Pointers

COMP2017/COMP9017

Dr. John Stavrakakis

FACULTY OF ENGINEERING

THE UNIVERSITY OF SYDNEY

Simple variables

- C has a number of simple types
  - float, int, char etc
  - each implies an interpretation of the bit pattern stored in the memory.

- *Declarations* label and reserve memory:

```
int     counter;
```

reserve memory for an integer and call it “counter”

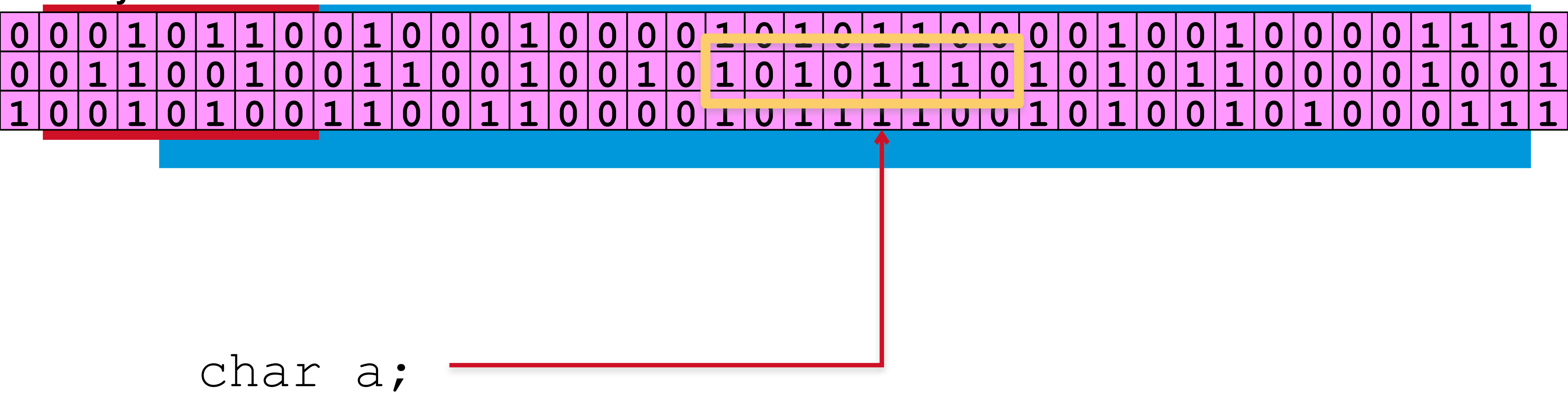
- *Initialisation* or *assignment* specifies content:

```
int   counter = 0;
counter = 0;
```

Memory

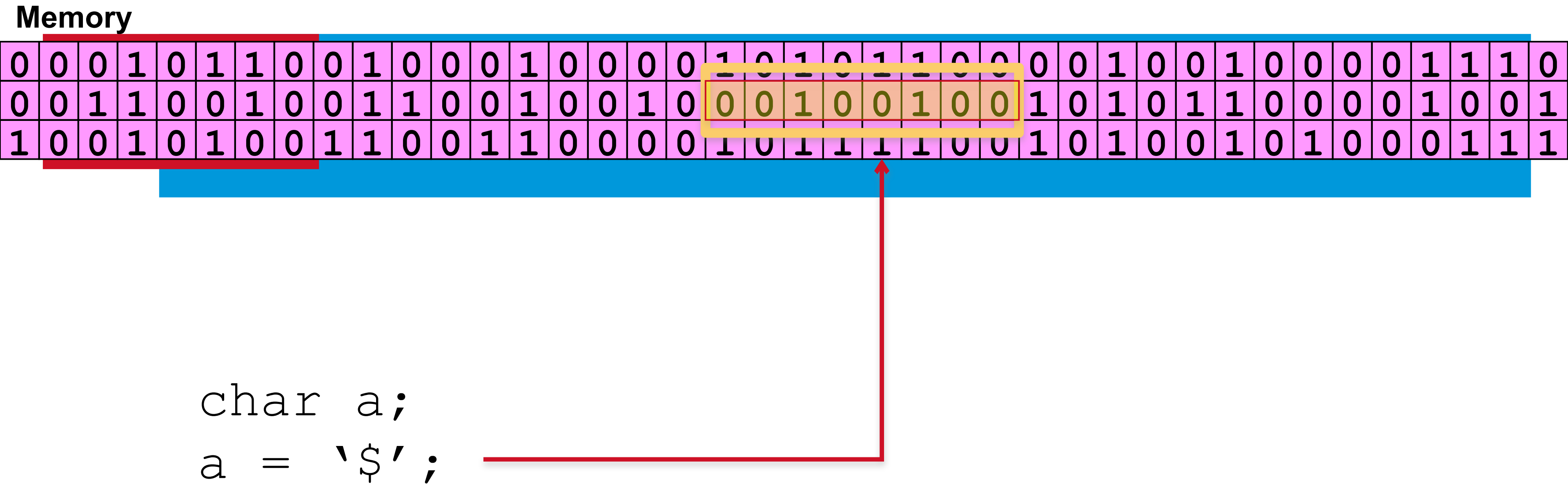
0	0	0	1	0	1	1	0	0	1	0	0	0	1	0	0	0	0	1	0	1	0	1	1	0	0	0	0	1	0	0	1	0	0	0	0	1	1	1	0
0	0	1	1	0	0	1	0	0	1	1	0	0	1	0	0	1	0	1	0	1	0	1	1	1	0	1	0	1	0	1	1	0	0	0	0	1	0	0	1
1	0	0	1	0	1	0	0	1	1	0	0	1	1	0	0	0	0	1	0	1	1	1	1	0	0	1	0	1	0	0	1	0	1	0	0	0	1	1	1

Memory



0	0	0	1	0	1	1	0	0	1	0	0	0	1	0	0	0	0	1	0	1	0	1	1	0	0	0	0	1	0	0	1	0	0	0	0	1	1	1	0
0	0	1	1	0	0	1	0	0	1	1	0	0	1	0	0	1	0	1	0	1	0	1	1	1	0	1	0	1	0	1	1	0	0	0	0	1	0	0	1
1	0	0	1	0	1	0	0	1	1	0	0	1	1	0	0	0	0	1	0	1	1	1	1	0	0	1	0	1	0	0	1	0	1	0	0	0	1	1	1

```
char a;
```



```
0 0 0 1 0 1 1 0 0 1 0 0 0 1 0 0 0 0 1 0 1 0 1 1 0 0 0 0 1 0 0 1 0 0 0 0 1 1 1 0
0 0 1 1 0 0 1 0 0 1 1 0 0 1 0 0 1 0 0 0 1 0 0 1 0 0 1 0 1 0 1 1 0 0 0 0 1 0 0 1
1 0 0 1 0 1 0 0 1 1 0 0 1 1 0 0 0 0 1 0 1 1 1 1 0 0 1 0 1 0 0 1 0 1 0 0 0 1 1 1
```

```
char a;
a = ‘$’;
```

Arrays

- Arrays are indexed collections of the same type

- Declaration of an array:

```
int     counters[MAX];
char    alphabet[26];
```

- Initialisation of an array:

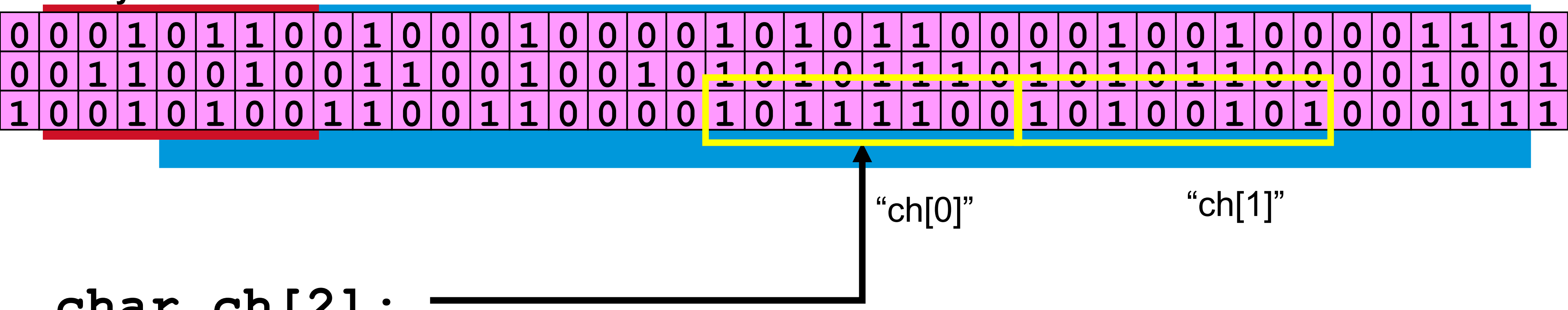
```
for (i = 0; i < MAX; i++)

      counters[i] = i;
```

Memory

0	0	0	1	0	1	1	0	0	1	0	0	0	1	0	0	0	0	1	0	1	0	1	1	0	0	0	0	1	0	0	1	0	0	0	0	1	1	1	0
0	0	1	1	0	0	1	0	0	1	1	0	0	1	0	0	1	0	1	0	1	0	1	1	1	0	1	0	1	0	1	1	0	0	0	0	1	0	0	1
1	0	0	1	0	1	0	0	1	1	0	0	1	1	0	0	0	0	1	0	1	1	1	1	0	0	1	0	1	0	0	1	0	1	0	0	0	1	1	1

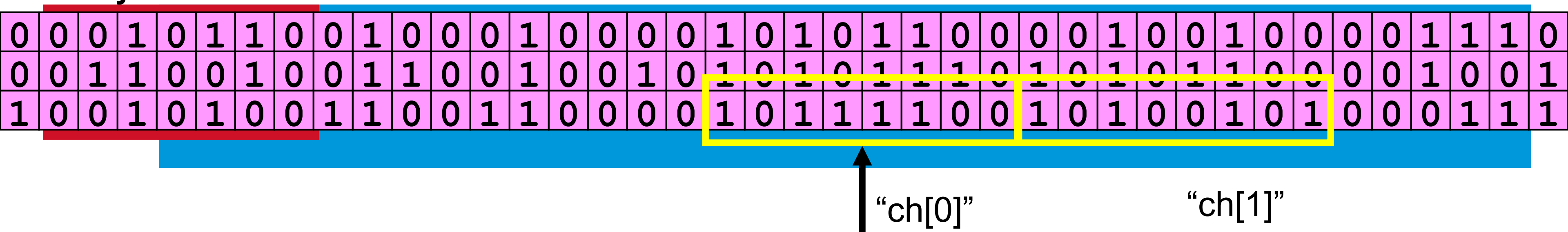
Memory



```
0 0 0 1 0 1 1 0 0 1 0 0 0 1 0 0 0 0 1 0 1 0 1 1 0 0 0 0 1 0 0 1 0 0 0 0 1 1 1 0
0 0 1 1 0 0 1 0 0 1 1 0 0 1 0 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 1 0 0 0 0 1 0 0 1
1 0 0 1 0 1 0 0 1 1 0 0 1 1 0 0 0 0 1 0 1 1 1 1 0 0 1 0 1 0 0 1 0 1 0 0 0 1 1 1
```

```
char ch[2];
```

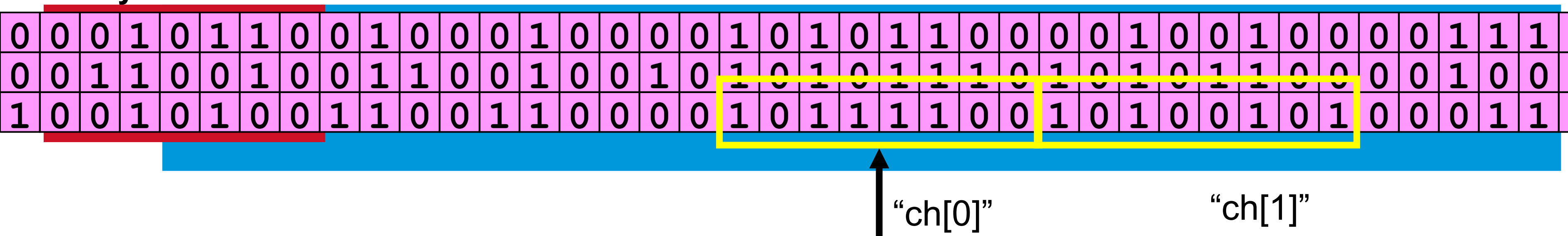
Memory



```
0 0 0 1 0 1 1 0 0 1 0 0 0 1 0 0 0 0 1 0 1 0 1 1 0 0 0 0 1 0 0 1 0 0 0 0 1 1 1 0
0 0 1 1 0 0 1 0 0 1 1 0 0 1 0 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 1 0 0 0 0 1 0 0 1
1 0 0 1 0 1 0 0 1 1 0 0 1 1 0 0 0 0 1 0 1 1 1 1 0 0 1 0 1 0 0 1 0 1 0 0 0 1 1 1
```

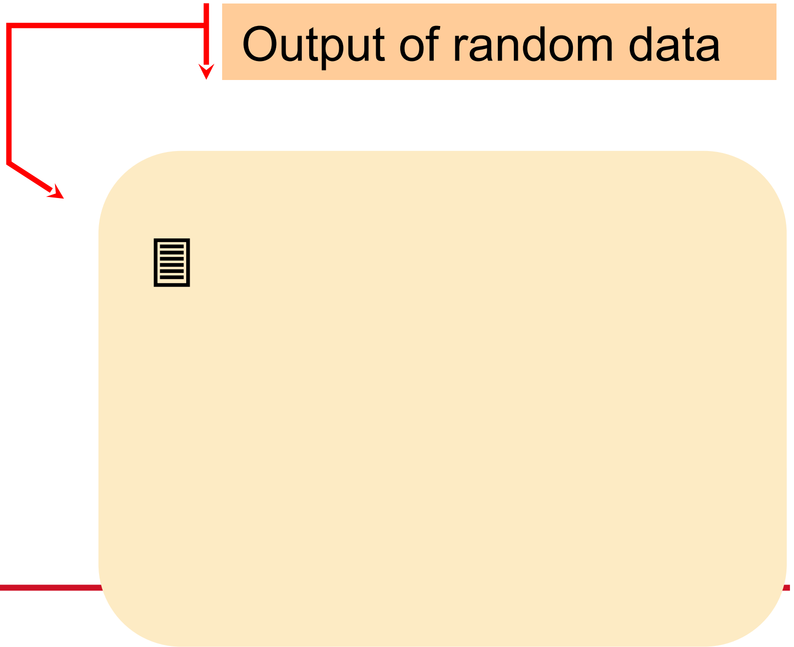
```
char ch[2];
printf(”%c\n”, ch[1]);
```

Memory



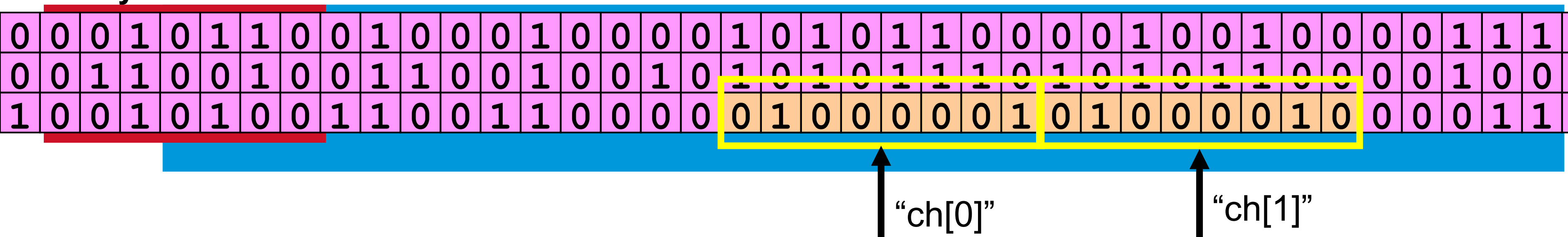
```
0 0 0 1 0 1 1 0 0 1 0 0 0 1 0 0 0 0 1 0 1 0 1 1 0 0 0 0 1 0 0 1 0 0 0 0 1 1 1 0
0 0 1 1 0 0 1 0 0 1 1 0 0 1 0 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 1 0 0 0 0 1 0 0 1
1 0 0 1 0 1 0 0 1 1 0 0 1 1 0 0 0 0 1 0 1 1 1 1 0 0 1 0 1 0 0 1 0 1 0 0 0 1 1 1
```

```
char ch[2];
printf(”%c\n”, ch[1]);
```



Output of random data

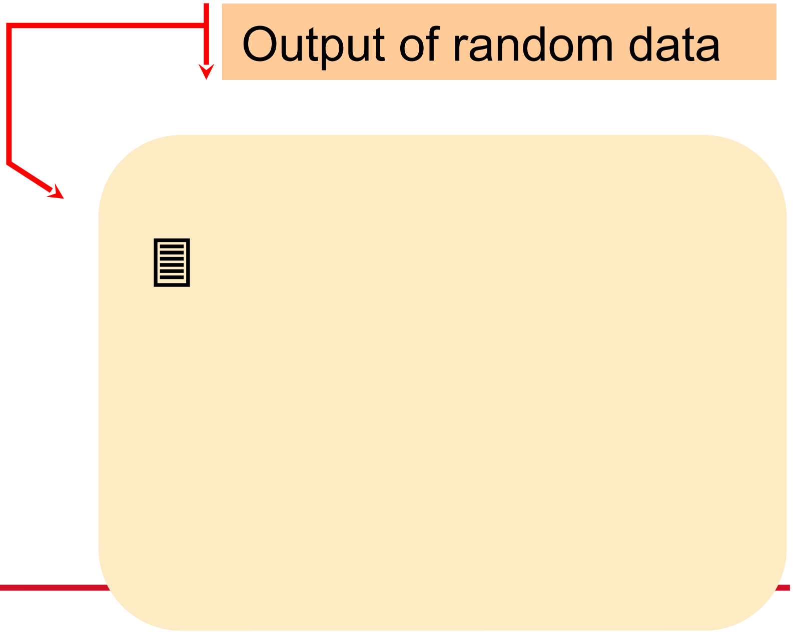
Memory

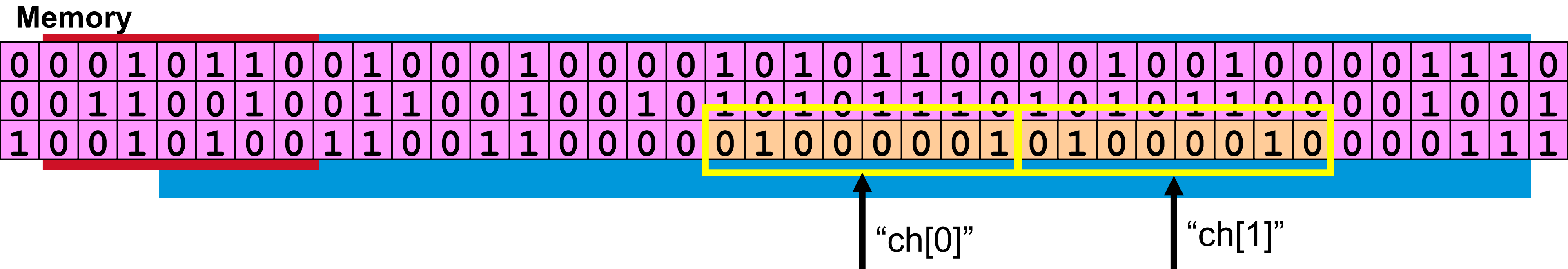


```
0 0 0 1 0 1 1 0 0 1 0 0 0 1 0 0 0 0 1 0 1 0 1 1 0 0 0 0 1 0 0 1 0 0 0 0 1 1 1 0
0 0 1 1 0 0 1 0 0 1 1 0 0 1 0 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 1 0 0 0 0 1 0 0 1
1 0 0 1 0 1 0 0 1 1 0 0 1 1 0 0 0 0 0 1 0 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 0 1 1 1
```

```
char ch[2];
printf(”%c\n”, ch[1]);
ch[0] = ’A’;
ch[1] = ’B’;
```

Output of random data





```
0 0 0 1 0 1 1 0 0 1 0 0 0 1 0 0 0 0 1 0 1 0 1 1 0 0 0 0 1 0 0 1 0 0 0 0 1 1 1 0
0 0 1 1 0 0 1 0 0 1 1 0 0 1 0 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 1 0 0 0 0 1 0 0 1
1 0 0 1 0 1 0 0 1 1 0 0 1 1 0 0 0 0 0 1 0 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 0 1 1 1
```

```
char ch[2];
printf(″%c\n″, ch[1]);
ch[0] = ′A′;
ch[1] = ′B′;
printf(″%c%c\n″, ch[0], ch[1]);
```

Output of random data

Output of initialised data

```
▤
AB
```

Strings

- Strings may be initialised at the time of declaration using an “array-like” notational convenience:

```
char myHobby[] = ″rowing″;
```

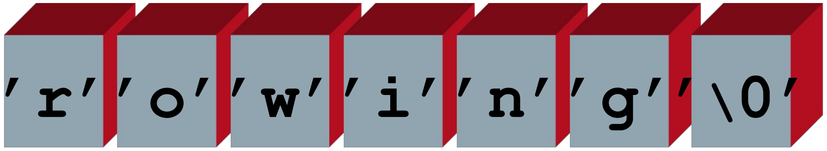
The compiler can determine the required size by counting characters, so the array size is **optional**. A larger size *may* be specified.

Strings

- Strings resemble an array of characters.
- However, in *C*, all strings are NULL-terminated.

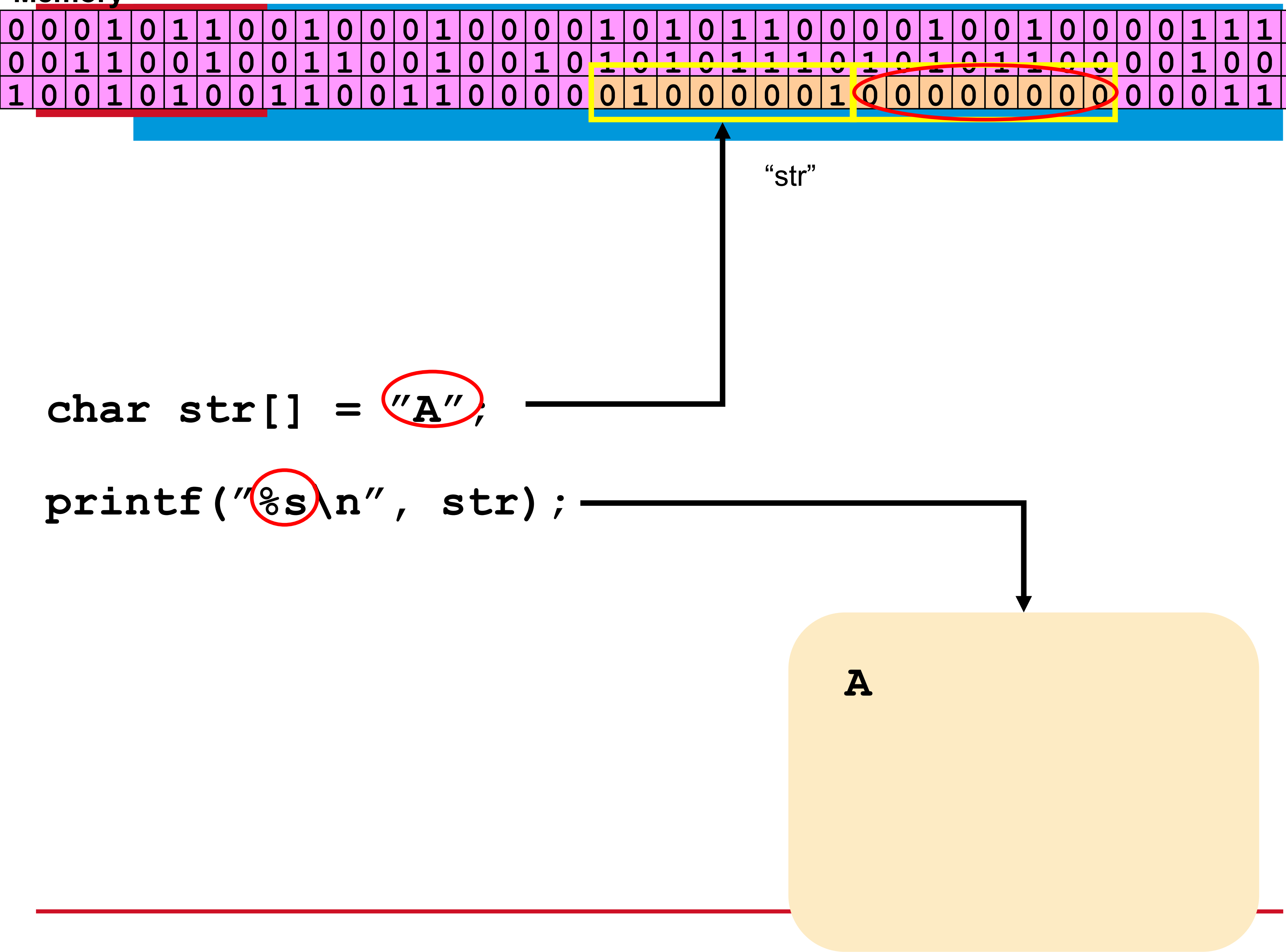
Note: NULL is the binary value 0 (denoted `‘\0’`), not the ASCII representation of the character 0.

```
char myHobby[] = ”rowing”;
```



’r’ ’o’ ’w’ ’i’ ’n’ ’g’ ’\0’

Memory



```
0 0 0 1 0 1 1 0 0 1 0 0 0 1 0 0 0 0 1 0 1 0 1 1 0 0 0 0 1 0 0 1 0 0 0 0 1 1 1 0
0 0 1 1 0 0 1 0 0 1 1 0 0 1 0 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 1 0 0 0 0 1 0 0 1
1 0 0 1 0 1 0 0 1 1 0 0 1 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1
```

“str”

```
char str[] = "A";

printf("%s\n", str);
```

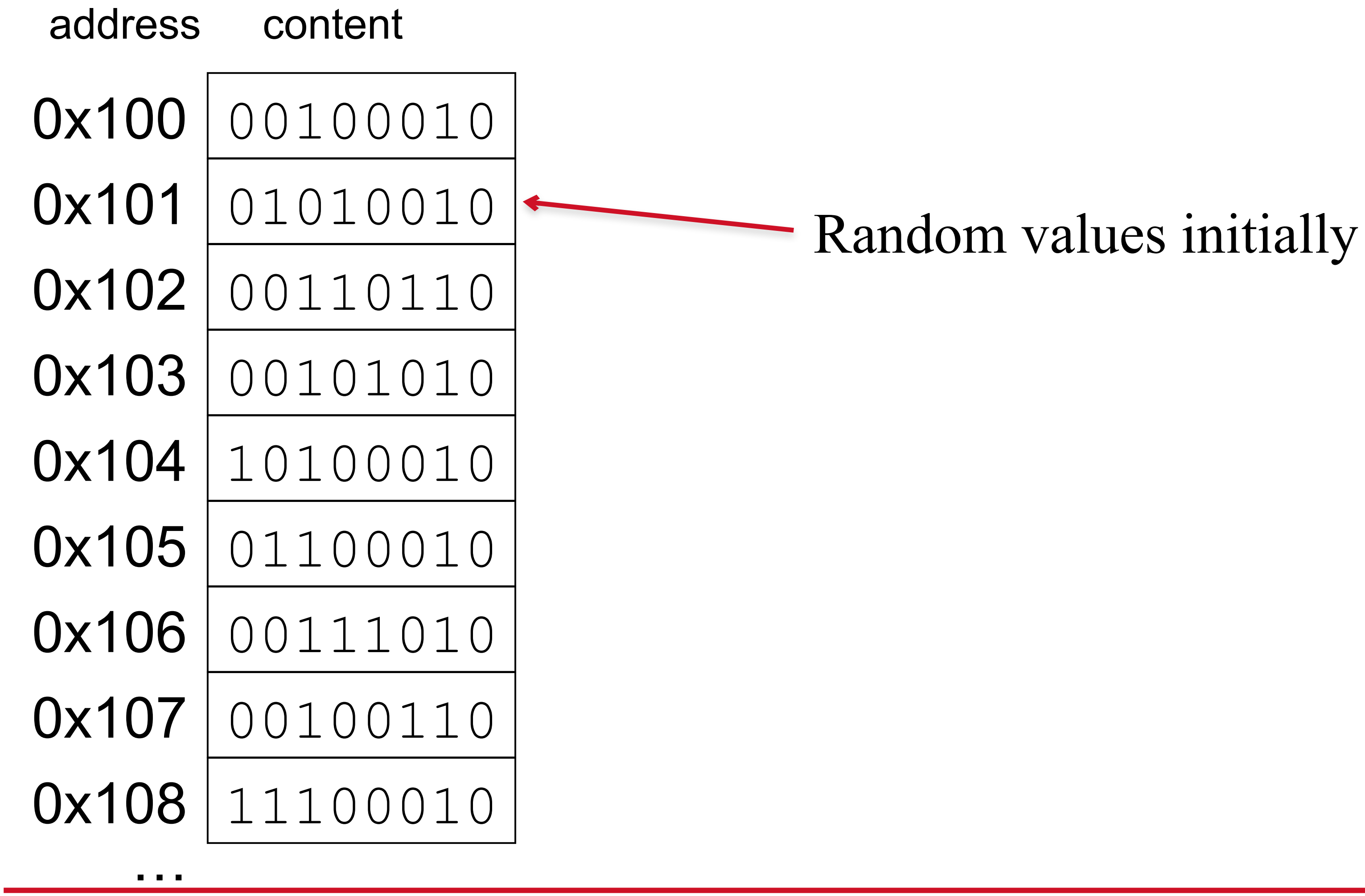
```
A
```

Memory

address	content
0x100	00100010
0x101	01010010
0x102	00110110
0x103	00101010
0x104	10100010
0x105	01100010
0x106	00111010
0x107	00100110
0x108	11100010

…

Memory



address	content
0x100	00100010
0x101	01010010
0x102	00110110
0x103	00101010
0x104	10100010
0x105	01100010
0x106	00111010
0x107	00100110
0x108	11100010
…	

Random values initially

Memory

address	content
0x100	00100010
0x101	01010010
0x102	00110110
0x103	00101010
0x104	10100010
0x105	01100010
0x106	00111010
0x107	00100110
0x108	11100010

…

- a ***pointer*** is essentially a memory address
- we can find out the address of a variable using the **&** operator

Memory

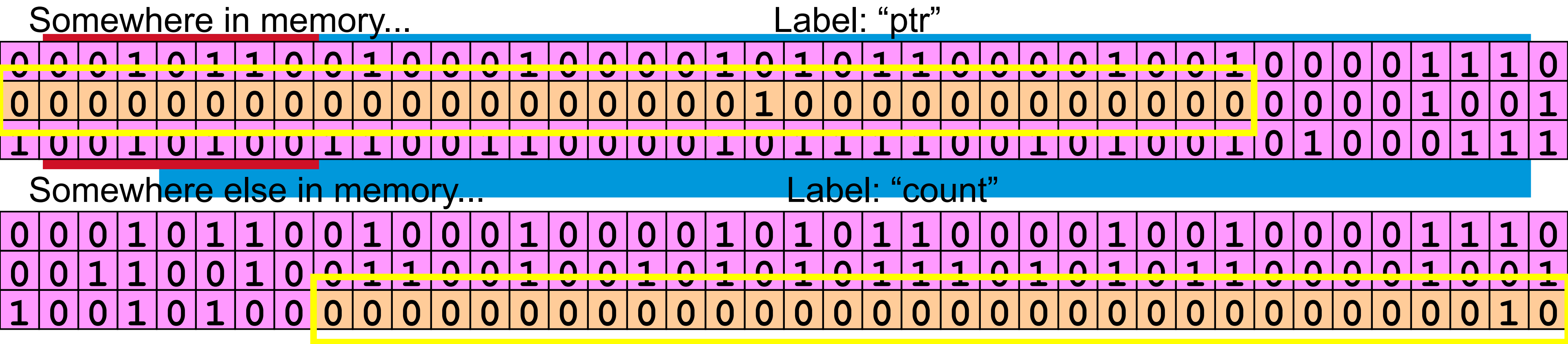
address	content
0x100	00100010
0x101	01010010
0x102	00110110
0x103	00101010
0x104	10100010
0x105	01100010
0x106	00111010
0x107	00100110
0x108	11100010
…	

```
char initial = ‘A’;

char * initp = &initial
```

`&initial` is the ***address of*** `initial`

`initp` is a ***pointer*** to `initial`



Somewhere in memory... Label: “ptr”

```
0 0 0 1 0 1 1 0 0 1 0 0 0 1 0 0 0 0 1 0 1 0 1 1 0 0 0 0 1 0 0 1 0 0 0 0 1 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1
1 0 0 1 0 1 0 0 1 1 0 0 1 1 0 0 0 0 1 0 1 1 1 1 0 0 1 0 1 0 0 1 0 1 0 0 0 1 1 1
```

Somewhere else in memory... Label: “count”

```
0 0 0 1 0 1 1 0 0 1 0 0 0 1 0 0 0 0 1 0 1 0 1 1 0 0 0 0 1 0 0 1 0 0 0 0 1 1 1 0
0 0 1 1 0 0 1 0 0 1 1 0 0 1 0 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 1 0 0 0 0 1 0 0 1
1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
```

```
int count;

int *ptr;

count = 2;

ptr = &count;

printf(”%d\n”, count);

printf(”%d\n”, *ptr);

printf(”%d\n”, &count);

printf(”%d\n”, ptr);
```

variable name: “count”
address of count: 0x1000 = 4,096

Clearly, the value of a pointer can *only* be determined at run-time.

```
2

2

4096

4096
```

Pointers (notation)

› Pointer operators:

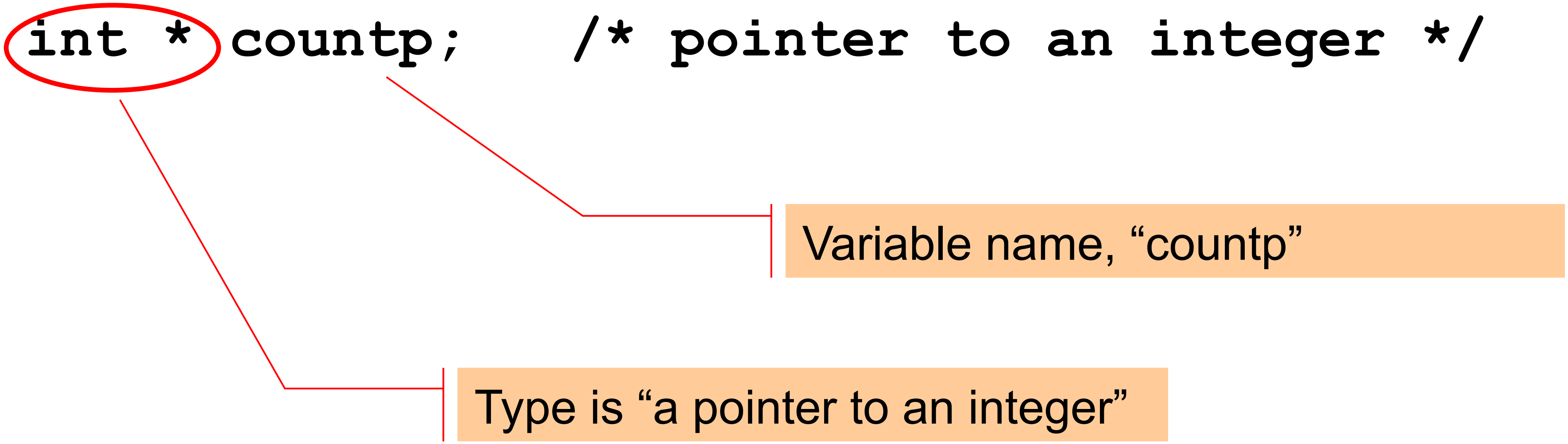
- address operator, ‘**&**’
- indirection operator, ‘*****’

Note that these operators are “overloaded”, that is they have more than one meaning.

- ‘**&**’ is also used in C as the bitwise ‘AND’ operator
- ‘*****’ is also used in C as the multiplication operator

Pointers (notation)

- The indirection operator, ‘*’, is used in a variable declaration to declare a “pointer to a variable of the specified type”:



```
int * countp;   /* pointer to an integer */
```

Variable name, “countp”

Type is “a pointer to an integer”

Pointers (notation)

What do the following mean?

```
float * amt;
```

```
int ** tricky;
```

Answers:

A pointer (labeled “amt”) to a *float*.

A pointer (labeled “tricky”) to a pointer to an *int*

THE UNIVERSITY OF SYDNEY

Pointers (notation)

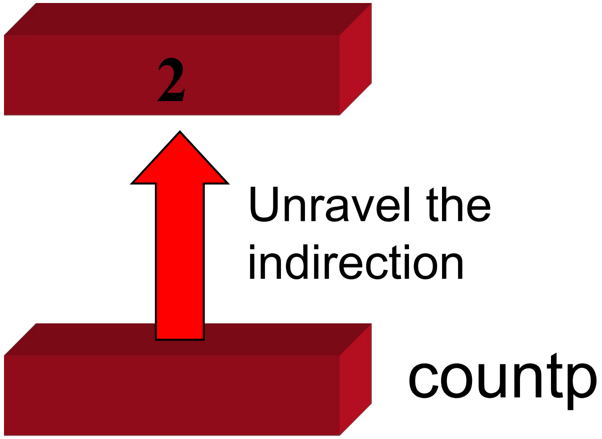
- The indirection operator, ‘`*`’, is used to “unravel” the indirection:

`countp` points to an integer variable that contains the value 2.

Then…

```
printf(”%d”, *countp);
```

…prints ‘2’ to standard output.



Pointers (notation)

What is output in the following?

```
printf(″%d″, count);
```

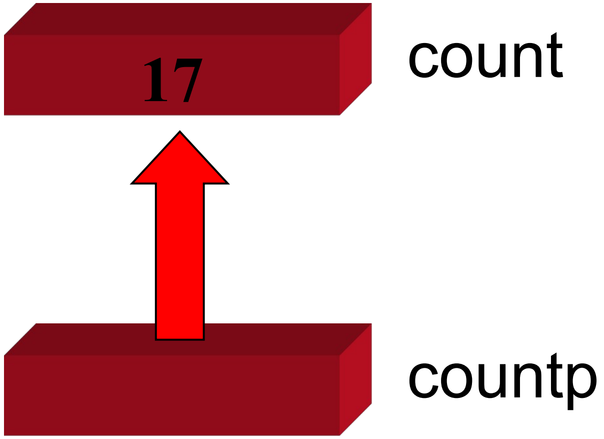
17

```
printf(″%d″, *countp);
```

17

```
printf(″%d″, countp);
```

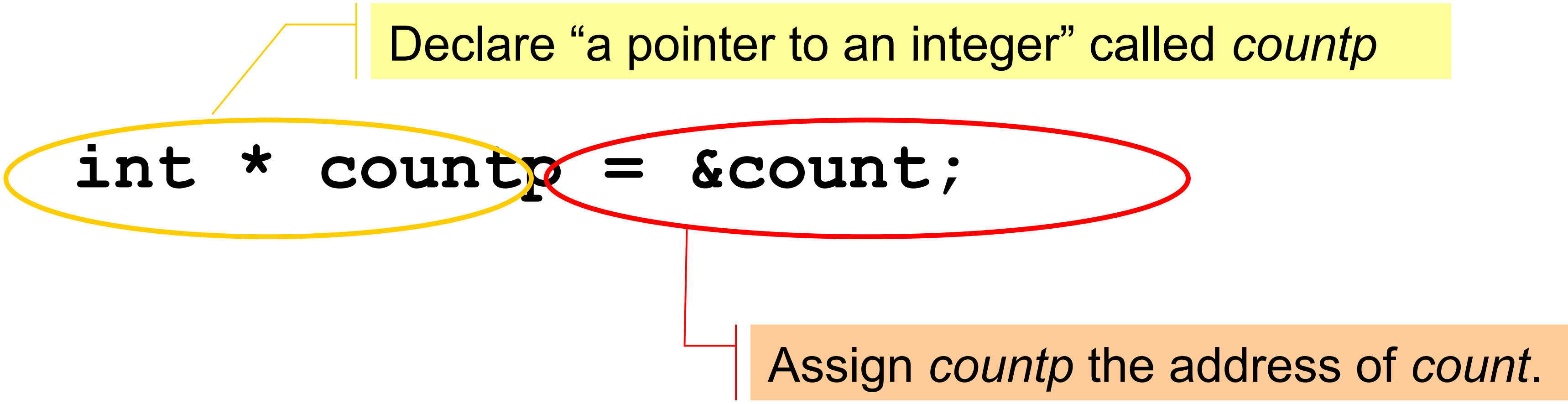
Don’t know… but it will be the <u>address</u> of *count*.



Pointers (notation)

- The address operator, ‘`&`’, is used to access the address of a variable.

- This completes the picture! A pointer can be assigned the address of a variable simply:



Declare “a pointer to an integer” called *countp*

```
int * countp = &count;
```

Assign *countp* the address of *count*.

Examples

An example of the the address operator in action...

Receiving an integer from standard input:

```
int age;
scanf(”%d”, &age);
```

This argument is required by *scanf()* to be a pointer. Since we are using a simple integer, *age*, we pass it’s address.

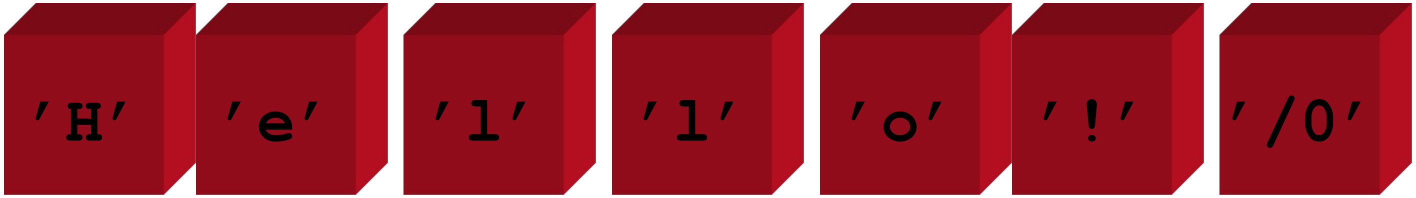
Pointers and arrays

Use of pointer notation to manipulate arrays...

```
char msg[] = ”Hello!”;
char *str = &msg[0];
```

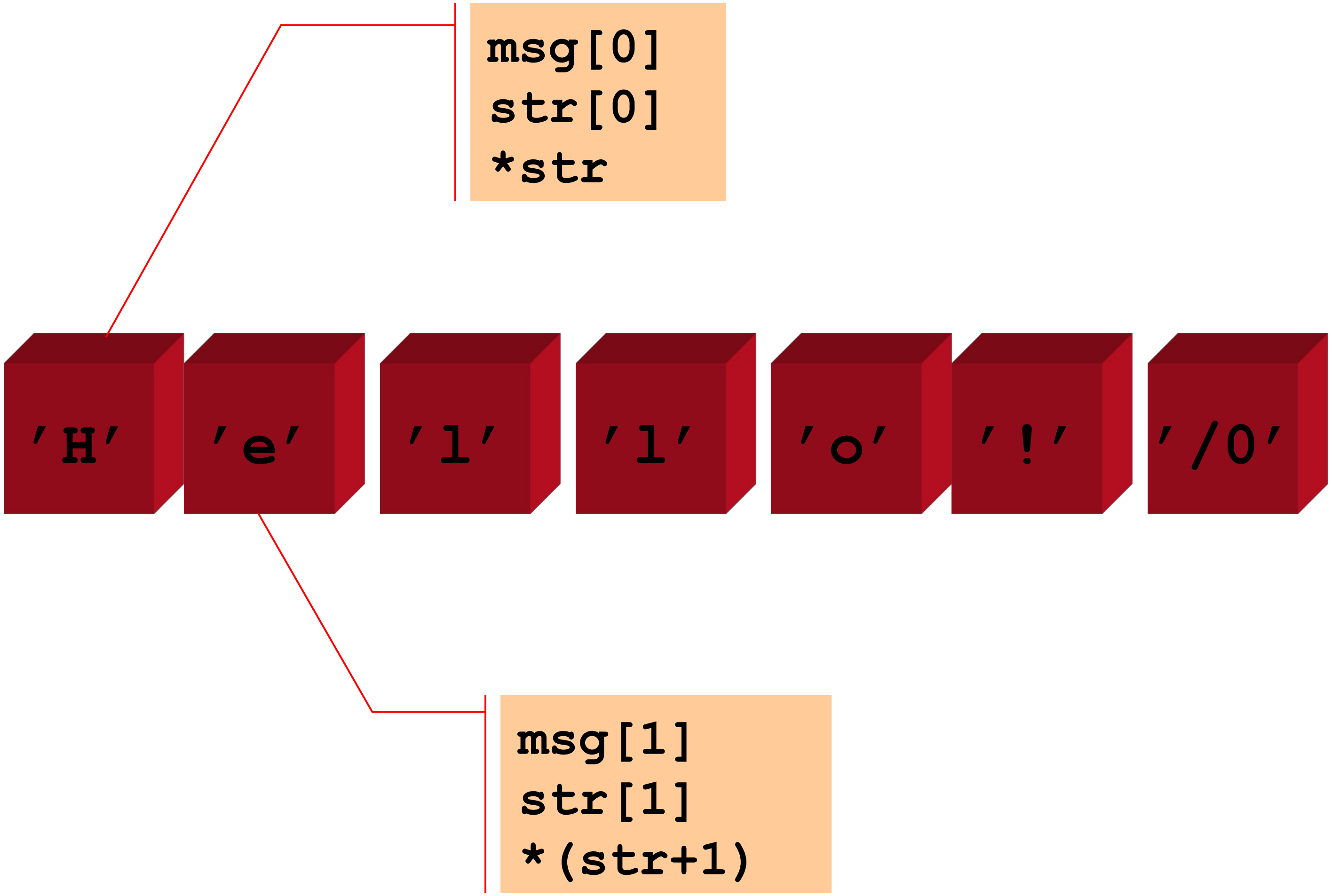
OR:

```
char *str = msg;
```



’H’ ’e’ ’l’ ’l’ ’o’ ’!’ ’/0’

Examples



```
msg[0]
str[0]
*str
```

'H' 'e' 'l' 'l' 'o' '!' '/0'

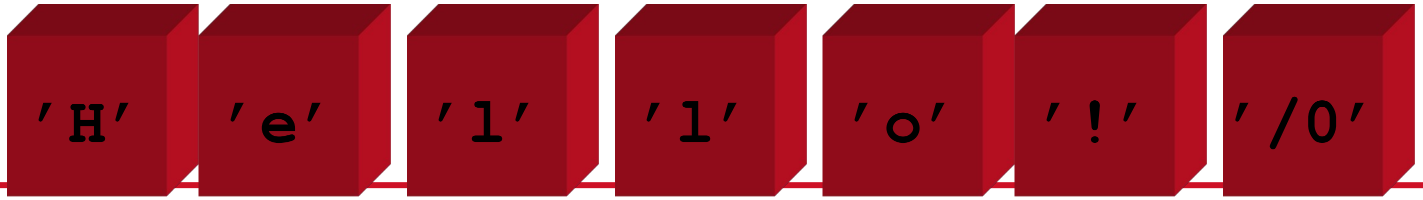
```
msg[1]
str[1]
*(str+1)
```

Examples

Pointer notation leads to some (intimidating?) shortcuts as part of the C idiom.

Moving through a string:

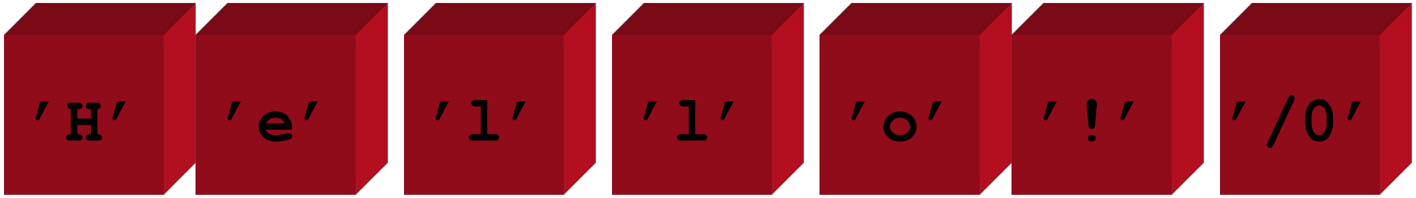
```
while (*str != ’\0’)
     str++;
```



Examples

The previous example may exploit the fact that C treats ‘0’ as FALSE:

```
while (*str)
     str++;
```



’H’ ’e’ ’l’ ’l’ ’o’ ’!’ ’/0’

Why use pointers?

- Some mathematical operations are more convenient using pointers
  - e.g., array operations
- However, we have only looked at *static* data. Pointers are *essential* in dealing with **dynamic data structures**.

- Imagine you are writing a text editor.
  - You could estimate the largest line-length and create arrays of that size (problematic).
  -

MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?
0x3A28213A
0x6339392C,
0x7363682E.
I HATE YOU.

Revision pointers

- What is the value held by p? and how much memory is used by p (in bytes) using 64 bits for its addressable memory?
- `int p;`
- `char p;`
- `void foo( int *p )`
- `char *p;`
- `char **p;`

Revision pointers

- What is the value held by p? and how much memory is used by p (in bytes)?
- `int p;`
- `char p;`
- `void foo( int *p )`
- `char *p;`
- `char **p;`
- `int **p;`
- `long *p;`
- `void *p;`
- `const unsigned long long int * const p;`
- `bubblebobble ************p;`

Pointer interpretation

- `char *p`
  - Address to a single char value
  - Address to a single char value that is the first in an array
- `char *argv[]`
  - Array of “the type” with unknown length
  - Type is `char *`

- `char **argv`
  - `*` Address to the first element to an array of type char *
  - Then, each element in `*` is an…
    - `*` address to the first element to an array of type char

char argv[][3]; // oh no!

Pointer interpretation

- Interpretations of `int **data;`

1. Pointer to pointer to single int value
2. Array of addresses that each point to a single int
3. Address that points to one array of int values
4. Array of addresses that point to arrays of int values

Pointer interpretation

- Interpretations of `int **data;`

1. Pointer to pointer to single int value
2. Array of addresses that each point to a single int
3. Address that points to one array of int values
4. Array of addresses that point to arrays of int values

- Thinking about each ***** as an array:

1. Array size ==1, Array size ==1
2. Array size >=1, Array size == 1
3. Array size ==1, Array size >= 1
4. Array size >=1, Array size >= 1

Pointer comments

- When you call a function in Java, compare passing a primitive type and Object type.

- You may have heard:
  - Pass by value
  - Pass by reference

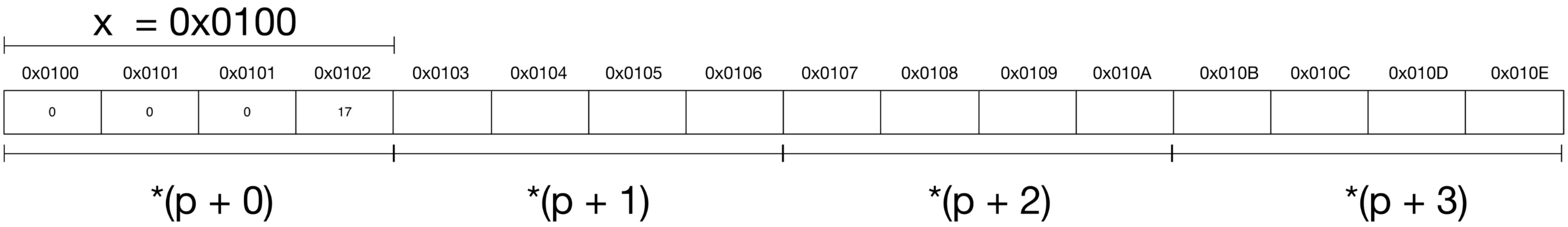
What is the meaning of this in C?

- void has no size, but sizeof(void*) is the size of an address

- Pointers are unsigned numbers, why?

Pointer arithmetic

- int *p = NULL;
- int x[4];
- p = x;



x = 0x0100

0x0100	0x0101	0x0101	0x0102	0x0103	0x0104	0x0105	0x0106	0x0107	0x0108	0x0109	0x010A	0x010B	0x010C	0x010D	0x010E
0	0	0	17												

*(p + 0)

*(p + 1)

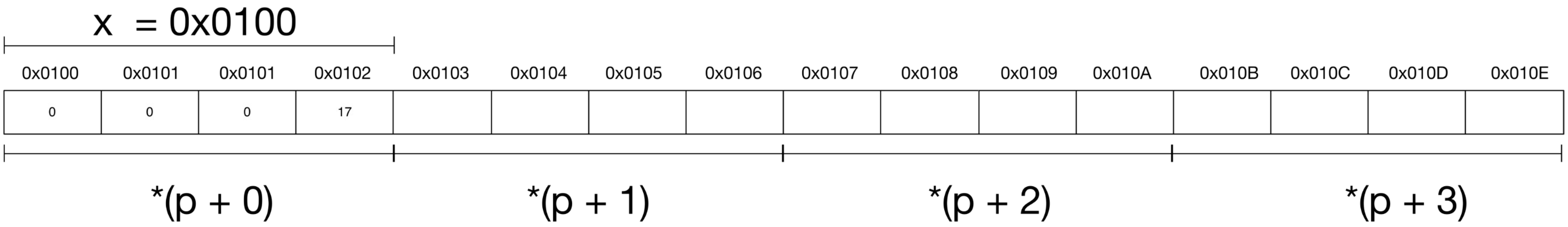
*(p + 2)

*(p + 3)

- Seeking to the nth byte from a starting address?

Pointer arithmetic

- int *p = NULL;
- int x[4];
- p = x;



x = 0x0100

0x0100	0x0101	0x0101	0x0102	0x0103	0x0104	0x0105	0x0106	0x0107	0x0108	0x0109	0x010A	0x010B	0x010C	0x010D	0x010E
0	0	0	17												
*(p + 0)				*(p + 1)				*(p + 2)				*(p + 3)			

- Seeking to the nth byte from a starting address?

```
void *get_address( sometype *data , int n) {
     unsigned char *ptr = (unsigned
char*)data;
     return (void*)(ptr + n);
}
```

sizeof operator

- Not all h/w architectures are the same
  - different sizes for basic types

- C specification does not dictate exactly how many bytes an int will be

- **sizeof** operator returns the number of bytes used to represent the given type or expression
  - `sizeof( char )`
  - `sizeof( int )`
  - `sizeof( float * )`
  - `sizeof ( 1 )`
  - `sizeof( p )`

sizeof operator

- Not all h/w architectures are the same
  - different sizes for basic types

- C specification does not dictate exactly how many bytes an int will be

- **sizeof** operator returns the number of bytes used to represent the given type or expression.
  - `sizeof( char )`
  - `sizeof( int ), sizeof( double )`
  - `sizeof( float * )`
  - `sizeof ( 1 ), sizeof ( 1/2 ), sizeof (1.0 / 2.0)`
  - `sizeof( p ) ????`

sizeof

- Special case for **p**, what is it?
  - `char p;`
  - `char *p;`
  - `char p[8];`

- But…
  - `char msg[100];`
  - `char *p = msg;`
  - `char msg2[] = “hello message”;`
  - `char *p = msg2;`
  - `char *p = “program has ended”;`

- **sizeof** needs to be used carefully

Less familiar types

- The types **`char`** will support the value range from `CHAR_MIN` to `CHAR_MAX` as defined in file <limits.h>
  - `#define UCHAR_MAX       255          /* max value for an unsigned char */`
  - `#define CHAR_MAX        127             /* max value for a char */`
  - `#define CHAR_MIN        (-128)          /* min value for a char */`

- Most C implementations default types as `signed` values, but a warning that you should not assume this.
- **`unsigned`** and **`signed`** enforce the sign usage
  - **`char ch;`**
  - **`signed char ch;`**
  - **`unsigned char ch;`**
  - **`unsigned int total;`**

THE UNIVERSITY OF SYDNEY

Less familiar `const`

- `const` prevents the value being modified
  - `const char *fileheader = “P1”`
  - `fileheader[1] = '3';` Illegal: change of char value

- It can be used to *help* avoid arbitrary changes to memory

- The value `const` protects depends where it appears
  - `char * const fileheader = “P1”`
  - `fileheader = “P3”;` Illegal: change of address value

- Reading right to left:
  - Is an address, points to a char, that is constant
  - Is an address, that is constant

Less familiar `const`

- `const` prevents the value being modified
  - `const char *fileheader = “P1”`
  - `fileheader[1] = '3';` Illegal: change of char value

- It can be used to *help* avoid arbitrary changes to memory

- The value `const` protects depends where it appears
  - `char * const fileheader = “P1”`
  - `fileheader = “P3”;` Illegal: change of address value

- You can cast if you know if the memory is writable

writable

Non-writable

```
char fileheader[] = {'P', '1'};
const char *dataptr = (char*)fileheader;
char *p = (char*)dataptr;
p[1] = '3';
```

P1 - fgets example reading from stdin

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

#define BUFLEN (64)

int main(int argc, char **argv) {
  int len;
  char buf[BUFLEN];
  while (fgets(buf, BUFLEN, stdin) != NULL) {
    len = strlen(buf);
    printf("%d\n", len);
  }
  return 0;
}
```

Floating point types

- Exact bit representation unknown, usually IEEE 754

- Generally, floating point number **x** is defined as:

$$x = sb^e \sum_{k=1}^{p} f_k b^{-k}, \quad e_{\min} \le e \le e_{\max}$$

- s sign
- b base of exponent (e.g. 2, 10, 16)
- e exponent
- p precision
- f_k nonnegative integer less than b

+0	-0
+ve / 0 = +infinite	-ve / 0 = -infinite
NaN (not a number)	Zero exponents…

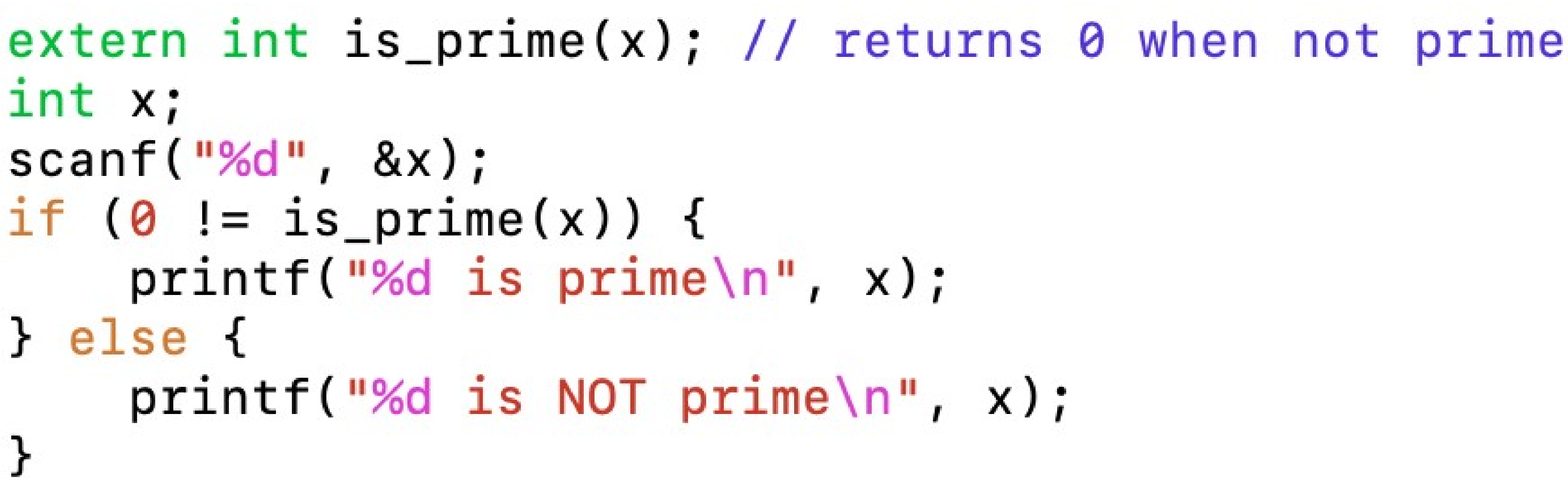
Security matters

COMP2017/COMP9017

FACULTY OF
ENGINEERING

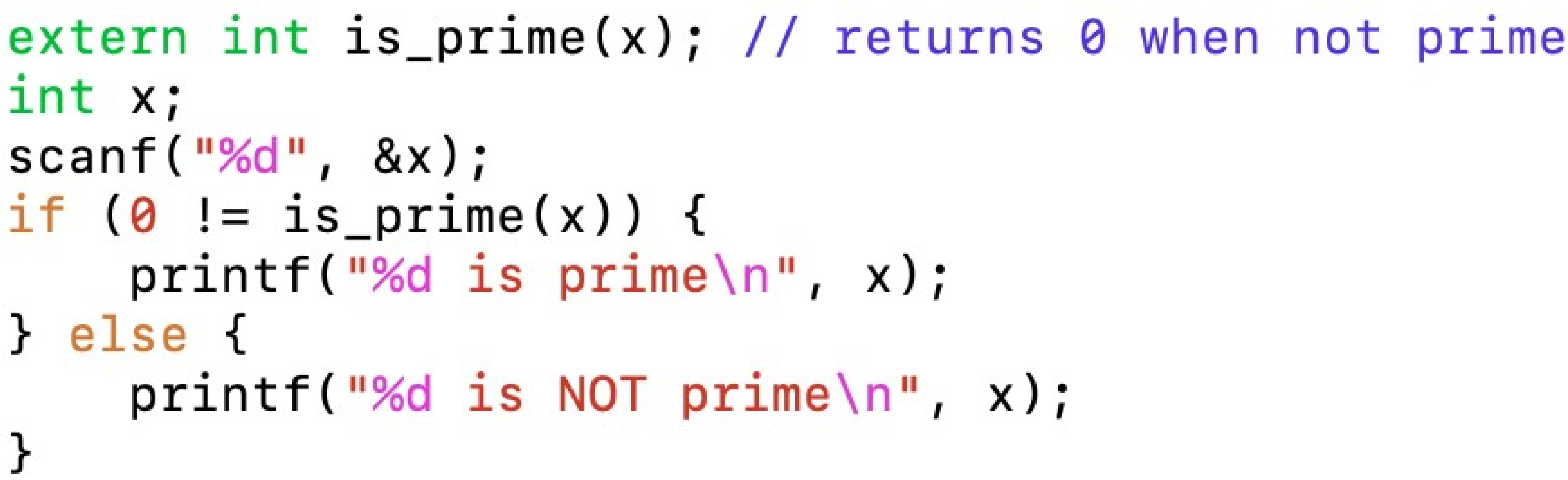
THE UNIVERSITY OF
SYDNEY

Uninitialised values



```c
extern int is_prime(x); // returns 0 when not prime
int x;
scanf("%d", &x);
if (0 != is_prime(x)) {
    printf("%d is prime\n", x);
} else {
    printf("%d is NOT prime\n", x);
}
```

Uninitialised values



```
extern int is_prime(x); // returns 0 when not prime
int x;
scanf("%d", &x);
if (0 != is_prime(x)) {
    printf("%d is prime\n", x);
} else {
    printf("%d is NOT prime\n", x);
}
```

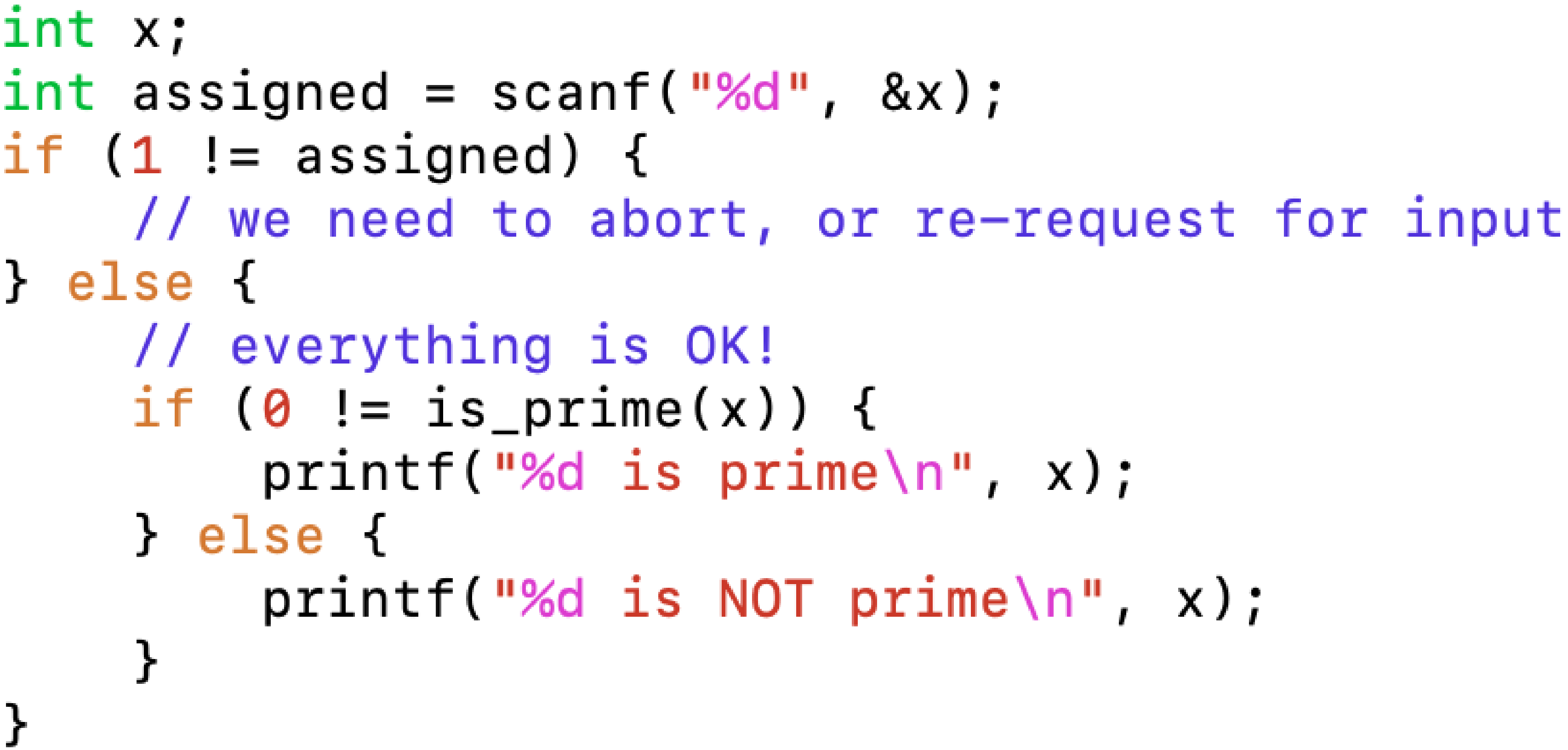
If scanf fails, **x** remains uninitialised!

That value carries through to the next step

```
$ echo “abc” | ./program
```

Uninitialised values

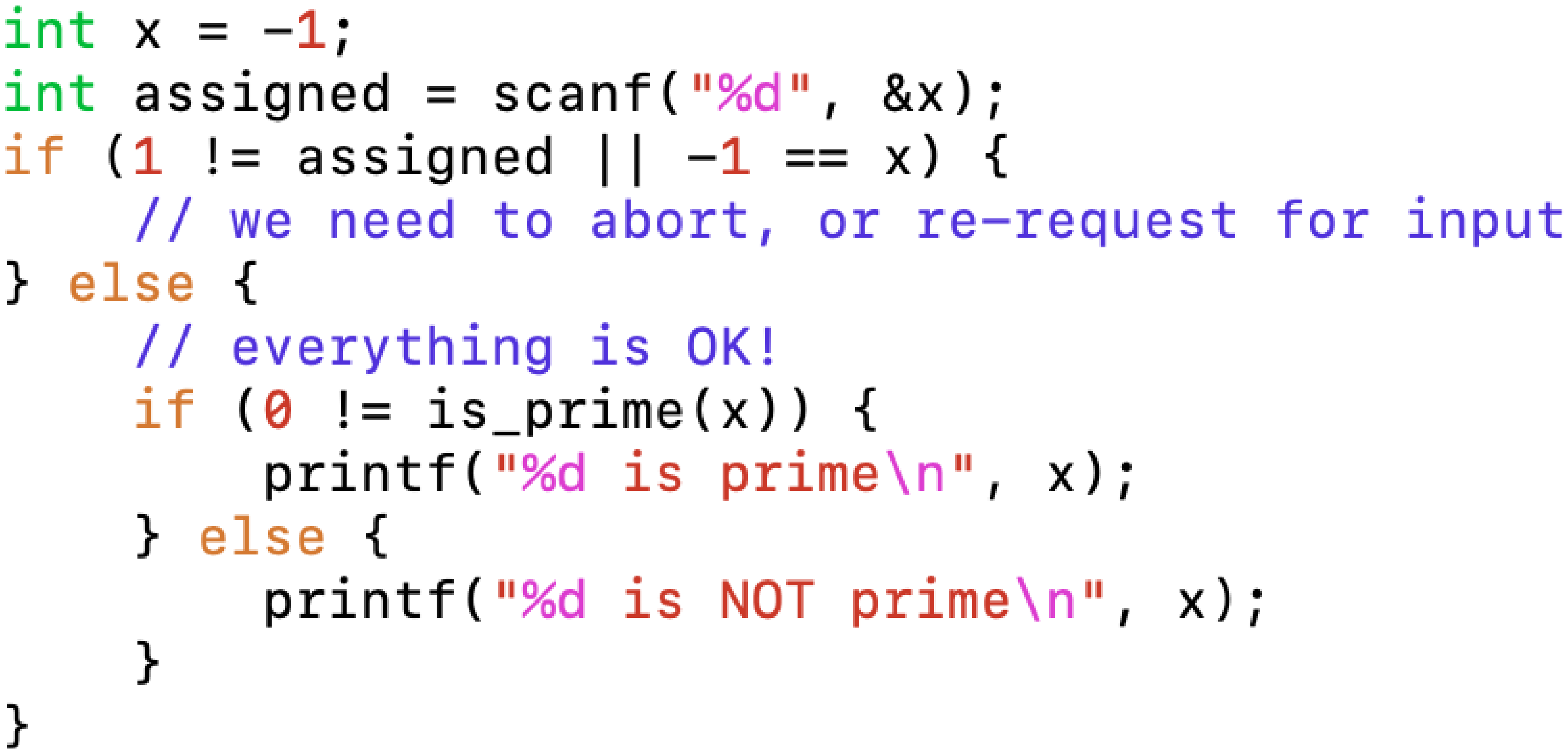
- FIX 1 – Detect errors where the value may not be correctly set
- **int** scanf(const char *restrict format, ...);
  - *return the number of input items assigned. This can be fewer than provided for, or even zero, in the event of a matching failure. –* ***man 3 scanf***



```
int x;
int assigned = scanf("%d", &x);
if (1 != assigned) {
    // we need to abort, or re-request for input
} else {
    // everything is OK!
    if (0 != is_prime(x)) {
        printf("%d is prime\n", x);
    } else {
        printf("%d is NOT prime\n", x);
    }
}
```

Uninitialised values

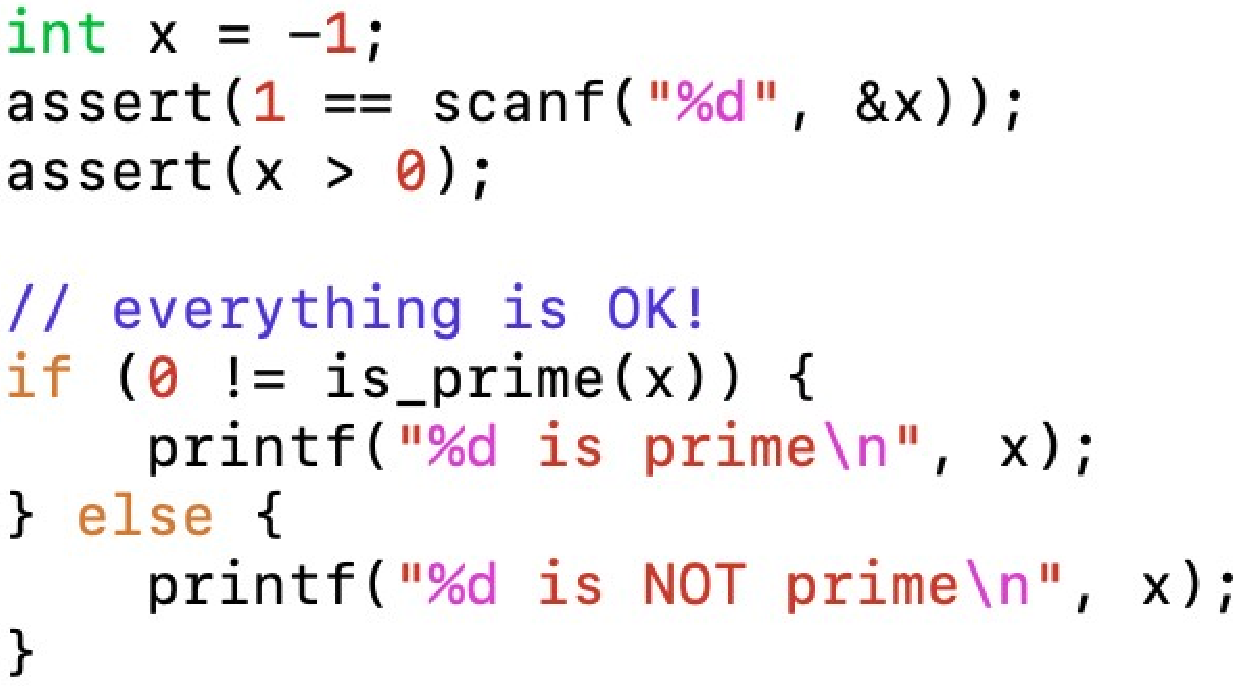
- FIX 1+2 – Use a sentinel value that is outside the useful range of data
  - This can also be a form of input validation – we do not expect **-1**



```c
int x = -1;
int assigned = scanf("%d", &x);
if (1 != assigned || -1 == x) {
    // we need to abort, or re-request for input
} else {
    // everything is OK!
    if (0 != is_prime(x)) {
        printf("%d is prime\n", x);
    } else {
        printf("%d is NOT prime\n", x);
    }
}
```

Uninitialised values

- FIX DEV STYLE – Use assert() statements to check your expectation of program state. Otherwise...stop the whole program!

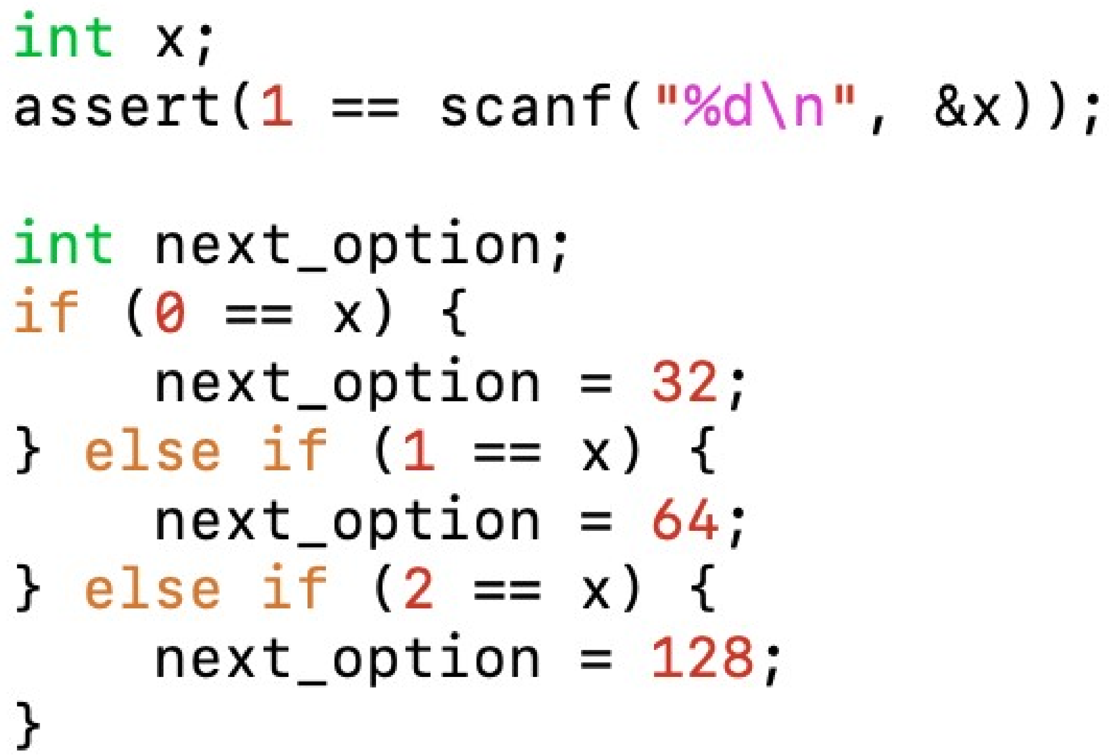


```c
int x = -1;
assert(1 == scanf("%d", &x));
assert(x > 0);

// everything is OK!
if (0 != is_prime(x)) {
    printf("%d is prime\n", x);
} else {
    printf("%d is NOT prime\n", x);
}
```

Uninitialised values

- Another example. What is the problem this time?

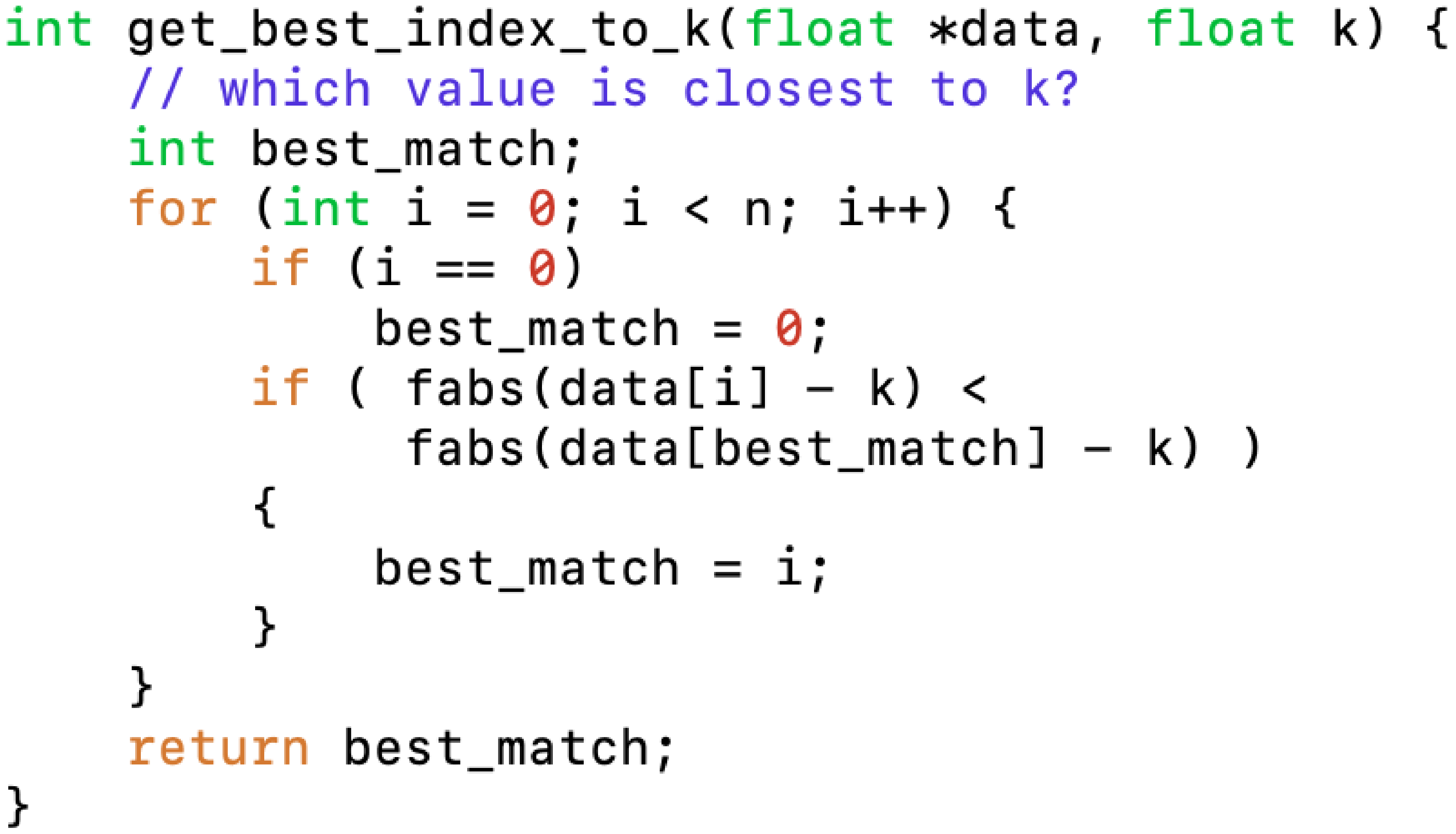


```
int x;
assert(1 == scanf("%d\n", &x));

int next_option;
if (0 == x) {
    next_option = 32;
} else if (1 == x) {
    next_option = 64;
} else if (2 == x) {
    next_option = 128;
}
```

Uninitialised values

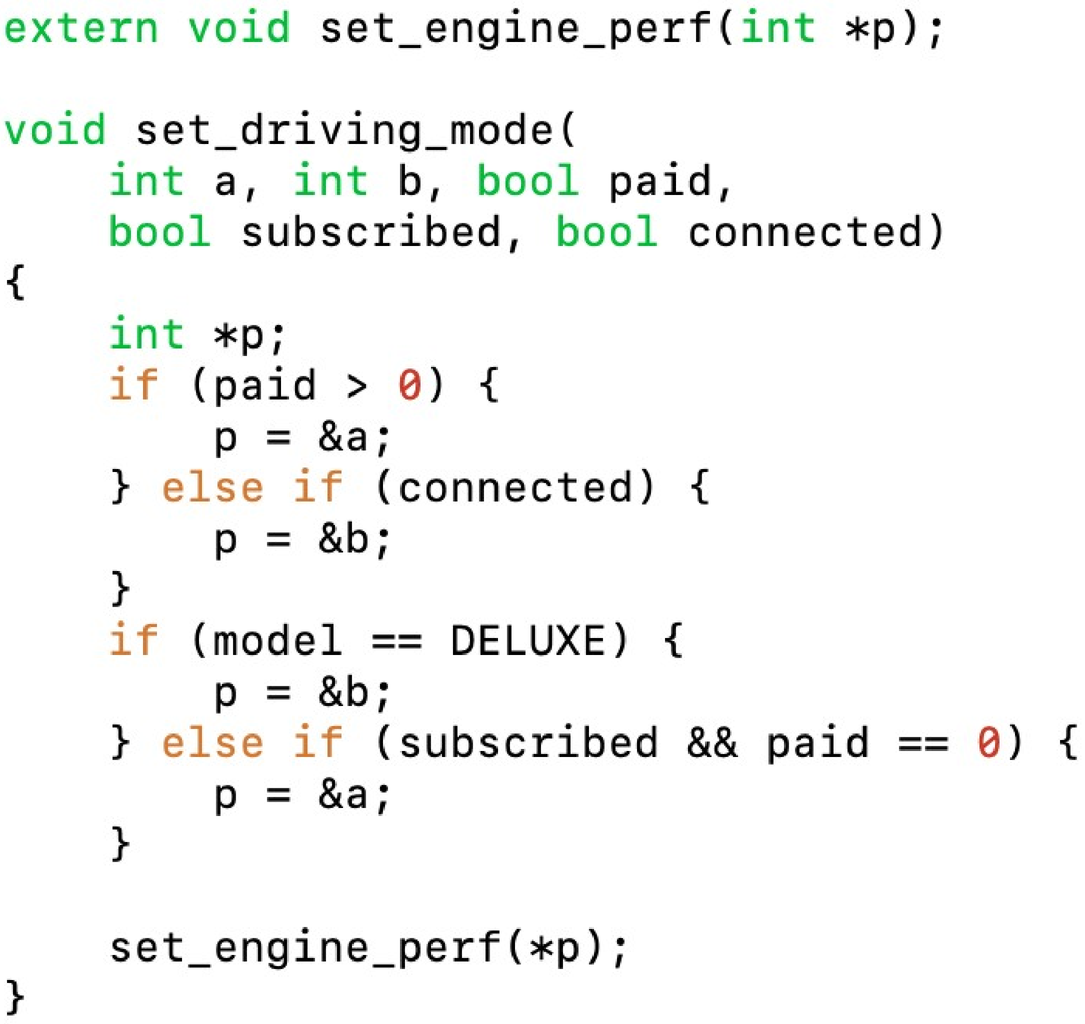
- Another example. What is the problem this time?



```
int get_best_index_to_k(float *data, float k) {
    // which value is closest to k?
    int best_match;
    for (int i = 0; i < n; i++) {
        if (i == 0)
            best_match = 0;
        if ( fabs(data[i] - k) <
             fabs(data[best_match] - k) )
        {
            best_match = i;
        }
    }
    return best_match;
}
```

Uninitialised values

- › Another example.
- › What is the problem?



```
extern void set_engine_perf(int *p);

void set_driving_mode(
    int a, int b, bool paid,
    bool subscribed, bool connected)
{
    int *p;
    if (paid > 0) {
        p = &a;
    } else if (connected) {
        p = &b;
    }
    if (model == DELUXE) {
        p = &b;
    } else if (subscribed && paid == 0) {
        p = &a;
    }

    set_engine_perf(*p);
}
```

Uninitialised values

- Uninitialised values can remain uninitialised by
  - By not checking the outcome of an operation
  - if there is a failure to have the correct flow control (if, switch, loops)

- Conventions still allow for the declaration of a variable, but for pointers, this should always be **NULL**

Enums

COMP2017/COMP9017

FACULTY OF ENGINEERING

THE UNIVERSITY OF SYDNEY

The picture so far – simple types

- simple data types:
  - int, char, float.....

- pointers to simple data types:
  - int *, char *, float *

Enumerated types

- enums (enumerated types) are another simple type
- enums map to int
- an enum associates a name with a value

Enumerated types

```
enum day_name
{
   Sun, Mon, Tue, Wed, Thu, Fri, Sat, day_undef
};
```

- Maps to integers, 0 .. 7
- Can do things like ‘Sun ++'
- very close to int

Enumerated types

```
enum month_name
{
   Jan, Feb, Mar, Apr, May, Jun,
   Jul, Aug, Sep, Oct, Nov, Dec,
   month_undef
};
```

Enumerated types

- we could always use integers to represent a set of elements
- but enums make your code much more readable
- eg red instead of 0
  - How many bytes for an array of enum?