1.20 Basic expressions

Expressions with arithmetic operators, compound assignment, increment/decrement operators, and the sizeof operator

Literal value formats

Integral value rollover

Some implicit type conversions

EE108 – Computing for Engineers

1

Overview

Aims

- Revise basic expressions and ensure proficiency with the main arithmetic related operators

Learning outcomes – you should be able to…

- Use arithmetic operators
- Use compound assignment operators
- Use the prefix/postfix increment/decrement operators
- Recognise and write the various forms of literal integer and floating point values
- Explain integral value roll over
- Use the sizeof operator
- Identify some implicit type conversions and and explain the consequences of those conversions

2

Expressions

Remember…

An expression is sequence of operators and operands that specifies a computation yielding a value

Basic arithmetic operators

Arithmetic operators (two operands)	Example
`+   addition`	`j + k; // add j and k`
`-   subtraction`	`j – 2; // subtract 2 from j`
`*   Multiplication`	`j * k; // multiply j * k`
`/   division`	`j / 2; // divide j by 2`
`%   remainder`	`j % 10; // remainder of j divided by 10`
Unary operators (one operand)	**Example**
`+   unary plus (does nothing useful)`	`+3; // +3 is the same as 3`

`-   unary minus (negates the operand)`	`-3; // minus 3`
	`-j; // the negative of j`

Note: whitespace between operators and operands is ignored by the compiler. In particular the unary + or - can have whitespace before the operand (unlike the examples above).

Use appropriate whitespace between operators to make code more human-readable.

Generally, use whitespace before and after two operand operators.

Don't put whitespace between unary plus or minus and their operand

Basic arithmetic and assignment

Only variables (or the declaration of const constants) can appear on the left hand side of the assignment (=) operator

Expressions on the right hand side can have any mix of variables, constants, literals, functions which return a value, or nested expressions

A variable can appear on both the right and left hand side of the assignment operator. First, the expression on the right is always calculated based on the current value of the variable. Then the result is assigned to the variable on the left as the very last step.

```
// assignment and basic expression examples
…
i = 7;
j = 3;
k = 2;

i = i - -j;              // value of i?
k = k * 2;               // value of k?
i = i * -j + k % 5;      // value of i?
```

Operator precedence

With i=1, j=2, and k=3 what is the value of the following expression? How did you perform the calculation (step by step)?

```
i = i * -j + k % 5;      // value of i?
```

When an expression has several operators, there are two ways to determine the order of calculation: add parentheses, (and), or rely on the precedence rules of C.

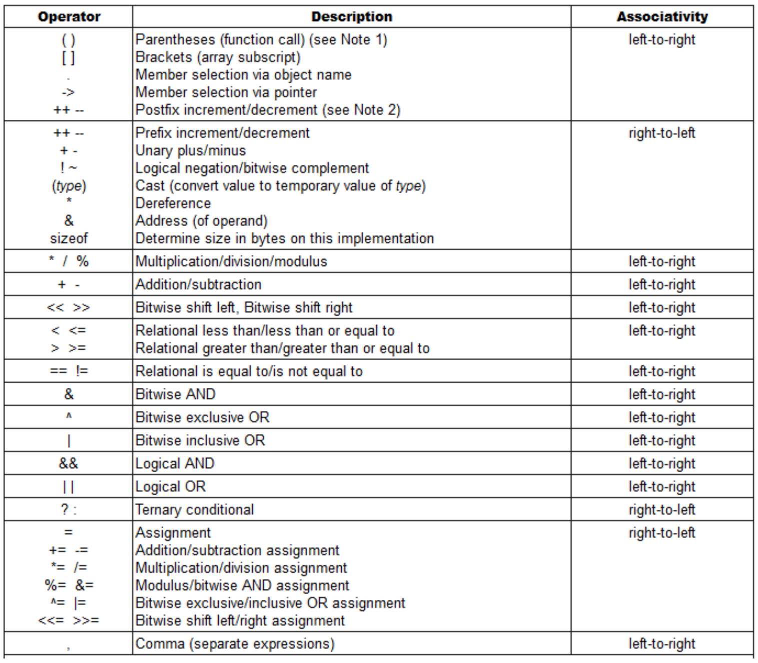
When parentheses are used, the parenthesized expressions are evaluated first (from innermost to outermost).

The precedence rules of C state that arithmetic operators are evaluated in the following order: (1) unary plus/minus; (2) multiply, divide, and remainder; (3) add and substract. (see next slide for further details.)

```
i * -j + k % 5         is equivalent to      (i * (-j)) + (k % 5)
```

Use parentheses for clarity or to specify non-default precedence of operators, e.g.

```
i * (-j + k) % 5         is equivalent to      (i * ((-j) + k)) % 5
```



Operator	Description	Associativity
( )	Parentheses (function call) (see Note 1)	left-to-right
[ ]	Brackets (array subscript)	
.	Member selection via object name	
->	Member selection via pointer	
++ --	Postfix increment/decrement (see Note 2)	
++ --	Prefix increment/decrement	right-to-left
+ -	Unary plus/minus	
! ~	Logical negation/bitwise complement	
(*type*)	Cast (convert value to temporary value of *type*)	
*	Dereference	
&	Address (of operand)	
sizeof	Determine size in bytes on this implementation	
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <=	Relational less than/less than or equal to	left-to-right
> >=	Relational greater than/greater than or equal to	
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
=	Assignment	right-to-left
+= -=	Addition/subtraction assignment	
*= /=	Multiplication/division assignment	
%= &=	Modulus/bitwise AND assignment	
^= |=	Bitwise exclusive/inclusive OR assignment	
<<= >>=	Bitwise shift left/right assignment	
,	Comma (separate expressions)	left-to-right

Associativity determines the order in which the equal precedence operators in each row are applied

Self test questions

Q. Add parentheses to show how C would interpret the following expressions

*a * b – c * d + e*

a / b % c / d

- a – b + c - + d

*a * - b / c - d*

More about mod %

In C, % is called the mod operator

- Despite the name, the operation it calculates is not the mod but the integral remainder
  - x % y is the remainder of x divided by y
- Note: The remainder of x divided by y always has the same sign as x

```
k = -8;
i = k % 5;       // what is the value of i?
```

The % operator only works with integral values

- To calculate the remainder of floating point values you must use the **fmod** function and include <math.h>
- We won't need fmod in this module – so just be aware of it

Compound assignment

Assignment statements of the following form are common in C programmes

```
pos = pos + 2; // the same variable appears on the left and right
x = x * 2;
```

Compound assignment is a shorthand for such statements. The compound assignments and their general meaning is as follows

`v += expr`	*means*	`v = v + (expr)`
`v -= expr`	*means*	`v = v - (expr)`
`v *= expr`	*means*	`v = v * (expr)`
`v /= expr`	*means*	`v = v / (expr)`
`v %= expr`	*means*	`v = v % (expr)`

Note: the **(** and **)** around expr. It means expr is evaluated first.

```
int x = 20, i = 1;

i += 2;           // i = i + (2);
i += 2 * x;       // i = i + (2 * x);
x *= i + 1;       // x = x * (i + 1)
```

Note: In some cases (where v is itself an expression with side effects), the general meaning in the table above is not quite accurate (e.g. for the increment/decrement operators on the next slide)

Self test questions

11

Q1. Rewrite the following without using compound assignment. Use parentheses if necessary to ensure the result is identical?

```
x += 1;

i *= i;

x %= x + 1;

x /= i * 10;

x -= -4;
```

27 September 2020

Prefix and postfix increment and decrement

12

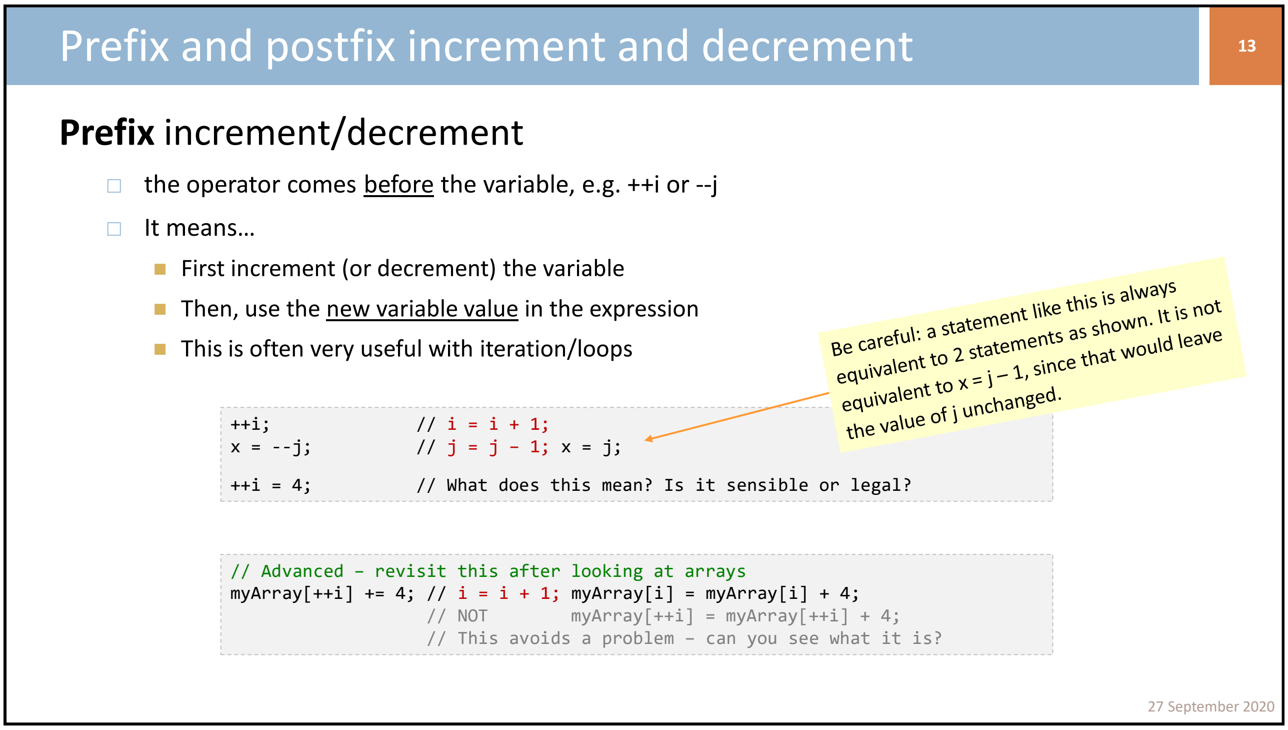
In repeating (loop) code it is very common to increment or decrement a variable by 1, e.g. i = i + 1

A shorthand for this is to use one of the prefix and postfix increment (++) and decrement (--) operators

We will explore the different purpose of the prefix and postfix increment carefully, but in summary:

- Prefix increment of a variable means, first increment the variable, then use the variable's (now incremented value) in the expression
- Postfix increment means, use the current (unchanged) value of the variable in the expression, and then increment it before the next statement is executed
- Prefix and postfix decrement work similarly but decrementing instead of incrementing

27 September 2020



Prefix and postfix increment and decrement

13

Prefix increment/decrement

- the operator comes before the variable, e.g. ++i or --j
- It means…
  - First increment (or decrement) the variable
  - Then, use the new variable value in the expression
  - This is often very useful with iteration/loops

Be careful: a statement like this is always equivalent to 2 statements as shown. It is not equivalent to x = j – 1, since that would leave the value of j unchanged.

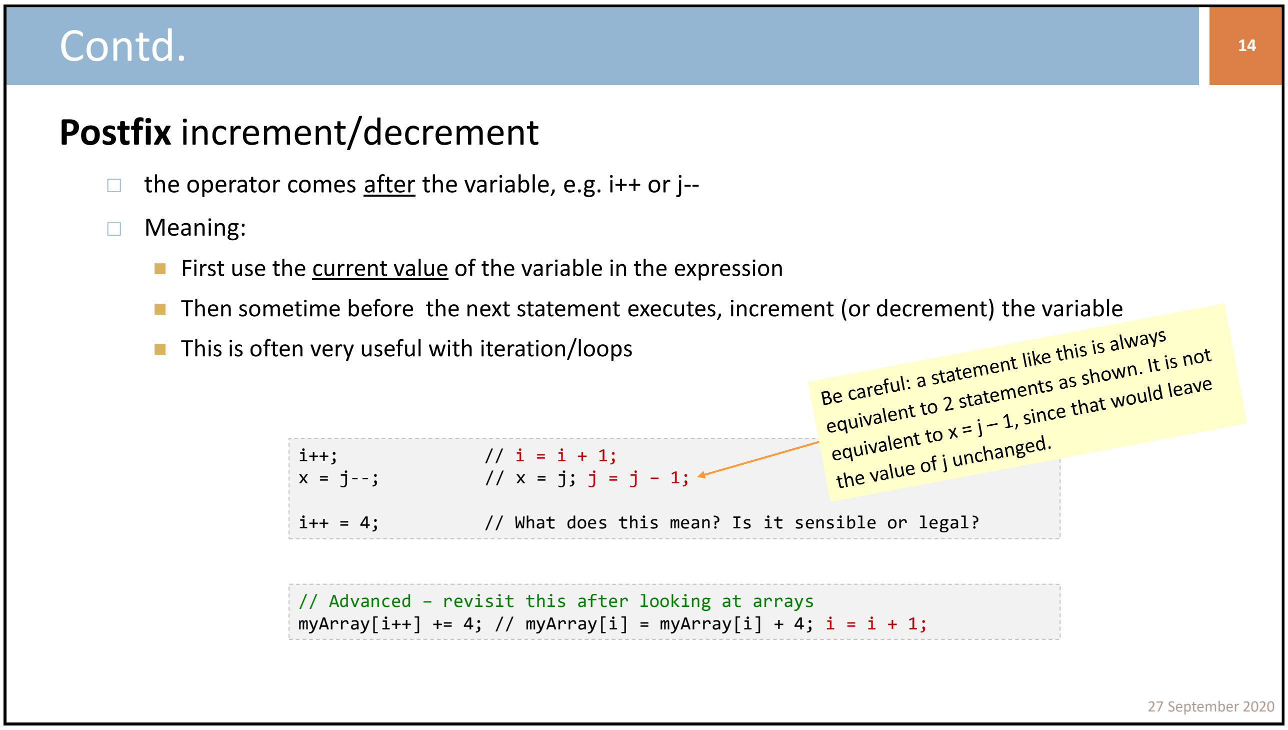
```
++i;             // i = i + 1;
x = --j;         // j = j - 1; x = j;

++i = 4;         // What does this mean? Is it sensible or legal?
```

```
// Advanced - revisit this after looking at arrays
myArray[++i] += 4; // i = i + 1; myArray[i] = myArray[i] + 4;
                   // NOT        myArray[++i] = myArray[++i] + 4;
                   // This avoids a problem - can you see what it is?
```

27 September 2020

13



Contd.

14

Postfix increment/decrement

- the operator comes after the variable, e.g. i++ or j--
- Meaning:
  - First use the current value of the variable in the expression
  - Then sometime before the next statement executes, increment (or decrement) the variable
  - This is often very useful with iteration/loops

Be careful: a statement like this is always equivalent to 2 statements as shown. It is not equivalent to x = j – 1, since that would leave the value of j unchanged.

```
i++;             // i = i + 1;
x = j--;         // x = j; j = j - 1;

i++ = 4;         // What does this mean? Is it sensible or legal?
```

```
// Advanced - revisit this after looking at arrays
myArray[i++] += 4; // myArray[i] = myArray[i] + 4; i = i + 1;
```

27 September 2020

14

Contd.

Sequencing of operators with side effects

- The following operators all have side effects on one of their operands (specifically, they change the value of it):
  ++ --
  =
  += -= *= /= etc.

Problem

- The C language does not define an order for evaluating all parts of an expression (despite operator precedence), e.g. if a is initially 2, what is the value of
  `(a=1) + (b=a)` ?
- If the same variable gets modified by 2 or more side effects in a single expression (*), the **result is undefined**
- Examples of problematic expressions

```
// Never write code like this!!
i = i++ + 2;
i = i++ + i;
i += i++;
```

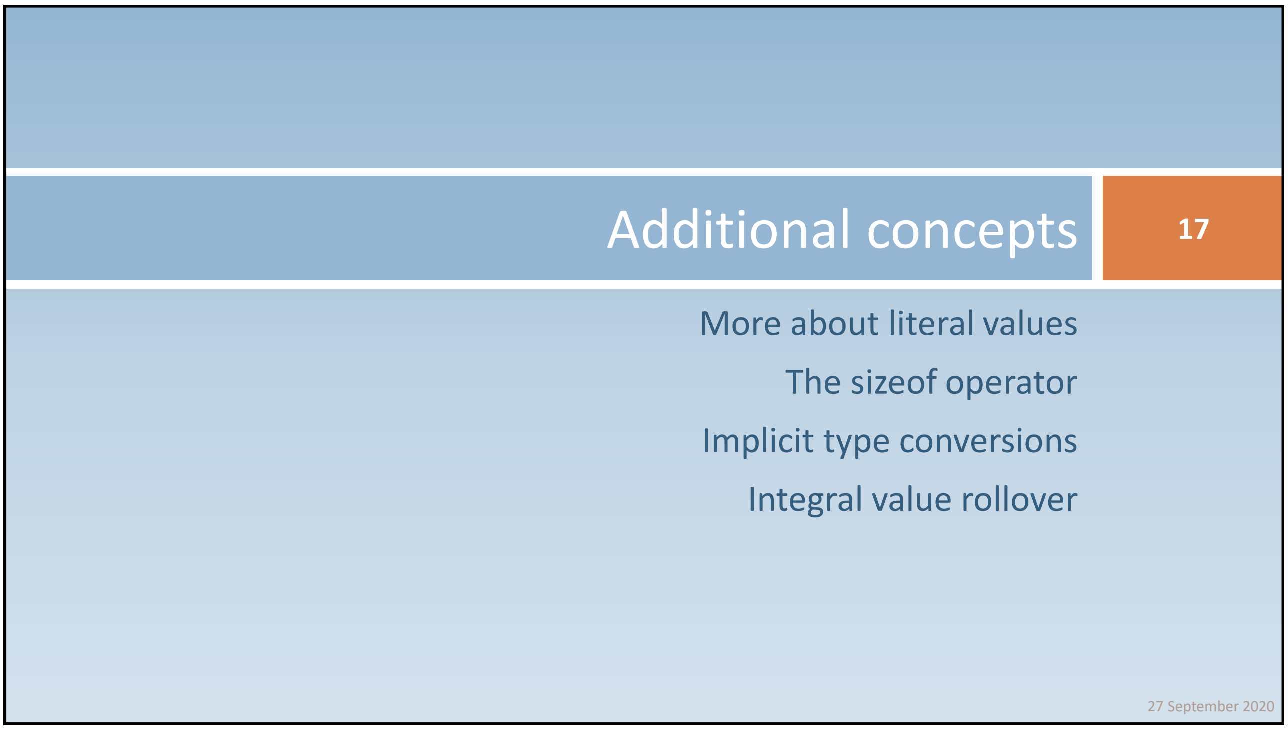
Self test questions

Q1. Assume that we set i=5 and x=10 before each statement below. What is the equivalent set of statements and the resulting values in each of the following expressions?

```
i++;
x *= i--;
i *= i--;
x += -i++;
--i;
x -= -i--;
x += -i++;
```

Q2. Assume that we set i=5 and j=3 before the expression below. What is the equivalent set of statements and the resulting values in each of the following expressions?

```
x = i++ * ++j;
```



Additional concepts

17

More about literal values

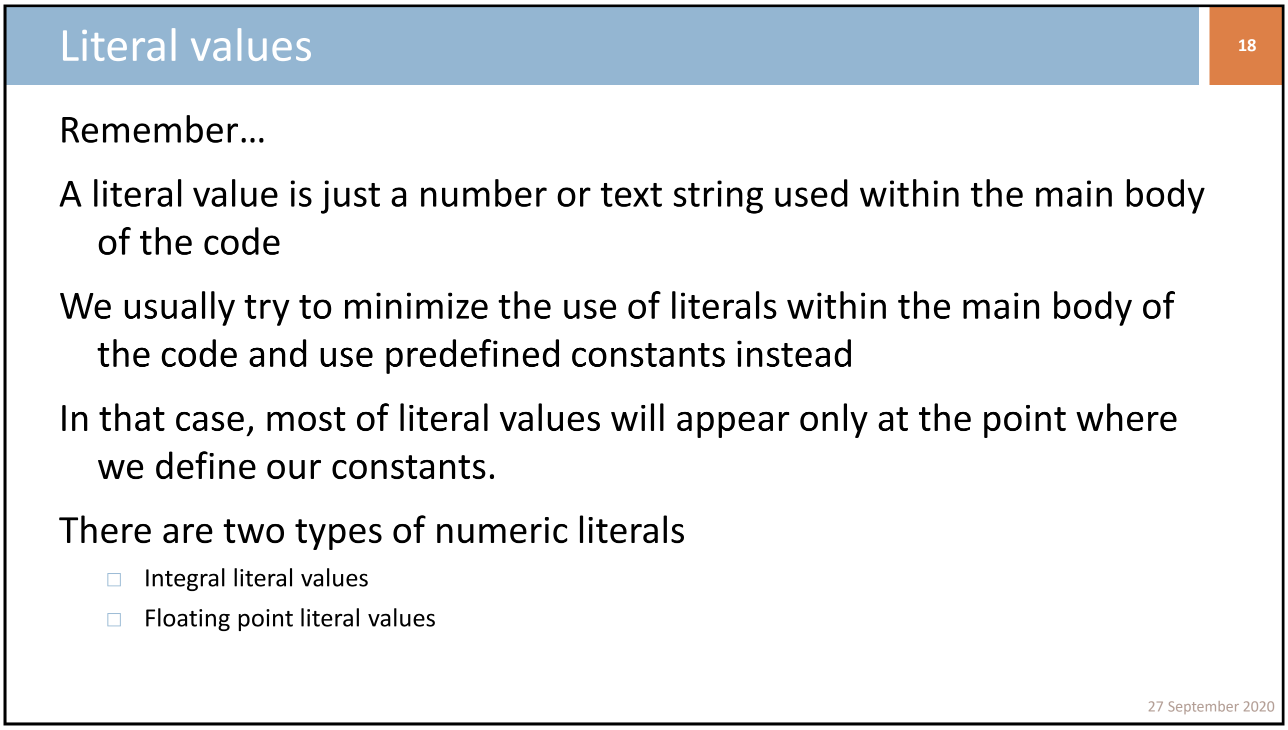
The sizeof operator

Implicit type conversions

Integral value rollover

27 September 2020

17



Literal values

18

Remember…

A literal value is just a number or text string used within the main body of the code

We usually try to minimize the use of literals within the main body of the code and use predefined constants instead

In that case, most of literal values will appear only at the point where we define our constants.

There are two types of numeric literals

- Integral literal values
- Floating point literal values

27 September 2020

18

Integral literal values

By default

- Integral literals are considered to be the smallest of following in which their value can be represented: int, long, and long long

You can override the default

- To force the compiler to treat a literal as a long or long long by adding the suffix L or LL, e.g

```
someLong = 15L;             // what type would this default to?
someLongLong = 0x7fffLL;     // what type would this default to?
```

- You can also specify that a literal should be interpreted as unsigned using the suffix U. E.g.

```
someUnsignedInt = 15U;
```

- L or LL and U also can be combined if required. Order doesn't matter E.g.

```
someUnsignedLong = 15UL;
someUnsignedLong = 15LU;
someUnsignedLongLong = 0x7fffULL;
```

Floating point literal values

To specify a floating point literal

- You must specify a decimal or exponent, e.g

`57.0    57.    57.e0    57E0    5.7e+1    .57e2    570e-1`

```
// Both of these represent the same number

someDouble = 57.0;
someDouble = 57.;

// All of the following represent the same number as above
// The e/E defines the exponent and means multiply by 10 to power of
// following number

someDouble = 57.e0;
someDouble = 57E0;
someDouble = 5.7e+1;
someDouble = .57e2;
someDouble = 570e-1;
```

Contd.

21

By default

- By default a floating point literal is considered to be of type double.

You can override the default

- You can force the compiler to treat a floating point literal as a float by adding the suffix F, e.g

```
someFloat = 57.0F;
```

- You can force the compiler to treat a floating point literal as a long double by adding the suffix L, e.g

```
someFloat = 57.0L;
```

27 September 2020

21

The sizeof operator

22

The **sizeof** operator is used to determine how many bytes will be used to store values of a particular type

It allows you to write a portable programme that can adjust to the different storage sizes on different architectures, e.g. an int is 2 bytes (16 bits) on Arduino Uno but 4 bytes (32 bits) on the ARM processor (commonly used in phones)

```
int myInt = 42;
unsigned long myULong;
const char MY_CONSTANT_CHAR = 1;
…
// for each of following, what is the result on Arduino?
result = sizeof(long long);    // sizeof used with type names
result = sizeof(short);


result = sizeof(myInt);        // sizeof used with variable name
result = sizeof(myULong);
```

27 September 2020

22

The sizeof operator contd.

The **sizeof** operator can also be used to report the memory required for constants, literals, and expressions (though this last use is not typical)

```
#define MY_DEFINE_CONST 2
const char MY_CONSTANT_CHAR = 1;
…
// for each of following, what is result on Arduino?
result = sizeof(MY_CONSTANT_CHAR);  // sizeof used with constant

result = sizeof(MY_DEFINE_CONSTANT); // sizeof used with #define or literal

result = sizeof(11000); // sizeof used with #define or literal

result = sizeof(11000L); // sizeof used with #define or literal
```

The sizeof operator also has a special meaning with arrays which we will see later in the course.

Type conversions - implicit

Mixing types in expressions

- In C, you can mix types (e.g. ints, floats, and longs) in a single expression
- Implicit conversions are usually required to make this work

Implicit conversions

- When the two operands of an operator have different types, they must be converted according to <u>one</u> of two rules
- **Rule 1:** If either operand is a floating point type… (see next slides)
- **Rule 2:** if both operands are integral types… (see next slides)

Integral to floating point type conversions

Rule 1: If either operand of a two-operand operator is a floating point type, then:

The compiler identifies the widest of the two (floating point) types used by the operands (long double, double, or float)

The compiler then converts the other operand to that type

NOTE: When integer types are converted to floating point, there may be a loss of precision

```
long lv = 0x7fff;
int i = 32000;
char c = 32;
float f = 1.0;

…
… = i + f;         // i gets converted to float, expression result is a float
… = lv + f;        // lv gets converted to float, expression result is a float
… = c + f;         // c gets converted to float, expression result is a float
```

Integral to integral type conversions

Rule 2: If both operands are integral types:

The compiler first performs integral promotion: char and short types get “promoted to” the int type before being used in <u>any</u> expression calculation.

After integral promotion, the compiler identifies the widest of the two types used by the operands (unsigned long long, long long, unsigned long, long, unsigned int, or int) and converts the other operand to that type

⚠ ***Beware:*** *when a signed type is converted to unsigned the (two’s complement) binary representation is not changed – this is different to abs value. Positive numbers will be preserved, but negative signed numbers will be interpreted as large positive unsigned numbers which can lead to difficult and obscure bugs*

Recommendation: Don’t mix signed and unsigned types in a single expression using implicit conversions if it is possible the signed type may be negative

```
long lv = 0x7fffffff;
int i = 32000;
char c = 32;

…
… = lv + i;         // i gets converted to long, expression result is long
… = lv + (c + 1);   // c gets converted to int (to add 1) and this
                    // intermediate result gets converted to long to add to lv
```

Type conversions in assignment

For assignment, C converts the type of the expression on the right hand side (RHS) of the assignment operator to the type of the variable/constant on the left hand side (LHS)

LHS integral, RHS floating point: the fractional part of the RHS is dropped

LHS and RHS types are of the same basic type (both integral or both floating point)

- If the LHS type is wide enough for the RHS type there is no problem
- If the LHS type is narrower than the RHS type and the RHS value is within the range of the LHS type, then everything is OK
- If the LHS type is narrower than the RHS type and the RHS value exceeds the range of the LHS, then the value stored will be garbage – the compiler may warn of "truncation" if it detects this (but the programme will still compile so you must watch out!)

```
long lv;
int i;
float f;

lv = i;      // i is converted to long
f = i;       // i is converted to float
i = 98.76;   // i is now 98
lv = 10000;  // RHS fits in range of LHS, so OK
i = lv;      // lv truncated to sizeof int, but 10000 still within range so OK
i = 100000;  // rhs truncated to sizeof int and overflows so garbage
```

Self test questions

Q1. Detail all the implicit conversions required in the following code

```
long lv;
int i;
char c;
unsigned char uc;
float f;

i = i + c;
lv = i + c;
i = lv + c;

i = f + (2 * c);

i = 100 / 2.5;
f = 100 / 2.5;

c += 200;
uc = c + 1;
```

Integral variable rollover

When an integral variable exceeds its maximum value, it will “roll over” to its minimum value

Likewise, when an integral variable attempts to take a value less than its minimum value, it will “roll over” to its maximum value

```
int iv;
char cv;
unsigned char uc;

iv = 32767; // same as 0x7fff
iv += 1;    // iv rolls over and is now -32768

iv = -32768;
iv -= 1;    // iv rolls over in the negative direction and is now 32767

cv = 127;
cv += 1;    // +ve roll over, cv is now -128

cv = -128;
cv -= 1;    // -ve roll over, cv is now 127

uc = 255;
uc += 1;    // +ve roll over, cv is now 0

uc = 0;
uc -= 1;    // -ve roll over, cv is now 255
```

Self test questions

Q1. What is the value of the following

```
unsigned char uc;

uc = 250;
uc += 10;

uc = 3;
uc -= 6;
```