Pointers

COMP2017/COMP9017

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:

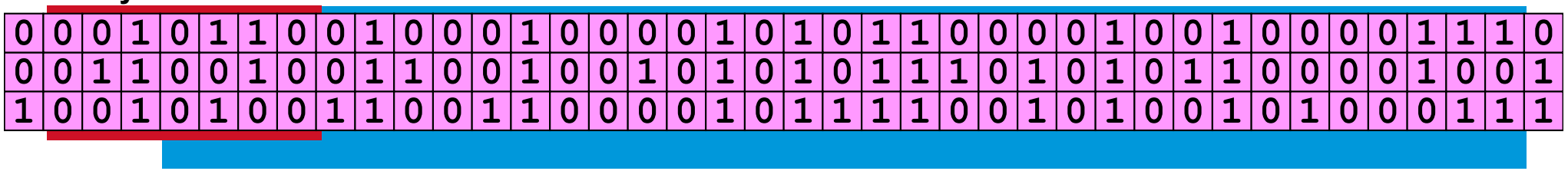
```
int counter;
```

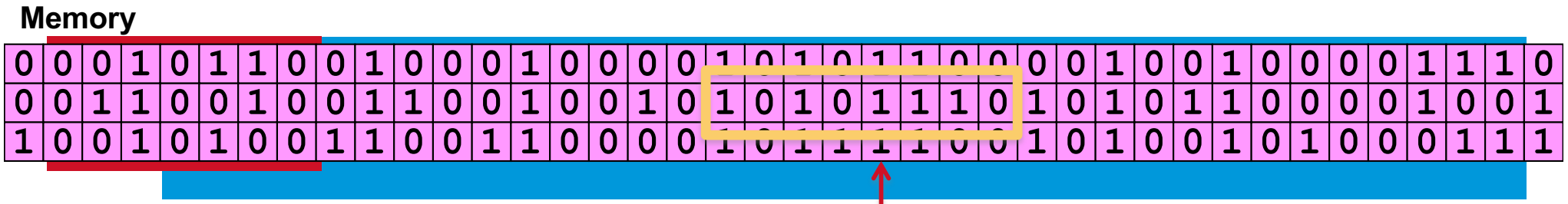
reserve memory for an integer and call it "counter"

- *Initialisation* or *assignment* specifies content:

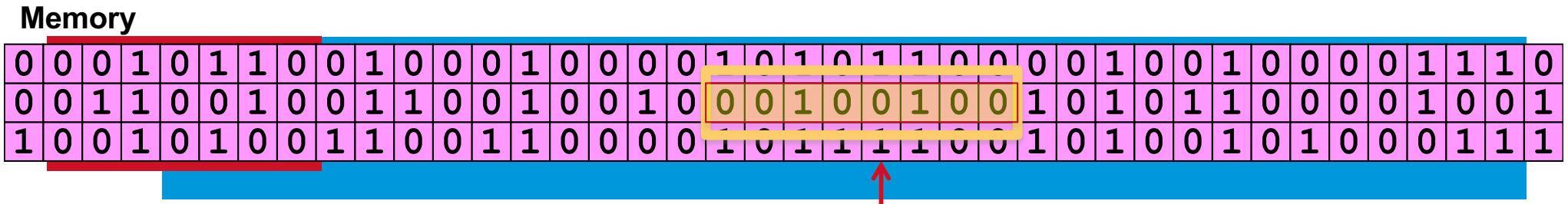
```
int  counter = 0;
counter = 0;
```

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

```
char a;
```


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 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

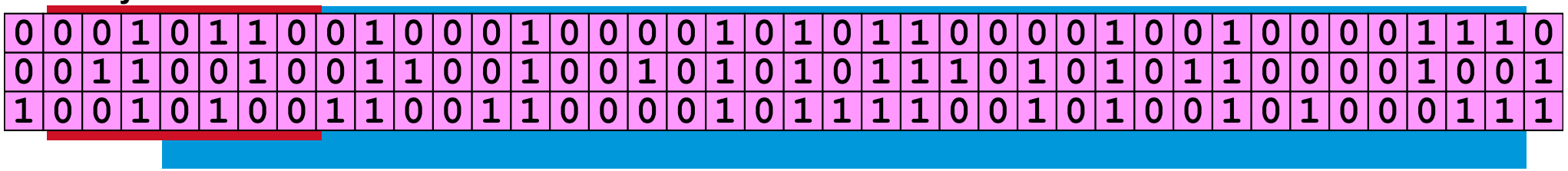
- Arrays are indexed collections of the same type
- Declaration of an array:

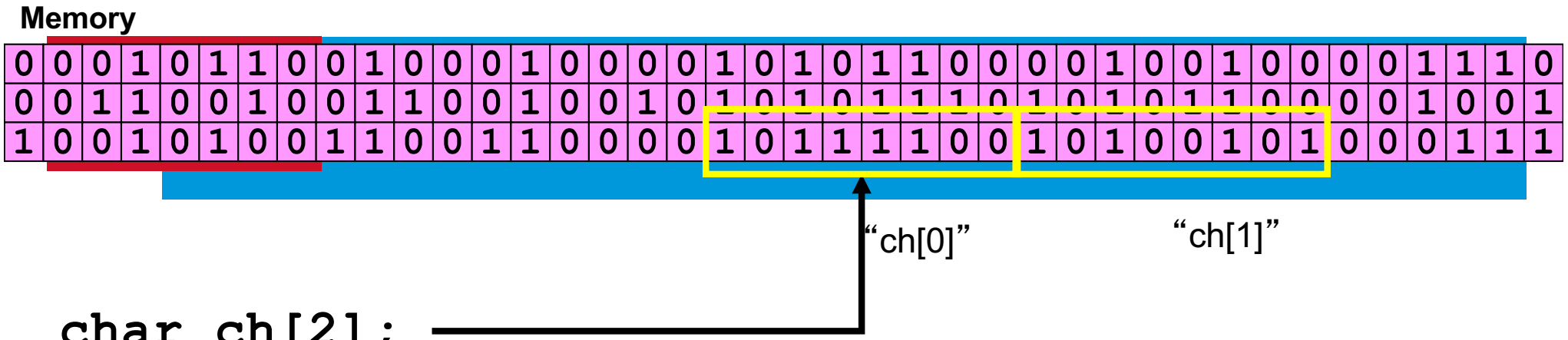
```
int  counters[MAX];
char alphabet[26];
```

- Initialisation of an array:

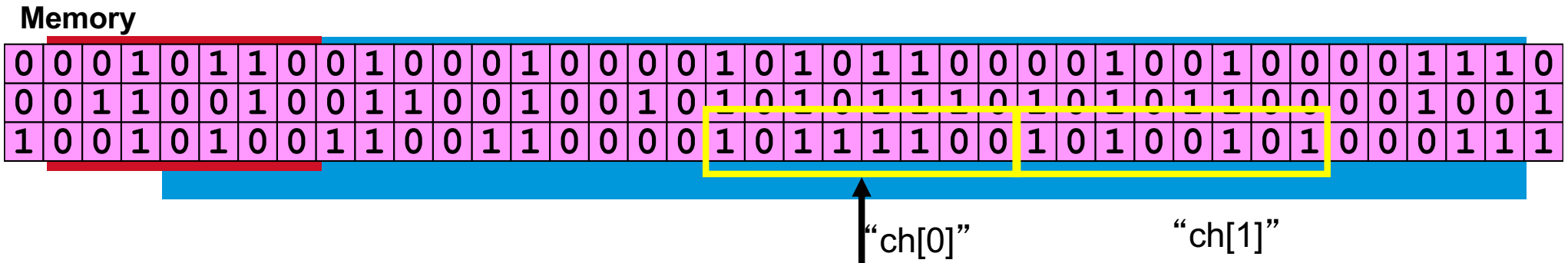
```
for (i = 0; i < MAX; i++)
    counters[i] = i;
```

Memory

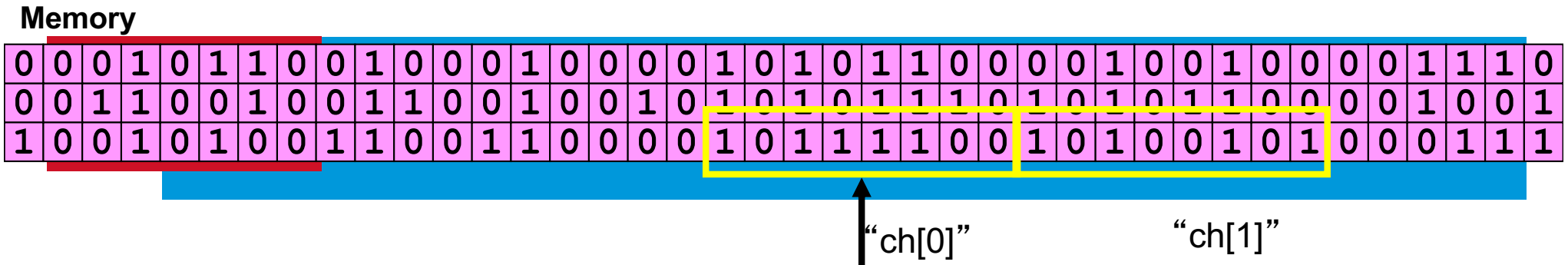

0 0 0 1 0 1 1 0 0 1 0 0 0 1 0 0 0 0 1 0 1 0 1 1 0 0 0 0 1 0 0 1 0 0 0 0 1 1 1 0
0 0 1 1 0 0 1 0 0 1 1 0 0 1 0 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 1 0 0 0 0 1 0 0 1
1 0 0 1 0 1 0 0 1 1 0 0 1 1 0 0 0 0 1 0 1 1 1 1 0 0 1 0 1 0 0 1 0 1 0 0 0 1 1 1


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
“ch[0]”
“ch[1]”

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

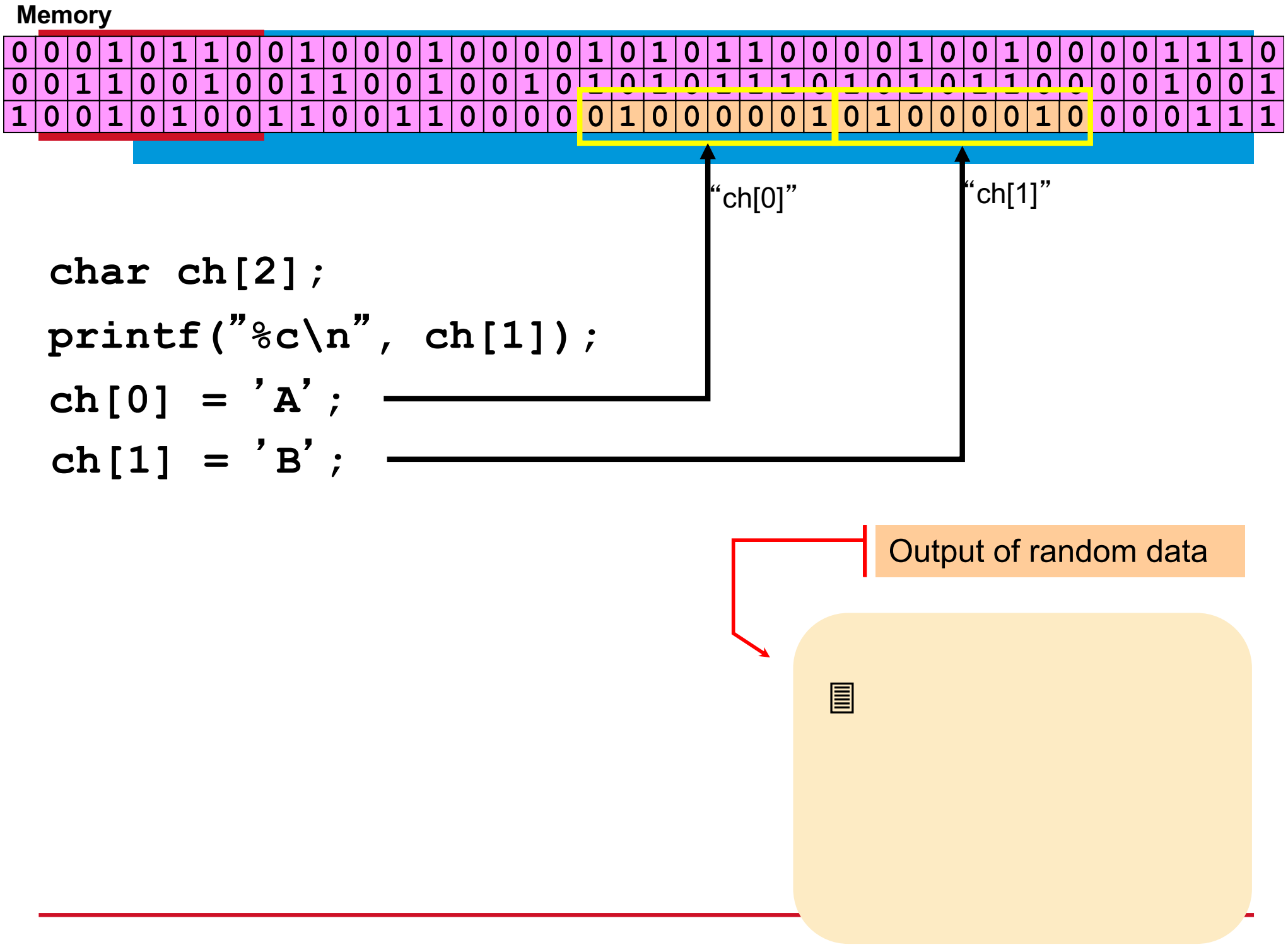


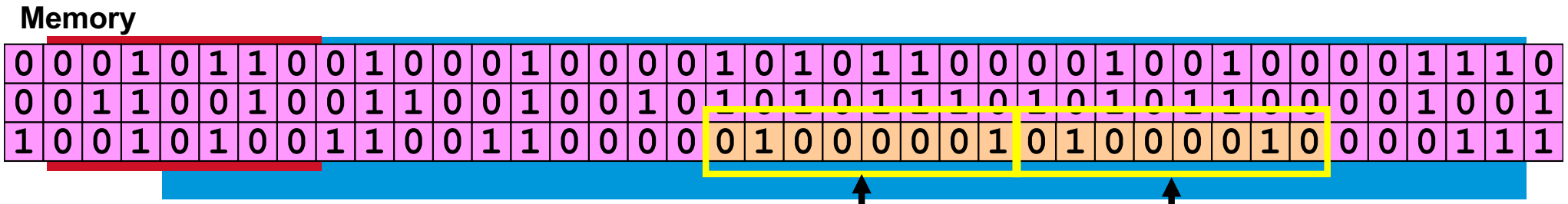

```
char ch[2];
printf(”%c\n”, ch[1]);
```




```
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
"ch[0]"
"ch[1]"
char ch[2];
printf("%c\n", ch[1]);
ch[0] = 'A';
ch[1] = 'B';
Output of random data



"ch[0]"

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

```
char myHobby[] = ”rowing”;
```

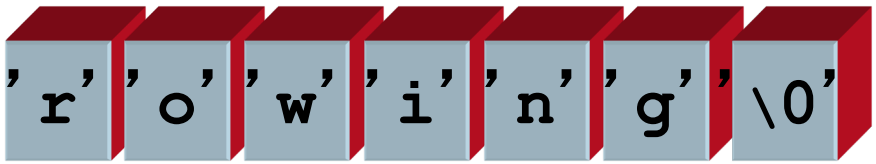
The compiler can determine the required size by counting characters, so the array size is **optional**. A larger size *may* be specified.

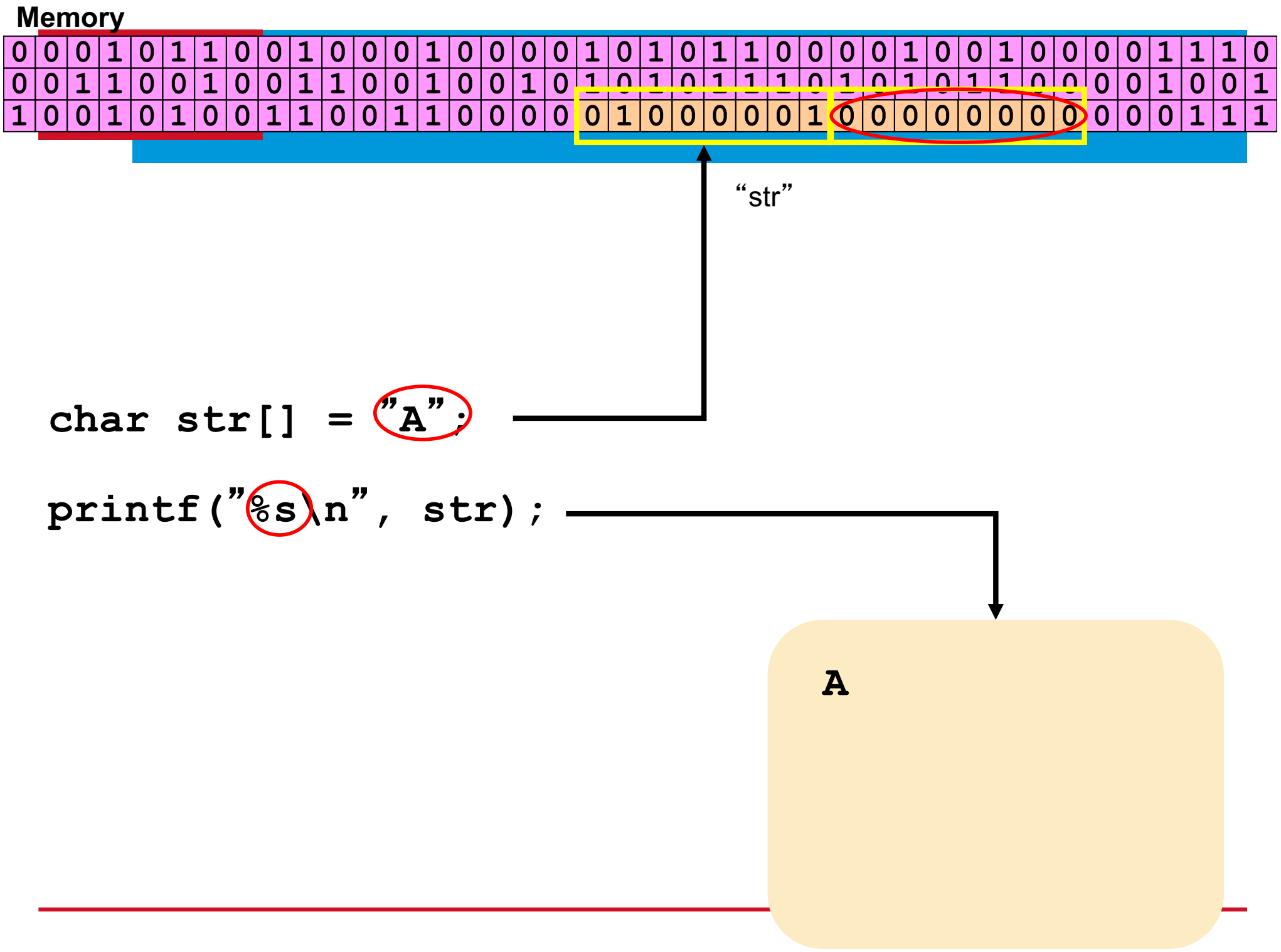
Strings

- 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";
```




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

THE UNIVERSITY OF SYDNEY

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

THE UNIVERSITY OF SYDNEY

Memory

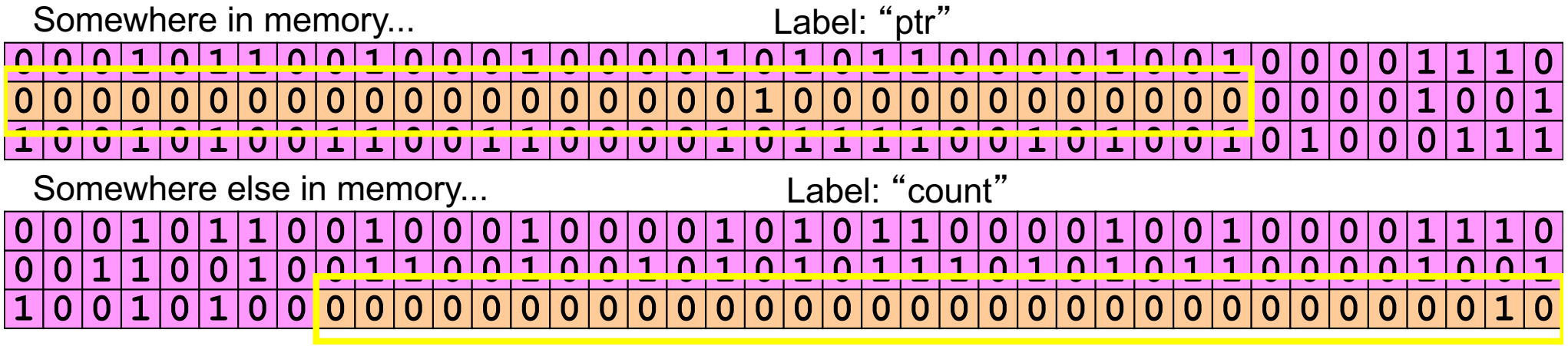
address	content
0x100	00100010
0x101	01010010
0x102	00110110
0x103	00101010
0x104	10100010
0x105	01100010
0x106	00111010
0x107	00100110
0x108	11100010
...	

```
char initial = 'A';

char * initp = &initial
```

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

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




```
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*.

Pointers (notation)

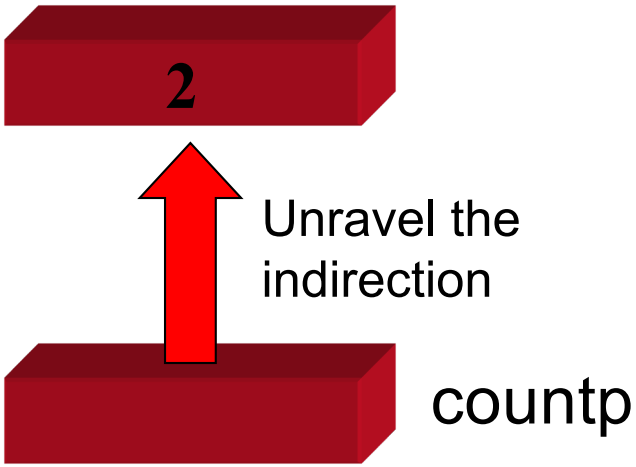
- The indirection operator, '`*`', is used to "unravel" the indirection:

`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);
```

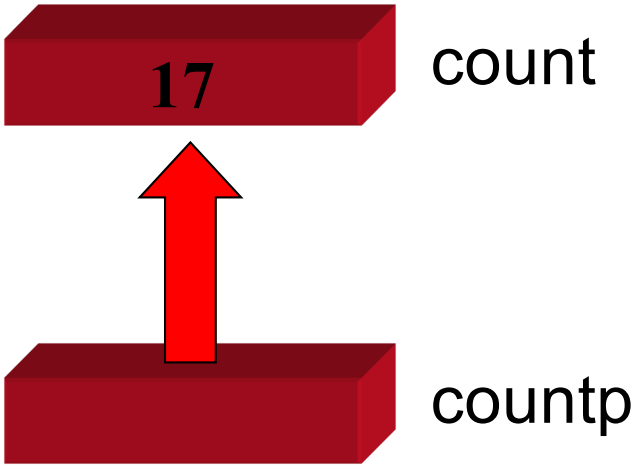
17

```
printf("%d", *countp);
```

17

```
printf("%d", countp);
```

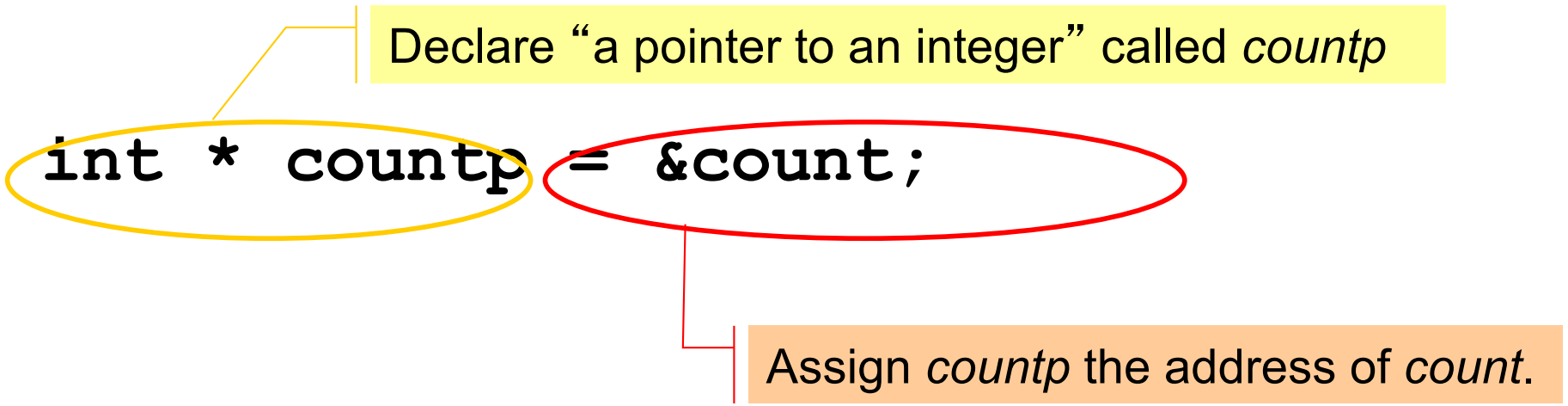
Don't know… but it will be the address of *count*. Why?



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:



Examples

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

Receiving an integer from standard input:

```
int age;
scanf("%d", &age);
```

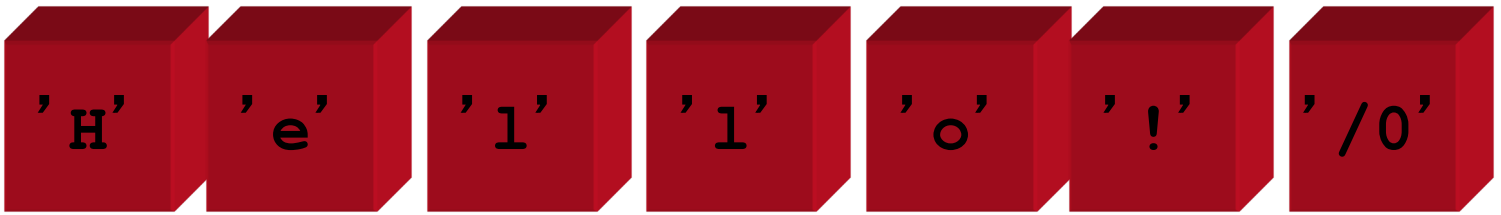
This argument is required by *scanf()* to be a pointer. Since we are using a simple integer, *age*, we pass it's address.

Use of pointer notation to manipulate arrays...

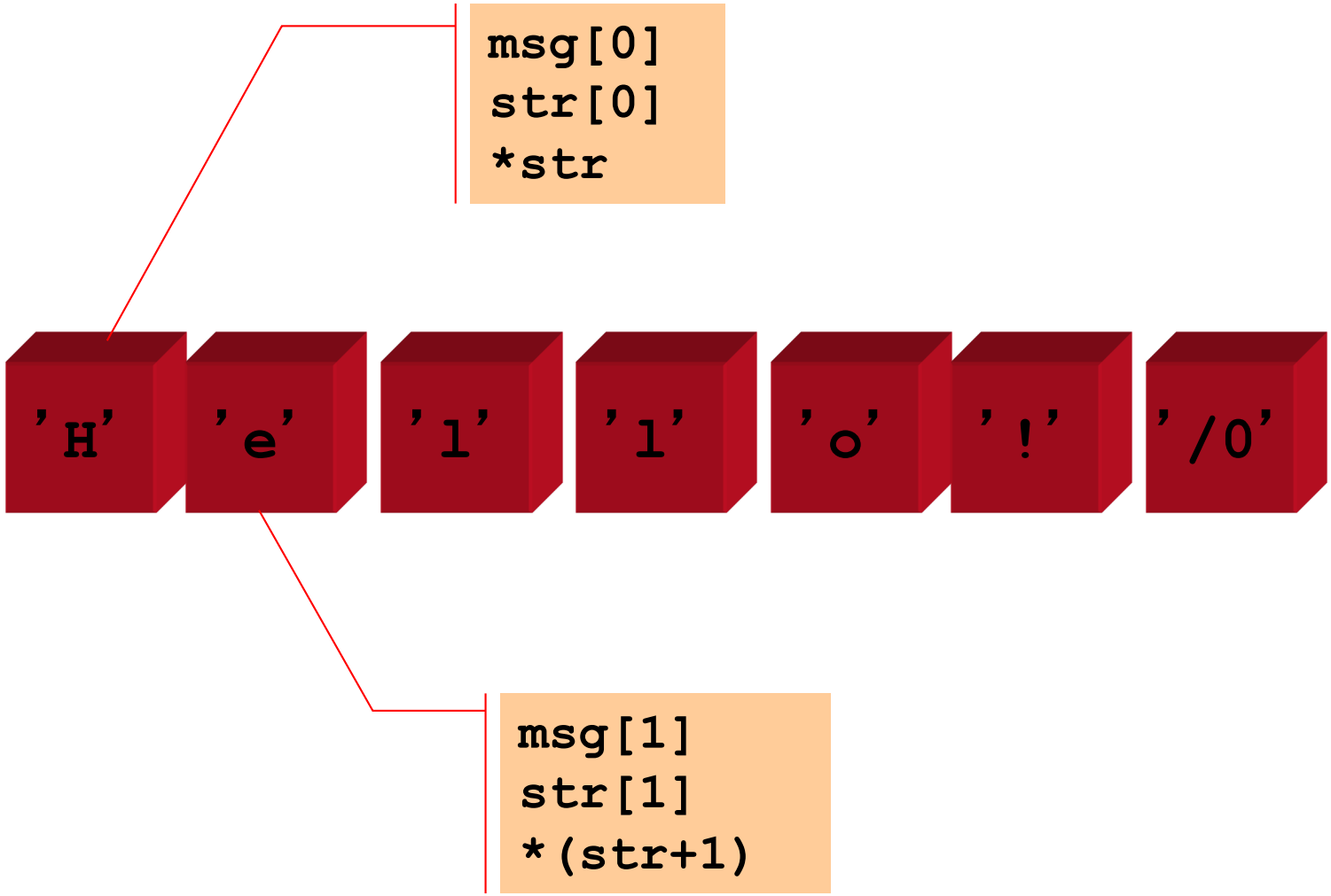
```
char msg[] = "Hello!";
char *str = &msg[0];
```

OR:

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



Examples

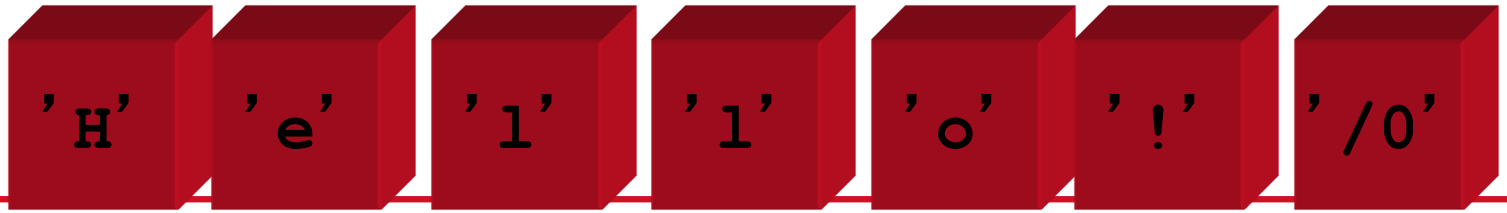


Examples

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

Moving through a string:

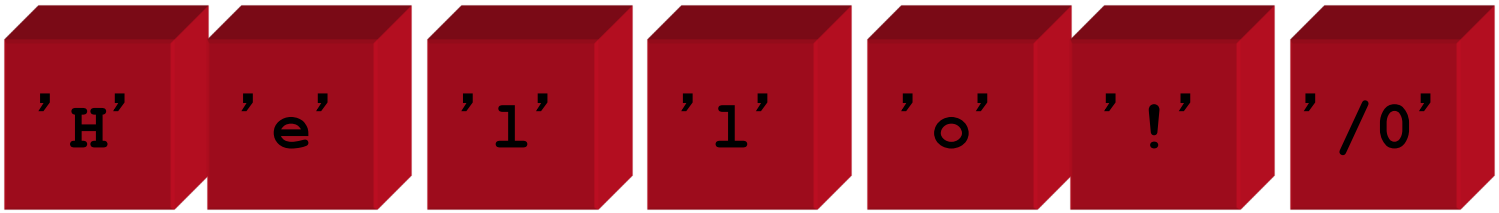
```
while (*str != '\0')
    str++;
```



Examples

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

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



THE UNIVERSITY OF SYDNEY

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).
  - Or you could dynamically allocate memory as needed, using pointers.


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)?
- `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;`

THE UNIVERSITY OF SYDNEY

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

Pointer interpretation

- Interpretations of `int **data;`
  1. Pointer to pointer to single int value
  2. Array of addresses that 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

THE UNIVERSITY OF SYDNEY

Pointer interpretation

- Interpretations of `int **data;`
  1. Pointer to pointer to single int value
  2. Array of addresses that 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

THE UNIVERSITY OF SYDNEY

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

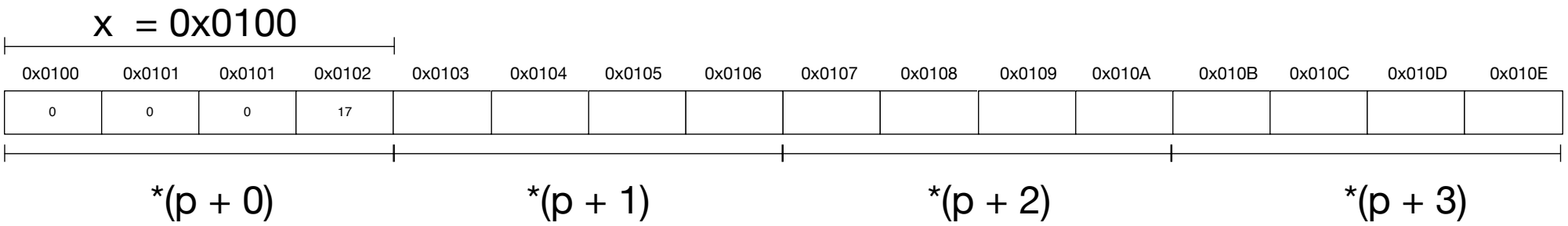
What is the meaning of this in C?

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

› Pointers are unsigned numbers, why?

Pointer arithmetic

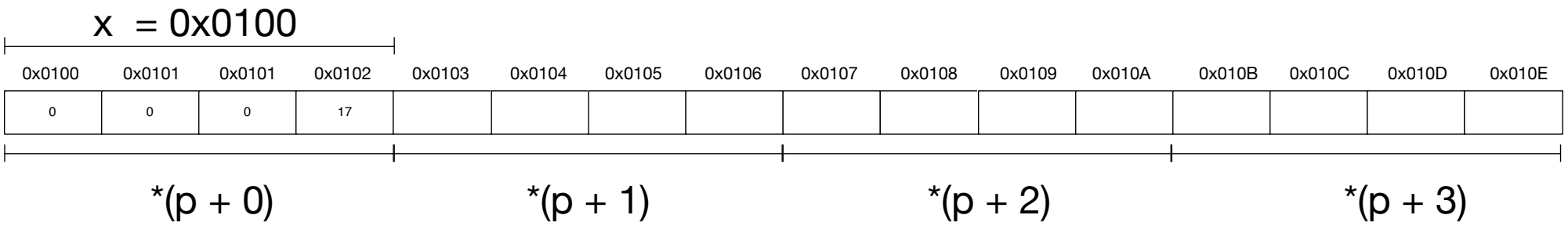
- int *p = NULL;
- int x[4];
- p = x;



- Seeking to the nth byte from a starting address?

Pointer arithmetic

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



- 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

THE UNIVERSITY OF SYDNEY

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

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

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

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

writable

Non-writable

THE UNIVERSITY OF SYDNEY

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…

Enums

COMP2017/COMP9017

FACULTY OF
ENGINEERING

THE UNIVERSITY OF
SYDNEY

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?