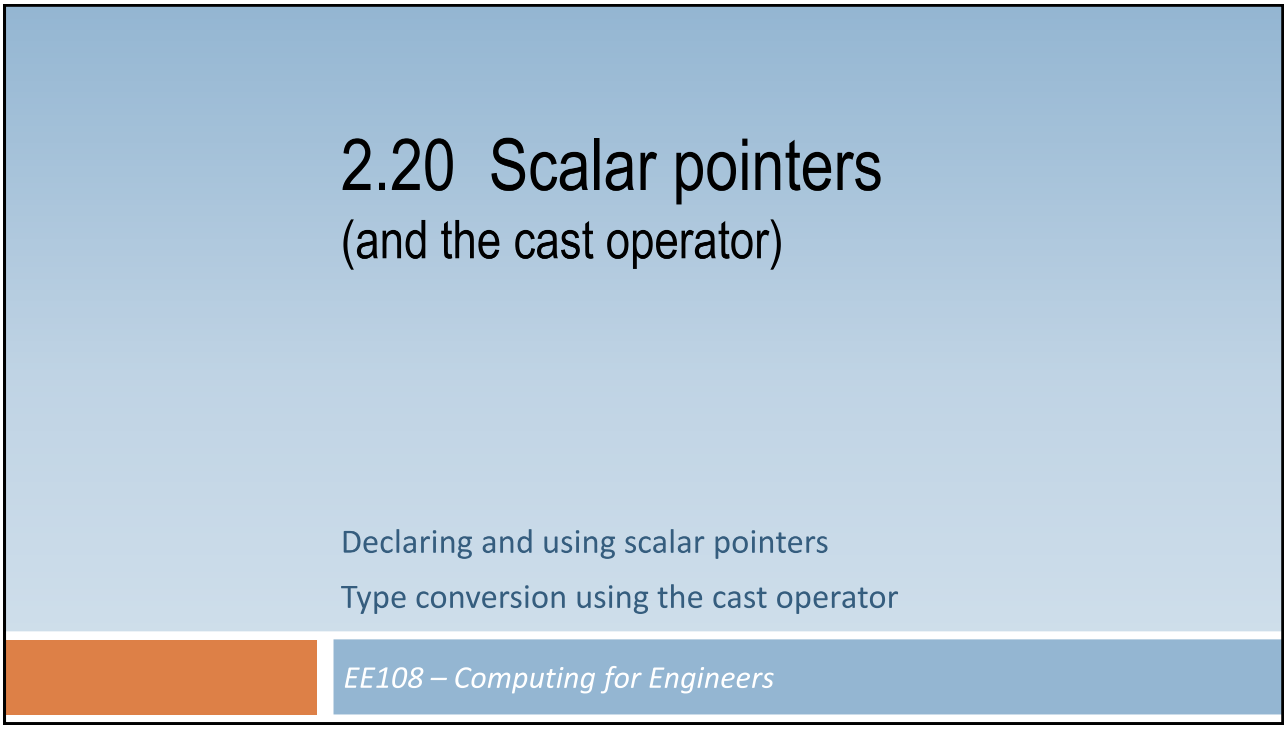


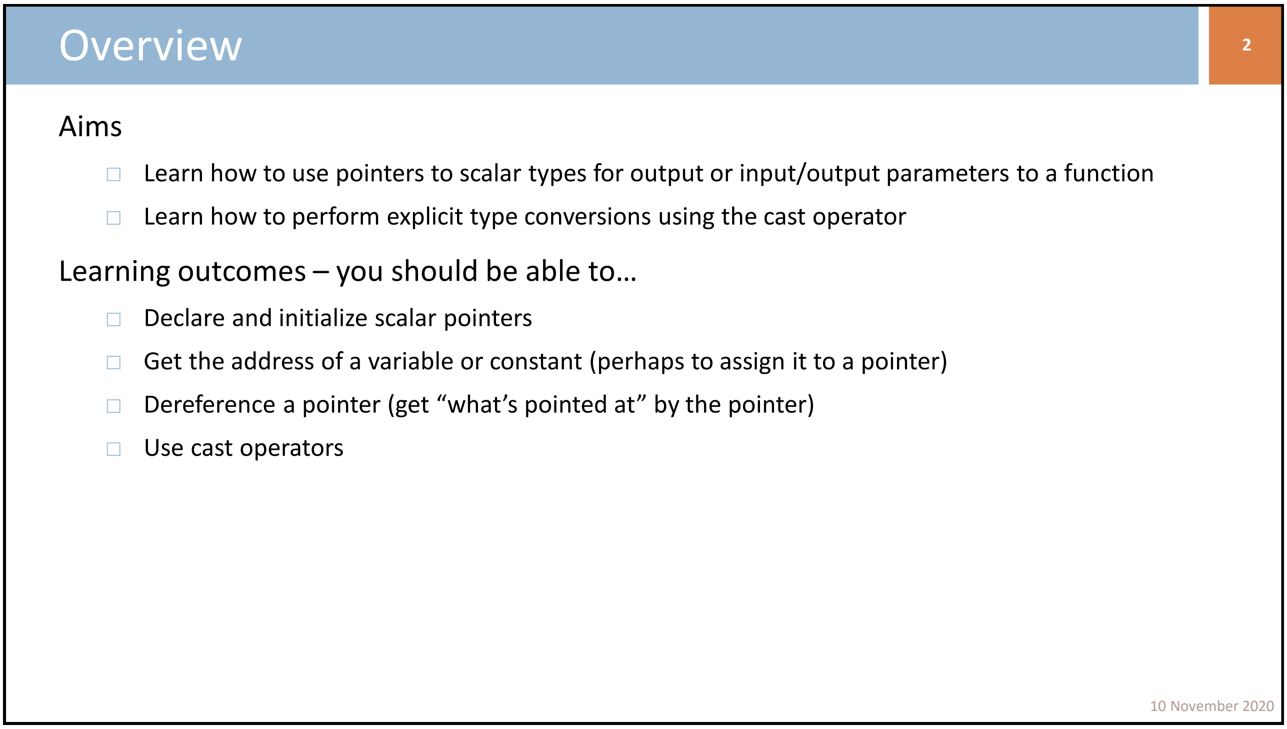
# 2.20 Scalar pointers
## (and the cast operator)

Declaring and using scalar pointers

Type conversion using the cast operator

*EE108 – Computing for Engineers*

1



## Overview

2

Aims

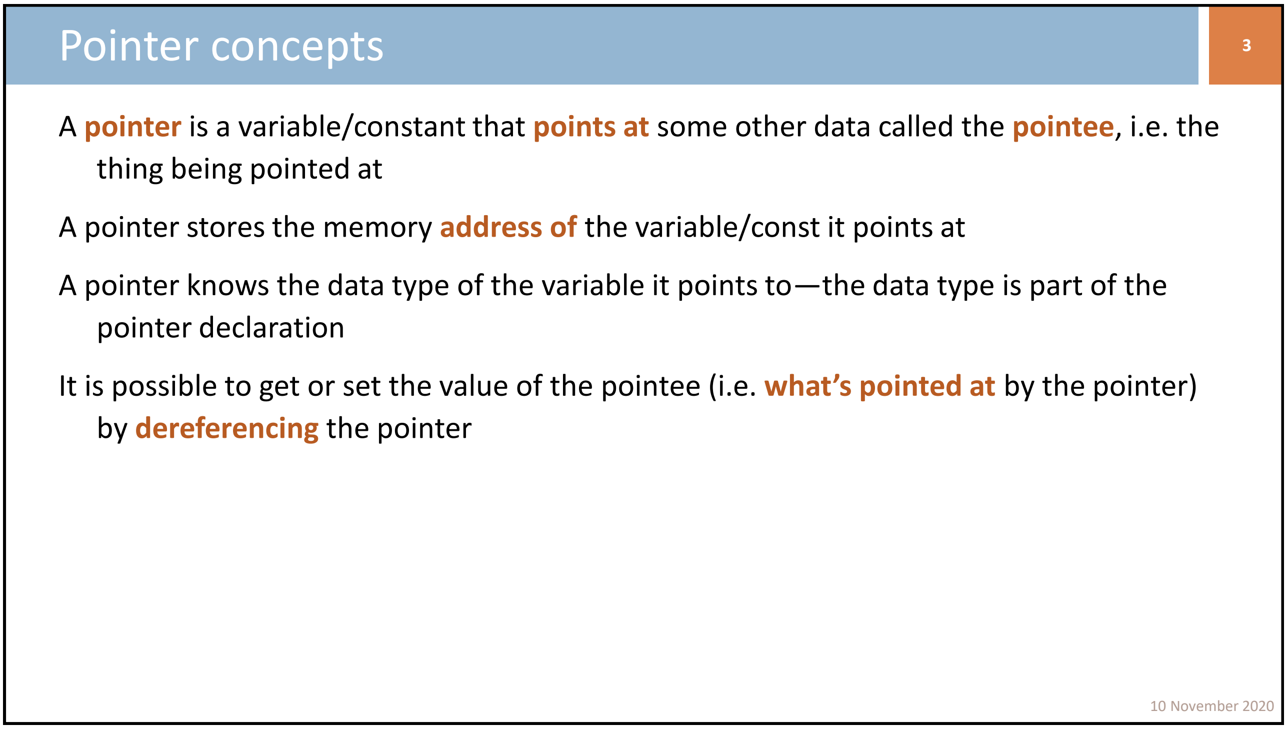
- □ Learn how to use pointers to scalar types for output or input/output parameters to a function
- □ Learn how to perform explicit type conversions using the cast operator

Learning outcomes – you should be able to…

- □ Declare and initialize scalar pointers
- □ Get the address of a variable or constant (perhaps to assign it to a pointer)
- □ Dereference a pointer (get “what’s pointed at” by the pointer)
- □ Use cast operators

10 November 2020

2



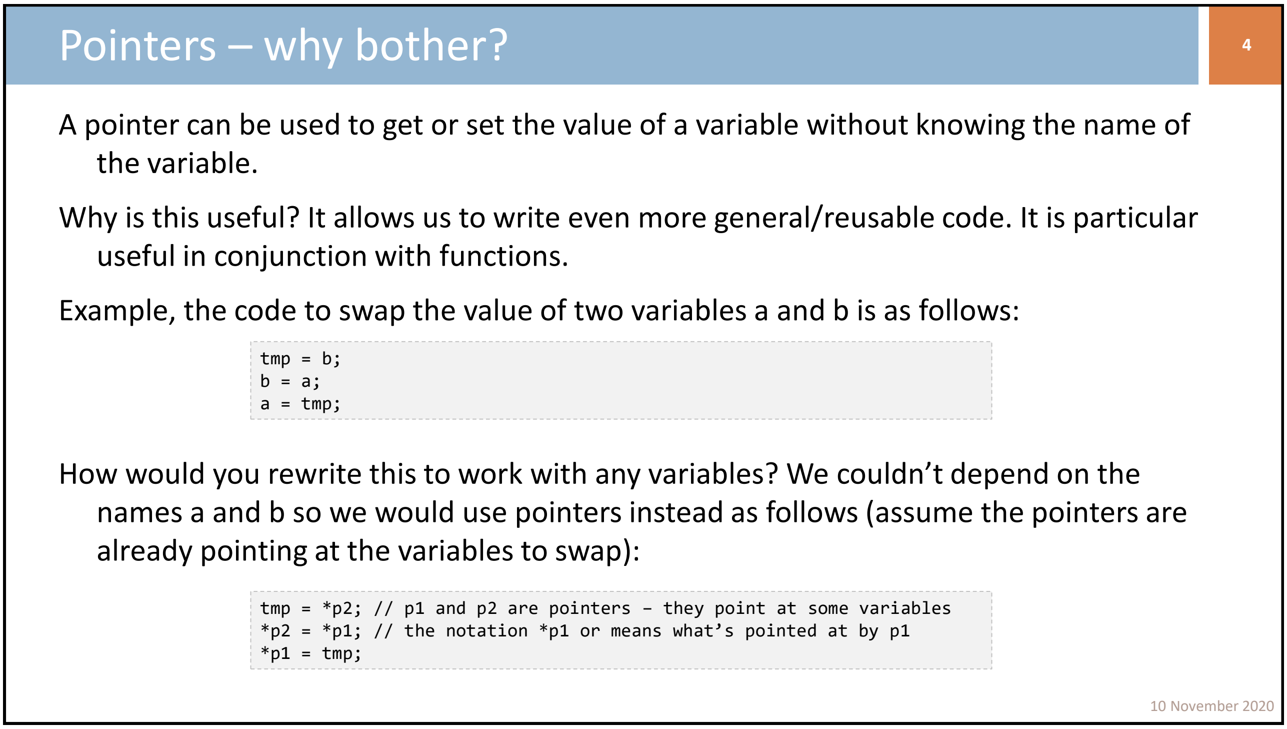
## Pointer concepts

A **pointer** is a variable/constant that **points at** some other data called the **pointee**, i.e. the thing being pointed at

A pointer stores the memory **address of** the variable/const it points at

A pointer knows the data type of the variable it points to—the data type is part of the pointer declaration

It is possible to get or set the value of the pointee (i.e. **what's pointed at** by the pointer) by **dereferencing** the pointer



## Pointers – why bother?

A pointer can be used to get or set the value of a variable without knowing the name of the variable.

Why is this useful? It allows us to write even more general/reusable code. It is particular useful in conjunction with functions.

Example, the code to swap the value of two variables a and b is as follows:

```
tmp = b;
b = a;
a = tmp;
```

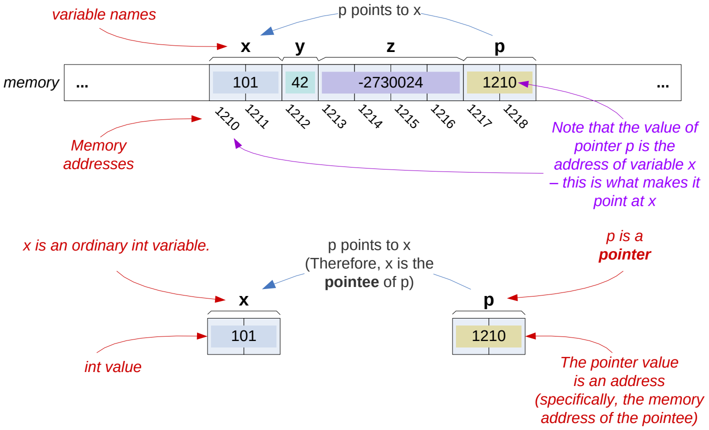
How would you rewrite this to work with any variables? We couldn't depend on the names a and b so we would use pointers instead as follows (assume the pointers are already pointing at the variables to swap):

```
tmp = *p2; // p1 and p2 are pointers – they point at some variables
*p2 = *p1; // the notation *p1 or means what's pointed at by p1
*p1 = tmp;
```

# Visualizing scalar pointers

Suppose x is an int (2 bytes wide), y is a char (1 byte wide), z is a long (4 bytes wide), and p is a pointer to an int. (All pointers have the width of a memory address which we assume here is 2 bytes).

Suppose that p has been set to point at x.



*variable names*

p points to x

x y z p

*memory* ... 101 42 -2730024 1210 ...

1210 1211 1212 1213 1214 1215 1216 1217 1218

*Memory addresses*

*Note that the value of pointer p is the address of variable x – this is what makes it point at x*

*x is an ordinary int variable.*

p points to x (Therefore, x is the **pointee** of p)

*p is a* ***pointer***

x 101

p 1210

*int value*

*The pointer value is an address (specifically, the memory address of the pointee)*

# Declaring a pointer

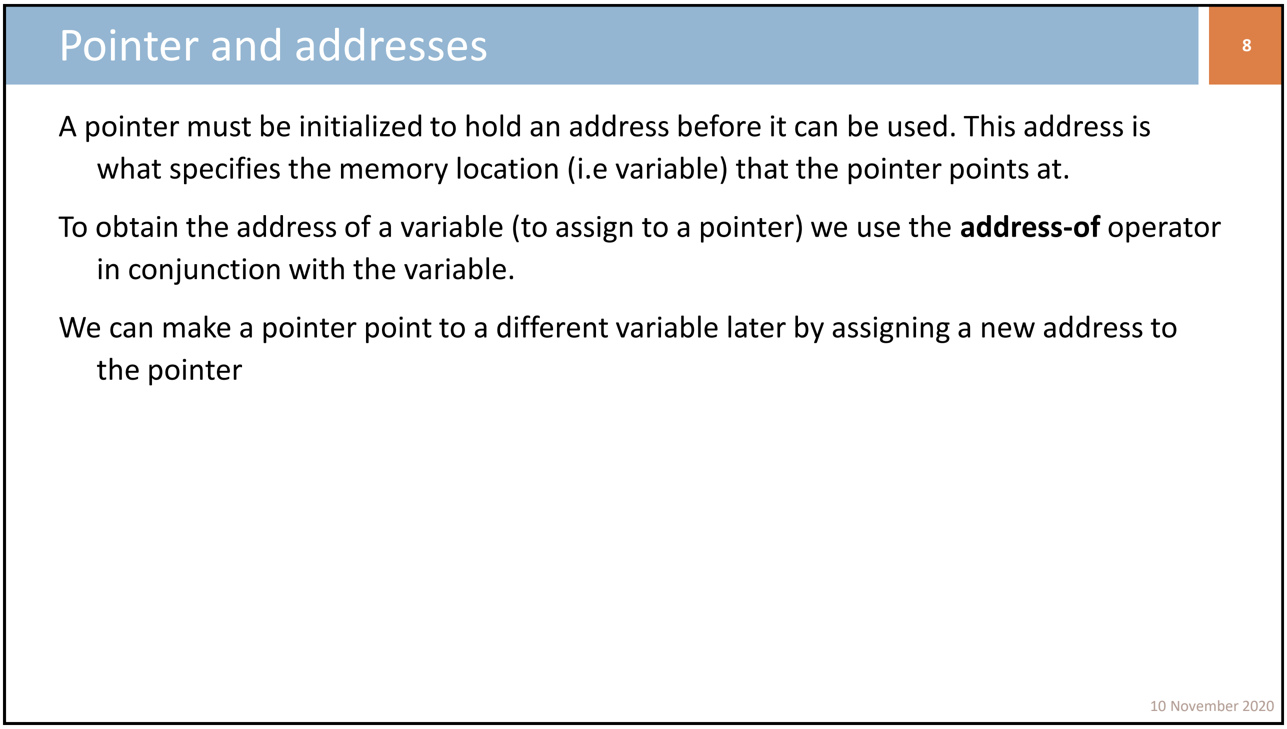
A pointer declaration includes a * to indicate that it is a pointer

- the * goes between the data type pointed at and the pointer name (usually just before the pointer name)
- The data type being pointed at is specified to the left of the *

```
int *p1;          // uninitialized pointer to an int type
char *ptr2;        // uninitialized pointer to a char type
float *pThird;     // uninitialized pointer to a float type
```

As usual it is possible to declare the variable without any initializer as shown above

***Recommendation: where possible, use the 'p' prefix on pointer variables to remind you that they are pointers, e.g. pValue, pThing, etc.***

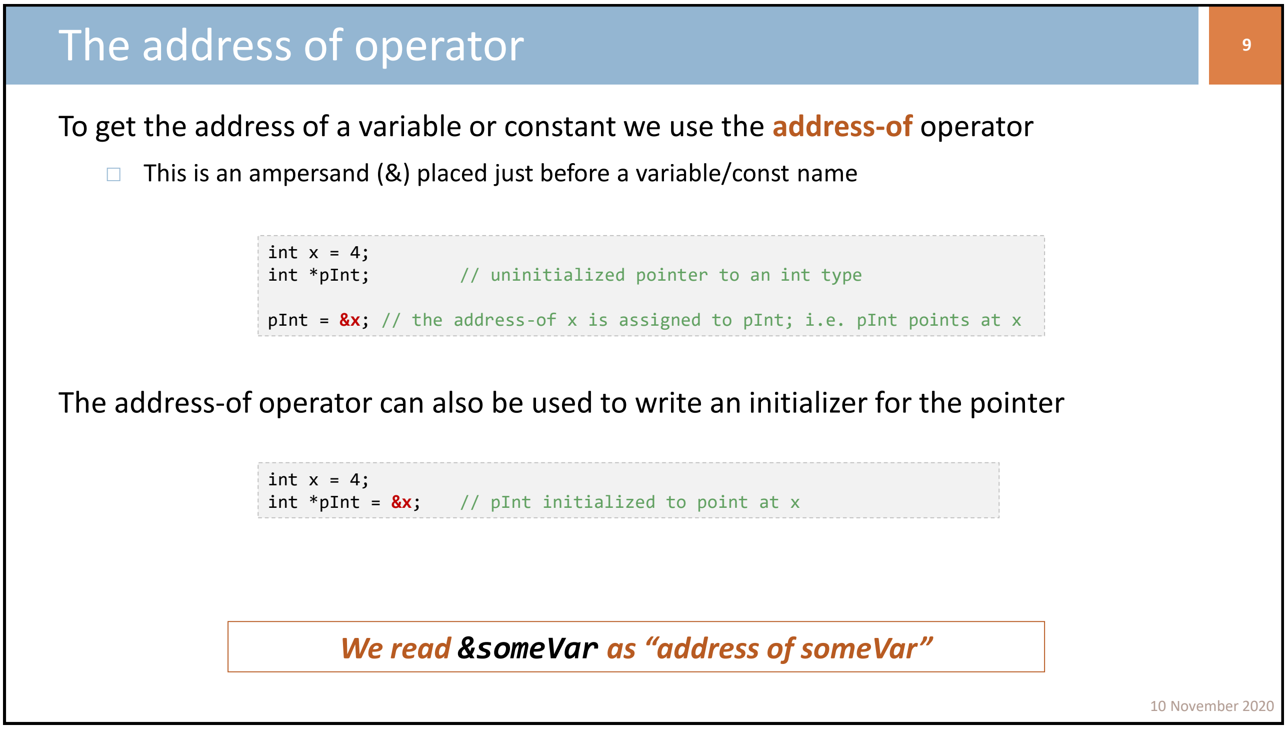


# Pointer and addresses

A pointer must be initialized to hold an address before it can be used. This address is what specifies the memory location (i.e variable) that the pointer points at.

To obtain the address of a variable (to assign to a pointer) we use the **address-of** operator in conjunction with the variable.

We can make a pointer point to a different variable later by assigning a new address to the pointer



# The address of operator

To get the address of a variable or constant we use the **address-of** operator

- This is an ampersand (&) placed just before a variable/const name

```
int x = 4;
int *pInt;         // uninitialized pointer to an int type

pInt = &x; // the address-of x is assigned to pInt; i.e. pInt points at x
```

The address-of operator can also be used to write an initializer for the pointer

```
int x = 4;
int *pInt = &x;    // pInt initialized to point at x
```

***We read* `&someVar` *as "address of someVar"***

# The NULL pointer

Sometimes it can be convenient to explicitly set or initialize a pointer to a special address to indicate that it points at nothing at all

In this case we set or initialize the pointer to address zero (usually using the predefined constant NULL) – this is taken to mean that the pointer points at nothing

- The constant NULL is defined by default in the Arduino environment and defined in the include file stddef.h for C programmes in other environments

```
int x = 4;
int *pInt = &x;    // pInt points to x
int *pInt2 = NULL;  // initialize pointer to point at nothing
int *pInt3; // uninitialized pointer; it points at something random

pInt = NULL;    // we can set a pointer to point at nothing
```

The NULL pointer is often used when supplying a value is optional (see later under pointer comparisons)

# The dereference operator (what’s pointed at by)

The **dereference operator** is used to get at the pointee, i.e. the variable/const that the pointer points to

Using the dereference operator we can get (and set if it is not const) the value of the pointee

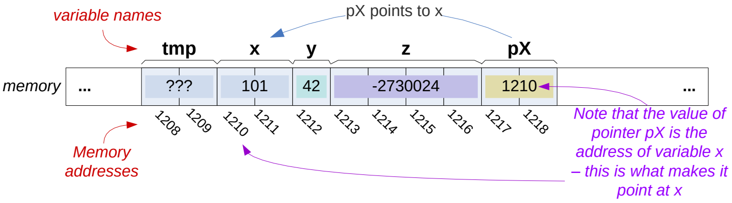
```
int x = 4;
int y;
int *pInt = &x;    // pInt points to x

y = *pInt;    // y is now the value of what’s pointed at by pInt, i.e. 4

*pInt = 10;   // the value of what’s pointed at by pInt (i.e. x) is
              // now 10
```

***We read `*somePtr` as “what’s pointed at by somePtr”***

***WARNING: never dereference an uninitialized pointer or a pointer to NULL. Always initialized the pointer to point at an existing variable/constant before use.***

# Putting it all together

Suppose x is an int (2 bytes wide), y is a char (1 byte wide), z is a long (4 bytes wide), and pX is a pointer to an int. Suppose also that pX is initialized to point at x.



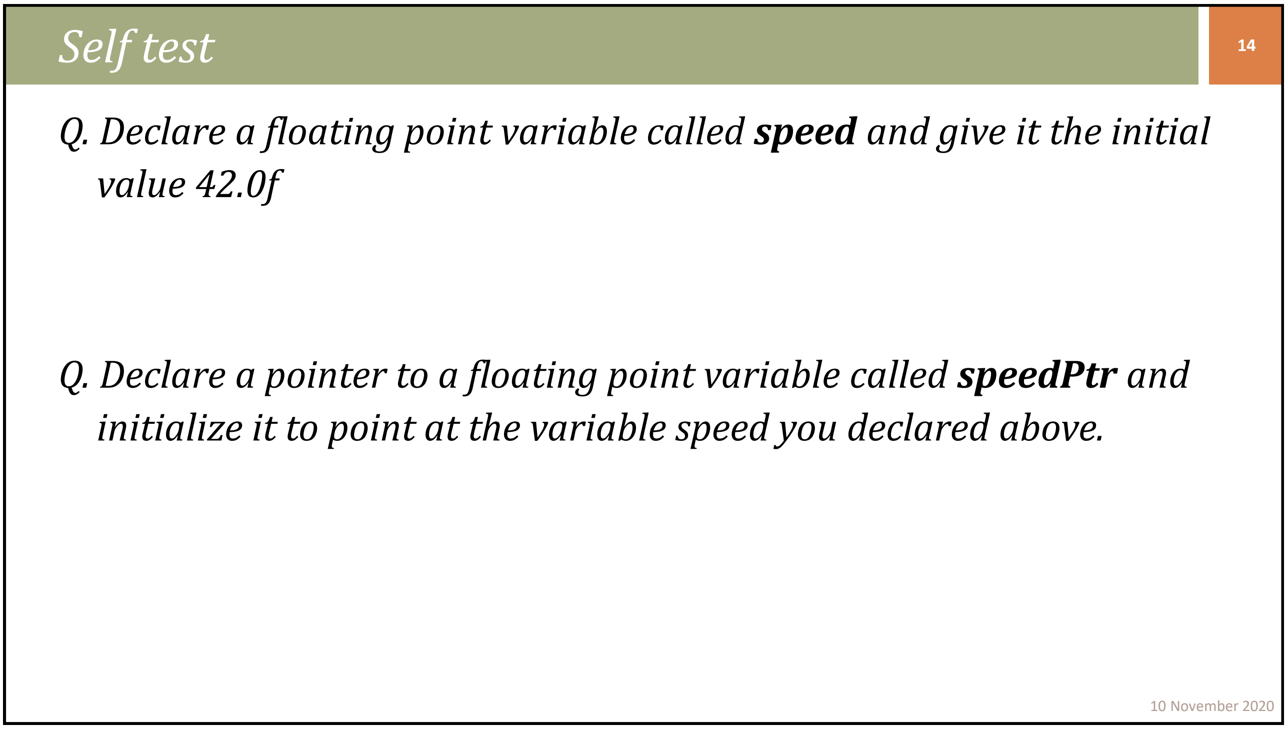
```
int tmp;
int x = 101;
char y = 42;
long z = -2730024;
int *pX;

// make pX point at x
pX = &x;
```

```
Serial.println(x); // what is displayed?

// assign 9999 to “what’s pointed at by” pX
*pX = 9999;

Serial.println(x); // what is displayed?

x = 555;
tmp = *pX;    // assign what’s pointed at by pX to tmp

Serial.println(tmp); // what is displayed?
```

# Contd. – access the pointer vs access the pointee

```
int tmp;
int x = 101;
int y = 42;
int *ptr1;
int *ptr2;
```

```
// make ptr1 point at x
ptr1 = &x;

Serial.println(x); // what is displayed?

// assign 9999 to “what’s pointed at by” ptr1
*ptr1 = 9999;

Serial.println(x); // what is displayed?

// make ptr1 point at y
ptr1 = &y;

Serial.println(*ptr1); // what is displayed?

// make ptr2 point at whatever ptr1 is pointing at
ptr2 = ptr1;

// assign 555 to “what’s pointed at by” ptr2
*ptr2 = 555;

Serial.println(x); // what is displayed?
Serial.println(y); // what is displayed?
Serial.println(*ptr1); // what is displayed?
Serial.println(*ptr2); // what is displayed?
```



## *Self test*

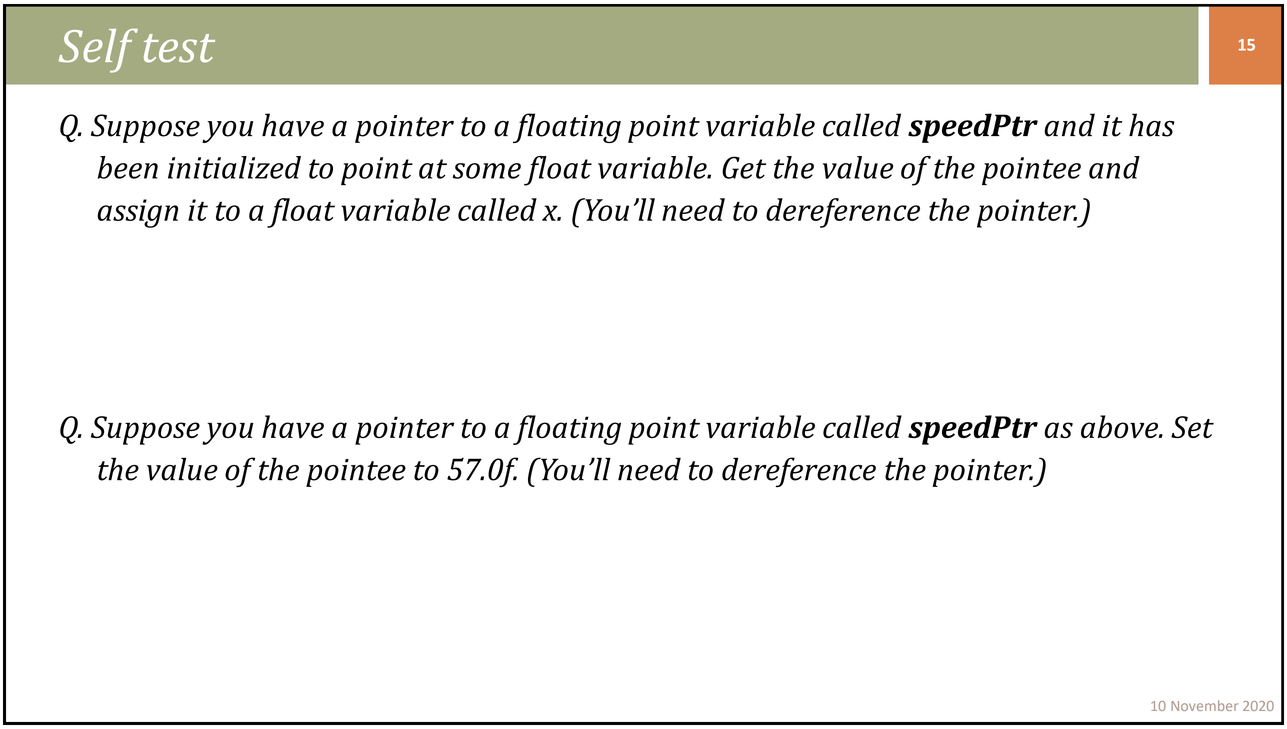
14

*Q. Declare a floating point variable called **speed** and give it the initial value 42.0f*

*Q. Declare a pointer to a floating point variable called **speedPtr** and initialize it to point at the variable speed you declared above.*

10 November 2020

14



## *Self test*

15

*Q. Suppose you have a pointer to a floating point variable called **speedPtr** and it has been initialized to point at some float variable. Get the value of the pointee and assign it to a float variable called x. (You'll need to dereference the pointer.)*

*Q. Suppose you have a pointer to a floating point variable called **speedPtr** as above. Set the value of the pointee to 57.0f. (You'll need to dereference the pointer.)*

10 November 2020

15

# The most common use of scalar pointers (pass by reference)

The most common use of scalar pointers is as function parameters when we need to get more than one value out of a function

Why?

- In the C programming language (and C++, Java, etc.) arguments that calling code “passes” to a function are **passed-by-value**
- This means that when the function is called each **argument value is copied** to the corresponding parameter seen by the function’s code
- This in turn means that if the function changes a parameter value it has no effect on any variable the calling code might have used as an argument (because it is only the copy that is modified)

```
void loop() {
  int x = 4;
  …
  modifyValue(x);
  Serial.println(x);  // what value will be printed?
}

void modifyValue(int value) {
  value = value + 1;
}
```

# Scalar pointers for pass-by-reference

If we need to change a variable value using a function (without using a return value) we must use a pointer to that variable

- Where the function declares a pointer parameter, the caller must pass in a pointer/address. We can do this by passing in the address-of an existing variable. This address is used to initialize the pointer parameter used within the function.
- Any changes the function makes to the pointee (by dereferencing the pointer) will actually be changing the original variable whose address we passed in (because that’s what the pointee is)
- When we use a pointer to a variable rather than the variable itself as an argument, the technique is called **pass-by-reference**.

```
void loop() {
  int x = 4;
…
  modifyValue(&x);    // pass in address of x instead of value of x
  Serial.println(x);  // what value will be printed?
}

void modifyValue(int* pValue) { // function takes pointer instead of int
  *pValue = *pValue + 1;     // dereference the pointer to get pointee value
}
```

# Contd.

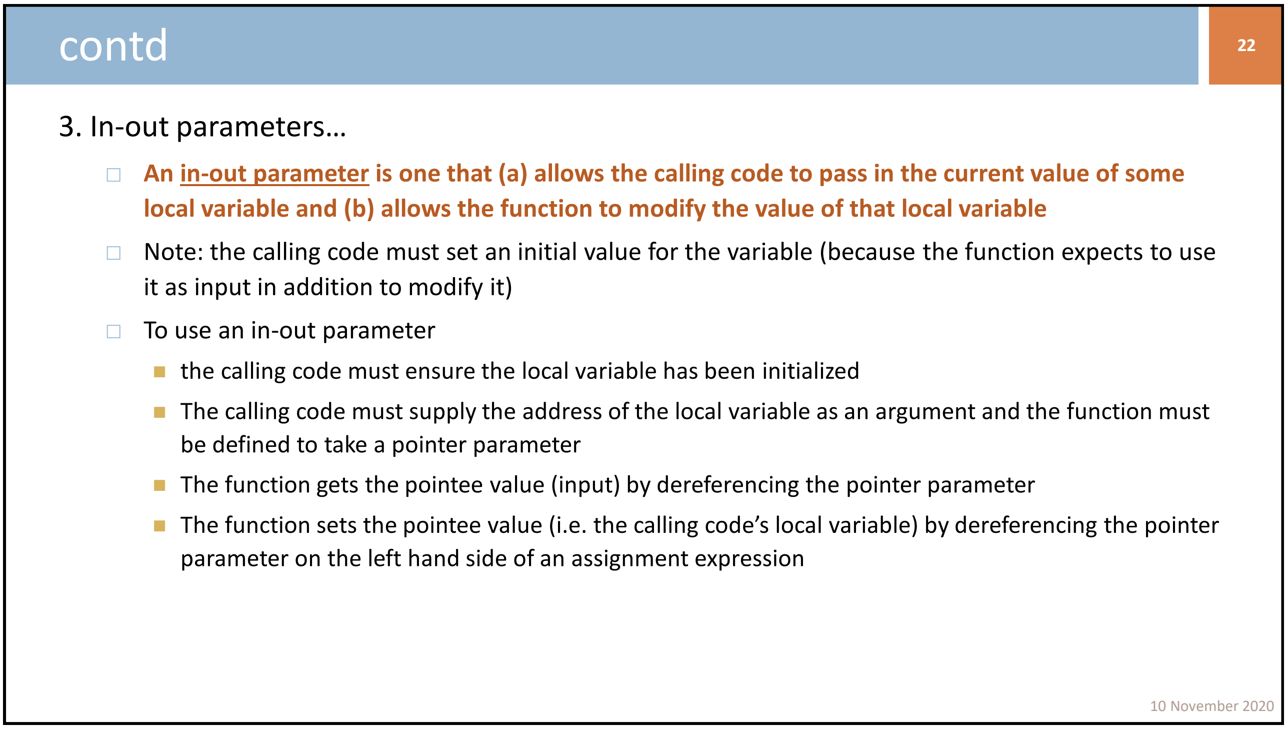
If you already have a pointer initialized to point at the variable you need, then you can pass that pointer to a function that requires a pointer parameter – you don't need to take the address-of again

```
void loop() {
  int x = 4;
  int *p = &x; // p points at x
  …
  modifyValue(p);    // pass in the value of p (which is the address of x)
  Serial.println(x);  // what value will be printed?
}

void modifyValue(int* pValue) { // function takes pointer instead of int
  *pValue = *pValue + 1;     // dereference the pointer to get pointee value
}
```

# Input, Output, and In-out parameters

Now with our new pointer technique we can classify function parameters in one of 3 ways

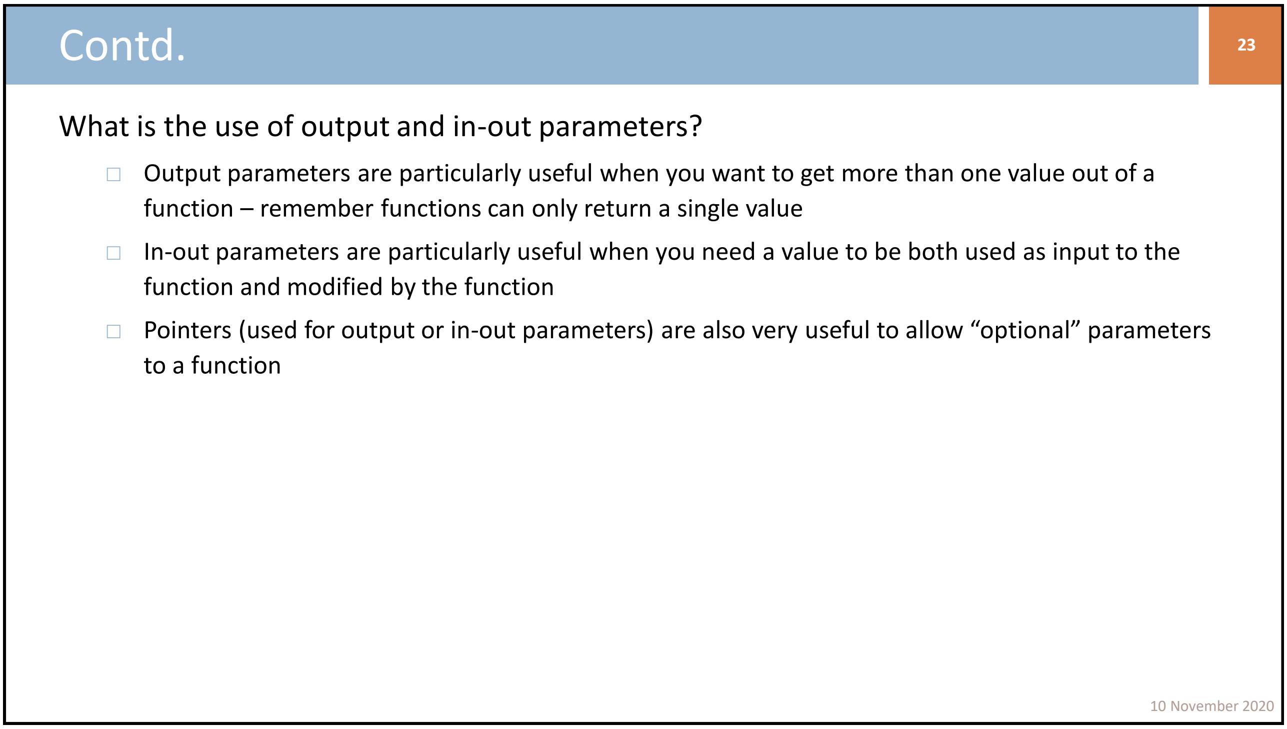
1. Input parameters
   - These are ordinary parameters which the function requires as input and which the calling code does not want, need, or expect to be modified by the function
2. Output parameters
   - **An output parameter is one that allows a local  variable in the calling code to be modified by a function**
   - Note: Since it is an output only variable, the calling code does not need to set an initial value for the variable (because it will be overwritten by the function)
   - To use an output parameter
     - the calling code must supply the address of the local variable as an argument and the function must be defined to take a pointer parameter
     - The function sets the pointee value (i.e. the calling code's local variable) by dereferencing the pointer on the left hand side of an assignment expression<br>(e.g. *ptr = 42; )
     - **NOTE: the function must not try to get the pointee value since the initial pointee value might not be initialized – remember the caller was trying to get output, not supply some value**



## contd

3. In-out parameters…

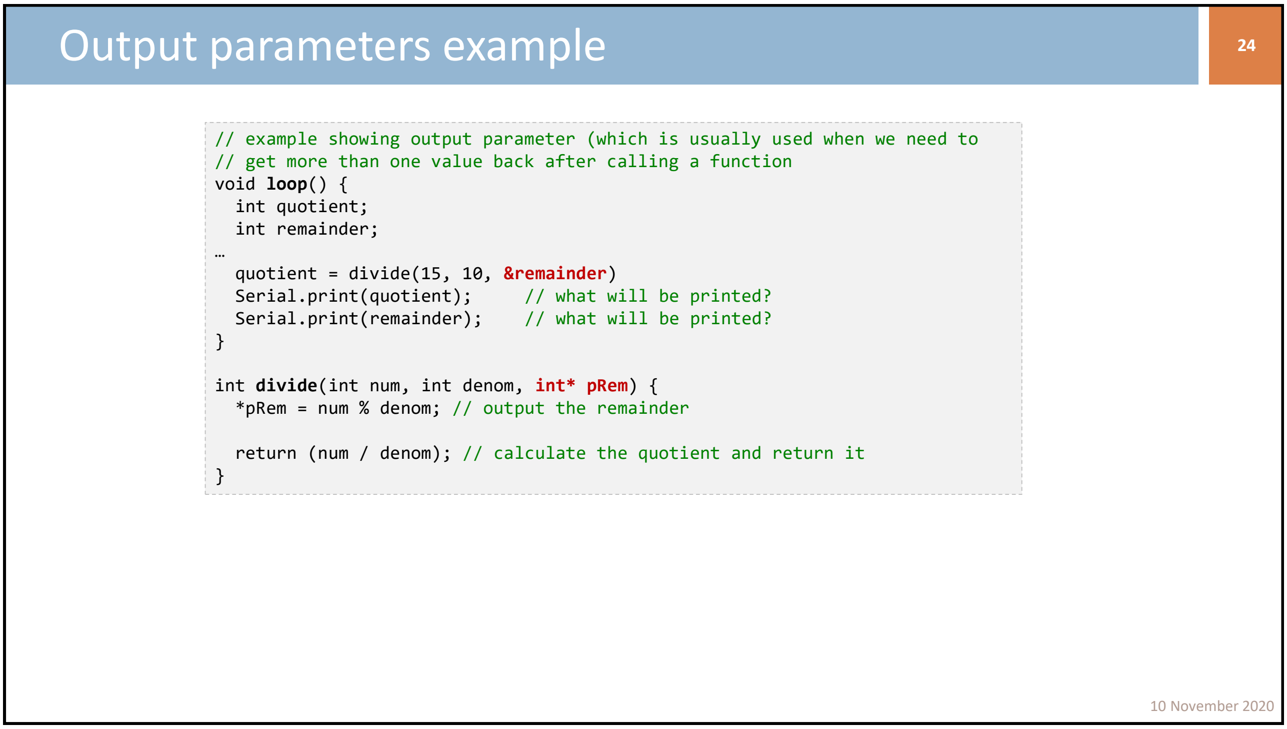
- **An <u>in-out parameter</u> is one that (a) allows the calling code to pass in the current value of some local variable and (b) allows the function to modify the value of that local variable**
- Note: the calling code must set an initial value for the variable (because the function expects to use it as input in addition to modify it)
- To use an in-out parameter
  - the calling code must ensure the local variable has been initialized
  - The calling code must supply the address of the local variable as an argument and the function must be defined to take a pointer parameter
  - The function gets the pointee value (input) by dereferencing the pointer parameter
  - The function sets the pointee value (i.e. the calling code's local variable) by dereferencing the pointer parameter on the left hand side of an assignment expression



## Contd.

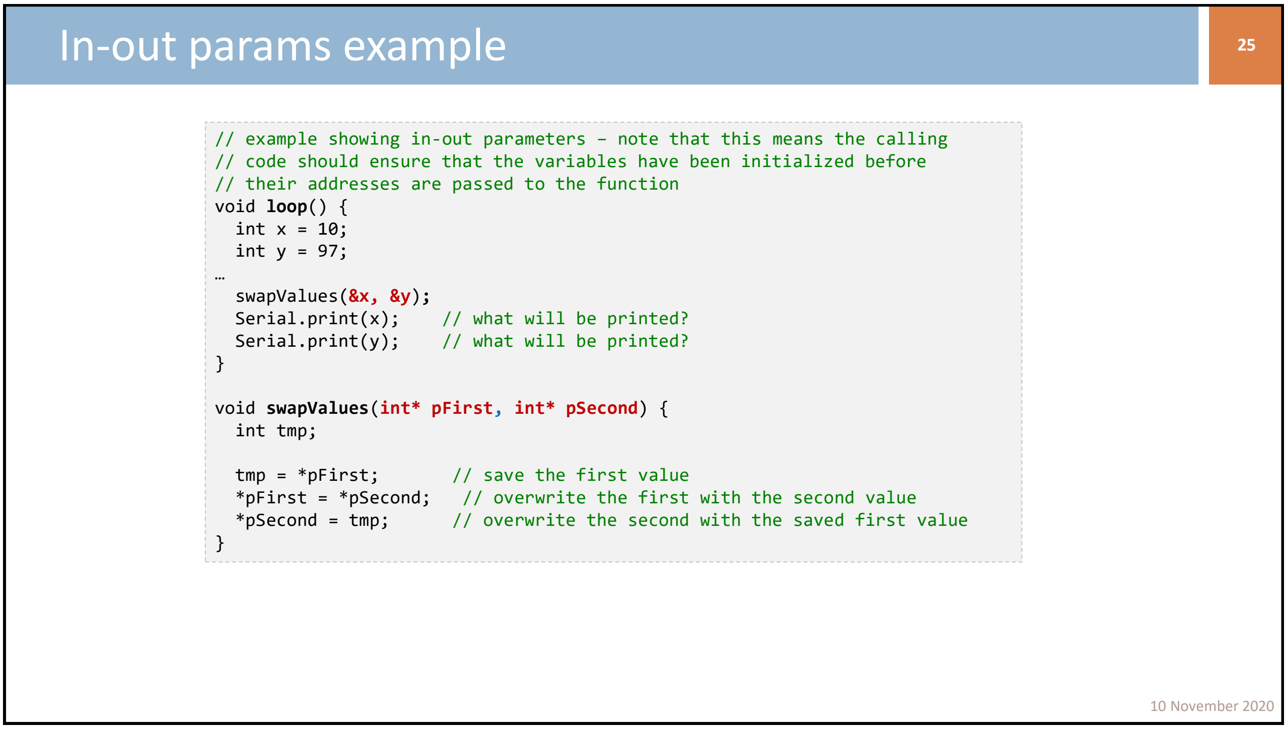
What is the use of output and in-out parameters?

- Output parameters are particularly useful when you want to get more than one value out of a function – remember functions can only return a single value
- In-out parameters are particularly useful when you need a value to be both used as input to the function and modified by the function
- Pointers (used for output or in-out parameters) are also very useful to allow "optional" parameters to a function



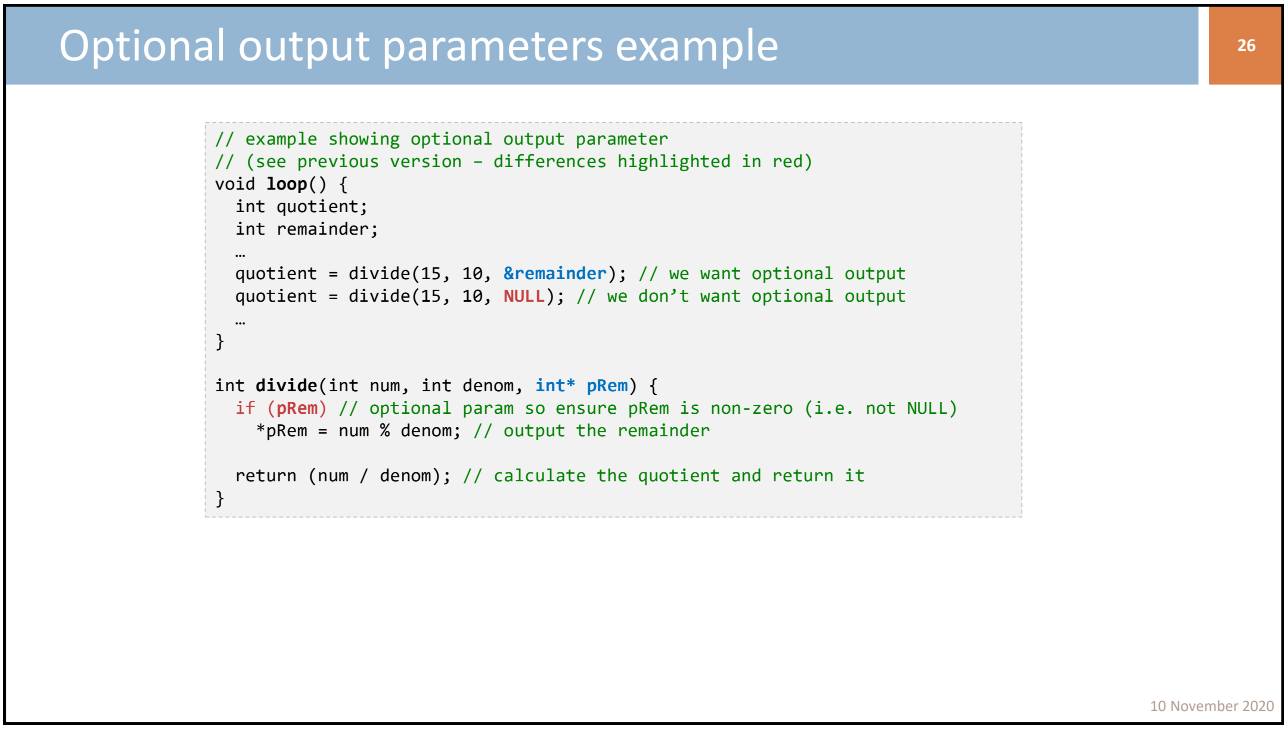
## Output parameters example

```
// example showing output parameter (which is usually used when we need to
// get more than one value back after calling a function
void loop() {
  int quotient;
  int remainder;
…
  quotient = divide(15, 10, &remainder)
  Serial.print(quotient);     // what will be printed?
  Serial.print(remainder);    // what will be printed?
}

int divide(int num, int denom, int* pRem) {
  *pRem = num % denom; // output the remainder

  return (num / denom); // calculate the quotient and return it
}
```



## In-out params example

```
// example showing in-out parameters – note that this means the calling
// code should ensure that the variables have been initialized before
// their addresses are passed to the function
void loop() {
  int x = 10;
  int y = 97;
…
  swapValues(&x, &y);
  Serial.print(x);    // what will be printed?
  Serial.print(y);    // what will be printed?
}

void swapValues(int* pFirst, int* pSecond) {
  int tmp;

  tmp = *pFirst;       // save the first value
  *pFirst = *pSecond;   // overwrite the first with the second value
  *pSecond = tmp;      // overwrite the second with the saved first value
}
```



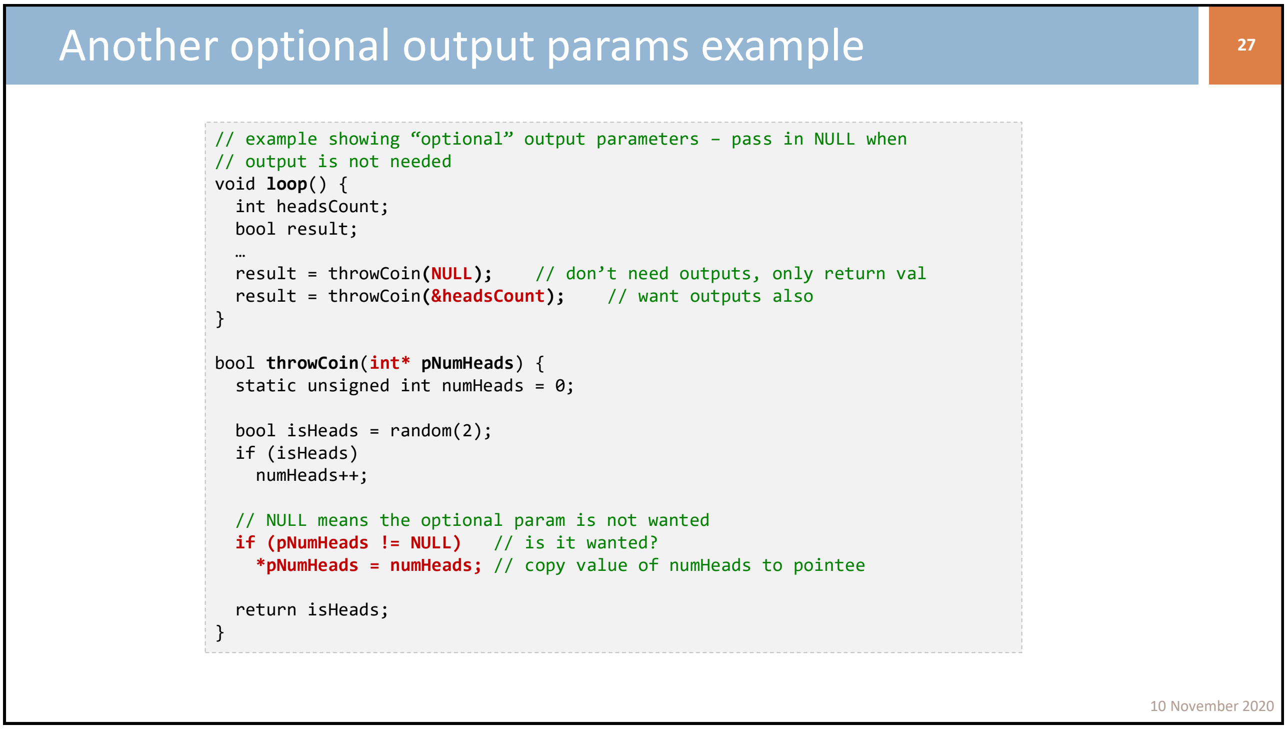
## Optional output parameters example

26

```
// example showing optional output parameter
// (see previous version – differences highlighted in red)
void loop() {
  int quotient;
  int remainder;
  …
  quotient = divide(15, 10, &remainder); // we want optional output
  quotient = divide(15, 10, NULL); // we don’t want optional output
  …
}

int divide(int num, int denom, int* pRem) {
  if (pRem) // optional param so ensure pRem is non-zero (i.e. not NULL)
    *pRem = num % denom; // output the remainder

  return (num / denom); // calculate the quotient and return it
}
```

10 November 2020

26



## Another optional output params example

27

```
// example showing “optional” output parameters – pass in NULL when
// output is not needed
void loop() {
  int headsCount;
  bool result;
  …
  result = throwCoin(NULL);    // don’t need outputs, only return val
  result = throwCoin(&headsCount);    // want outputs also
}

bool throwCoin(int* pNumHeads) {
  static unsigned int numHeads = 0;

  bool isHeads = random(2);
  if (isHeads)
    numHeads++;

  // NULL means the optional param is not wanted
  if (pNumHeads != NULL)   // is it wanted?
    *pNumHeads = numHeads; // copy value of numHeads to pointee

  return isHeads;
}
```

10 November 2020

27

## *Self test*

*Q. Declare a function **foo** that takes a single int output parameter.*

*Q. Declare a function called **bar** that takes a single char in-out parameter*

*Q. Declare an int variable called x, and call foo so that it puts it output value into x.*

*Q. Declare a char variable called c that is initialized 120, and call bar so that it both uses c as input and then puts it output value into c.*

## *Self test*

*Q. Write the definition of a function called **update** which takes a single in-out parameter called **value** that is optional. If value is being used, divide it by 2, unless it is already zero, in which case set it 100.*

*Q. Assuming an int variable called x exists, call the function (a) without using x, (b) using x as the argument to the in-out parameter.*

# Type conversions and casting

31

31

## Type conversions and casting

Previously (in notes 1.20) we saw that

- you can mix data types (e.g. int, float, long) in expressions
- When you mix data types like this, C will perform implicit conversions unless you tell it otherwise
  - E.g. When an int is added to a float the int is implictly converted to a float and the result of the expression is also a float

What happens if you want to prevent or override the implicit conversions in C?

The solution is to use the **cast** operator

32

# Type conversions - cast

The **cast** operator allows you to take control and tell the compiler what type conversion to make

The syntax is:

```
( typename ) expression
```

- Where typename is int, char, long, float, etc.
- The precedence of the cast operator is higher than many other operators so that a cast will be evaluated before an arithmetic operator

Some use cases and examples on next slide…

# Casting examples

```
int i=1, j=5, k=1000;
long l;
float f;

f = i / j;          // f is now zero! Why?
                    // because 5 goes into 1 zero times using integer math

f = (float) i / j;  // f is now 0.2 ! Why?
                    // the cast operator is evaluated first and
                    // (float)i is 1.0. Implicit conversion rules say
                    // float / int is evaluated as float / float
                    // so 1.0/5.0 is 0.2.

l = k * k;          // l is now garbage! Why? Because k*k is an int * int
                    // which evaluates to an int result. 1000000 cannot fit
                    // in an int

l = (long) (k * k); // l is still garbage! Why? Because the parentheses
                    // around (k*k) forces this to be evaluated before the
                    // cast and the result of an (int * int) is still an int
                    // and 1000000 still doesn’t fit in an int.

l = (long) k * k;   // l is correctly now 1000000. Why? Because a long * int
                    // is evaluated as long * long which gives a long result
```

# Cast for printing pointers

Altough it is frowned upon, it is possible to cast a pointer to be treated like something else such as an unsigned int

Without going much into details

- □ It isn't possible to print pointer values (i.e. memory addresses) directly
- □ To print a pointer value (memory address) we cast it to an unsigned int.
  *(WARNING: this trick does not work on every platform, but it works on many embedded platforms!)*

```
int x;
int *pX = &x; // pX points at value

// following is fine -- we're printing the pointee which is an integer
Serial.print(*pX);

// directly printing a pointer/address as follows is not supported
// Serial.print(pX);

// here’s the workaround – cast the 16 bit pointer/address to a 16 bit
// unsigned int. This changes the way C interprets the pointer value
// but doesn’t change the value itself. We use this trick in the example
// sketches.
Serial.print((unsigned int) pX);
```

# *Self test questions*

*Q1. Add cast operators to the following code to make all implicit conversions explicit. Refer back to notes 1.20 for implicit conversions*

```
int i = …;
float f = …;
char c = …;

i = f + (2 * c);
```

*Q2. Add cast operator(s) to the following code only where needed to ensure the correct result is obtained and no integer overflow/truncation occurs.*
*Hint: check whether the result of i*j will fit in a signed int or not. If not, cast appropriately.*
*Keep in mind that it matters where you add the cast because of the order in which operators (and hence intermediate results) are evaluated.*

```
int i = 200;
int j = 500;
float f;

f = i + i * j; // remember max int is 32767 on Arduino
```