Pointers

COMP2129

FACULTY OF
ENGINEERING &
INFORMATION
TECHNOLOGIES

THE UNIVERSITY OF
SYDNEY

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:

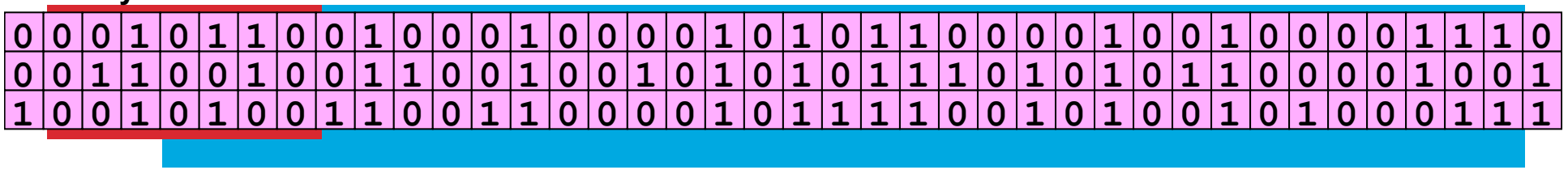
```
int counter;
```

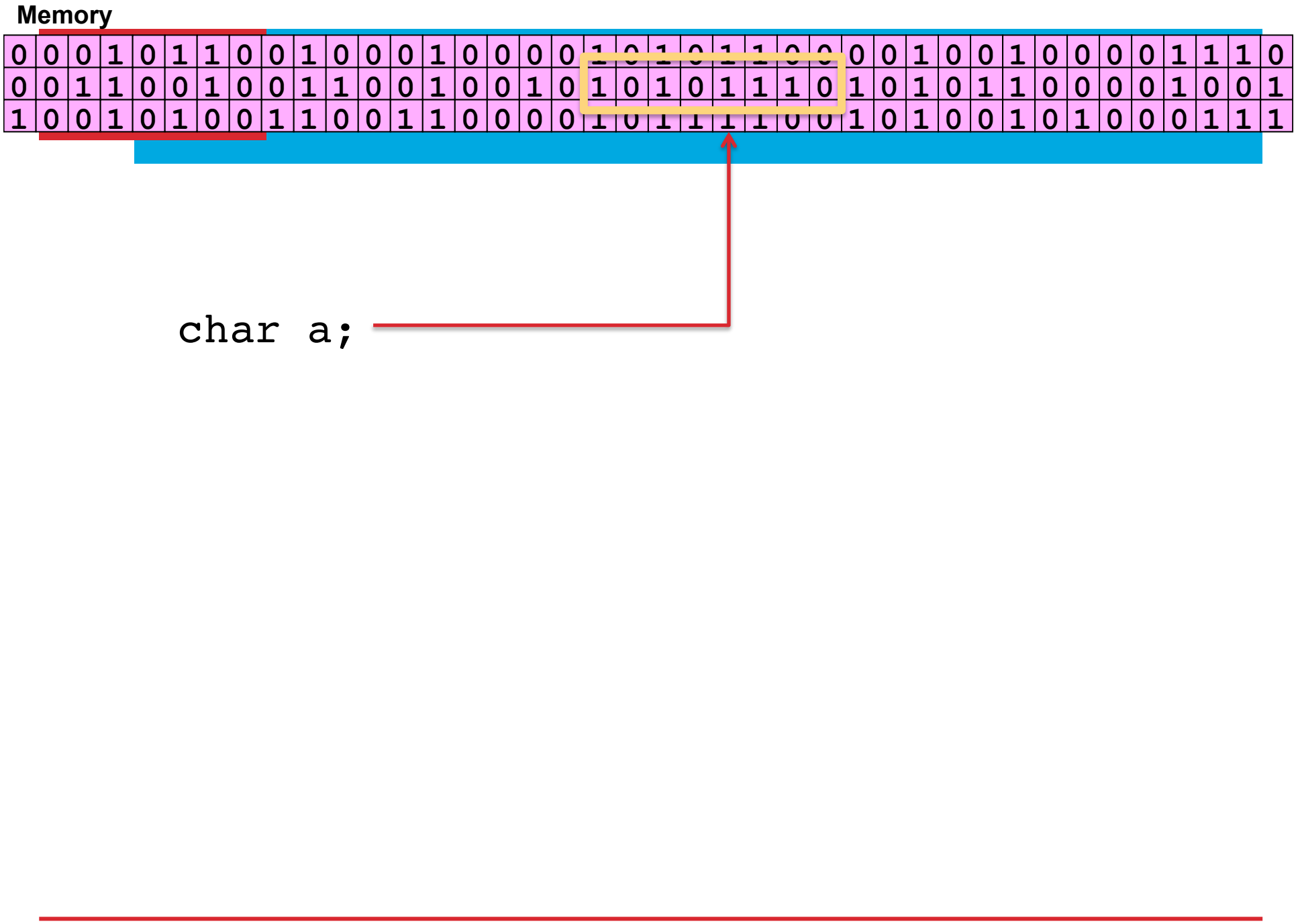
reserve memory for an integer and call it "counter"

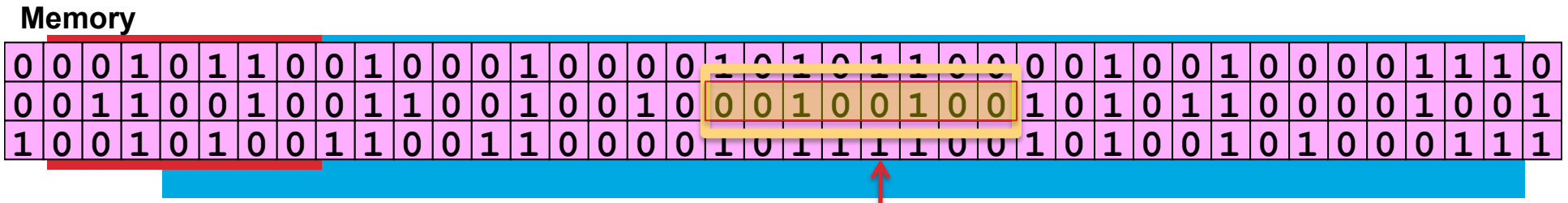
- *Initialisation* or *assignment* specifies content:

```
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;


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

```
char a;
a = '$';
```

Arrays

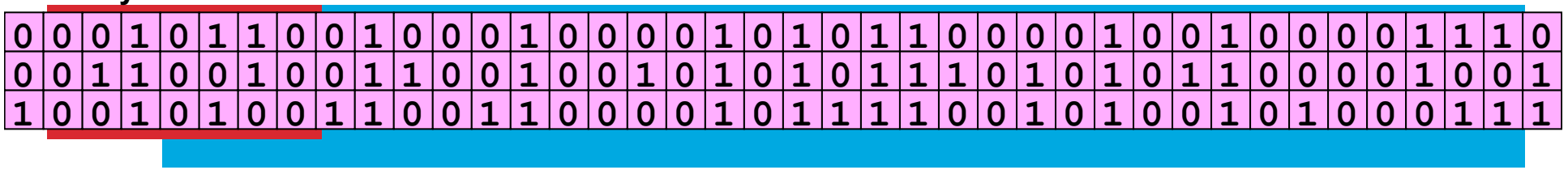
- Arrays are indexed collections of the same type
- Declaration of an array:

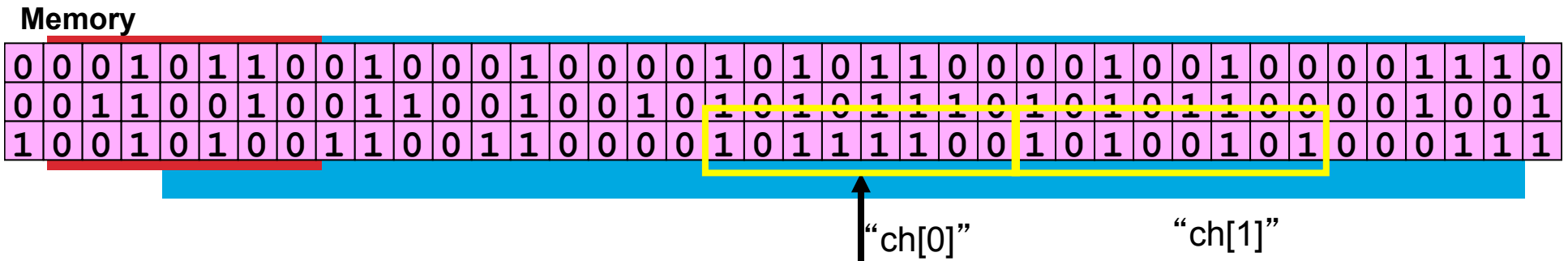
```
int  counters[MAX];
char alphabet[26];
```

- Initialisation of an array:

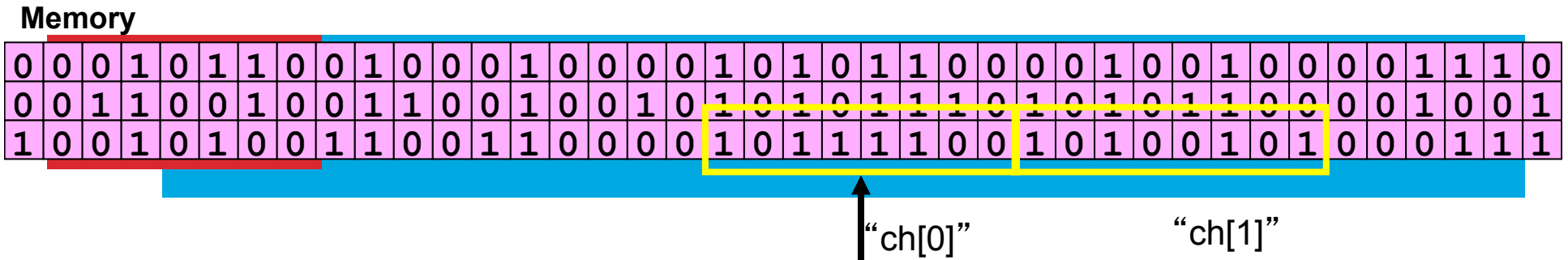
```
for (i = 0; i < MAX; i++)
    counters[i] = i;
```

Memory

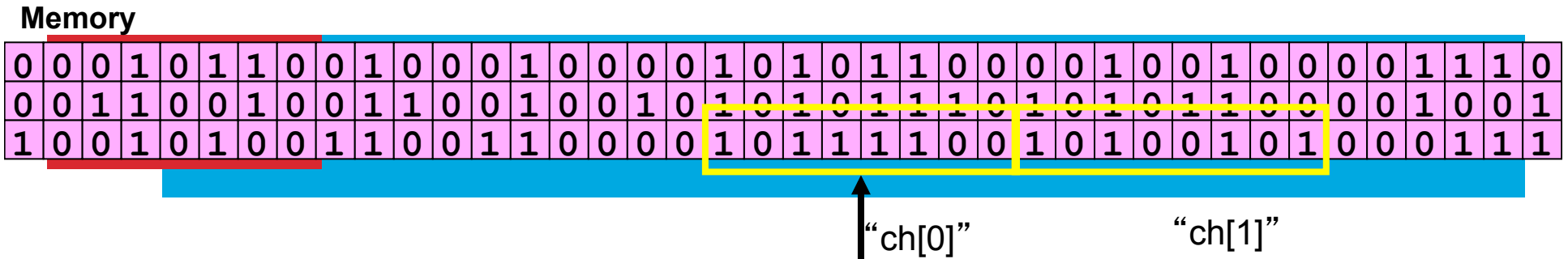

0 0 0 1 0 1 1 0 0 1 0 0 0 1 0 0 0 0 1 0 1 0 1 1 0 0 0 0 1 0 0 1 0 0 0 0 1 1 1 0
0 0 1 1 0 0 1 0 0 1 1 0 0 1 0 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 1 0 0 0 0 1 0 0 1
1 0 0 1 0 1 0 0 1 1 0 0 1 1 0 0 0 0 1 0 1 1 1 1 0 0 1 0 1 0 0 1 0 1 0 0 0 1 1 1


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];
```

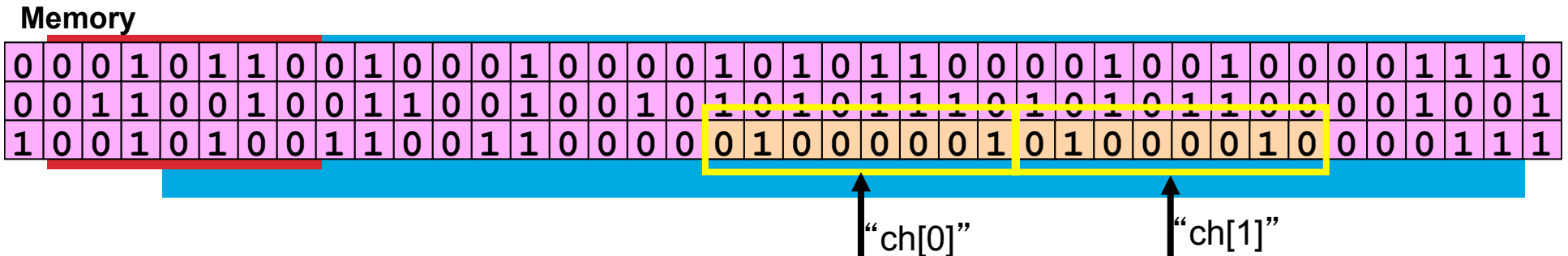



```
char ch[2];
printf("%c\n", ch[1]);
```




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

Output of random data

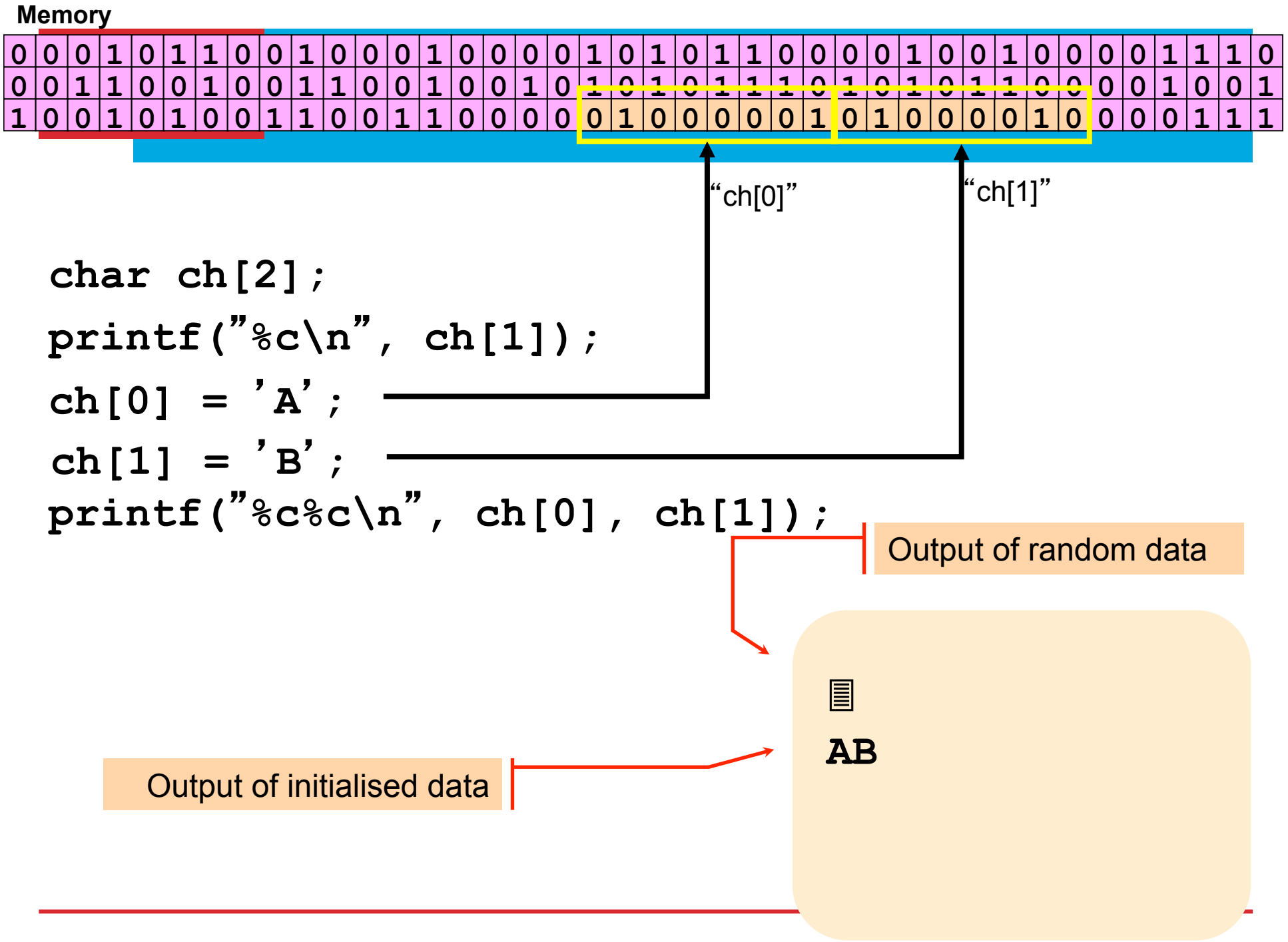


"ch[0]"

"ch[1]"

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

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 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 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';
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:

```
char myHobby[] = ”rowing”;
```

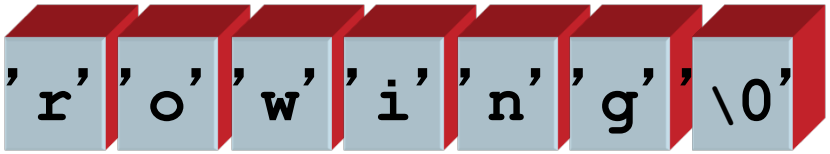
The compiler can determine the required size by counting characters, so the array size is **optional**. A larger size *may* be specified.

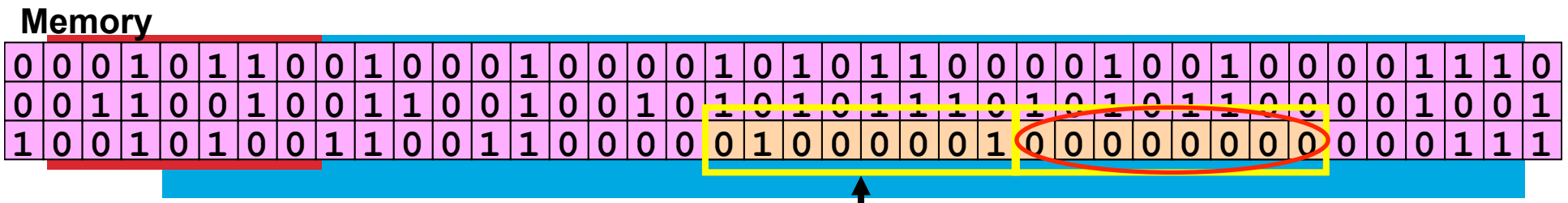
Strings

- 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";
```





"str"

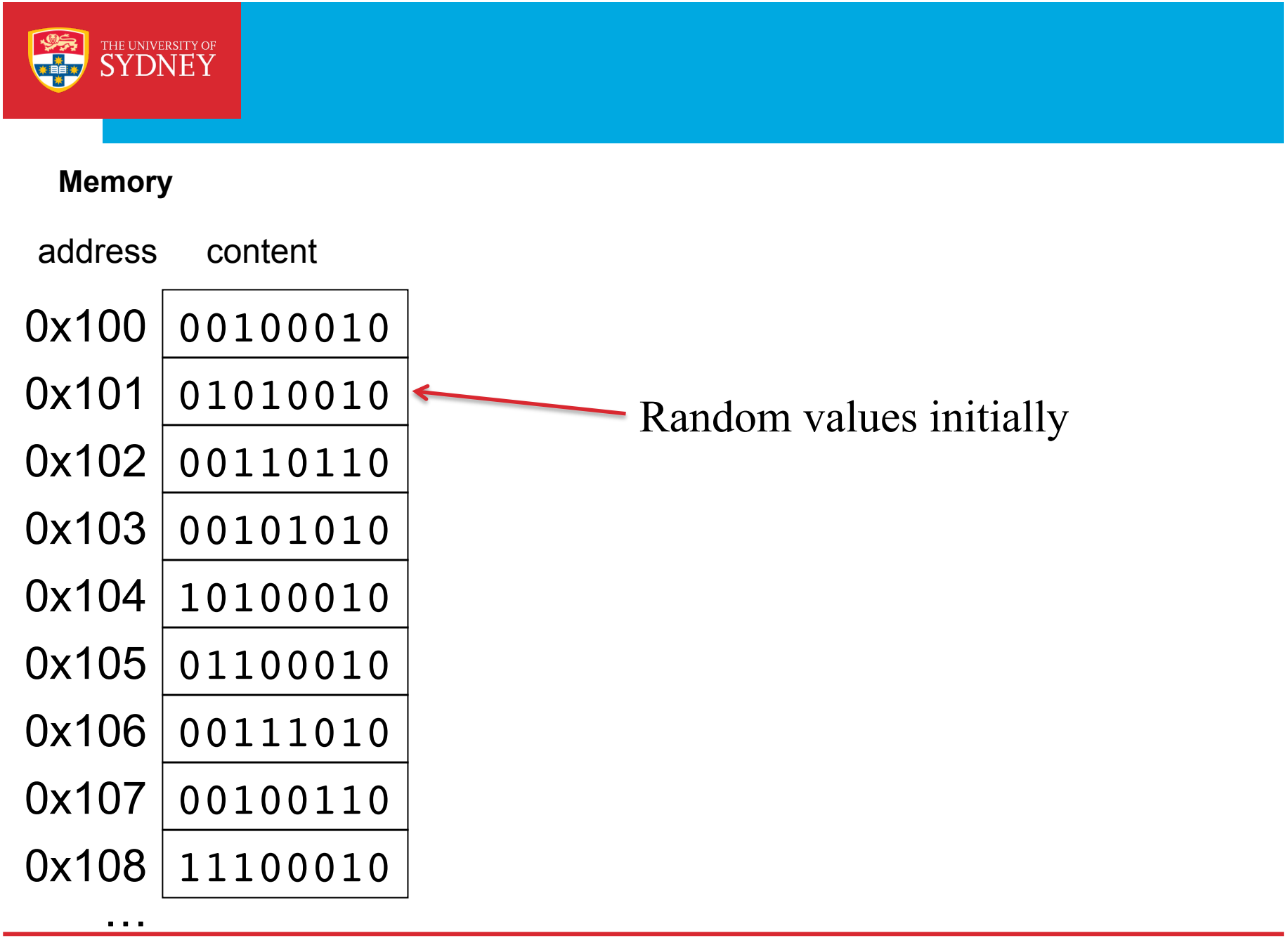
```
char str[] = "A";

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
...	


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
...
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

Memory

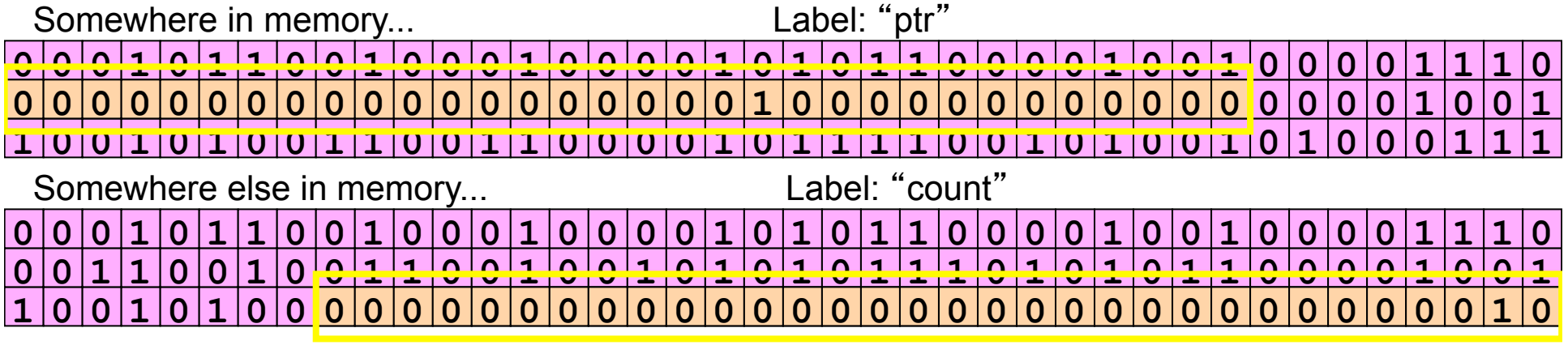
address	content
0x100	00100010
0x101	01010010
0x102	00110110
0x103	00101010
0x104	10100010
0x105	01100010
0x106	00111010
0x107	00100110
0x108	11100010
...	

```
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

THE UNIVERSITY OF SYDNEY

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"

THE UNIVERSITY OF SYDNEY

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)

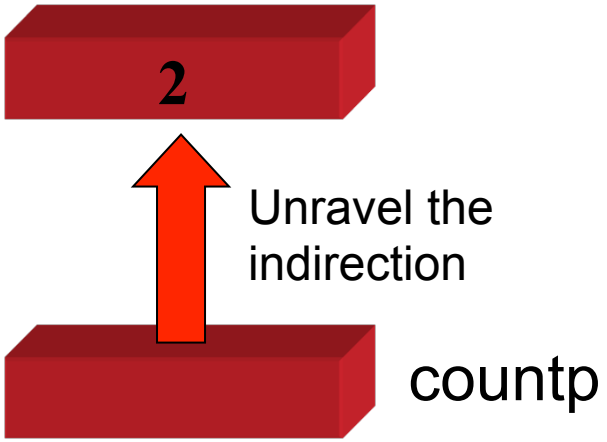
- The indirection operator, ‘*’, is used to “unravel” the indirection:

`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);
```

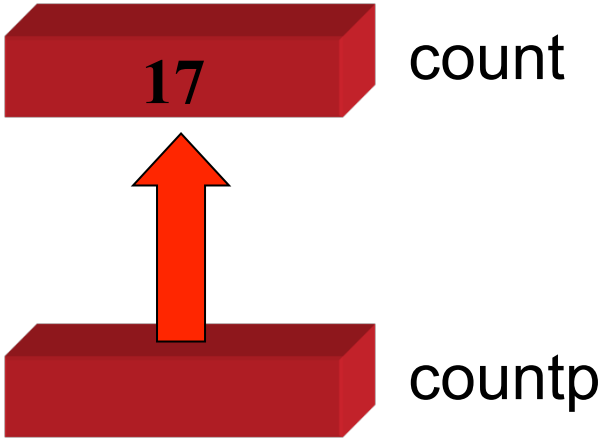
17

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

17

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

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

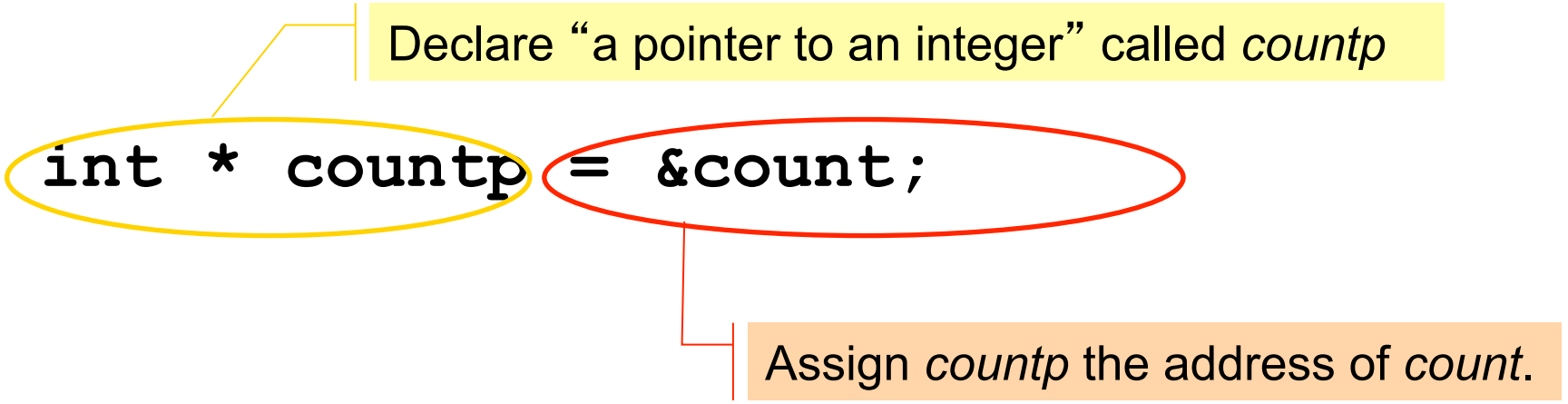


THE UNIVERSITY OF SYDNEY

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:



Examples

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

Receiving an integer from standard input:

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

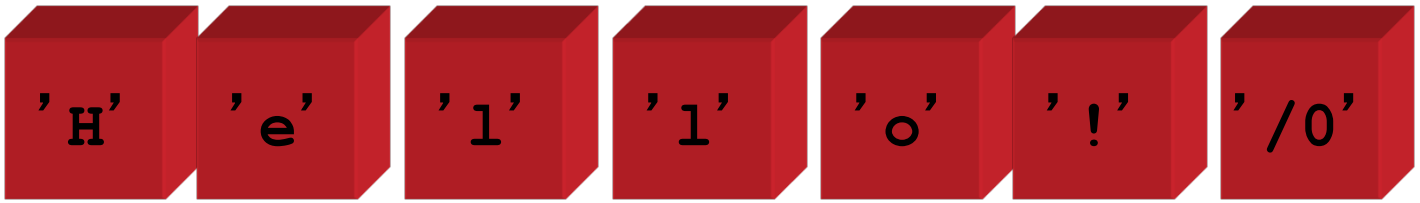
This argument is required by *scanf()* to be a pointer. Since we are using a simple integer, *age*, we pass it's address.

Use of pointer notation to manipulate arrays...

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

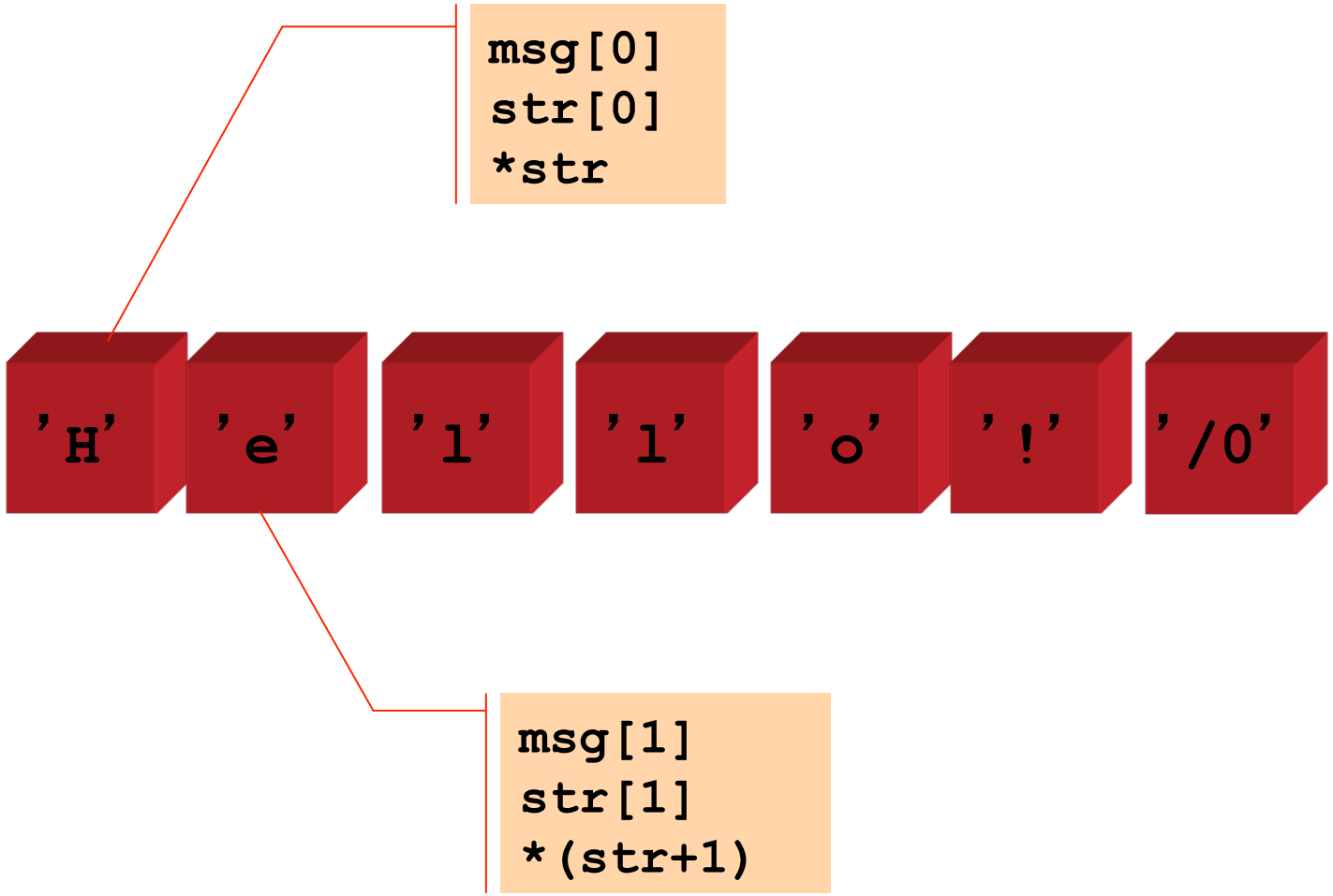
OR:

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



THE UNIVERSITY OF SYDNEY

Examples

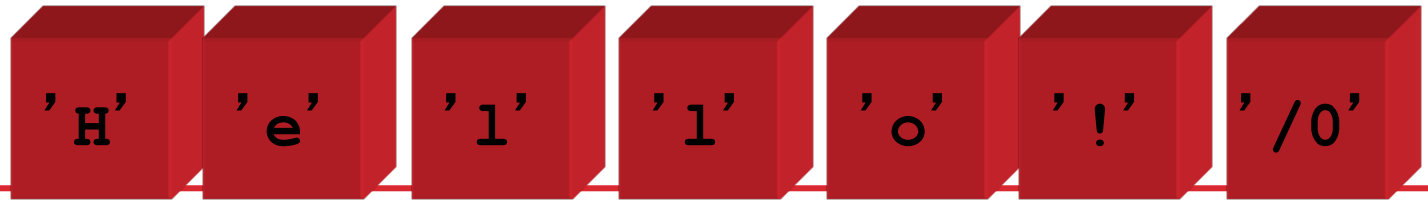


Examples

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

Moving through a string:

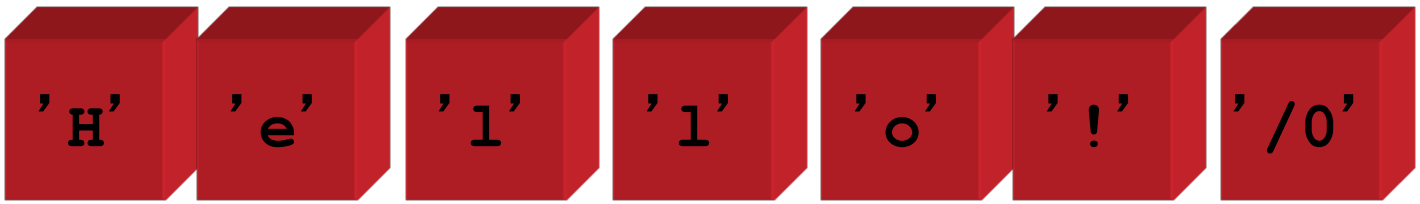
```
while (*str != '\0')
    str++;
```



Examples

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

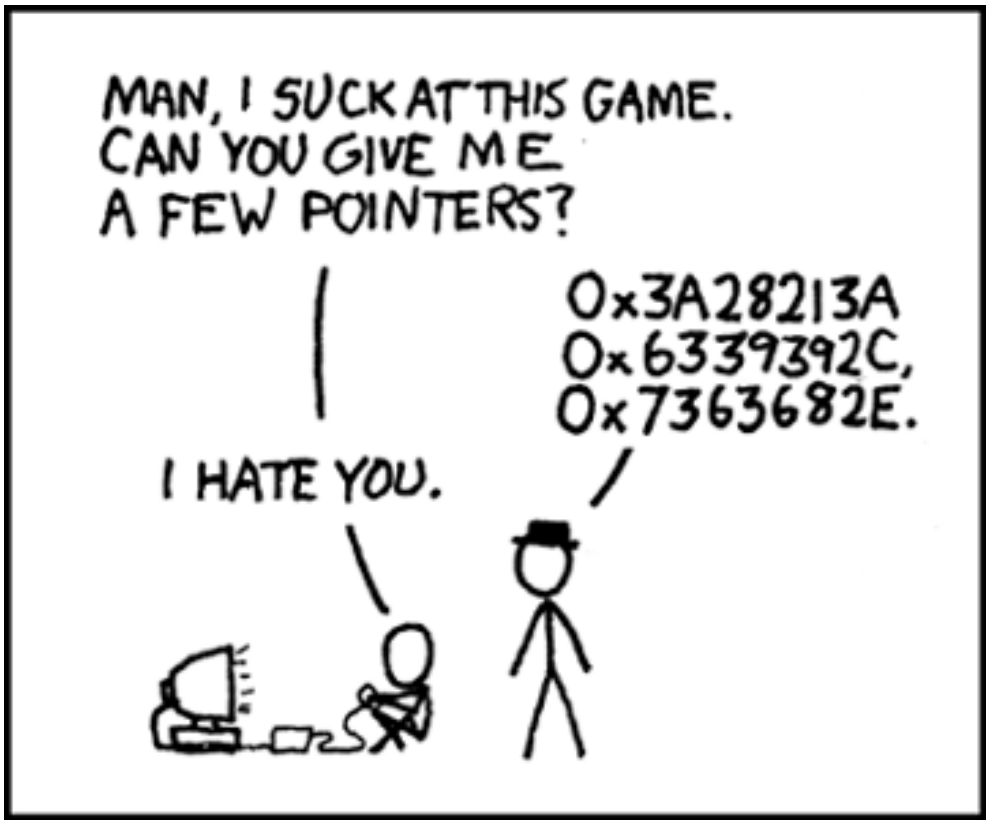
```
while (*str)
    str++;
```



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.