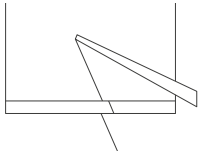
Bring ideas to life
VIA University College



C Data Types, Memory Management, Pointers and Arrays

ESW1

At the end of this session, you should

- Be able to explain the basic data types of C
- Be familiar with various mathematical operators in C
- Be able to implement a string in C using character arrays
- Understand variables and declarations
- Understand pointers and arrays
- Be able to create functions that
  - Uses *pass by value*
  - Uses *pass by reference*

C Integer Data Types

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

Can you see a problem here?

Standard Integer Definitions

`stdint.h` Defines standard integers with fixed data sizes

E.g.

Specifier	Signing	Bits	Bytes	Minimum Value	Maximum Value
int8_t	Signed	8	1	-2^7 which equals -128	$2^7 - 1$ which is equal to 127
uint8_t	Unsigned	8	1	0	$2^8 - 1$ which equals 255
int16_t	Signed	16	2	-2^{15} which equals -32,768	$2^{15} - 1$ which equals 32,767
uint16_t	Unsigned	16	2	0	$2^{16} - 1$ which equals 65,535
int32_t	Signed	32	4	-2^{31} which equals -2,147,483,648	$2^{31} - 1$ which equals 2,147,483,647
uint32_t	Unsigned	32	4	0	$2^{32} - 1$ which equals 4,294,967,295
int64_t	Signed	64	8	-2^{63} which equals -9,223,372,036,854,775,808	$2^{63} - 1$ which equals 9,223,372,036,854,775,807
uint64_t	Unsigned	64	8	0	$2^{64} - 1$ which equals 18,446,744,073,709,551,615
intptr_t				integer type capable of holding a pointer	

A number of constants are defined:

E.g. INT8_MAX, INT8_MIN, INT64_MAX, INT64_MIN etc.

Using standard integer definitions is good habit in C – sets the programmer in 100% control

C Floating-Point Types

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

For small CPUs without Floating Point Units (FPU) it is hard/expensive to work in floating point

Defining Constants

What is the difference?

- `#define` handled by pre-processor
- `const` handled by compiler (Type safe)
- `enum` can be used to show what is possible as parameters

```
#define LENGTH 10
#define WIDTH 5
#define NEWLINE '\n'

OR

int main()
{
        const int LENGTH = 10;
        const int WIDTH = 5;
        const char NEWLINE = '\n';

OR

typedef enum {
        RED, GREEN, BLUE=10
} COLOR;

void set_color(COLOR cl) {
```

Escape Sequences in C-Strings

You probably know them from Java

Escape sequence	Meaning	Escape sequence	Meaning
\\	\ character	\n	Newline
\'	' character	\r	Carriage return
\"	" character	\t	Horizontal tab
\?	? character	\v	Vertical tab
\a	Alert or bell	\ooo	Octal number of one to three digits
\b	Backspace	\xhh . . .	Hexadecimal number of one or more digits

```
char some_text[] = “This is a text\ndivided in two lines with \”quotation marks\””
```

Arithmetic operators

You know them from Java

Operator	Description	Example (A=10, B=20)
+	Adds two operands	A + B yields 30
-	Subtracts second operand from the first	A - B yields -10
*	Multiply both operands	A * B yields 200
/	Divide numerator by de-numerator	B / A yields 2
%	Modulus Operator and remainder of after an integer division	B % A yields 0
++	Increment operator increases integer value by one	A++ yields 11
--	Decrement operator decreases integer value by one	A-- yields 9

Arithmetic operators (Bitwise operators)

You know **most of** them from Java

Operator	Description	Example
~	Bitwise NOT (Ones Complement)	~a the bitwise NOT of a
&	Bitwise AND	a & b the bitwise AND of a and b
|	Bitwise OR	a | b the bitwise OR of a and b
^	Bitwise XOR	a ^ b the bitwise XOR of a and b
<<	Bitwise left shift	a << 4 a bits shifted 4 times to the left
>>	Bitwise right shift	a >> 4 a bits shifted 4 times to the right

Very much used in drivers and embedded programming

Question?

What is the difference of ++x and x++ ??

```
int a = 10, b = 20;

printf("What value has a?: %d\n", a++);
printf("What value has a now?: %d\n", a);

printf("What value has b?: %d\n", ++b);
printf("What value has b now?: %d\n", b);
```

What is the output of this program snippet?

Relational operators

You know them from Java

Operator	Description	Example (A=10, B=20)
==	Checks if the value of two operands is equal or not, if yes then condition becomes true.	(A == B) is not true
!=	Checks if the value of two operands is equal or not, if values are not equal then condition becomes true.	(A != B) is true
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true

Logical Operators

You know them from Java

Operator	Description	Example (A=1, B=0)
&&	Called Logical AND operator. If both the operands are non zero then condition becomes true.	(A && B) is false
||	Called Logical OR Operator. If any of the two operands is non zero then condition becomes true.	(A || B) is true
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true

Operator precedence rules

Observe the operator precedence rules
Precedence (example): *, /, %, +, -
You can change evaluation sequence using parenthesis

6+ 4/2 = 8
(6+4)/2 = 5

- It's a good habit to use parentheses

Variables and declarations

Variables **must** be declared before used

```
int i, j, k;
char c, ch;
float f, salary;
double d;
```

Variables **must** be initialised

```
i = j = k = 27;
f = 2.0;
```

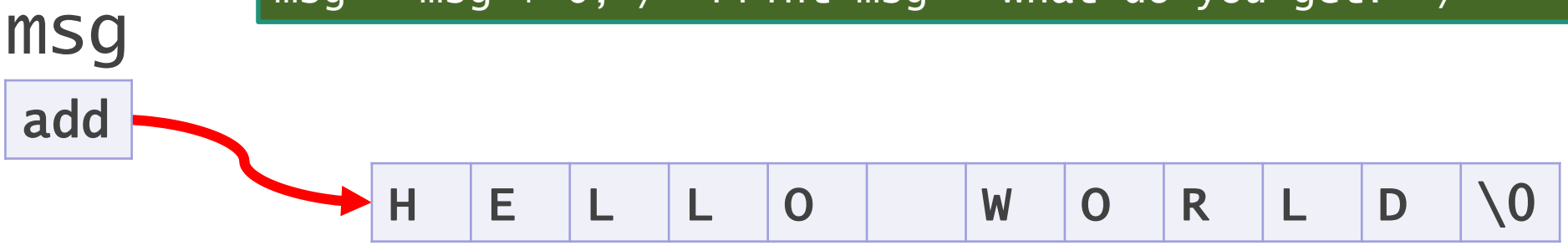
Declaration and initialisation example

```
int i = 27;
int k = 14, j = i;
```

Character arrays and pointers

C does NOT have a native string class/object!
Use arrays of characters instead

```
char text[] = ”Hello world”;
char* msg = text;   /* Print msg – what do you get? */
msg = msg + 6; /* Print msg – what do you get? */
```



Exercises

- Exercise 3-1: Basic data types and operations
- Exercise 3-2: Understanding character arrays

Pointers in C

Program

```
int j;
int* k;

j = 17;
++j;

k = &j;
*k = *k + 10;
```

Result

j can hold an integer
k can hold an address of a memory location where an integer variable is stored

now j is 17
now j is 18

k now "points" to j (&j takes the address of j)
and now *k (which is the same as j) is 28
(*k is the variable/memory location k points at)

A closer look at pointers

int a, b;

0x1000	a	?
0x1004	b	?

int* c, * d;

0x1008	c	?
0x100C	d	?

0x1000	a	?
0x1004	b	?

a = 10;
b = 20;

0x1008	c	?
0x100C	d	?

0x1000	a	10
0x1004	b	20

c = &a;
d = &b;

0x1008	c	0x1000
0x100C	d	0x1004

0x1000	a	10
0x1004	b	20

*c = *c + 5;

0x1008	c	0x1000
0x100C	d	0x1004

0x1000	a	15
0x1004	b	20

d = c;

0x1008	c	0x1000
0x100C	d	0x1000

0x1000	a	15
0x1004	b	20

*d = *d + 10;

0x1008	c	0x1000
0x100C	d	0x1000

0x1000	a	25
0x1004	b	20

Call by value, call by reference

Call by value

```
void swap(int x, int y)
{
   int temp;
   temp = x;
   x = y;
   y = temp;
}

int main () {
   int a = 5; int b = 3;
   swap (a,b);
   printf("a=%d and b=%d\n", a, b);
   …
}
```

What is printed out?

Call by reference

```
void swap(int* x, int* y)
{
   int temp;
   temp = *x;
   *x = *y;
   *y = temp;
}

int main () {
   int a = 5; int b = 3;
   swap (&a, &b);
   printf("a=%d and b=%d\n", a, b);
   …
}
```

What is printed out?

Pointers and arrays

Program

```
int a[10];

int* pa;

pa = a;

pa = a + 4;

*(pa + 2) = 27;
```

Result

- **a** is an array of 10 ints
- **pa** is a pointer to an int
- **pa** now points to **a** (first element of **a**)
- **pa** now points to the fifth element (**a[4]**) of **a**
- **a[6]** now holds the value 27

Address/pointer arithmetics

Pointers can be incremented and decremented

- If incremented, it points to the next element (depends on data type)
- C will move the pointer to the **correct** next location (alignment)

```
char msg[] = ”Hello”;
++msg;
++msg;
```

Address	Data
4990	‘H’
4991	‘e’
4992	‘l’
4993	‘l’
4994	‘o’
4995	‘\0’
4996	
4997	
4998	

```
int nos[] = {128, 255};
int* p = nos;
printf("%d\n", *p);
p++;
printf("%d\n", *p);
```

Address	Data
4990	00
4991	128
4992	00
4993	255
4994	
4995	
4996	
4997	
4998	

What is printed out?

Exercises

- Exercise 3-3: Functions, pass by value/reference
- Exercise 3-4: More on character arrays