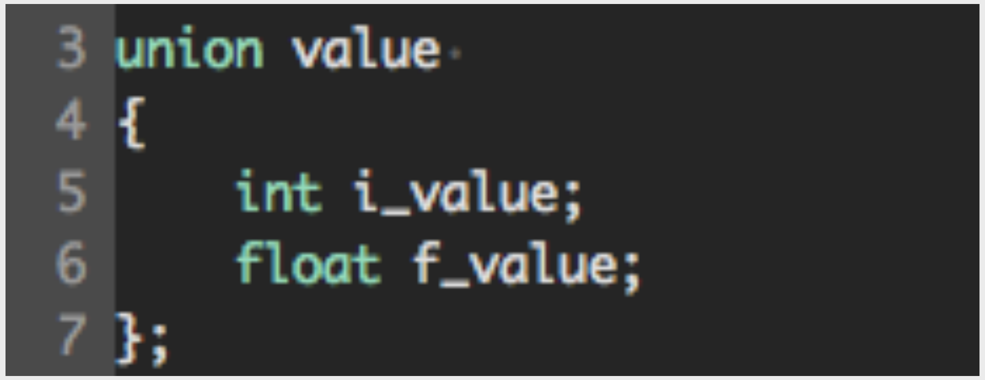
```
i_value are 10 address is 0x7fff5fbfe710
f_value is 100.000000 address is 0x7fff5fbfe710
i_value = 3
f_value = 0.000000
```



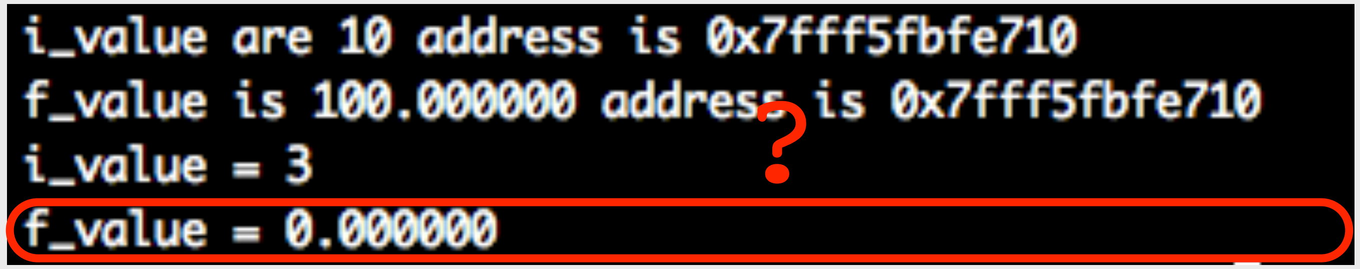
```
    int i;
    float f;

    data.f_value = 5.0;
    data.i_value = 3.0; /* data.f_value overwritten */
    i = data.i_value; /* legal */
    f = data.f_value; /* not legal, will generate unexpected results */
    printf("i_value = %d\n", data.i_value);
    printf("f_value = %f\n", data.f_value);
```

Unions

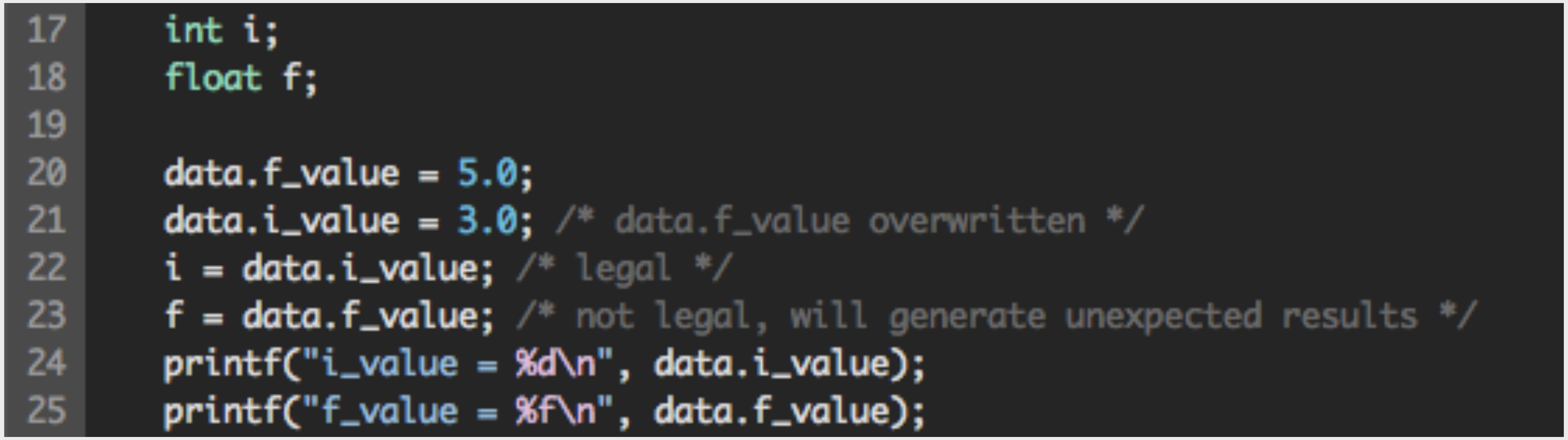
- Example: unions.c



```c
union value
{
    int i_value;
    float f_value;
};
```



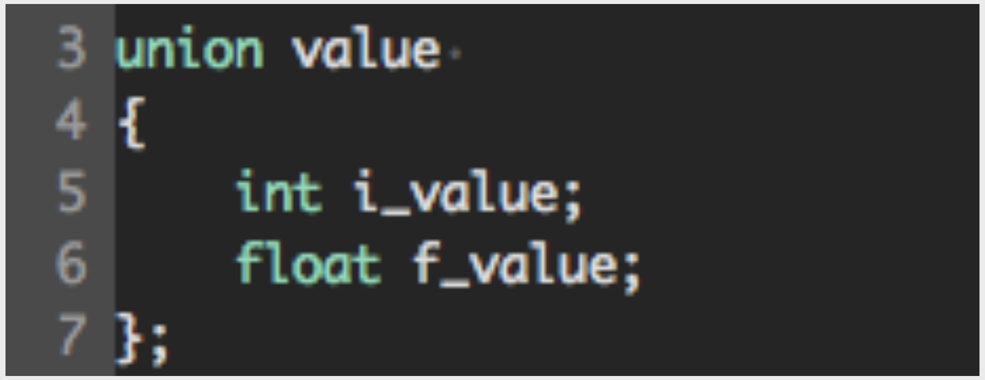
```
i_value are 10 address is 0x7fff5fbfe710
f_value is 100.000000 address is 0x7fff5fbfe710
i_value = 3
f_value = 0.000000
```



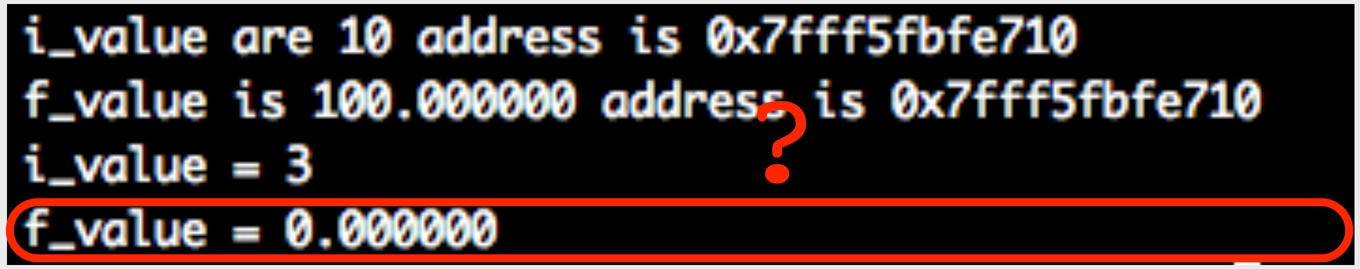
```c
    int i;
    float f;

    data.f_value = 5.0;
    data.i_value = 3.0; /* data.f_value overwritten */
    i = data.i_value; /* legal */
    f = data.f_value; /* not legal, will generate unexpected results */
    printf("i_value = %d\n", data.i_value);
    printf("f_value = %f\n", data.f_value);
```

Unions

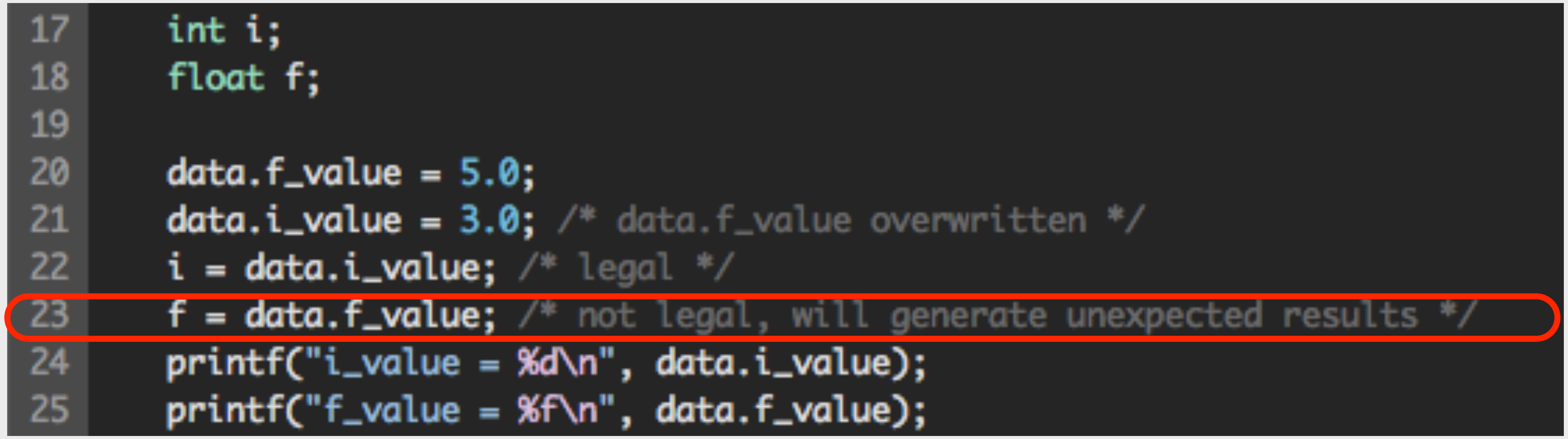
- Example: unions.c



```
union value
{
    int i_value;
    float f_value;
};
```



```
i_value are 10 address is 0x7fff5fbfe710
f_value is 100.000000 address is 0x7fff5fbfe710
i_value = 3
f_value = 0.000000
```



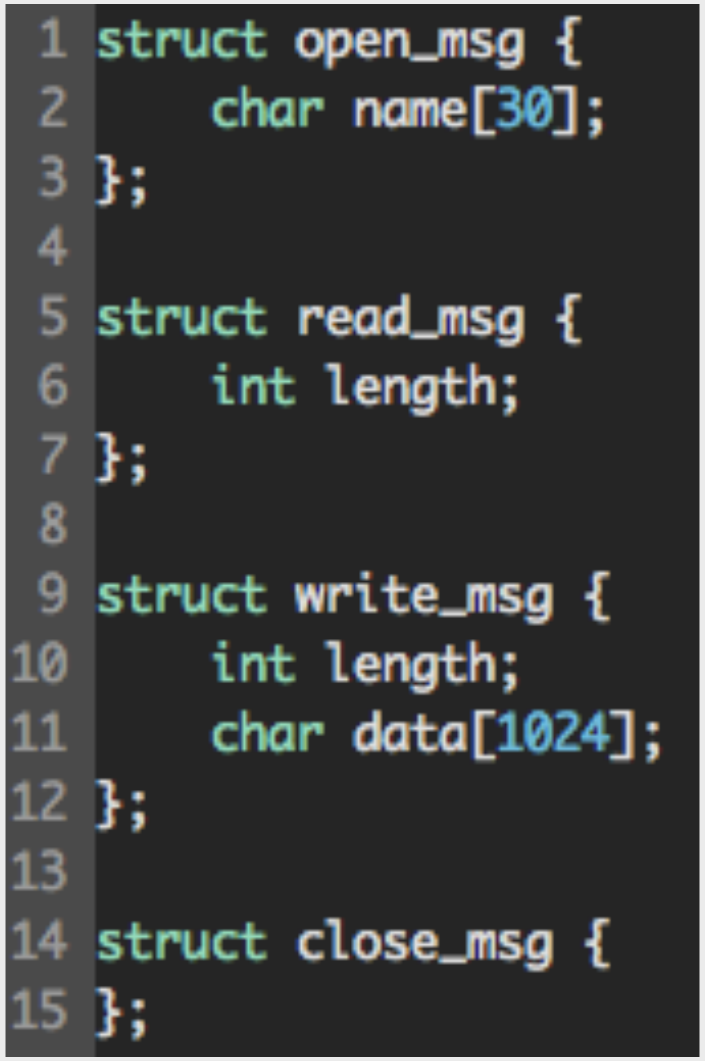
```
    int i;
    float f;

    data.f_value = 5.0;
    data.i_value = 3.0; /* data.f_value overwritten */
    i = data.i_value; /* legal */
    f = data.f_value; /* not legal, will generate unexpected results */
    printf("i_value = %d\n", data.i_value);
    printf("f_value = %f\n", data.f_value);
```

Unions

- Unions are frequently used in the area of communications.

Unions

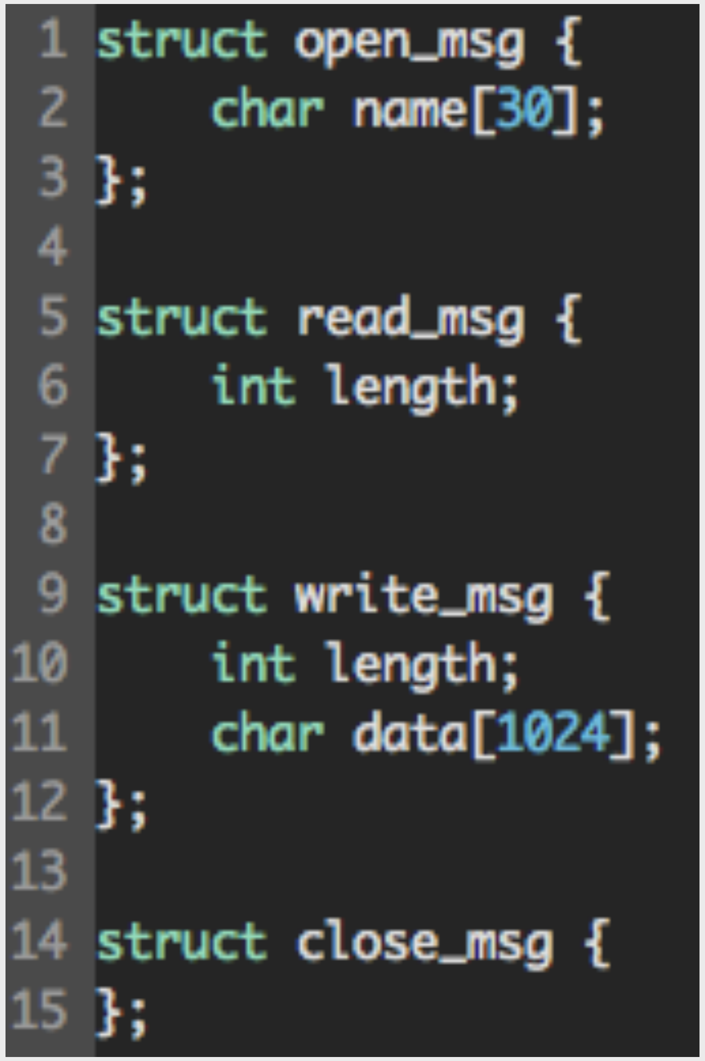
- Unions are frequently used in the area of communications.



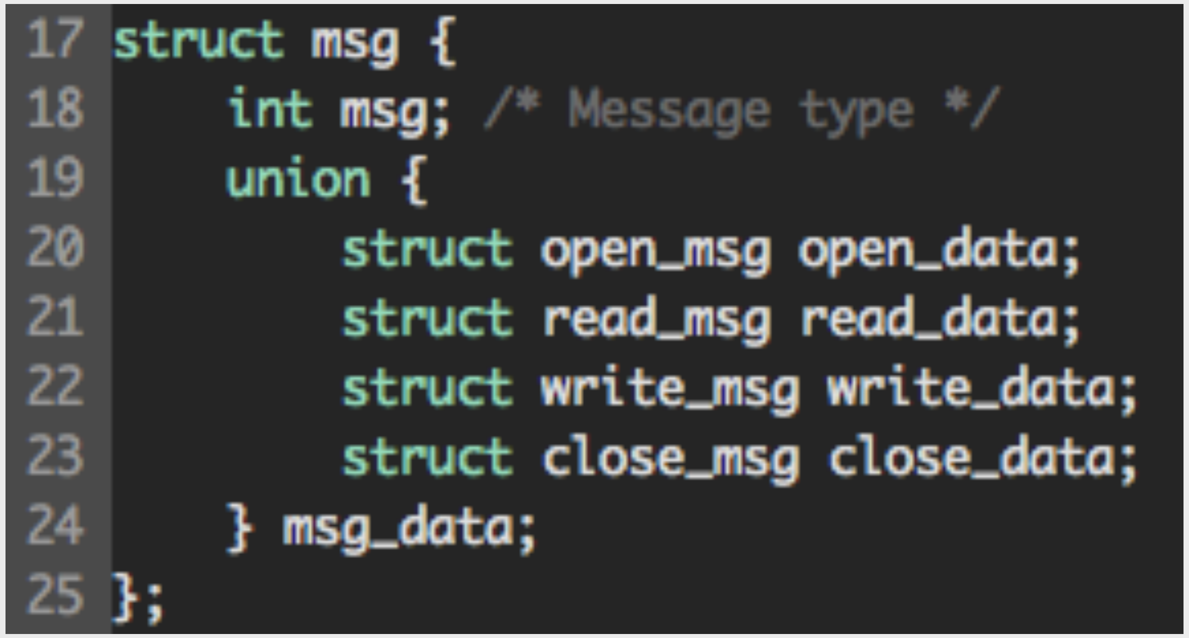
```c
struct open_msg {
    char name[30];
};

struct read_msg {
    int length;
};

struct write_msg {
    int length;
    char data[1024];
};

struct close_msg {
};
```

Unions

- Unions are frequently used in the area of communications.



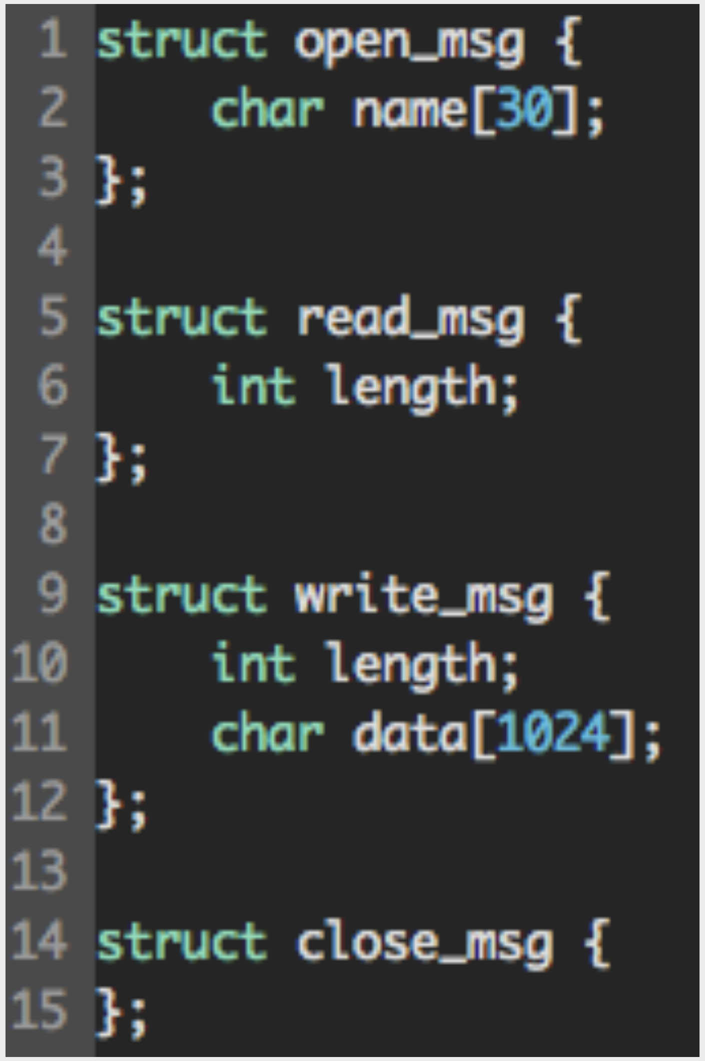
```c
struct open_msg {
    char name[30];
};

struct read_msg {
    int length;
};

struct write_msg {
    int length;
    char data[1024];
};

struct close_msg {
};
```



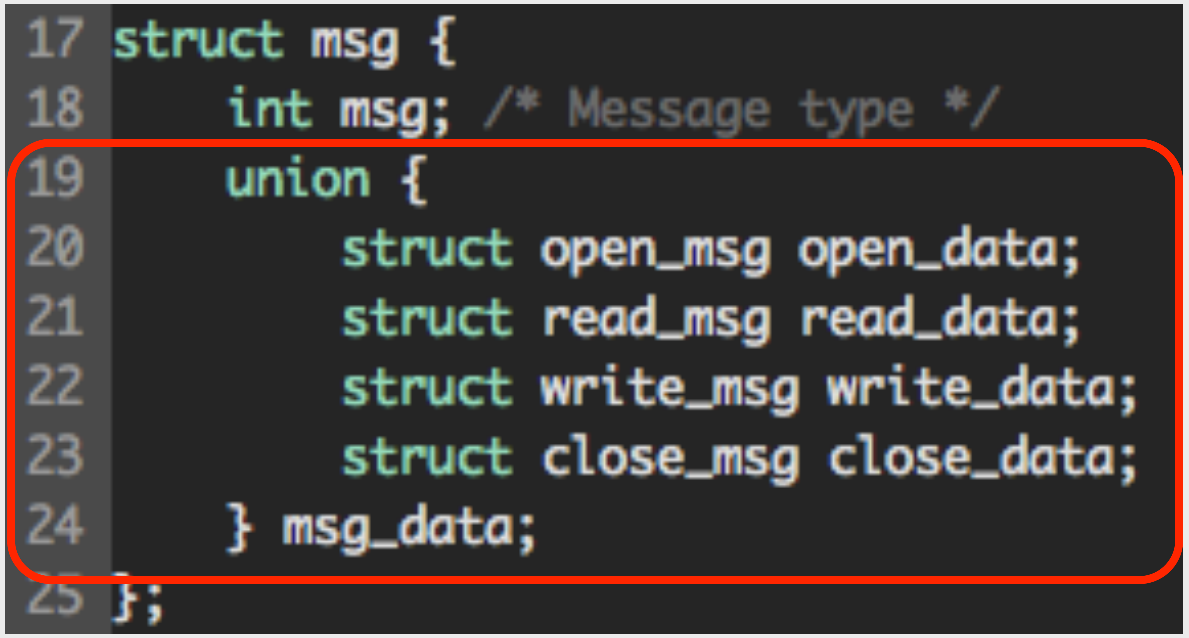
```c
struct msg {
    int msg; /* Message type */
    union {
        struct open_msg open_data;
        struct read_msg read_data;
        struct write_msg write_data;
        struct close_msg close_data;
    } msg_data;
};
```

Unions

- Unions are frequently used in the area of communications.



```c
struct open_msg {
    char name[30];
};

struct read_msg {
    int length;
};

struct write_msg {
    int length;
    char data[1024];
};

struct close_msg {
};
```



```c
struct msg {
    int msg; /* Message type */
    union {
        struct open_msg open_data;
        struct read_msg read_data;
        struct write_msg write_data;
        struct close_msg close_data;
    } msg_data;
};
```

typedef

- Use typedef to define your own variable types

```
typedef type-declaration;
```

```
typedef int count;
count flag; /* equals int flag */
```

```
#define count int
count flag; /* equals int flag */
```

typedef

- **typedef** can be used to define more complex objects that are beyond the scope of **#define**

```
typedef int[10] group;

group totals;
for (i = 0; i < 10, i++)
    totals[i] = 0;
```

```
struct complex_struct {
  double real;
  double imag;
};

typedef struct complex_strcut complex;

complex voltag1 = {3.5, 1.2};
```

enum

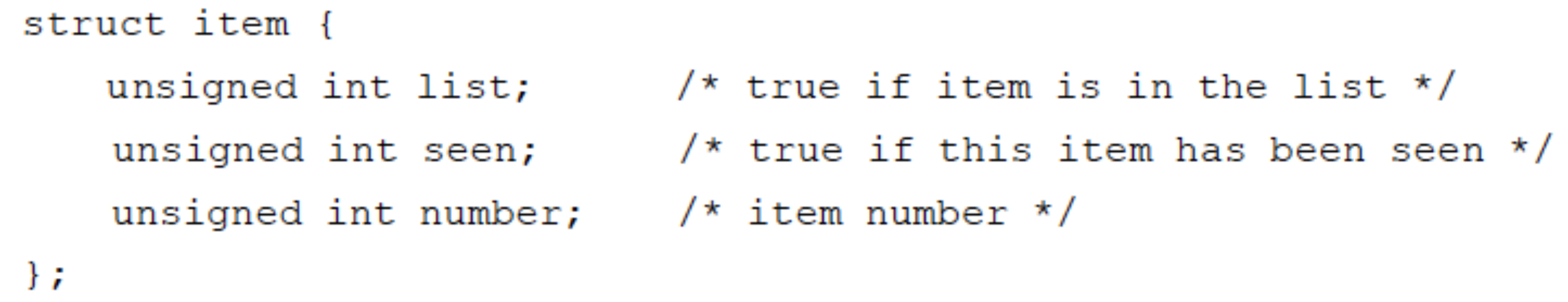
- The enumerated data type is designed for variables that contain only a limited set of values

```
enum enum-name { tag-1, tag-2, ... } variable-name;
```

```
enum week_day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY};

enum week_day today = TUESDAY;
```

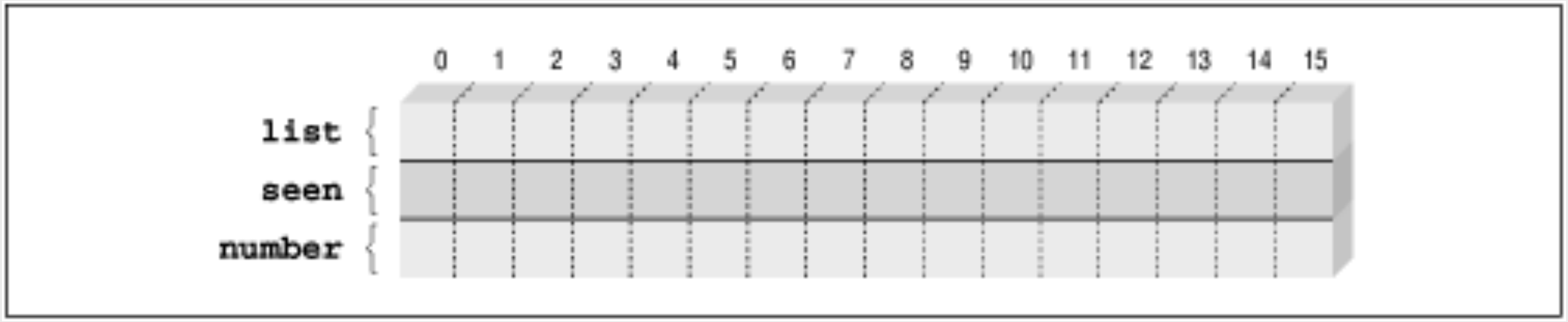
Bit Fields or Packed Structures

- Packed structures allow us to declare structures in a way that takes up a minimum amount of storage



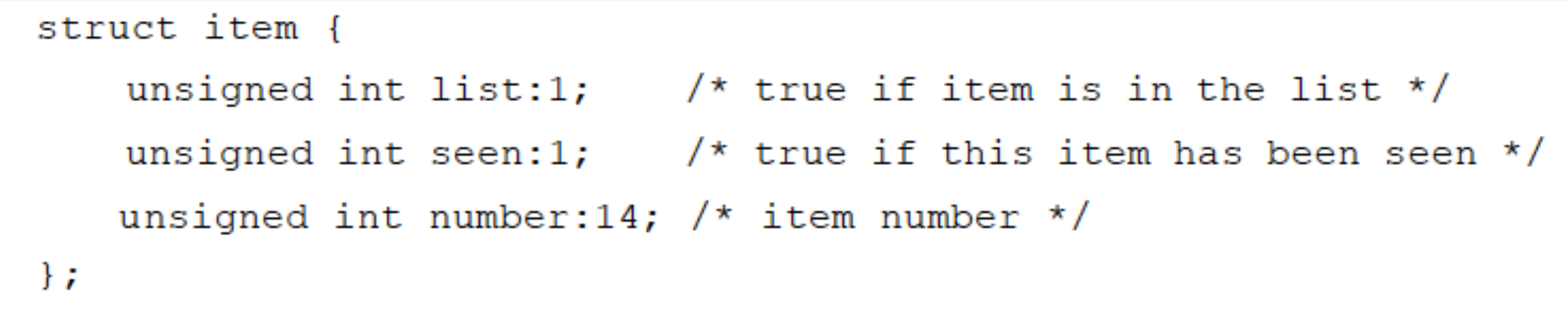
```
struct item {
    unsigned int list;      /* true if item is in the list */
    unsigned int seen;      /* true if this item has been seen */
    unsigned int number;    /* item number */
};
```

Each structure uses six bytes of storage (two bytes for each integer)

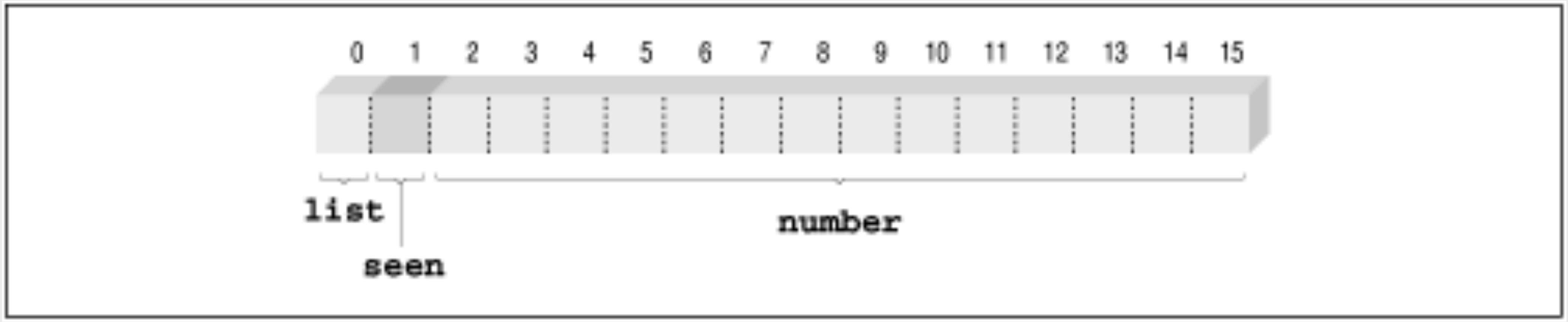


Bit Fields or Packed Structures

- Packed structured

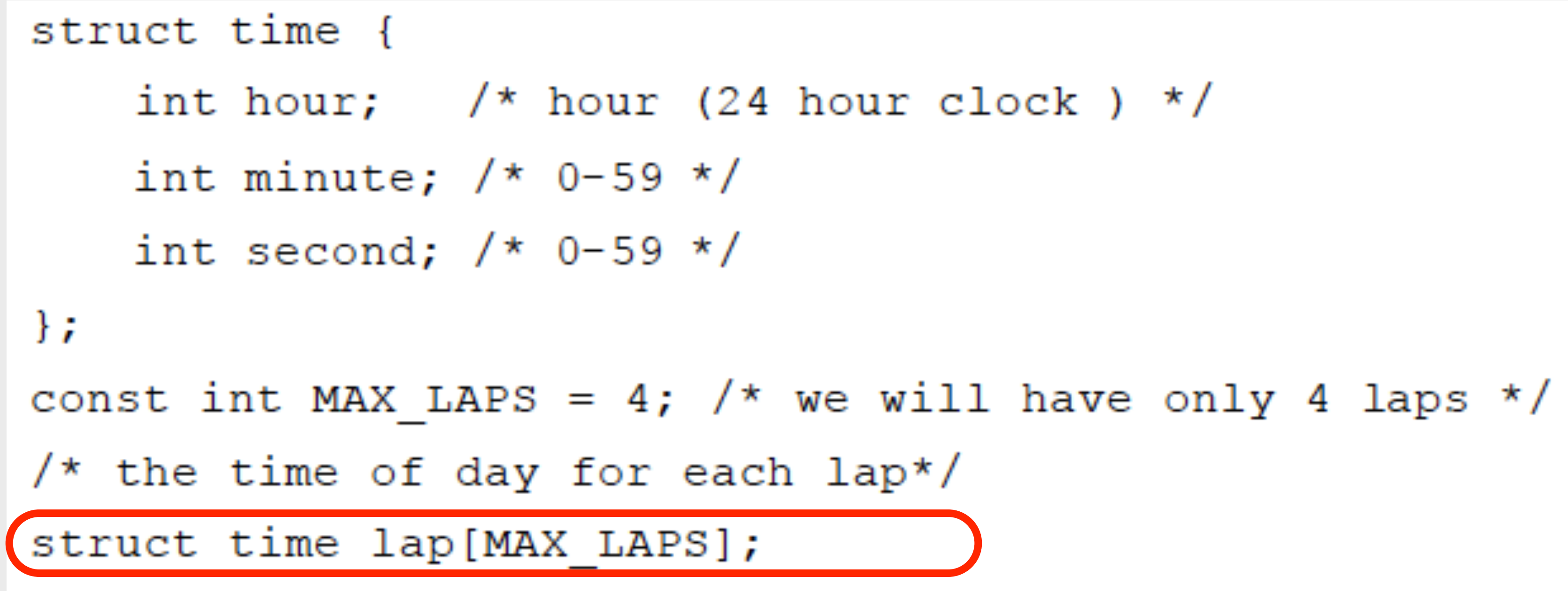


```
struct item {
    unsigned int list:1;    /* true if item is in the list */
    unsigned int seen:1;    /* true if this item has been seen */
   unsigned int number:14; /* item number */
};
```

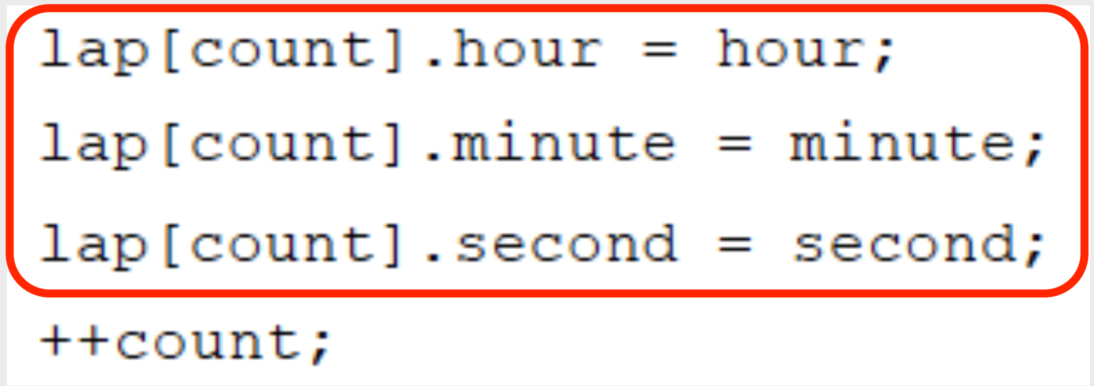


Arrays of Structures

- Initialize arrays of structures



```
struct time {
    int hour;   /* hour (24 hour clock ) */
    int minute; /* 0-59 */
    int second; /* 0-59 */
};
const int MAX_LAPS = 4; /* we will have only 4 laps */
/* the time of day for each lap*/
struct time lap[MAX_LAPS];
```



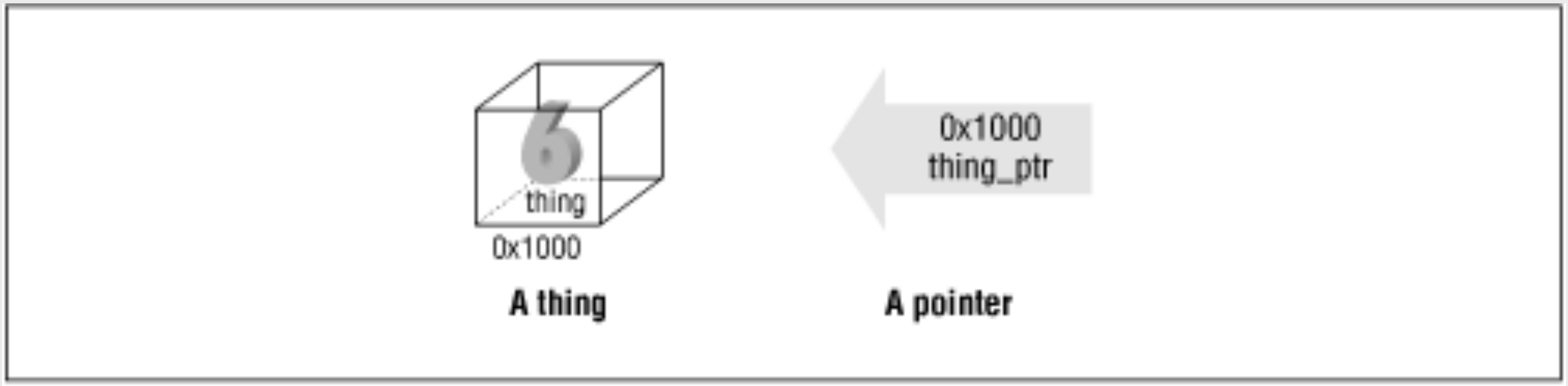
```
lap[count].hour = hour;
lap[count].minute = minute;
lap[count].second = second;
++count;
```

```
Initialization:

struct time
start_stop[2] = {
    {10, 0, 0},
    {12, 0, 0}
};
```

Simple Pointers

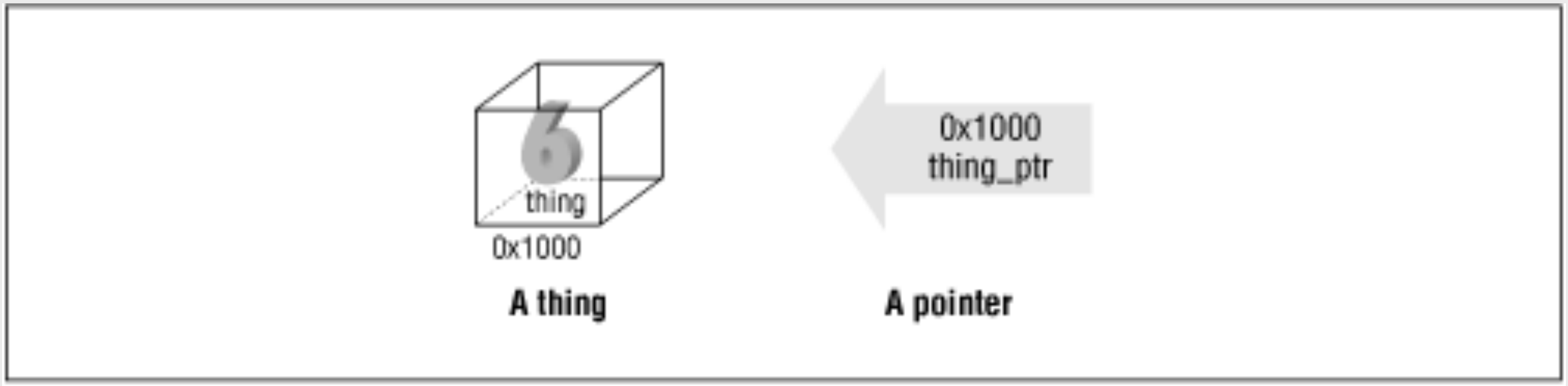
Variables vs. Pointers

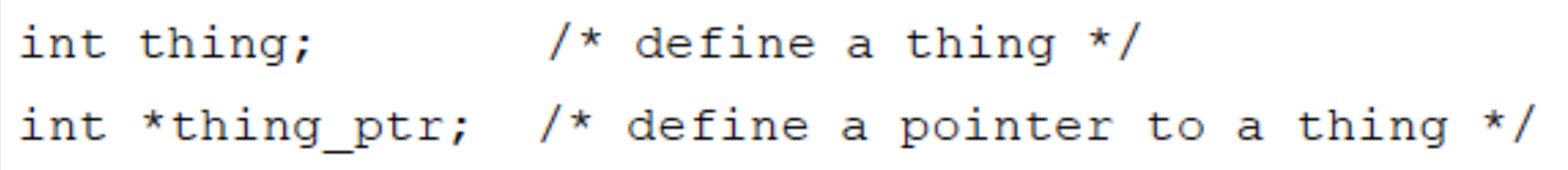
- A thing and a pointer to a thing



Variables vs. Pointers

- A thing and a pointer to a thing





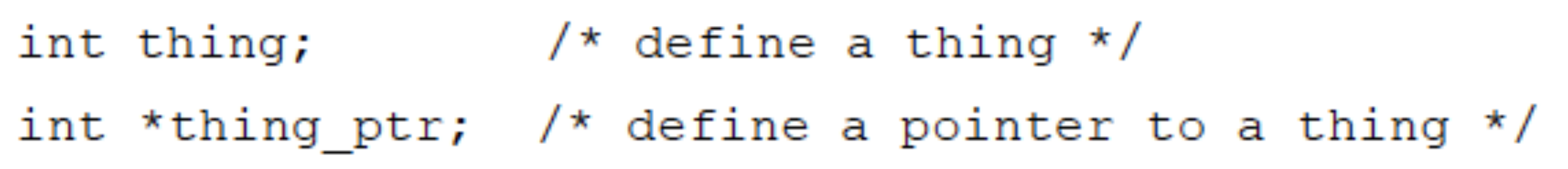
```
int thing;        /* define a thing */
int *thing_ptr;  /* define a pointer to a thing */
```

Pointers

- Pointer declaration

Pointers

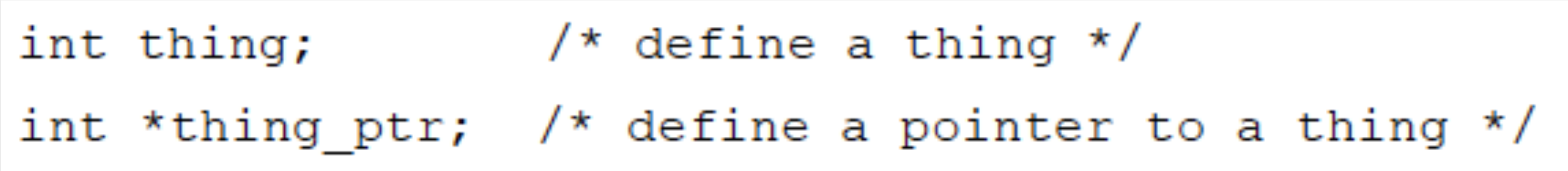
- Pointer declaration



```
int thing;        /* define a thing */
int *thing_ptr;   /* define a pointer to a thing */
```

Pointers

- Pointer declaration

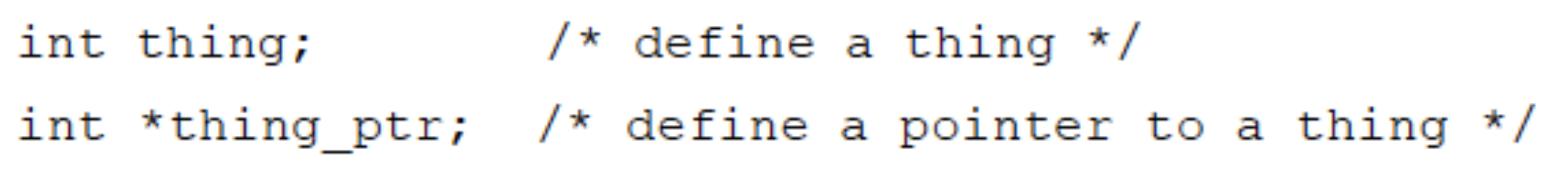


```
int thing;         /* define a thing */
int *thing_ptr;   /* define a pointer to a thing */
```

- Pointer operations

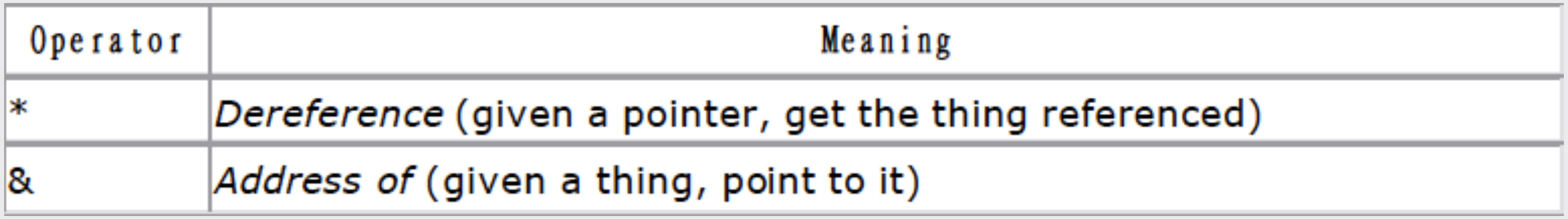
Pointers

- Pointer declaration



```
int thing;          /* define a thing */
int *thing_ptr;    /* define a pointer to a thing */
```

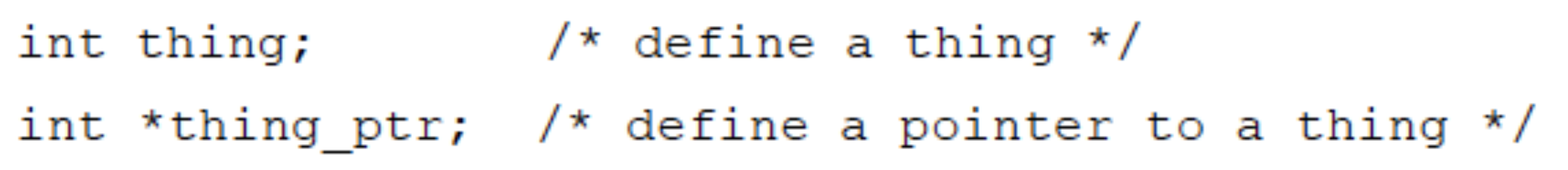
- Pointer operations



Operator	Meaning
*	*Dereference* (given a pointer, get the thing referenced)
&	*Address of* (given a thing, point to it)

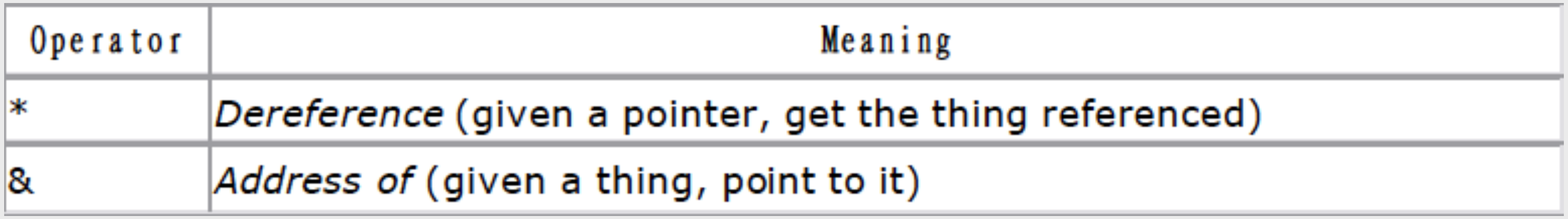
Pointers

- Pointer declaration

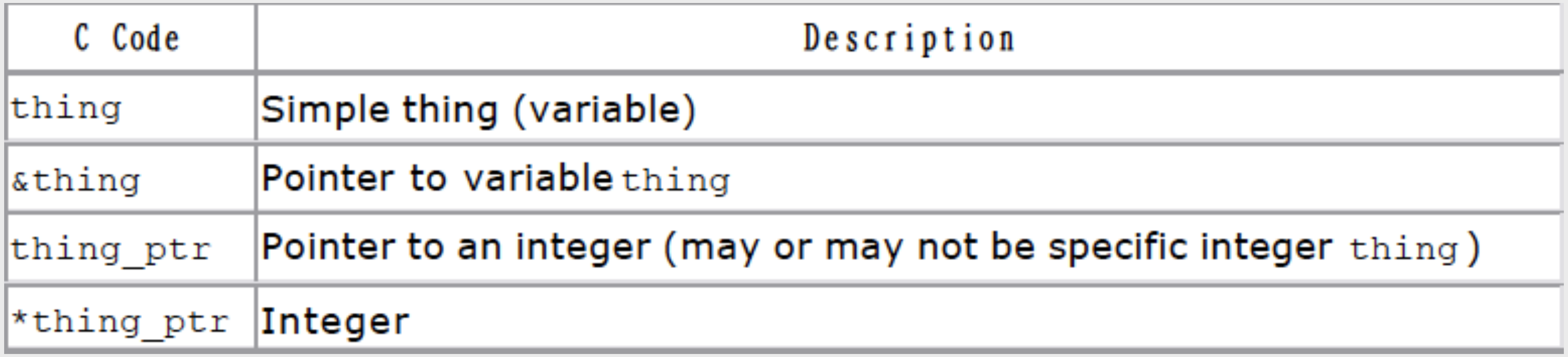


```
int thing;          /* define a thing */
int *thing_ptr;   /* define a pointer to a thing */
```

- Pointer operations

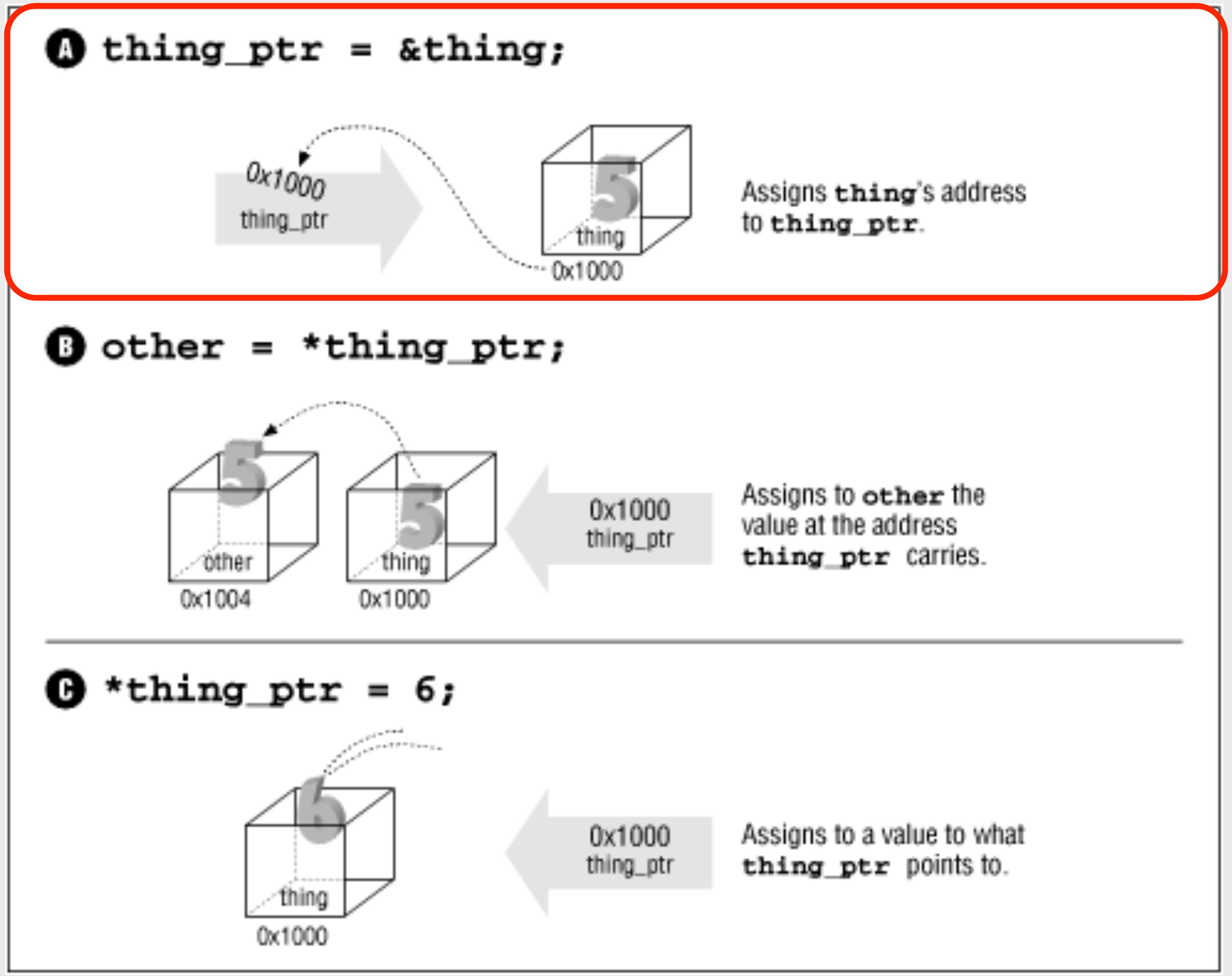


Operator	Meaning
*	*Dereference* (given a pointer, get the thing referenced)
&	*Address of* (given a thing, point to it)



C Code	Description
`thing`	Simple thing (variable)
`&thing`	Pointer to variable `thing`
`thing_ptr`	Pointer to an integer (may or may not be specific integer `thing`)
`*thing_ptr`	Integer

Pointer Operations



A `thing_ptr = &thing;`

Assigns **thing**'s address to **thing_ptr**.

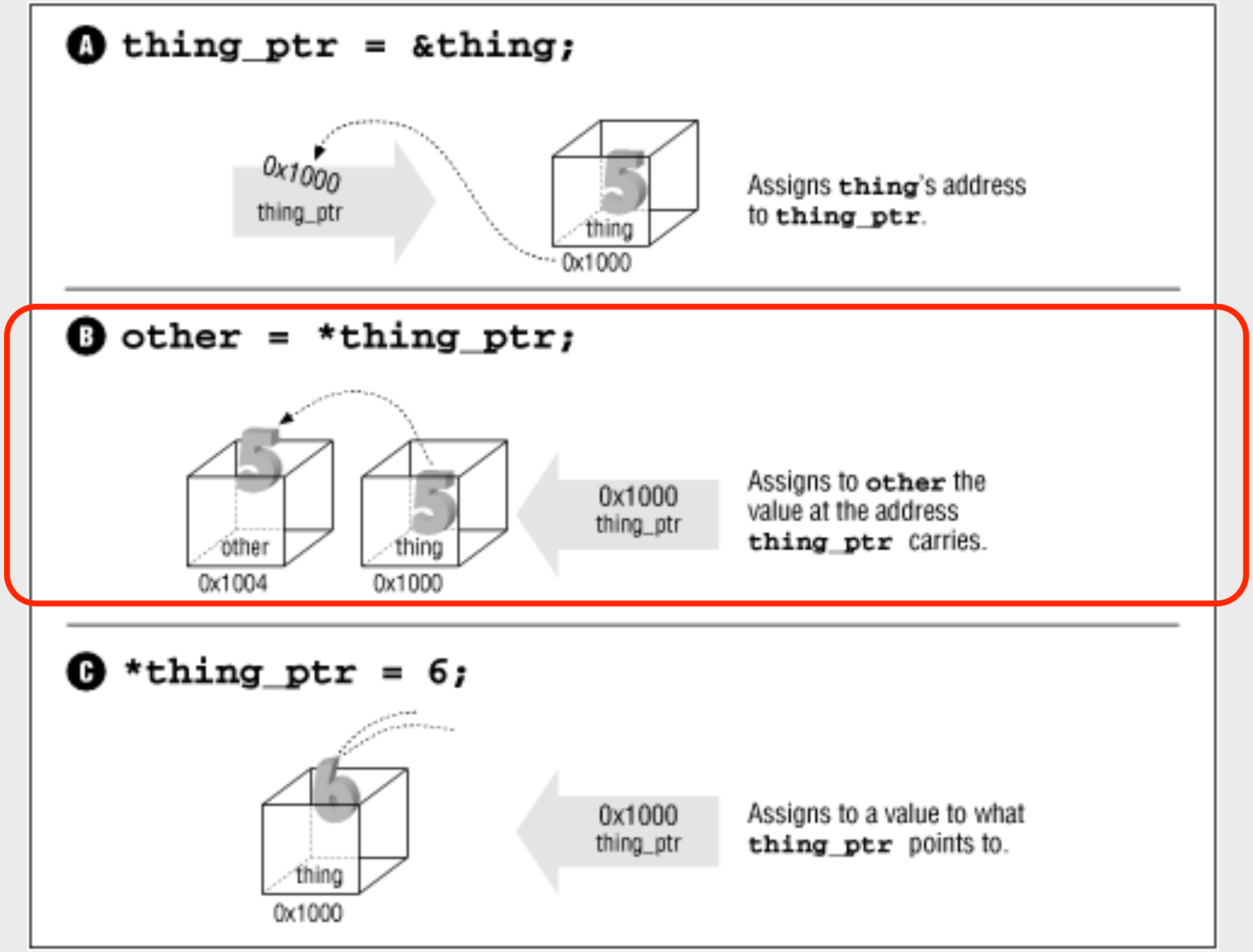
B `other = *thing_ptr;`

Assigns to **other** the value at the address **thing_ptr** carries.

C `*thing_ptr = 6;`

Assigns to a value to what **thing_ptr** points to.

Pointer Operations



```
thing_ptr = &thing;
```

Assigns **thing**'s address to **thing_ptr**.

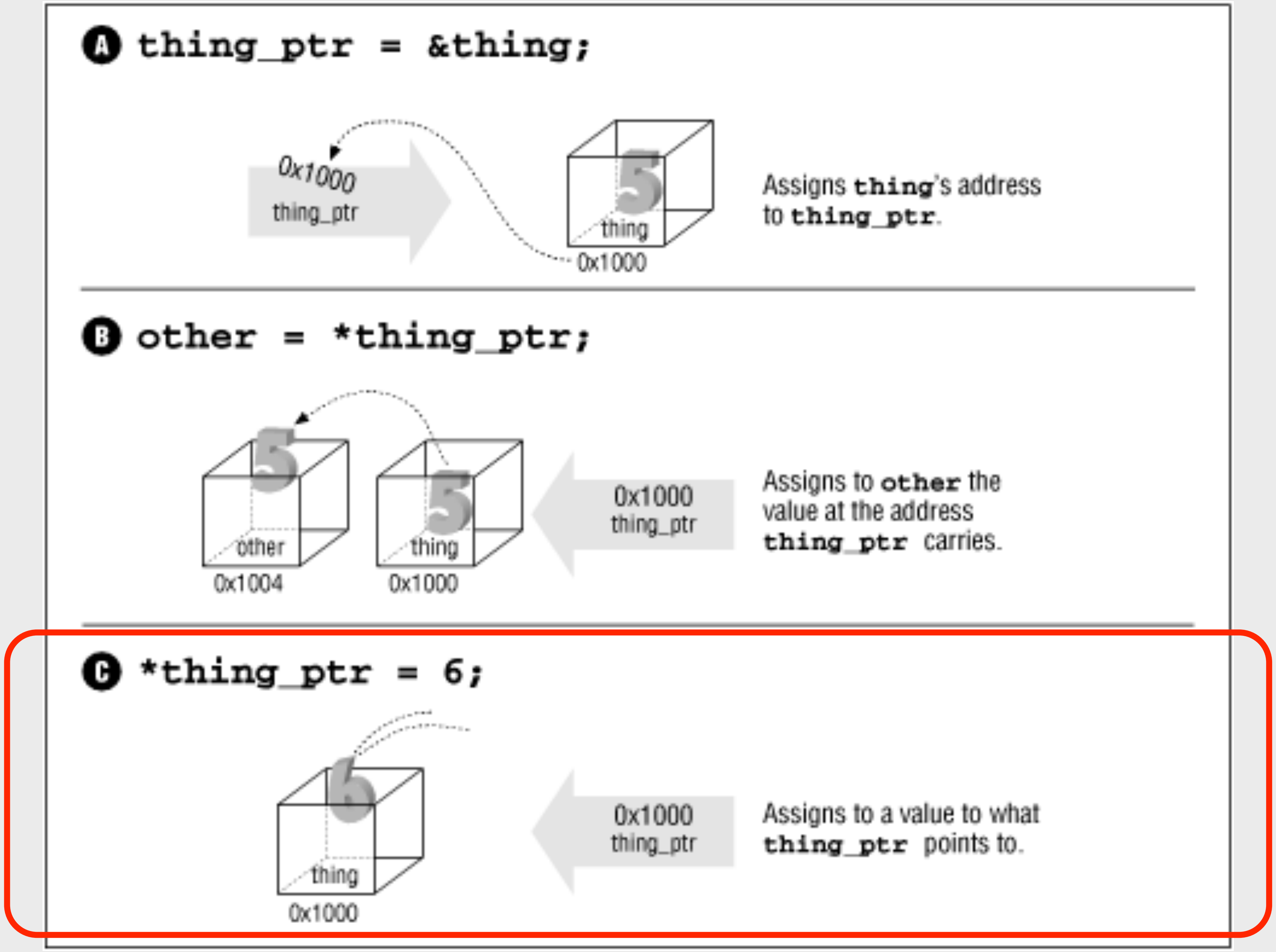
```
other = *thing_ptr;
```

Assigns to **other** the value at the address **thing_ptr** carries.

```
*thing_ptr = 6;
```

Assigns to a value to what **thing_ptr** points to.

Pointer Operations



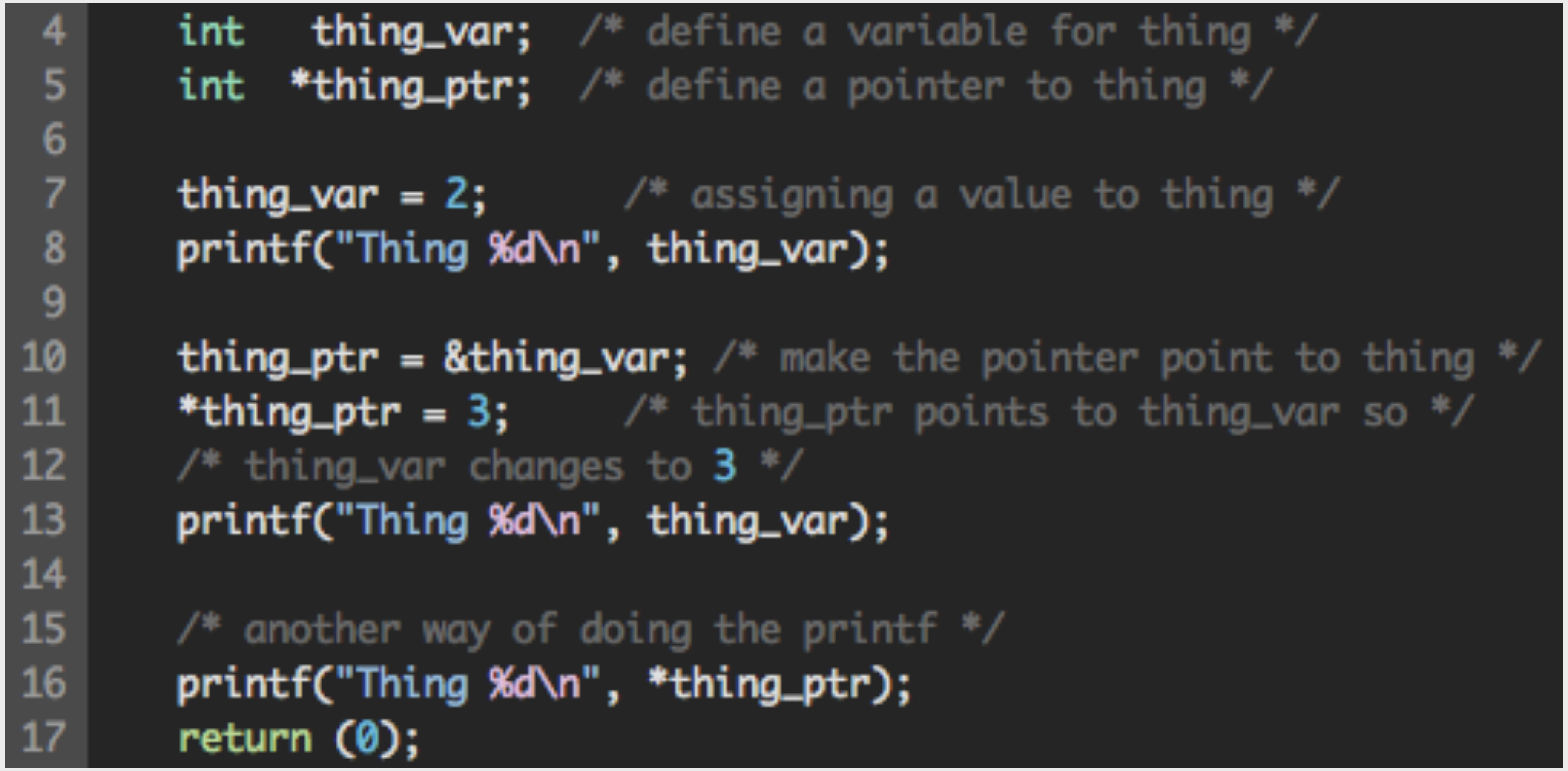
A
```
thing_ptr = &thing;
```
Assigns **thing**'s address to **thing_ptr**.

B
```
other = *thing_ptr;
```
Assigns to **other** the value at the address **thing_ptr** carries.

C
```
*thing_ptr = 6;
```
Assigns to a value to what **thing_ptr** points to.

Pointer Operations

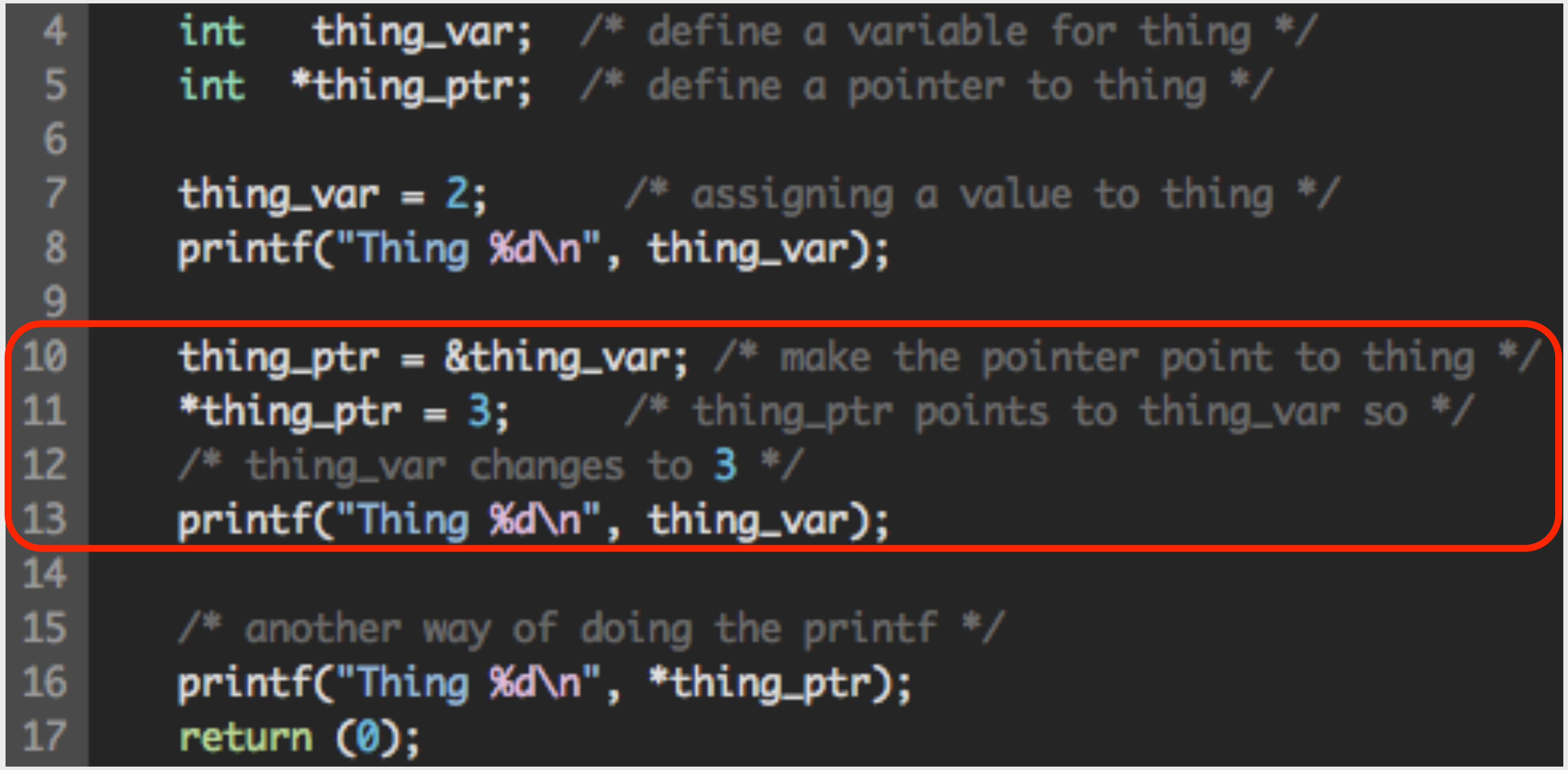
- Example: thing.c



```c
    int   thing_var;  /* define a variable for thing */
    int  *thing_ptr;  /* define a pointer to thing */

    thing_var = 2;      /* assigning a value to thing */
    printf("Thing %d\n", thing_var);

    thing_ptr = &thing_var; /* make the pointer point to thing */
    *thing_ptr = 3;     /* thing_ptr points to thing_var so */
    /* thing_var changes to 3 */
    printf("Thing %d\n", thing_var);

    /* another way of doing the printf */
    printf("Thing %d\n", *thing_ptr);
    return (0);
```

Pointer Operations

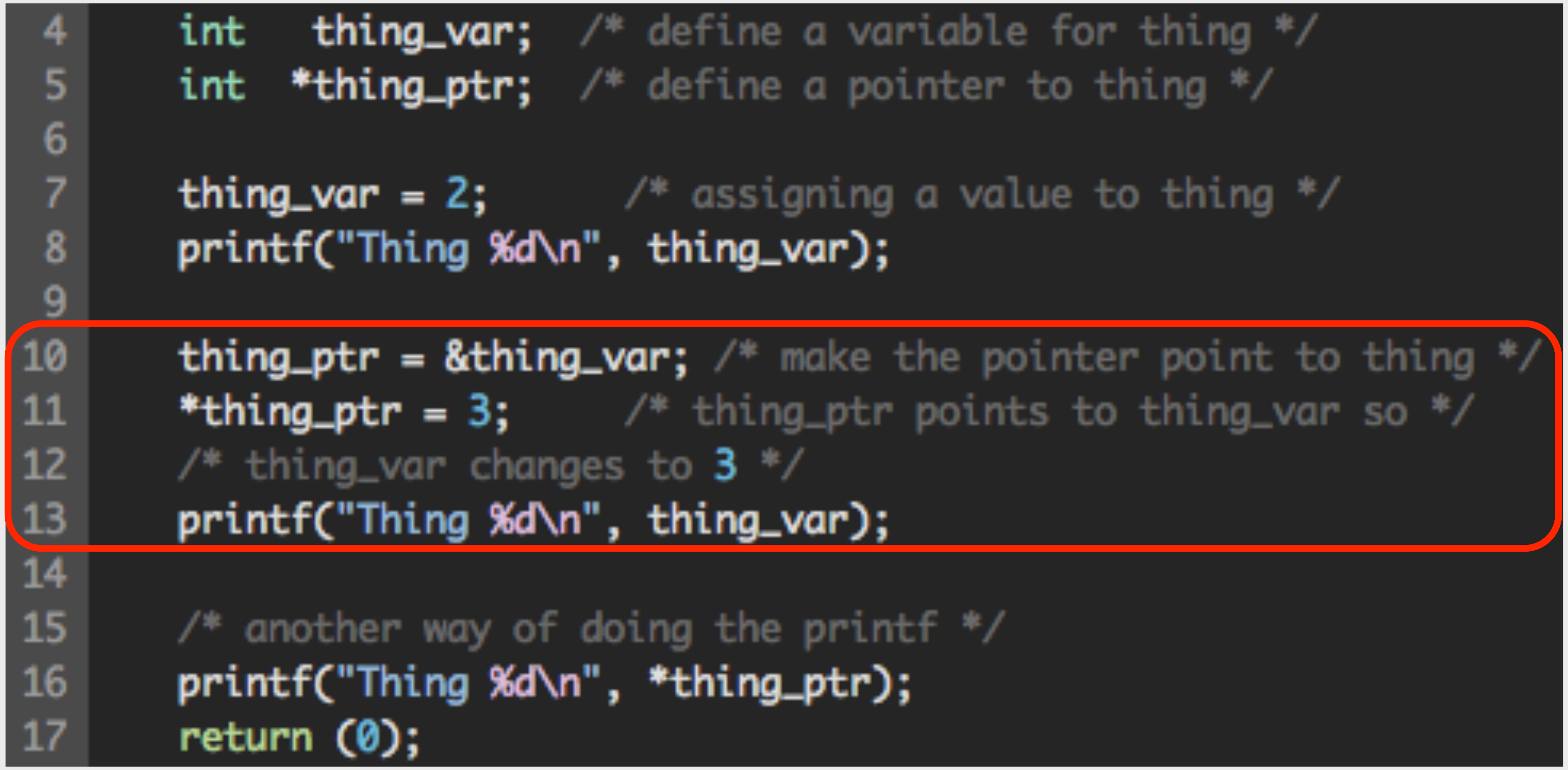
- Example: thing.c



```
    int   thing_var;  /* define a variable for thing */
    int  *thing_ptr;  /* define a pointer to thing */

    thing_var = 2;      /* assigning a value to thing */
    printf("Thing %d\n", thing_var);

    thing_ptr = &thing_var; /* make the pointer point to thing */
    *thing_ptr = 3;     /* thing_ptr points to thing_var so */
    /* thing_var changes to 3 */
    printf("Thing %d\n", thing_var);

    /* another way of doing the printf */
    printf("Thing %d\n", *thing_ptr);
    return (0);
```

Pointer Operations

- Example: thing.c

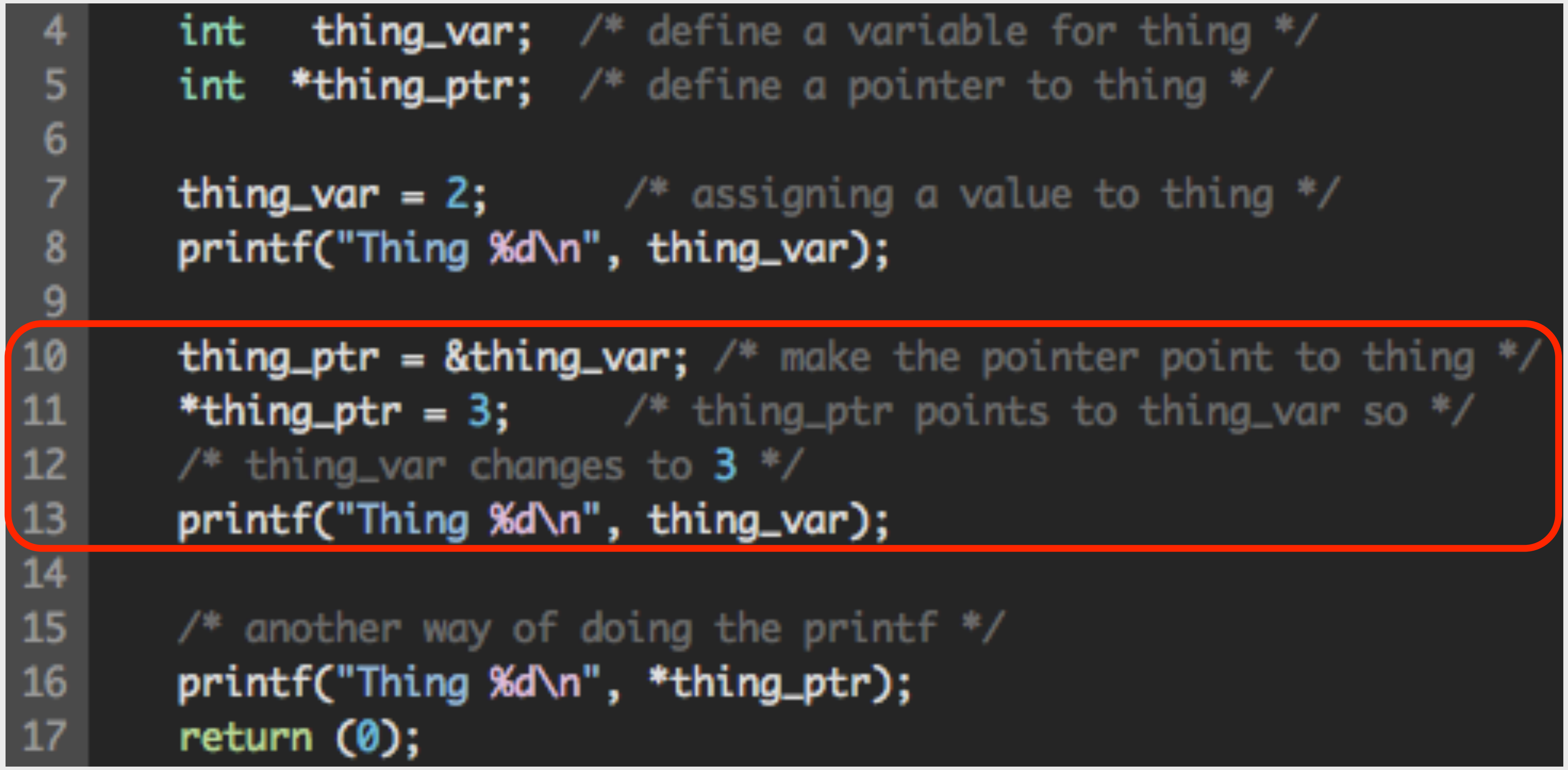


```c
int   thing_var;  /* define a variable for thing */
int  *thing_ptr;  /* define a pointer to thing */

thing_var = 2;      /* assigning a value to thing */
printf("Thing %d\n", thing_var);

thing_ptr = &thing_var; /* make the pointer point to thing */
*thing_ptr = 3;     /* thing_ptr points to thing_var so */
/* thing_var changes to 3 */
printf("Thing %d\n", thing_var);

/* another way of doing the printf */
printf("Thing %d\n", *thing_ptr);
return (0);
```

Output

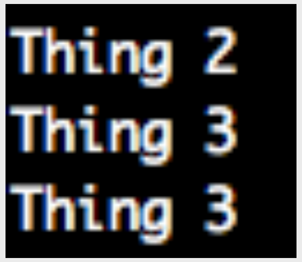
Pointer Operations

- Example: thing.c



```c
    int   thing_var;  /* define a variable for thing */
    int  *thing_ptr;  /* define a pointer to thing */

    thing_var = 2;       /* assigning a value to thing */
    printf("Thing %d\n", thing_var);

    thing_ptr = &thing_var; /* make the pointer point to thing */
    *thing_ptr = 3;      /* thing_ptr points to thing_var so */
    /* thing_var changes to 3 */
    printf("Thing %d\n", thing_var);

    /* another way of doing the printf */
    printf("Thing %d\n", *thing_ptr);
    return (0);
```

Output



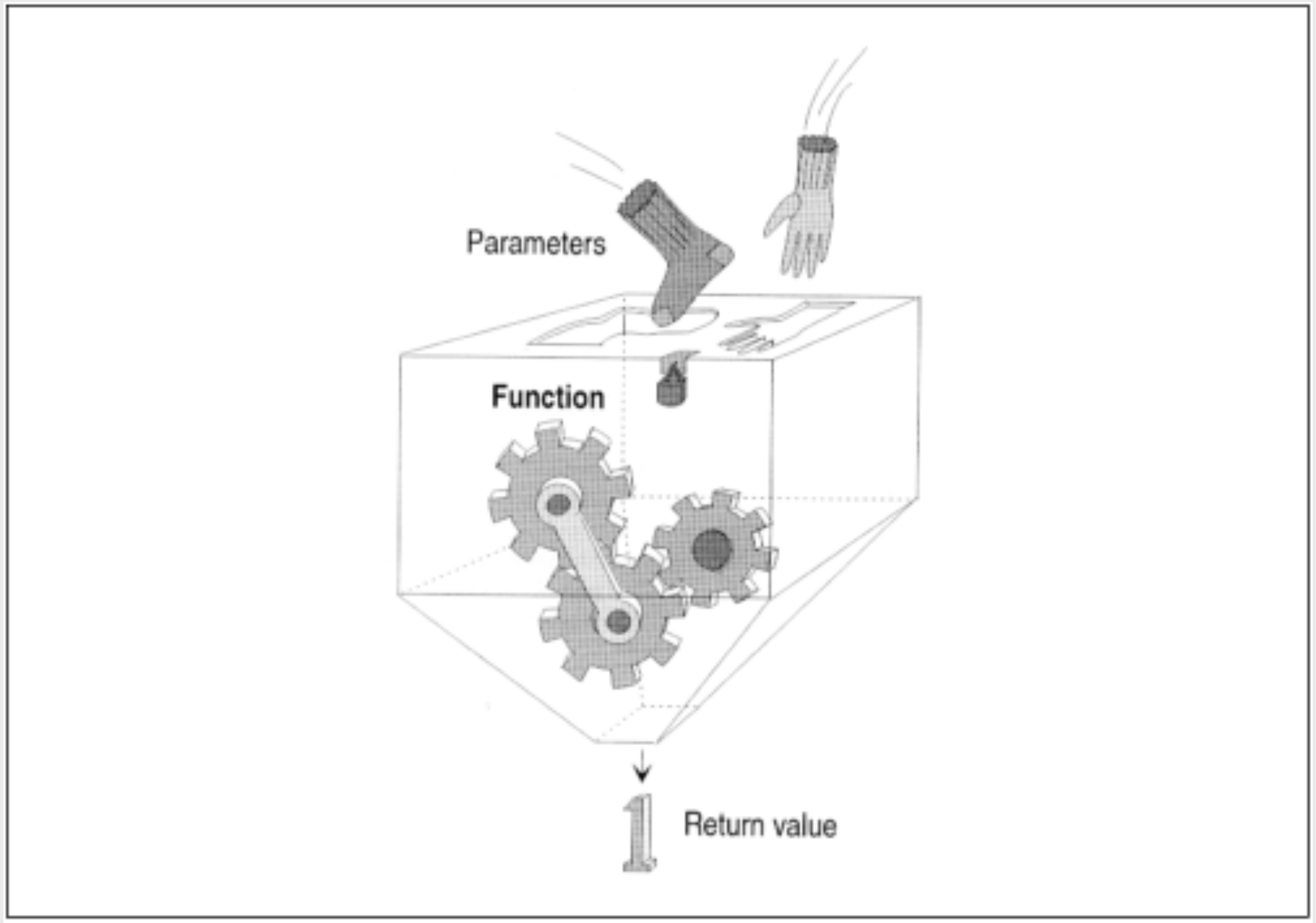
```
Thing 2
Thing 3
Thing 3
```

Pointer as Function Arguments

- C passes parameters to functions via “call by value”

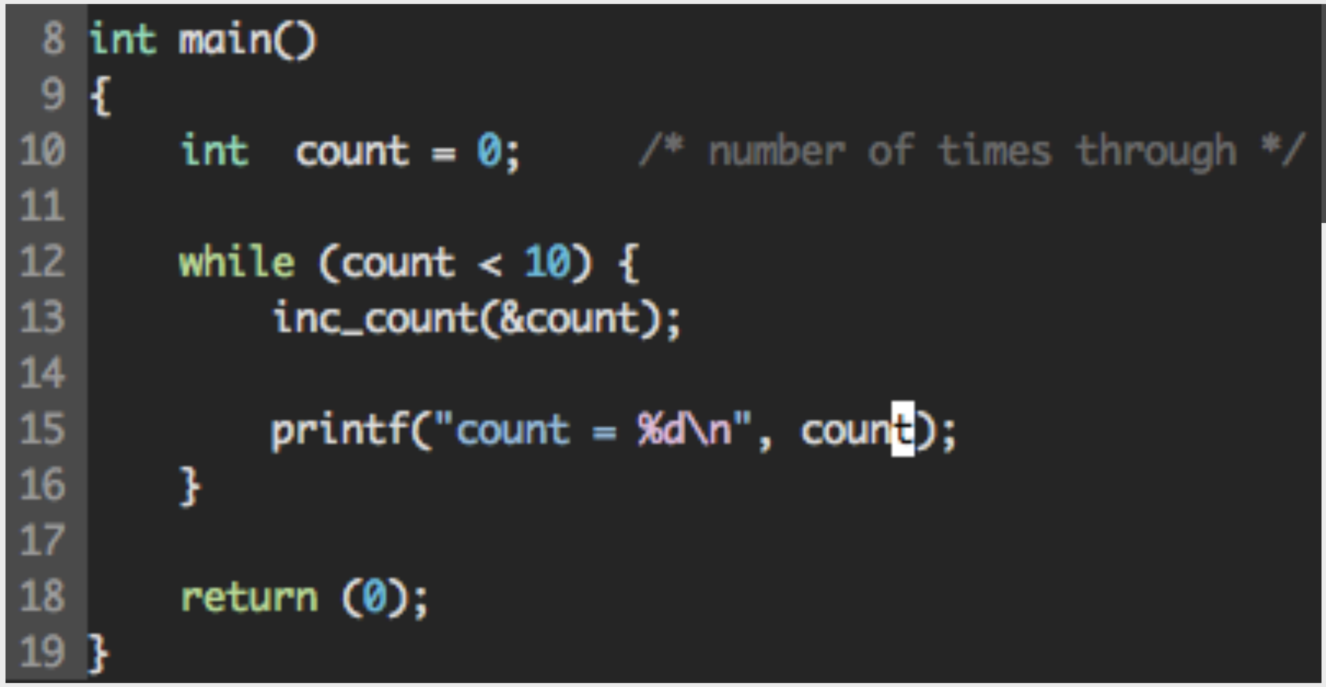
Pointer as Function Arguments

- C passes parameters to functions via “call by value”

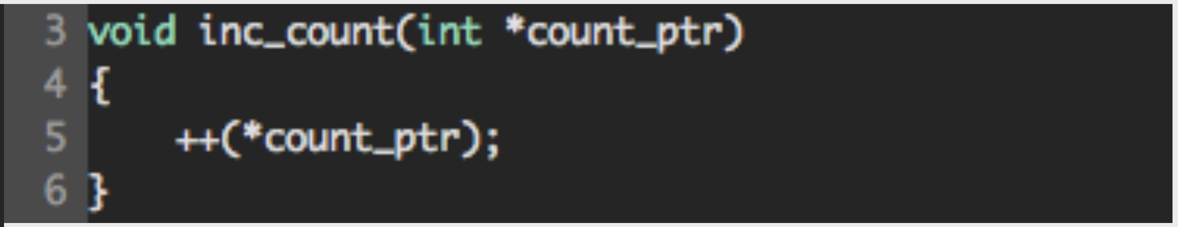


Call by Reference

- Example: call.c



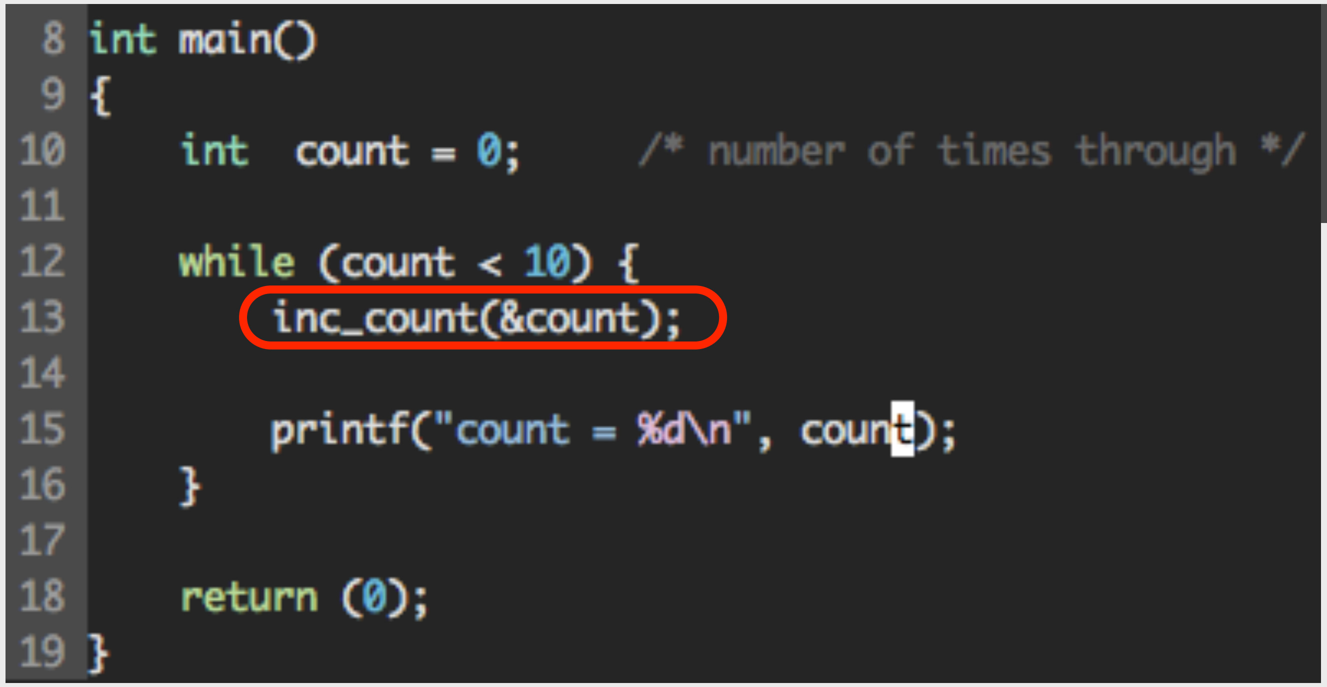
```c
int main()
{
    int  count = 0;     /* number of times through */

    while (count < 10) {
        inc_count(&count);

        printf("count = %d\n", count);
    }

    return (0);
}
```



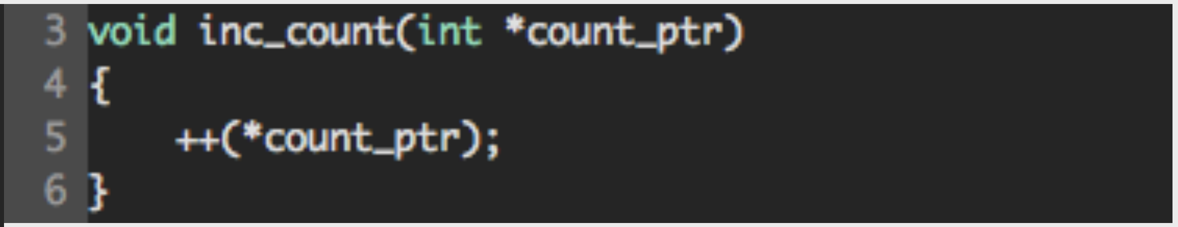
```c
void inc_count(int *count_ptr)
{
    ++(*count_ptr);
}
```

Call by Reference

- Example: call.c



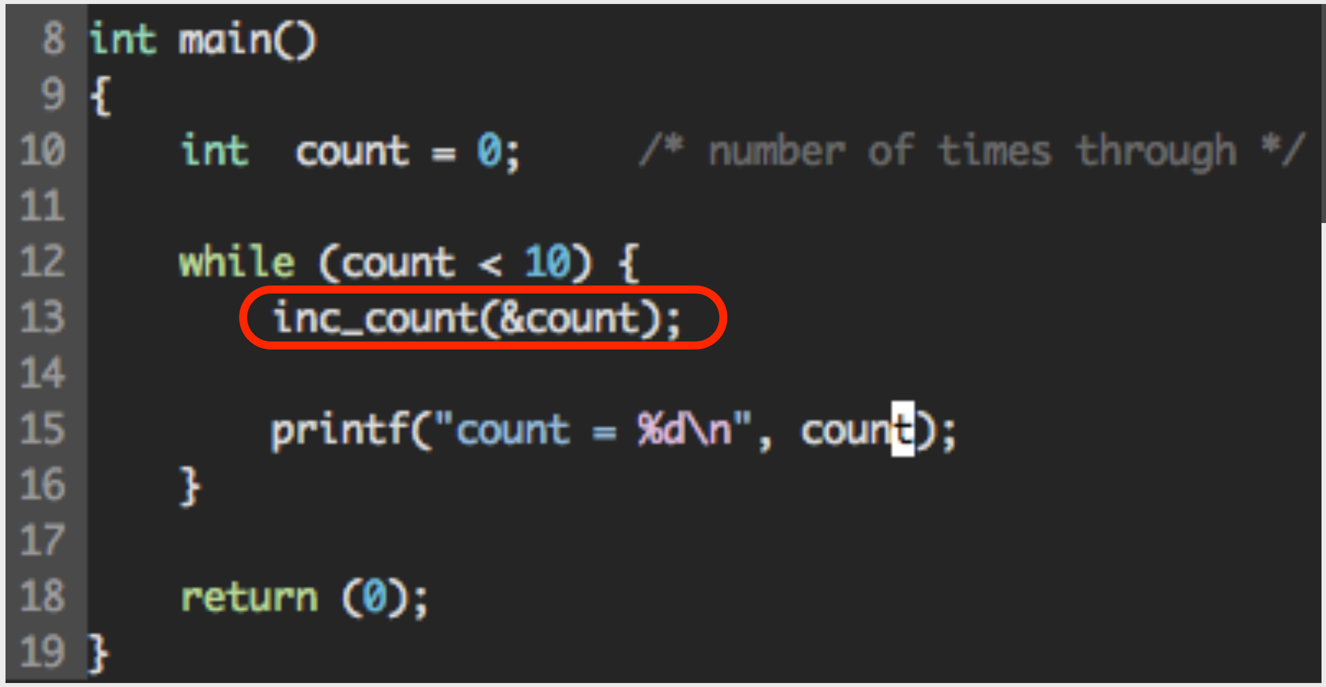
```
int main()
{
    int  count = 0;      /* number of times through */

    while (count < 10) {
        inc_count(&count);

        printf("count = %d\n", count);
    }

    return (0);
}
```



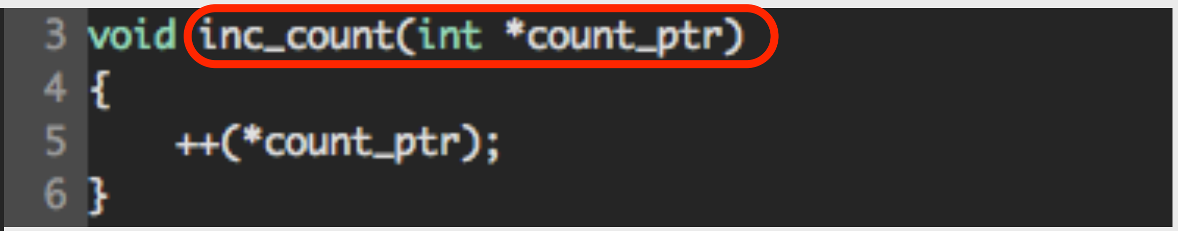
```
void inc_count(int *count_ptr)
{
    ++(*count_ptr);
}
```

Call by Reference

- Example: call.c



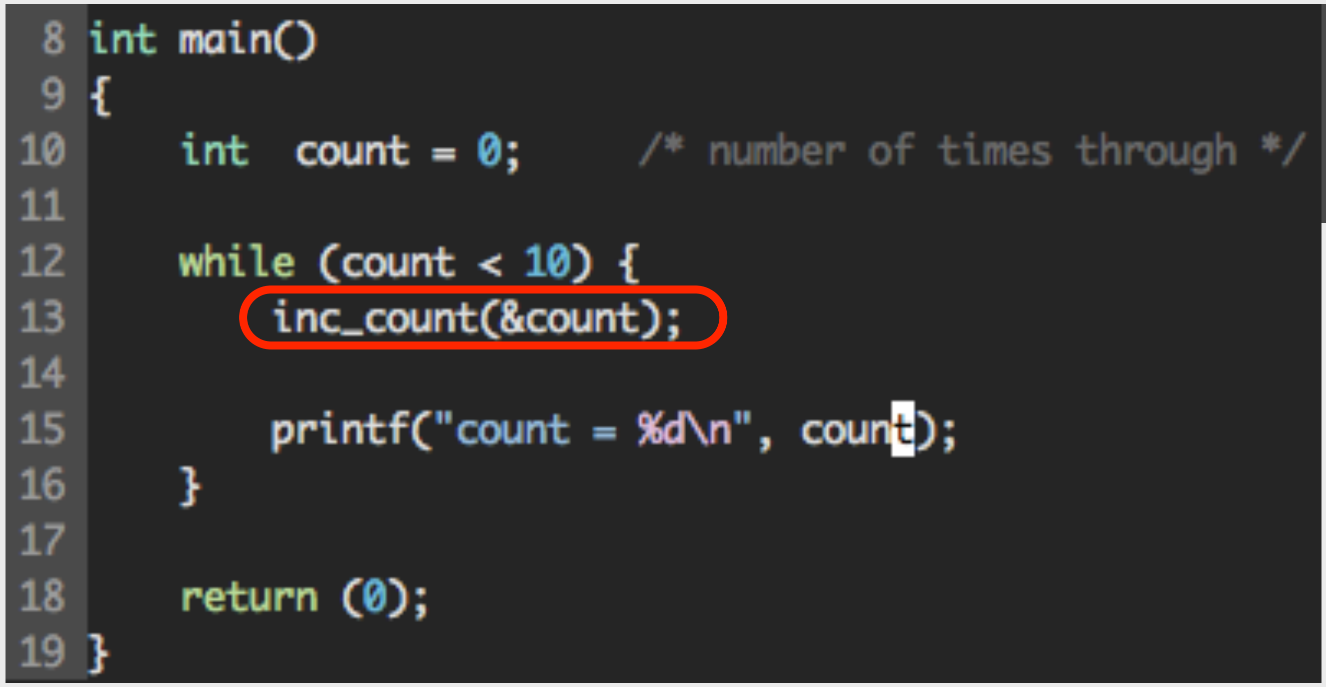
```
int main()
{
    int  count = 0;      /* number of times through */

    while (count < 10) {
        inc_count(&count);

        printf("count = %d\n", count);
    }

    return (0);
}
```



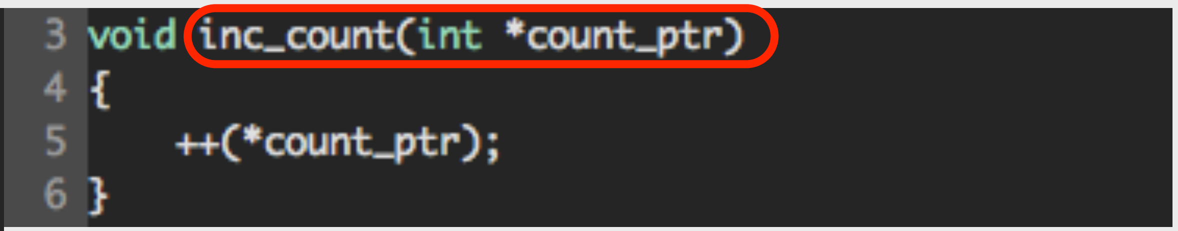
```
void inc_count(int *count_ptr)
{
    ++(*count_ptr);
}
```

Call by Reference

- Example: call.c



```c
int main()
{
    int  count = 0;       /* number of times through */

    while (count < 10) {
        inc_count(&count);

        printf("count = %d\n", count);
    }

    return (0);
}
```

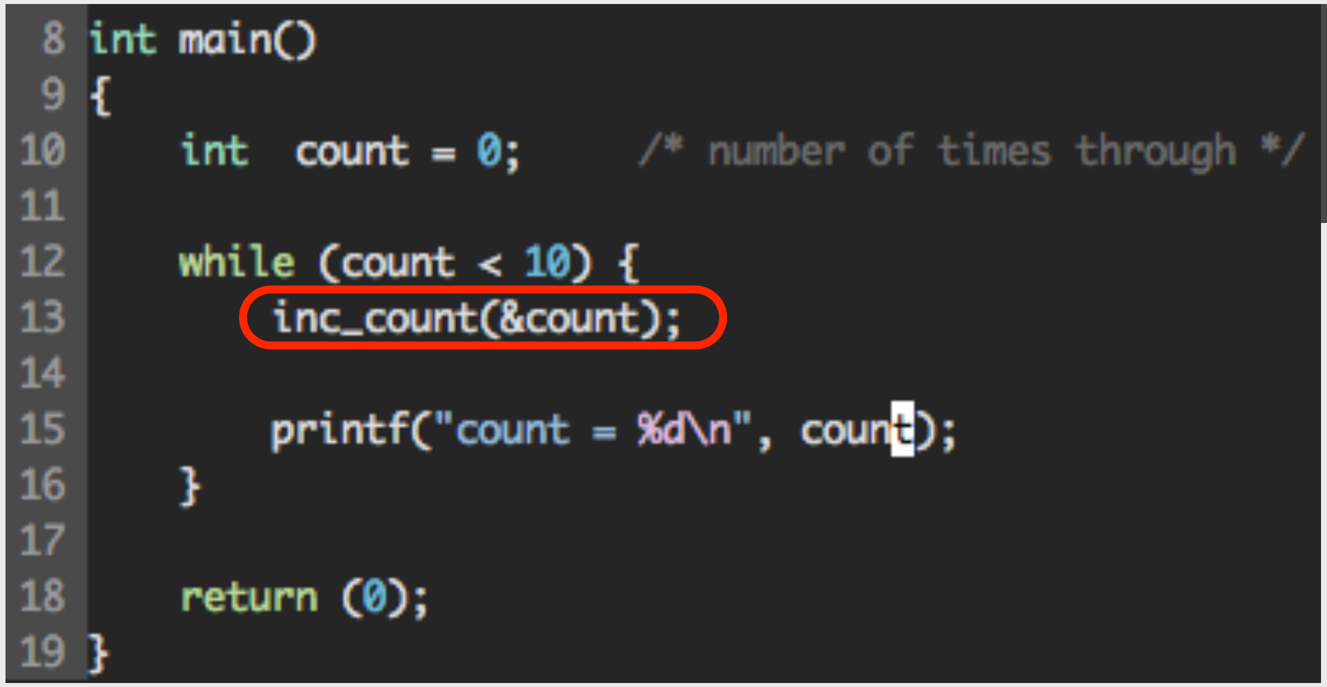


```c
void inc_count(int *count_ptr)
{
    ++(*count_ptr);
}
```

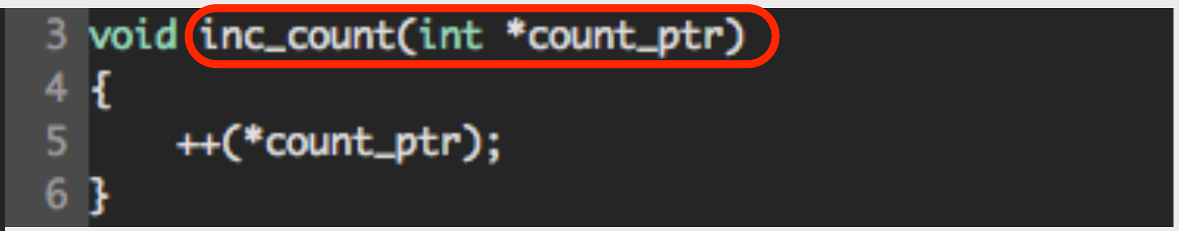
Output

Call by Reference

- Example: call.c

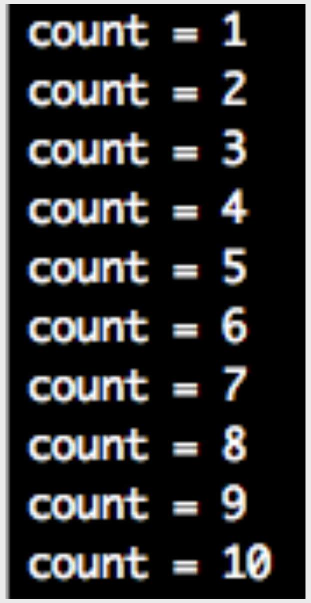


```c
int main()
{
    int  count = 0;      /* number of times through */

    while (count < 10) {
        inc_count(&count);

        printf("count = %d\n", count);
    }

    return (0);
}
```



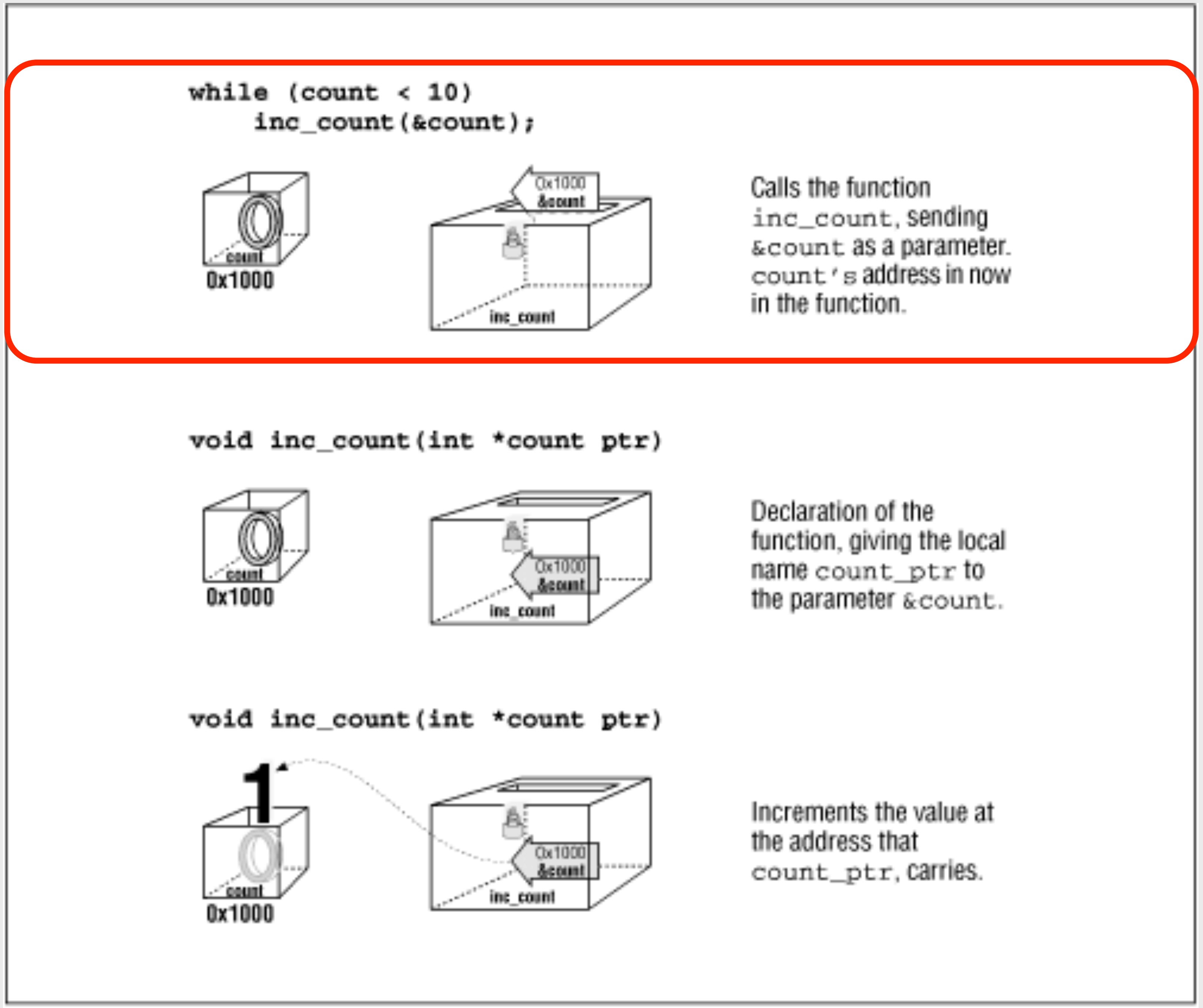
```c
void inc_count(int *count_ptr)
{
    ++(*count_ptr);
}
```

Output



```
count = 1
count = 2
count = 3
count = 4
count = 5
count = 6
count = 7
count = 8
count = 9
count = 10
```

Call by Reference



```
while (count < 10)
    inc_count(&count);
```

Calls the function `inc_count`, sending `&count` as a parameter. `count's` address in now in the function.

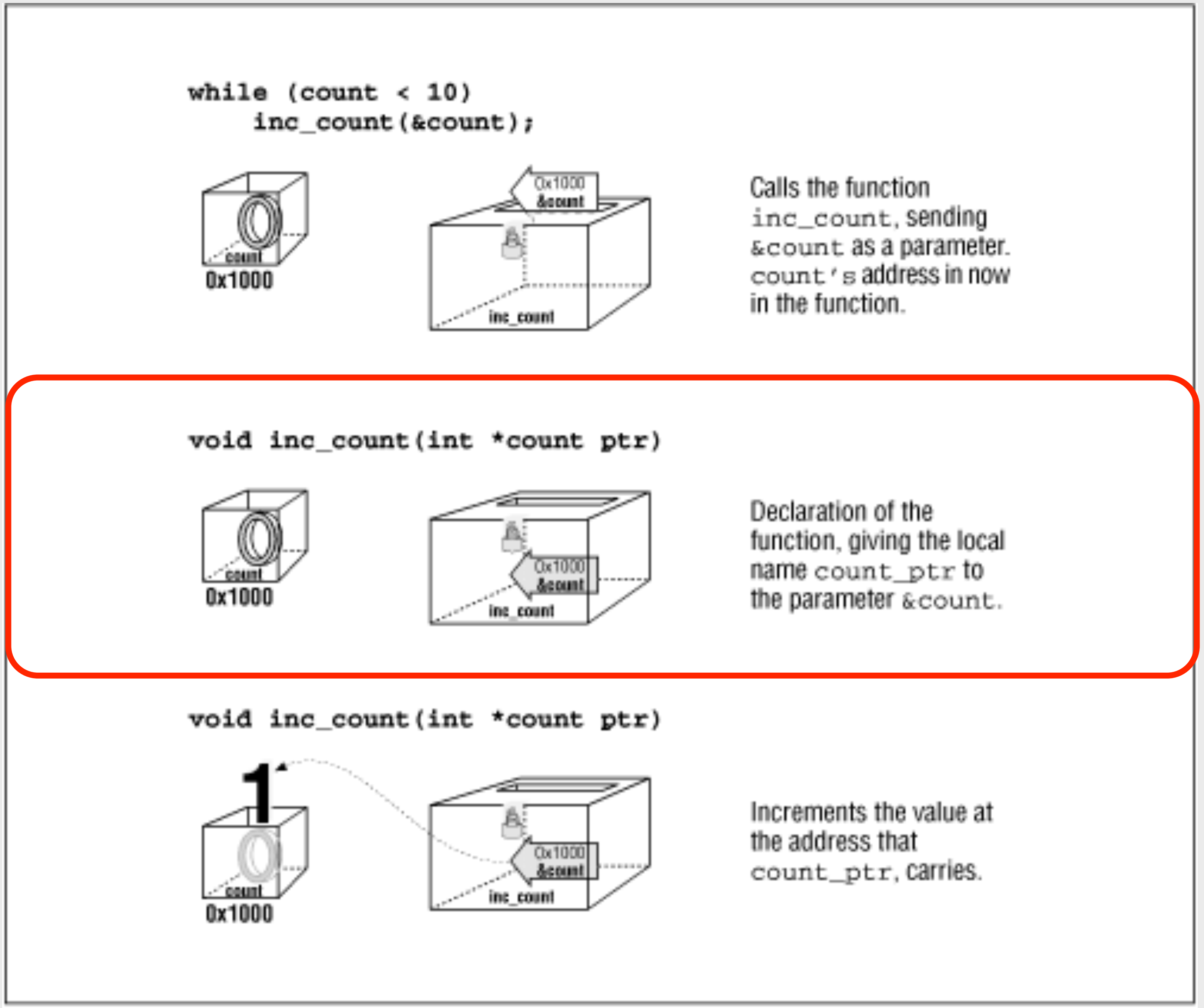
```
void inc_count(int *count ptr)
```

Declaration of the function, giving the local name `count_ptr` to the parameter `&count`.

```
void inc_count(int *count ptr)
```

Increments the value at the address that `count_ptr`, carries.

Call by Reference



```
while (count < 10)
    inc_count(&count);
```

Calls the function inc_count, sending &count as a parameter. count's address in now in the function.

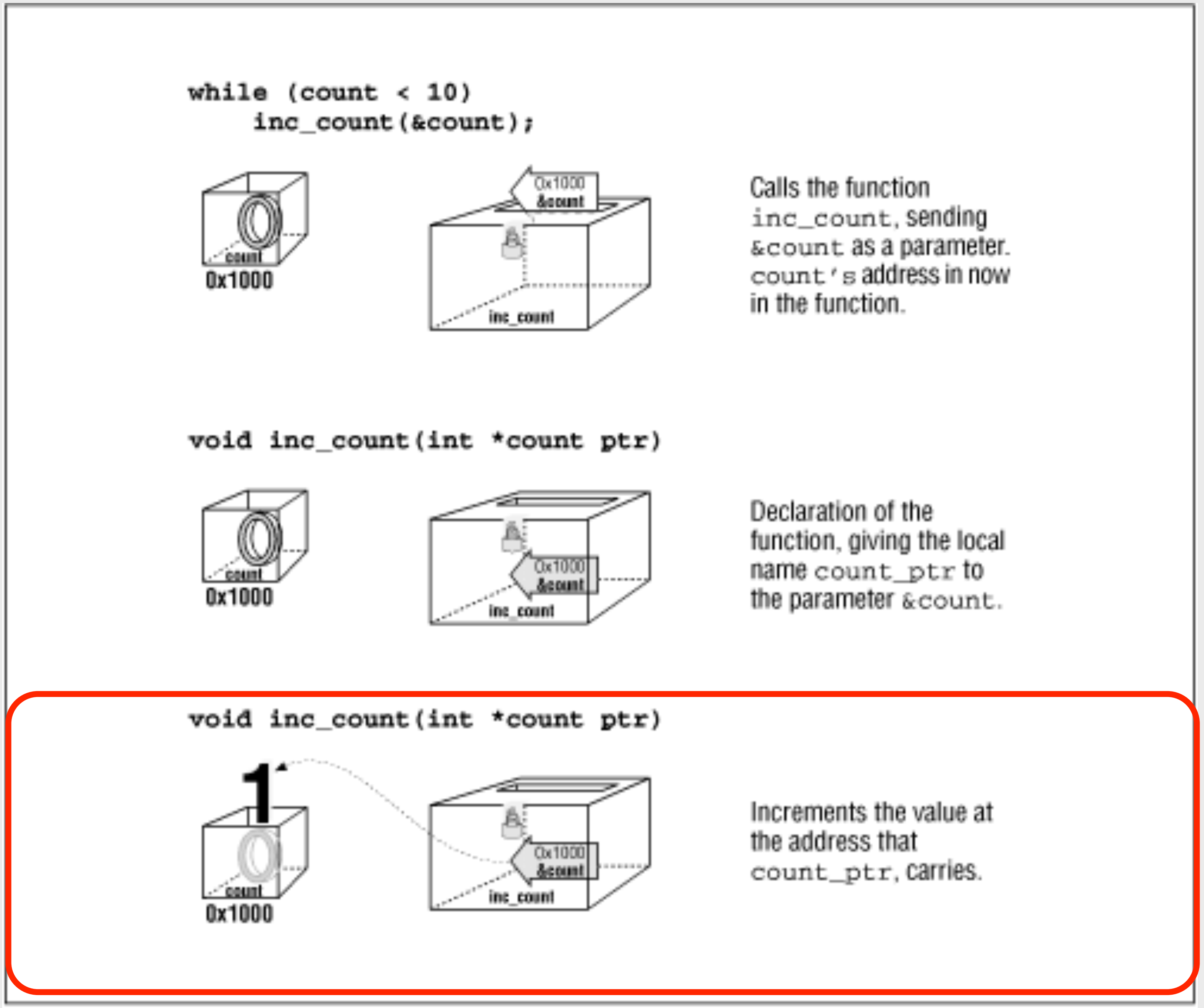
```
void inc_count(int *count ptr)
```

Declaration of the function, giving the local name count_ptr to the parameter &count.

```
void inc_count(int *count ptr)
```

Increments the value at the address that count_ptr, carries.

Call by Reference



```
while (count < 10)
    inc_count(&count);
```

Calls the function inc_count, sending &count as a parameter. count's address in now in the function.

```
void inc_count(int *count ptr)
```

Declaration of the function, giving the local name count_ptr to the parameter &count.

```
void inc_count(int *count ptr)
```

Increments the value at the address that count_ptr, carries.

`const` Pointers

const Pointers

```
1.
const int result = 5;
result = 10;  /* illegal */
```

`const` Pointers

```
1.
const int result = 5;
result = 10;  /* illegal */

              2.
              const int *answer_ptr = 10;

              answer_ptr = &var;  /* legal */ ?
              *answer_ptr = 20;   /* illegal */ ?
```

const Pointers

```
1.
const int result = 5;
result = 10;  /* illegal */
```

```
2.
const int *answer_ptr = 10;

answer_ptr = &var;   /* legal */ ?
*answer_ptr = 20;    /* illegal */ ?
```

That is because:

`answer_ptr` is a variable
`*answer_ptr` is a constant

`const` Pointers

const Pointers

```
3.
int const *name_ptr = 10;

name_ptr = &var; /* illegal */?
*name_ptr = 20;  /* legal */?
```

const Pointers

```
3.
int const *name_ptr = 10;

name_ptr = &var; /* illegal */?
*name_ptr = 20;  /* legal */?
```

```
4.
const int const *title_ptr = 10;
```

const Pointers

```
3.
int const *name_ptr = 10;

name_ptr = &var; /* illegal */?
*name_ptr = 20;  /* legal */?
```

That is because:

name_ptr is a constant
***name_ptr** is a variable

```
4.
const int const *title_ptr = 10;
```

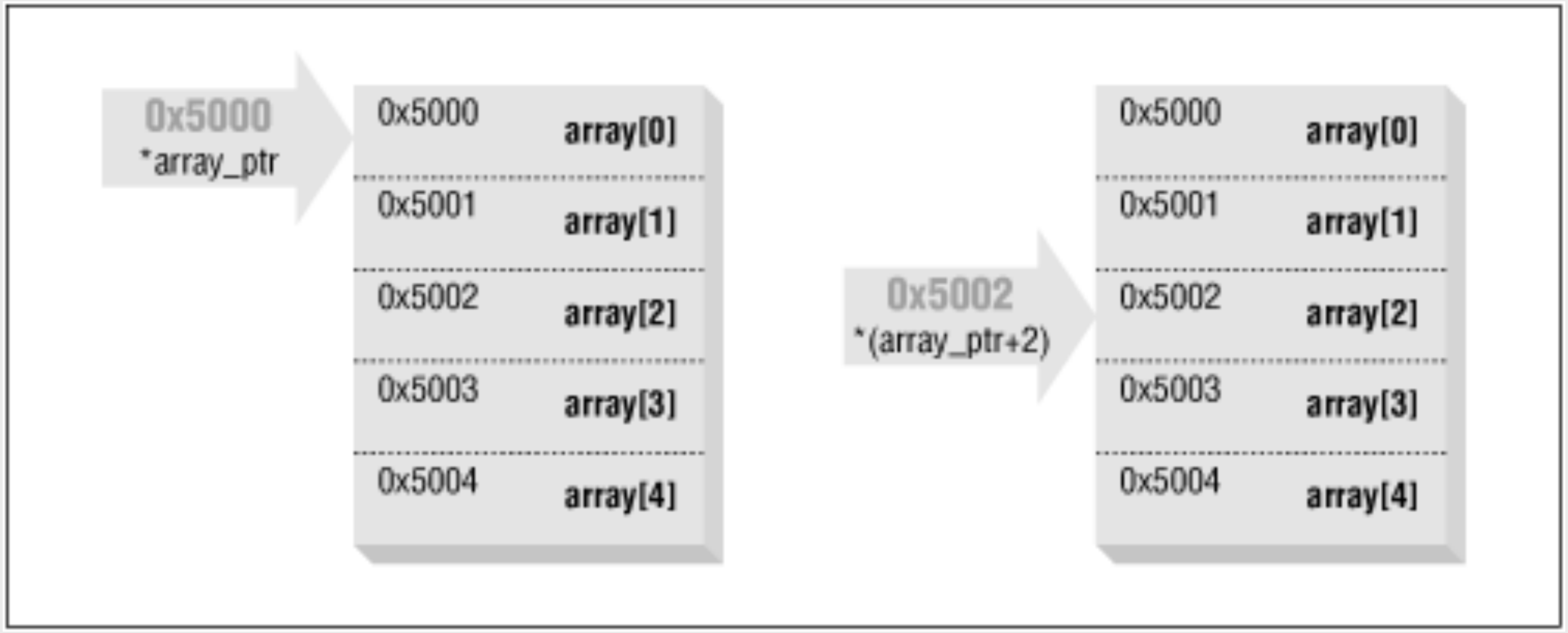
Pointers and Arrays

Pointers and Arrays

- Pointer arithmetic (addition and subtraction)

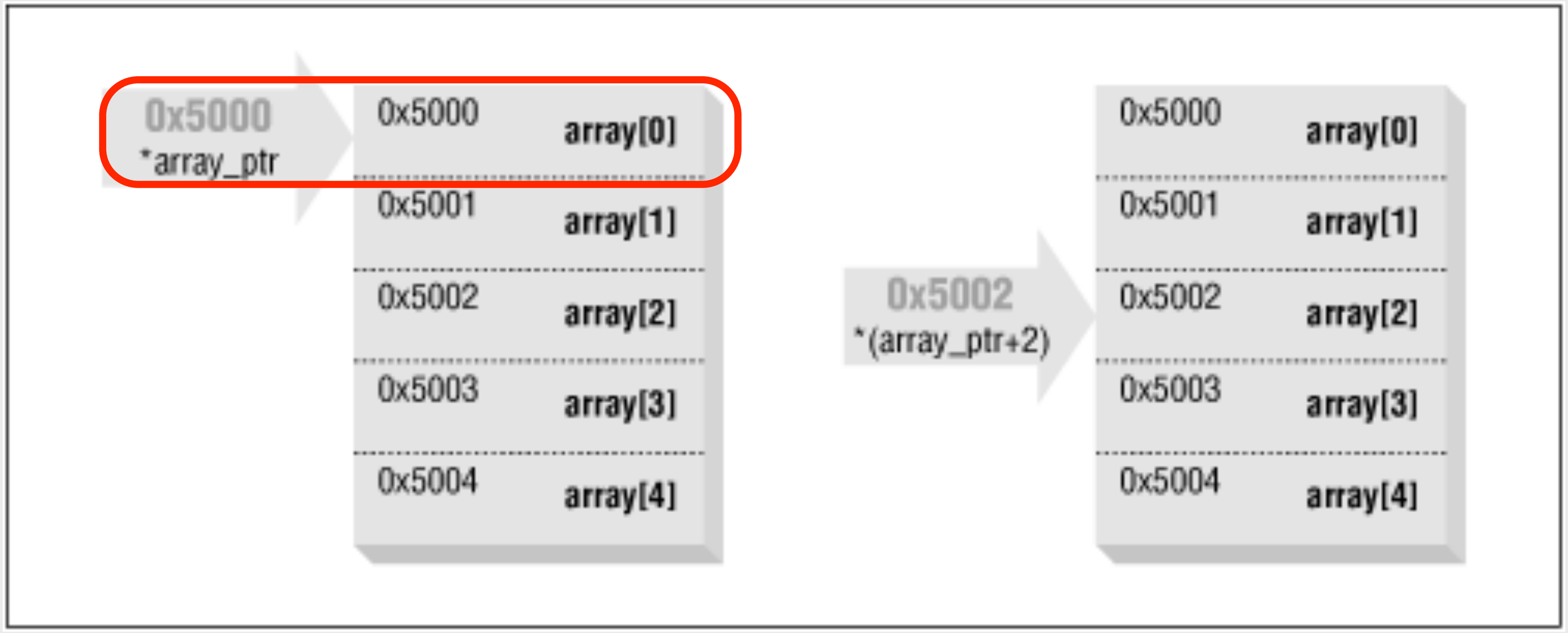
  ```
  char array[5];
  char *array_ptr = &array[0];
  ```

- `*(array_ptr + 1)` is the same as `array[1]`
- `(*array_ptr) + 1` is the same as `array[0] + 1`



Pointers and Arrays

- Pointer arithmetic (addition and subtraction)
  ```
  char array[5];
  char *array_ptr = &array[0];
  ```
- `*(array_ptr + 1)` is the same as `array[1]`
- `(*array_ptr) + 1` is the same as `array[0] + 1`

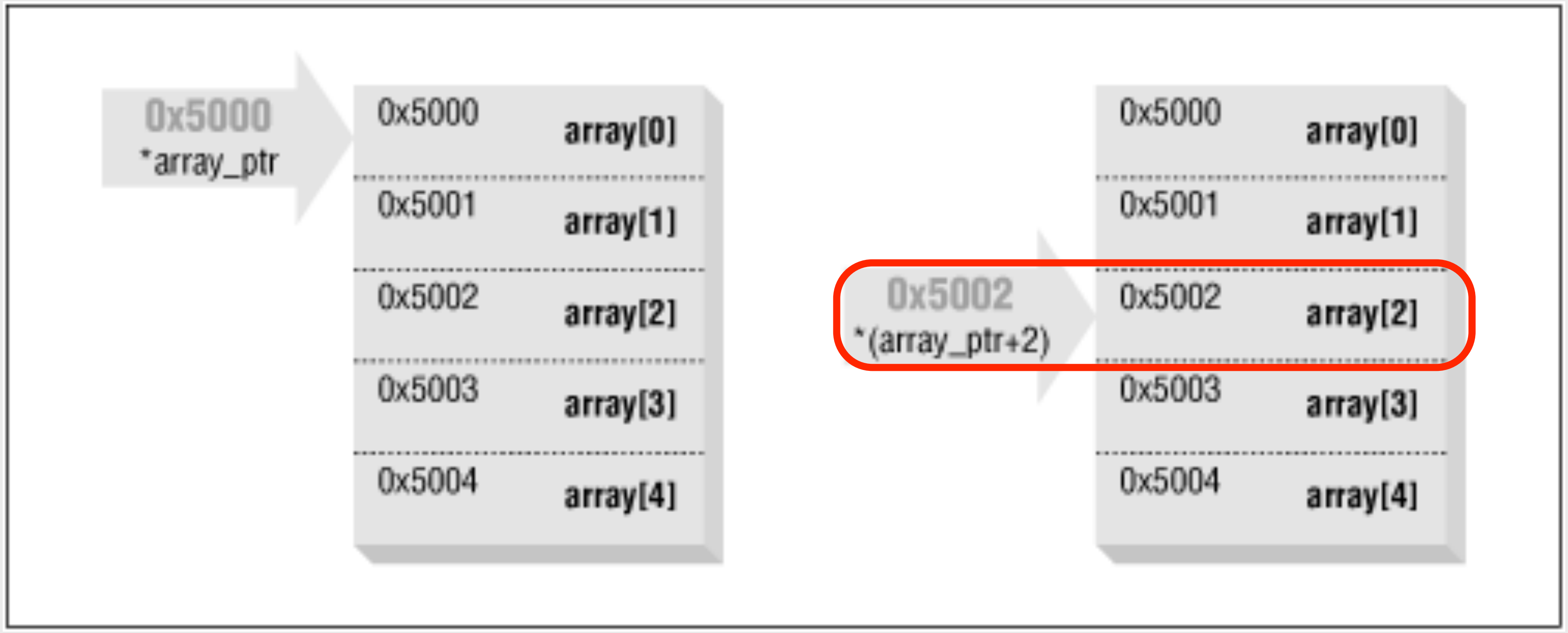


Pointers and Arrays

- Pointer arithmetic (addition and subtraction)

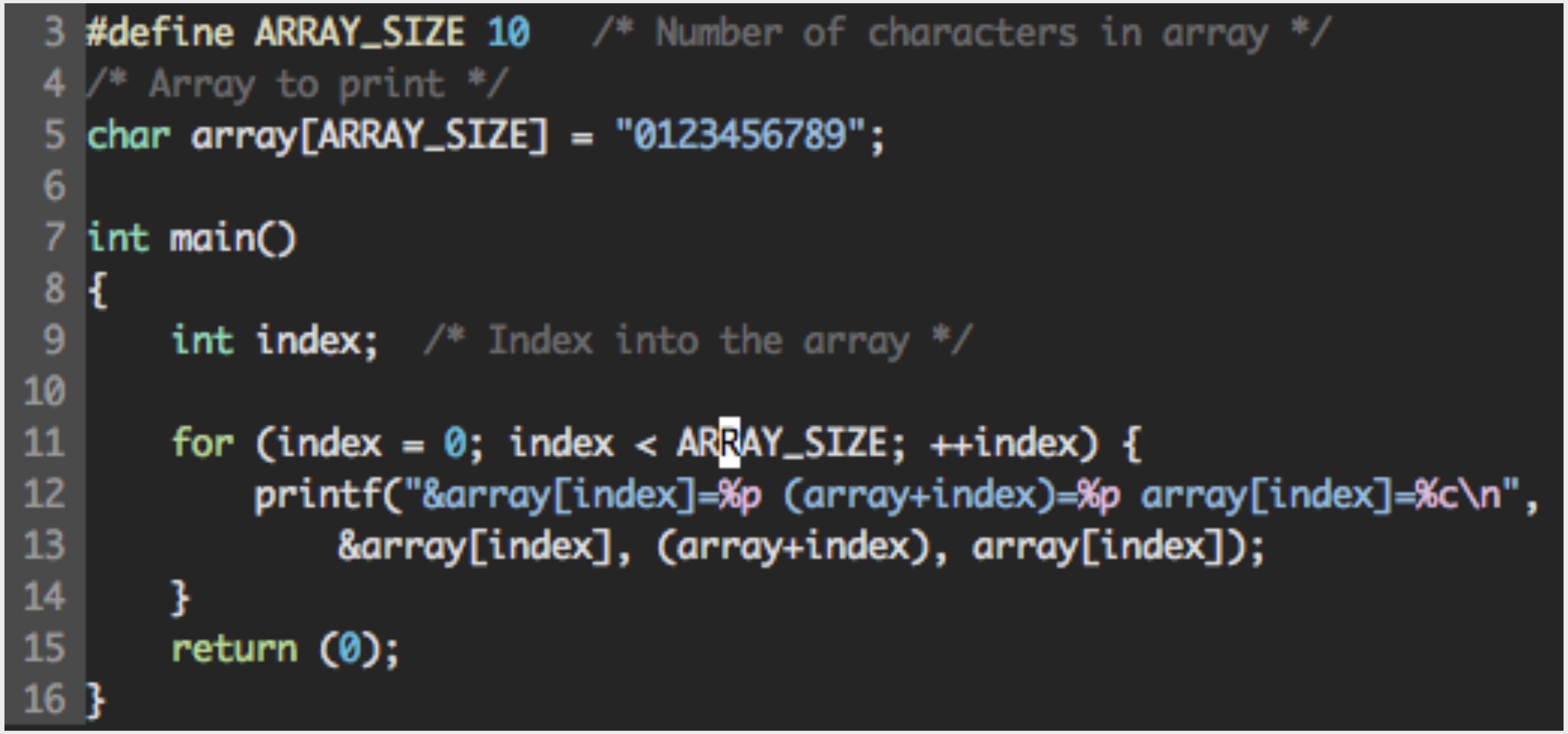
```
char array[5];
char *array_ptr = &array[0];
```

- `*(array_ptr + 1)` is the same as `array[1]`
- `(*array_ptr) + 1` is the same as `array[0] + 1`



Pointers and Arrays

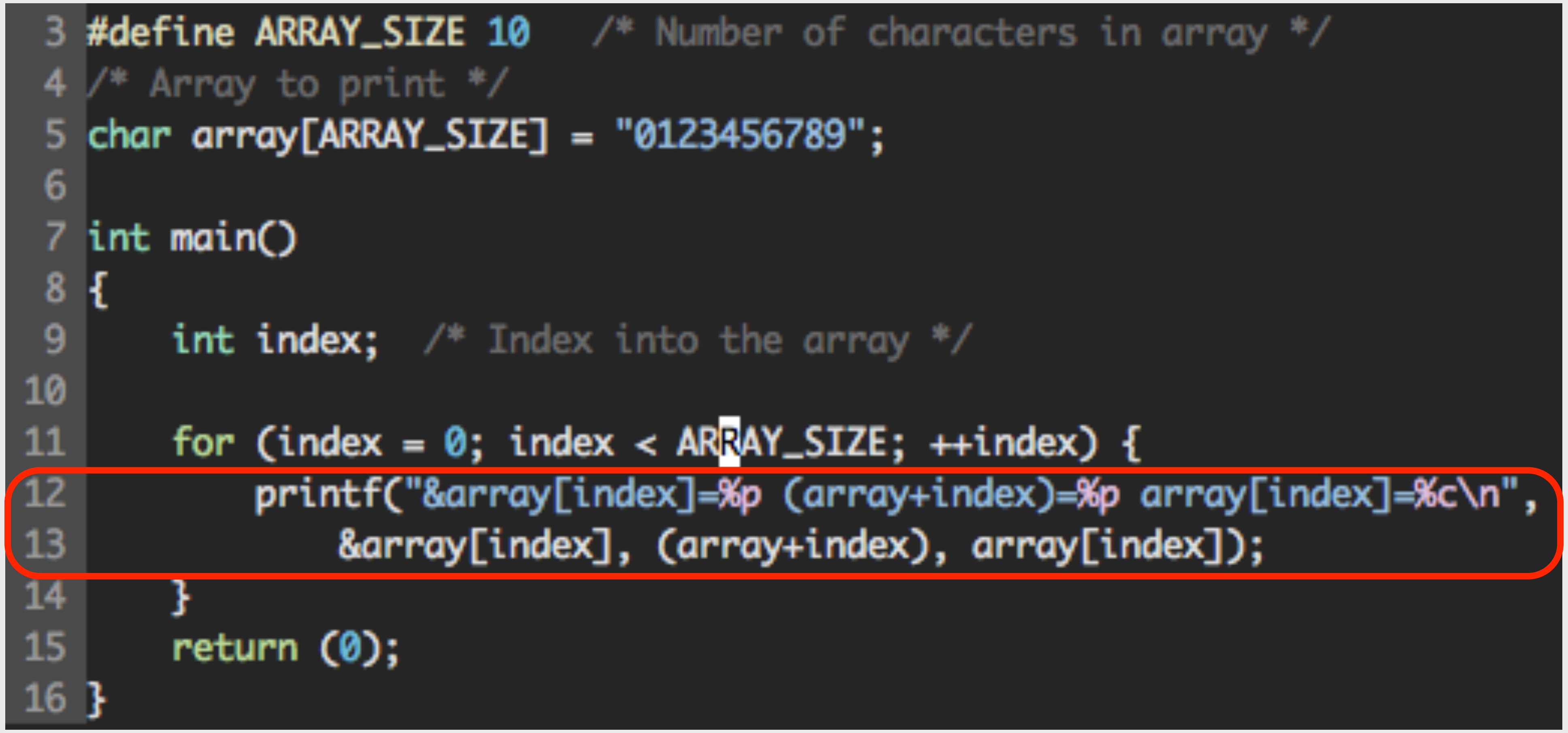
- Example: array-p.c



```c
#define ARRAY_SIZE 10   /* Number of characters in array */
/* Array to print */
char array[ARRAY_SIZE] = "0123456789";

int main()
{
    int index;  /* Index into the array */

    for (index = 0; index < ARRAY_SIZE; ++index) {
        printf("&array[index]=%p (array+index)=%p array[index]=%c\n",
            &array[index], (array+index), array[index]);
    }
    return (0);
}
```

Pointers and Arrays

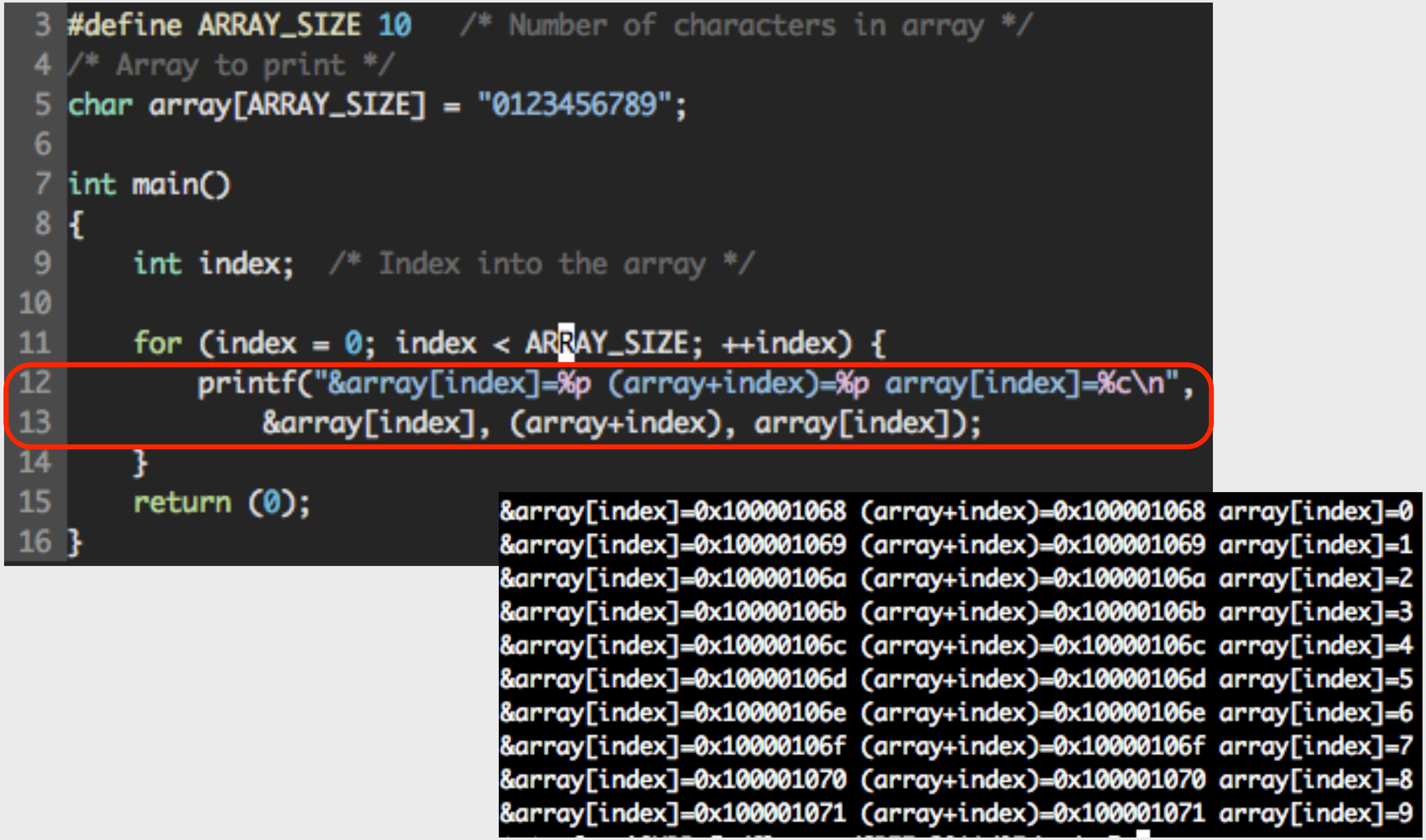
- Example: array-p.c



```c
#define ARRAY_SIZE 10   /* Number of characters in array */
/* Array to print */
char array[ARRAY_SIZE] = "0123456789";

int main()
{
    int index;  /* Index into the array */

    for (index = 0; index < ARRAY_SIZE; ++index) {
        printf("&array[index]=%p (array+index)=%p array[index]=%c\n",
            &array[index], (array+index), array[index]);
    }
    return (0);
}
```

Pointers and Arrays

- Example: array-p.c



```c
#define ARRAY_SIZE 10   /* Number of characters in array */
/* Array to print */
char array[ARRAY_SIZE] = "0123456789";

int main()
{
    int index;  /* Index into the array */

    for (index = 0; index < ARRAY_SIZE; ++index) {
        printf("&array[index]=%p (array+index)=%p array[index]=%c\n",
          &array[index], (array+index), array[index]);
    }
    return (0);
}
```

```
&array[index]=0x100001068 (array+index)=0x100001068 array[index]=0
&array[index]=0x100001069 (array+index)=0x100001069 array[index]=1
&array[index]=0x10000106a (array+index)=0x10000106a array[index]=2
&array[index]=0x10000106b (array+index)=0x10000106b array[index]=3
&array[index]=0x10000106c (array+index)=0x10000106c array[index]=4
&array[index]=0x10000106d (array+index)=0x10000106d array[index]=5
&array[index]=0x10000106e (array+index)=0x10000106e array[index]=6
&array[index]=0x10000106f (array+index)=0x10000106f array[index]=7
&array[index]=0x100001070 (array+index)=0x100001070 array[index]=8
&array[index]=0x100001071 (array+index)=0x100001071 array[index]=9
```

Pointers and Arrays

- C provide a shorthand to dealing with array

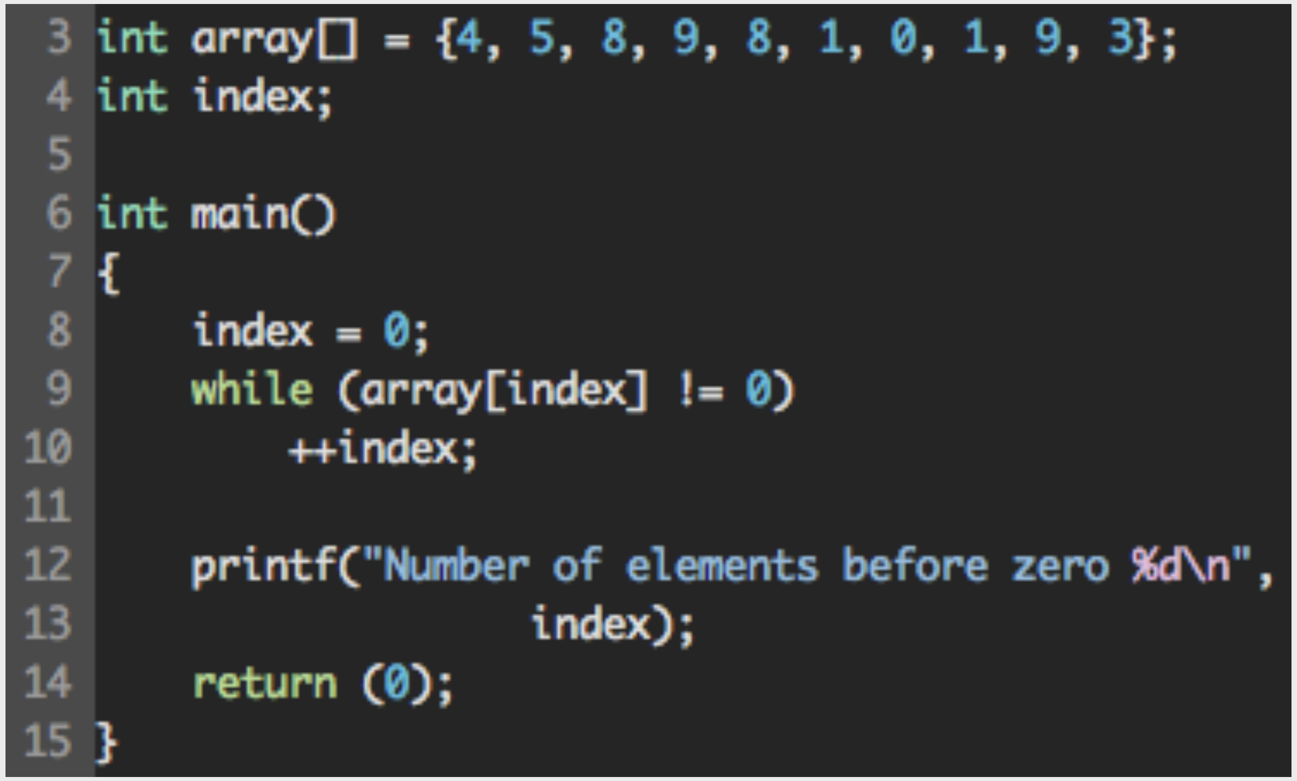
  ```
  array_ptr = &array[0];
  ```

  we can write

  ```
  array_ptr = array;
  ```

Pointers and Arrays

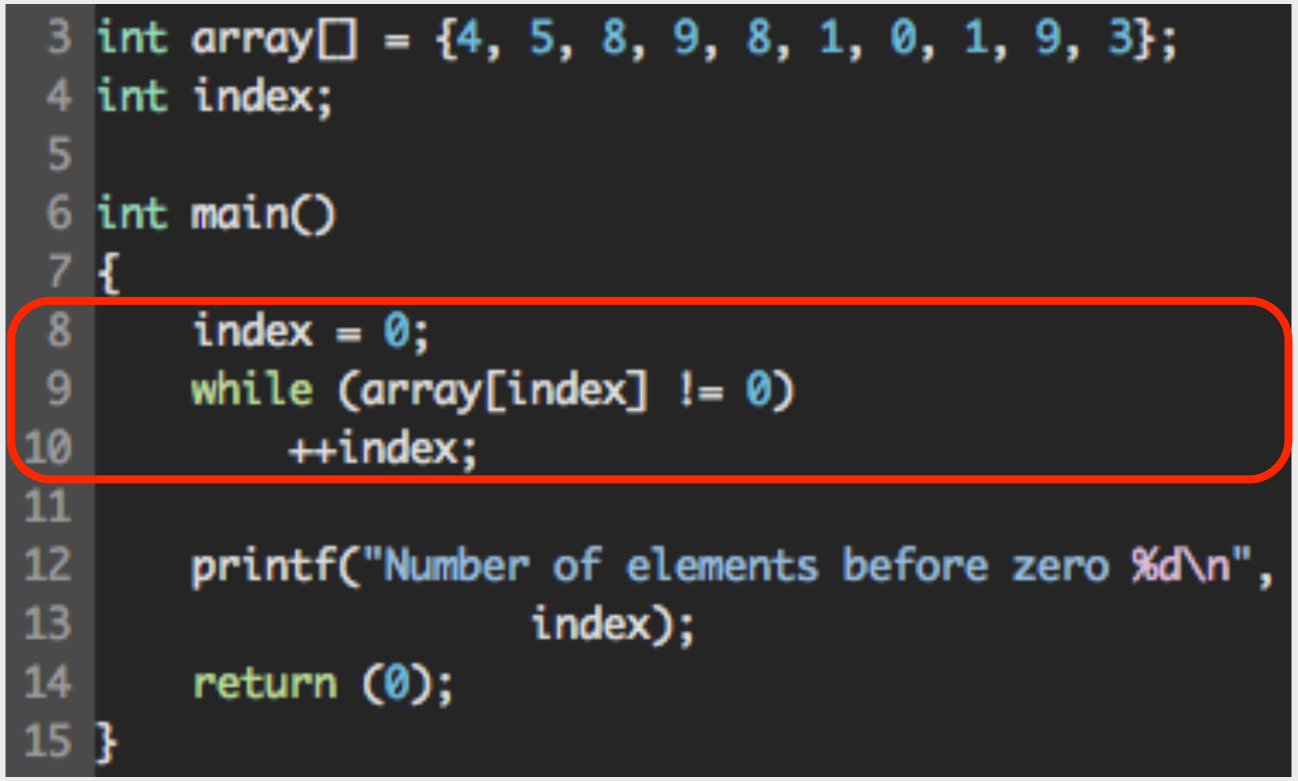
- Example: ptr2.c/ptr3.c



```c
int array[] = {4, 5, 8, 9, 8, 1, 0, 1, 9, 3};
int index;

int main()
{
    index = 0;
    while (array[index] != 0)
        ++index;

    printf("Number of elements before zero %d\n",
                  index);
    return (0);
}
```

Pointers and Arrays

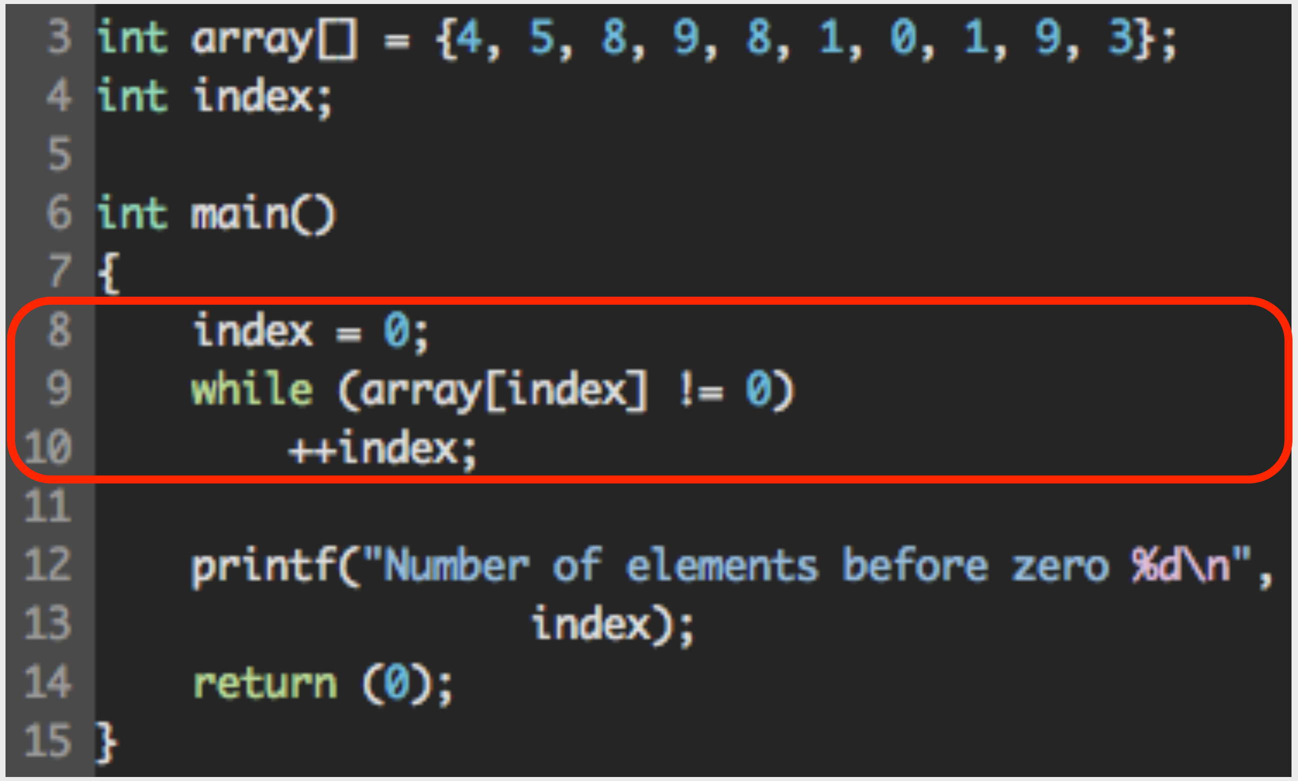
- Example: ptr2.c/ptr3.c



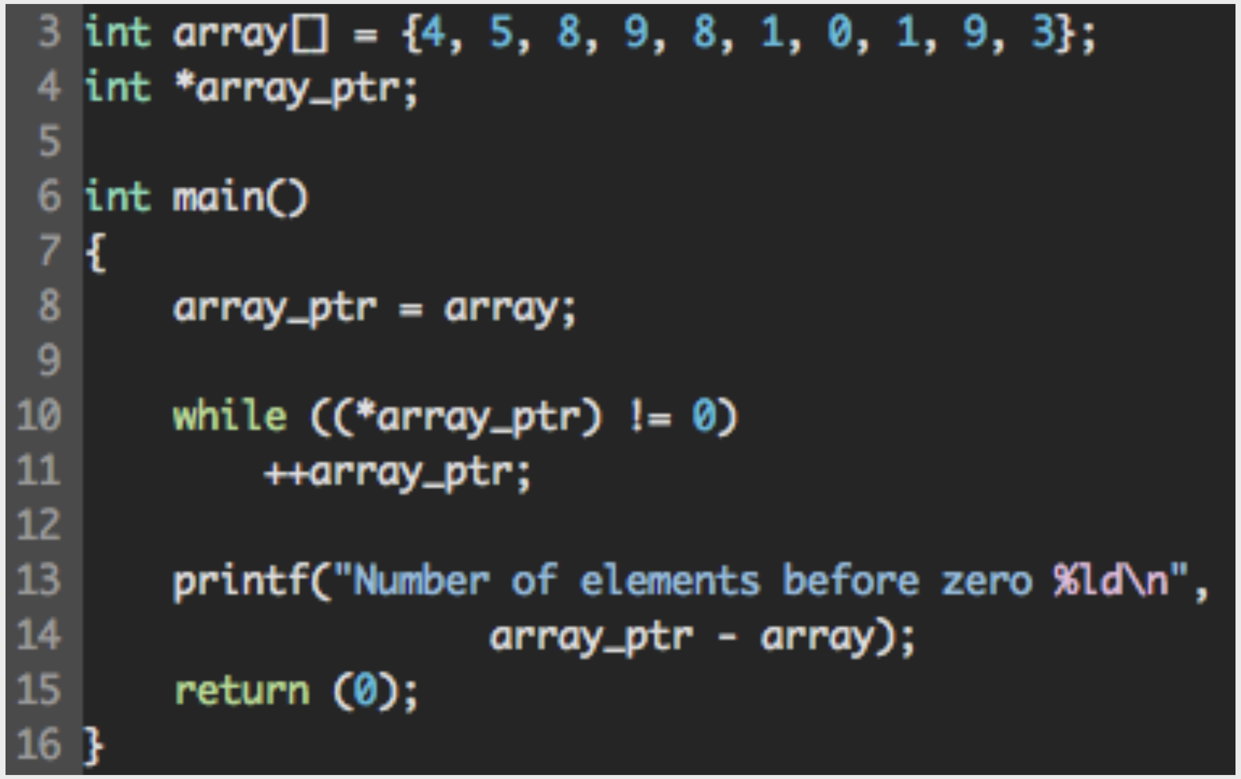
```
int array[] = {4, 5, 8, 9, 8, 1, 0, 1, 9, 3};
int index;

int main()
{
    index = 0;
    while (array[index] != 0)
        ++index;

    printf("Number of elements before zero %d\n",
                  index);
    return (0);
}
```

Pointers and Arrays

- Example: ptr2.c/ptr3.c



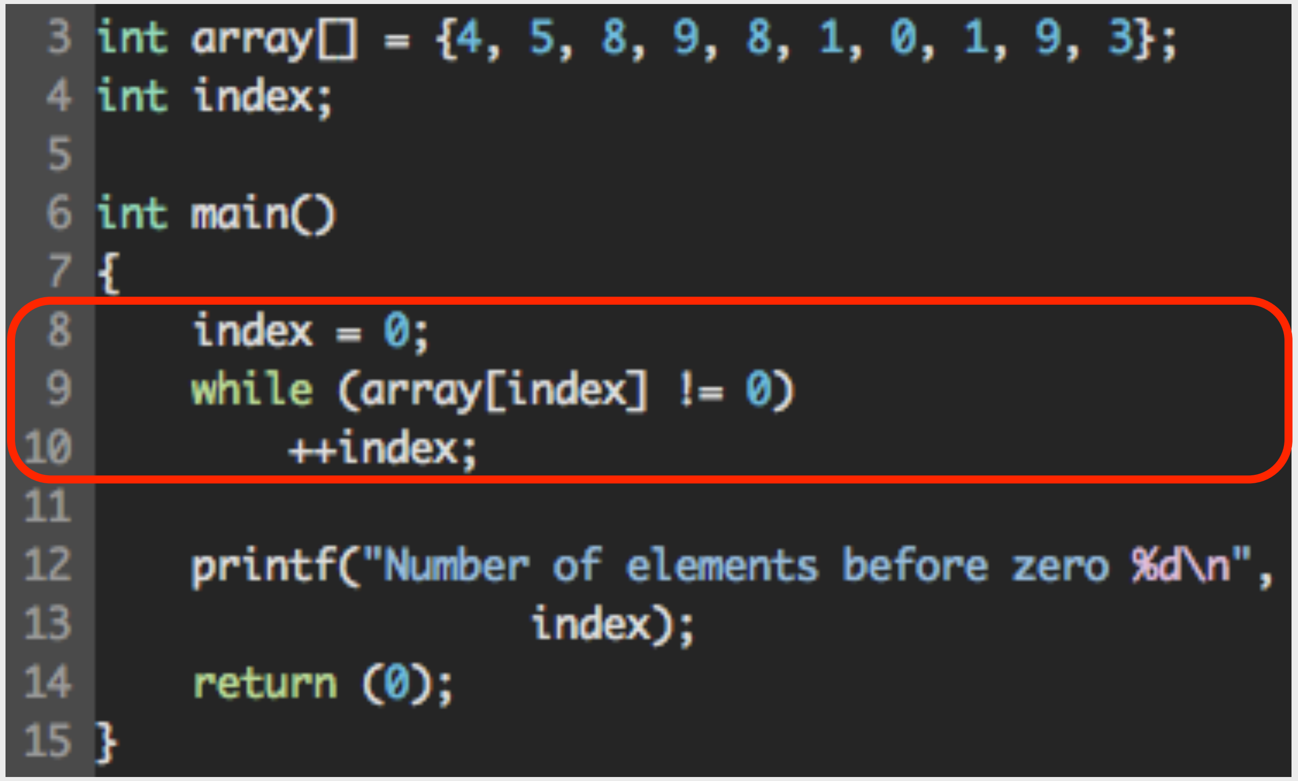
```c
int array[] = {4, 5, 8, 9, 8, 1, 0, 1, 9, 3};
int index;

int main()
{
    index = 0;
    while (array[index] != 0)
        ++index;

    printf("Number of elements before zero %d\n",
                  index);
    return (0);
}
```



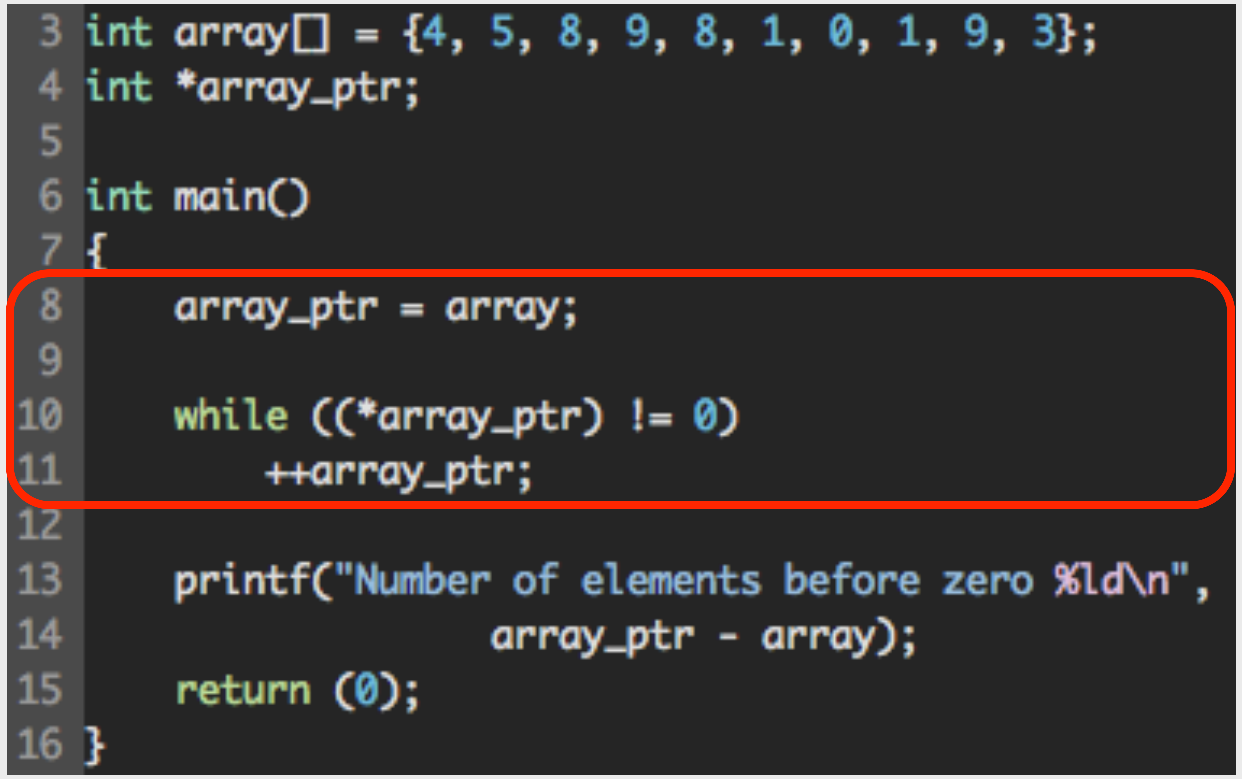
```c
int array[] = {4, 5, 8, 9, 8, 1, 0, 1, 9, 3};
int *array_ptr;

int main()
{
    array_ptr = array;

    while ((*array_ptr) != 0)
        ++array_ptr;

    printf("Number of elements before zero %ld\n",
                  array_ptr - array);
    return (0);
}
```

Pointers and Arrays

- Example: ptr2.c/ptr3.c

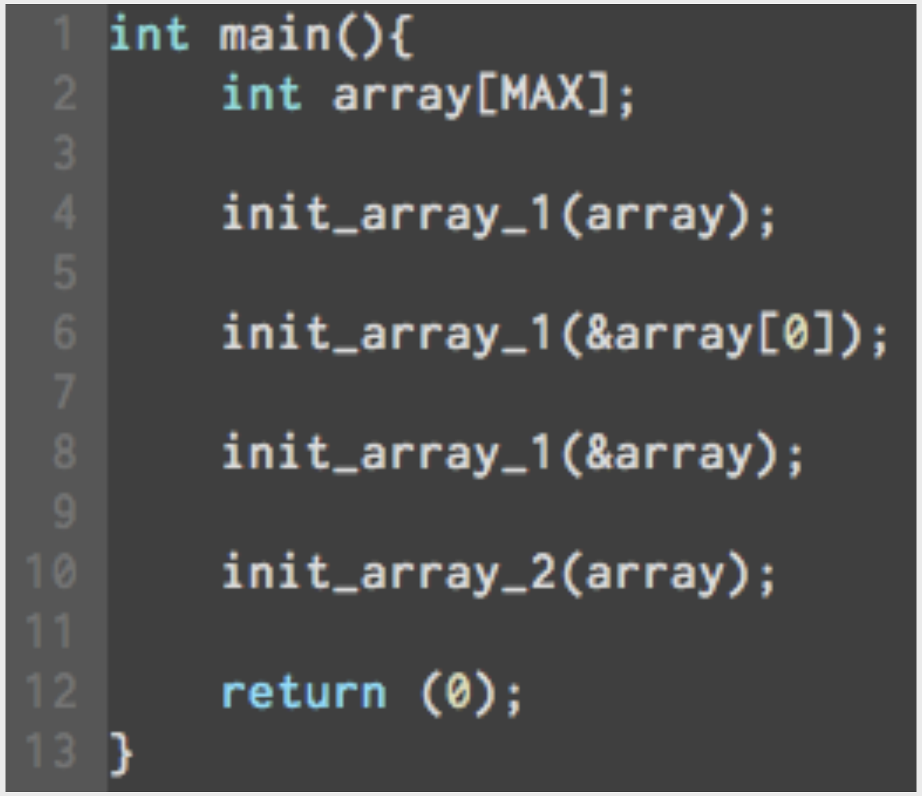


```c
int array[] = {4, 5, 8, 9, 8, 1, 0, 1, 9, 3};
int index;

int main()
{
    index = 0;
    while (array[index] != 0)
        ++index;

    printf("Number of elements before zero %d\n",
                    index);
    return (0);
}
```



```c
int array[] = {4, 5, 8, 9, 8, 1, 0, 1, 9, 3};
int *array_ptr;

int main()
{
    array_ptr = array;

    while ((*array_ptr) != 0)
        ++array_ptr;

    printf("Number of elements before zero %ld\n",
                    array_ptr - array);
    return (0);
}
```

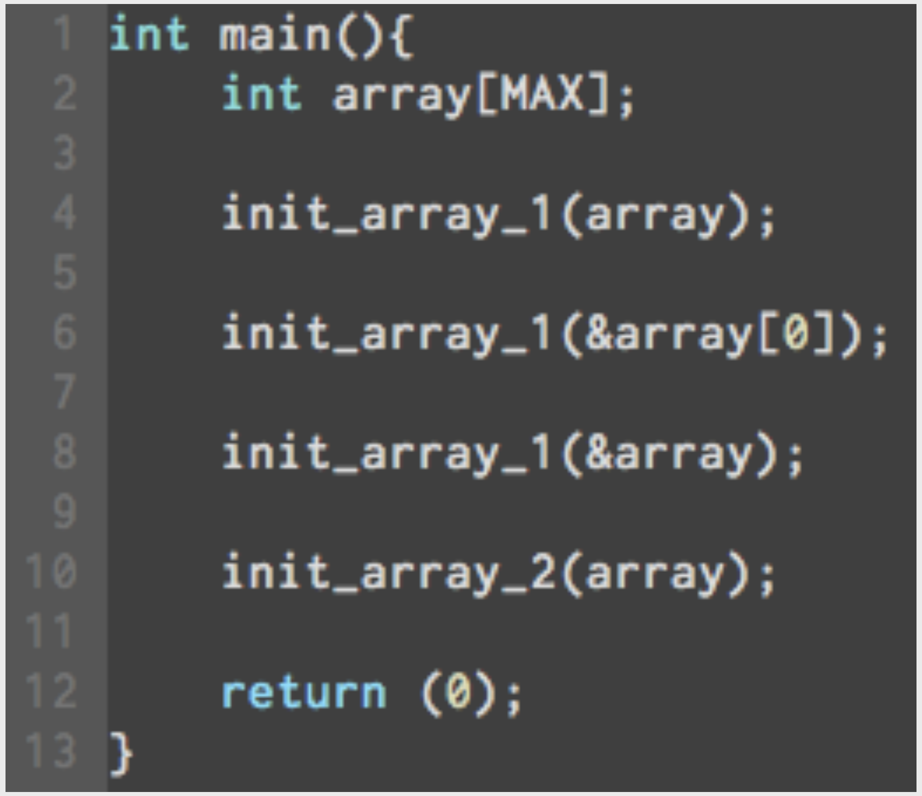
Passing an Array to a Function



```
int main(){
    int array[MAX];

    init_array_1(array);

    init_array_1(&array[0]);

    init_array_1(&array);

    init_array_2(array);

    return (0);
}
```

Passing an Array to a Function

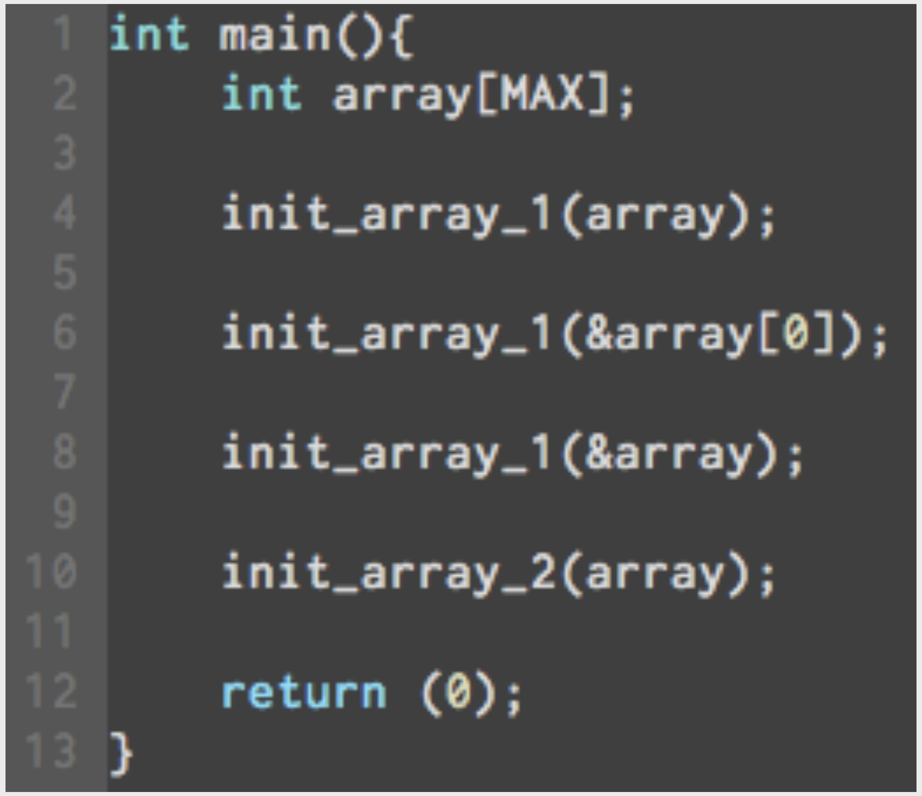
- When passing an array to a procedure, C will automatically change the array into a pointer



```c
int main(){
    int array[MAX];

    init_array_1(array);

    init_array_1(&array[0]);

    init_array_1(&array);

    init_array_2(array);

    return (0);
}
```

Passing an Array to a Function

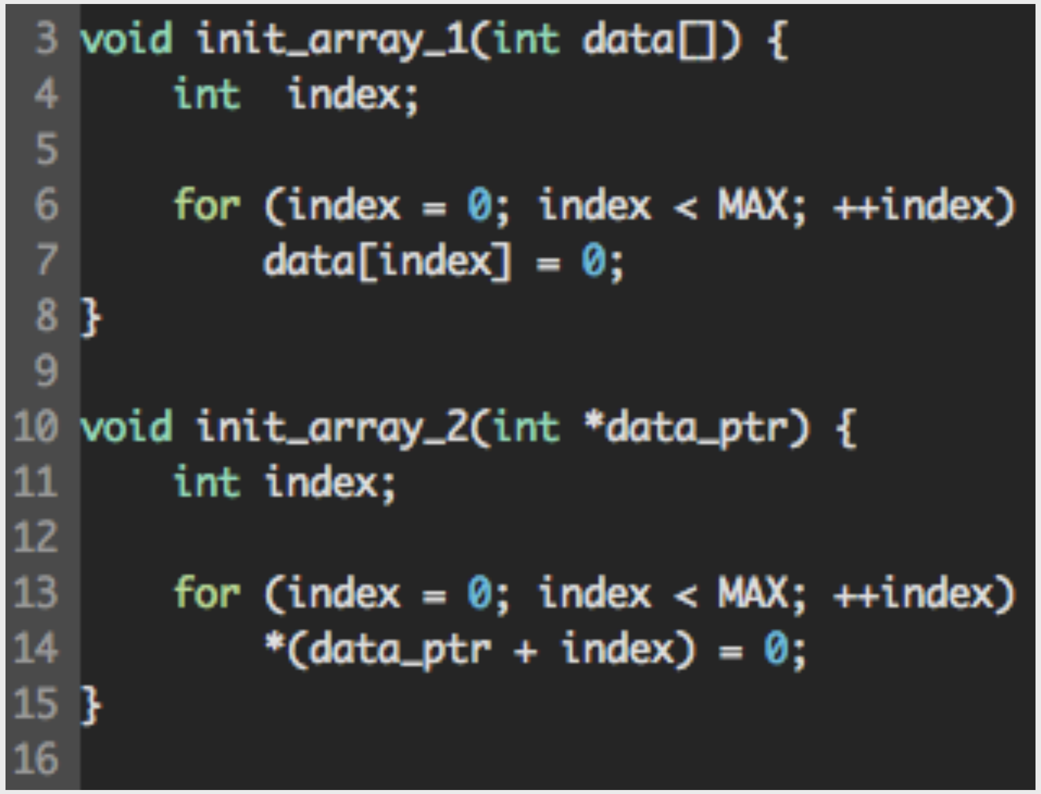
- When passing an array to a procedure, C will automatically change the array into a pointer
- So, if you put **&** before the array, C will issue a warning



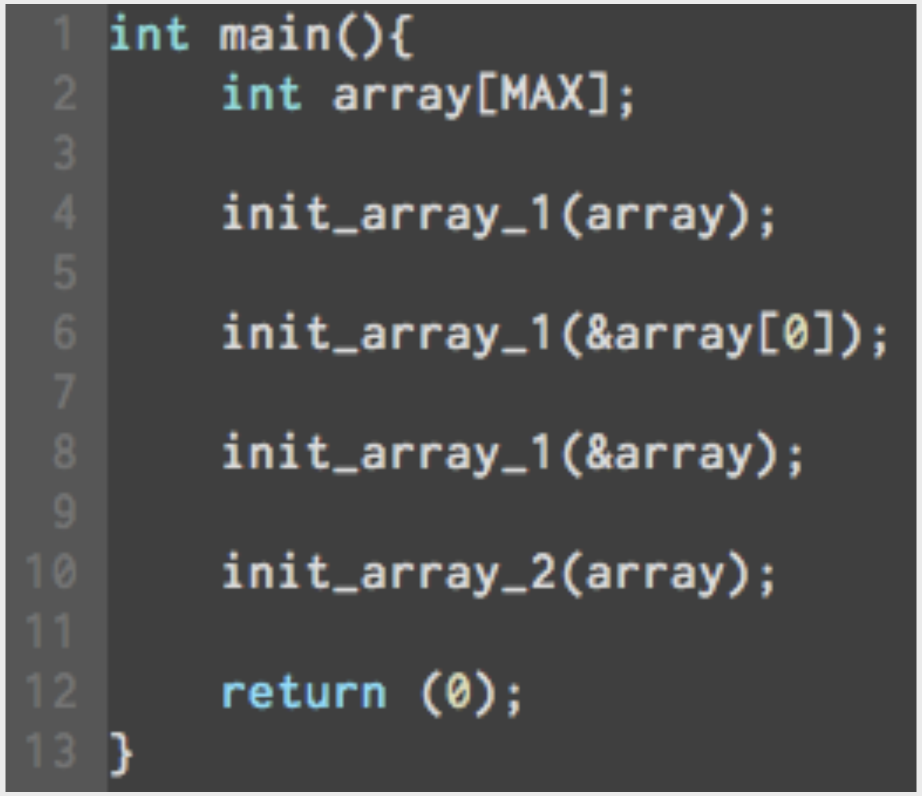
```
int main(){
    int array[MAX];

    init_array_1(array);

    init_array_1(&array[0]);

    init_array_1(&array);

    init_array_2(array);

    return (0);
}
```

Passing an Array to a Function

- When passing an array to a procedure, C will automatically change the array into a pointer
- So, if you put **&** before the array, C will issue a warning



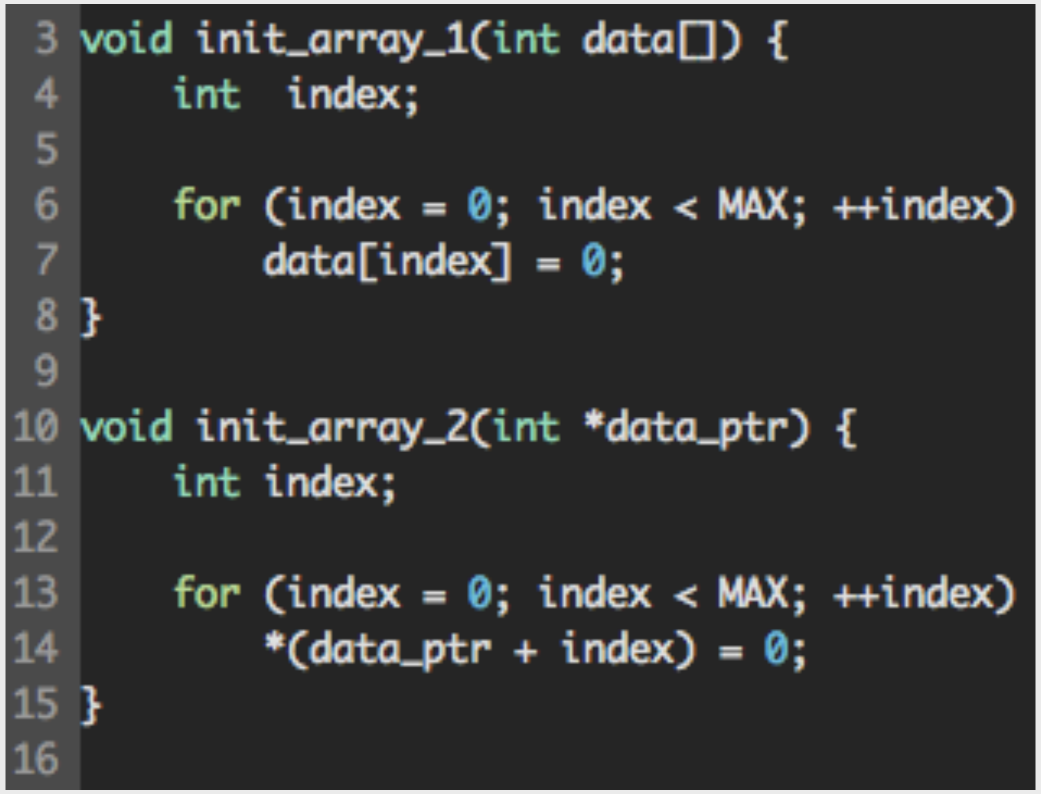
```
void init_array_1(int data[]) {
    int  index;

    for (index = 0; index < MAX; ++index)
        data[index] = 0;
}

void init_array_2(int *data_ptr) {
    int index;

    for (index = 0; index < MAX; ++index)
        *(data_ptr + index) = 0;
}

```



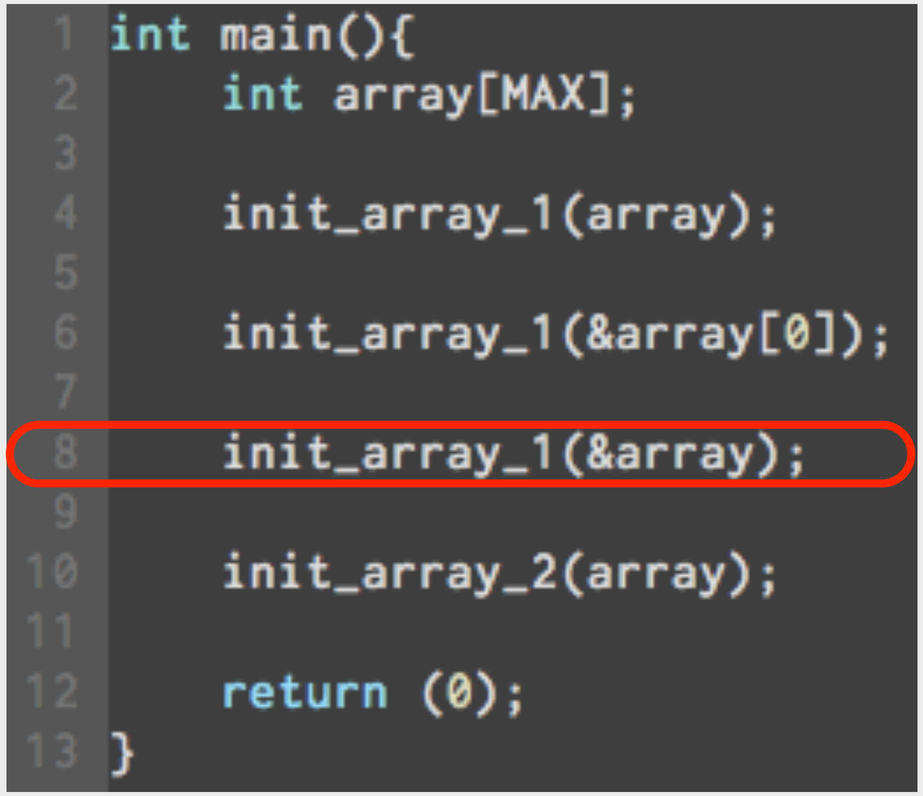
```
int main(){
    int array[MAX];

    init_array_1(array);

    init_array_1(&array[0]);

    init_array_1(&array);

    init_array_2(array);

    return (0);
}
```

Passing an Array to a Function

- When passing an array to a procedure, C will automatically change the array into a pointer
- So, if you put **&** before the array, C will issue a warning



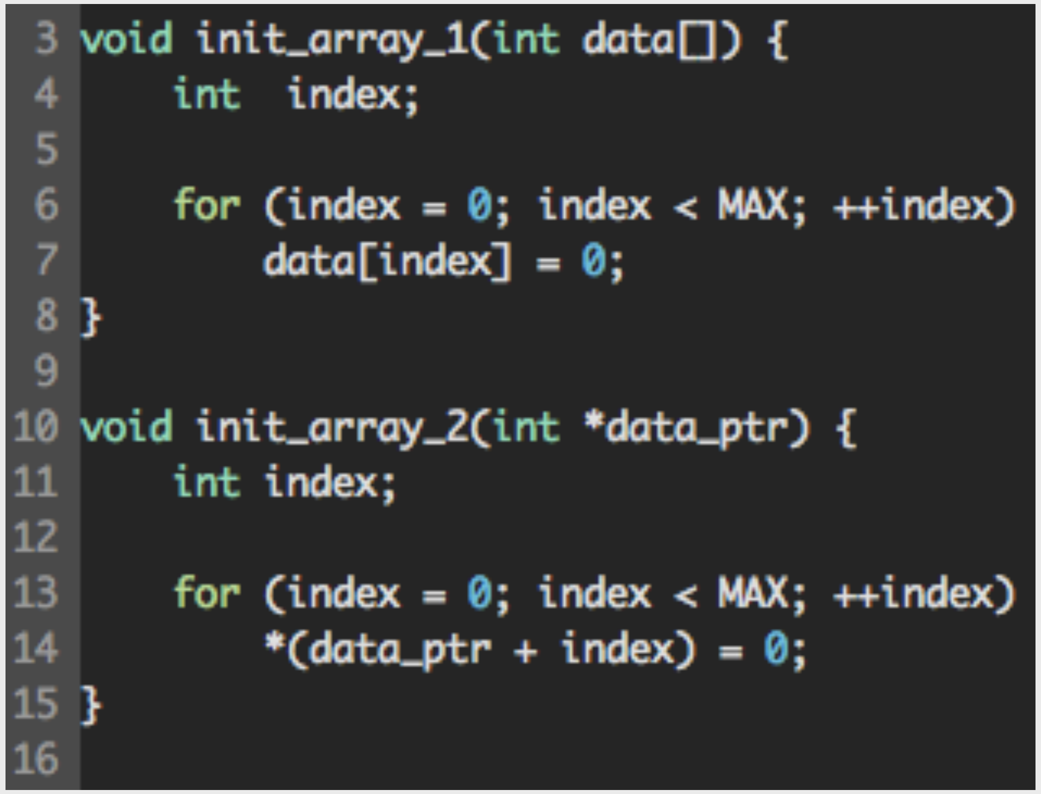
```c
void init_array_1(int data[]) {
    int  index;

    for (index = 0; index < MAX; ++index)
        data[index] = 0;
}

void init_array_2(int *data_ptr) {
    int index;

    for (index = 0; index < MAX; ++index)
        *(data_ptr + index) = 0;
}

```



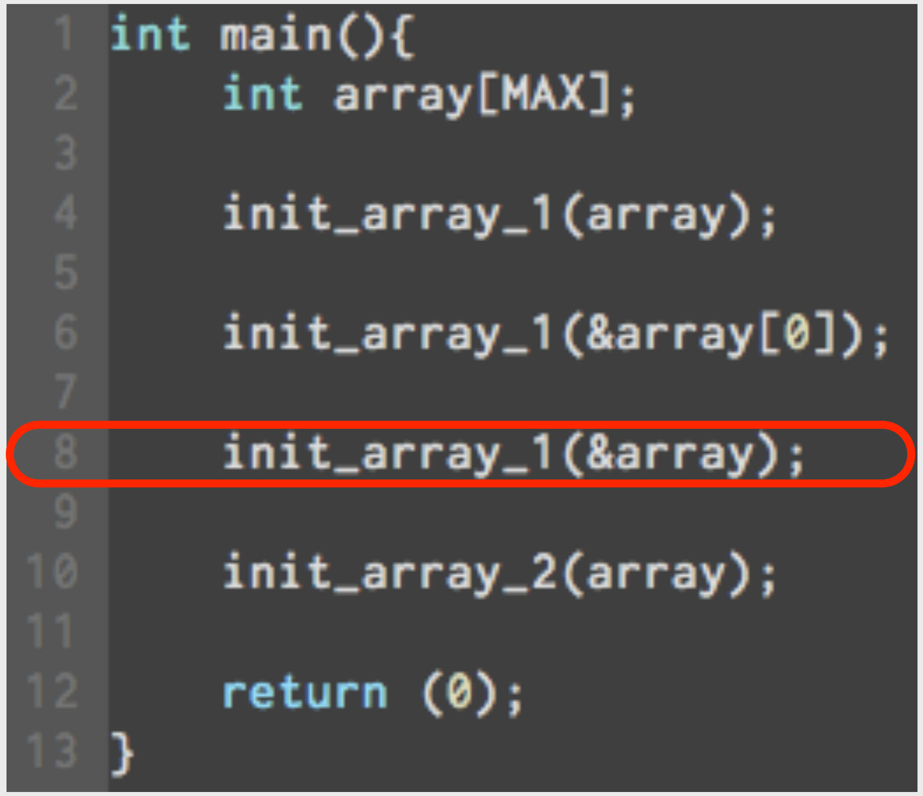
```c
int main(){
    int array[MAX];

    init_array_1(array);

    init_array_1(&array[0]);

    init_array_1(&array);

    init_array_2(array);

    return (0);
}
```

Passing an Array to a Function

- When passing an array to a procedure, C will automatically change the array into a pointer
- So, if you put **&** before the array, C will issue a warning



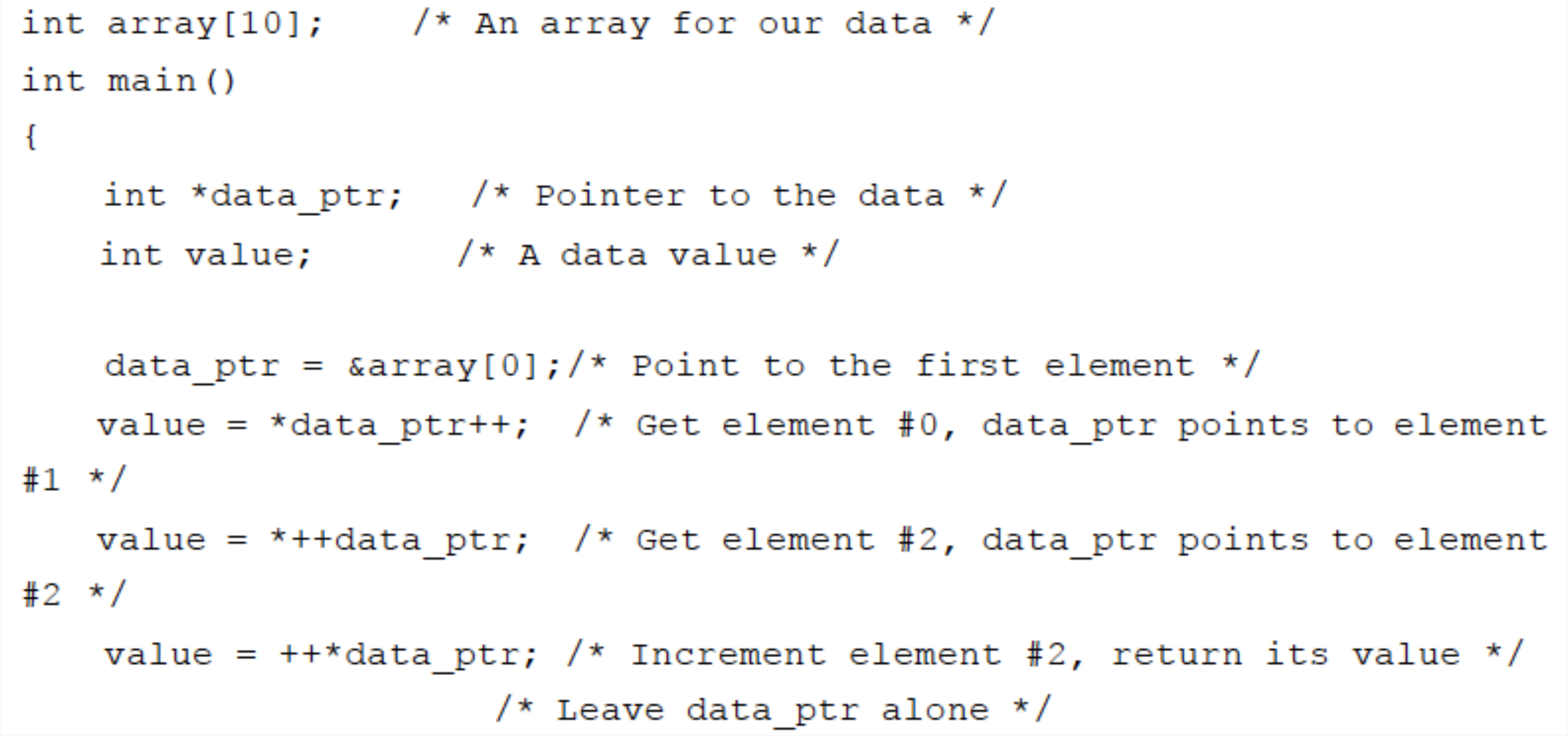
```c
void init_array_1(int data[]) {
    int  index;

    for (index = 0; index < MAX; ++index)
        data[index] = 0;
}

void init_array_2(int *data_ptr) {
    int index;

    for (index = 0; index < MAX; ++index)
        *(data_ptr + index) = 0;
}

```



```c
int main(){
    int array[MAX];

    init_array_1(array);

    init_array_1(&array[0]);

    init_array_1(&array);

    init_array_2(array);

    return (0);
}
```

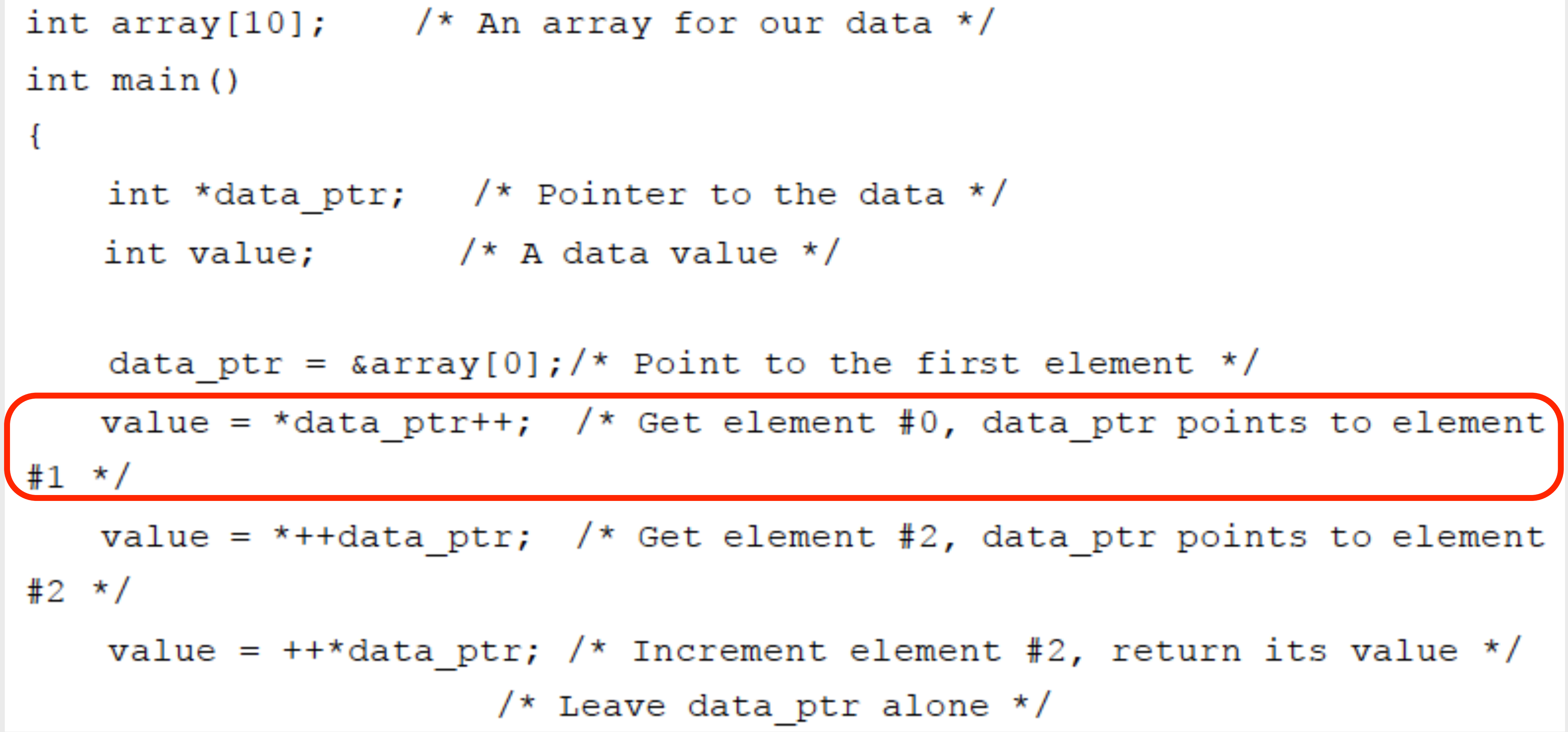
Warning!!

Bad Pointer Usage



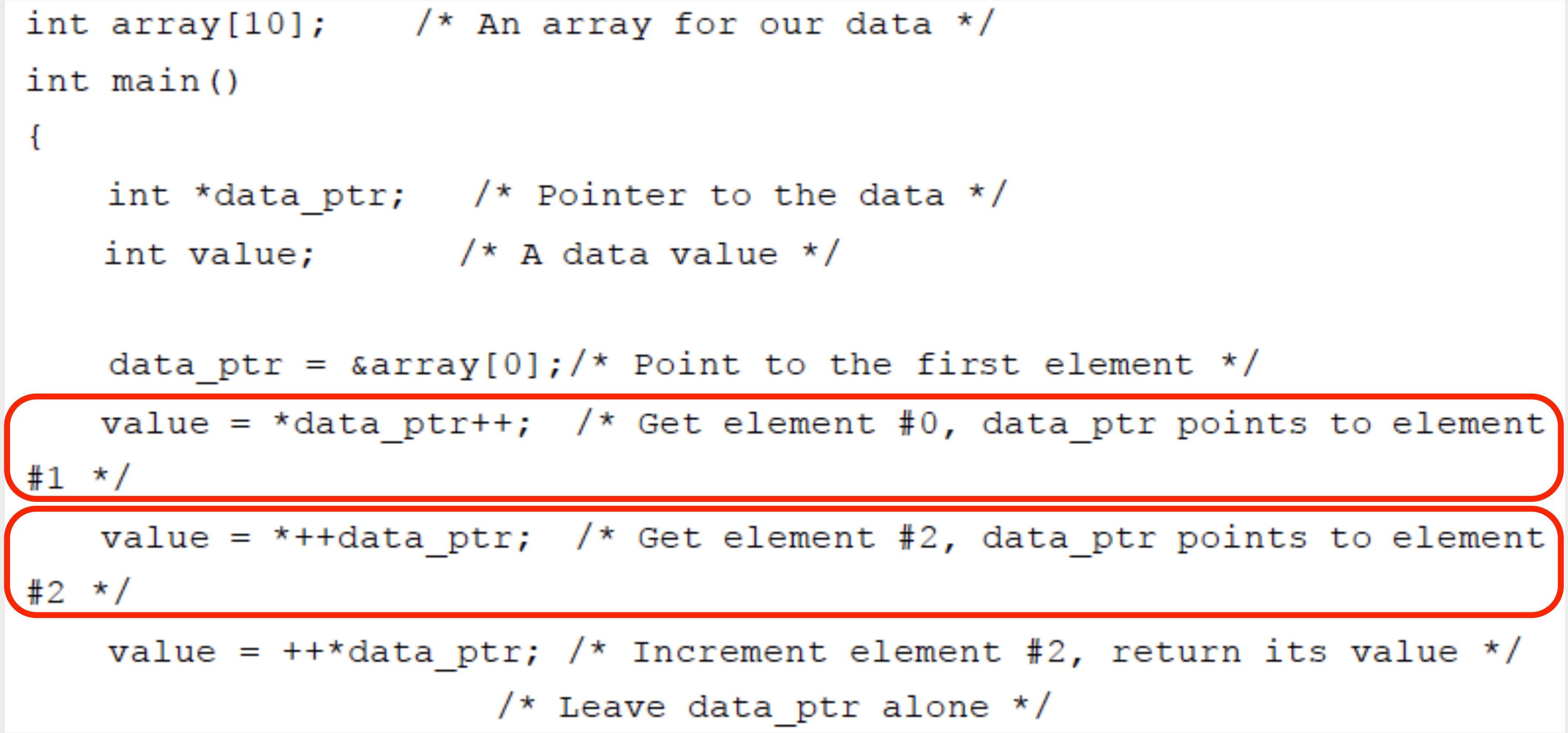
```
int array[10];    /* An array for our data */
int main()
{
    int *data_ptr;   /* Pointer to the data */
    int value;       /* A data value */

    data_ptr = &array[0];/* Point to the first element */
    value = *data_ptr++;  /* Get element #0, data_ptr points to element
#1 */
    value = *++data_ptr;  /* Get element #2, data_ptr points to element
#2 */
    value = ++*data_ptr; /* Increment element #2, return its value */
                       /* Leave data_ptr alone */
```

Bad Pointer Usage



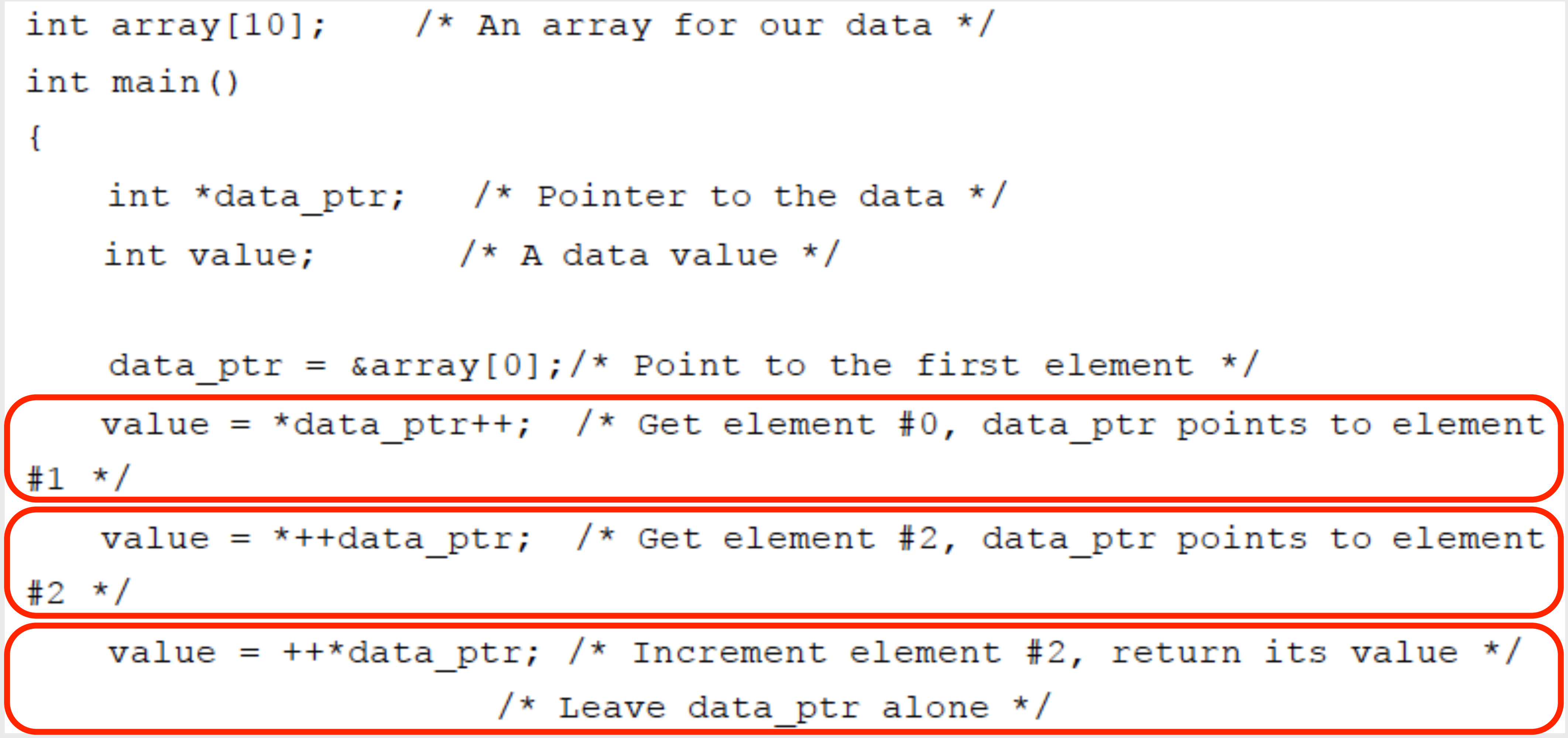
```
int array[10];    /* An array for our data */
int main()
{
    int *data_ptr;   /* Pointer to the data */
    int value;       /* A data value */

    data_ptr = &array[0];/* Point to the first element */
    value = *data_ptr++;  /* Get element #0, data_ptr points to element
#1 */
    value = *++data_ptr;  /* Get element #2, data_ptr points to element
#2 */
    value = ++*data_ptr; /* Increment element #2, return its value */
                          /* Leave data_ptr alone */
```

Bad Pointer Usage



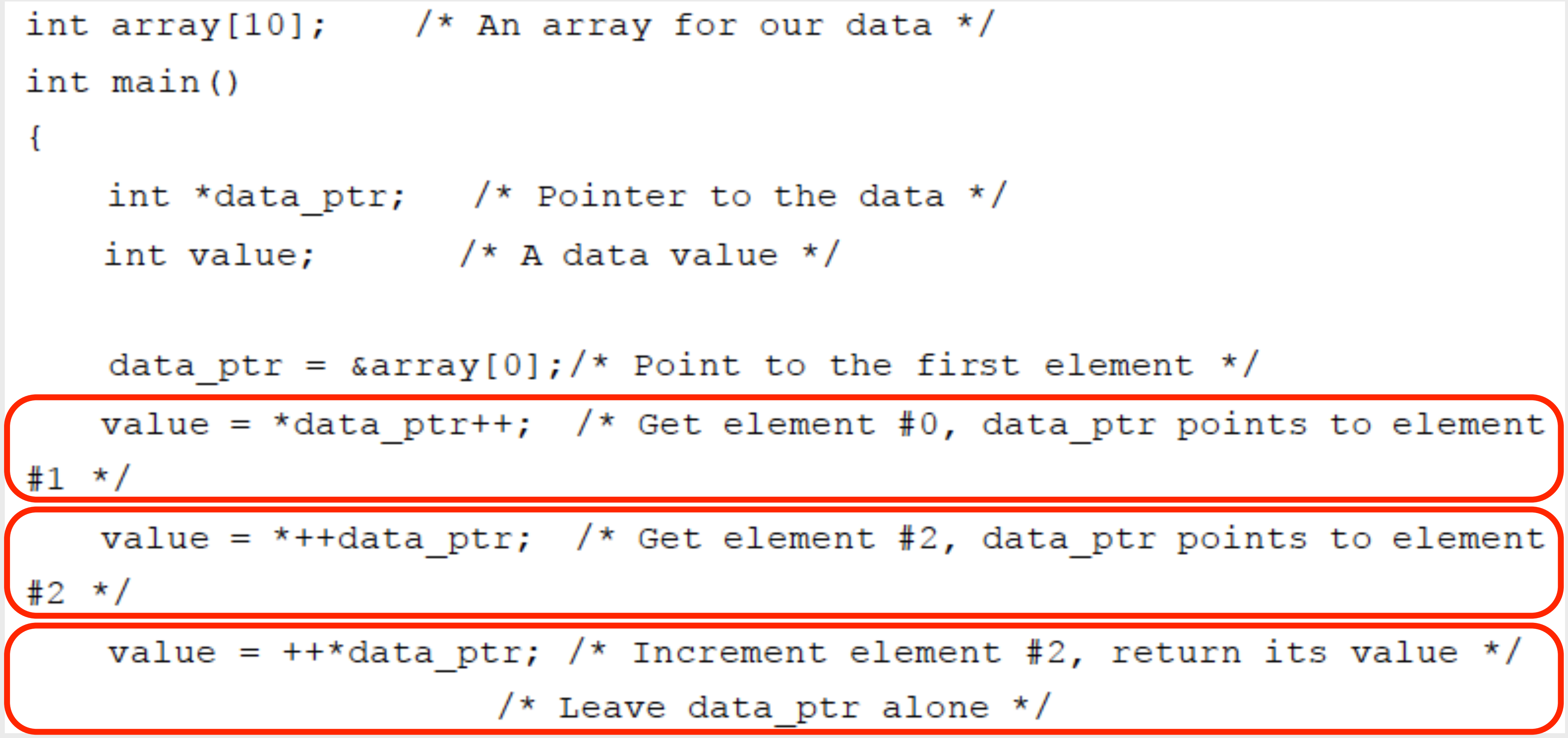
```
int array[10];    /* An array for our data */
int main()
{
    int *data_ptr;    /* Pointer to the data */
    int value;        /* A data value */

    data_ptr = &array[0];/* Point to the first element */
    value = *data_ptr++;  /* Get element #0, data_ptr points to element
#1  */
    value = *++data_ptr;  /* Get element #2, data_ptr points to element
#2  */
    value = ++*data_ptr; /* Increment element #2, return its value */
                          /* Leave data_ptr alone */
```

Bad Pointer Usage



```
int array[10];    /* An array for our data */
int main()
{
    int *data_ptr;   /* Pointer to the data */
    int value;       /* A data value */

    data_ptr = &array[0];/* Point to the first element */
    value = *data_ptr++;  /* Get element #0, data_ptr points to element
#1  */
    value = *++data_ptr;  /* Get element #2, data_ptr points to element
#2  */
    value = ++*data_ptr; /* Increment element #2, return its value */
                        /* Leave data_ptr alone */
```

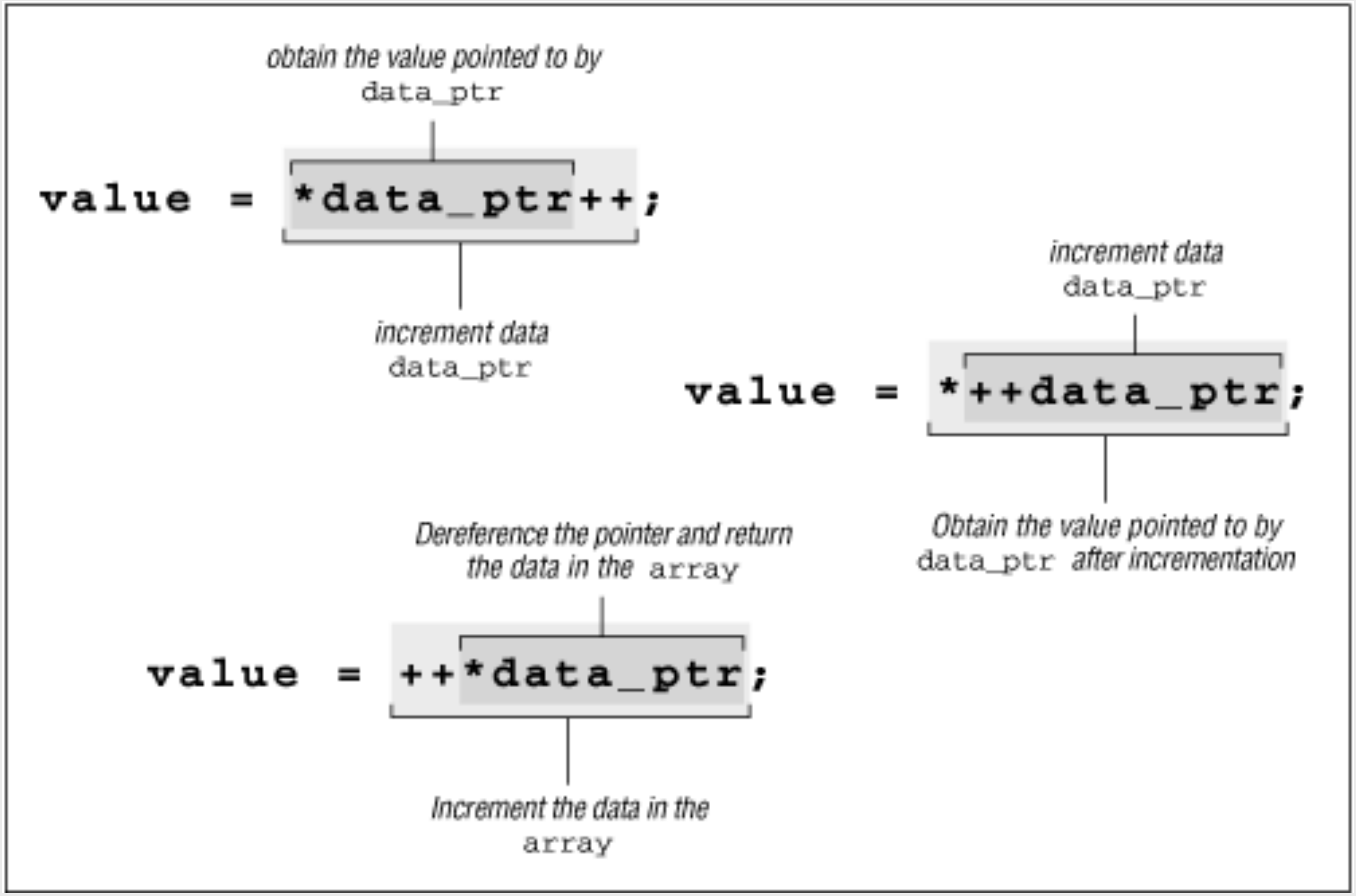
Bad Pointer Usage



```
int array[10];    /* An array for our data */
int main()
{
    int *data_ptr;   /* Pointer to the data */
    int value;       /* A data value */

    data_ptr = &array[0];/* Point to the first element */
    value = *data_ptr++;  /* Get element #0, data_ptr points to element
#1  */
    value = *++data_ptr;  /* Get element #2, data_ptr points to element
#2  */
    value = ++*data_ptr; /* Increment element #2, return its value */
                   /* Leave data_ptr alone */
```

BAD!!!

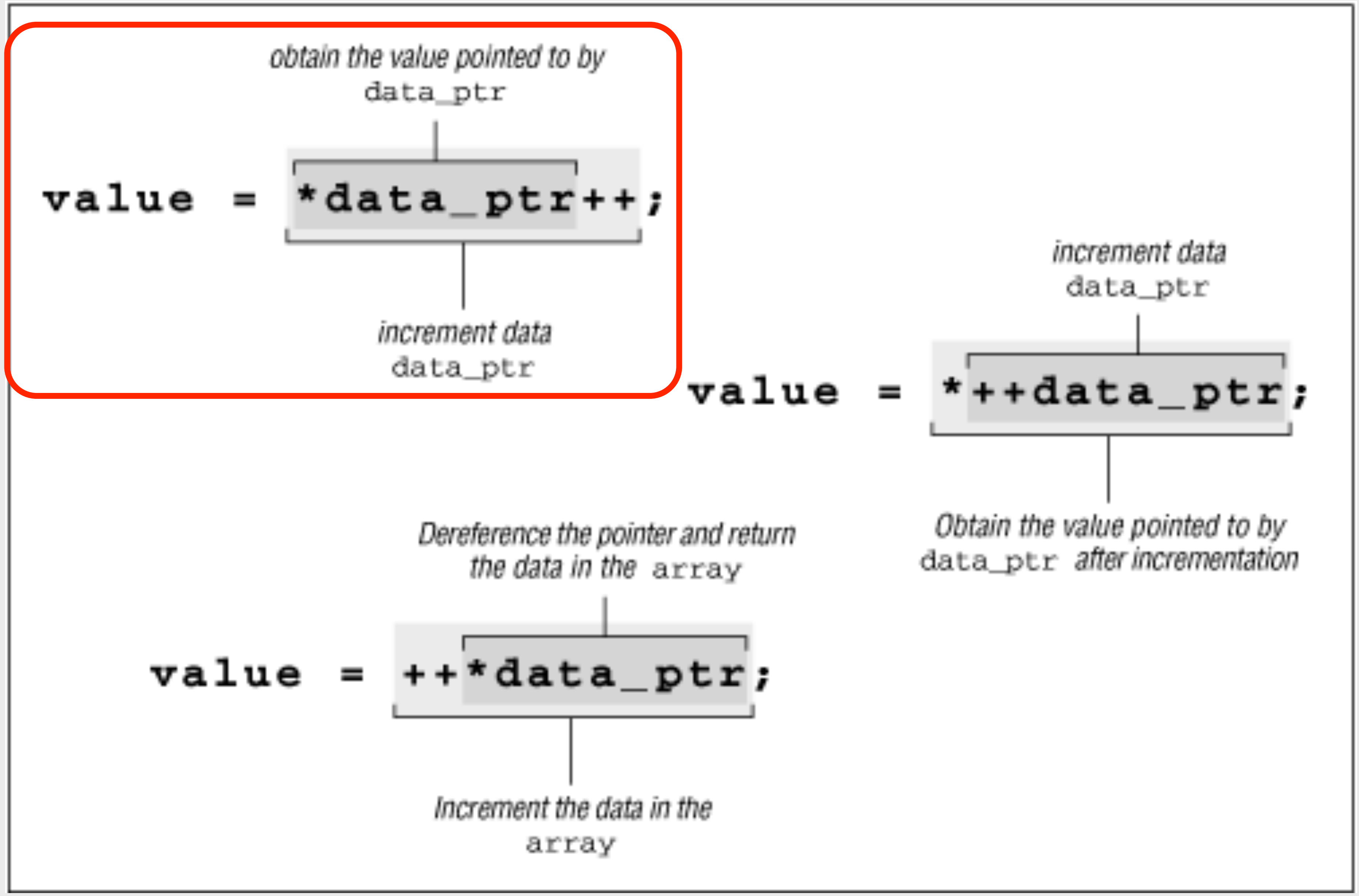
Bad Pointer Usage



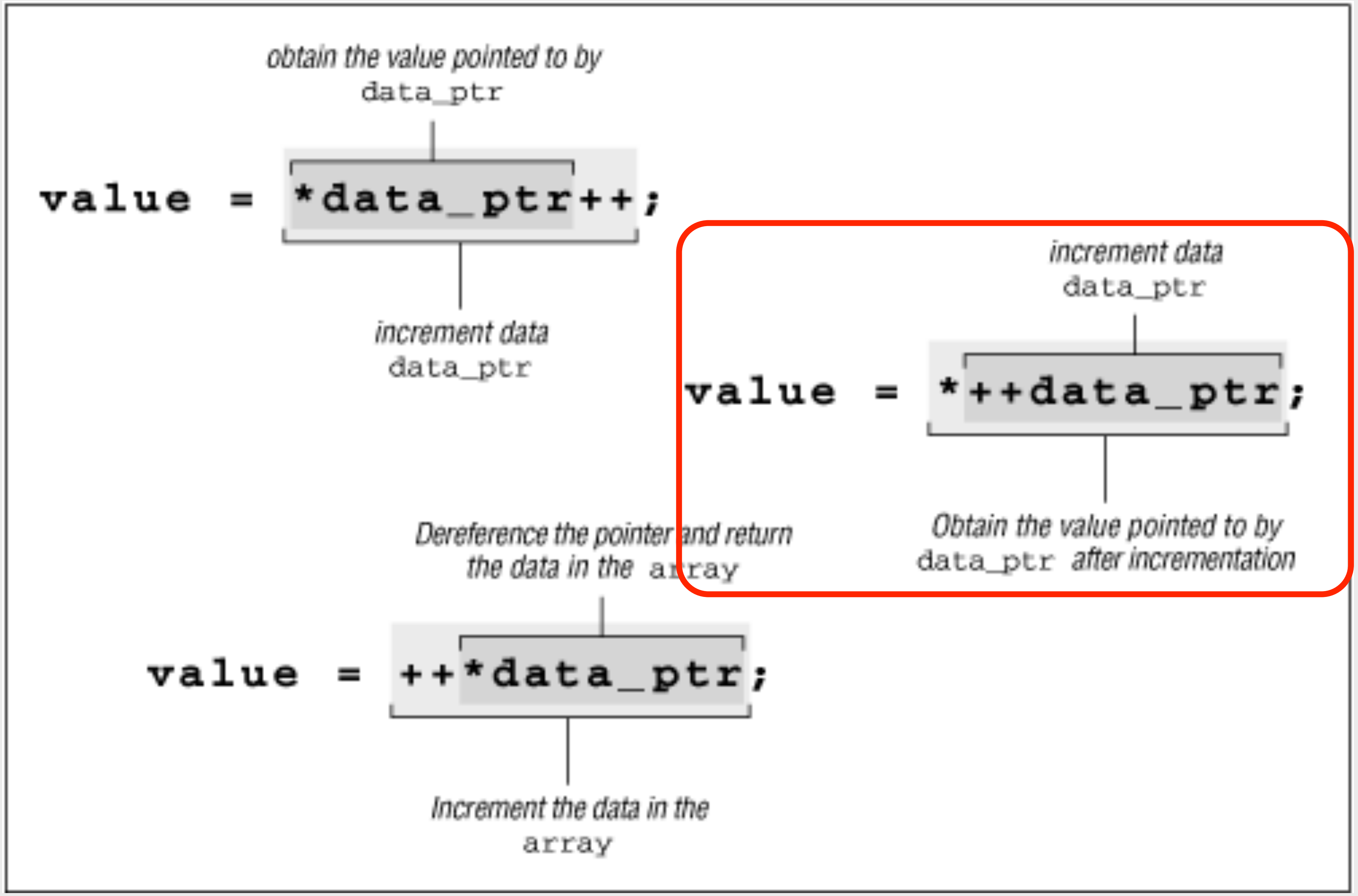
```
value = *data_ptr++;
value = *++data_ptr;
value = ++*data_ptr;
```

obtain the value pointed to by data_ptr
increment data data_ptr
increment data data_ptr
Obtain the value pointed to by data_ptr after incrementation
Dereference the pointer and return the data in the array
Increment the data in the array

Bad Pointer Usage


obtain the value pointed to by data_ptr
value = *data_ptr++;
increment data data_ptr
increment data data_ptr
value = *++data_ptr;
Obtain the value pointed to by data_ptr after incrementation
Dereference the pointer and return the data in the array
value = ++*data_ptr;
Increment the data in the array

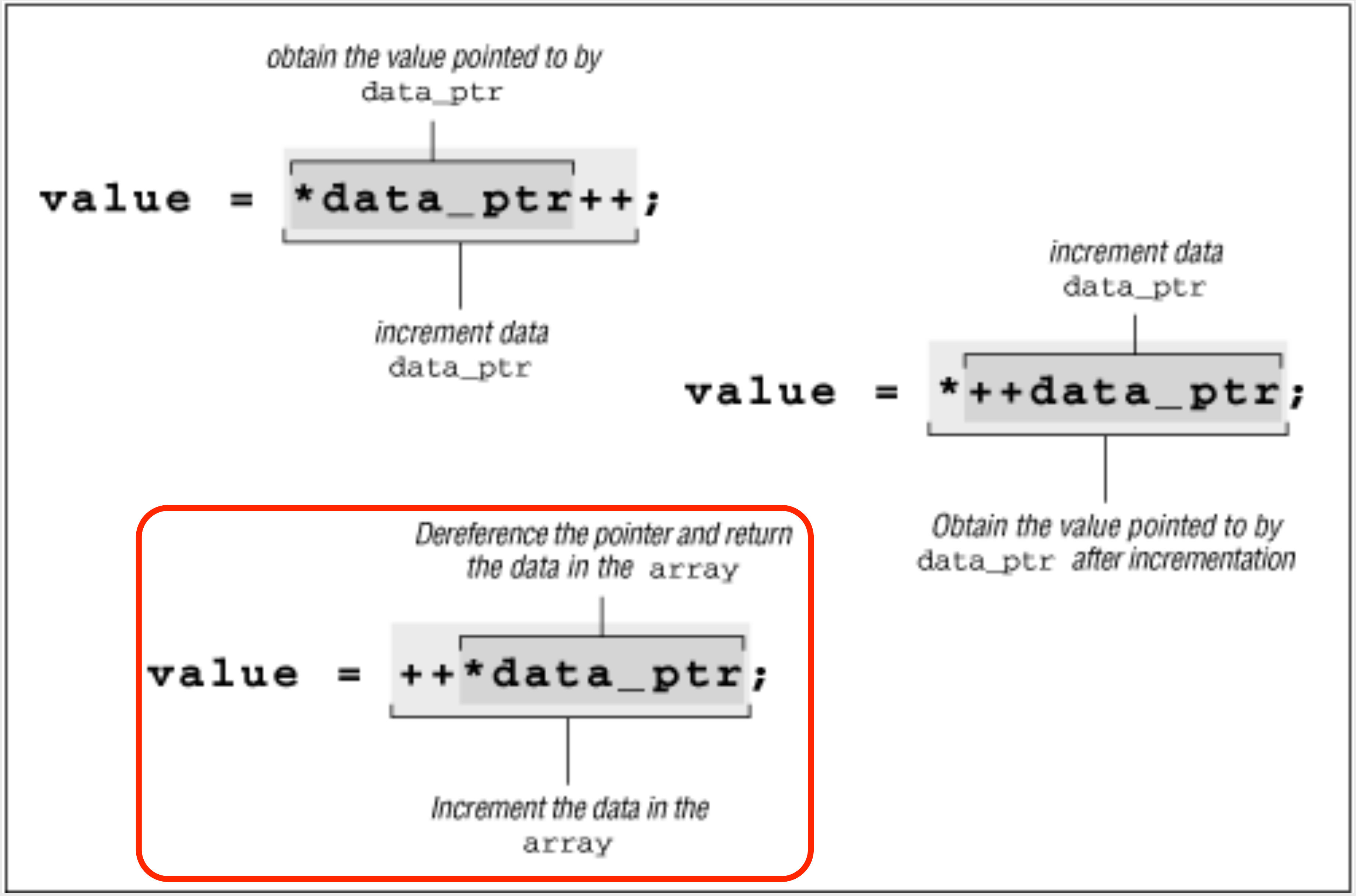
Bad Pointer Usage



```
value = *data_ptr++;
value = *++data_ptr;
value = ++*data_ptr;
```

obtain the value pointed to by data_ptr
increment data data_ptr
increment data data_ptr
Obtain the value pointed to by data_ptr after incrementation
Dereference the pointer and return the data in the array
Increment the data in the array

Bad Pointer Usage



obtain the value pointed to by data_ptr

```
value = *data_ptr++;
```

increment data data_ptr

increment data data_ptr

```
value = *++data_ptr;
```

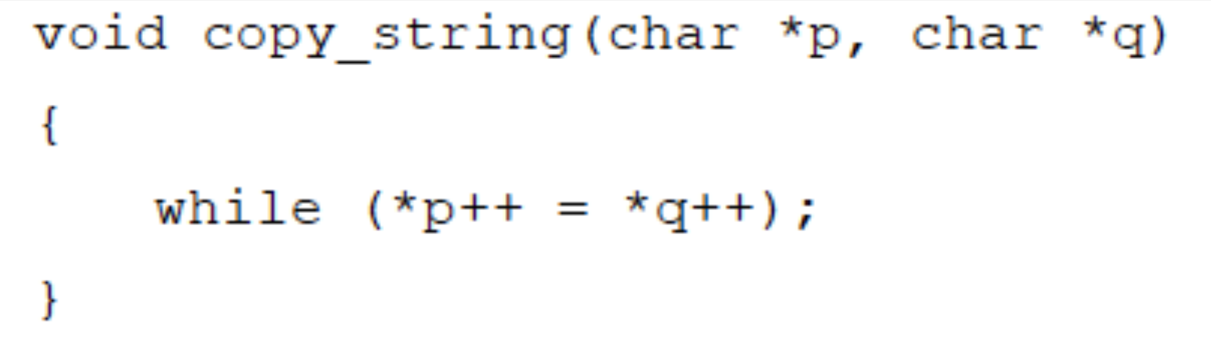
Obtain the value pointed to by data_ptr after incrementation

Dereference the pointer and return the data in the array

```
value = ++*data_ptr;
```

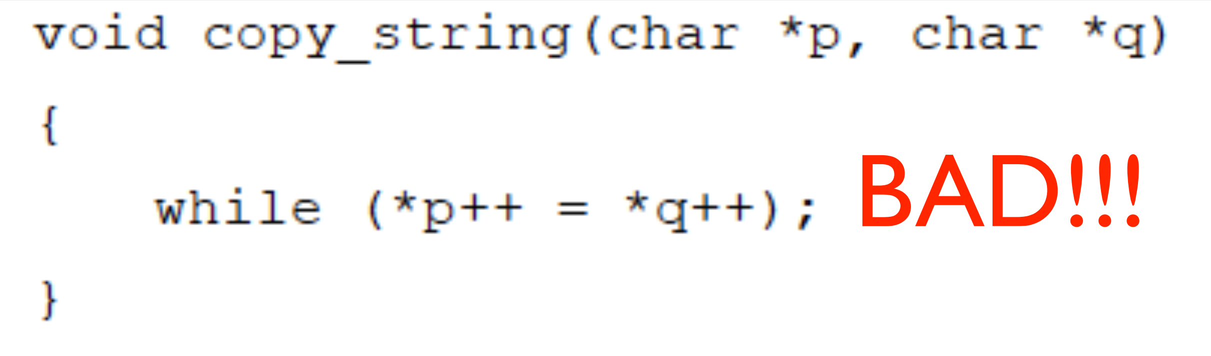
Increment the data in the array

Copy A String



```
void copy_string(char *p, char *q)
{
    while (*p++ = *q++);
}
```

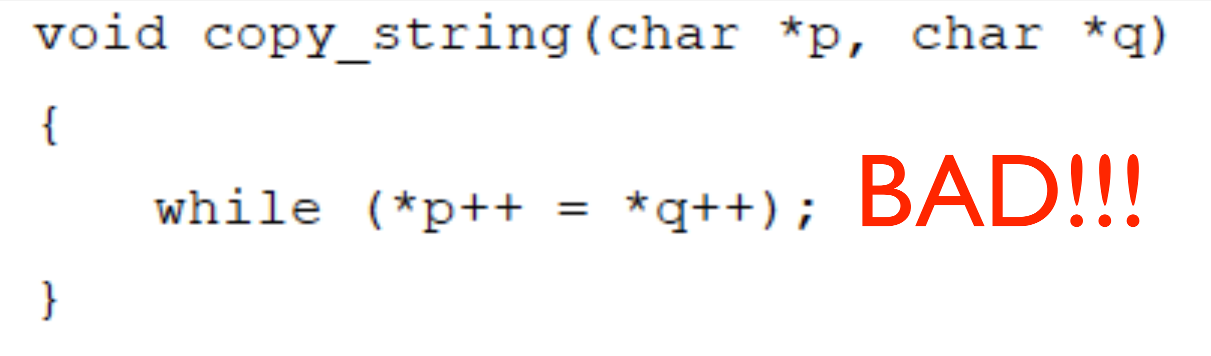
Copy A String



```
void copy_string(char *p, char *q)
{
    while (*p++ = *q++);
}
```

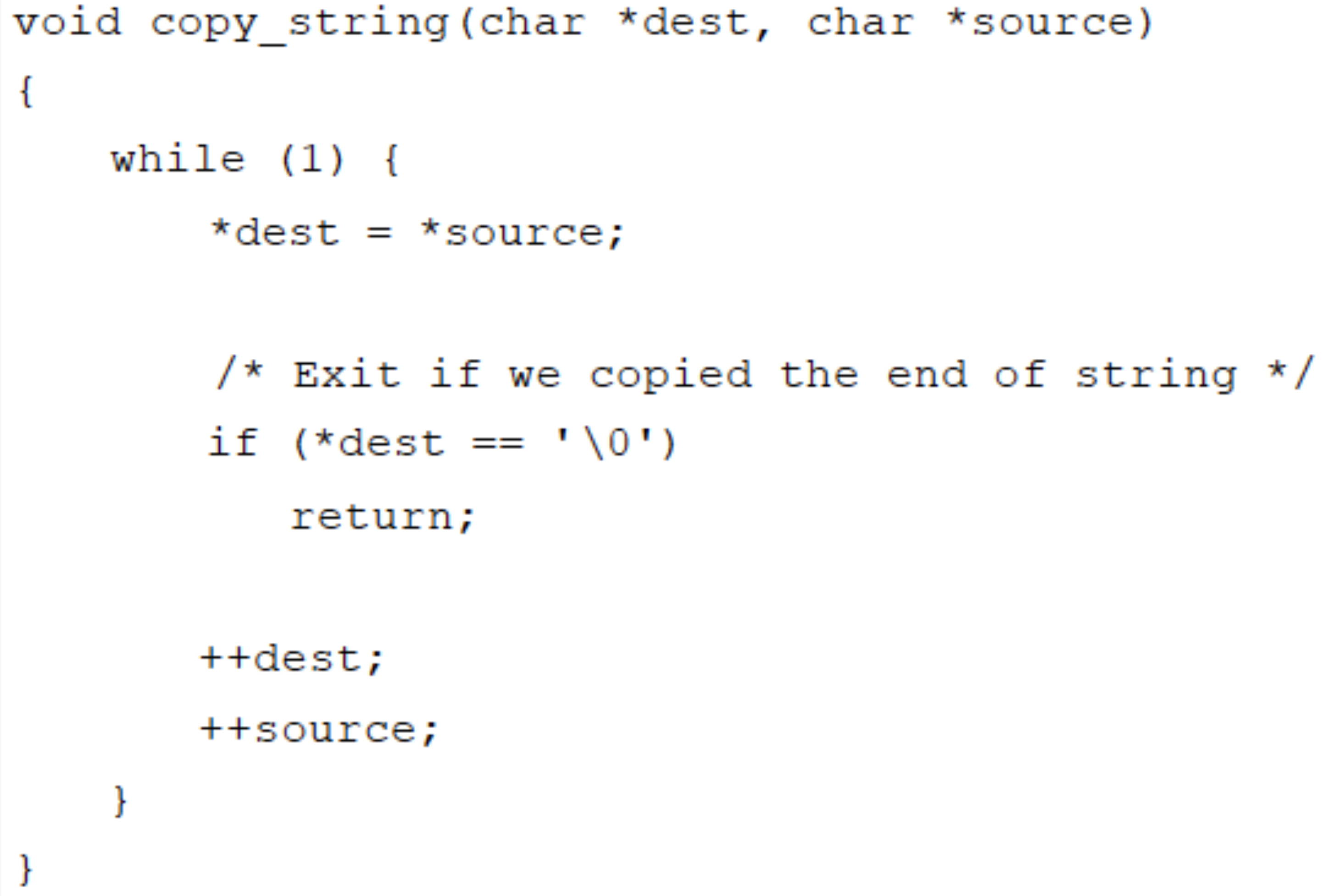
BAD!!!

Copy A String



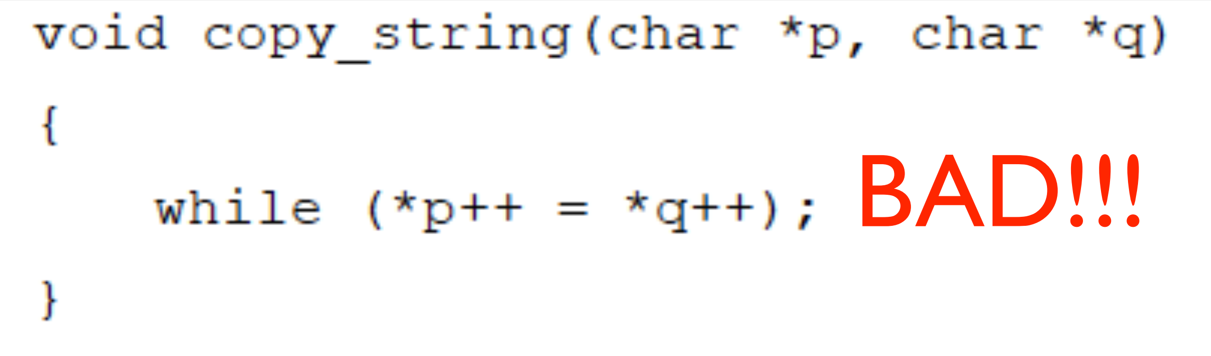
```
void copy_string(char *p, char *q)
{
    while (*p++ = *q++);
}
```

BAD!!!



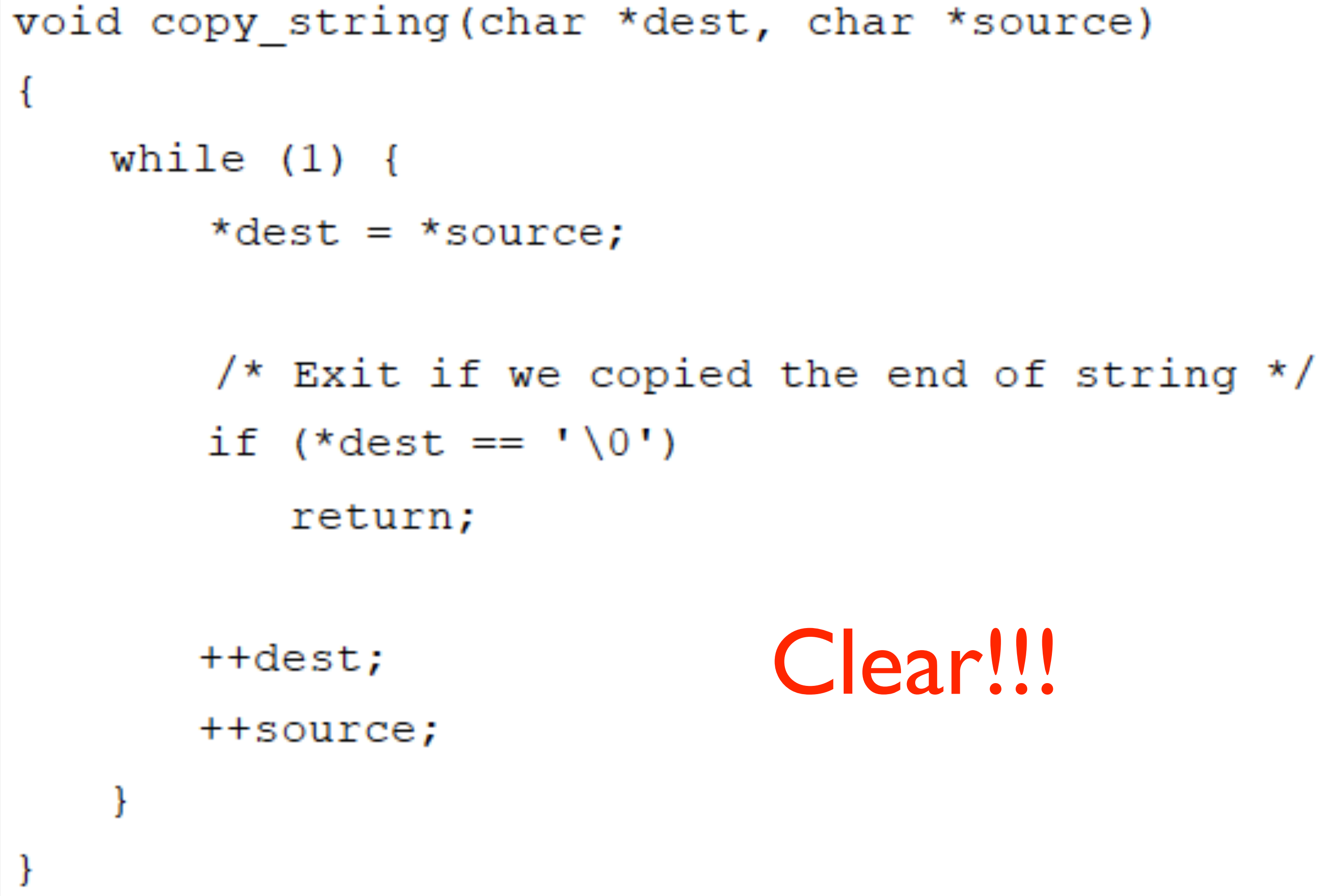
```
void copy_string(char *dest, char *source)
{
    while (1) {
        *dest = *source;

        /* Exit if we copied the end of string */
        if (*dest == '\0')
            return;

        ++dest;
        ++source;
    }
}
```

Copy A String



```
void copy_string(char *p, char *q)
{
    while (*p++ = *q++);
}
```

BAD!!!



```
void copy_string(char *dest, char *source)
{
    while (1) {
        *dest = *source;

        /* Exit if we copied the end of string */
        if (*dest == '\0')
            return;

        ++dest;
        ++source;
    }
}
```

Clear!!!

Command-Line Arguments

Command-Line Arguments

- The **main** function actually takes two arguments

Command-Line Arguments

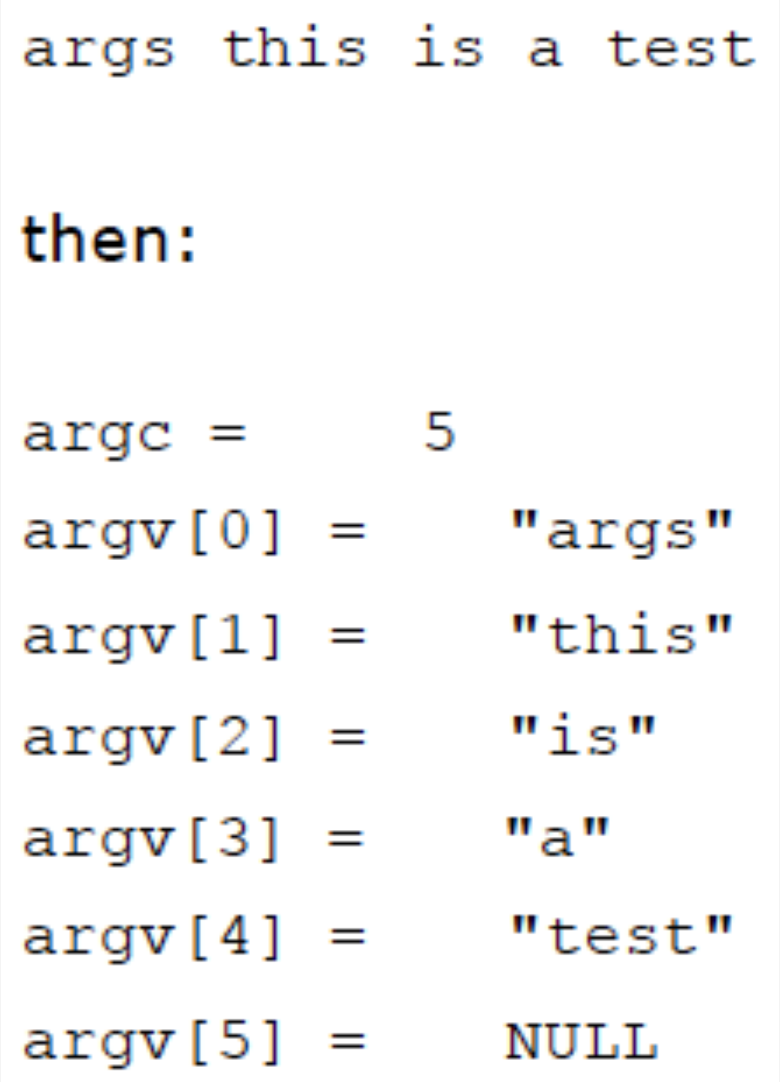
- The **main** function actually takes two arguments
  - **`argc`**: the number of arguments on the command line (including the program name)

Command-Line Arguments

- The **main** function actually takes two arguments
  - **`argc`**: the number of arguments on the command line (including the program name)
  - **`argv`**: contains the actual arguments

Command-Line Arguments

- The **main** function actually takes two arguments
  - **`argc`**: the number of arguments on the command line (including the program name)
  - **`argv`**: contains the actual arguments



```
args this is a test

then:

argc =     5
argv[0] =     "args"
argv[1] =     "this"
argv[2] =     "is"
argv[3] =     "a"
argv[4] =     "test"
argv[5] =     NULL
```

Command-Line Arguments

- A standard Unix command has the form

  ```
  command options file1 file2 file3 ...
  ```

  where options are preceded by a dash (-) and are usually a single letter. For example:

  ```
  print_file [-v] [-llength] [-oname]
  [file1] [file2] ...
  ```

Command-Line Arguments

Command-Line Arguments

- Use a **`while`** loop cycle through the command-line options

```
while ((argc > 1) && (argv[1][0] == ‘-’)) {
```

Command-Line Arguments

- Use a **`while`** loop cycle through the command-line options

```
while ((argc > 1) && (argv[1][0] == ‘-’)) {
```

- At the end of the loop is the code

```
  --argc;
  ++argv;
}
```

Command-Line Arguments

- Use a **`while`** loop cycle through the command-line options

```
while ((argc > 1) && (argv[1][0] == ‘-’)) {
```

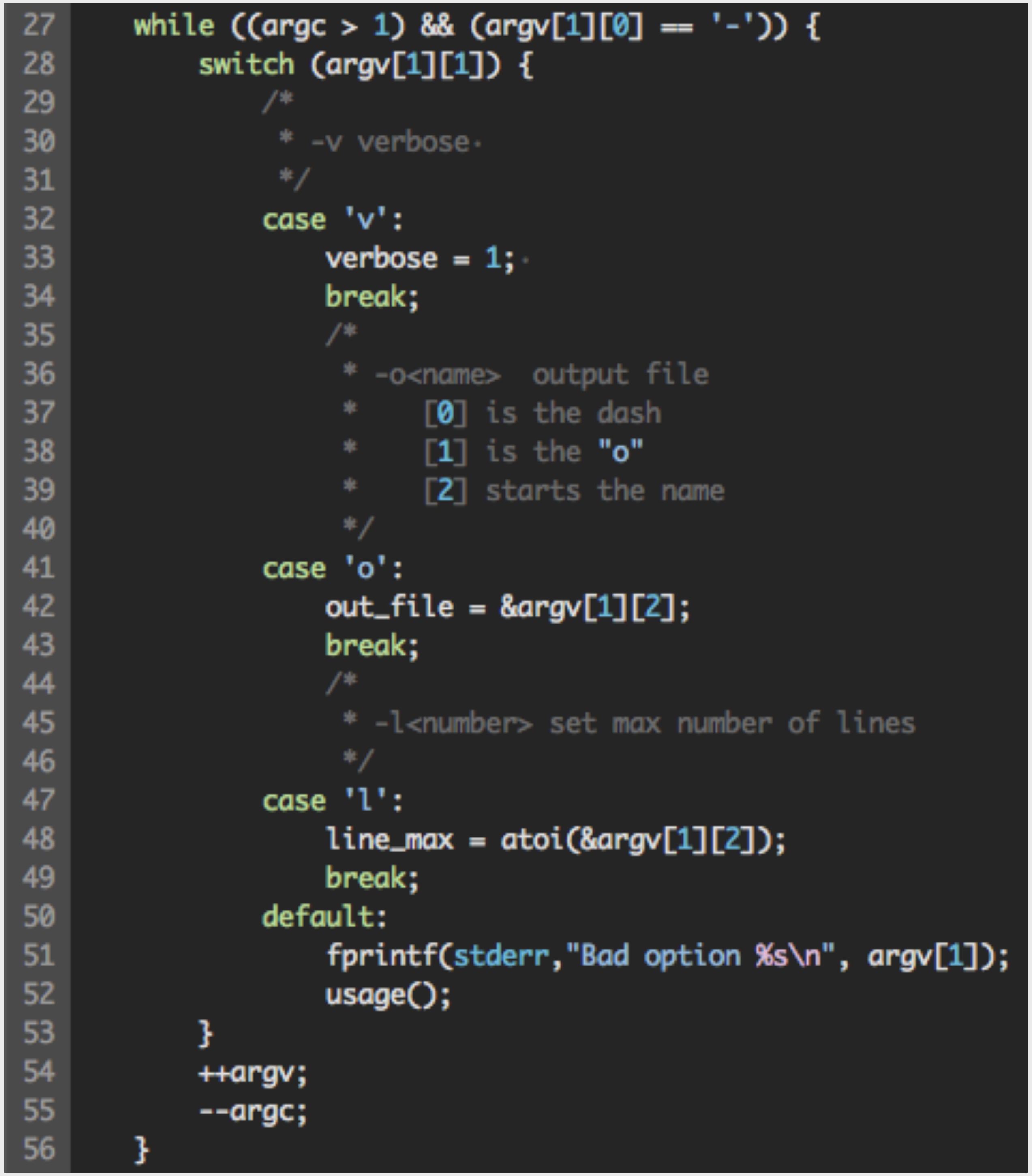
- At the end of the loop is the code

```
    --argc;
    ++argv;
}
```

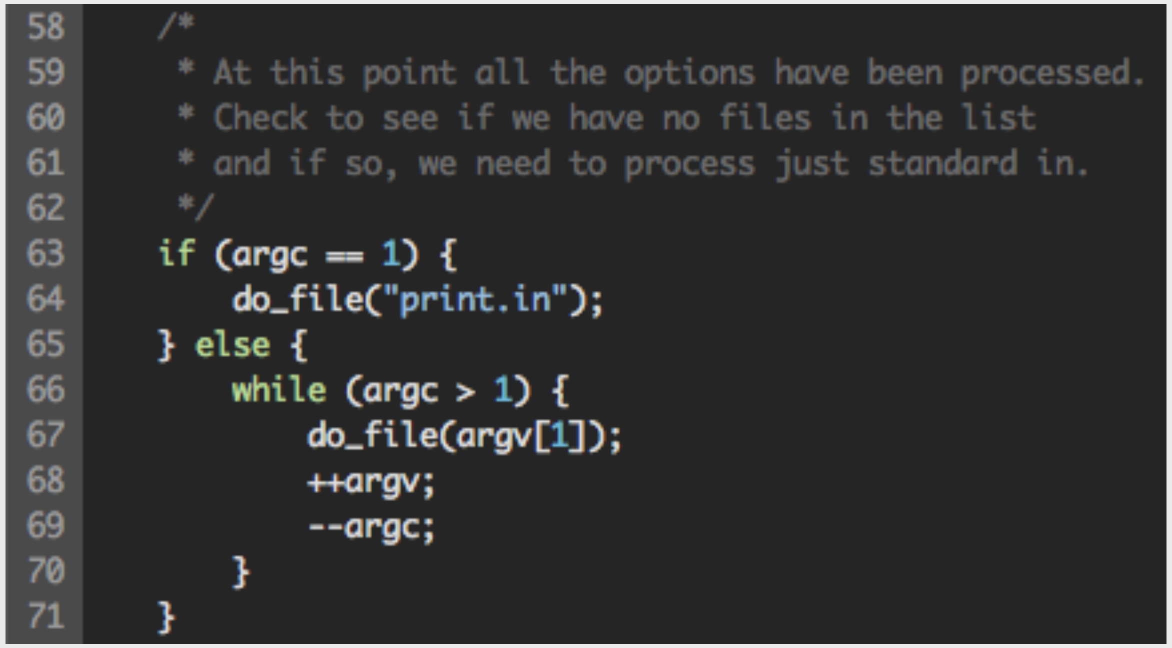
- The **`switch`** statement is used to decode the options

Command-Line Arguments

- Example: print.c



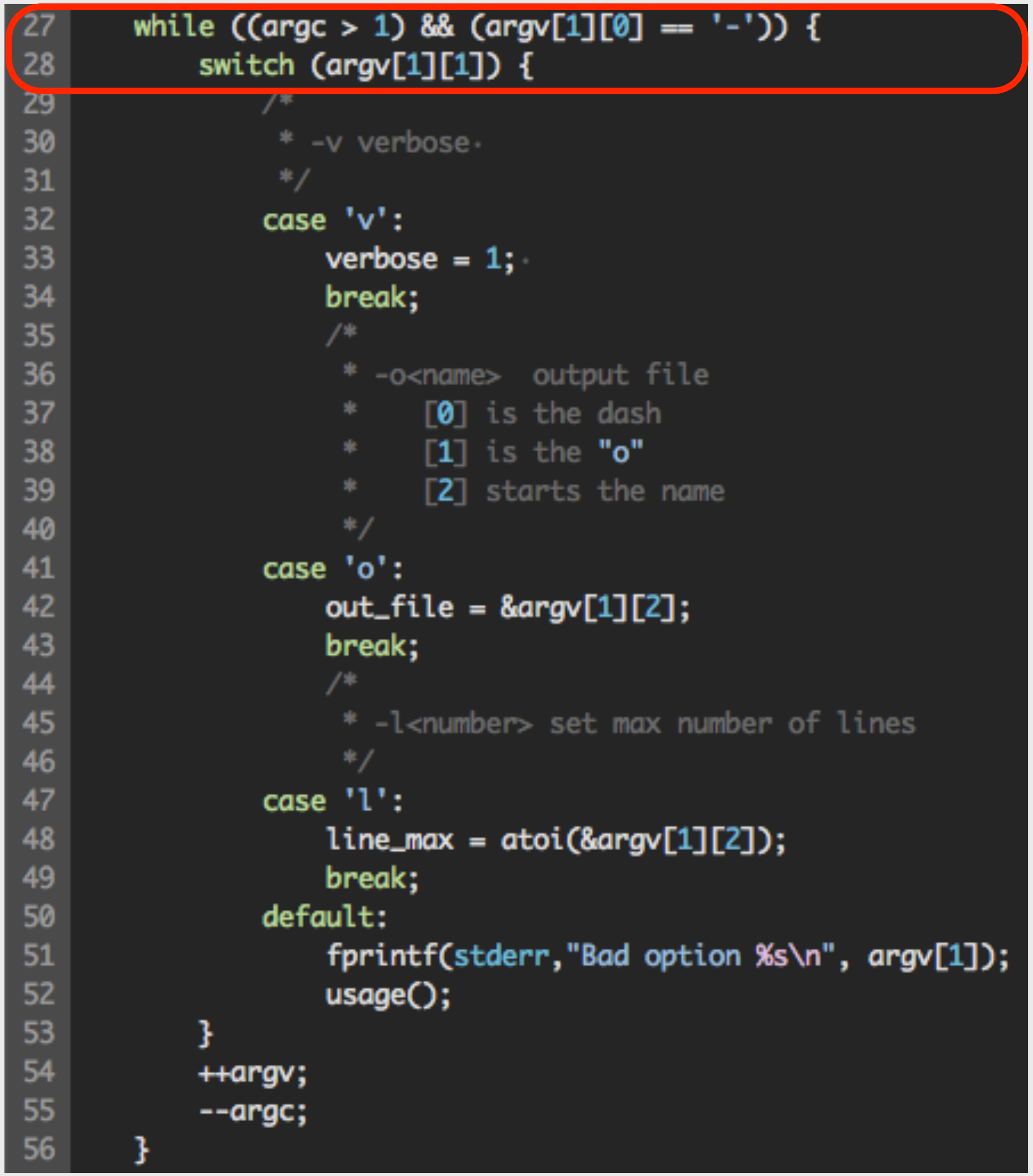
```c
    while ((argc > 1) && (argv[1][0] == '-')) {
        switch (argv[1][1]) {
            /*
             * -v verbose
             */
            case 'v':
                verbose = 1;
                break;
                /*
                 * -o<name>  output file
                 *     [0] is the dash
                 *     [1] is the "o"
                 *     [2] starts the name
                 */
            case 'o':
                out_file = &argv[1][2];
                break;
                /*
                 * -l<number> set max number of lines
                 */
            case 'l':
                line_max = atoi(&argv[1][2]);
                break;
            default:
                fprintf(stderr,"Bad option %s\n", argv[1]);
                usage();
        }
        ++argv;
        --argc;
    }
```



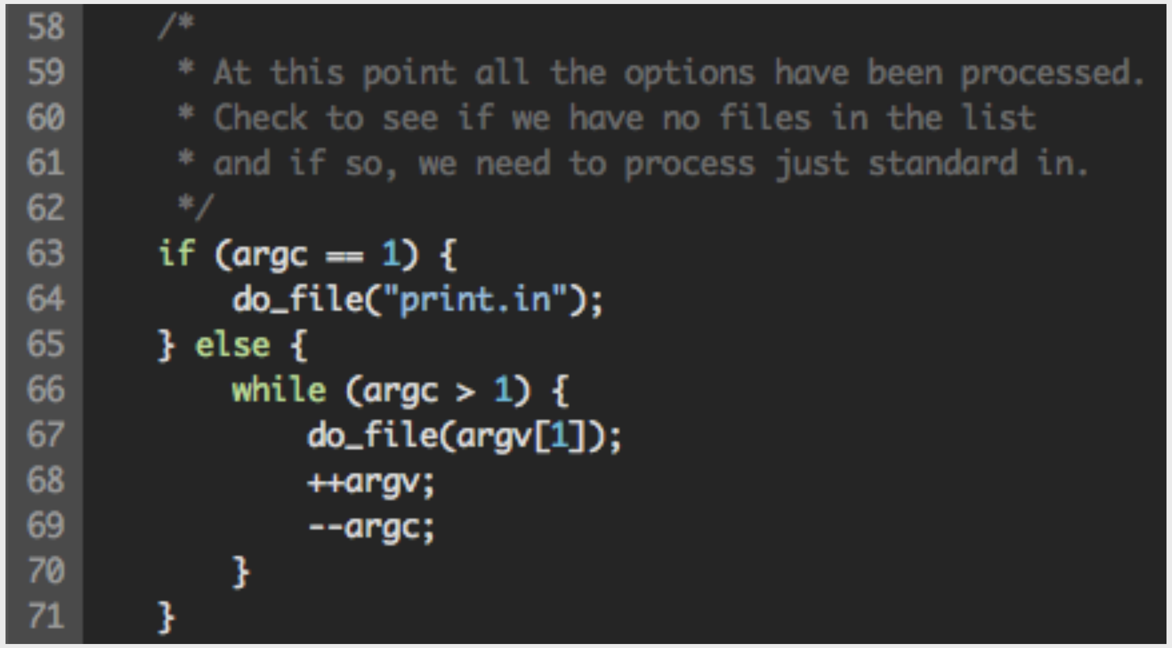
```c
    /*
     * At this point all the options have been processed.
     * Check to see if we have no files in the list
     * and if so, we need to process just standard in.
     */
    if (argc == 1) {
        do_file("print.in");
    } else {
        while (argc > 1) {
            do_file(argv[1]);
            ++argv;
            --argc;
        }
    }
```

Command-Line Arguments

- Example: print.c



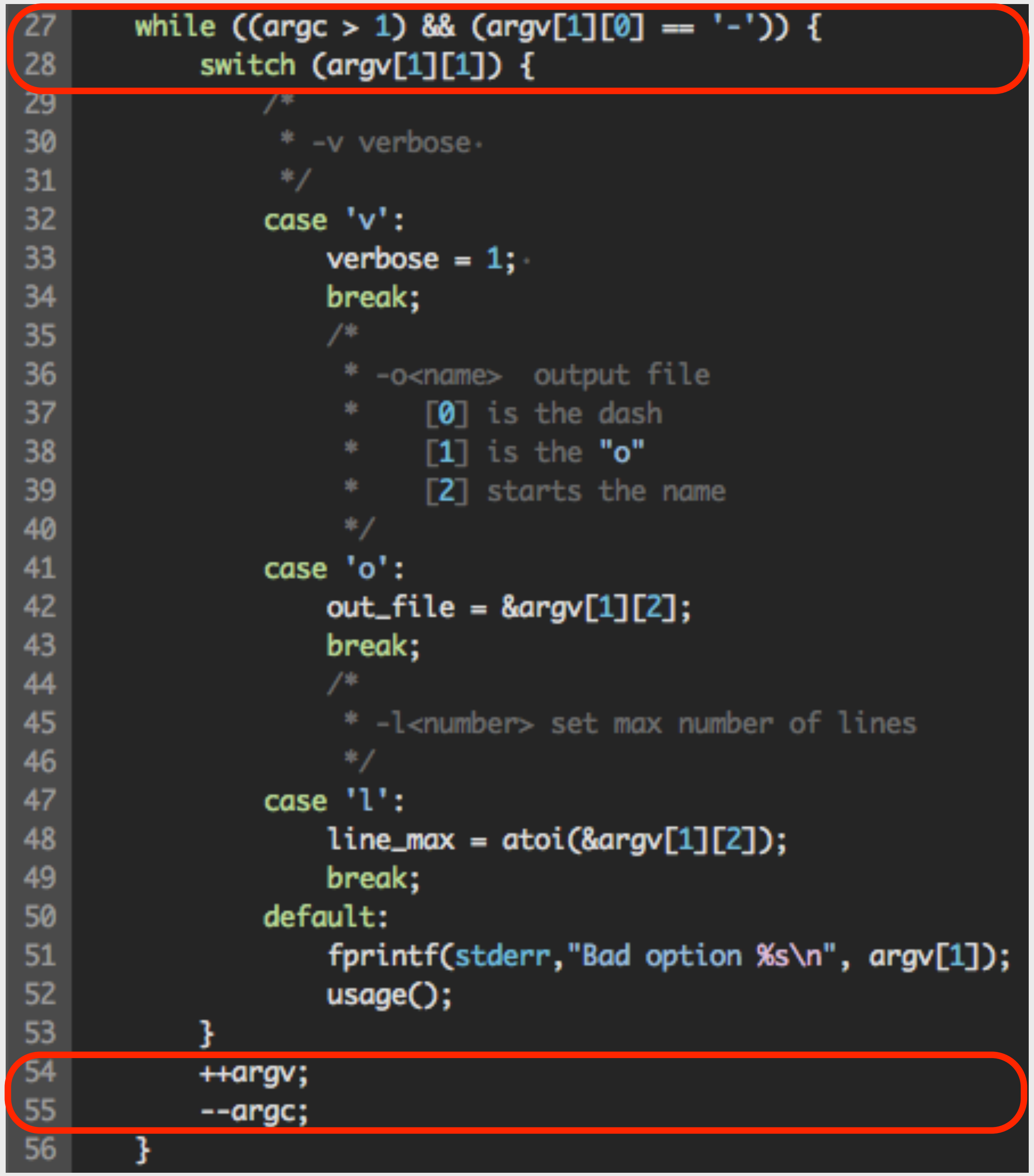
```c
    while ((argc > 1) && (argv[1][0] == '-')) {
        switch (argv[1][1]) {
            /*
             * -v verbose
             */
            case 'v':
                verbose = 1;
                break;
                /*
                 * -o<name>  output file
                 *    [0] is the dash
                 *    [1] is the "o"
                 *    [2] starts the name
                 */
            case 'o':
                out_file = &argv[1][2];
                break;
                /*
                 * -l<number> set max number of lines
                 */
            case 'l':
                line_max = atoi(&argv[1][2]);
                break;
            default:
                fprintf(stderr,"Bad option %s\n", argv[1]);
                usage();
        }
        ++argv;
        --argc;
    }
```



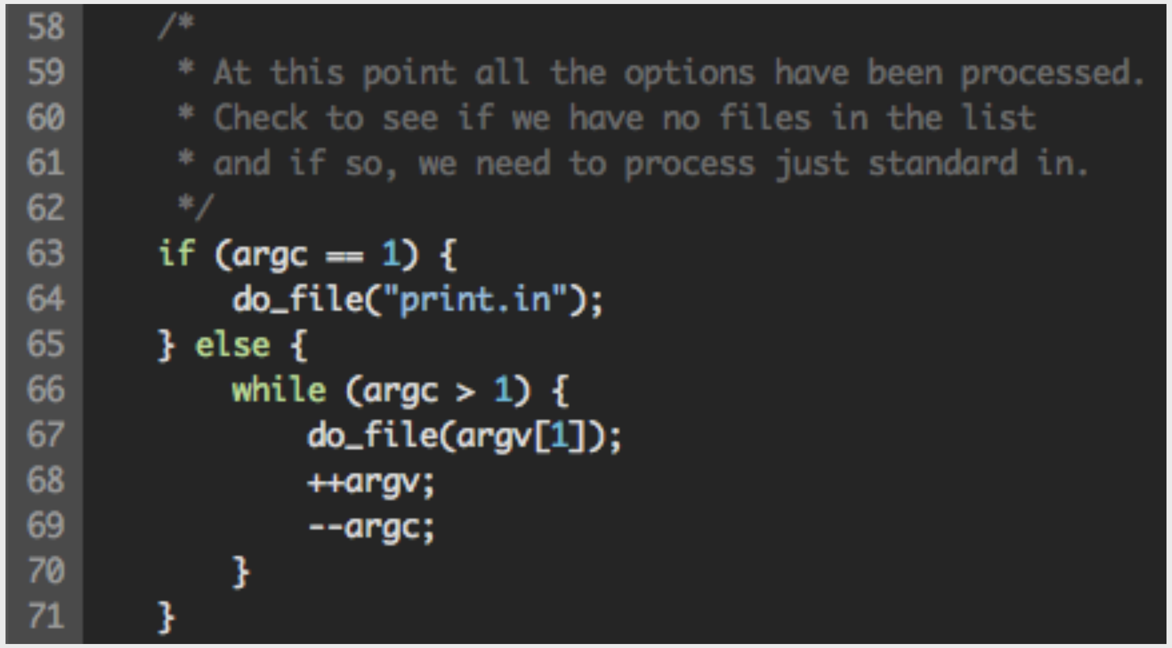
```c
    /*
     * At this point all the options have been processed.
     * Check to see if we have no files in the list
     * and if so, we need to process just standard in.
     */
    if (argc == 1) {
        do_file("print.in");
    } else {
        while (argc > 1) {
            do_file(argv[1]);
            ++argv;
            --argc;
        }
    }
```

Command-Line Arguments

- Example: print.c



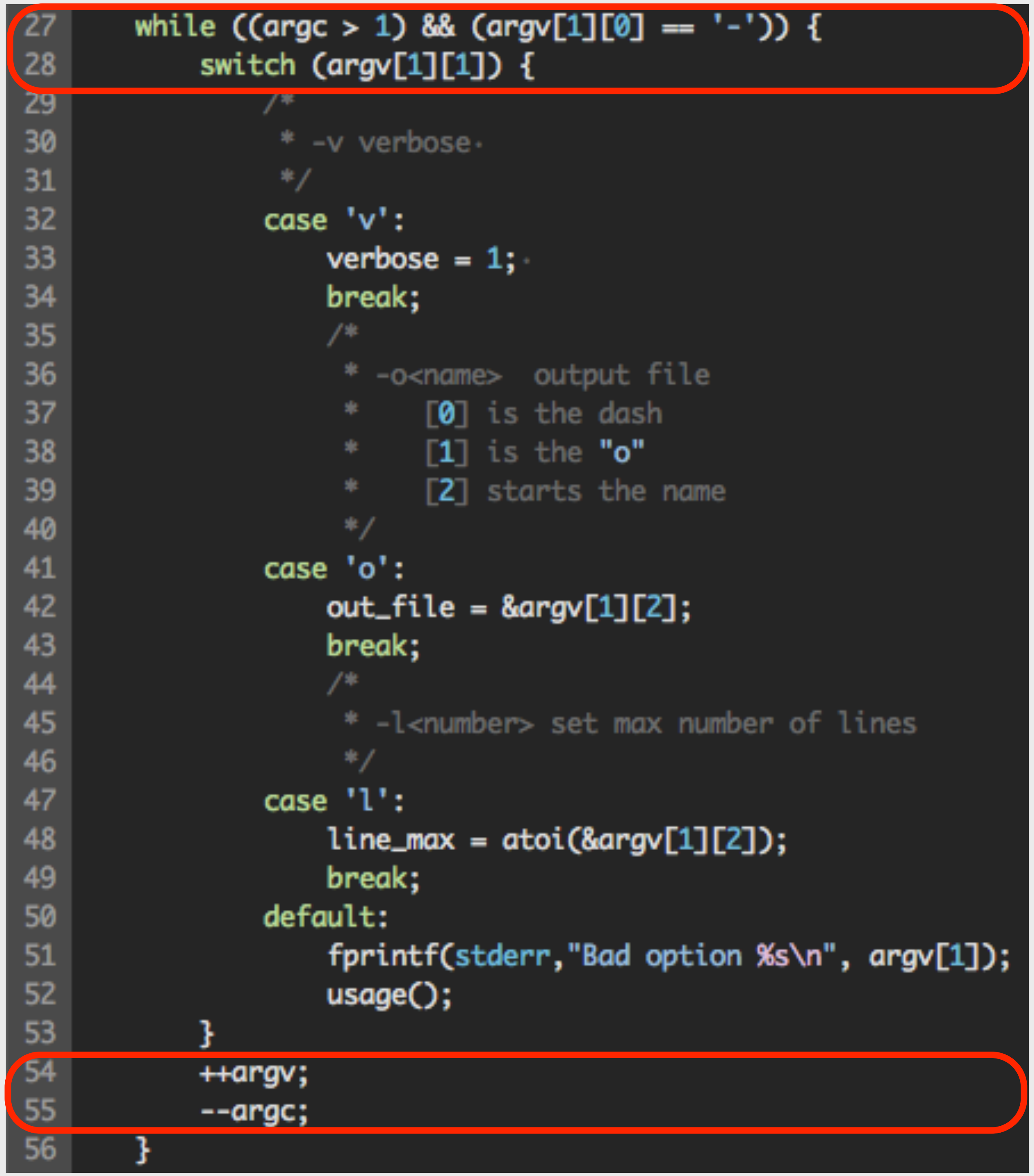
```c
    while ((argc > 1) && (argv[1][0] == '-')) {
        switch (argv[1][1]) {
            /*
             * -v verbose
             */
            case 'v':
                verbose = 1;
                break;
                /*
                 * -o<name>  output file
                 *     [0] is the dash
                 *     [1] is the "o"
                 *     [2] starts the name
                 */
            case 'o':
                out_file = &argv[1][2];
                break;
                /*
                 * -l<number> set max number of lines
                 */
            case 'l':
                line_max = atoi(&argv[1][2]);
                break;
            default:
                fprintf(stderr,"Bad option %s\n", argv[1]);
                usage();
        }
        ++argv;
        --argc;
    }
```



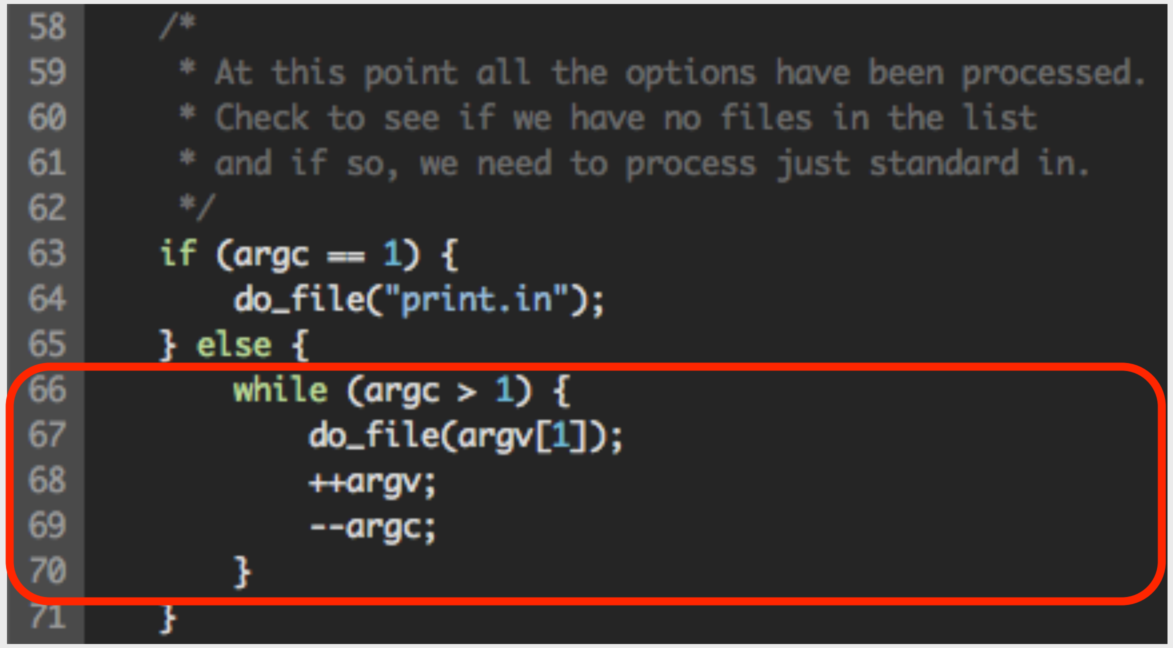
```c
    /*
     * At this point all the options have been processed.
     * Check to see if we have no files in the list
     * and if so, we need to process just standard in.
     */
    if (argc == 1) {
        do_file("print.in");
    } else {
        while (argc > 1) {
            do_file(argv[1]);
            ++argv;
            --argc;
        }
    }
```

Command-Line Arguments

- Example: print.c



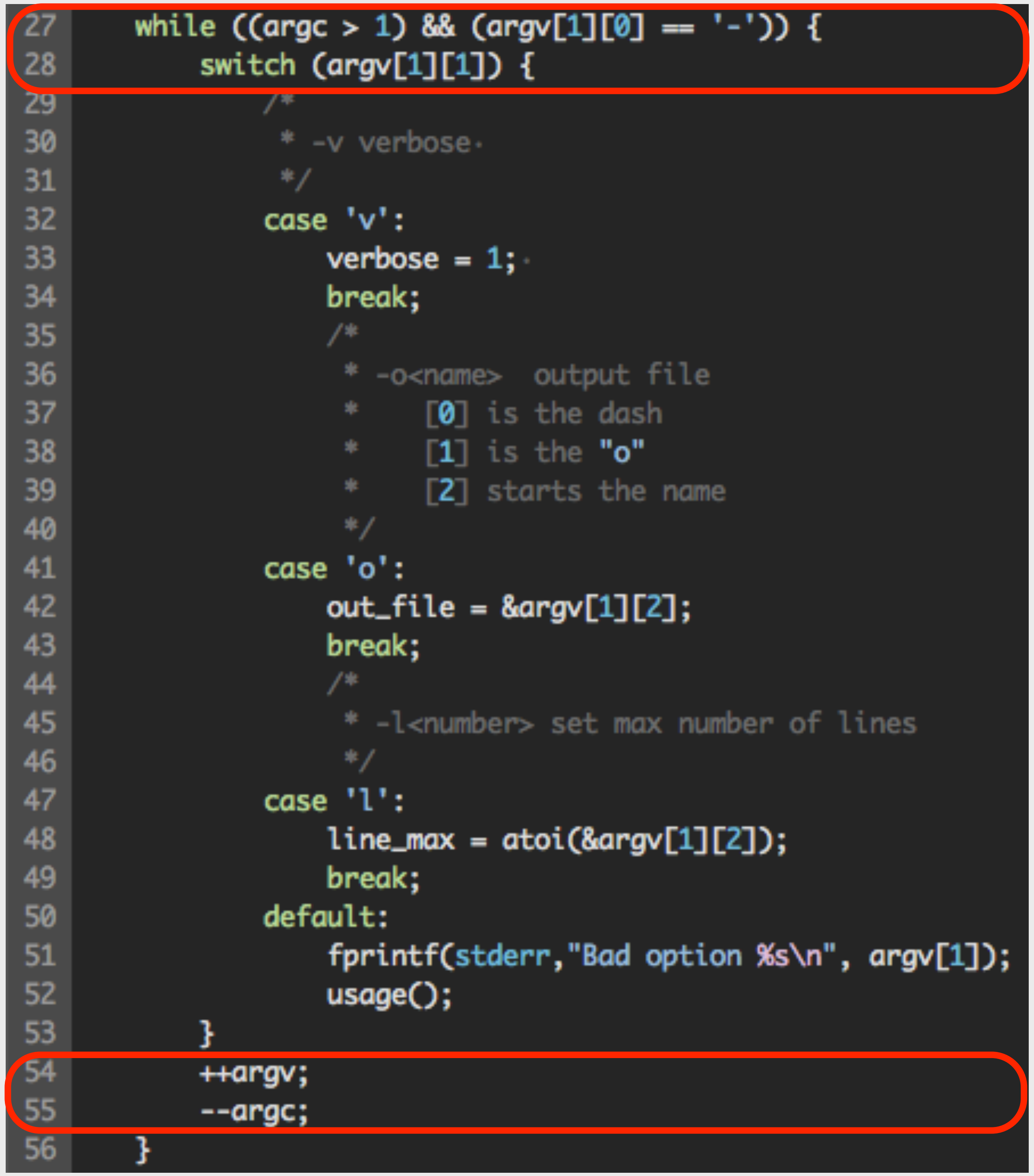
```
    while ((argc > 1) && (argv[1][0] == '-')) {
        switch (argv[1][1]) {
            /*
             * -v verbose
             */
            case 'v':
                verbose = 1;
                break;
                /*
                 * -o<name>  output file
                 *     [0] is the dash
                 *     [1] is the "o"
                 *     [2] starts the name
                 */
            case 'o':
                out_file = &argv[1][2];
                break;
                /*
                 * -l<number> set max number of lines
                 */
            case 'l':
                line_max = atoi(&argv[1][2]);
                break;
            default:
                fprintf(stderr,"Bad option %s\n", argv[1]);
                usage();
        }
        ++argv;
        --argc;
    }
```



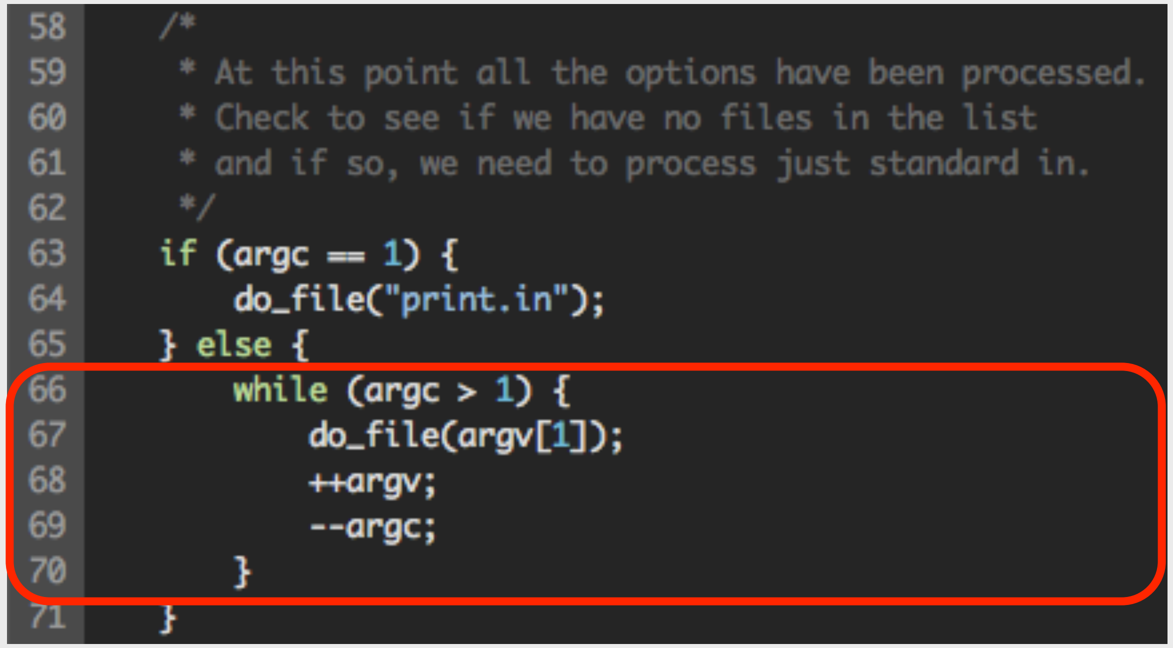
```
    /*
     * At this point all the options have been processed.
     * Check to see if we have no files in the list
     * and if so, we need to process just standard in.
     */
    if (argc == 1) {
        do_file("print.in");
    } else {
        while (argc > 1) {
            do_file(argv[1]);
            ++argv;
            --argc;
        }
    }
```

Command-Line Arguments

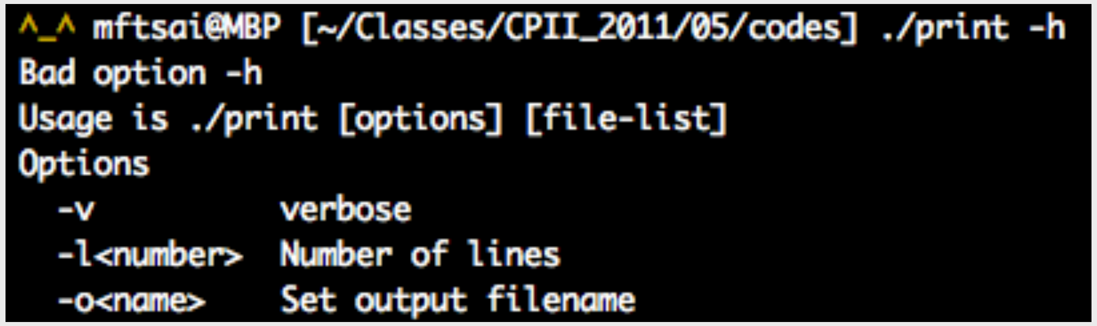
- Example: print.c



```c
    while ((argc > 1) && (argv[1][0] == '-')) {
        switch (argv[1][1]) {
            /*
             * -v verbose
             */
            case 'v':
                verbose = 1;
                break;
                /*
                 * -o<name>  output file
                 *     [0] is the dash
                 *     [1] is the "o"
                 *     [2] starts the name
                 */
            case 'o':
                out_file = &argv[1][2];
                break;
                /*
                 * -l<number> set max number of lines
                 */
            case 'l':
                line_max = atoi(&argv[1][2]);
                break;
            default:
                fprintf(stderr,"Bad option %s\n", argv[1]);
                usage();
        }
        ++argv;
        --argc;
    }
```



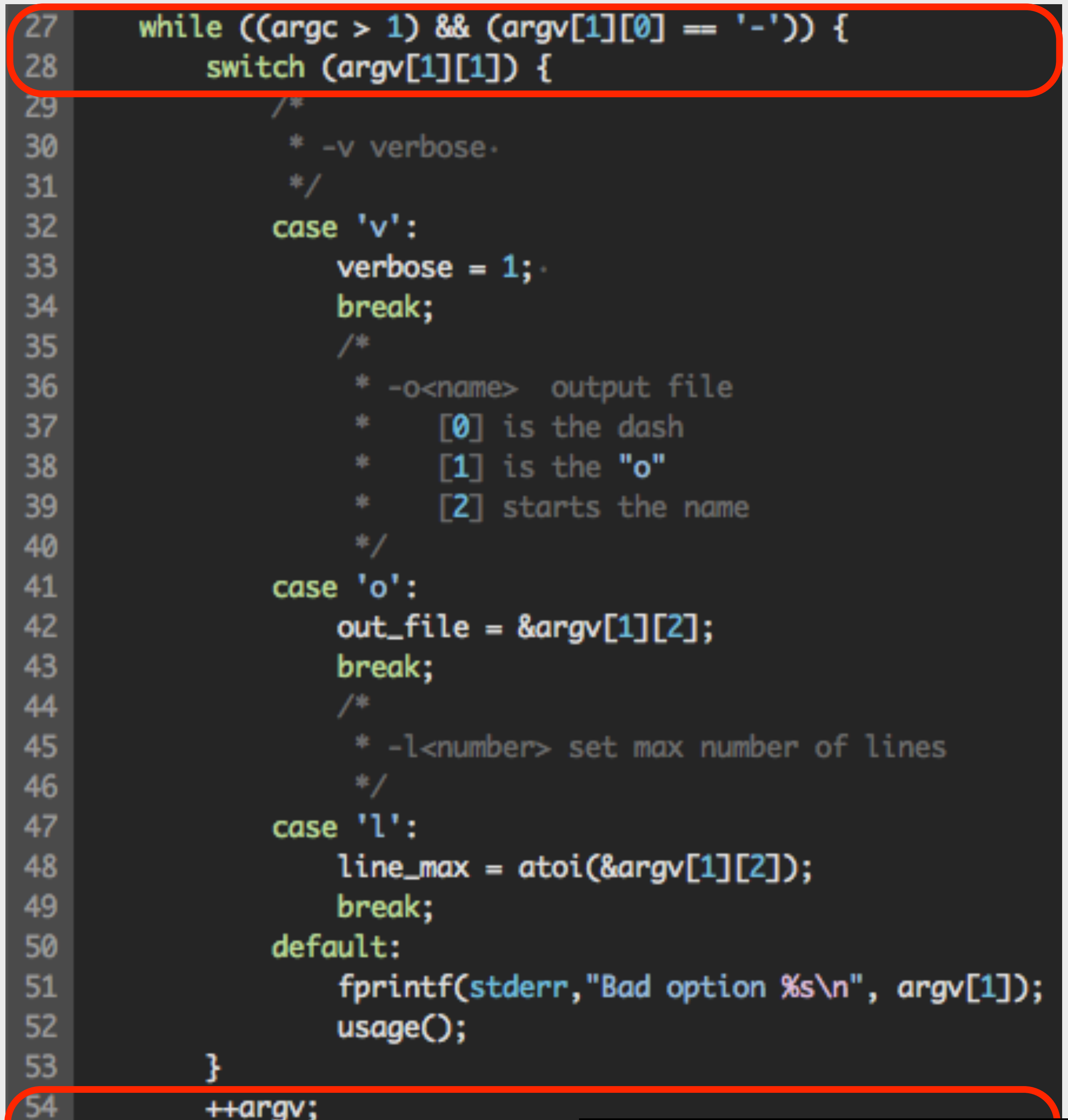
```c
    /*
     * At this point all the options have been processed.
     * Check to see if we have no files in the list
     * and if so, we need to process just standard in.
     */
    if (argc == 1) {
        do_file("print.in");
    } else {
        while (argc > 1) {
            do_file(argv[1]);
            ++argv;
            --argc;
        }
    }
```



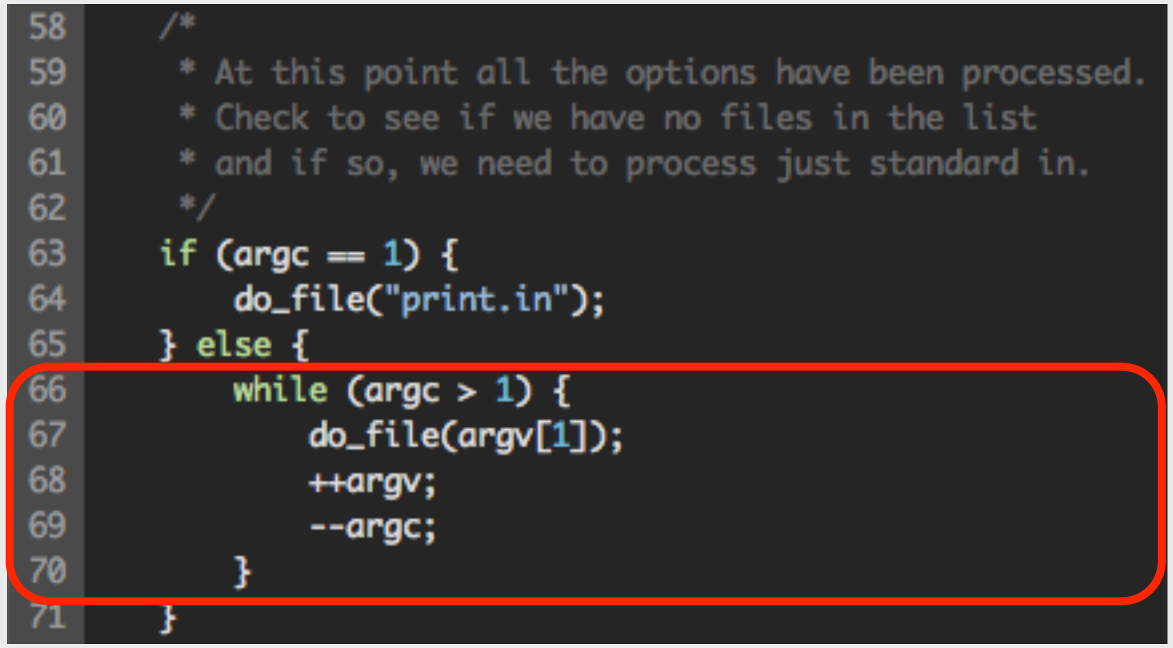
```
^_^ mftsai@MBP [~/Classes/CPII_2011/05/codes] ./print -h
Bad option -h
Usage is ./print [options] [file-list]
Options
  -v          verbose
  -l<number>  Number of lines
  -o<name>    Set output filename
```

Command-Line Arguments

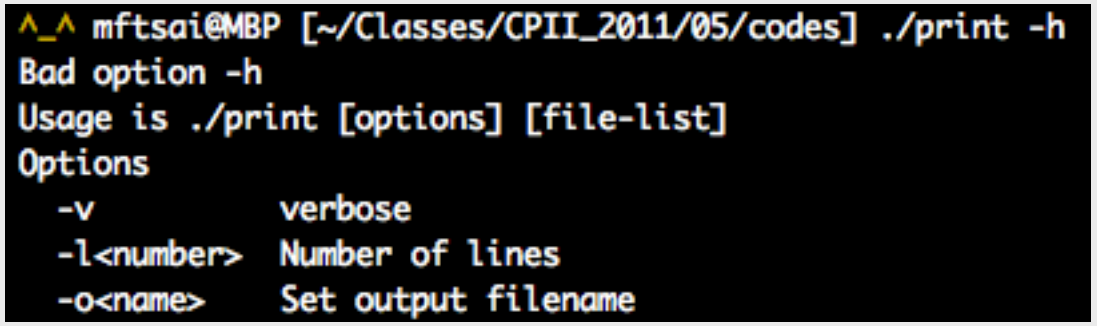
- Example: print.c



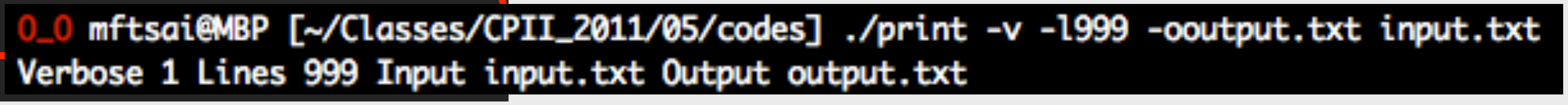
```c
    while ((argc > 1) && (argv[1][0] == '-')) {
        switch (argv[1][1]) {
            /*
             * -v verbose
             */
            case 'v':
                verbose = 1;
                break;
                /*
                 * -o<name>  output file
                 *     [0] is the dash
                 *     [1] is the "o"
                 *     [2] starts the name
                 */
            case 'o':
                out_file = &argv[1][2];
                break;
                /*
                 * -l<number> set max number of lines
                 */
            case 'l':
                line_max = atoi(&argv[1][2]);
                break;
            default:
                fprintf(stderr,"Bad option %s\n", argv[1]);
                usage();
        }
        ++argv;
        --argc;
    }
```



```c
    /*
     * At this point all the options have been processed.
     * Check to see if we have no files in the list
     * and if so, we need to process just standard in.
     */
    if (argc == 1) {
        do_file("print.in");
    } else {
        while (argc > 1) {
            do_file(argv[1]);
            ++argv;
            --argc;
        }
    }
```



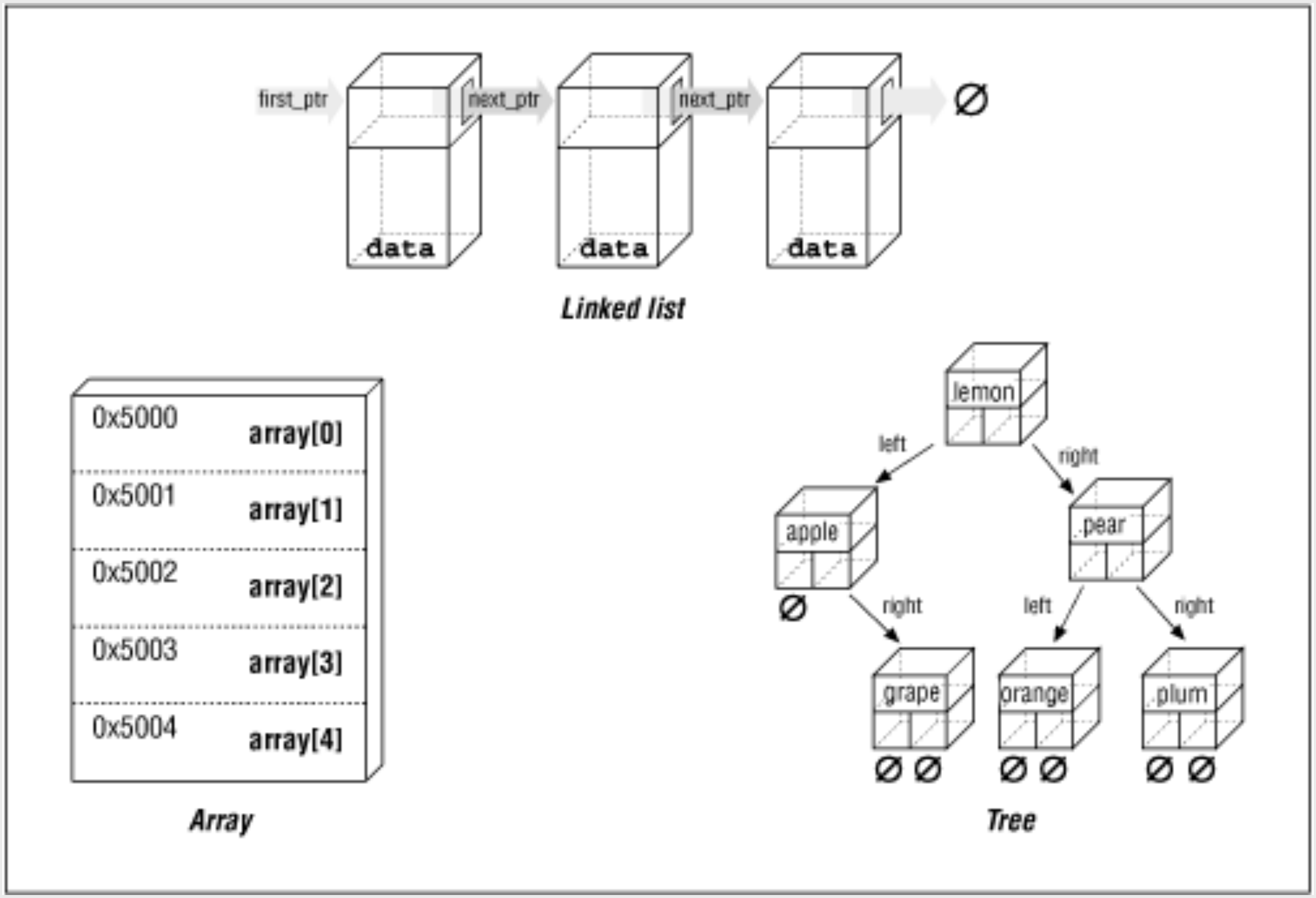
```
^_^ mftsai@MBP [~/Classes/CPII_2011/05/codes] ./print -h
Bad option -h
Usage is ./print [options] [file-list]
Options
  -v          verbose
  -l<number>  Number of lines
  -o<name>    Set output filename
```



```
0_0 mftsai@MBP [~/Classes/CPII_2011/05/codes] ./print -v -l999 -ooutput.txt input.txt
Verbose 1 Lines 999 Input input.txt Output output.txt
```

Advanced Pointers

Dynamic Data Structures

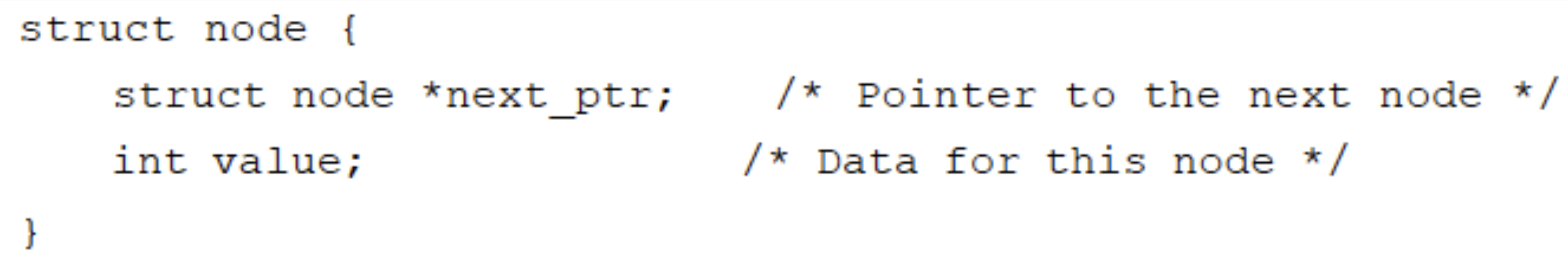


Pointers and Structures

- Structures can contain pointers

Pointers and Structures

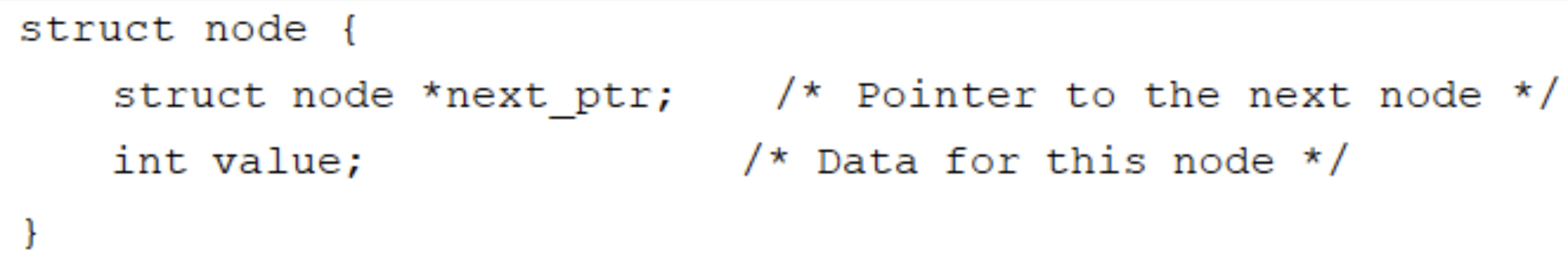
- Structures can contain pointers



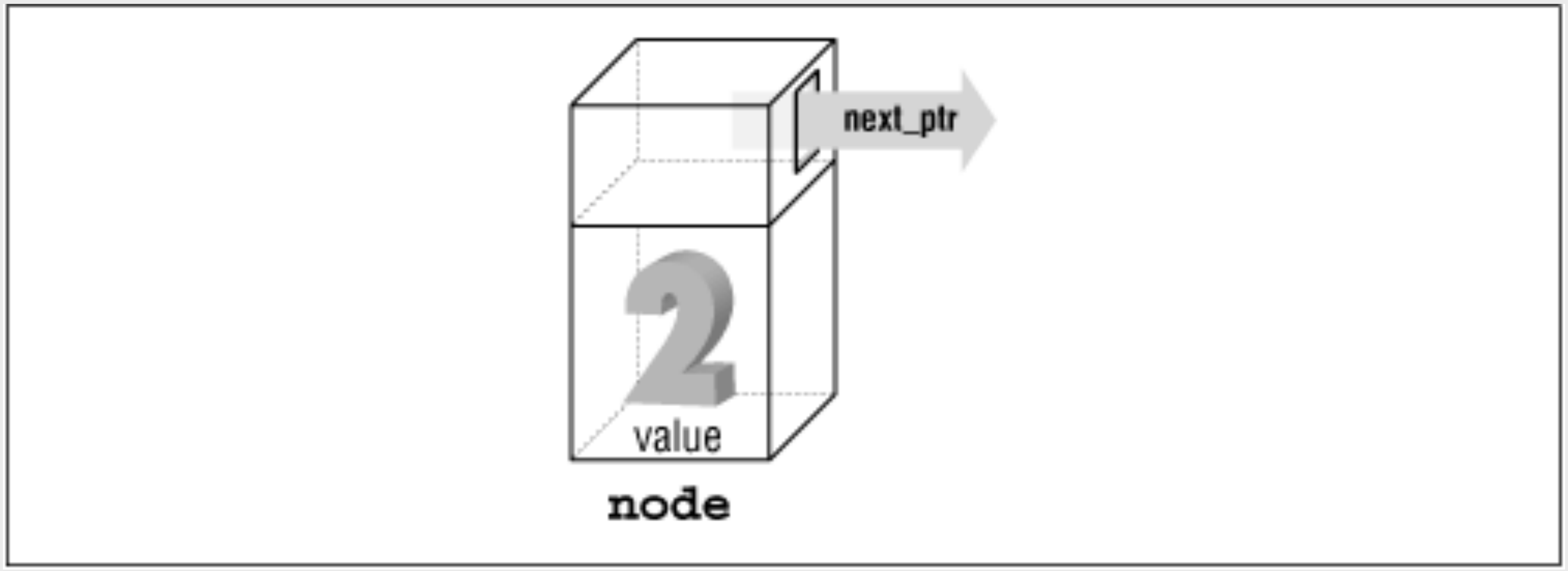
```
struct node {
    struct node *next_ptr;    /* Pointer to the next node */
    int value;               /* Data for this node */
}
```

Pointers and Structures

- Structures can contain pointers



```
struct node {
    struct node *next_ptr;    /* Pointer to the next node */
    int value;               /* Data for this node */
}
```



Pointers and Structures

- How to create the structure node

Pointers and Structures

- How to create the structure node

1. Declare nodes explicitly

```
struct node *node_1;
struct node *node_2;
```

But, only for a limited nodes!!

Pointers and Structures

- How to create the structure node

1. Declare nodes explicitly

```
struct node *node_1;
struct node *node_2;
```

But, only for a limited nodes!!

2. use **`malloc`** for allocation

Dynamic allocation!!
More flexible!!

Pointers and Structures

Pointers and Structures

- **`malloc`**

Pointers and Structures

- **`malloc`**
  - allocates storage for a variable and then returns a pointer

Pointers and Structures

- **`malloc`**
  - allocates storage for a variable and then returns a pointer
  - create a new, **unnamed** variable and returns a pointer to it

Pointers and Structures

- `malloc`
  - allocates storage for a variable and then returns a pointer
  - create a new, **unnamed** variable and returns a pointer to it
  - The “things” created by `malloc` can be referenced only through pointers, never by name

Pointers and Structures

Pointers and Structures

- Definition of malloc

```
void *malloc(unsigned int);
```

Pointers and Structures

- Definition of malloc

```
void *malloc(unsigned int);
```

- a single argument: the number of bytes to allocate

Pointers and Structures

- Definition of malloc

  ```
  void *malloc(unsigned int);
  ```

- a single argument: the number of bytes to allocate
- `void *` is used to indicate that malloc returns a generic pointer.

Pointers and Structures

- Definition of malloc

```
void *malloc(unsigned int);
```

- a single argument: the number of bytes to allocate
- `void *` is used to indicate that malloc returns a generic pointer.
- C uses **`void`** for two purposes:

Pointers and Structures

- Definition of malloc

  ```
  void *malloc(unsigned int);
  ```

- a single argument: the number of bytes to allocate
- **`void *`** is used to indicate that malloc returns a generic pointer.
- C uses **`void`** for two purposes:
  - When used as a type in a function declaration, **`void`** indicates the function returns no value

Pointers and Structures

- Definition of malloc

  ```
  void *malloc(unsigned int);
  ```

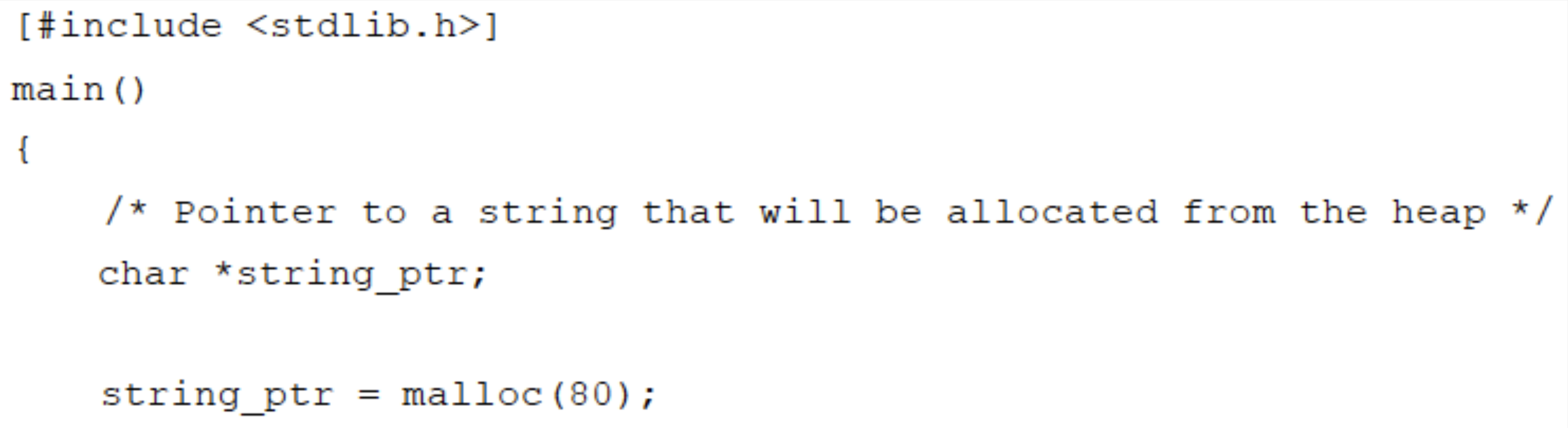
- a single argument: the number of bytes to allocate
- **`void *`** is used to indicate that malloc returns a generic pointer.
- C uses **`void`** for two purposes:
  - When used as a type in a function declaration, **`void`** indicates the function returns no value
  - When used in a pointer declaration, **`void`** defines a generic pointer

Pointers and Structures

- Allocating Memory for a String

Pointers and Structures

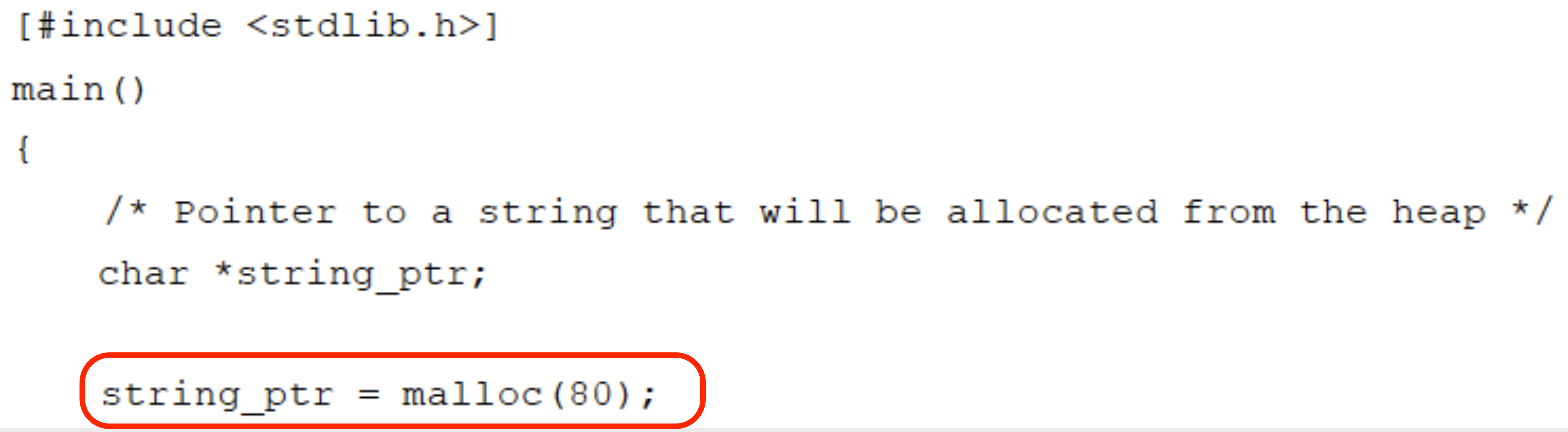
- Allocating Memory for a String



```
[#include <stdlib.h>]
main()
{
    /* Pointer to a string that will be allocated from the heap */
    char *string_ptr;

    string_ptr = malloc(80);
```

Pointers and Structures

- Allocating Memory for a String



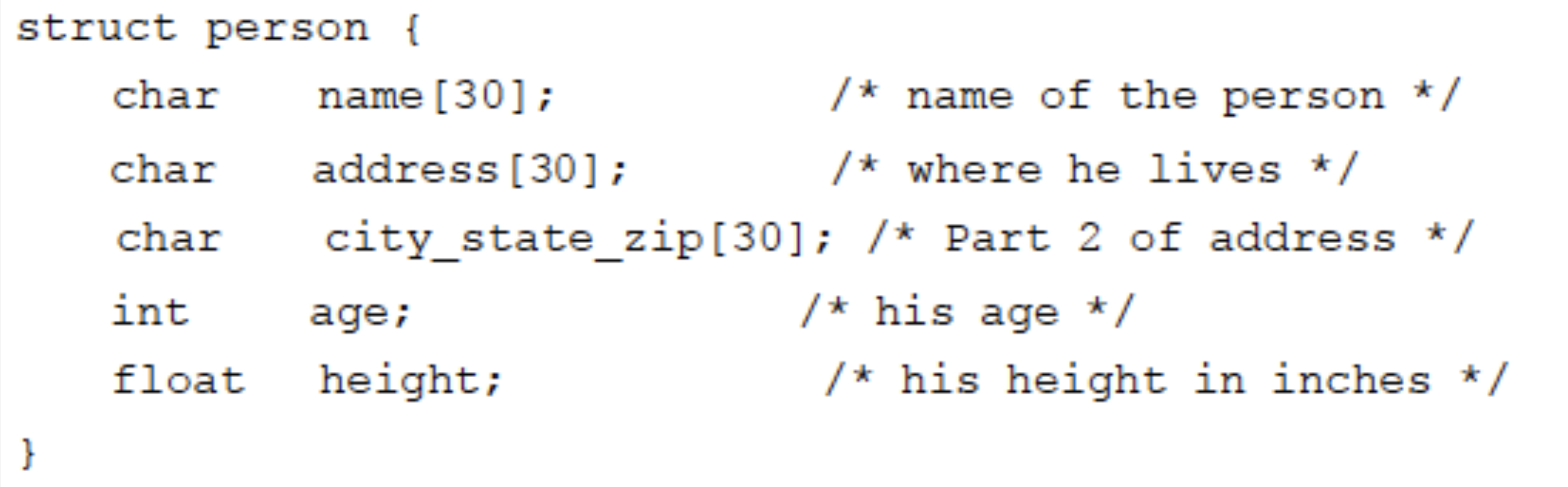
```
[#include <stdlib.h>]
main()
{
    /* Pointer to a string that will be allocated from the heap */
    char *string_ptr;

    string_ptr = malloc(80);
```

Pointers and Structures

- Suppose we are working on a complex database that contains a mailing list

Pointers and Structures

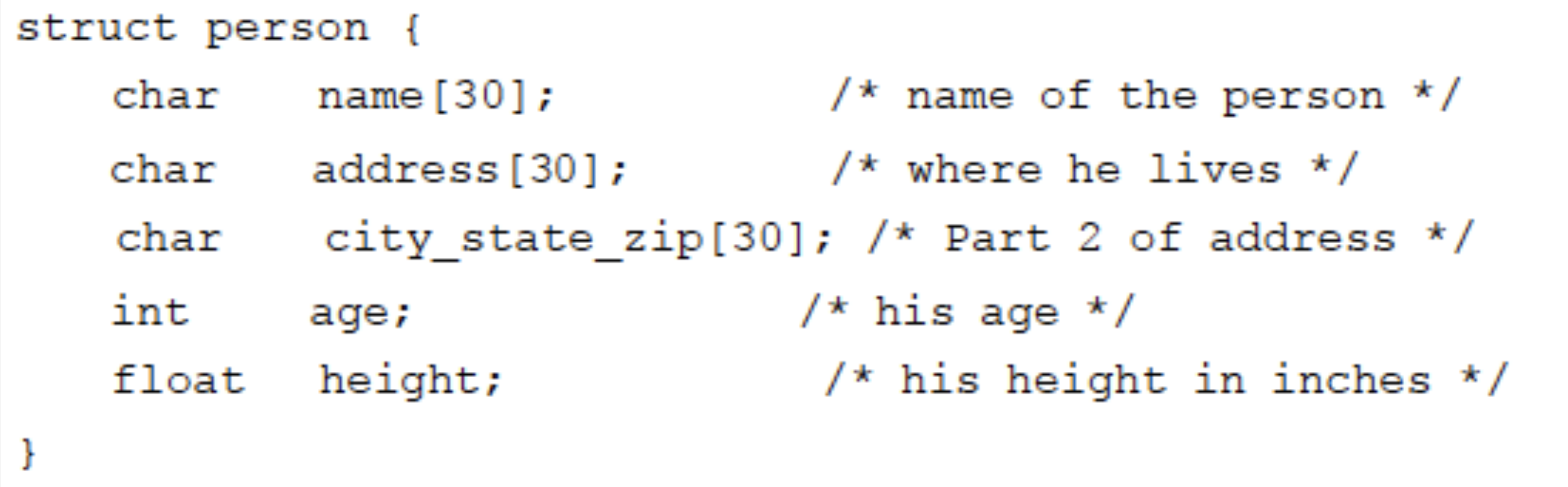
- Suppose we are working on a complex database that contains a mailing list



```
struct person {
    char    name[30];          /* name of the person */
    char    address[30];       /* where he lives */
    char    city_state_zip[30]; /* Part 2 of address */
    int     age;              /* his age */
    float   height;            /* his height in inches */
}
```

Pointers and Structures

- Suppose we are working on a complex database that contains a mailing list

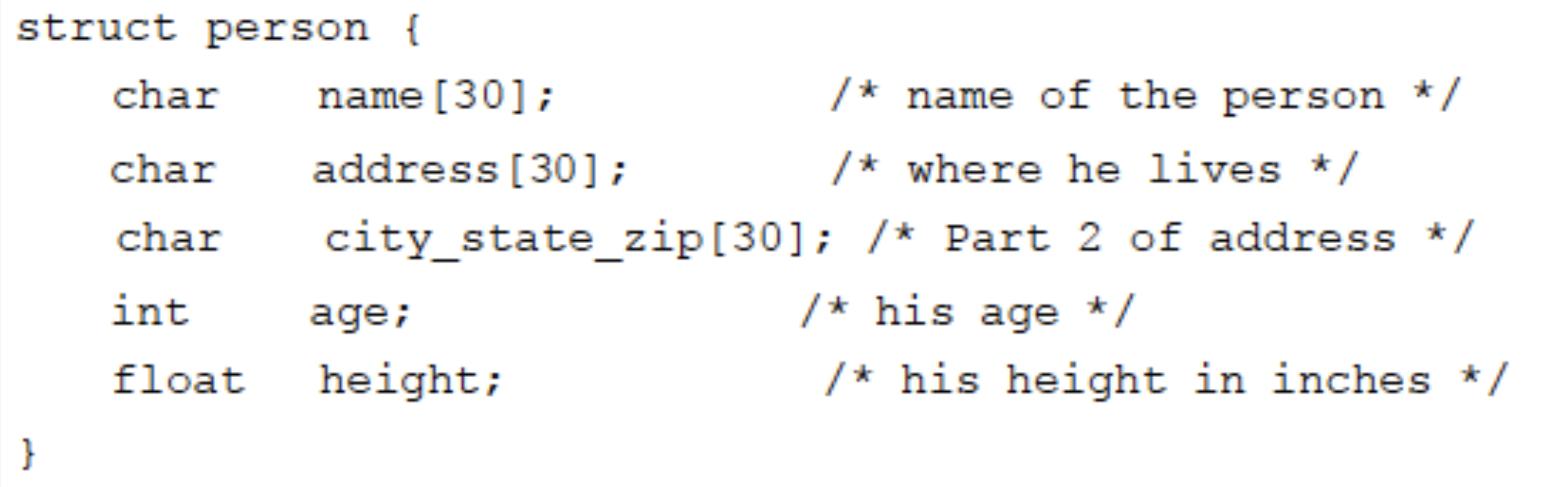


```
struct person {
    char    name[30];           /* name of the person */
    char    address[30];        /* where he lives */
    char    city_state_zip[30]; /* Part 2 of address */
    int     age;                /* his age */
    float   height;             /* his height in inches */
}
```

- Use **`malloc`** to allocate space on an as-needed basis

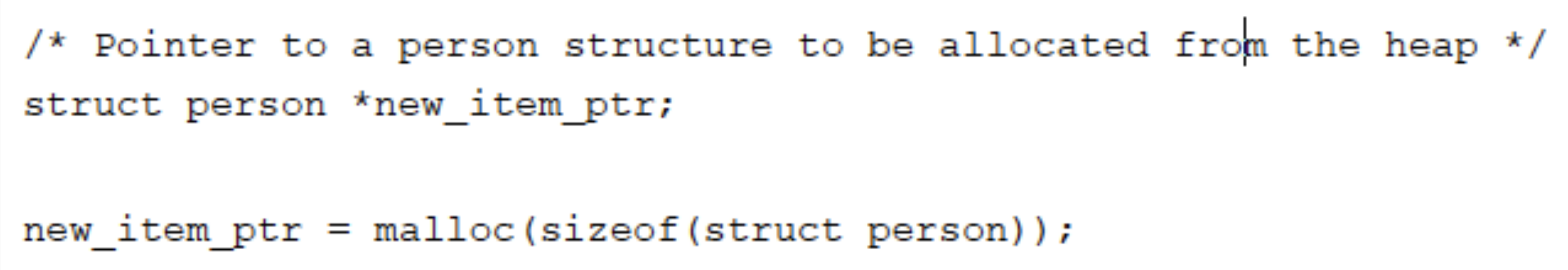
Pointers and Structures

- Suppose we are working on a complex database that contains a mailing list



```
struct person {
    char    name[30];           /* name of the person */
    char    address[30];        /* where he lives */
    char    city_state_zip[30]; /* Part 2 of address */
    int     age;                /* his age */
    float   height;             /* his height in inches */
}
```

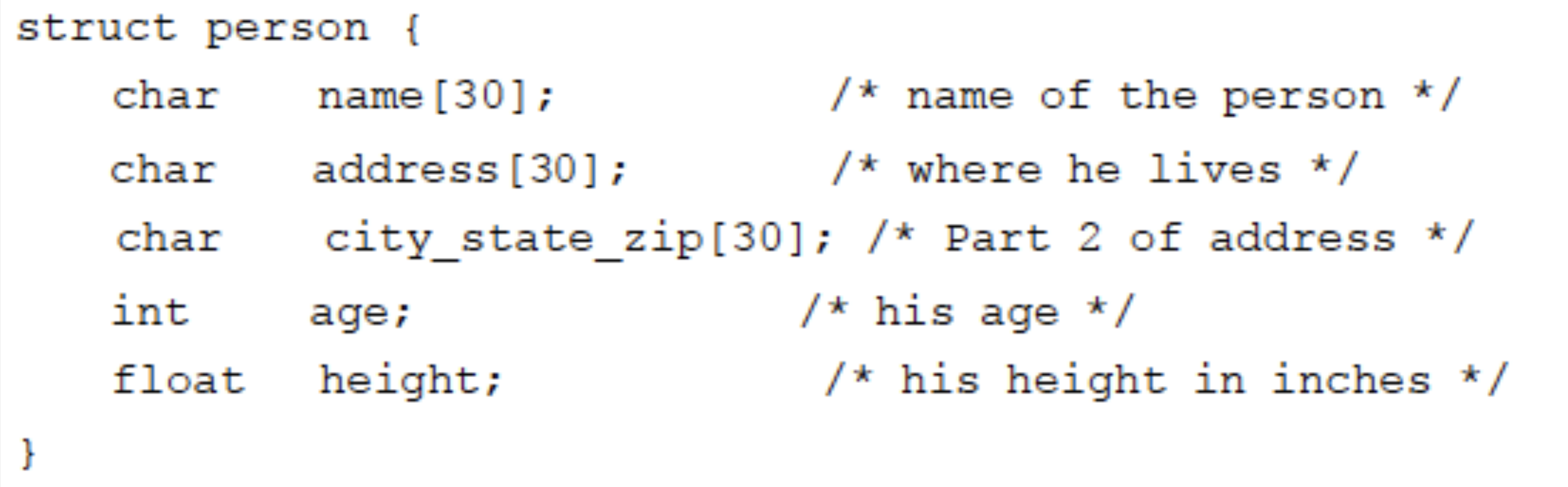
- Use **`malloc`** to allocate space on an as-needed basis



```
/* Pointer to a person structure to be allocated from the heap */
struct person *new_item_ptr;

new_item_ptr = malloc(sizeof(struct person));
```

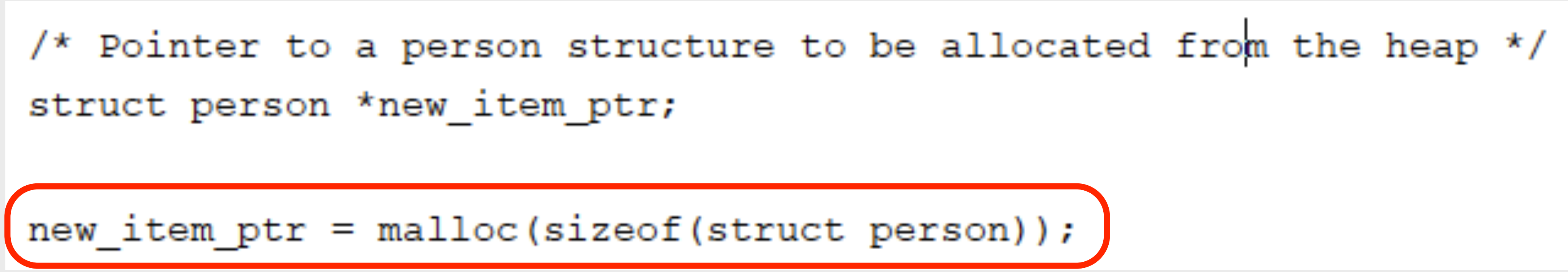
Pointers and Structures

- Suppose we are working on a complex database that contains a mailing list



```
struct person {
    char    name[30];          /* name of the person */
    char    address[30];       /* where he lives */
    char    city_state_zip[30]; /* Part 2 of address */
    int     age;              /* his age */
    float   height;            /* his height in inches */
}
```

- Use **`malloc`** to allocate space on an as-needed basis



```
/* Pointer to a person structure to be allocated from the heap */
struct person *new_item_ptr;

new_item_ptr = malloc(sizeof(struct person));
```

Pointers and Structures

Pointers and Structures

- Access the data in the structure

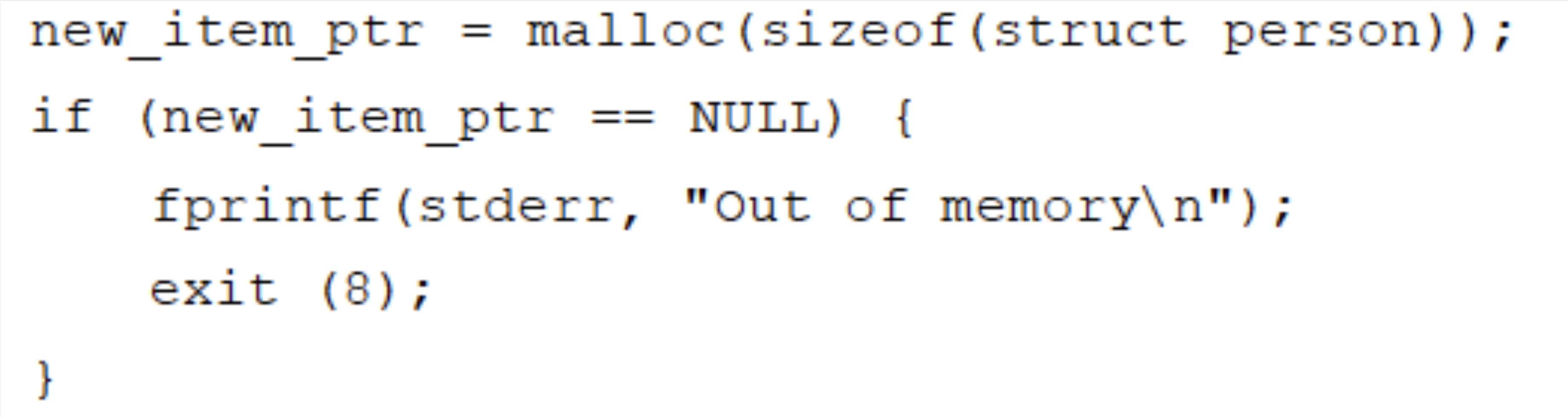
```
new_item_ptr -> name
new_item_ptr -> address
new_item_ptr -> city_address_zip
new_item_ptr -> age
new_item_ptr -> height
```

Pointers and Structures

- Check the return value of each **`malloc`** call to ensure that you really got the memory

Pointers and Structures

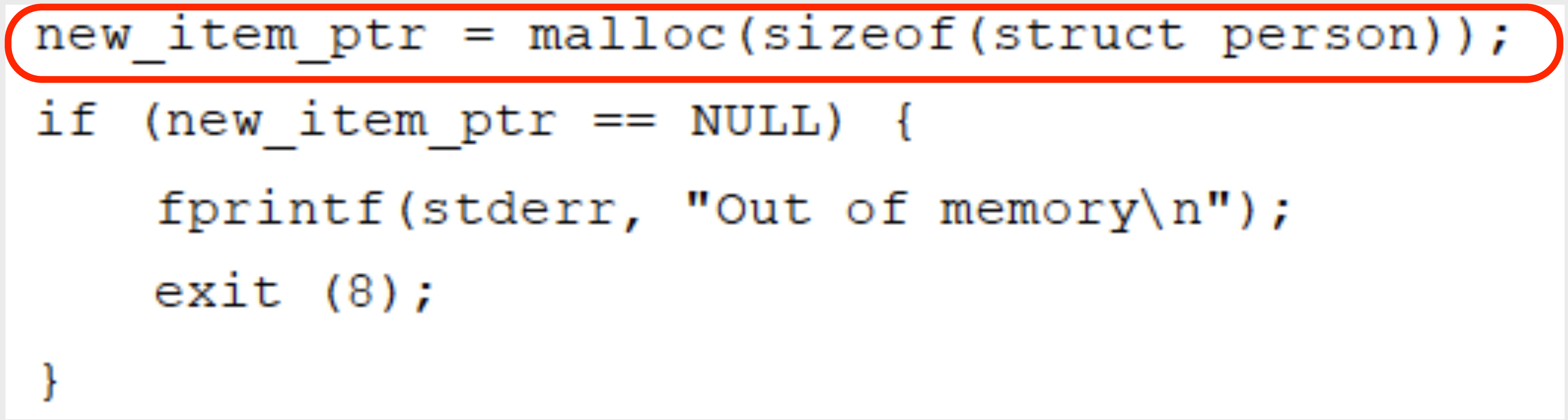
- Check the return value of each **`malloc`** call to ensure that you really got the memory



```
new_item_ptr = malloc(sizeof(struct person));
if (new_item_ptr == NULL) {
    fprintf(stderr, "Out of memory\n");
    exit (8);
}
```

Pointers and Structures

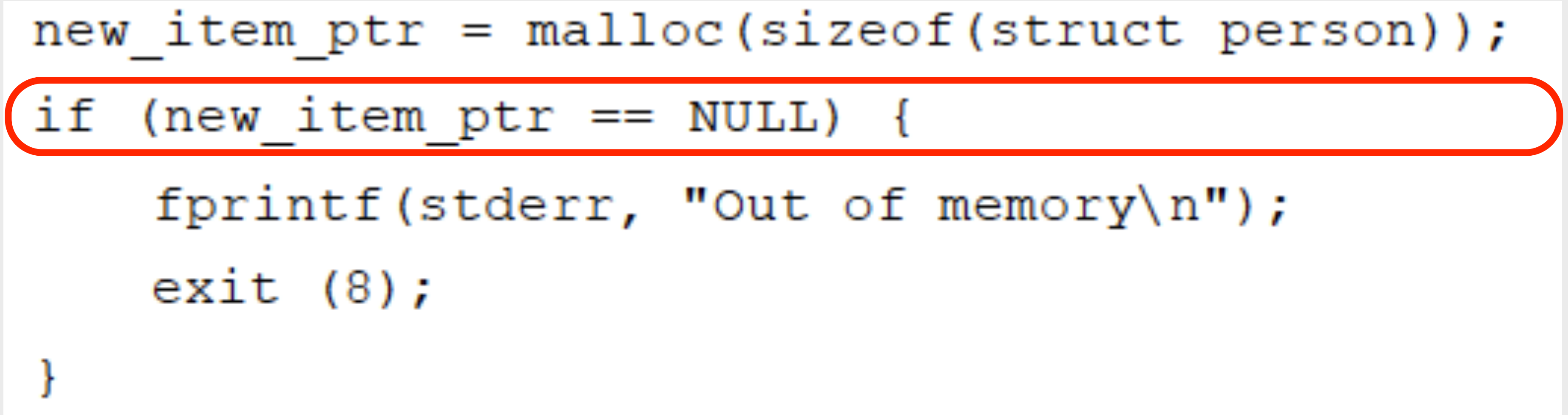
- Check the return value of each **`malloc`** call to ensure that you really got the memory



```
new_item_ptr = malloc(sizeof(struct person));
if (new_item_ptr == NULL) {
    fprintf(stderr, "Out of memory\n");
    exit (8);
}
```

Pointers and Structures

- Check the return value of each **`malloc`** call to ensure that you really got the memory



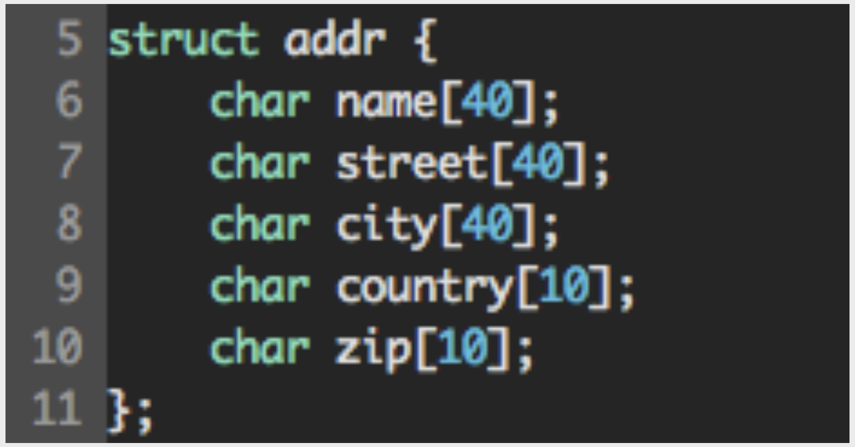
```
new_item_ptr = malloc(sizeof(struct person));
if (new_item_ptr == NULL) {
    fprintf(stderr, "Out of memory\n");
    exit (8);
}
```

Pointers and Structures

- Example: addr.c

Pointers and Structures

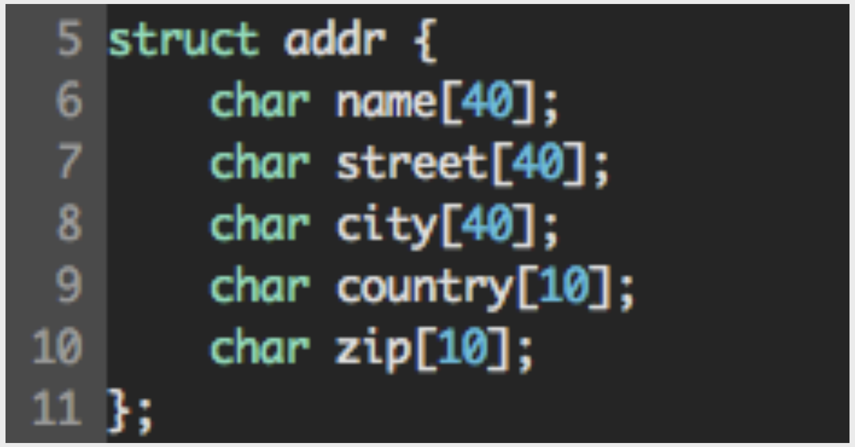
- Example: addr.c



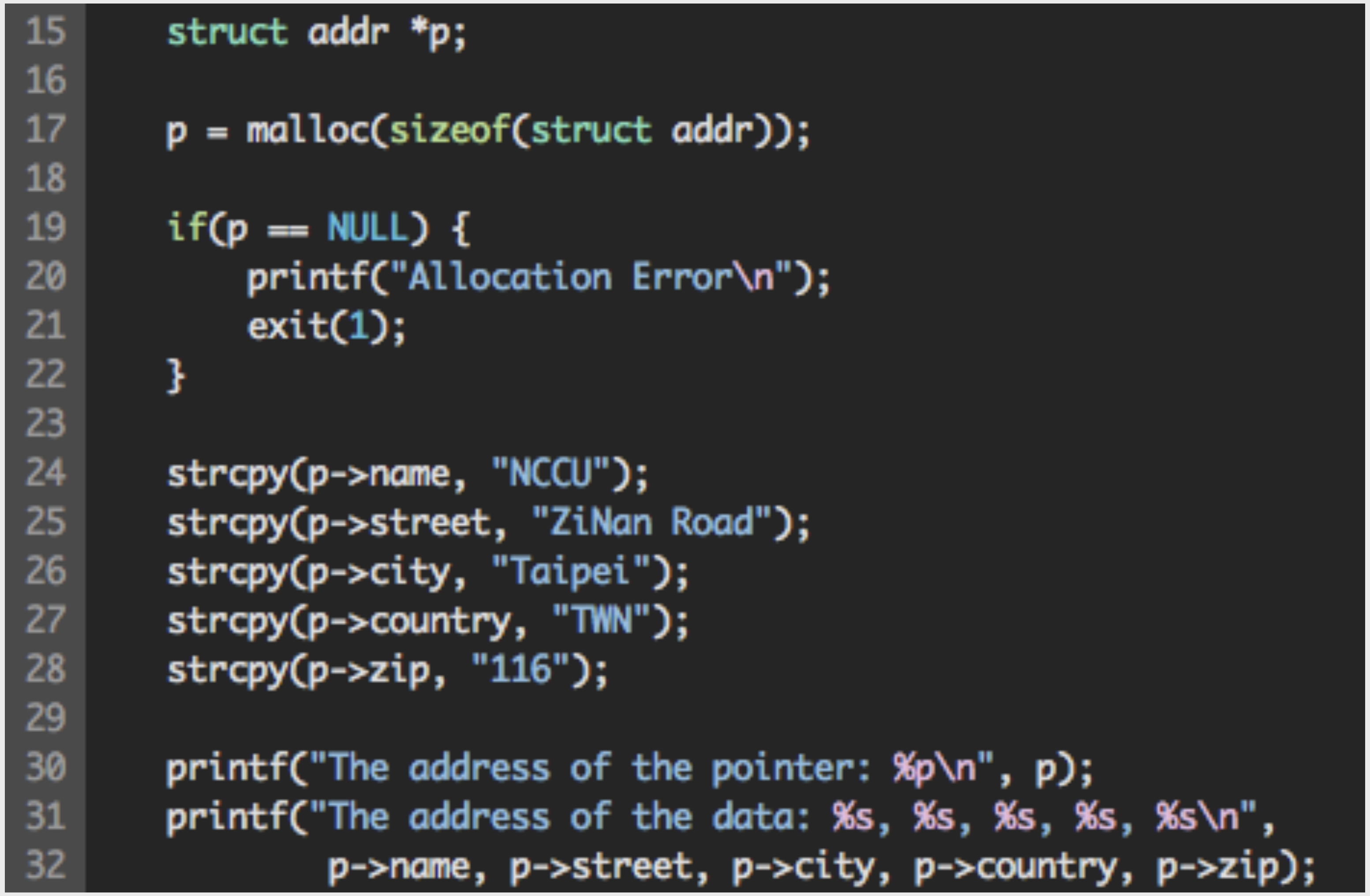
```
struct addr {
    char name[40];
    char street[40];
    char city[40];
    char country[10];
    char zip[10];
};
```

Pointers and Structures

- Example: addr.c



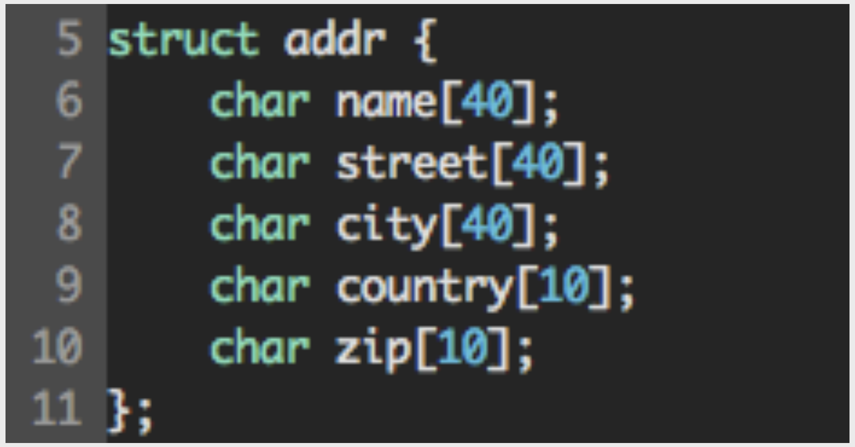
```c
struct addr {
    char name[40];
    char street[40];
    char city[40];
    char country[10];
    char zip[10];
};
```



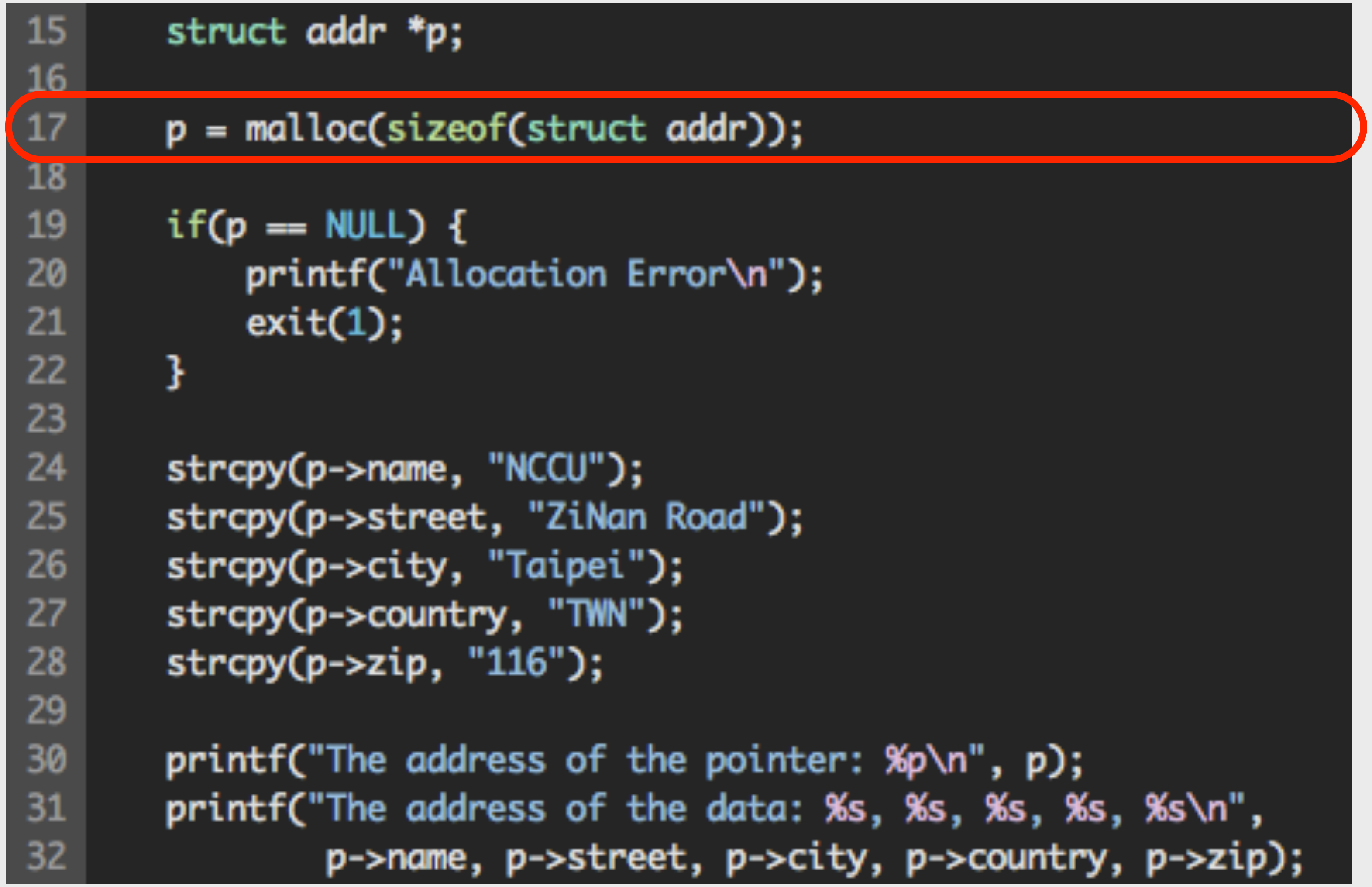
```c
    struct addr *p;

    p = malloc(sizeof(struct addr));

    if(p == NULL) {
        printf("Allocation Error\n");
        exit(1);
    }

    strcpy(p->name, "NCCU");
    strcpy(p->street, "ZiNan Road");
    strcpy(p->city, "Taipei");
    strcpy(p->country, "TWN");
    strcpy(p->zip, "116");

    printf("The address of the pointer: %p\n", p);
    printf("The address of the data: %s, %s, %s, %s, %s\n",
            p->name, p->street, p->city, p->country, p->zip);
```

Pointers and Structures

- Example: addr.c



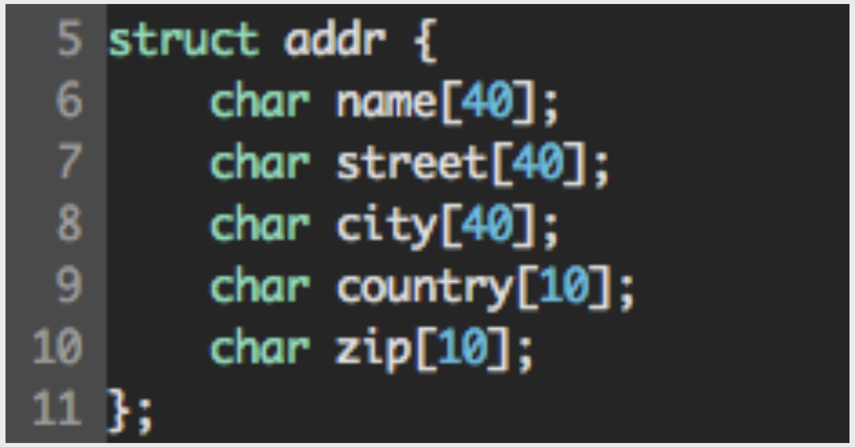
```c
struct addr {
    char name[40];
    char street[40];
    char city[40];
    char country[10];
    char zip[10];
};
```



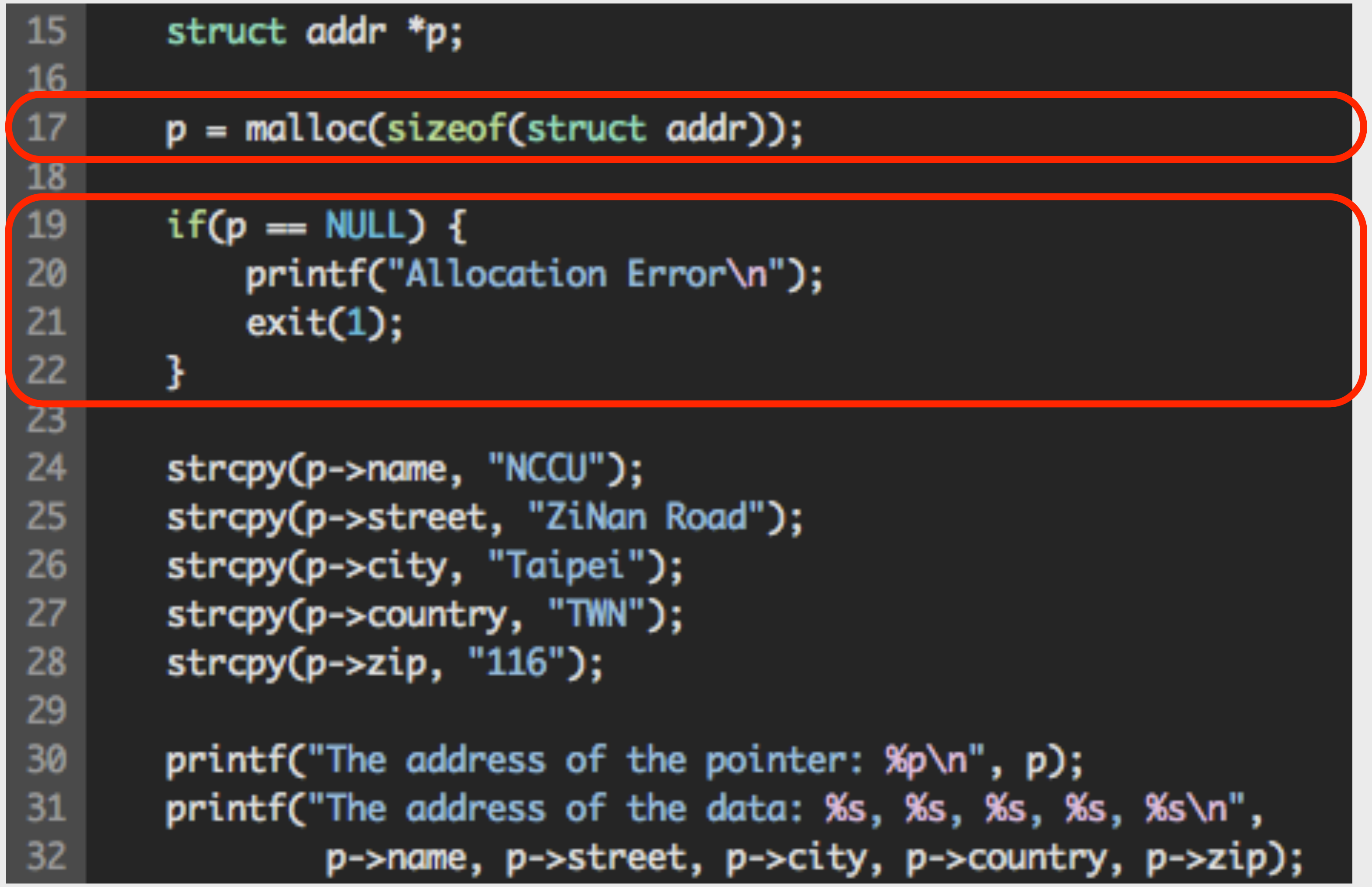
```c
    struct addr *p;

    p = malloc(sizeof(struct addr));

    if(p == NULL) {
        printf("Allocation Error\n");
        exit(1);
    }

    strcpy(p->name, "NCCU");
    strcpy(p->street, "ZiNan Road");
    strcpy(p->city, "Taipei");
    strcpy(p->country, "TWN");
    strcpy(p->zip, "116");

    printf("The address of the pointer: %p\n", p);
    printf("The address of the data: %s, %s, %s, %s, %s\n",
            p->name, p->street, p->city, p->country, p->zip);
```

Pointers and Structures

- Example: addr.c



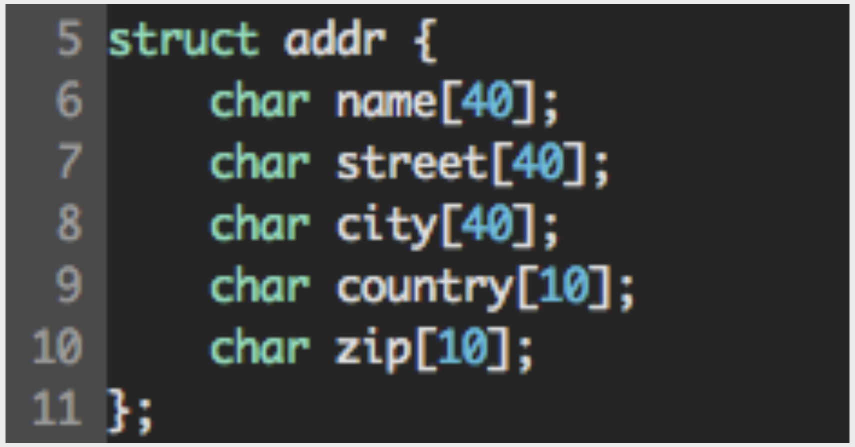
```c
struct addr {
    char name[40];
    char street[40];
    char city[40];
    char country[10];
    char zip[10];
};
```



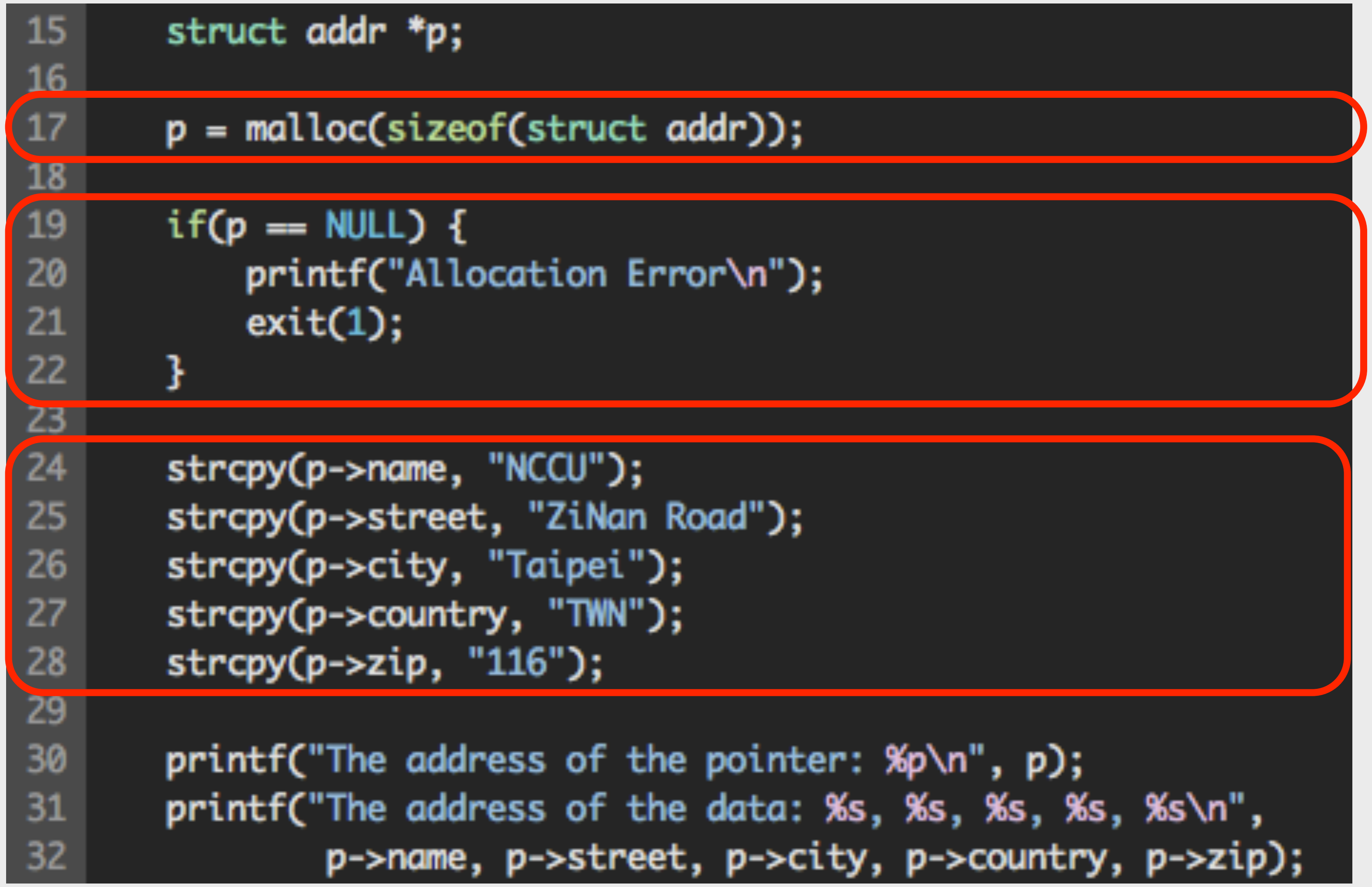
```c
    struct addr *p;

    p = malloc(sizeof(struct addr));

    if(p == NULL) {
        printf("Allocation Error\n");
        exit(1);
    }

    strcpy(p->name, "NCCU");
    strcpy(p->street, "ZiNan Road");
    strcpy(p->city, "Taipei");
    strcpy(p->country, "TWN");
    strcpy(p->zip, "116");

    printf("The address of the pointer: %p\n", p);
    printf("The address of the data: %s, %s, %s, %s, %s\n",
            p->name, p->street, p->city, p->country, p->zip);
```

Pointers and Structures

- Example: addr.c



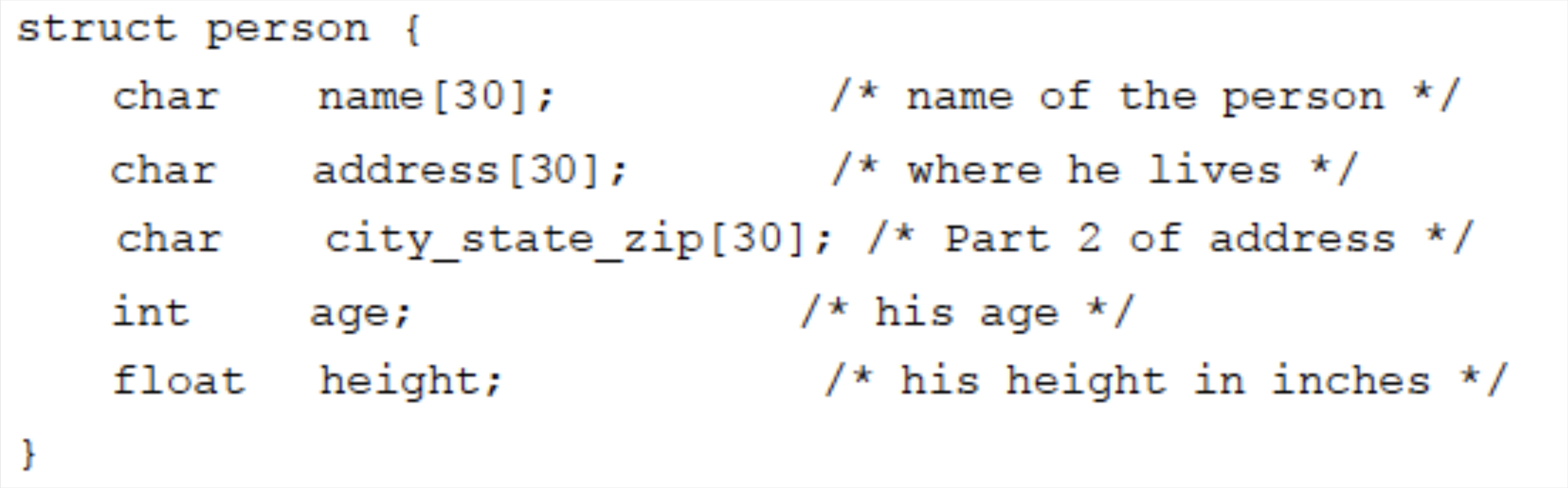
```c
struct addr {
    char name[40];
    char street[40];
    char city[40];
    char country[10];
    char zip[10];
};
```



```c
    struct addr *p;

    p = malloc(sizeof(struct addr));

    if(p == NULL) {
        printf("Allocation Error\n");
        exit(1);
    }

    strcpy(p->name, "NCCU");
    strcpy(p->street, "ZiNan Road");
    strcpy(p->city, "Taipei");
    strcpy(p->country, "TWN");
    strcpy(p->zip, "116");

    printf("The address of the pointer: %p\n", p);
    printf("The address of the data: %s, %s, %s, %s, %s\n",
            p->name, p->street, p->city, p->country, p->zip);
```

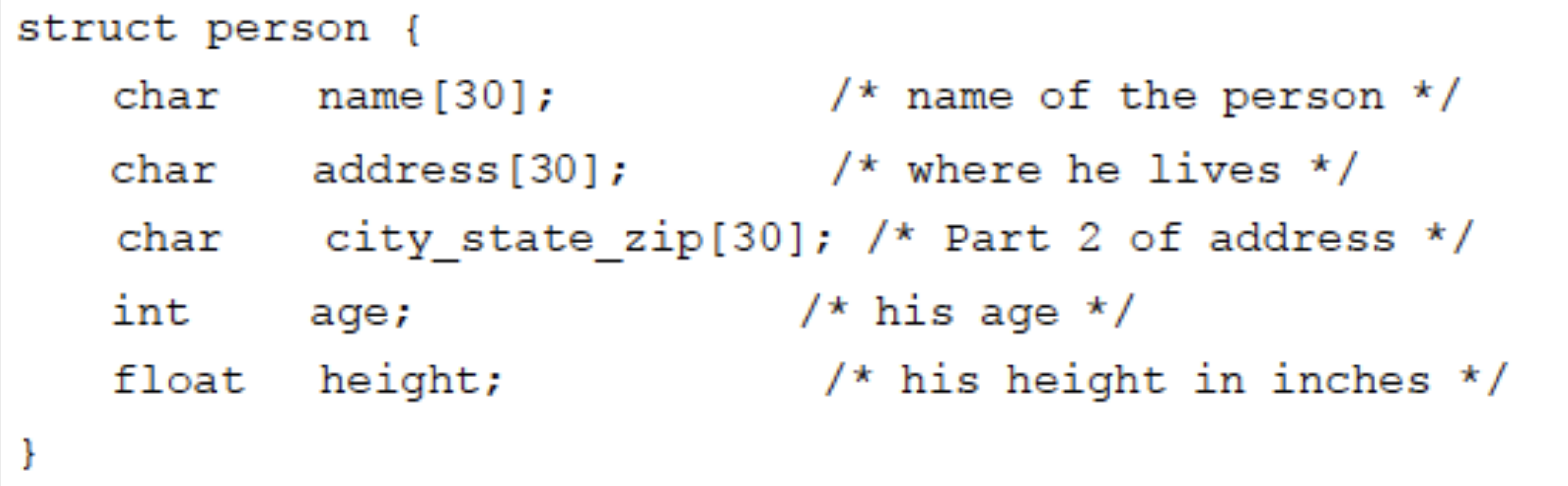
Pointers and Structures

Pointers and Structures



```
struct person {
    char    name[30];           /* name of the person */
    char    address[30];        /* where he lives */
    char    city_state_zip[30]; /* Part 2 of address */
    int     age;                /* his age */
    float   height;             /* his height in inches */
}
```

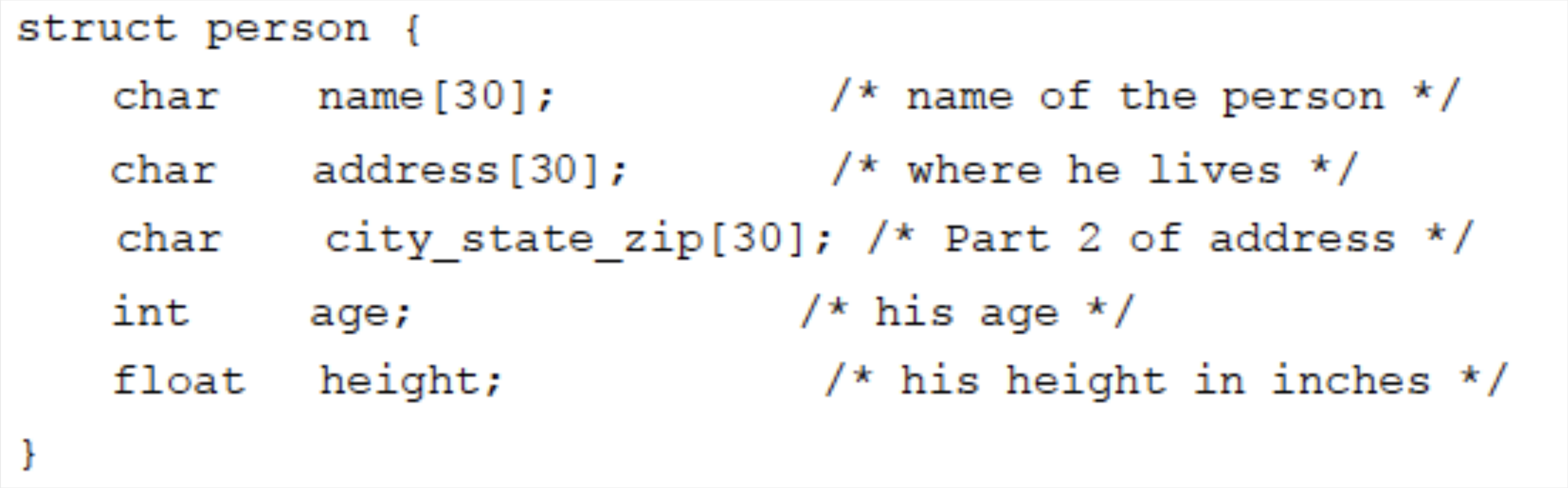
Pointers and Structures



```
struct person {
    char    name[30];           /* name of the person */
    char    address[30];        /* where he lives */
    char    city_state_zip[30]; /* Part 2 of address */
    int     age;                /* his age */
    float   height;             /* his height in inches */
}
```

- **`struct person *one_item_ptr;`**
  - **`one_item_ptr`** is a pointer (in stack) and the allocated memory is on heap

Pointers and Structures



```
struct person {
    char    name[30];           /* name of the person */
    char    address[30];        /* where he lives */
    char    city_state_zip[30]; /* Part 2 of address */
    int     age;                /* his age */
    float   height;             /* his height in inches */
}
```

- **`struct person *one_item_ptr;`**
  - **`one_item_ptr`** is a pointer (in stack) and the allocated memory is on heap
- **`struct person another_item;`**
  - **`another_item`** is a variable located in stack

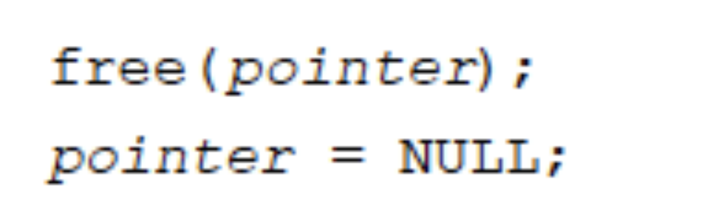
free

`free`

- `malloc` gets memory from the heap. To free the memory after are done with it, use the function `free`
- The general form of the `free` function is:

free

- **malloc** gets memory from the heap. To free the memory after are done with it, use the function **free**
- The general form of the **free** function is:

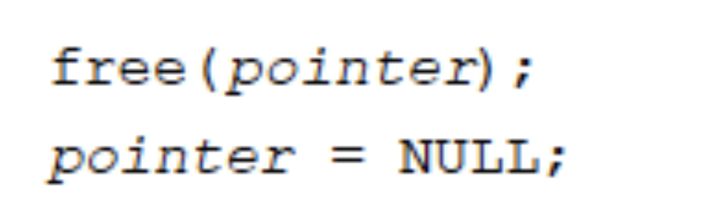


```
free(pointer);
pointer = NULL;
```

`free`

- **`malloc`** gets memory from the heap. To free the memory after are done with it, use the function **`free`**

- The general form of the **`free`** function is:



```
free(pointer);
pointer = NULL;
```

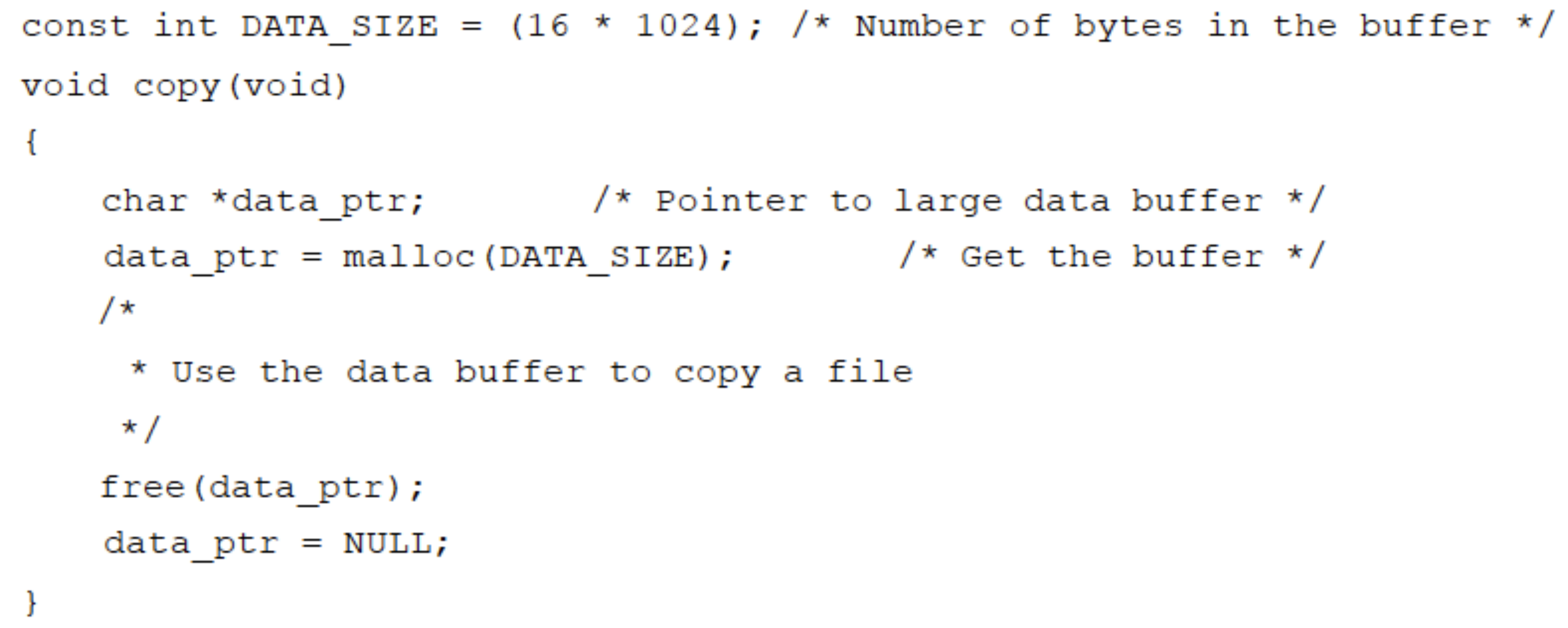
- Note: you don't have to set pointer to **`NULL`**; however, doing so prevents us from trying to use the freed memory

`free`

- Example

free

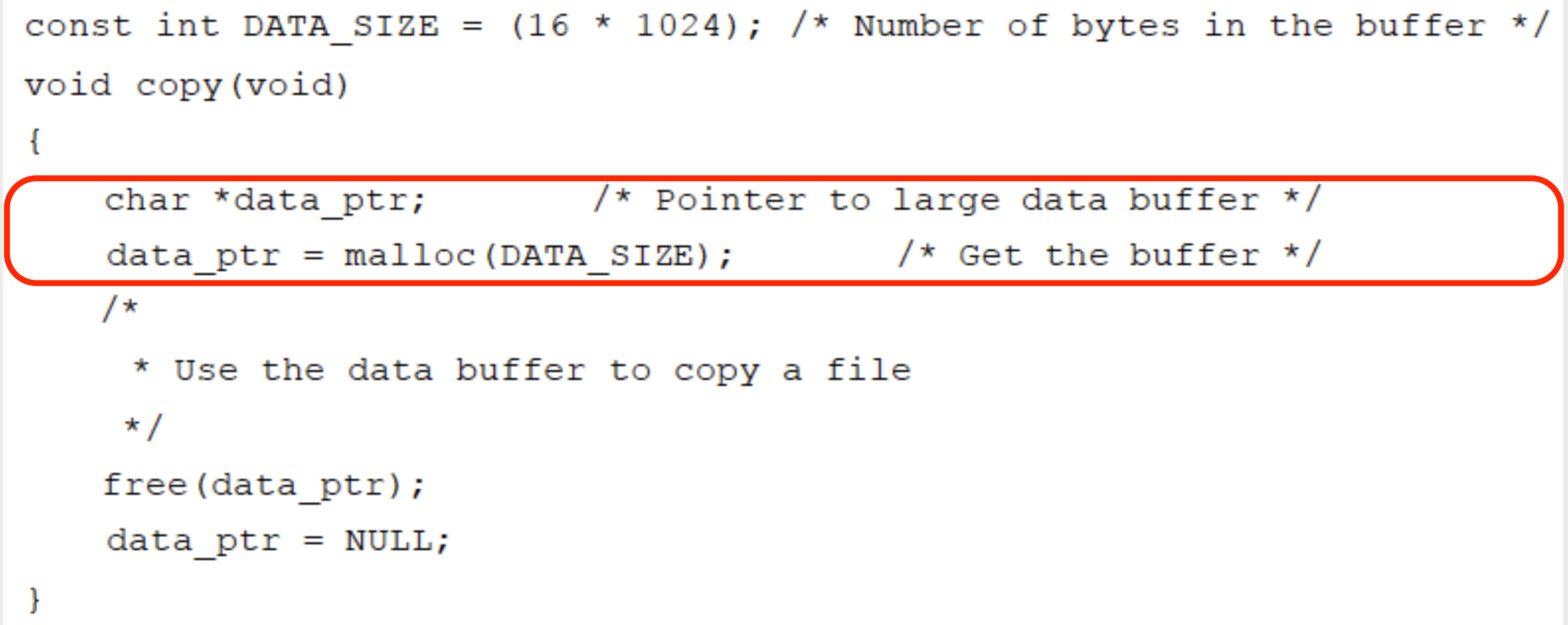
- Example



```
const int DATA_SIZE = (16 * 1024); /* Number of bytes in the buffer */
void copy(void)
{
    char *data_ptr;       /* Pointer to large data buffer */
    data_ptr = malloc(DATA_SIZE);       /* Get the buffer */
    /*
     * Use the data buffer to copy a file
     */
    free(data_ptr);
    data_ptr = NULL;
}
```

free

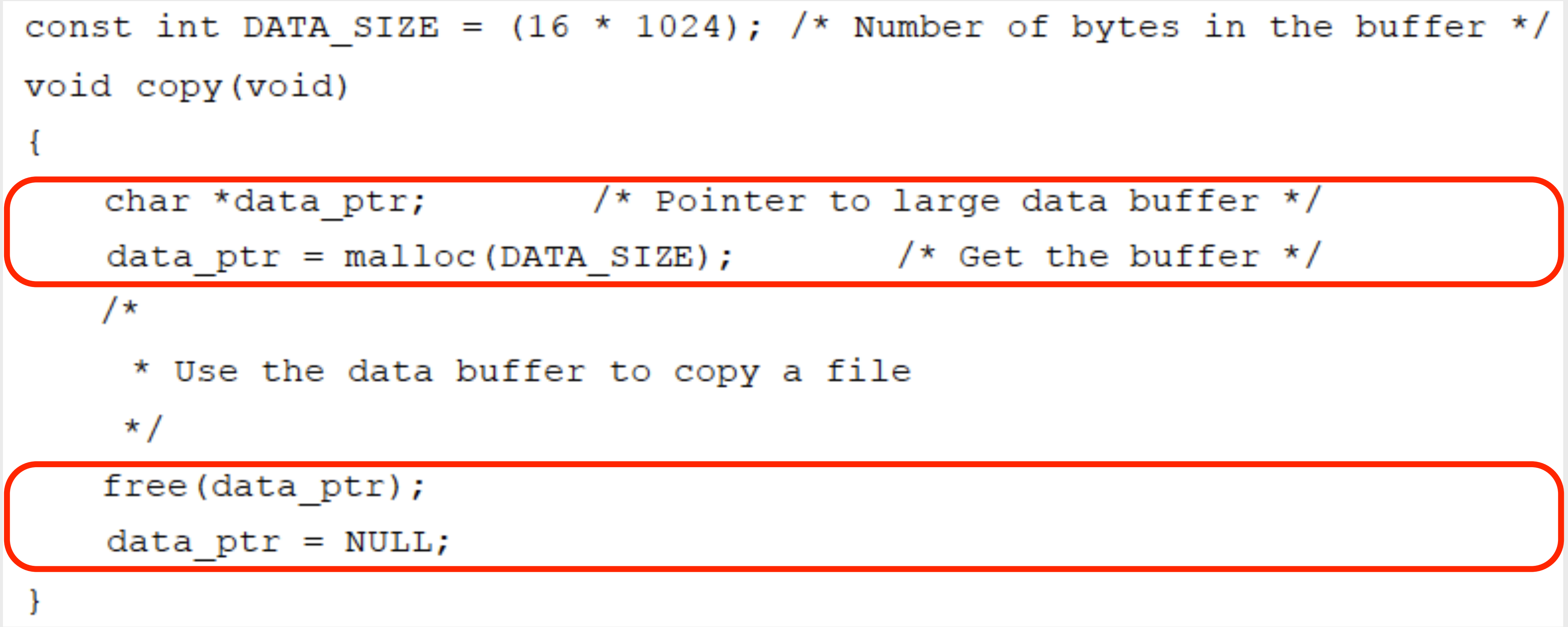
- Example



```
const int DATA_SIZE = (16 * 1024); /* Number of bytes in the buffer */
void copy(void)
{
    char *data_ptr;          /* Pointer to large data buffer */
    data_ptr = malloc(DATA_SIZE);          /* Get the buffer */
    /*
     * Use the data buffer to copy a file
     */
    free(data_ptr);
    data_ptr = NULL;
}
```

free

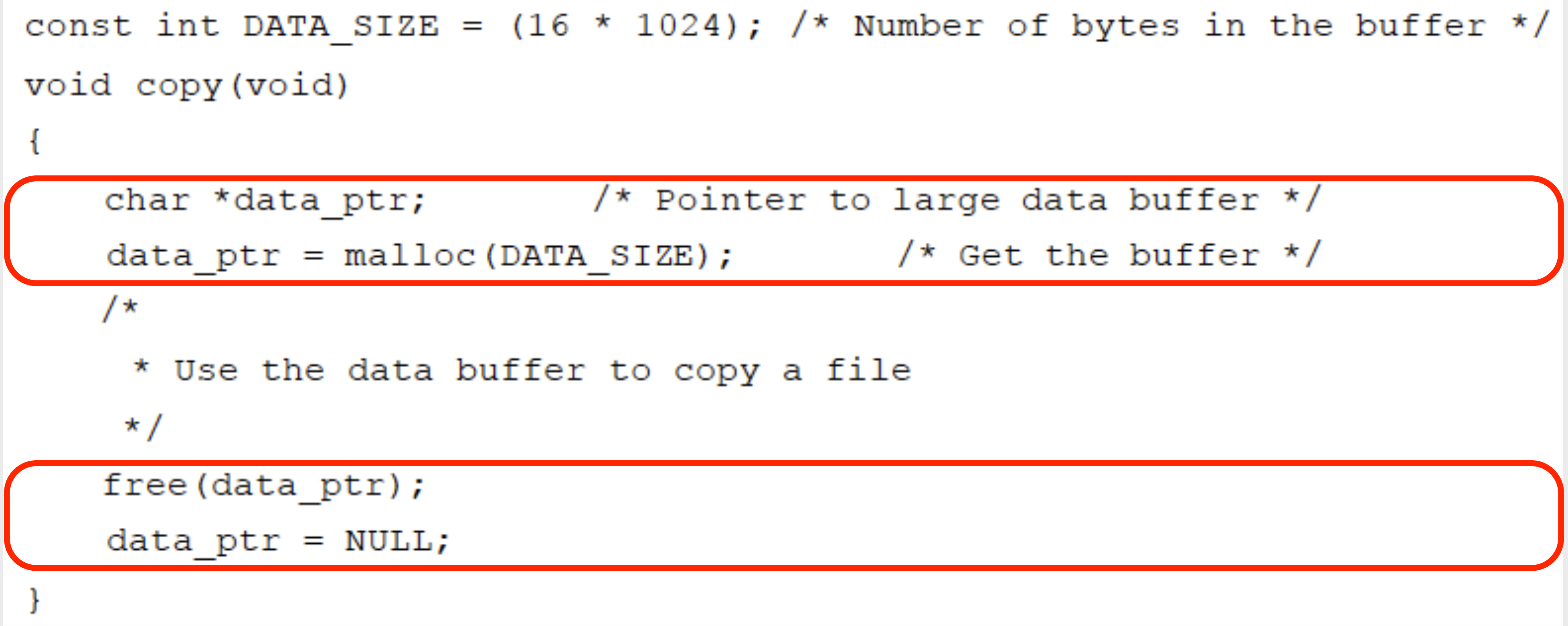
- Example



```c
const int DATA_SIZE = (16 * 1024); /* Number of bytes in the buffer */
void copy(void)
{
    char *data_ptr;          /* Pointer to large data buffer */
    data_ptr = malloc(DATA_SIZE);        /* Get the buffer */
    /*
     * Use the data buffer to copy a file
     */
    free(data_ptr);
    data_ptr = NULL;
}
```

free

- Example



```
const int DATA_SIZE = (16 * 1024); /* Number of bytes in the buffer */
void copy(void)
{
    char *data_ptr;         /* Pointer to large data buffer */
    data_ptr = malloc(DATA_SIZE);         /* Get the buffer */
    /*
     * Use the data buffer to copy a file
     */
    free(data_ptr);
    data_ptr = NULL;
}
```

- Notes:
    - without free function!! ==> memory leak
    - using a pointer after a free call!!

`malloc` an array

- How to **dynamically create** an array

`malloc` an array

- How to **dynamically create** an array

```
int *a;
a = (int*) malloc(sizeof(int) * 10);
```

`malloc` an array

- How to **dynamically create** an array

```
int *a;
a = (int*) malloc(sizeof(int) * 10);
```

```
int *a;
a = (int*) calloc(10, sizeof(int));
/* initialize to 0 */
```

`malloc` A 2D array

- General concept

`malloc` A 2D array

- General concept

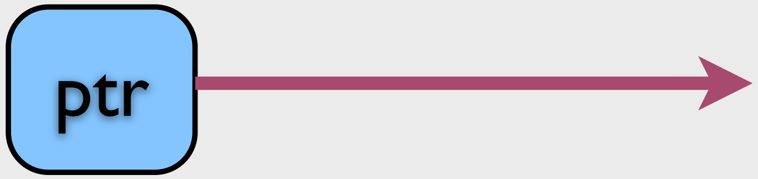
malloc A 2D array

- General concept

`int **ia`

`malloc` A 2D array

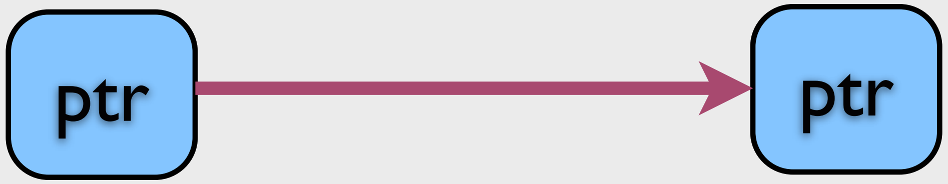
- General concept



```
int **ia
```

`malloc` A 2D array

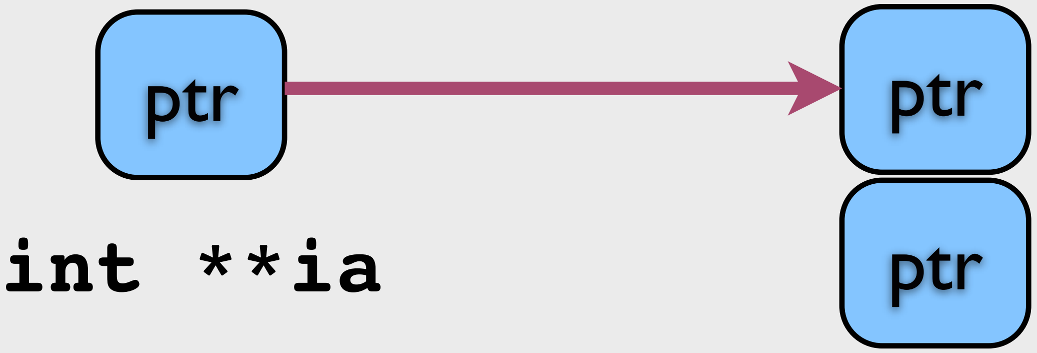
- General concept



`int **ia`

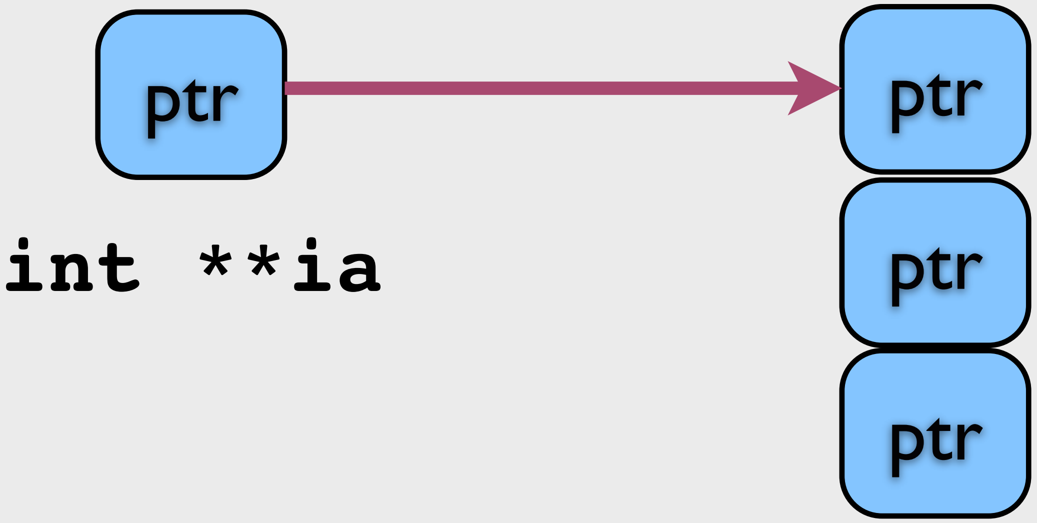
`malloc` A 2D array

- General concept



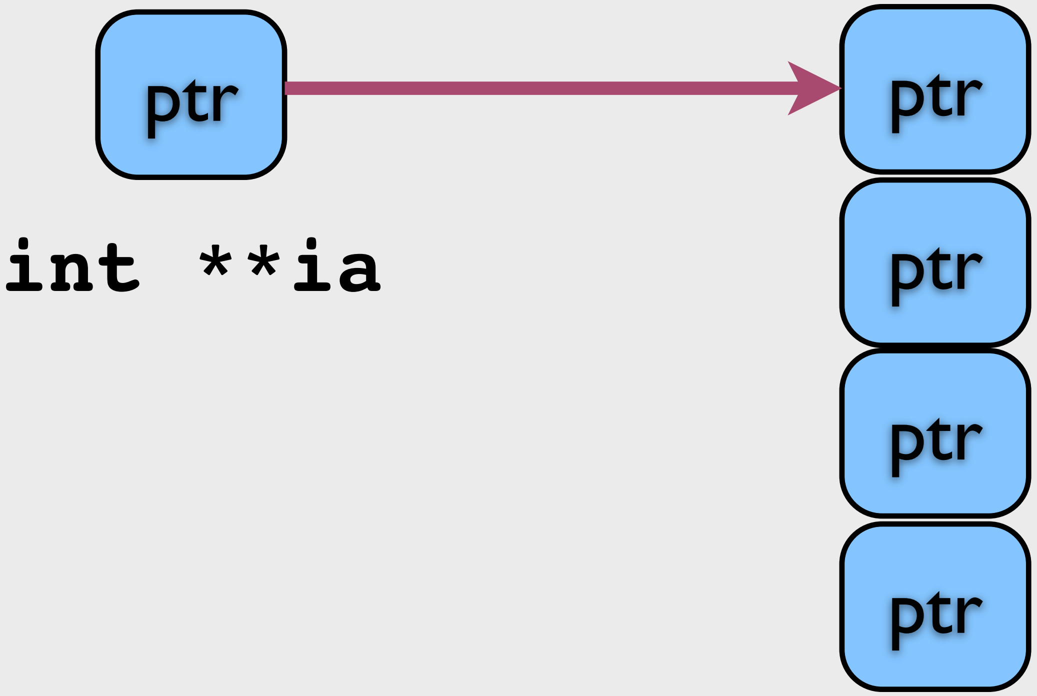
`malloc` A 2D array

- General concept



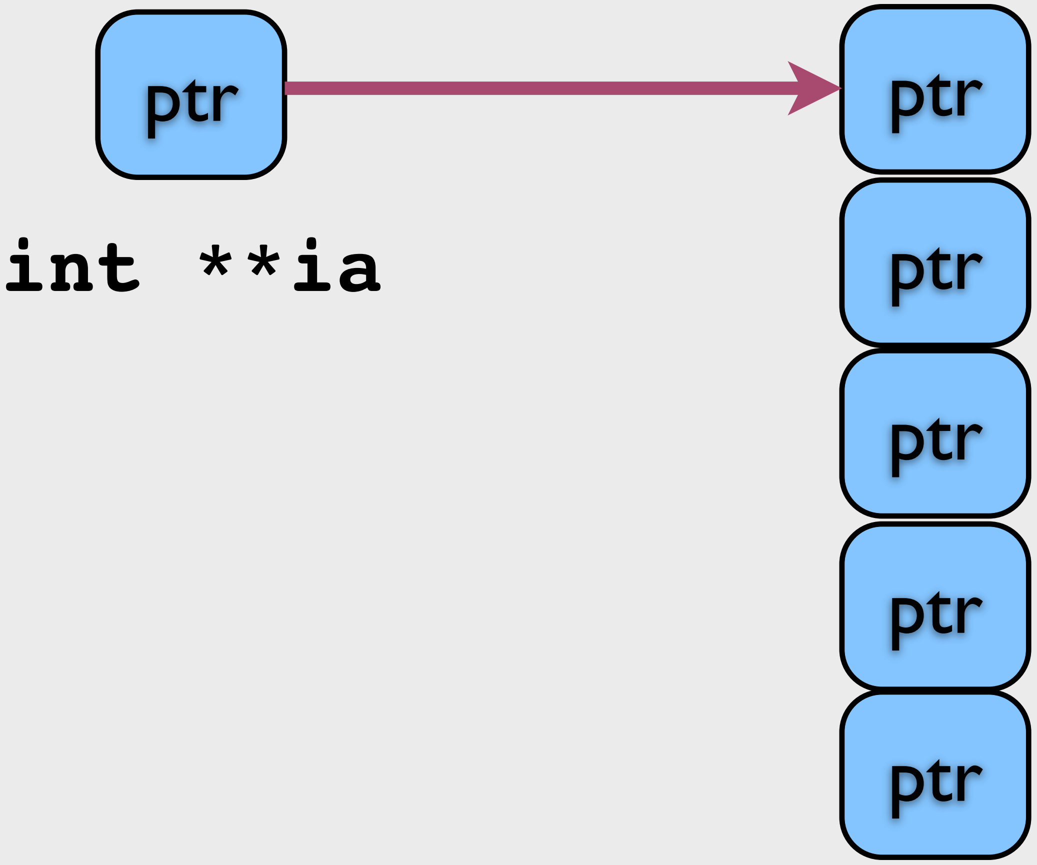
`malloc` A 2D array

- General concept



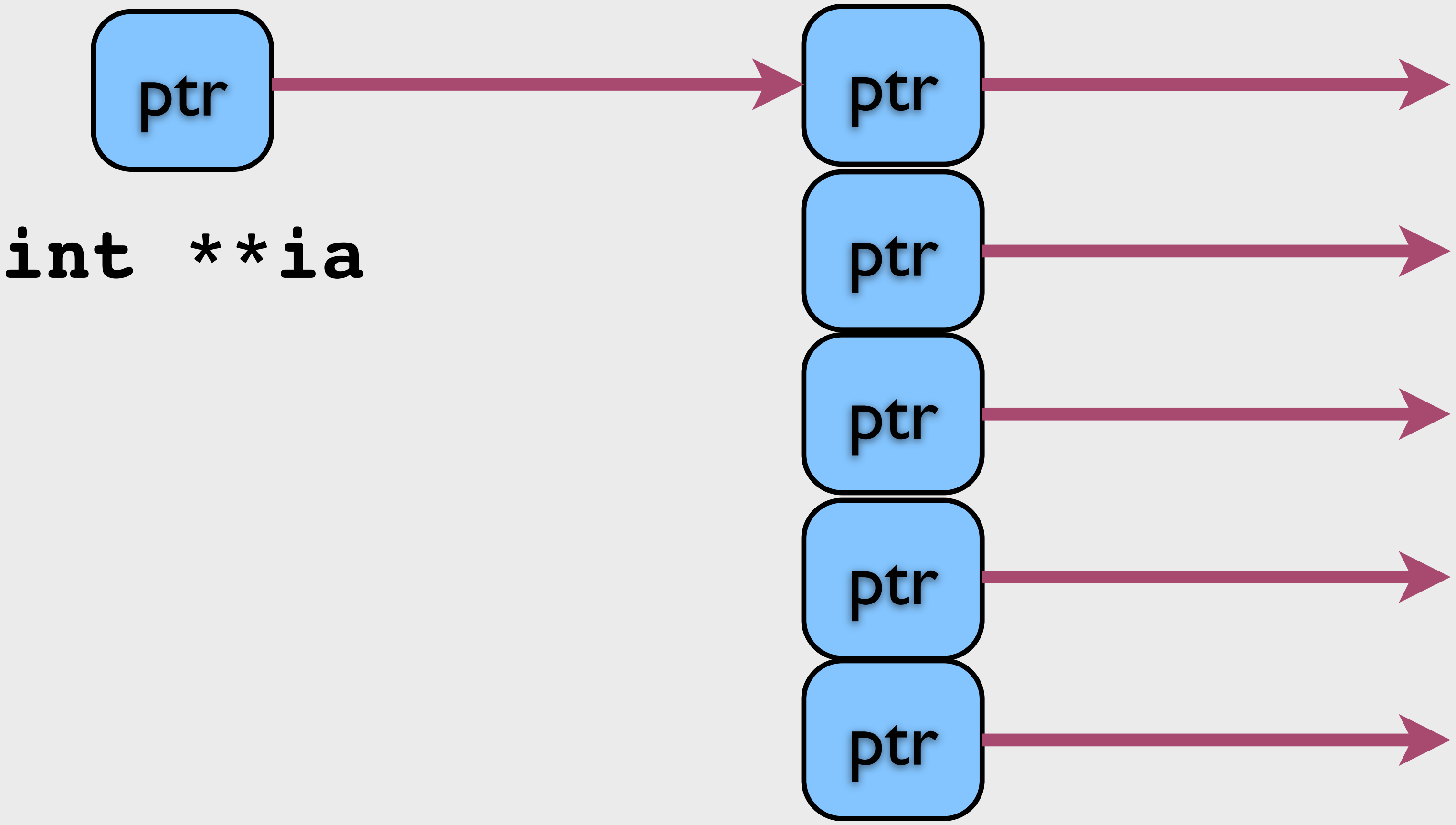
`malloc` A 2D array

- General concept



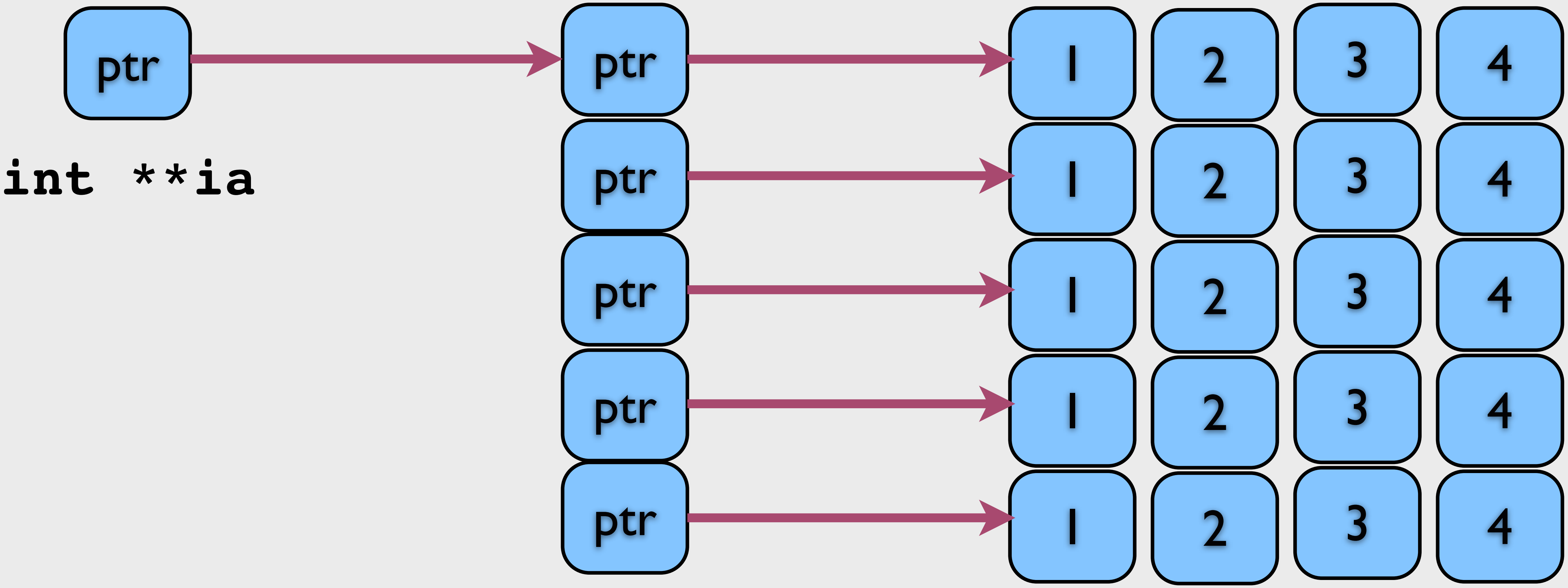
`malloc` A 2D array

- General concept



`malloc` A 2D array

- General concept



`malloc` A 2D array (1)

- Illustration

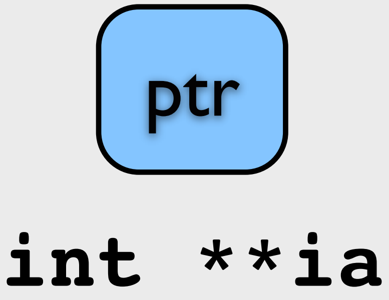
`malloc` A 2D array (1)

- Illustration

ptr

`malloc` A 2D array (1)

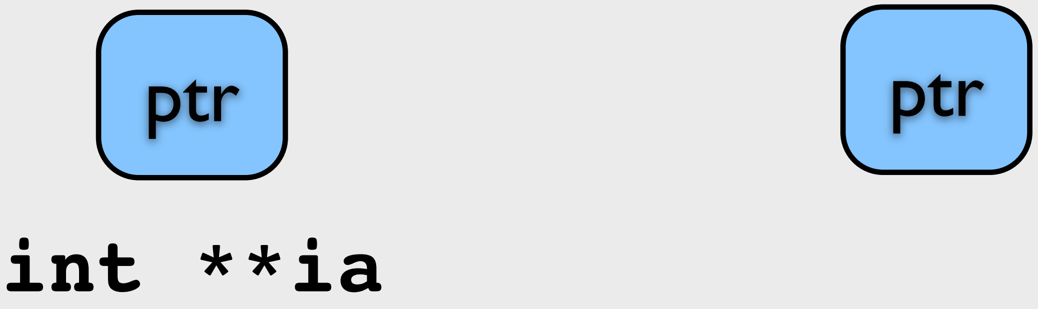
- Illustration



`int **ia`

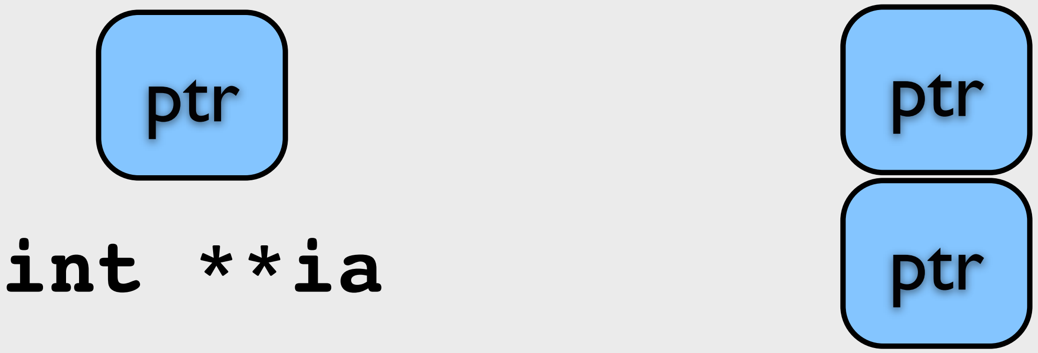
`malloc` A 2D array (1)

- Illustration



`malloc` A 2D array (1)

- Illustration



ptr

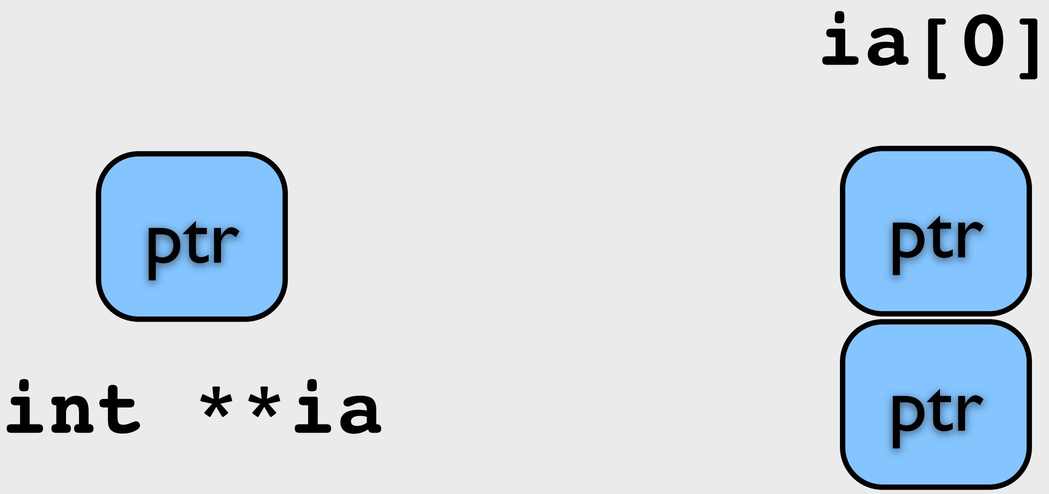
int **ia

ptr

ptr

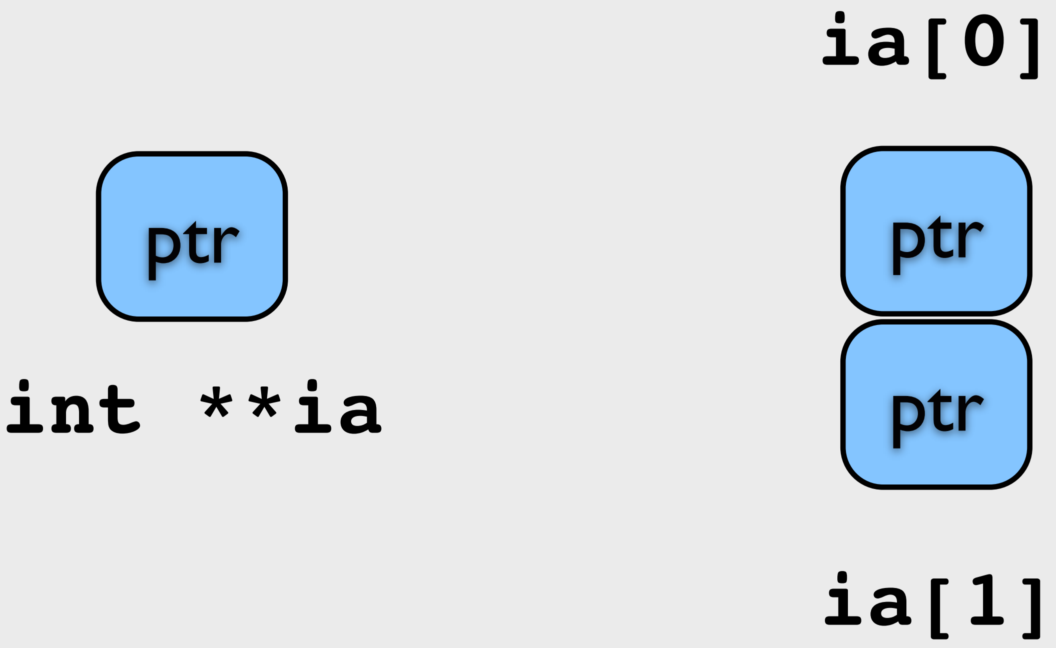
`malloc` A 2D array (1)

- Illustration



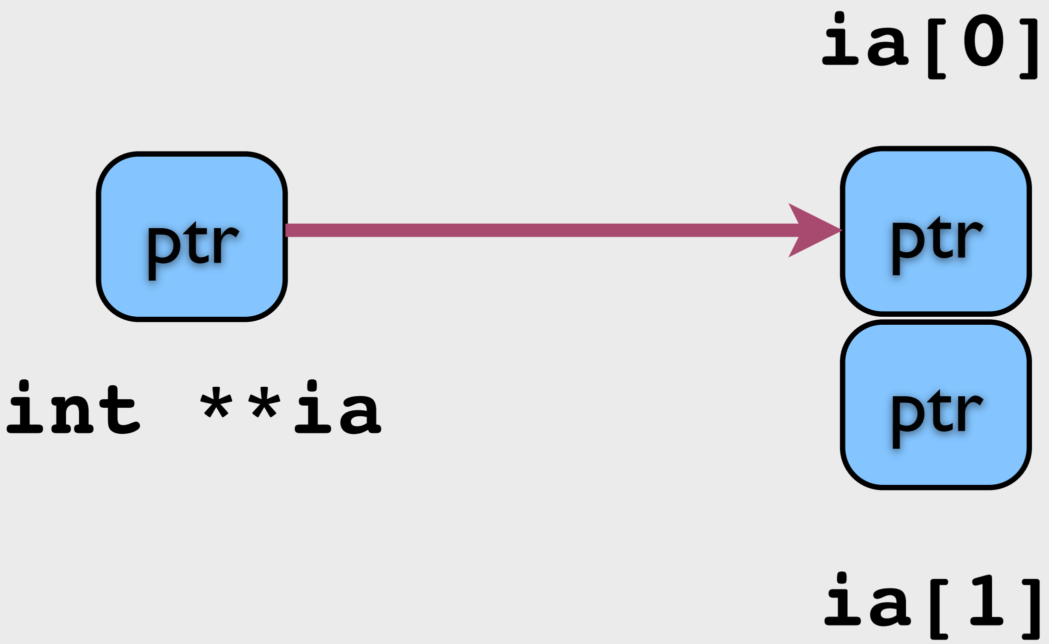
`malloc` A 2D array (1)

- Illustration



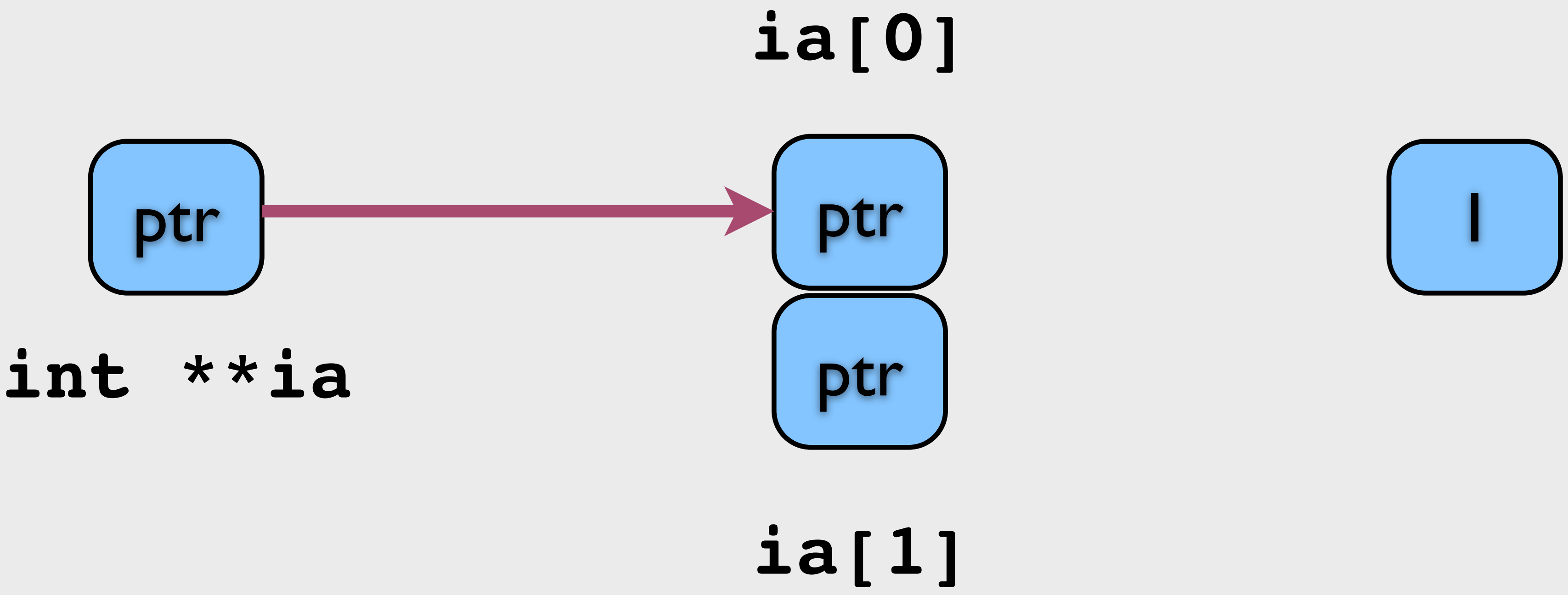
`malloc` A 2D array (1)

- Illustration



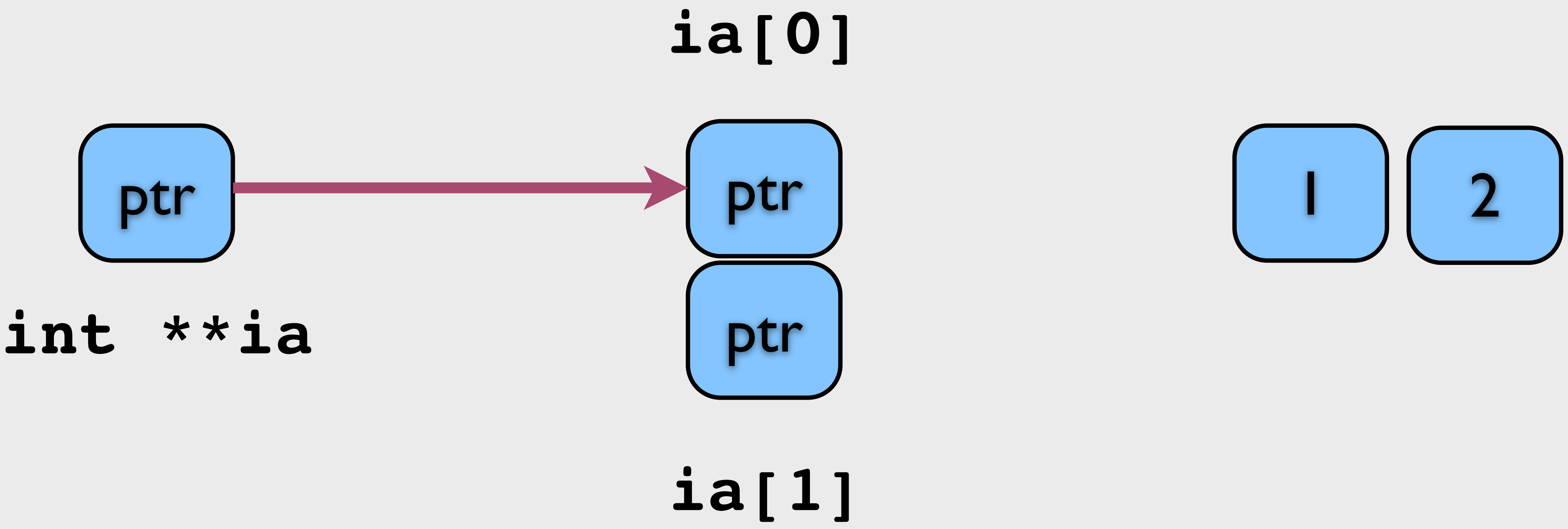
`malloc` A 2D array (1)

- Illustration



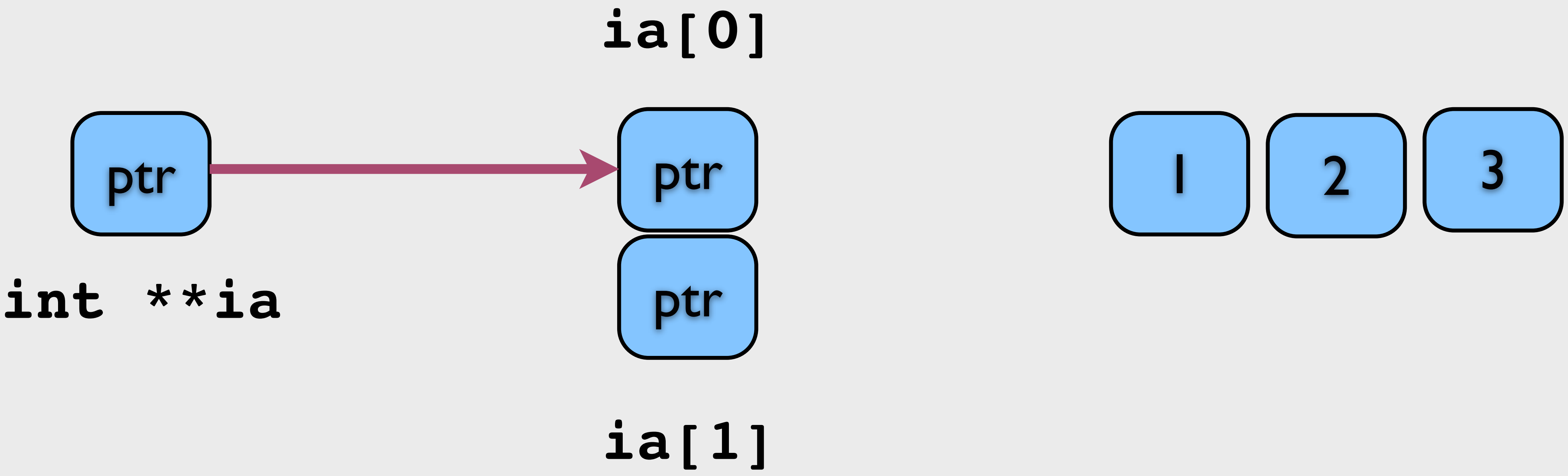
`malloc` A 2D array (1)

- Illustration



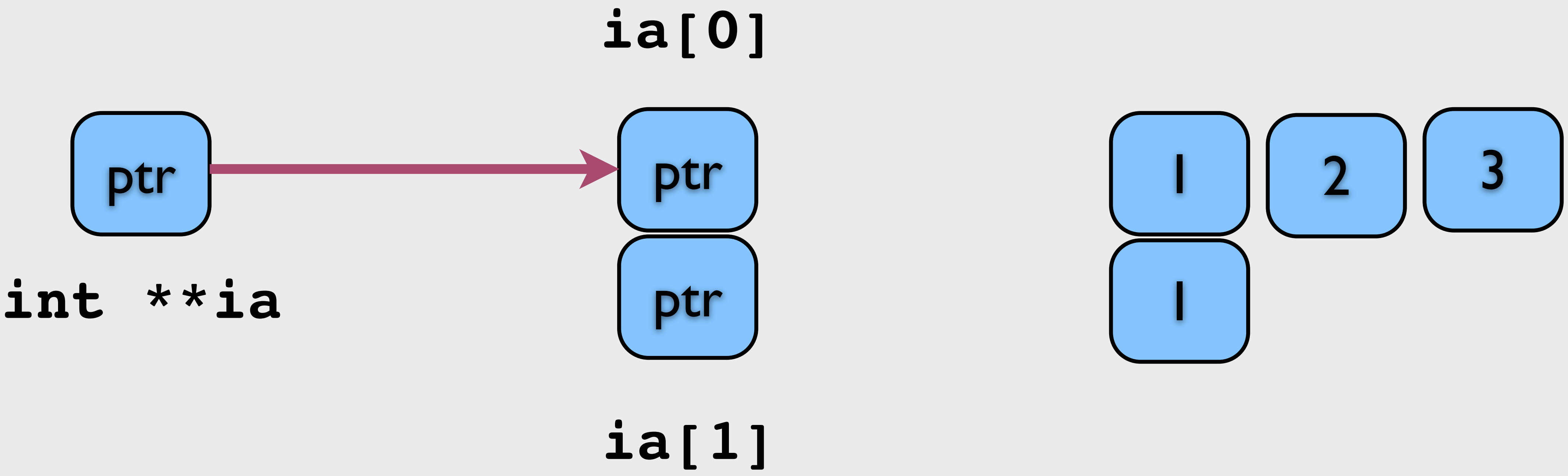
malloc A 2D array (1)

- Illustration



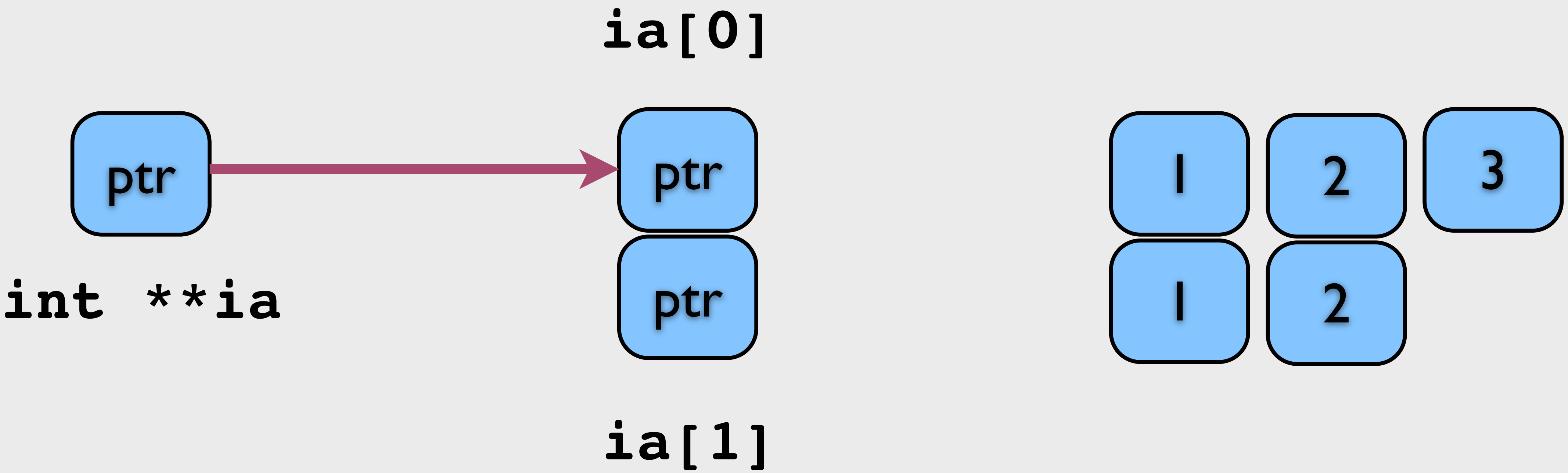
`malloc` A 2D array (1)

- Illustration



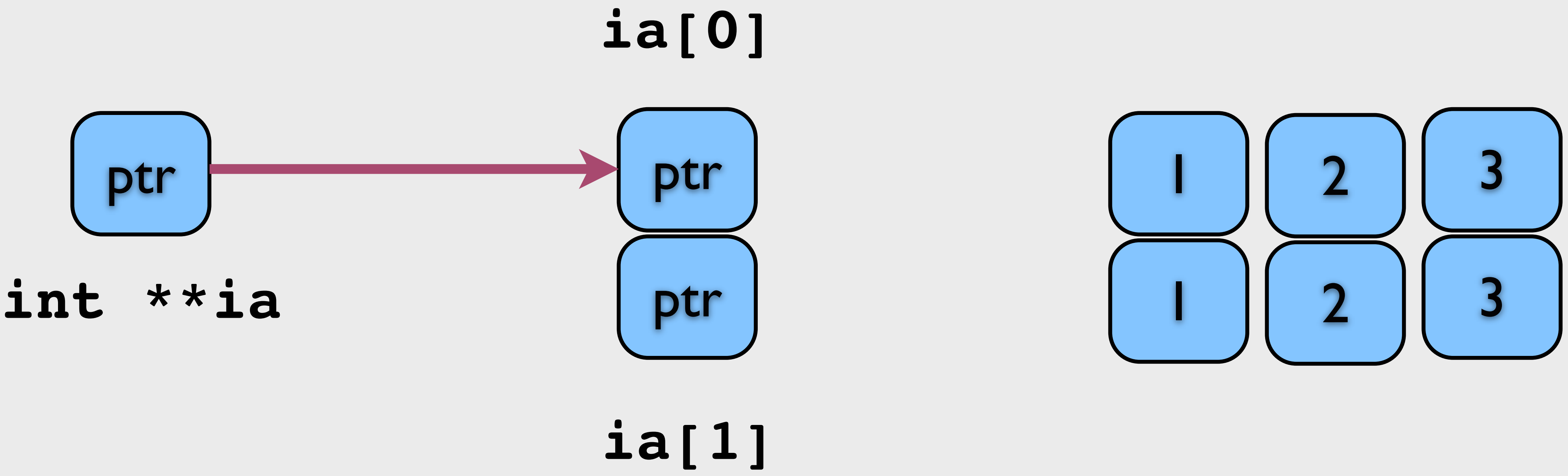
`malloc` A 2D array (1)

- Illustration



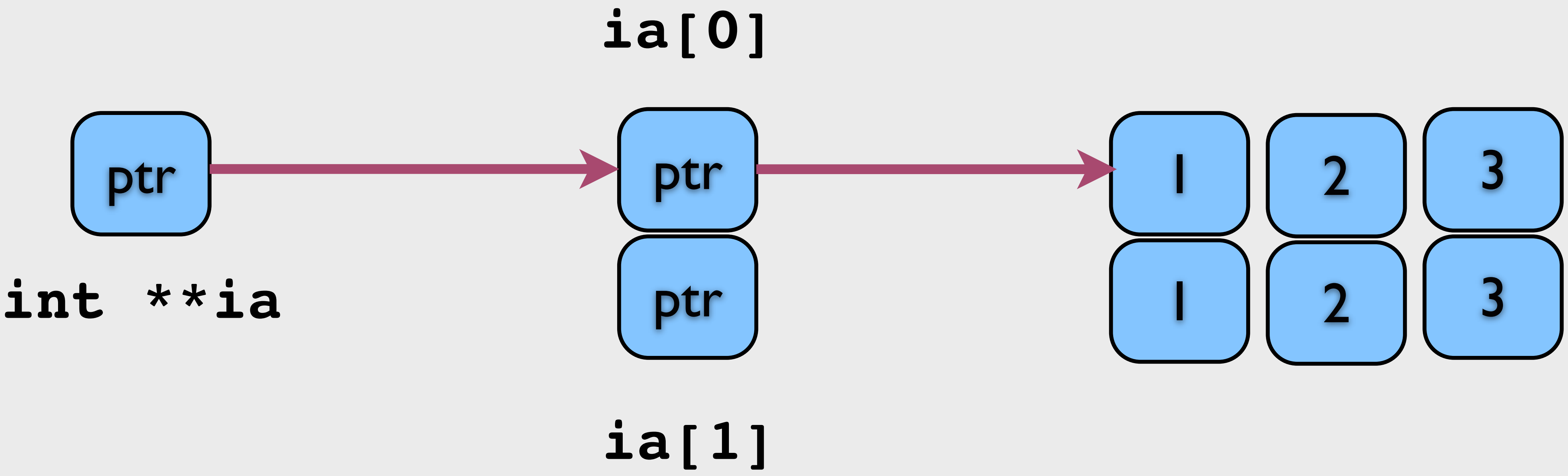
`malloc` A 2D array (1)

- Illustration



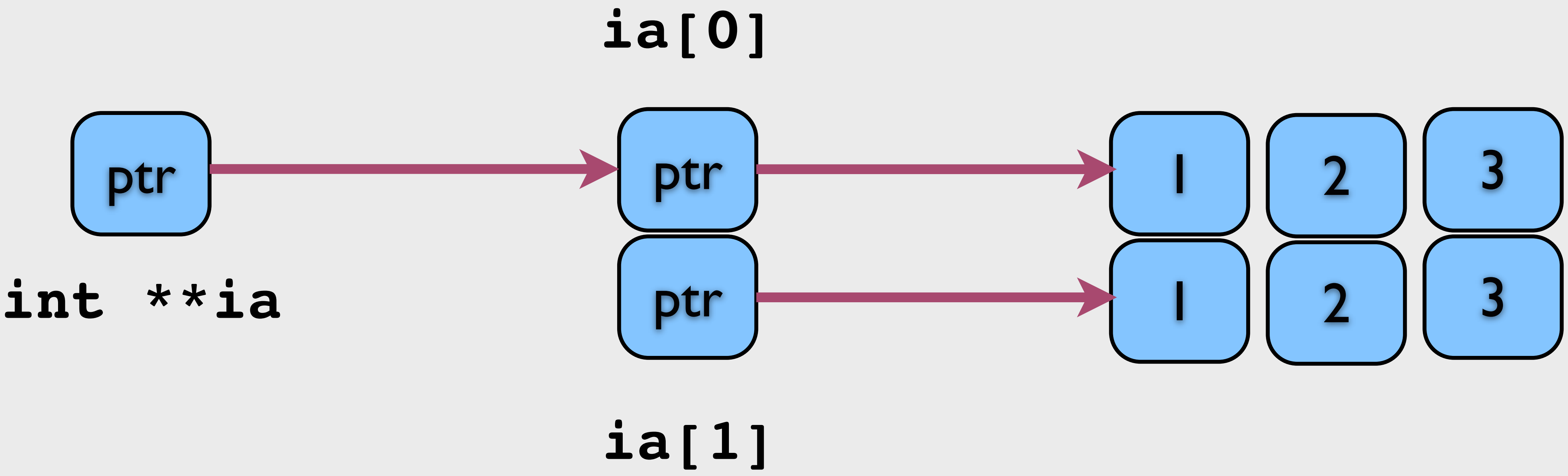
`malloc` A 2D array (1)

- Illustration



`malloc` A 2D array (1)

- Illustration



`malloc` A 2D array (1)

- Create a dynamic 2D array

malloc A 2D array (1)

- Create a dynamic 2D array

```
int **ia = (int **)malloc(sizey * sizeof(void *));
```

malloc A 2D array (1)

- Create a dynamic 2D array

```
int **ia = (int **)malloc(sizey * sizeof(void *));

for(y = 0; y != sizey; ++y)
      ia[y] = (int *)malloc(sizex * sizeof(int));
```

`malloc` A 2D array (1)

`malloc` A 2D array (1)

- How to free the array

`malloc` A 2D array (1)

- How to free the array

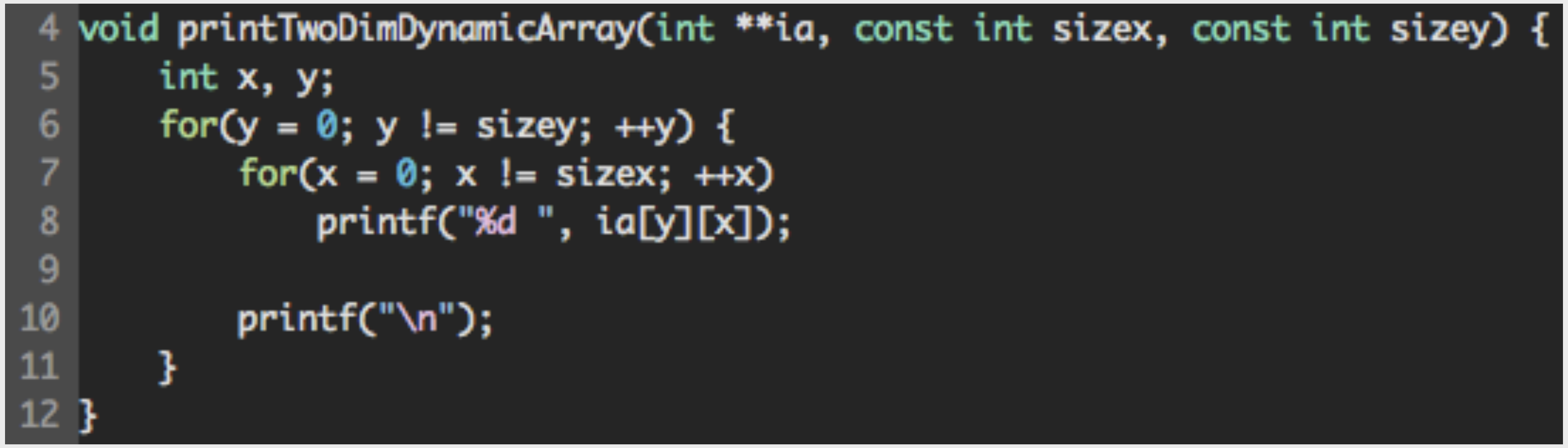
```
for(y = 0; y != sizey; ++y) {
  free(ia[y]);  /* free the internal array first */
}
free(ia);  /* free the external array */
```

`malloc` A 2D array (1)

- Example: array_dynamic_two_dim_1.c

`malloc` A 2D array (1)

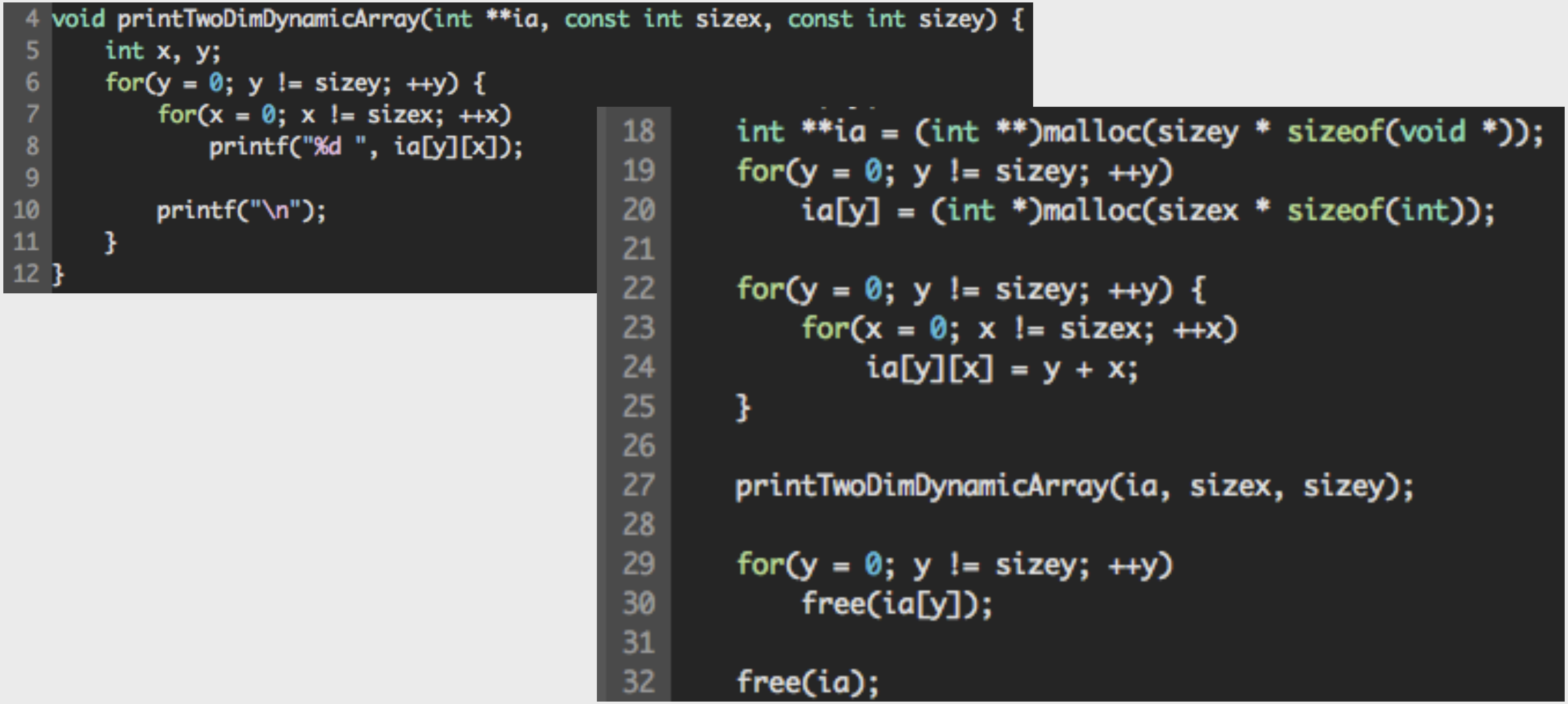
- Example: array_dynamic_two_dim_1.c



```c
void printTwoDimDynamicArray(int **ia, const int sizex, const int sizey) {
    int x, y;
    for(y = 0; y != sizey; ++y) {
        for(x = 0; x != sizex; ++x)
            printf("%d ", ia[y][x]);

        printf("\n");
    }
}
```

`malloc` A 2D array (1)

- Example: array_dynamic_two_dim_1.c

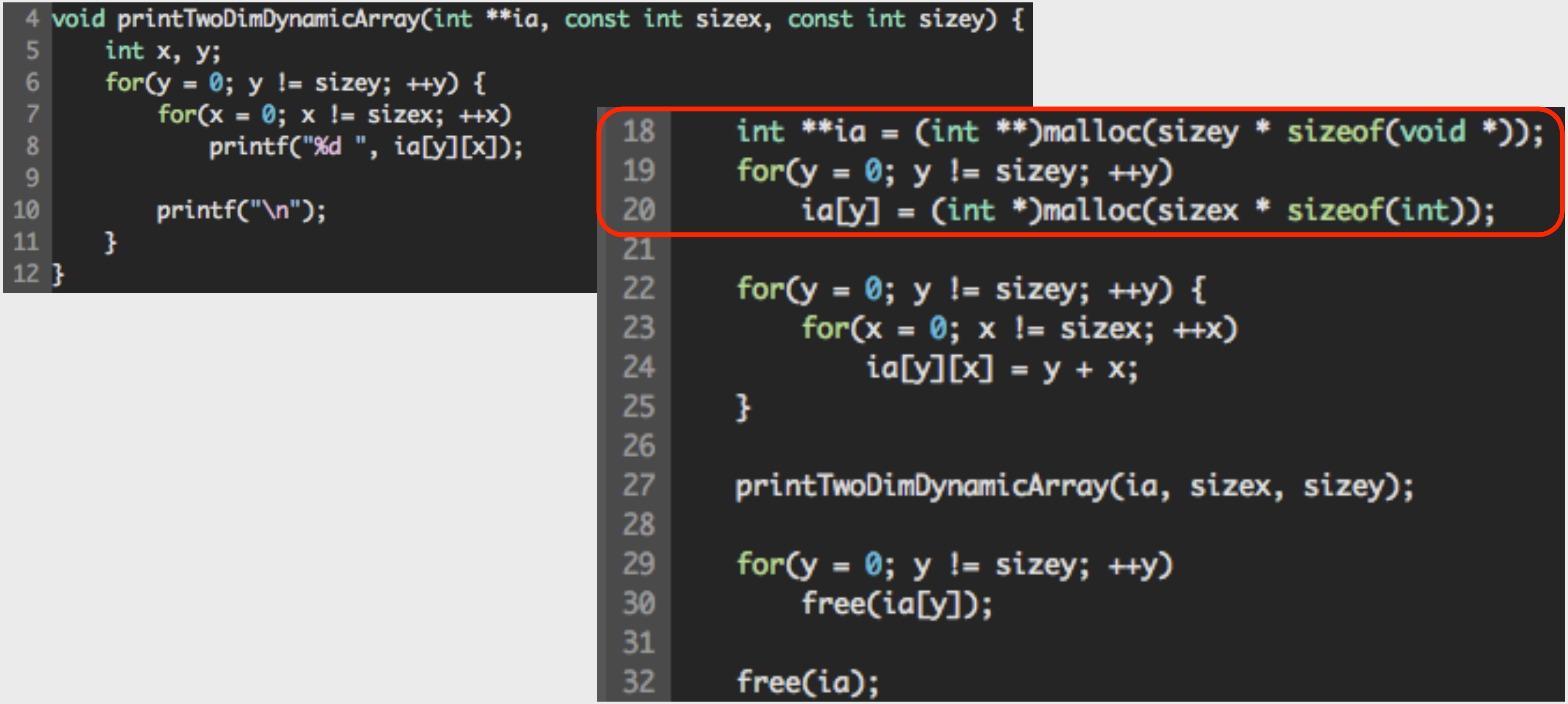


```c
void printTwoDimDynamicArray(int **ia, const int sizex, const int sizey) {
    int x, y;
    for(y = 0; y != sizey; ++y) {
        for(x = 0; x != sizex; ++x)
            printf("%d ", ia[y][x]);

        printf("\n");
    }
}
```

```c
    int **ia = (int **)malloc(sizey * sizeof(void *));
    for(y = 0; y != sizey; ++y)
        ia[y] = (int *)malloc(sizex * sizeof(int));

    for(y = 0; y != sizey; ++y) {
        for(x = 0; x != sizex; ++x)
            ia[y][x] = y + x;
    }

    printTwoDimDynamicArray(ia, sizex, sizey);

    for(y = 0; y != sizey; ++y)
        free(ia[y]);

    free(ia);
```

`malloc` A 2D array (1)

- Example: array_dynamic_two_dim_1.c

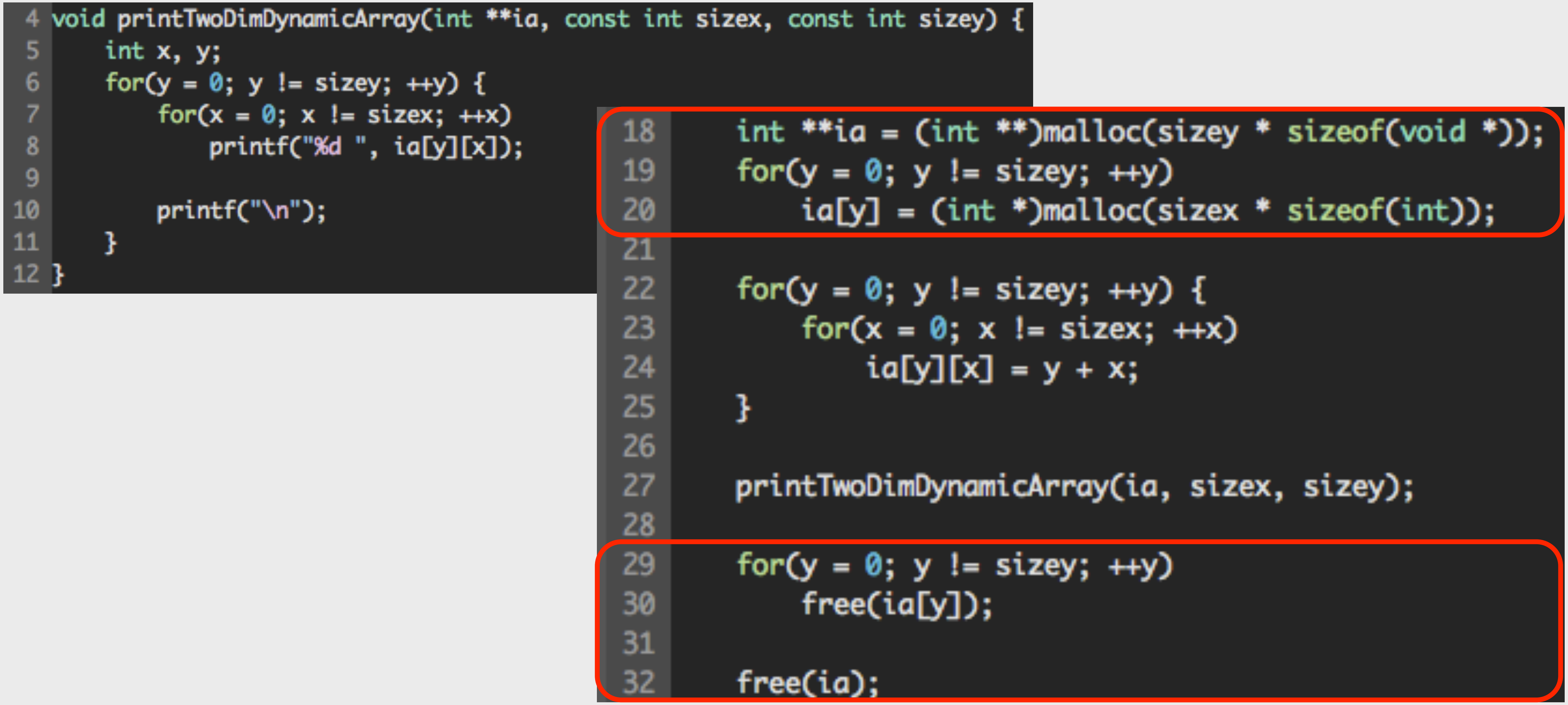


```c
void printTwoDimDynamicArray(int **ia, const int sizex, const int sizey) {
    int x, y;
    for(y = 0; y != sizey; ++y) {
        for(x = 0; x != sizex; ++x)
            printf("%d ", ia[y][x]);

        printf("\n");
    }
}
```

```c
    int **ia = (int **)malloc(sizey * sizeof(void *));
    for(y = 0; y != sizey; ++y)
        ia[y] = (int *)malloc(sizex * sizeof(int));

    for(y = 0; y != sizey; ++y) {
        for(x = 0; x != sizex; ++x)
            ia[y][x] = y + x;
    }

    printTwoDimDynamicArray(ia, sizex, sizey);

    for(y = 0; y != sizey; ++y)
        free(ia[y]);

    free(ia);
```

malloc A 2D array (1)

- Example: array_dynamic_two_dim_1.c

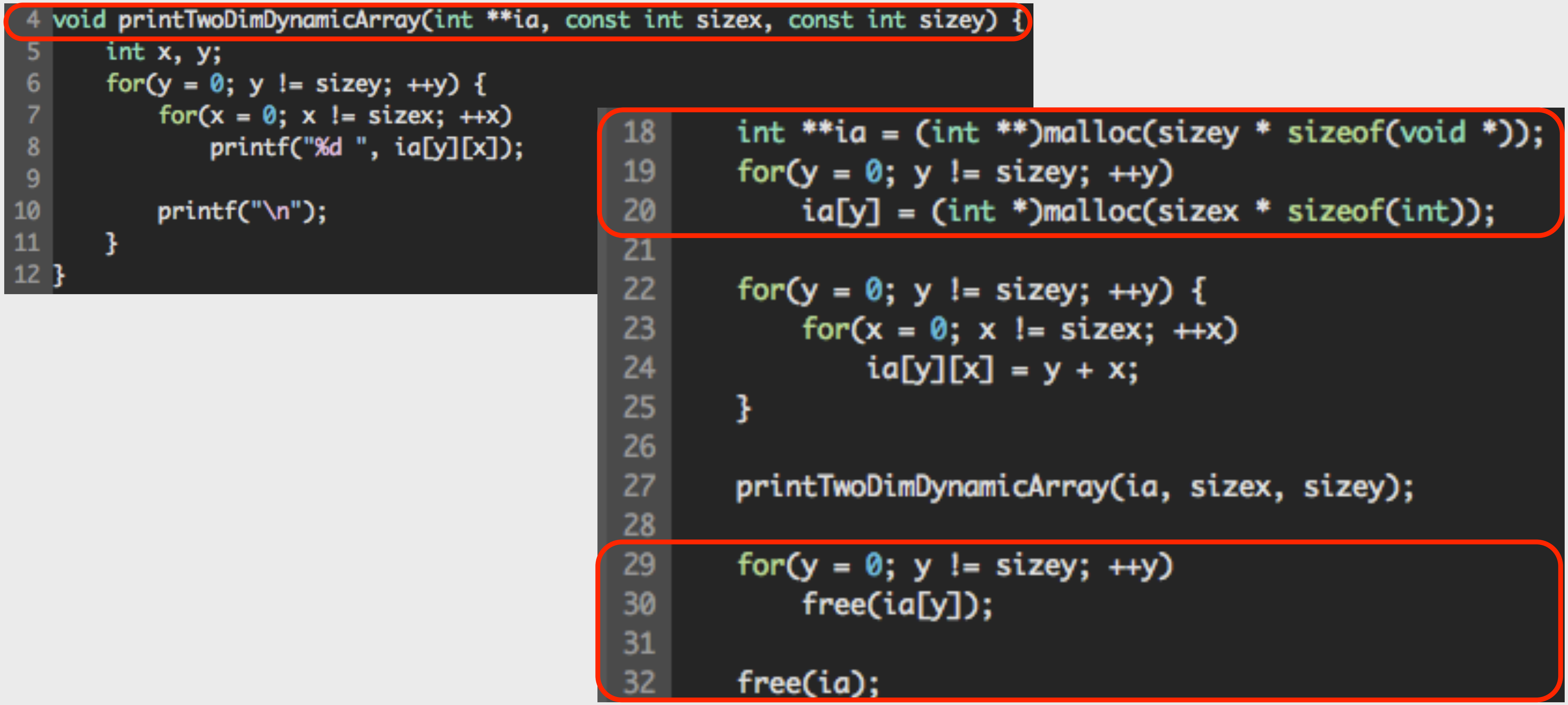


```c
void printTwoDimDynamicArray(int **ia, const int sizex, const int sizey) {
    int x, y;
    for(y = 0; y != sizey; ++y) {
        for(x = 0; x != sizex; ++x)
            printf("%d ", ia[y][x]);

        printf("\n");
    }
}
```

```c
    int **ia = (int **)malloc(sizey * sizeof(void *));
    for(y = 0; y != sizey; ++y)
        ia[y] = (int *)malloc(sizex * sizeof(int));

    for(y = 0; y != sizey; ++y) {
        for(x = 0; x != sizex; ++x)
            ia[y][x] = y + x;
    }

    printTwoDimDynamicArray(ia, sizex, sizey);

    for(y = 0; y != sizey; ++y)
        free(ia[y]);

    free(ia);
```

malloc A 2D array (1)

- Example: array_dynamic_two_dim_1.c

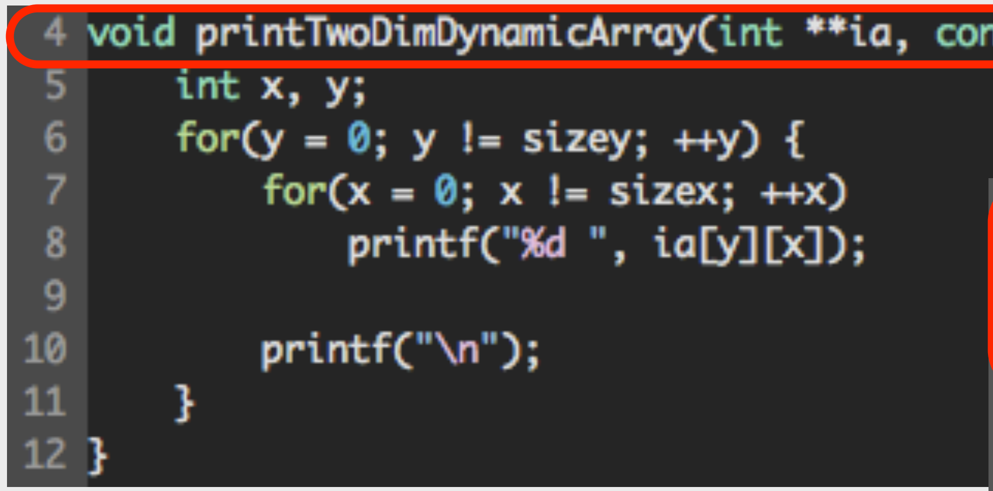


```
void printTwoDimDynamicArray(int **ia, const int sizex, const int sizey) {
    int x, y;
    for(y = 0; y != sizey; ++y) {
        for(x = 0; x != sizex; ++x)
            printf("%d ", ia[y][x]);

        printf("\n");
    }
}
```

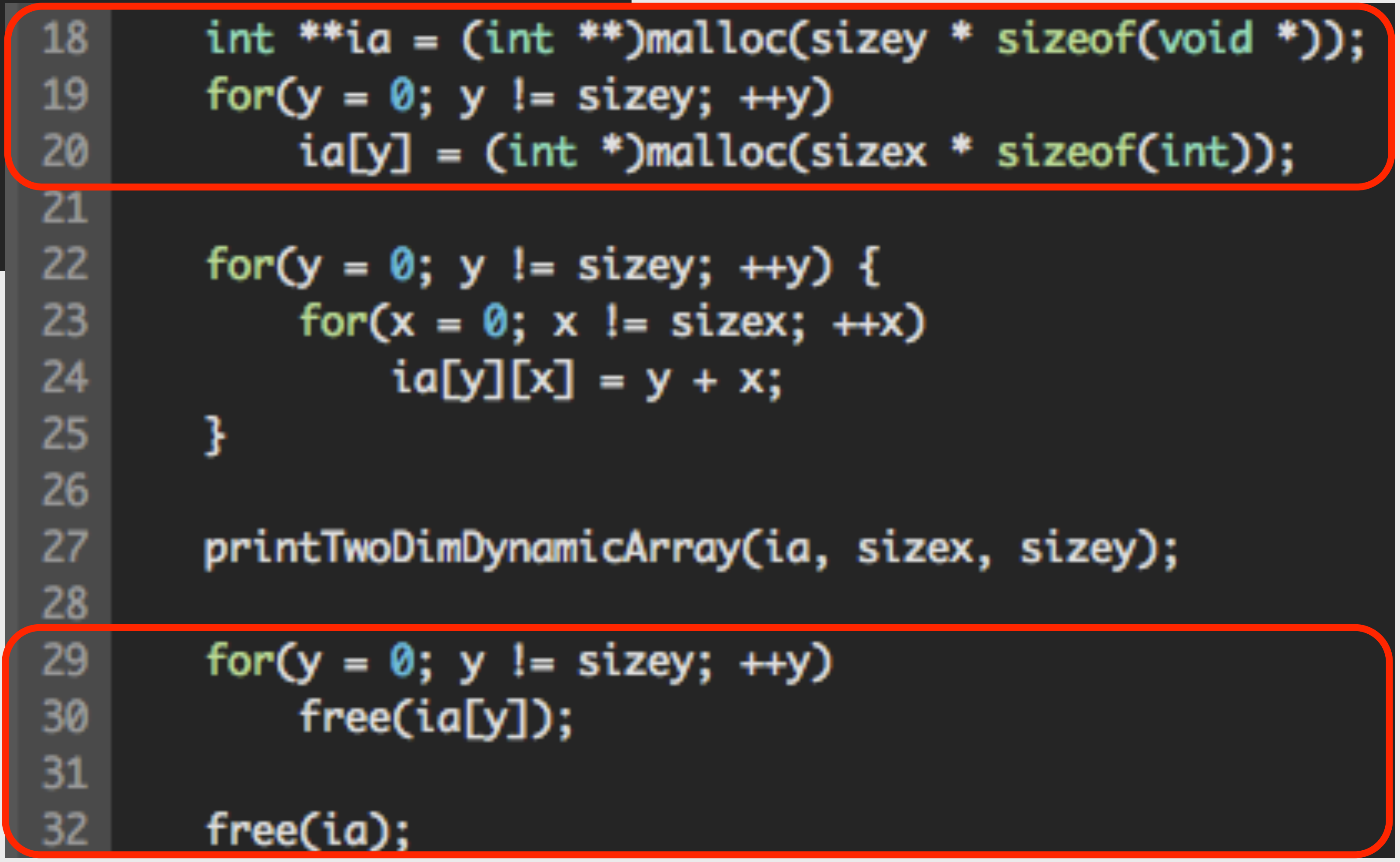
```
    int **ia = (int **)malloc(sizey * sizeof(void *));
    for(y = 0; y != sizey; ++y)
        ia[y] = (int *)malloc(sizex * sizeof(int));

    for(y = 0; y != sizey; ++y) {
        for(x = 0; x != sizex; ++x)
            ia[y][x] = y + x;
    }

    printTwoDimDynamicArray(ia, sizex, sizey);

    for(y = 0; y != sizey; ++y)
        free(ia[y]);

    free(ia);
```

`malloc` A 2D array (1)

- Example: array_dynamic_two_dim_1.c



```c
void printTwoDimDynamicArray(int **ia, const int sizex, const int sizey) {
    int x, y;
    for(y = 0; y != sizey; ++y) {
        for(x = 0; x != sizex; ++x)
            printf("%d ", ia[y][x]);

        printf("\n");
    }
}
```



```c
    int **ia = (int **)malloc(sizey * sizeof(void *));
    for(y = 0; y != sizey; ++y)
        ia[y] = (int *)malloc(sizex * sizeof(int));

    for(y = 0; y != sizey; ++y) {
        for(x = 0; x != sizex; ++x)
            ia[y][x] = y + x;
    }

    printTwoDimDynamicArray(ia, sizex, sizey);

    for(y = 0; y != sizey; ++y)
        free(ia[y]);

    free(ia);
```

- Problem
  - memory fragments
  - multiple free

`malloc` A 2D array (2)

- Illustration

`malloc` A 2D array (2)

- Illustration

ptr

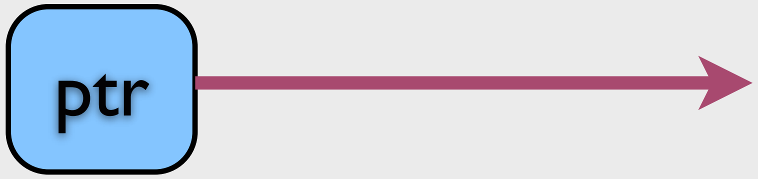
`malloc` A 2D array (2)

- Illustration

`int **ia`

`malloc` A 2D array (2)

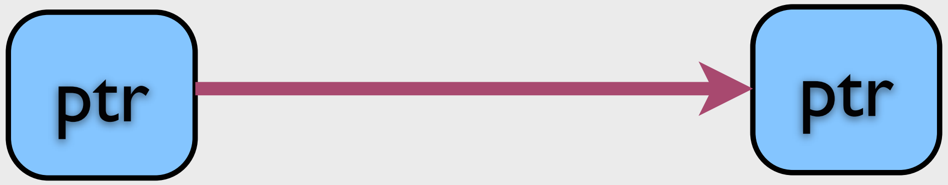
- Illustration



`int **ia`

`malloc` A 2D array (2)

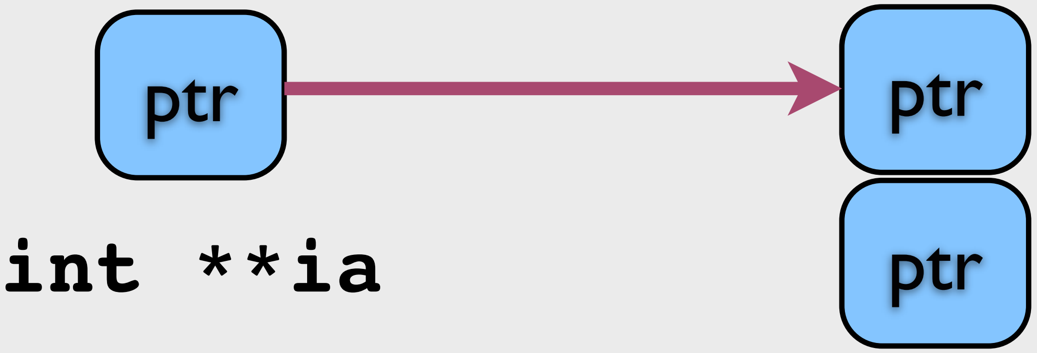
- Illustration



`int **ia`

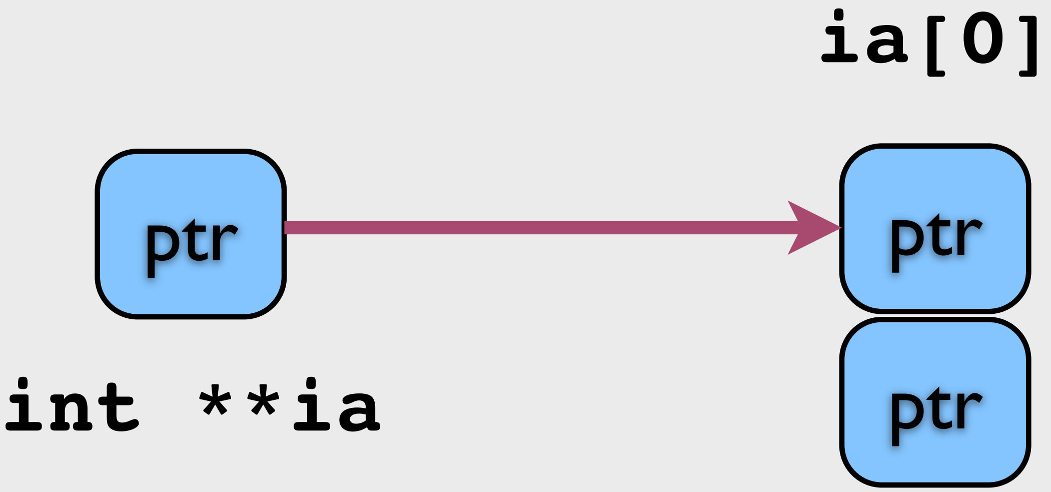
`malloc` A 2D array (2)

- Illustration



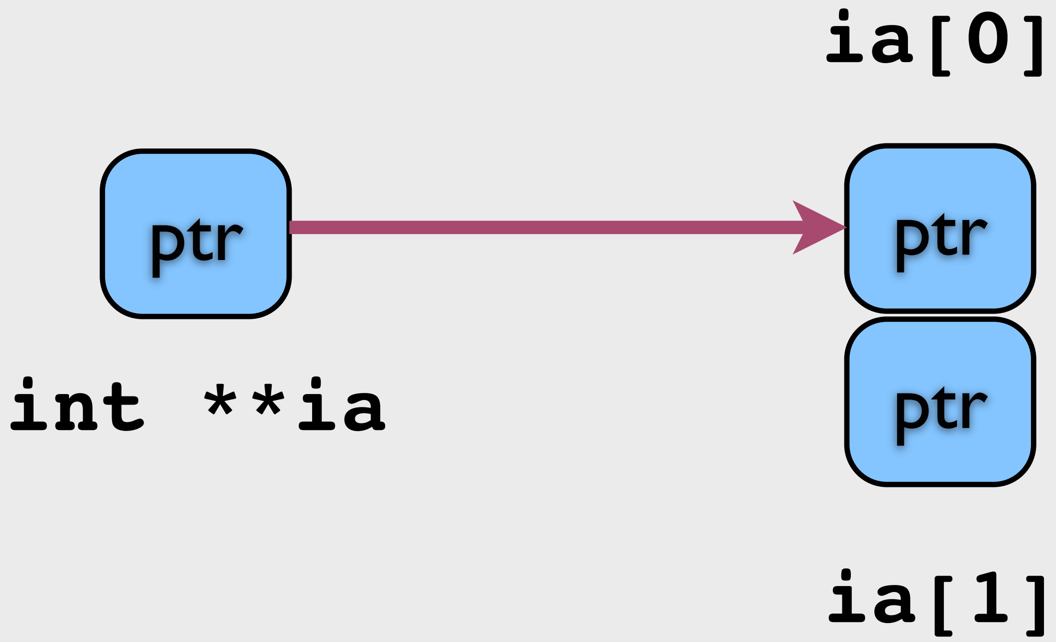
`malloc` A 2D array (2)

- Illustration



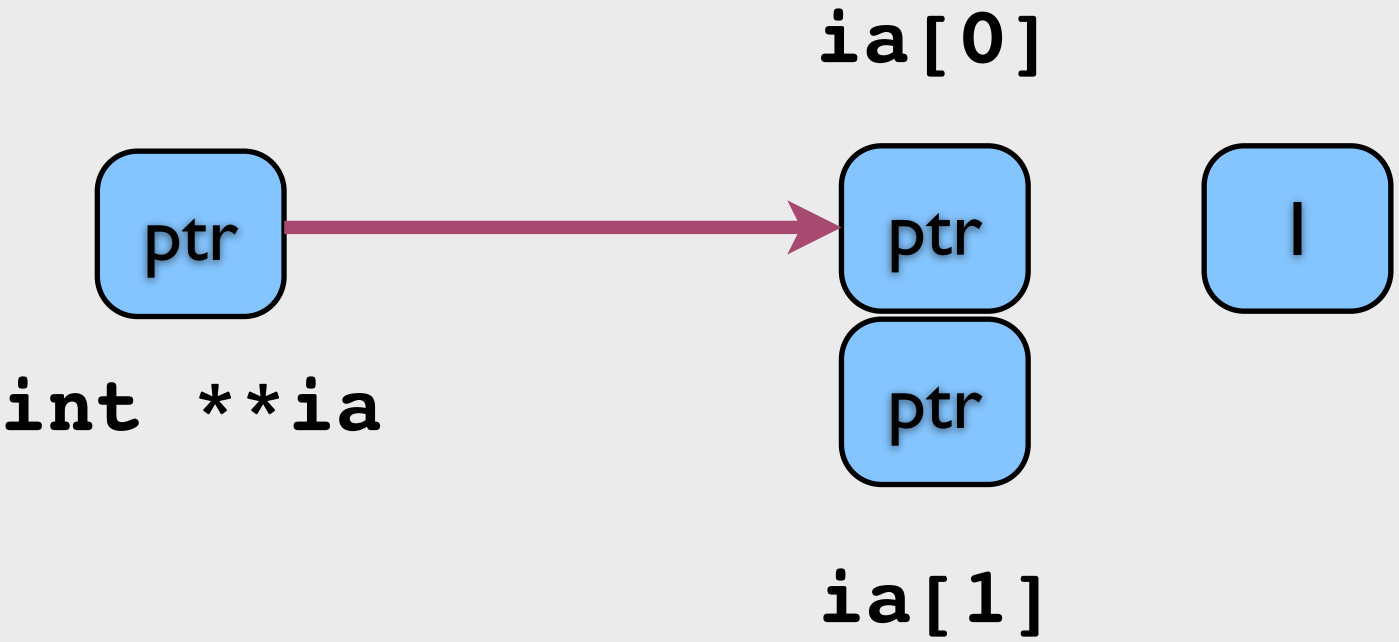
`malloc` A 2D array (2)

- Illustration



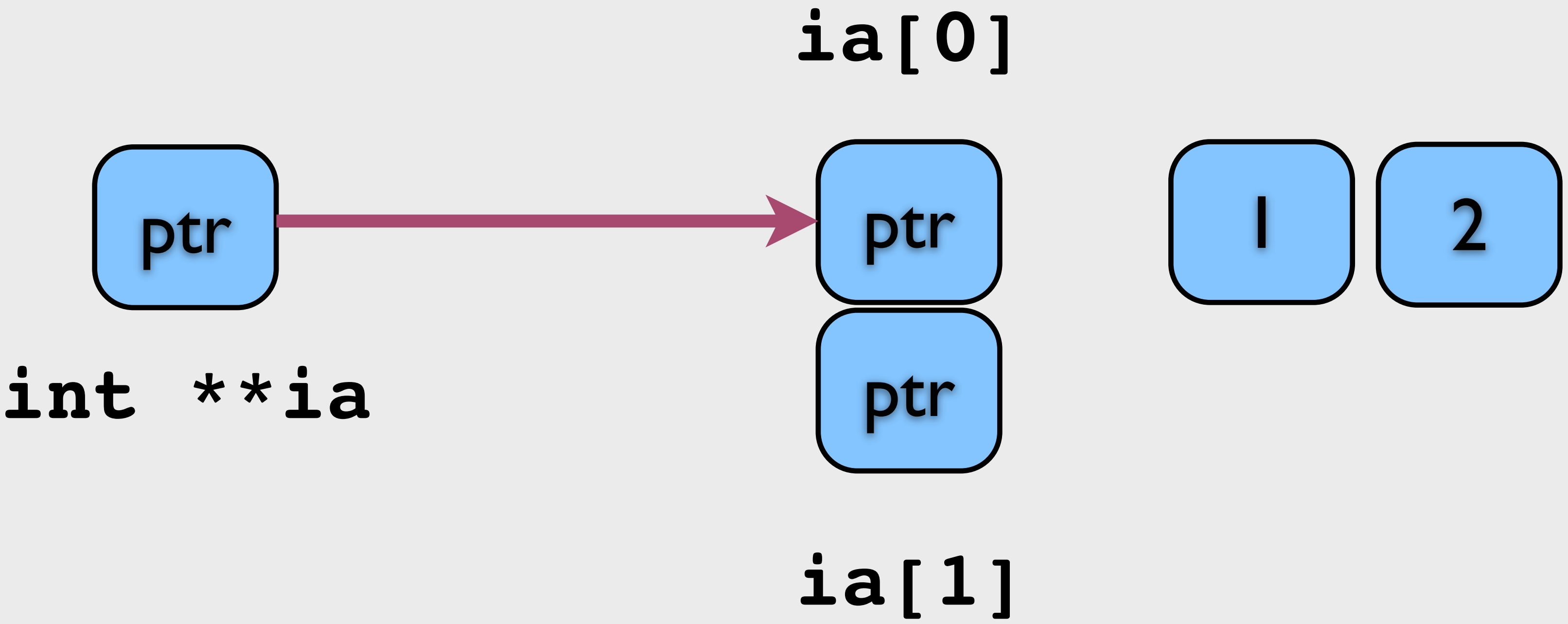
`malloc` A 2D array (2)

- Illustration



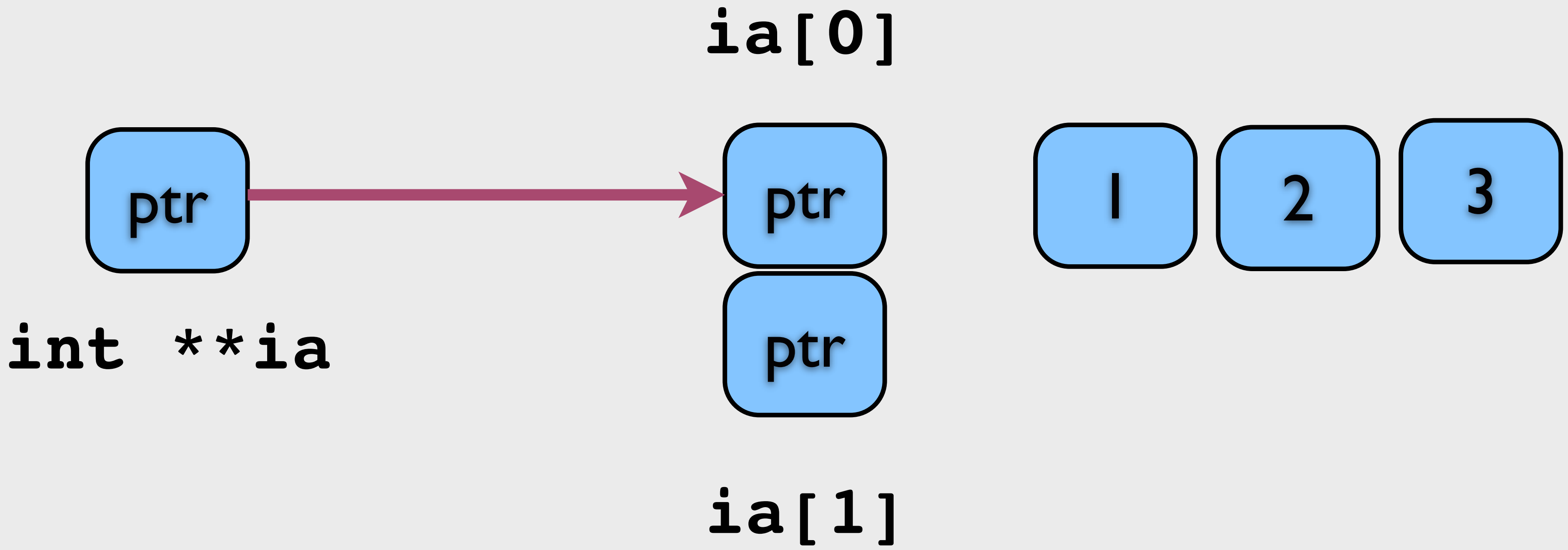
`malloc` A 2D array (2)

- Illustration



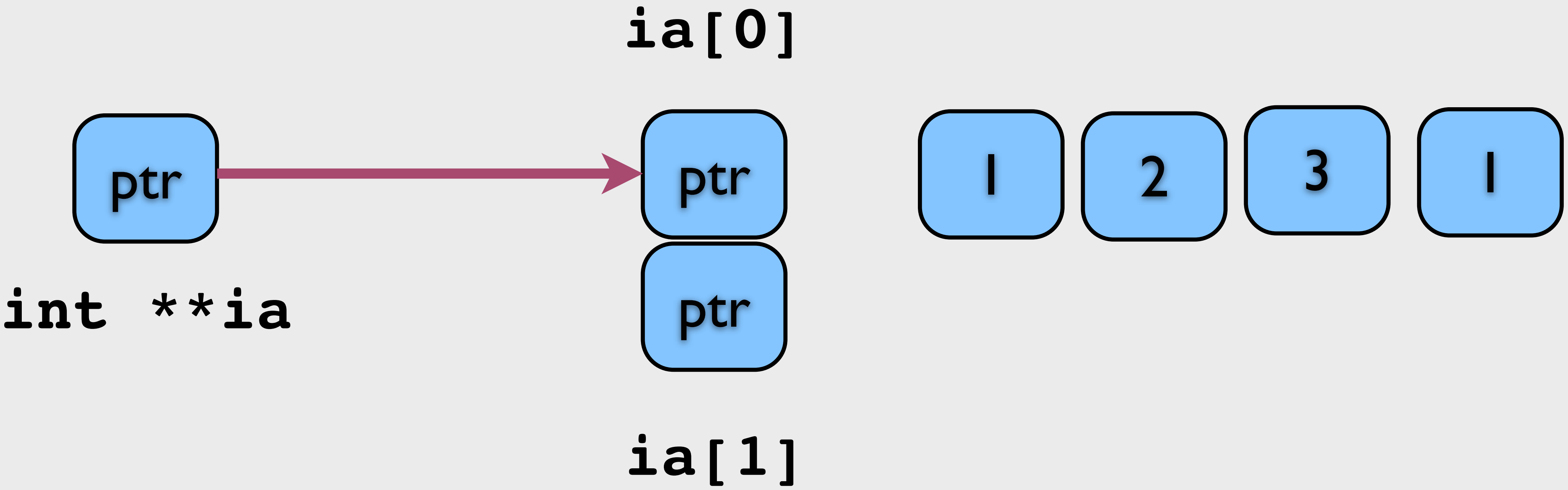
`malloc` A 2D array (2)

- Illustration



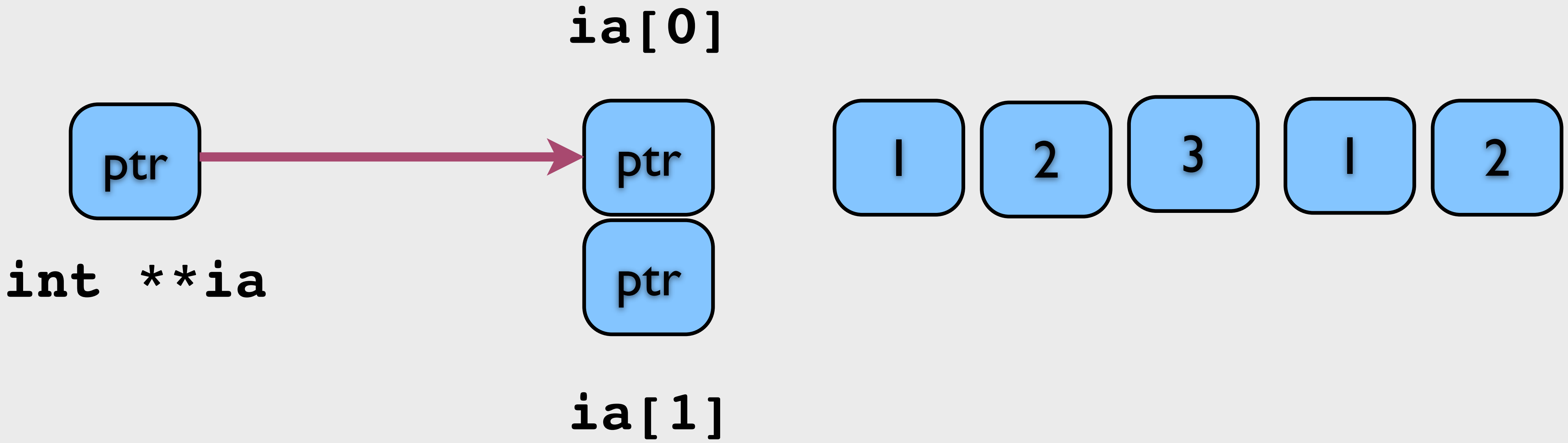
`malloc` A 2D array (2)

- Illustration



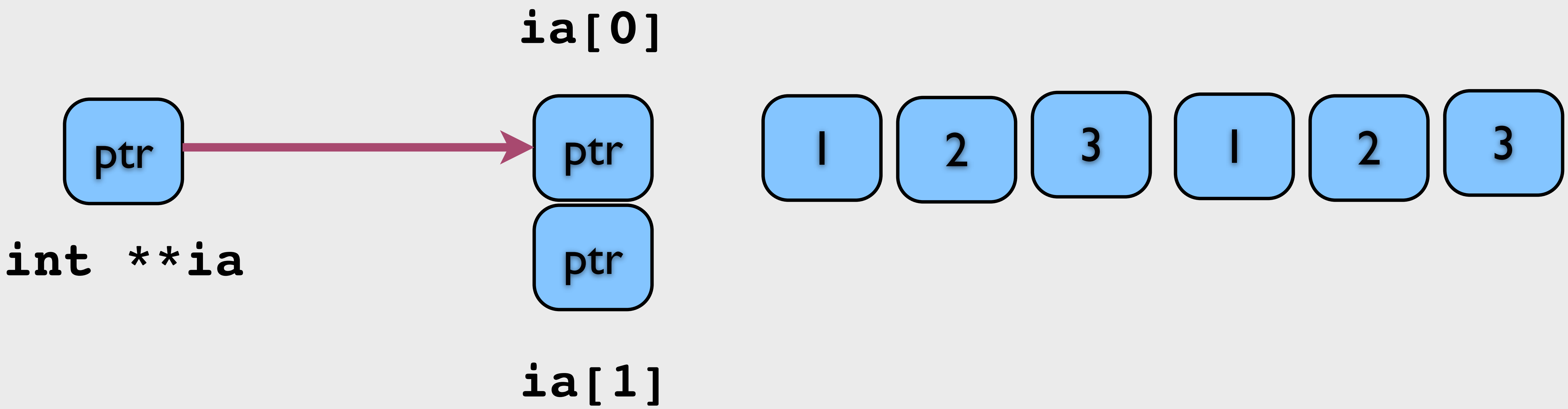
`malloc` A 2D array (2)

- Illustration



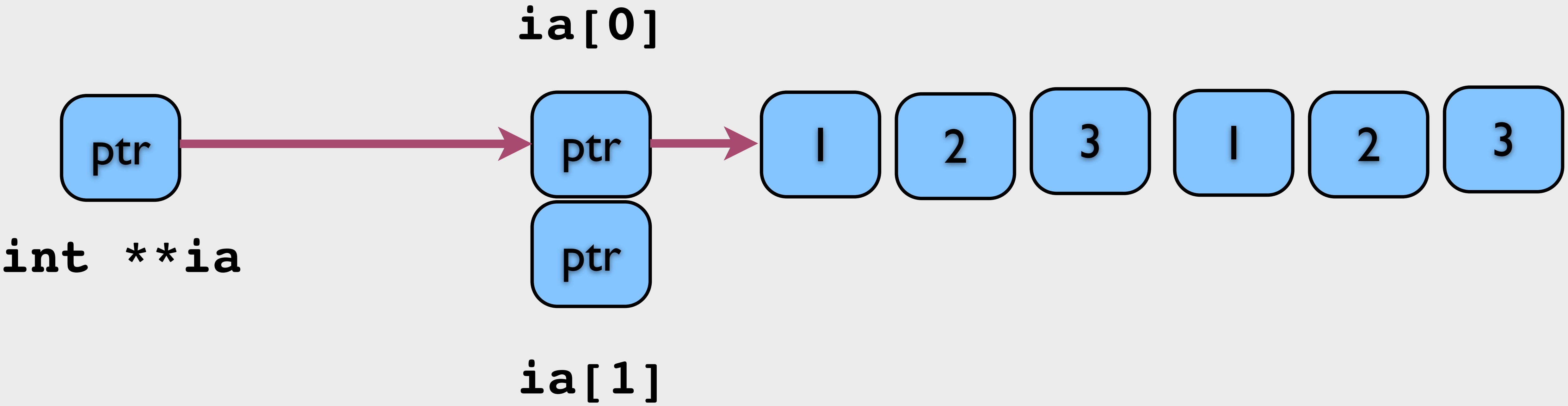
`malloc` A 2D array (2)

- Illustration



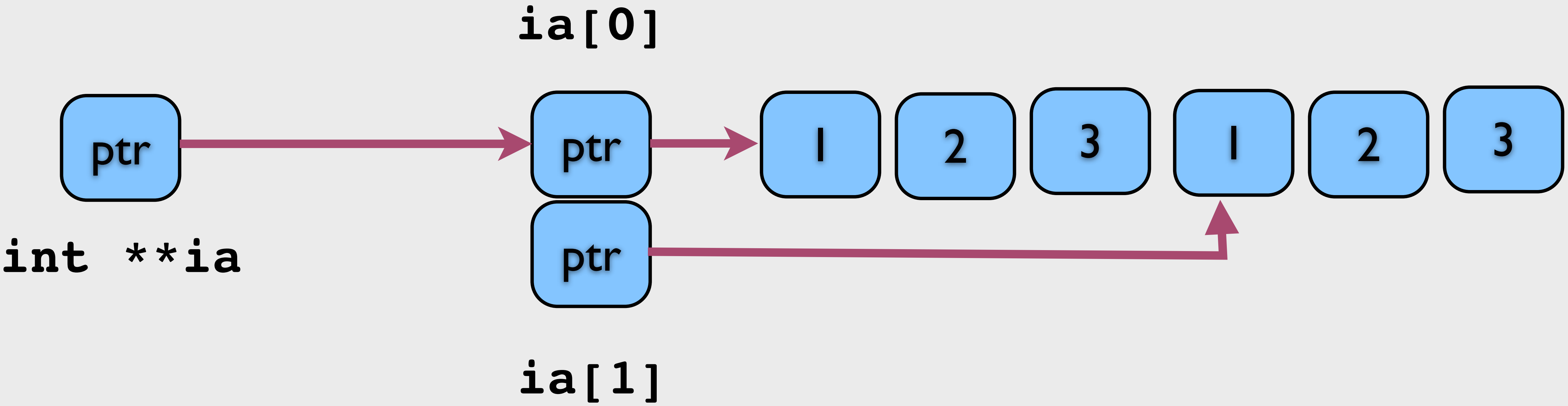
`malloc` A 2D array (2)

- Illustration



malloc A 2D array (2)

- Illustration



`malloc` A 2D array (2)

malloc A 2D array (2)

```
int **ia = (int **)malloc(sizey * sizeof(void *));
```

malloc A 2D array (2)

```
int **ia = (int **)malloc(sizey * sizeof(void *));
```

- **malloc** once for the internal arrays

```
int *iax = (int *) malloc(sizey * sizex * sizeof(int));
```

malloc A 2D array (2)

```
int **ia = (int **)malloc(sizey * sizeof(void *));
```

- **malloc** once for the internal arrays

```
int *iax = (int *) malloc(sizey * sizex * sizeof(int));
```

- assign the internal arrays to the external array

malloc A 2D array (2)

```
int **ia = (int **)malloc(sizey * sizeof(void *));
```

- **malloc** once for the internal arrays

```
int *iax = (int *) malloc(sizey * sizex * sizeof(int));
```

- assign the internal arrays to the external array

```
for(y = 0; y != sizey; ++y, iax += sizex) {
   ia[y] = iax;
}
```

malloc A 2D array (2)

```
int **ia = (int **)malloc(sizey * sizeof(void *));
```

- **malloc** once for the internal arrays

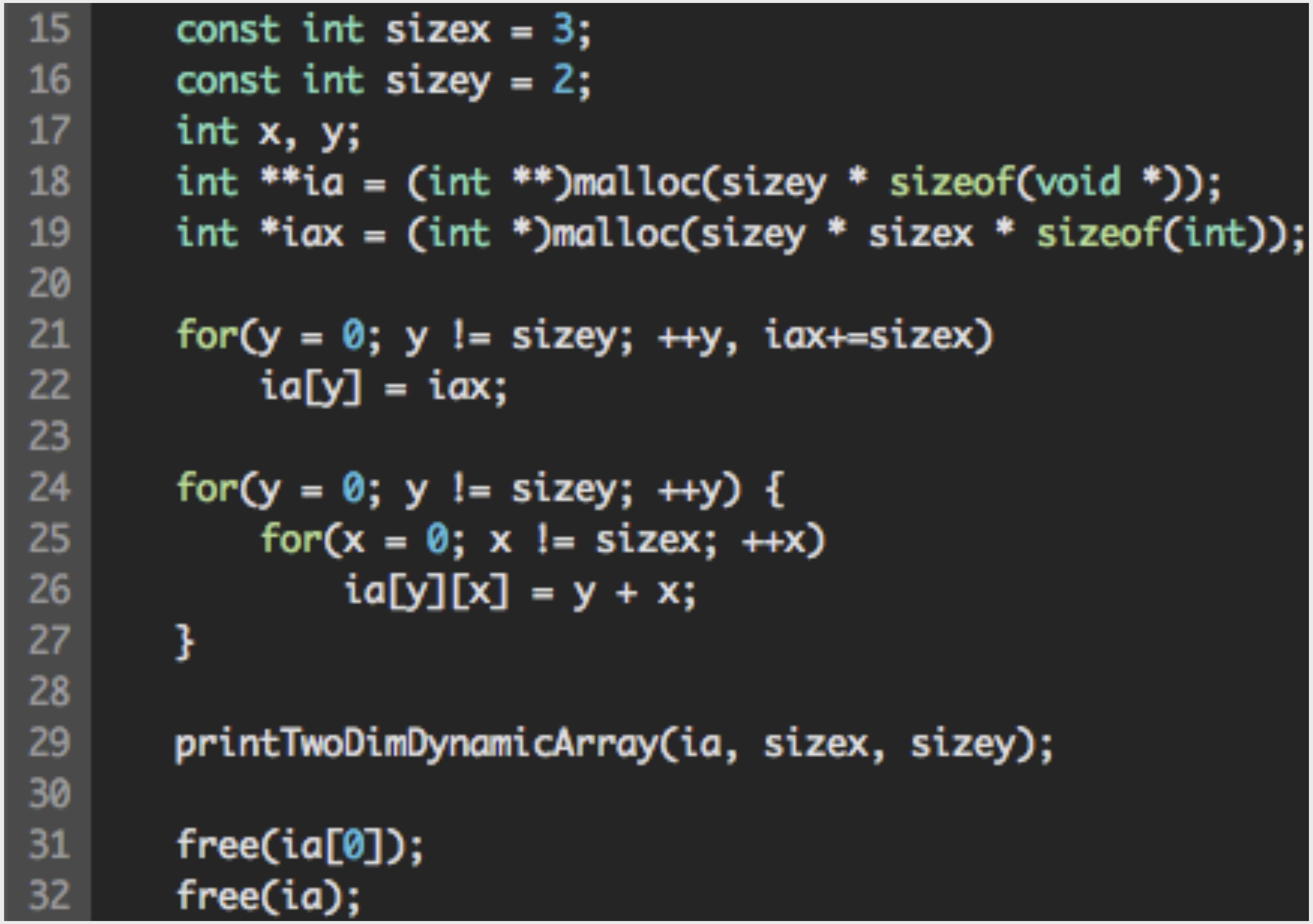
```
int *iax = (int *) malloc(sizey * sizex * sizeof(int));
```

- assign the internal arrays to the external array

```
for(y = 0; y != sizey; ++y, iax += sizex) {
  ia[y] = iax;
}
free(ia[0]);  /* free the internal arrays */
free(ia); /* free the external array */
```

malloc A 2D array (2)

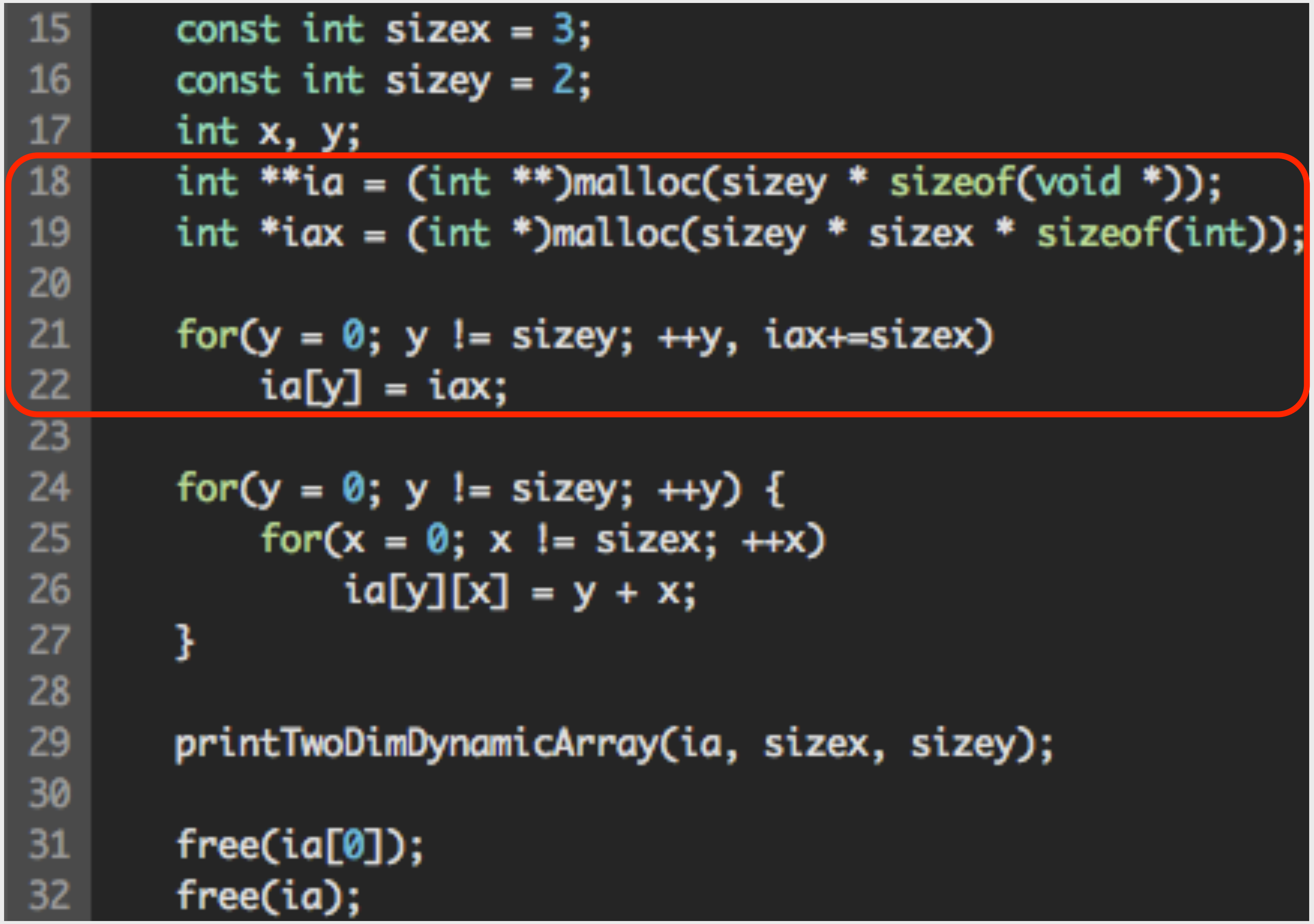
- Example: array_dynamic_two_dim_2.c



```c
const int sizex = 3;
const int sizey = 2;
int x, y;
int **ia = (int **)malloc(sizey * sizeof(void *));
int *iax = (int *)malloc(sizey * sizex * sizeof(int));

for(y = 0; y != sizey; ++y, iax+=sizex)
    ia[y] = iax;

for(y = 0; y != sizey; ++y) {
    for(x = 0; x != sizex; ++x)
        ia[y][x] = y + x;
}

printTwoDimDynamicArray(ia, sizex, sizey);

free(ia[0]);
free(ia);
```

`malloc` A 2D array (2)

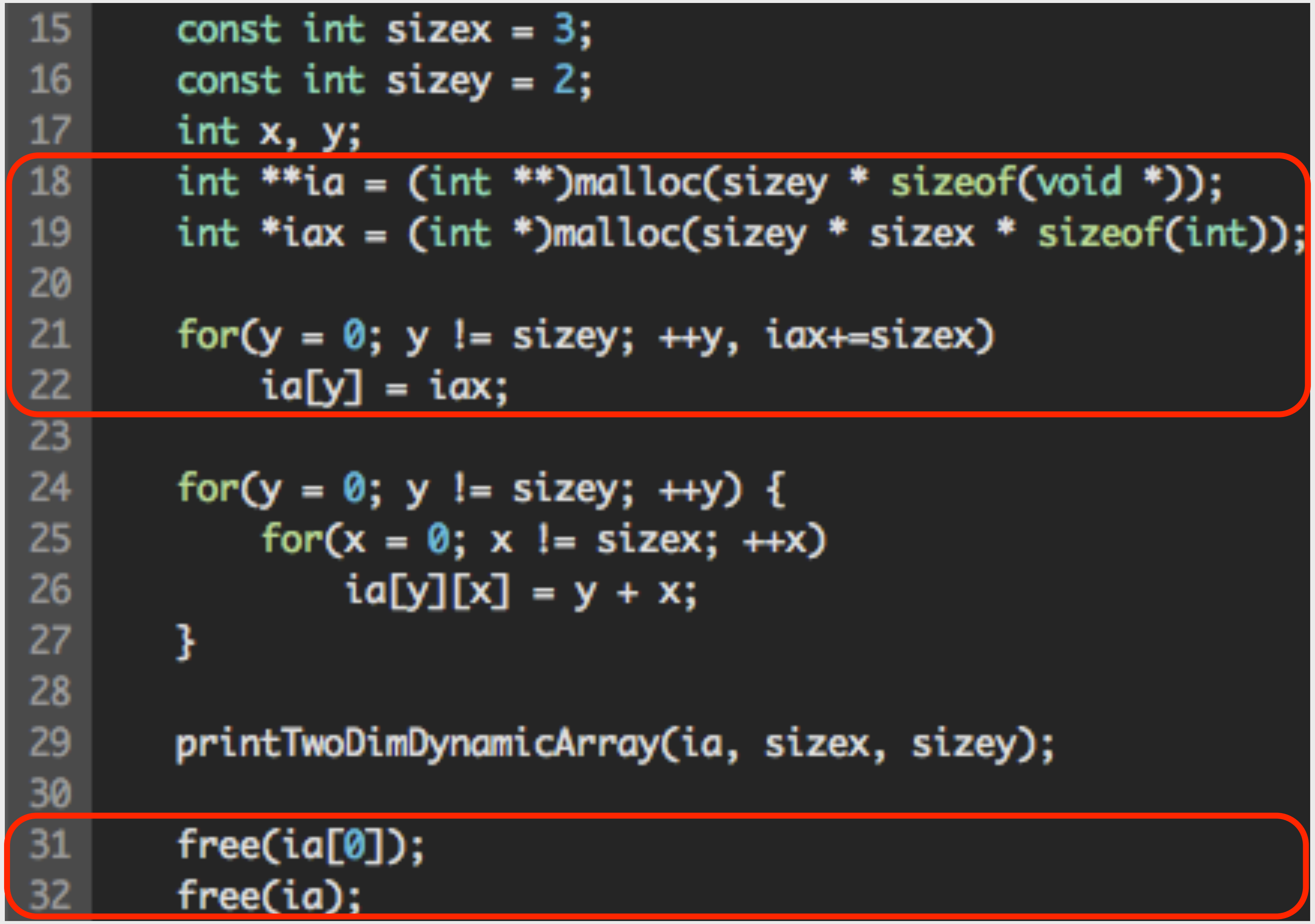
- Example: array_dynamic_two_dim_2.c



```c
const int sizex = 3;
const int sizey = 2;
int x, y;
int **ia = (int **)malloc(sizey * sizeof(void *));
int *iax = (int *)malloc(sizey * sizex * sizeof(int));

for(y = 0; y != sizey; ++y, iax+=sizex)
    ia[y] = iax;

for(y = 0; y != sizey; ++y) {
    for(x = 0; x != sizex; ++x)
        ia[y][x] = y + x;
}

printTwoDimDynamicArray(ia, sizex, sizey);

free(ia[0]);
free(ia);
```

`malloc` A 2D array (2)

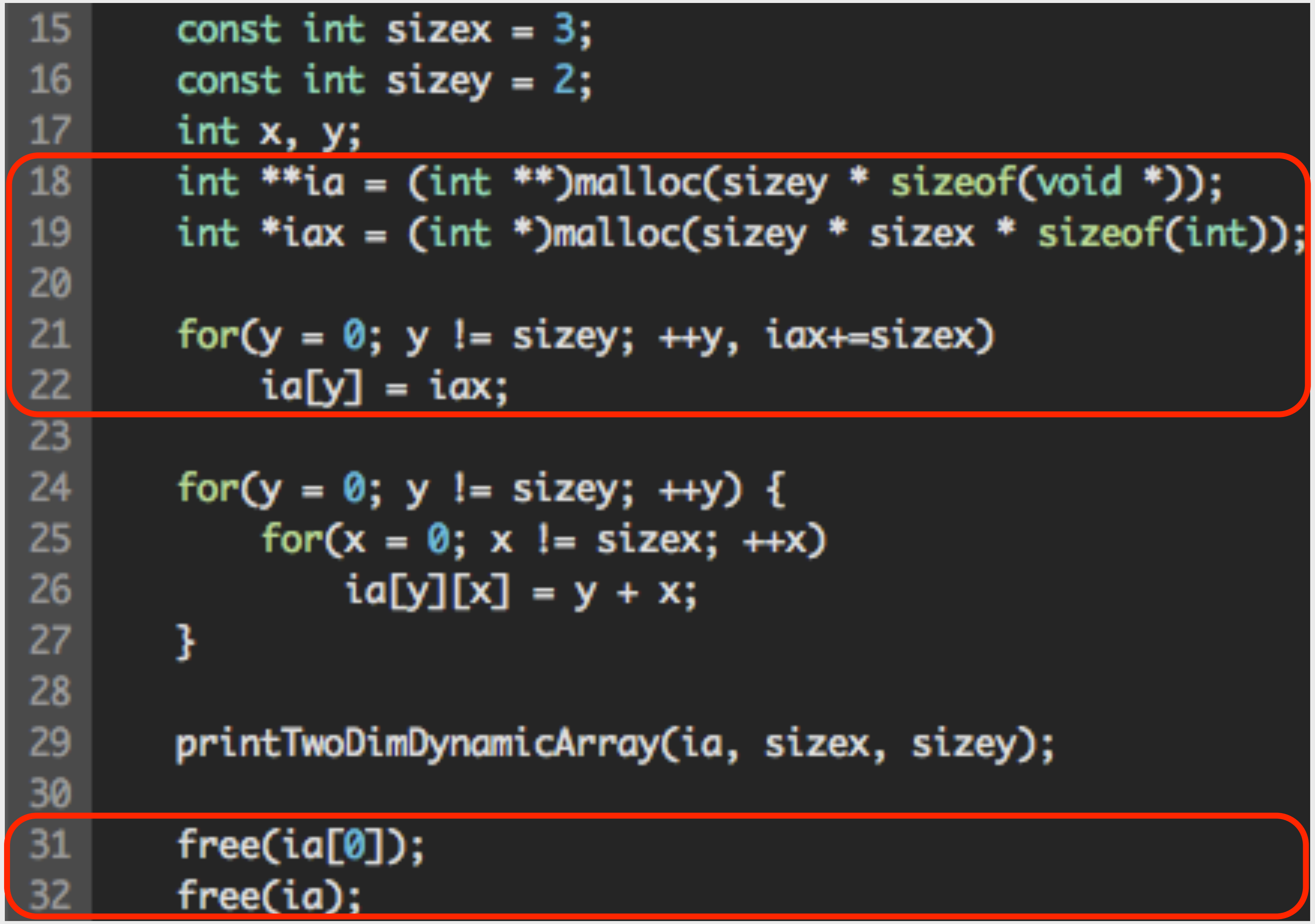
- Example: array_dynamic_two_dim_2.c



```
    const int sizex = 3;
    const int sizey = 2;
    int x, y;
    int **ia = (int **)malloc(sizey * sizeof(void *));
    int *iax = (int *)malloc(sizey * sizex * sizeof(int));

    for(y = 0; y != sizey; ++y, iax+=sizex)
        ia[y] = iax;

    for(y = 0; y != sizey; ++y) {
        for(x = 0; x != sizex; ++x)
            ia[y][x] = y + x;
    }

    printTwoDimDynamicArray(ia, sizex, sizey);

    free(ia[0]);
    free(ia);
```

malloc A 2D array (2)

- Example: array_dynamic_two_dim_2.c



```
    const int sizex = 3;
    const int sizey = 2;
    int x, y;
    int **ia = (int **)malloc(sizey * sizeof(void *));
    int *iax = (int *)malloc(sizey * sizex * sizeof(int));

    for(y = 0; y != sizey; ++y, iax+=sizex)
        ia[y] = iax;

    for(y = 0; y != sizey; ++y) {
        for(x = 0; x != sizex; ++x)
            ia[y][x] = y + x;
    }

    printTwoDimDynamicArray(ia, sizex, sizey);

    free(ia[0]);
    free(ia);
```

- Problem
  - Still need to free twice

`malloc` A 2D array (3)

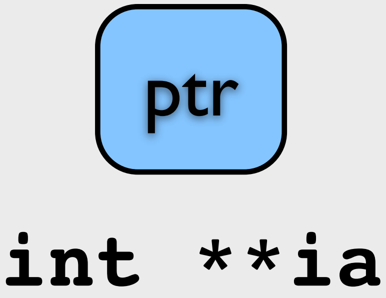
- Illustration

`malloc` A 2D array (3)

- Illustration

`malloc` A 2D array (3)

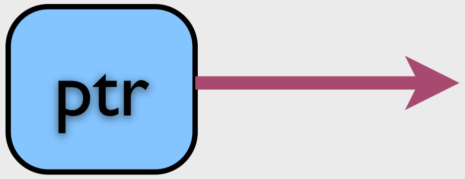
- Illustration



`int **ia`

`malloc` A 2D array (3)

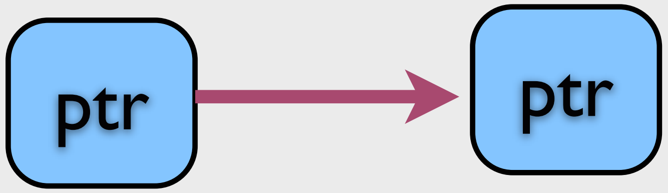
- Illustration



`int **ia`

`malloc` A 2D array (3)

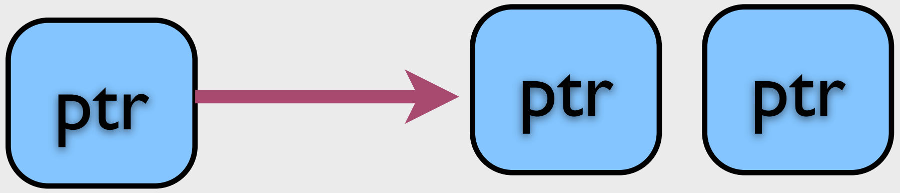
- Illustration



`int **ia`

`malloc` A 2D array (3)

- Illustration



ptr

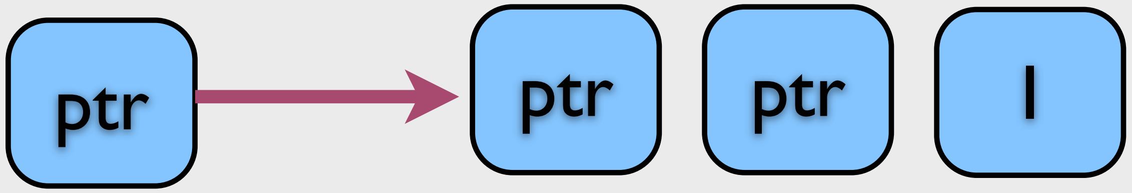
ptr

ptr

`int **ia`

`malloc` A 2D array (3)

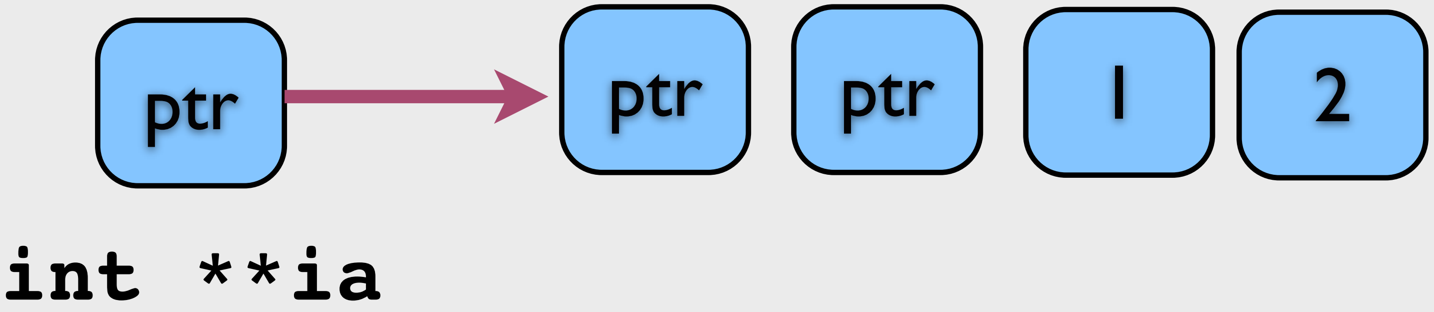
- Illustration



`int **ia`

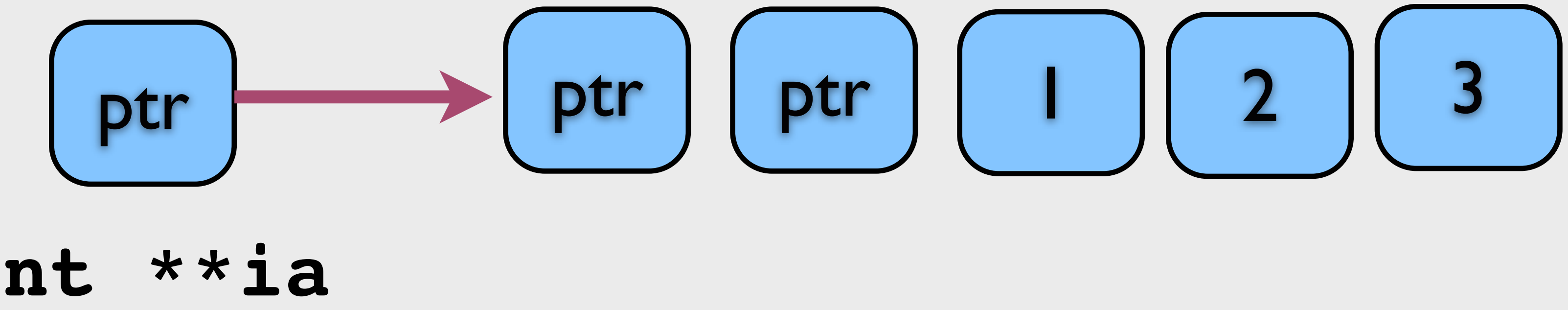
`malloc` A 2D array (3)

- Illustration



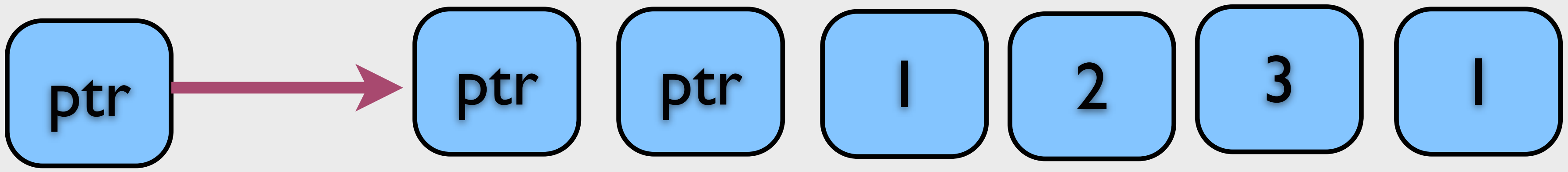
malloc A 2D array (3)

- Illustration



`malloc` A 2D array (3)

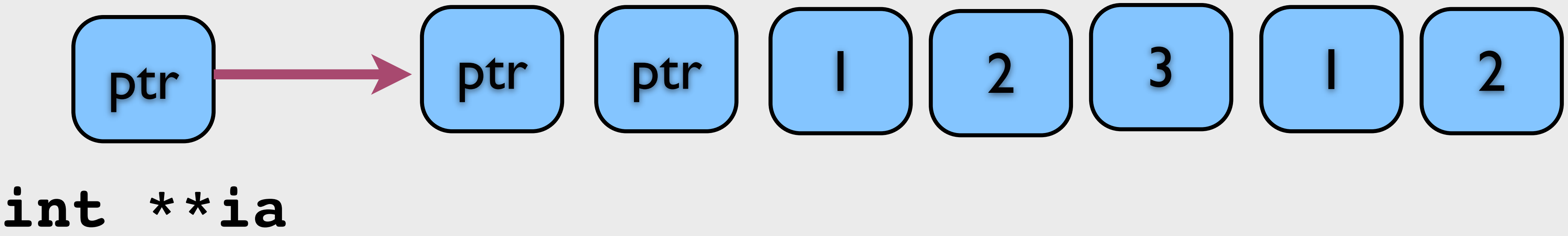
- Illustration



`int **ia`

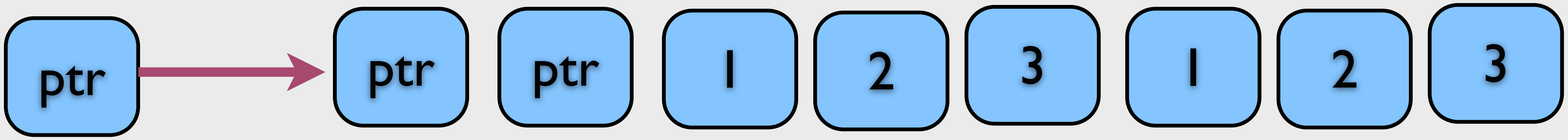
`malloc` A 2D array (3)

- Illustration



`malloc` A 2D array (3)

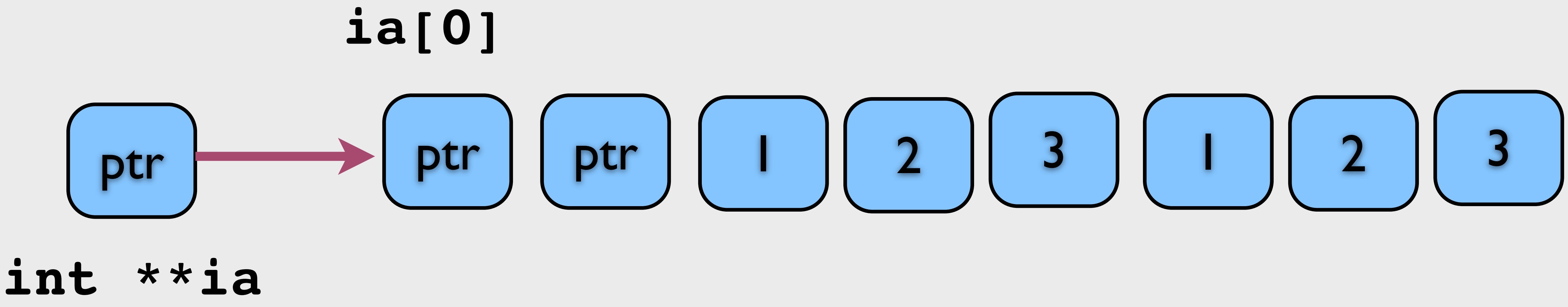
- Illustration



`int **ia`

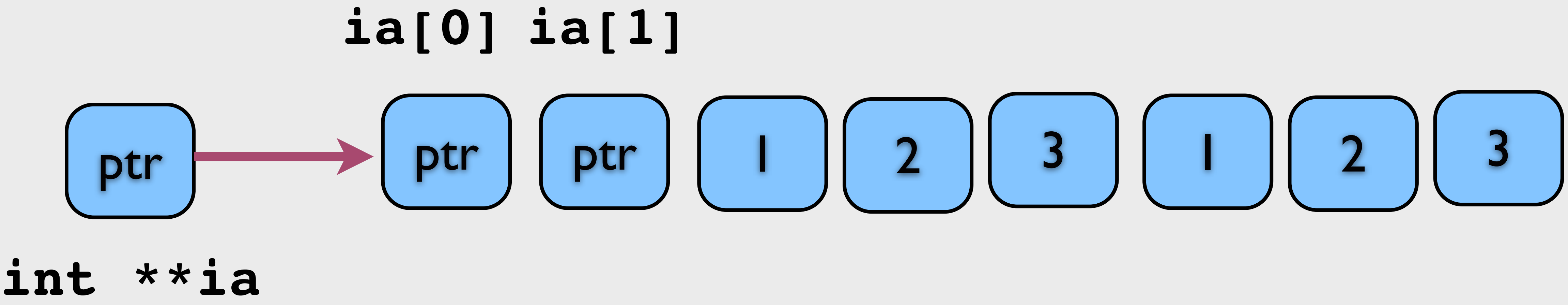
`malloc` A 2D array (3)

- Illustration



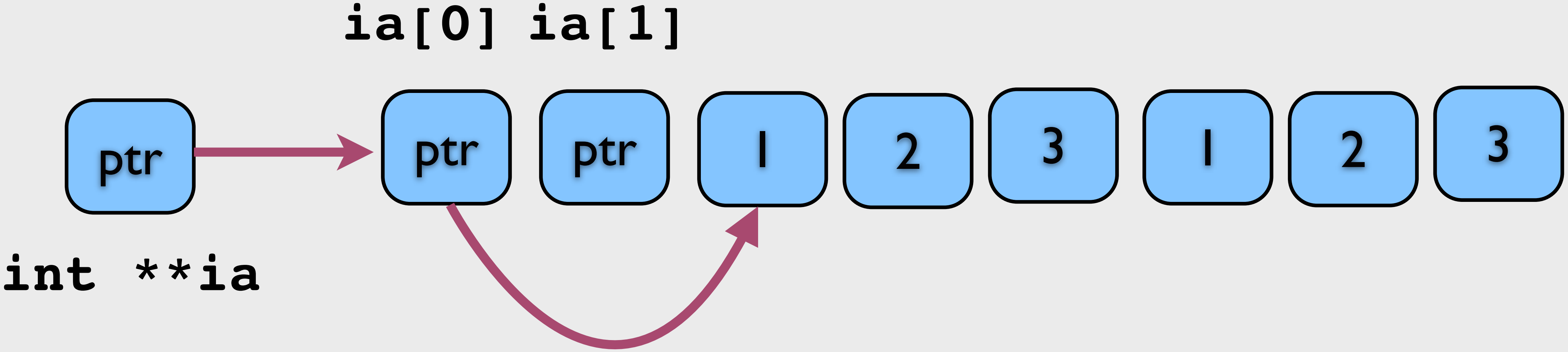
`malloc` A 2D array (3)

- Illustration



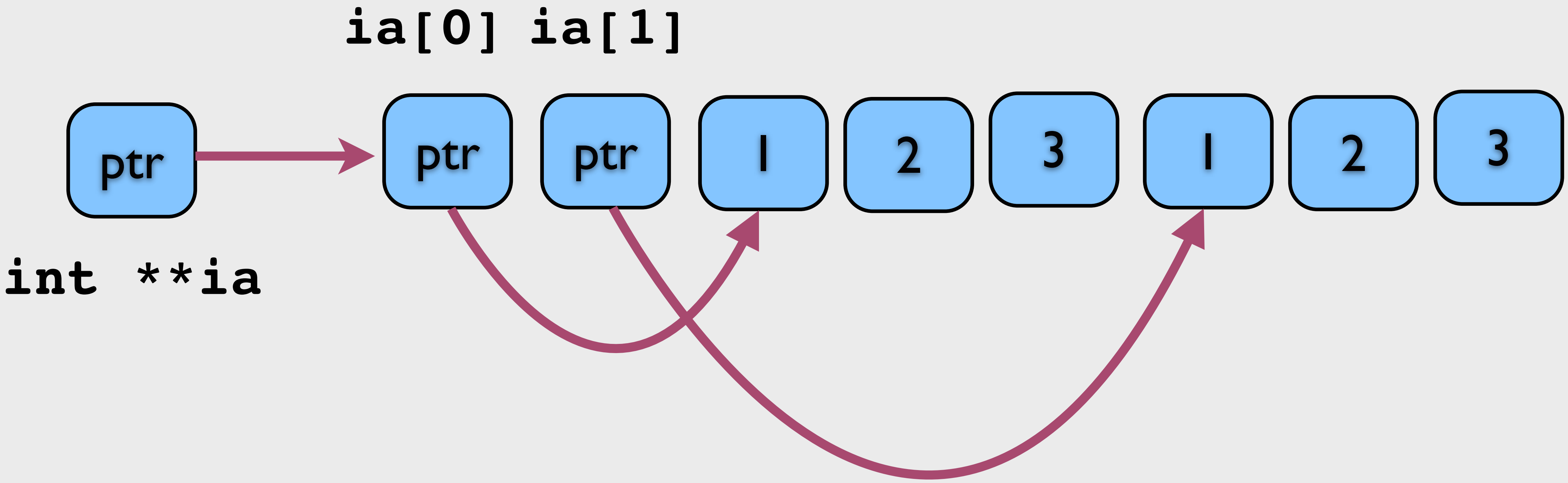
`malloc` A 2D array (3)

- Illustration



`malloc` A 2D array (3)

- Illustration



`malloc` A 2D array (3)

`malloc` A 2D array (3)

- malloc once

```
int **ia = (int **)malloc(sizey * sizeof(void
*) + sizey * sizex * sizeof(int));
```

`malloc` A 2D array (3)

- malloc once

```
int **ia = (int **)malloc(sizey * sizeof(void
*) + sizey * sizex * sizeof(int));
```

- assign the internal arrays to the external array

malloc A 2D array (3)

- malloc once

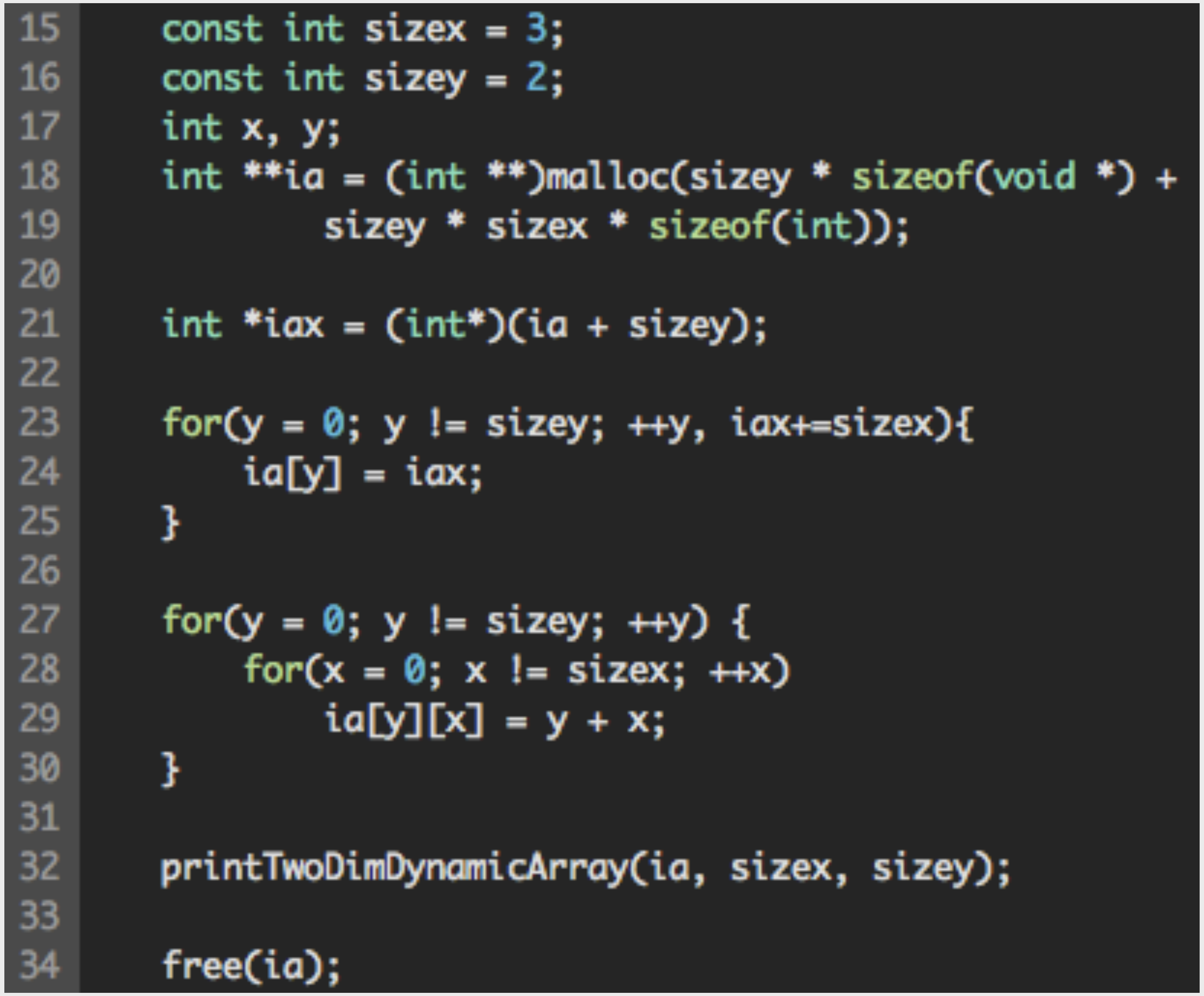
```
int **ia = (int **)malloc(sizey * sizeof(void
*) + sizey * sizex * sizeof(int));
```

- assign the internal arrays to the external array

```
int *iax = (int*) (ia+sizey)

for(y = 0; y != sizey; ++y, iax += sizex) {

   ia[y] = iax;

}

free(ia); /* free ia only */
```

`malloc` A 2D array (3)

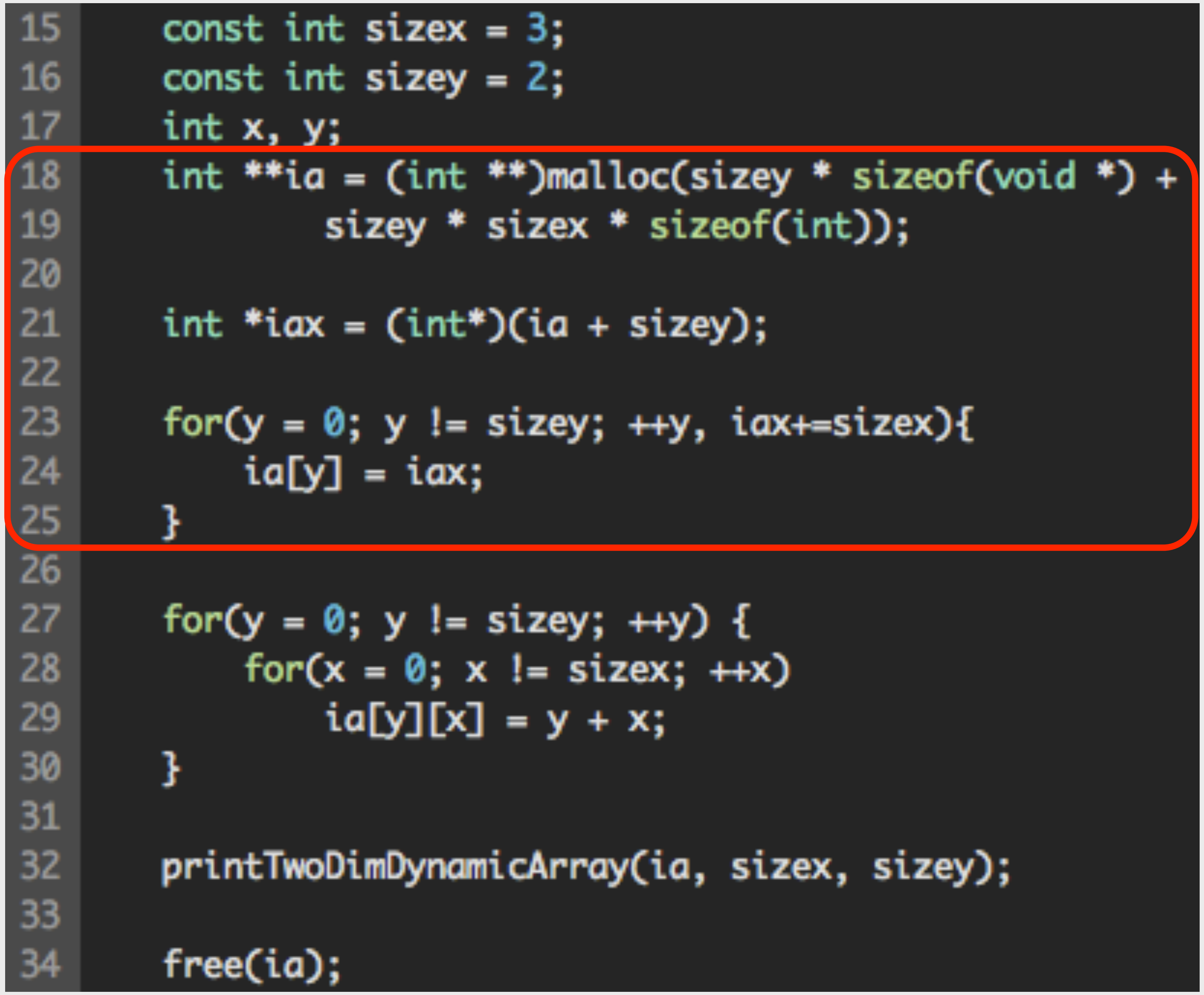
- Example: array_dynamic_two_dim_3.c



```c
const int sizex = 3;
const int sizey = 2;
int x, y;
int **ia = (int **)malloc(sizey * sizeof(void *) +
        sizey * sizex * sizeof(int));

int *iax = (int*)(ia + sizey);

for(y = 0; y != sizey; ++y, iax+=sizex){
    ia[y] = iax;
}

for(y = 0; y != sizey; ++y) {
    for(x = 0; x != sizex; ++x)
        ia[y][x] = y + x;
}

printTwoDimDynamicArray(ia, sizex, sizey);

free(ia);
```

`malloc` A 2D array (3)

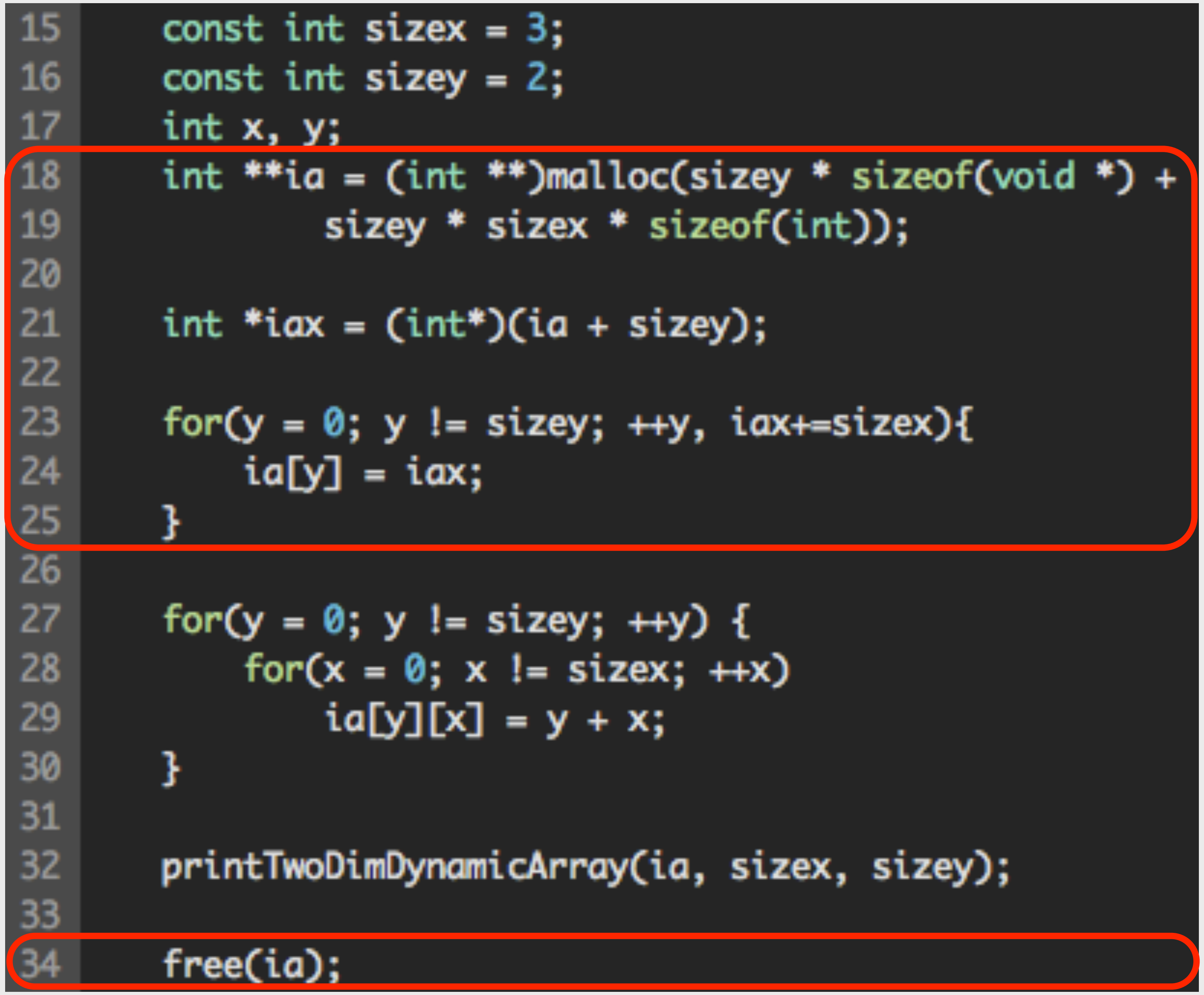
- Example: array_dynamic_two_dim_3.c



```c
    const int sizex = 3;
    const int sizey = 2;
    int x, y;
    int **ia = (int **)malloc(sizey * sizeof(void *) +
            sizey * sizex * sizeof(int));

    int *iax = (int*)(ia + sizey);

    for(y = 0; y != sizey; ++y, iax+=sizex){
        ia[y] = iax;
    }

    for(y = 0; y != sizey; ++y) {
        for(x = 0; x != sizex; ++x)
            ia[y][x] = y + x;
    }

    printTwoDimDynamicArray(ia, sizex, sizey);

    free(ia);
```

`malloc` A 2D array (3)

- Example: array_dynamic_two_dim_3.c



```
    const int sizex = 3;
    const int sizey = 2;
    int x, y;
    int **ia = (int **)malloc(sizey * sizeof(void *) +
            sizey * sizex * sizeof(int));

    int *iax = (int*)(ia + sizey);

    for(y = 0; y != sizey; ++y, iax+=sizex){
        ia[y] = iax;
    }

    for(y = 0; y != sizey; ++y) {
        for(x = 0; x != sizex; ++x)
            ia[y][x] = y + x;
    }

    printTwoDimDynamicArray(ia, sizex, sizey);

    free(ia);
```