1.10 Basic data and output

Programme data (constants and variables)

Scope and storage class

EE108 – Computing for Engineers

1

Overview

- Aims
  - Briefly revise variables and constants, scope, and storage, focusing on C and Energia C specific features
- Learning outcomes – you should be able to…
  - Define and use constants
  - Declare and initialize global, local, and block local variables
  - Choose the appropriate scope and storage for a task
  - Choose the appropriate C data type for a number (based on the values it can take on)

2

Programme data

3

literals, constants, and variables

27 September 2020

3

Terminology

4

- The following words all mean approximately the same thing… at least for embedded systems
  - [Arduino] Sketch
  - Programme/Program
  - App
  - Application
  - [Software] system
- I will tend to use the terms Programme or App in this module as these terms have more general use than "sketch"

27 September 2020

4

Programme data

- Previously…
  - we saw the basic building blocks of a programme were expressions and statements
  - Expressions and (usually statements) operate on programme data using it as input and perhaps generating new data as output
- Programme data takes one of 3 forms:
  - **Literal values**
  - **Constants**
  - **Variables**

Literal values

- Literal values are just numbers or strings used within the main body of the code, e.g.

```
// Literal value examples
  …
  Serial.begin(9600);       // 9600 is a literal integer
  Serial.println("hello");  // "hello" is a literal string (sequence of characters)
  delay(100);               // 100 is a literal integer
```

- Unfortunately, too many literals makes a programme difficult to understand and maintain, e.g.

```
// Examples: what do the following literal values really mean?
  …
  pinMode(5,INPUT);   // what is pin 5 connected to?
  …
  result = r * 6.26;  // what is special about 6.26?
```

Literal values – bad example (python language)

```
# The problem with this code was that most of the literal numbers refer to absolute screen
# co-ordinates. Initially it was only run on one (large screen) laptop.
# When run on another system, the hard-coded numbers no longer worked.
# Worse, because there are only a variety of different numbers you can’t even do a
search/replace.

# [M]atrix navagator
N_nlines  = int(15)
N_ncols   = int(88)
N_begin_y = int(0)
N_begin_x = int(0)
NavScreen = screen.subwin(N_nlines, N_ncols, N_begin_y, N_begin_x)
NavScreen.box()NavScreen.hline(2, 1, curses.ACS_HLINE, N_ncols-2)
NavScreen.addstr(1,N_ncols/2 -6,"[M]atrix navagator")
NavScreen.refresh()
# [P]ose
P_nlines  = int(28)
P_ncols   = int(176-N_ncols)
P_begin_y = int(0)
P_begin_x = int(N_ncols)
PoseScreen = screen.subwin(P_nlines, P_ncols, P_begin_y, P_begin_x)
PoseScreen.box()
PoseScreen.hline(2, 1, curses.ACS_HLINE, P_ncols-2)
PoseScreen.addstr(1,P_ncols/2 -3,"[P]ose")
PoseScreen.refresh()
```

Constants

- The solution to avoid using too many literals is to make effective use of **constants**
  - A constant is a named value that cannot change once it has been defined
- There are two main approaches available
  - Using the **#define** preprocessor directive
  - Using the **const** keyword
  - Use of the const keyword is usually preferred in modern C programming but doesn’t work in every situation (see future notes on arrays)

Remember: Name constants as ***ALL_CAPS_WITH_UNDERSCORES***

Constants – #define vs. const

The #define directive

```
#define NAME_OF_CONSTANT value
```

Note: there is ***no semicolon at the end*** *because this is not strictly a C-statement but a preprocessor directive. Adding a semicolon will break your programme.*

```
#define MAX_COUNT 20
#define DEFAULT_SPEED 0.5
```

The const keyword

```
const type NAME_OF_CONSTANT = value;
```

This is almost identical to a variable declaration except for the const keyword added to the start. ***Terminating the statement with a semicolon is mandatory!***

```
const int MAX_COUNT = 20;
const float DEFAULT_SPEED = 0.5;
```

Using constants to replace literals

```
// Example using #define

// Note: define constants at the top of the programme, before the setup and
// loop functions
#define PUSH_BUTTON_2 5
#define TWO_PI 6.26

  …
  pinMode(PUSH_BUTTON_2, INPUT);   // no comment necessary
  …
  result = r * TWO_PI;  // probably no comment necessary
```

```
// Example using const

// Note: declare and initialize const constants at the top of the programme,
// before the setup and loop functions
const int PUSH_BUTTON_2 = 5;
const float TWO_PI = 6.26;

  …
  pinMode(PUSH_BUTTON_2, INPUT);   // no comment necessary
  …
  result = r * TWO_PI;  // probably no comment necessary
```

Self test questions

Q1. Using (a) #define, and (b) the const keyword, declare a constant called TEMPERATURE with value 19.7.

Q2. In as many different ways as you can, declare/initialize the values of a constant to represent the highest floor in the building to which the elevator can go. Use a sensible constant name and the recommended naming style.

Variables

- Most useful programmes need to perform a series of calculations and require somewhere to store intermediate or temporary values
- Long lived programmes often need to keep track of values over time
- **Variables** are named memory locations used to store values while the programme is running
- A variable is so called, because its value can vary (i.e. be changed) over time

Variable declarations

A variable must be declared before it can be used

```
type variableName;
```

A declaration tells the C compiler how much memory to reserve to hold any value you subsequently give the variable. Specifically, the type specifies the amount of memory space required.

```
// Example variable declarations
int counter;
float speed;
```

Note: Declaring a variable as above does not tell the compiler what value the variable should have. In fact ***the variable's value is undefined at this time****, where undefined means some random value, not "empty" and not necessarily "zero".*

It is permitted to declare several variables on one line as shown below, but this can make code harder to read and should usually be avoided. In particular, don't do this in EE108!

```
int counter, result, code;
float xVal, yVal, zVal;
```

Remember: Name your variables using `lowerMixedCase`

Variable assignment and initialization

A variable must also have defined value before it can be used in an expression. Variables are given values by assignment, where the assignment operator is the single equals sign.

```
float speed;

// Example variable assignment (after the variable declaration)
speed = 0.5;
```

A variable can also be assigned an initial value at the point where it is declared.

```
// Example variable declarations including initial value
int counter = 4;
float speed = 0.5;
```

Initial values are also permitted when several variables are declared on a line. However this can make code even harder to read and should probably be avoided.

```
int a=1, b=2, c=3; // values a, b, and c are all initialized
int a, b, c=10;    // in this case only c is initialized,
                   // a and b have undefined values
```

Self test questions

Q1. Declare an integer variable called numBytes.

Q2. Using assignment, set the variable numBytes (declared above) to be 1000.

Q3. Combine the answers to Q1 and Q2 in a single statement.

Self test questions

Q1. In as many different ways as you can, declare and set the values of two variables to represent the following: the floor an elevator is currently at (2), and the floor to which it is going (5). Use sensible variable names and recommended naming style.

Types

- In C, every variable (and constant using const) must have a **type**
- The type tells the compiler how much memory storage to reserve for the variable
  - E.g. an int is stored in 2 bytes on the Arduino Uno
  - E.g. a float is stored in 4 bytes on the Arduino Uno
- C has two kinds of basic types
  - **Integral types** (representing a subset of mathematical integers)
  - **Floating point types** (representing a subset of real numbers)

Integral data types in standard C

Data type	Arduino Storage size in memory	Signed value range	Unsigned value range
`char`	8 bits / 1 byte	-128…127	0…255, i.e. 0…(2^8-1)
`short` / `short int`	16 bits / 2 bytes	-32768…32767	0…65535, i.e. 0…(2^{16}-1)
`int`	16 bits / 2 bytes	-32768…32767	0…65535, i.e. 0…(2^{16}-1)
`long` / `long int`	32 bits / 4 bytes	Approx -2e9…2e9	Approx 0…4e9, i.e. 0…(2^{32}-1)
`long long`	64 bits / 8 bytes	Approx -9e18…9e18	Approx 0…18e18, i.e. 0…(2^{64}-1)

The short, int, long, and long long types always default to signed values in C.
Whether a char type defaults to being signed or unsigned depends on the system – for Arduino it defaults to signed.

You can declare any integral type as explicitly signed or unsigned by adding the **`signed`** or **`unsigned`** **modifier keyword** as a prefix in the declaration, e.g.

```
int i;           // with no modifier, int defaults to signed
signed short j;  // explicitly marked as signed
unsigned char k; // explicitly marked as unsigned
```

Additional Integral data types in Arduino C

Data type	Storage size in memory	Equivalent to	values
`byte`	8 bits / 1 byte	`unsigned char`	0…255
`word`	16 bits / 2 bytes	`unsigned short / unsigned int`	0…65535

remember:
$255=2^8 - 1$
$65535 = 2^{16} - 1$

The byte and word types are useful with small embedded systems. The byte type allows you to be frugal with the available memory, using only one byte to store small numbers. The byte type is also commonly used in communications with external systems and devices (e.g. an LCD display).

The byte type also corresponds to a single register/memory location while the word type corresponds to a pair of adjacent registers/memory locations.

```
byte b = 255;  // max value of a byte
word w = 65535; // max value of a word
```

Integral types – which one to use?

Choose the smallest data type that is "big enough" for the range of values you need to use/store

Use unsigned values where appropriate

In particular, if a negative value makes no sense, it may be wise to use an unsigned data type

Floating point data types in C

Data type	Storage size on Arduino	Min precision	Max value
`float`	32 bits / 4 bytes	Approx 1e-38	Approx 3e38
`double`	32 bits / 4 bytes	Approx 1e-38	Approx 3e38
`long double`	32 bits / 4 bytes	Approx 1e-38	Approx 3e38

Floating point values are always signed. It is not possible to use the signed/unsigned modifier.

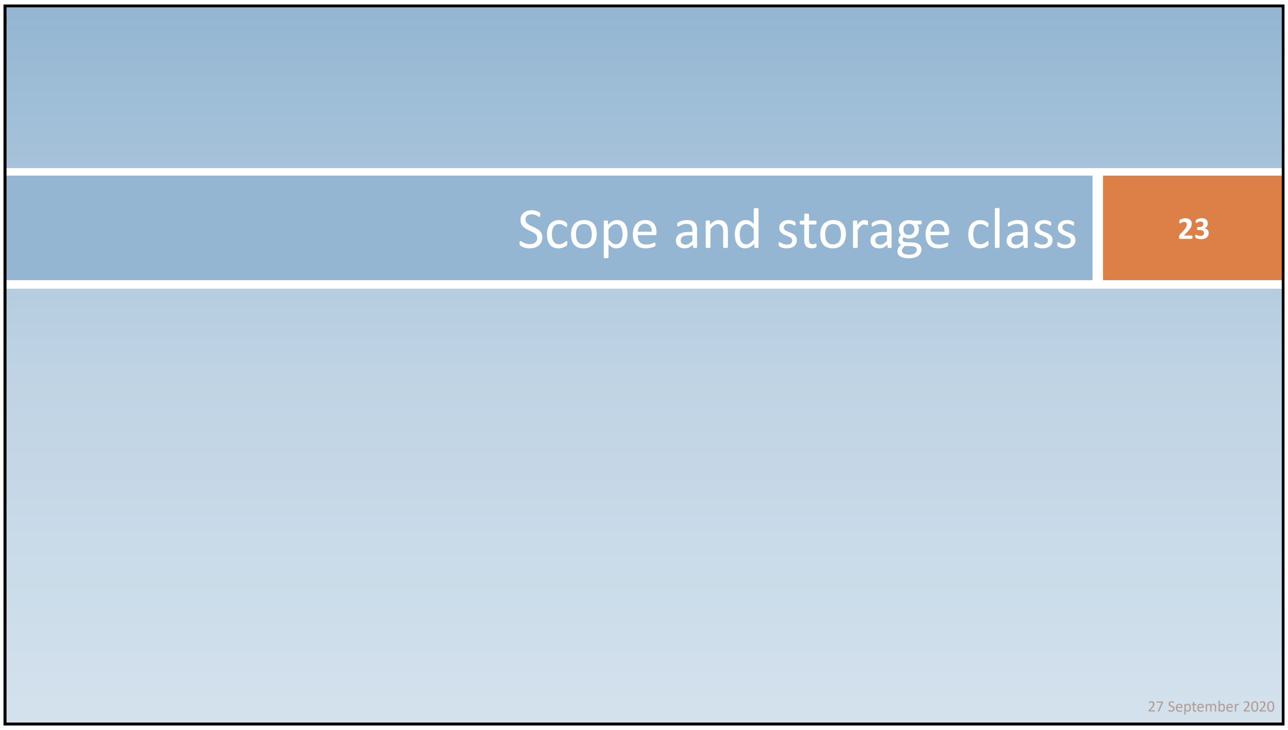
All floating point numbers have limited precision and are subject to rounding errors. E.g. in floating point the result of 0.3 + 0.6 is 0.89999999

Furthermore, a 32 bit float (or double) can only represent about 8 significant digits.

Self test questions

Q1. Declare a variable of appropriate type to represent a count of people entering a football stadium. The stadium capacity is 60,000. (Why is the capacity relevant?)

Q2. Declare a variable of appropriate type to represent the distance from a wheeled robot to a target (in meters).

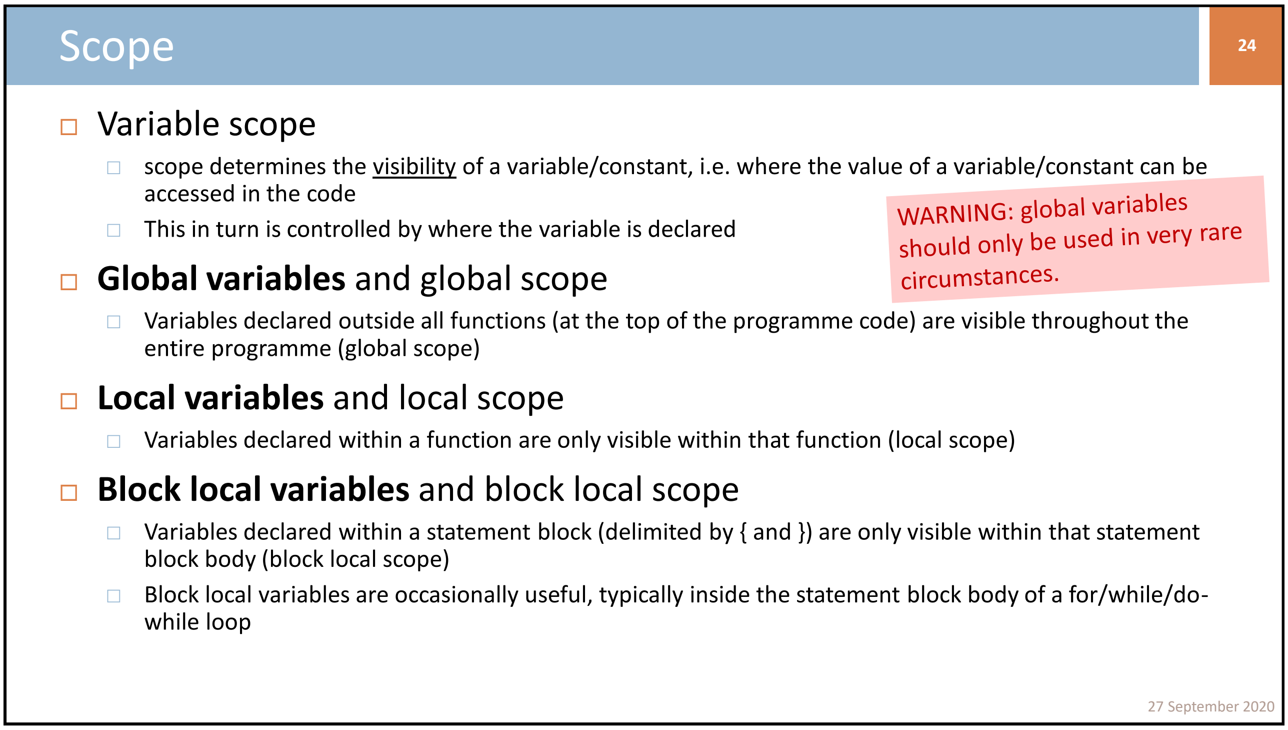


Scope and storage class

23

27 September 2020

23



Scope

24

- Variable scope
  - scope determines the visibility of a variable/constant, i.e. where the value of a variable/constant can be accessed in the code
  - This in turn is controlled by where the variable is declared
- **Global variables** and global scope
  - Variables declared outside all functions (at the top of the programme code) are visible throughout the entire programme (global scope)
- **Local variables** and local scope
  - Variables declared within a function are only visible within that function (local scope)
- **Block local variables** and block local scope
  - Variables declared within a statement block (delimited by { and }) are only visible within that statement block body (block local scope)
  - Block local variables are occasionally useful, typically inside the statement block body of a for/while/do-while loop

WARNING: global variables should only be used in very rare circumstances.

27 September 2020

24

Scope example

```
// global variables (declared outside of all functions)
int gSomeGlobal = 1; // this variable is accessible globally

// setup code goes here and runs once after the system resets
void setup()
{
  Serial.begin(9600);
}

// your main code goes in the loop function and is run repeatedly
void loop()
{
  int someLocal = 2; // this variable is accessible only in the loop function

  { // beginning of statement block
    int someBlockLocal = 3; // this variable is accessible only in this block
  } // end of statement block

  Serial.println(someBlockLocal); // not accessible in this scope – won’t compile
  Serial.println(someLocal); // OK – accessible throughout the loop function
  Serial.println(gSomeGlobal); // OK – accessible throughout programme

  delay(10000);
}
```

Storage class

- Storage class
  - The storage class of a variable (or constant) determines how long the memory used by the variable will be reserved for that variable
  - It defines the “lifetime” during which the variable’s location in memory is valid
- **Automatic storage**
  - With automatic storage the variable lifetime is tied to that of the enclosing scope so when the enclosing scope ends, the variable’s value gets forgotten
  - E.g. a local variable ceases to exist and its current value is forgotten when the function it is in ends/returns
  - Note: Local variables have automatic storage by default
- **Static storage**
  - The variable lifetime lasts as long as the programme runs so the variable value persists between function calls and scopes
  - Global variables have static storage by default
  - Local variables can have static storage specified using the **static** keyword

Storage class example

```
// global variables
int gSomeGlobal = 0; // this variable has static storage

void setup()
{
  Serial.begin(9600);
}

void loop()
{
  static int someStaticLocal = 100; // this variable has static storage
  int someLocal = 200; // this variable has automatic storage

  gSomeGlobal = gSomeGlobal + 1;
  someStaticLocal = someStaticLocal + 1;
  someLocal = someLocal + 1;

  // value of gSomeGlobal persists when loop returns, so keeps increasing
  Serial.println(gSomeGlobal);
  // value of someStaticLocal persists when loop returns, so keeps increasing
  Serial.println(someStaticLocal);
  // value of someLocal does not persist when loop returns, so always 201
  Serial.println(someLocal);

  delay(1000);
}
```

Storage and scope guidelines, additional details

Limit the scope of variables as much as possible.

Avoid global variables unless they are provably required.

Unless you specifically need a variable to be accessible from multiple functions declare it within a local scope just where it is needed. This is a key mechanism for abstraction/encapsulation.

If you need a local variable value to persist between function calls, declare it static.

An automatic local variable is reinitialized every time the function is entered. Similarly, an automatic block local variable is reinitialized every time the statement block is entered.

Static local variables and global variables are initialized just once. Globals are initialised at programme startup whereas static local variables are initialised the first time the enclosing function is executed.

A local variable with the same name as global variable "hides" the global variable within the function because the name will refer to the local variable. The behaviour with block local variables is similar except that they can also hide local variables.

Storage class example 2

29

```
int var1 = 11; // global variable
int var2 = 12; // global variable

…

void loop()
{
  int var2 = 220; // hides the global variable var2
  int var3 = 230;

  {
    int var1 = 3100; // hides the global variable var1
    int var3 = 3300; // hides the local variable var3 from enclosing scope

    Serial.println(var1); // prints 3100 -- uses the block local variable
    Serial.println(var2); // prints 220 -- uses the enclosing local variable
    Serial.println(var3); // prints 3300 -- uses the block local variable
  }
  Serial.println(var1); // prints 11 -- uses the global variable
  Serial.println(var2); // prints 220 -- uses the local variable
  Serial.println(var3); // prints 230 -- uses the local variable

  delay(1000);
}
```

27 September 2020

29

Scope / storage – valid combinations

30

27 September 2020

30

Self test questions

31

Q1. What is the appropriate scope for a temporary variable used during a calculation?

Q2. What is the appropriate storage for a local variable that remembers the milliseconds from the end of the loop function until the next time it runs.

Q3. How would you declare the variable in Q2?

27 September 2020

31

Final comments

32

- The following functions are useful and will be used in labs etc. – look them up in the Arduino help > Reference
  - delay(), delayMicroseconds()
  - millis(), micros()
  - Serial
    - begin()
    - print()
    - println()
  - pinModel(), digitalRead(), digitalWrite(), analogRead()
  - random()

27 September 2020

32