Pointers

COMP2017/COMP9017

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;
```

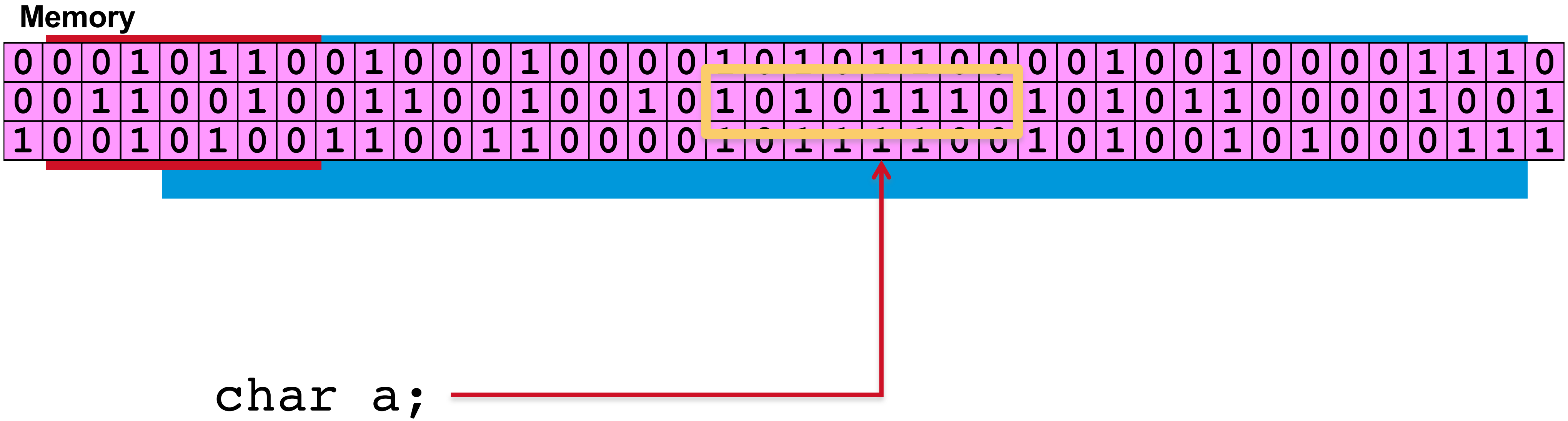
reserve memory for an integer and call it "counter"

- *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



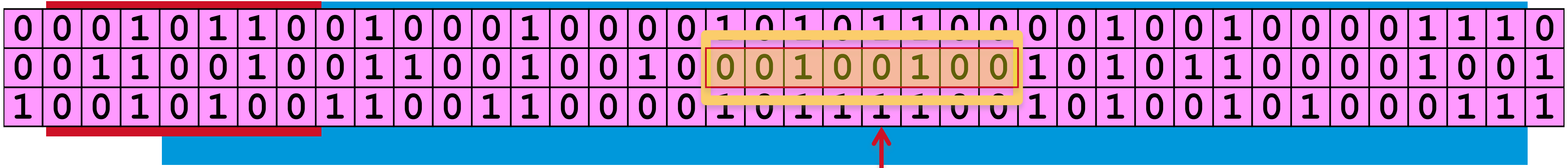
Memory

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;
```

8 bits

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 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

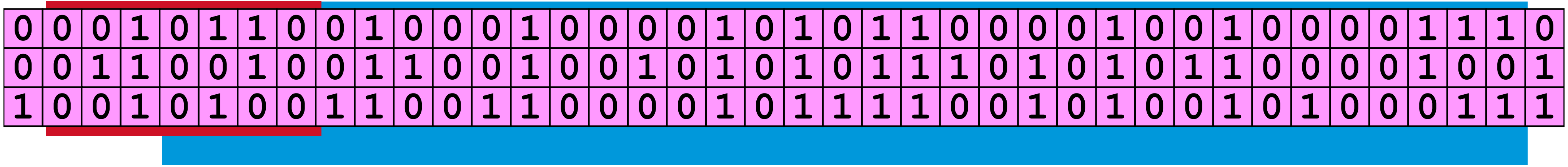
- Declaration of an array:

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

- Initialisation of an array:

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

Memory



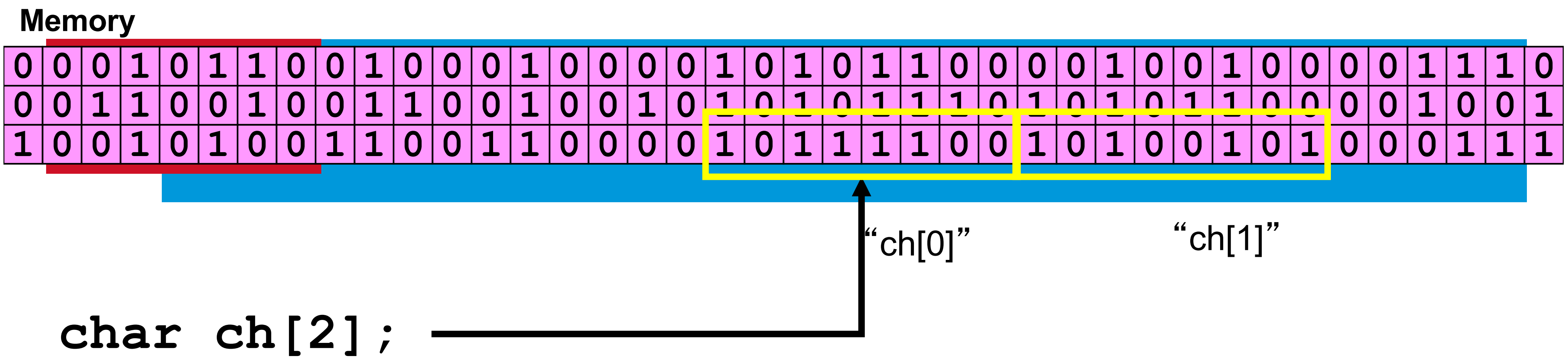
0	0	0	1	0	1	1	0	0	1	0	0	0	1	0	0	0	0	1	0	1	0	1	1	0	0	0	0	1	0	0	1	0	0	0	0	1	1	1	0
0	0	1	1	0	0	1	0	0	1	1	0	0	1	0	0	1	0	1	0	1	0	1	1	1	0	1	0	1	0	1	1	0	0	0	0	1	0	0	1
1	0	0	1	0	1	0	0	1	1	0	0	1	1	0	0	0	0	1	0	1	1	1	1	0	0	1	0	1	0	0	1	0	1	0	0	0	1	1	1

Char a[10] = "geek";	Char *p = "geek";
1) a is an array	1) p is a pointer variable
2) sizeof(a) = 10 bytes	2) sizeof(p) = 4 bytes
3) a and &a are same	3) p and &p aren't same
4) geek is stored in stack section of memory	4) p is stored at stack but geek is stored at code section of memory

5) char a[10] = "geek";
a = "hello"; //invalid
> a, itself being an address and string constant is also an address, so not possible.
	5) char *p = "geek";
	p = "india"; //valid

6) a++ is invalid	6) p++ is valid

7) char a[10] = "geek";
a[0] = 'b'; //valid
	7) char *p = "geek";
	p[0] = 'k'; //invalid
	> Code section is r- only.



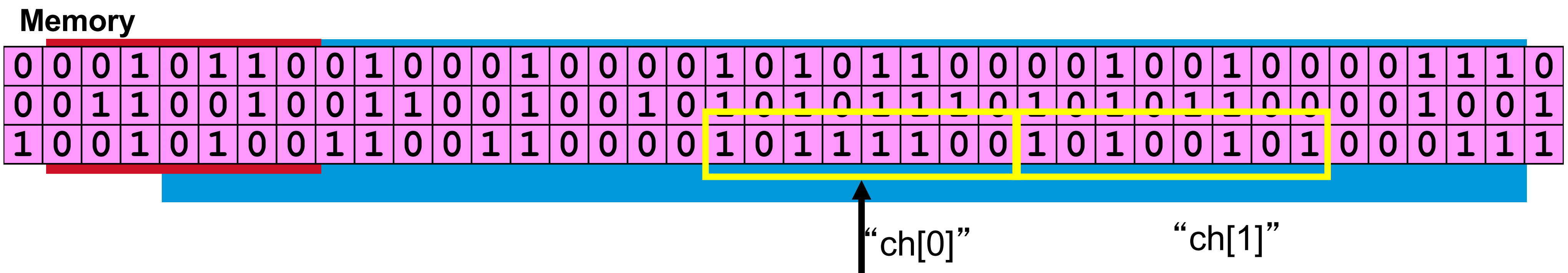
```
0 0 0 1 0 1 1 0 0 1 0 0 0 1 0 0 0 0 1 0 1 0 1 1 0 0 0 0 1 0 0 1 0 0 0 0 1 1 1 0
0 0 1 1 0 0 1 0 0 1 1 0 0 1 0 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 1 0 0 0 0 1 0 0 1
1 0 0 1 0 1 0 0 1 1 0 0 1 1 0 0 0 0 1 0 1 1 1 1 0 0 1 0 1 0 0 1 0 1 0 0 0 1 1 1
```

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

two groups of 8 bits
continuous block of memory

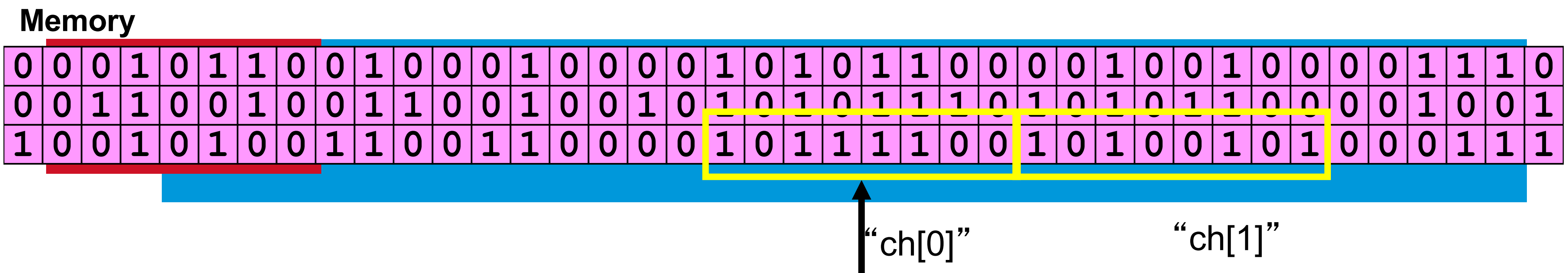
*(0x1000+1) -> add 1 to 0x1000 and then return the value at memory address at 0x1001

(*(0x1000))+1 -> return the value at memory address 0x1000, then add one to it



```
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]);
```



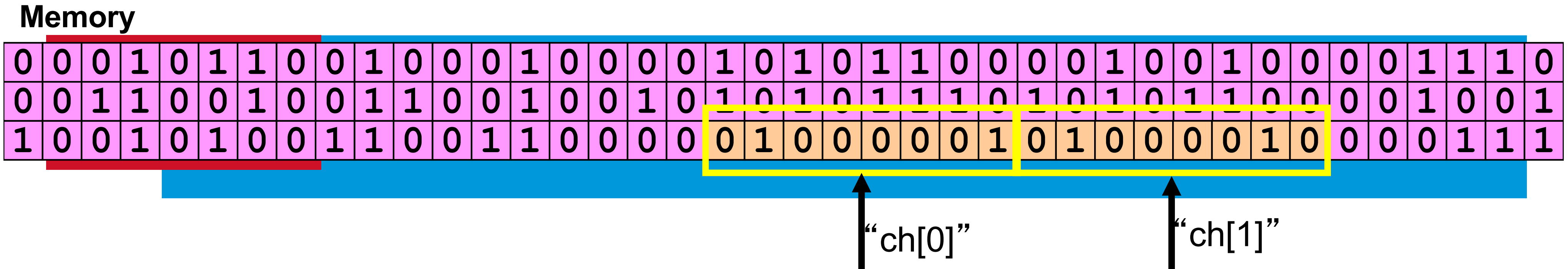
Memory

```
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

▤



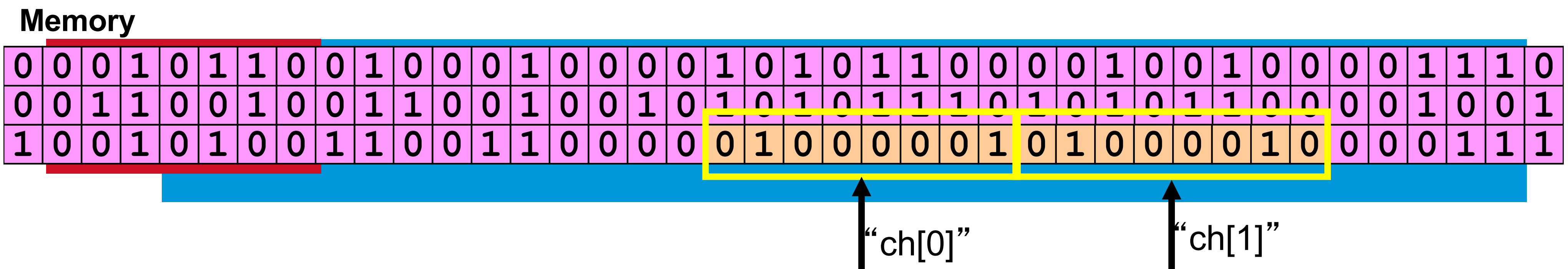
Memory

```
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

```
▤
AB
```

Output of initialised data

Strings

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

LHS - declaration of type
RHS - the data that stores in

```
char myHobby[] = ”rowing”;
```

string is actually a charcter array + null character

“” (double quote) will be carried around the program forever, it will be call
upon when we call printf function

7 bytes to store this array

automatically add null terminator at compile time, not run time

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

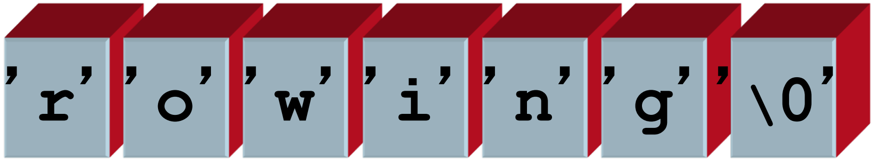
Strings

› Strings resemble an array of characters.

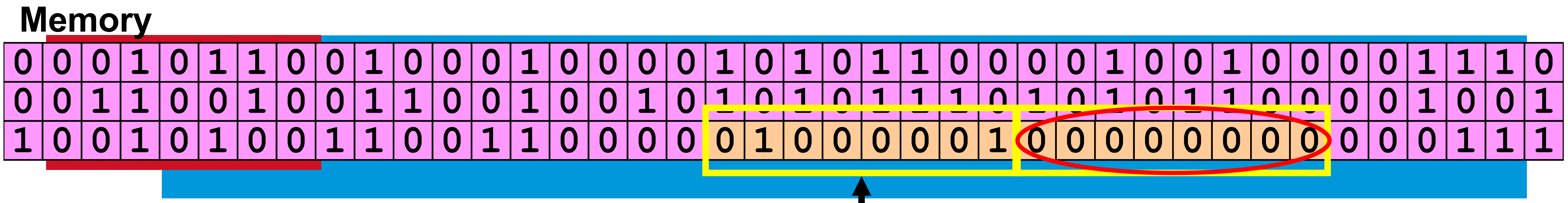
› However, in *C*, all strings are NULL-terminated.

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

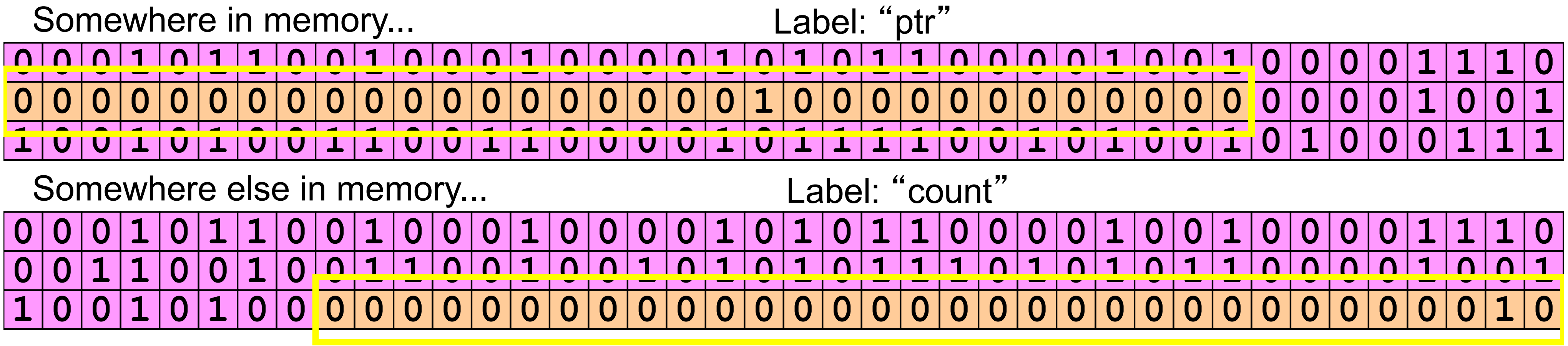
...

```
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 1 0 0 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:

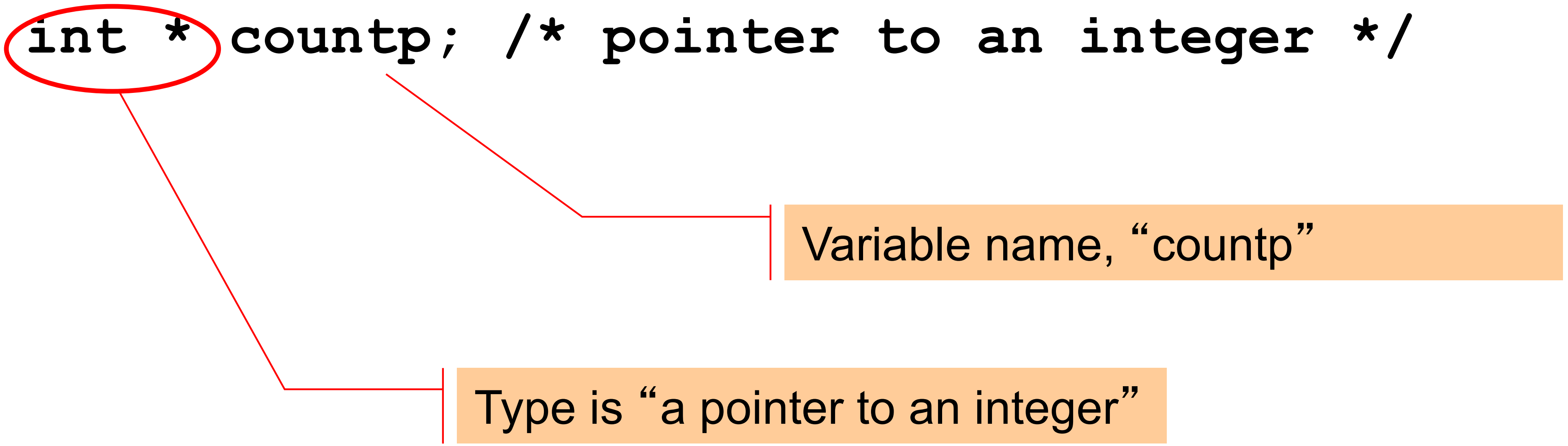
- address operator, ‘**&**’
- indirection operator, ‘*****’

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*.

Pointers (notation)

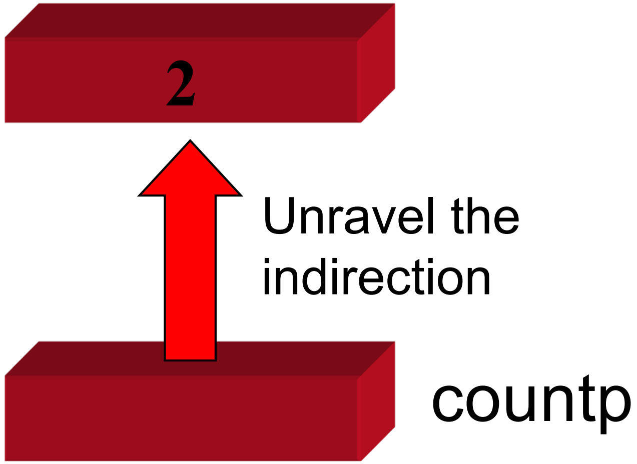
- The indirection operator, ‘`*`’, is used to “unravel” the indirection:

`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);
```

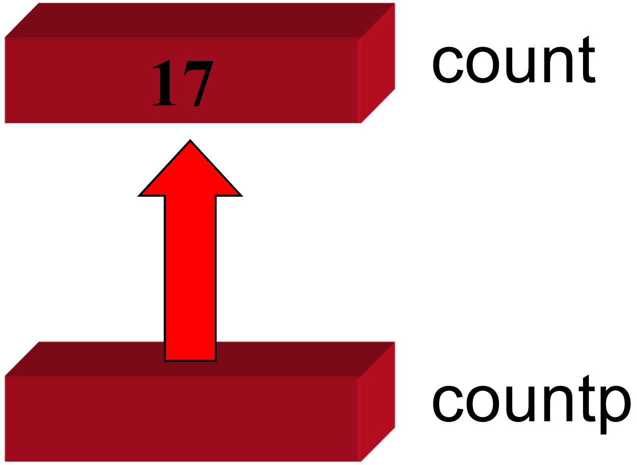
17

```
printf(”%d”, *countp);
```

17

```
printf(”%d”, countp);
```

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

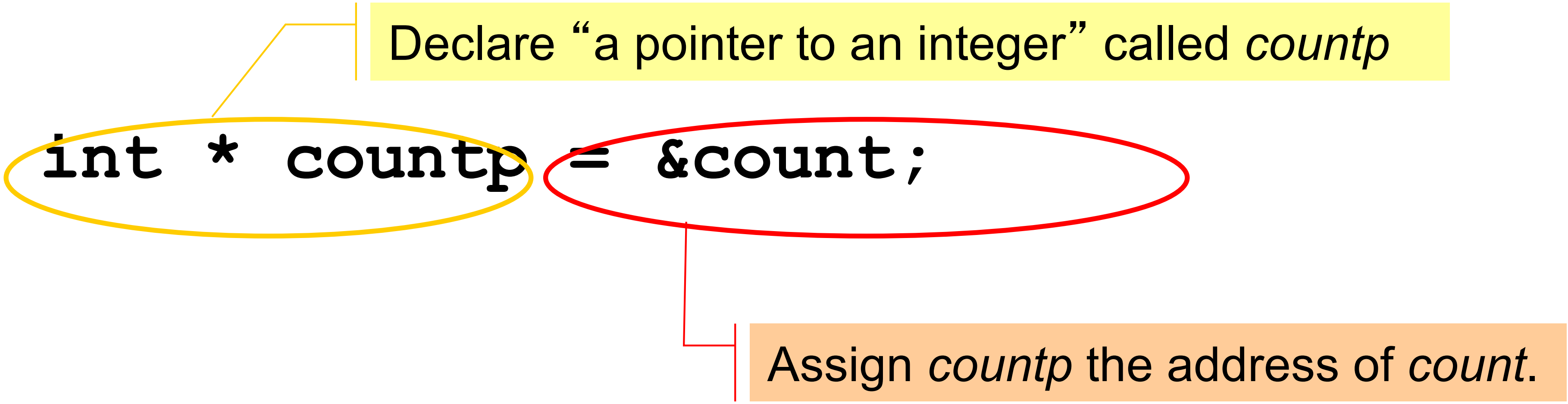


THE UNIVERSITY OF SYDNEY

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:



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

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

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.

THE UNIVERSITY OF SYDNEY

Pointers and arrays

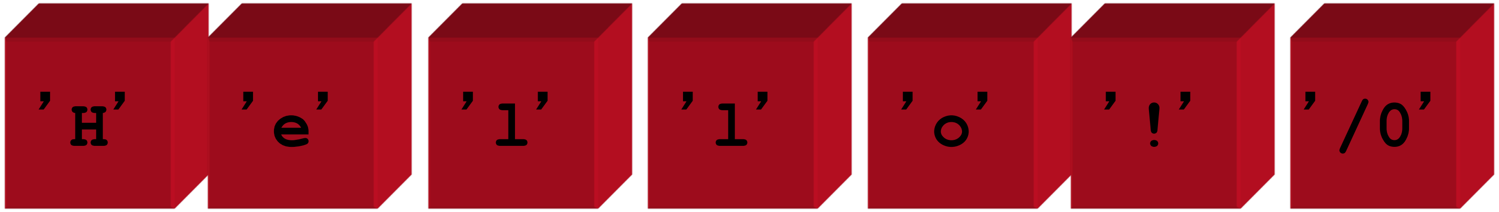
Use of pointer notation to manipulate arrays...

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

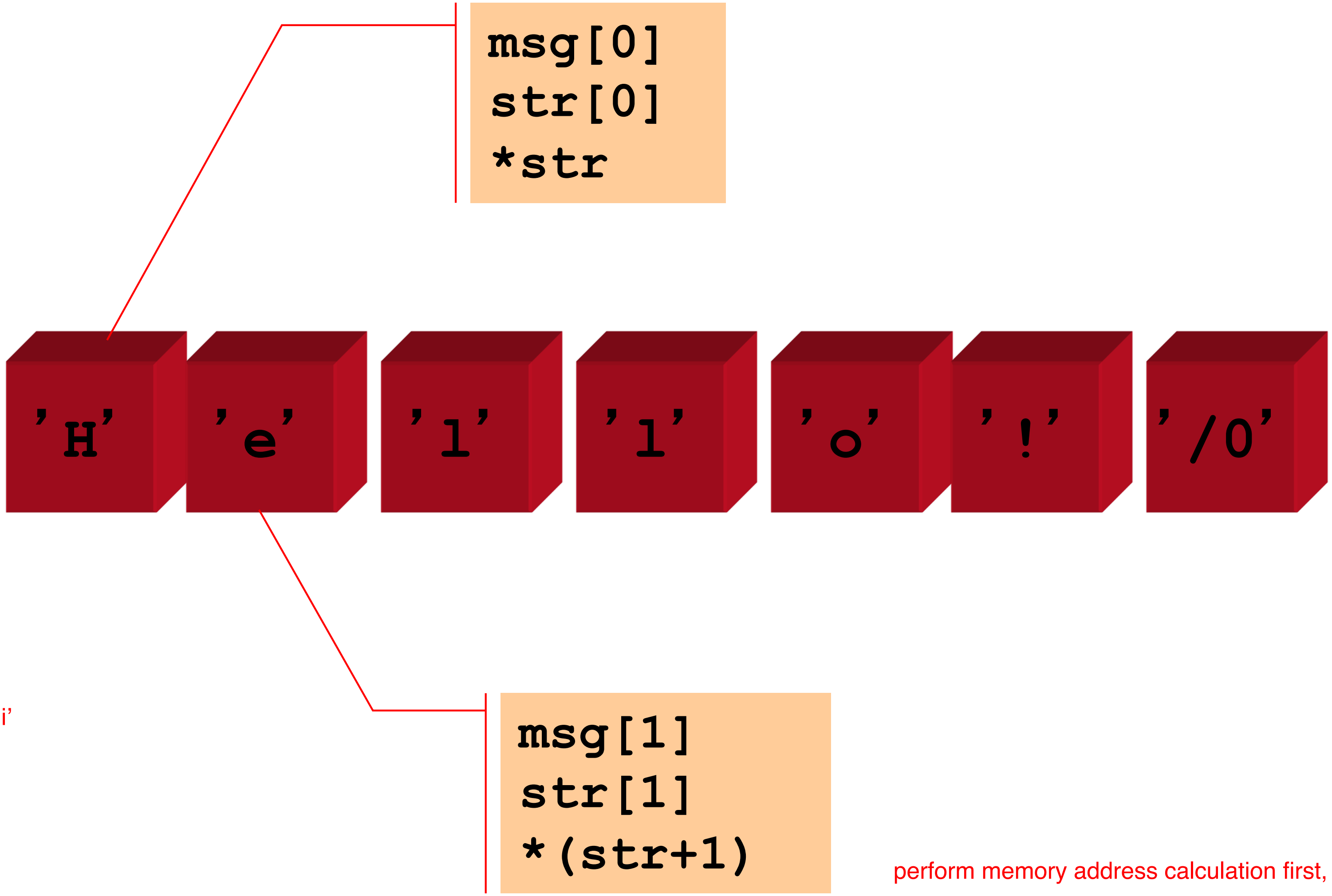
= &(msg[0])
= msg + 0 * sizeof(char)
= &(the value of ox1000) = &(‘H’)
= 0x1000

OR:

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



Examples



msg[0]
str[0]
*str

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

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

*(str+1) = ‘e’
*(str) + 1 = 73 + 1 = 74 = ‘i’

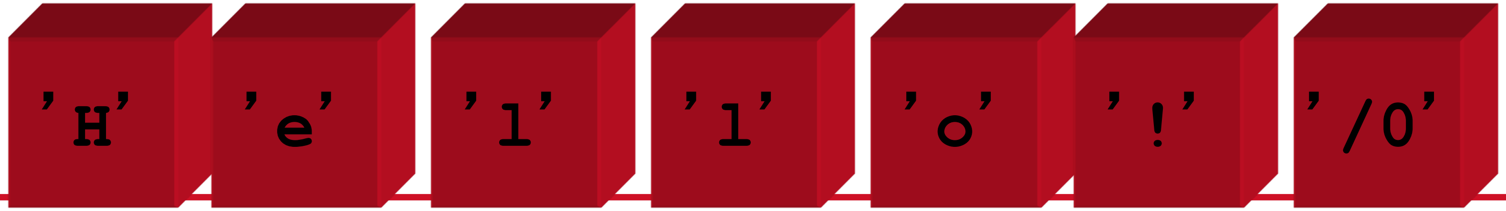
perform memory address calculation first, then dereference

Examples

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

Moving through a string:

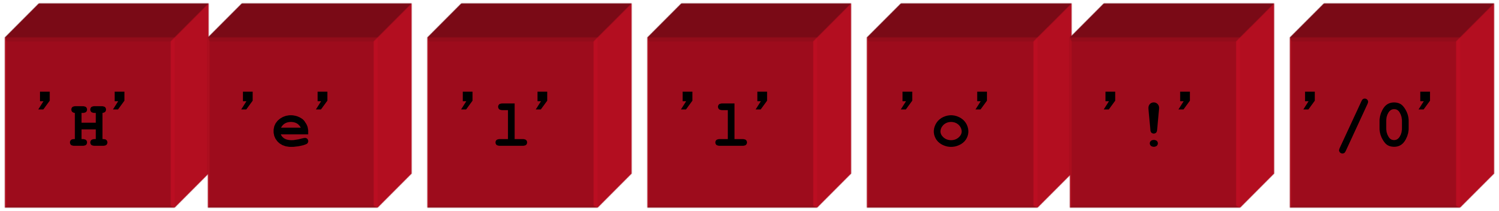
```
while (*str != ’\0’)
    str++;
```



Examples

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

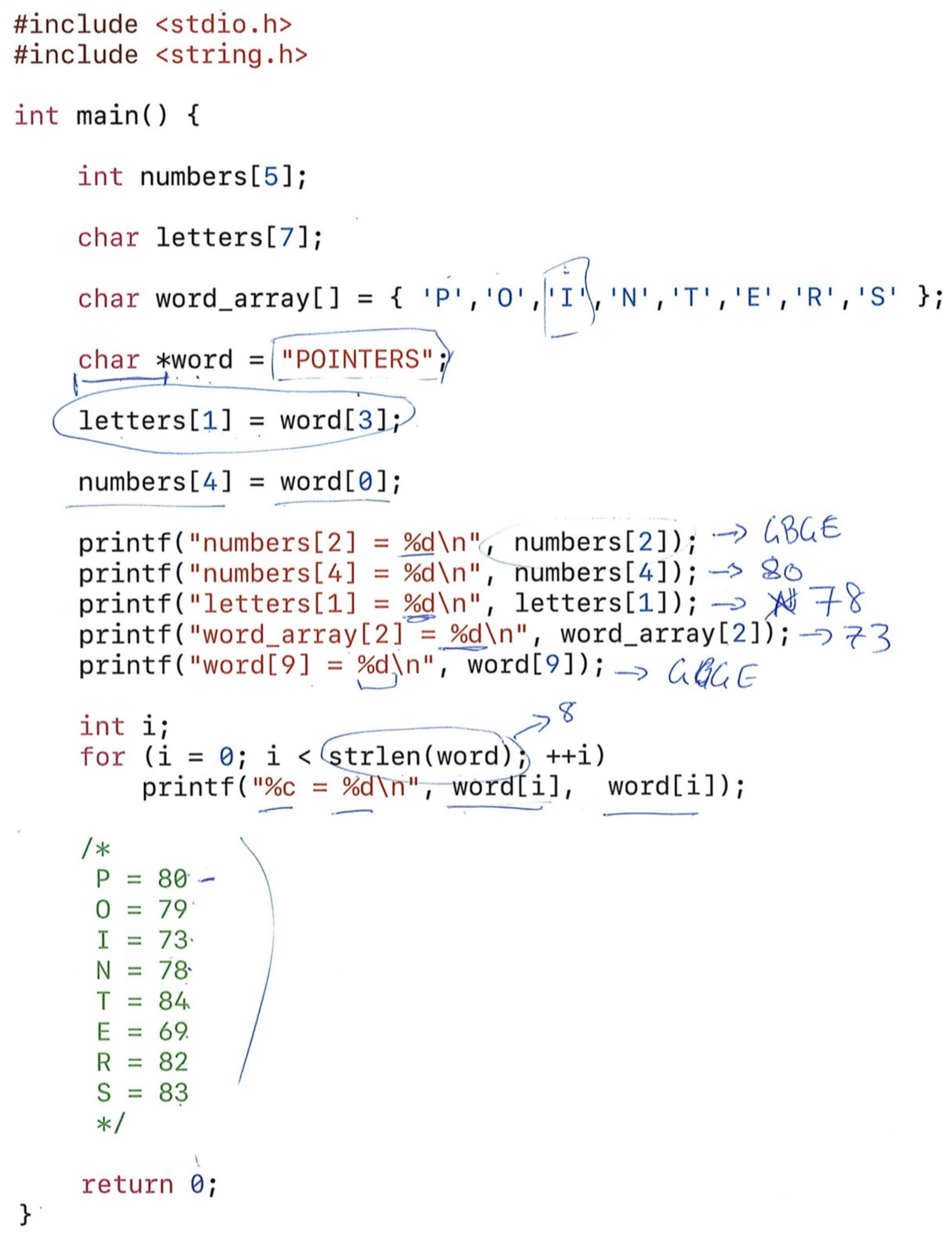
```
while (*str)
    str++;
```



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



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

int main() {

    int numbers[5];

    char letters[7];

    char word_array[] = { 'P','O','I','N','T','E','R','S' };

    char *word = "POINTERS";

    letters[1] = word[3];

    numbers[4] = word[0];

    printf("numbers[2] = %d\n", numbers[2]); → GBGE
    printf("numbers[4] = %d\n", numbers[4]); → 80
    printf("letters[1] = %d\n", letters[1]); → ~~N~~ 78
    printf("word_array[2] = %d\n", word_array[2]); → 73
    printf("word[9] = %d\n", word[9]); → GBGE

    int i;                                  → 8
    for (i = 0; i < strlen(word); ++i)
        printf("%c = %d\n", word[i],  word[i]);

    /*
     P = 80
     O = 79
     I = 73
     N = 78
     T = 84
     E = 69
     R = 82
     S = 83
     */

    return 0;
}
```

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;`

garbage character, 2 or 4 bytes garbage character, 1 byte
memory adress of pointer to int, 8 bytes
memory address of pointer to char, 8 bytes
memory address of pointer to pointer to char, 8 bytes

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 *`

arrays of address/pointers

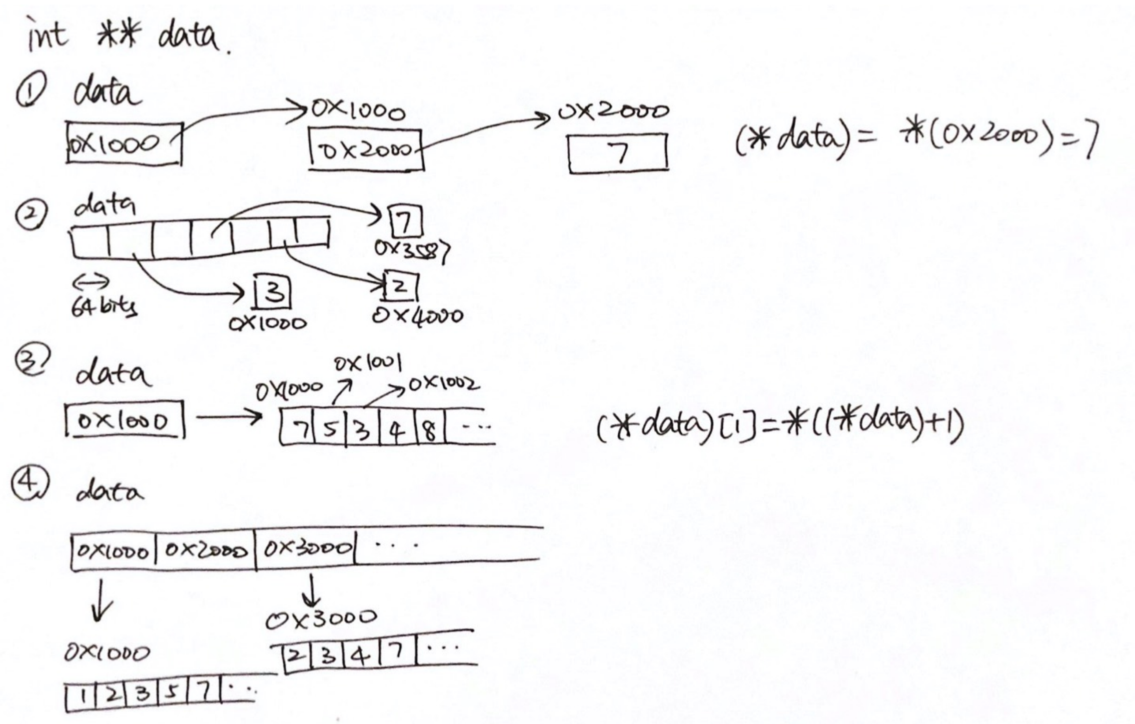
- `char **argv`
  - `*` Address to the first element to an array of type char *
  - Then, each element in `*` is an…
    - `*` address to the first element to an array of type char

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



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

char *word_array[] = {“POINTERS”};

an array of adresses, aka an array of strings
the size is the size of an address memory, which is 64 bits

char **word_array[] = {&word, word_array};
the address of address of char array

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

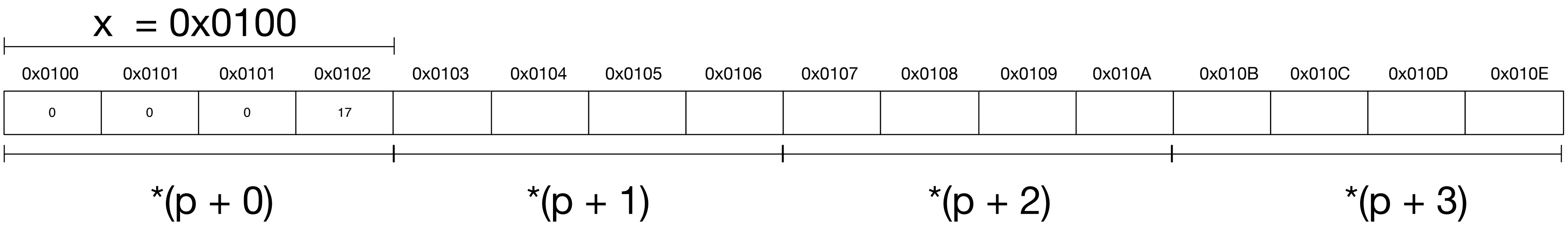
What is the meaning of this in C?

- 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;



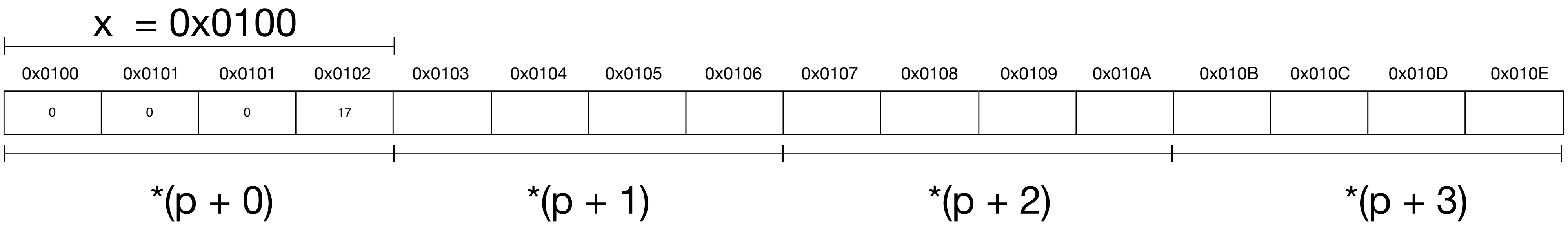
x = 0x0100

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

- 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 )` 1 byte
  - `sizeof( int ), sizeof( double )` at least 4, random
  - `sizeof( float * )` size of address
  - `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];`

1 byte
size of address, 4 or 8 bytes
8 bytes

› But…

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

100 bytes
size of address, 4 bytes or 8 bytes
14 bytes
size of address, 4 bytes or 8 bytes
size of address, 4 bytes or 8 bytes

› **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;`**

Less familiar `const`

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

const means, this particular variable is read only

- 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

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

Non-writable

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 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?