Variable
data-type

int
[Data Types]

Description
Integers are your primary data-type for number storage.

On the Arduino Uno (and other ATmega based boards) an int stores a 16-bit (2-byte) value. This yields a range of -32,768 to 32,767 (minimum value of -2^15 and a maximum value of (2^15) - 1). On the Arduino Due and SAMD based boards (like MKR1000 and Zero), an int stores a 32-bit (4-byte) value. This yields a range of -2,147,483,648 to 2,147,483,647 (minimum value of -2^31 and a maximum value of (2^31) - 1).

int’s store negative numbers with a technique called (2’s complement math). The highest bit, sometimes referred to as the "sign" bit, flags the number as a negative number. The rest of the bits are inverted and 1 is added.

The Arduino takes care of dealing with negative numbers for you, so that arithmetic operations work transparently in the expected manner. There can be an unexpected complication in dealing with the bitshift right operator (>>) however.

Syntax
```
int var = val;
```

`var` - your int variable name
`val` - the value you assign to that variable

Example Code
```
  int ledPin = 13;
```

Notes and Warnings
When signed variables are made to exceed their maximum or minimum capacity they *overflow*. The result of an overflow is unpredictable so this should be avoided. A typical symptom of an overflow is the variable "rolling over" from its maximum capacity to its minimum or vice versa, but this is not always the case. If you want this behavior, use unsigned int.

String

[Data Types]

Description

Text Strings can be represented in two ways. you can use the String data type, which is part of the core as of version 0019, or you can make a String out of an array of type char and null-terminate it. This page described the latter method. For more details on the String object, which gives you more functionality at the cost of more memory, see the String object page.

Syntax

All of the following are valid declarations for Strings.

```
char Str1[15];
char Str2[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o'};
char Str3[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o', '\0'};
char Str4[ ] = "arduino";
char Str5[8] = "arduino";
char Str6[15] = "arduino";
```

Possibilities for declaring Strings

- Declare an array of chars without initializing it as in Str1
- Declare an array of chars (with one extra char) and the compiler will add the required null character, as in Str2
- Explicitly add the null character, Str3
- Initialize with a String constant in quotation marks; the compiler will size the array to fit the String constant and a terminating null character, Str4
- Initialize the array with an explicit size and String constant, Str5
- Initialize the array, leaving extra space for a larger String, Str6

Null termination

Generally, Strings are terminated with a null character (ASCII code 0). This allows functions (like `Serial.print()`) to tell where the end of a String is. Otherwise, they would continue reading subsequent bytes of memory that aren’t actually part of the String.

This means that your String needs to have space for one more character than the text you want it to contain. That is why Str2 and Str5 need to be eight characters, even though "arduino" is only seven - the last position is automatically filled with a null character. Str4 will be automatically sized to eight characters, one for the extra null. In Str3, we’ve explicitly included the null character (written '\0') ourselves.

Note that it’s possible to have a String without a final null character (e.g. if you had specified the length of Str2 as seven instead of eight). This will break most functions that use Strings, so you shouldn’t do it intentionally. If you notice something behaving strangely (operating on characters not in the String), however, this could be the problem.

Single quotes or double quotes?

Strings are always defined inside double quotes ("Abc") and characters are always defined inside single quotes('A').

Wrapping long Strings

You can wrap long Strings like this:

```
char myString[] = "This is the first line"
" this is the second line"
" etcetera";
```

Arrays of Strings

It is often convenient, when working with large amounts of text, such as a project with an LCD display, to setup an array of Strings. Because Strings themselves are arrays, this is in actually an example of a two-dimensional array.

In the code below, the asterisk after the datatype `char` “char*” indicates that this is an array of “pointers”. All array names are actually pointers, so this is required to make an array of arrays. Pointers are one of the more esoteric parts of C for beginners to understand, but it isn’t necessary to understand pointers in detail to use them effectively here.

Example Code

```
char* myStrings[]={"This is String 1", "This is String 2", "This is
String 3",
"This is String 4", "This is String 5","This is String 6"};

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

void loop(){
for (int i = 0; i < 6; i++){
   Serial.println(myStrings[i]);
   delay(500);
   }
}
```

String()

[Data Types]

Description

Constructs an instance of the String class. There are multiple versions that construct Strings from different data types (i.e. format them as sequences of characters), including:

- a constant string of characters, in double quotes (i.e. a char array)
- a single constant character, in single quotes
- another instance of the String object
- a constant integer or long integer
- a constant integer or long integer, using a specified base
- an integer or long integer variable
- an integer or long integer variable, using a specified base
- a float or double, using a specified decimal places

Constructing a String from a number results in a string that contains the ASCII representation of that number. The default is base ten, so

```
String thisString = String(13);
```

gives you the String "13". You can use other bases, however. For example,

```
String thisString = String(13, HEX);
```

gives you the String "D", which is the hexadecimal representation of the decimal value 13. Or if you prefer binary,

```
String thisString = String(13, BIN);
```

gives you the String "1101", which is the binary representation of 13.

Syntax

```
String(val)
String(val, base)
String(val, decimalPlaces)
```

Parameters

`val`: a variable to format as a String - **Allowed data types:** string, char, byte, int, long, unsigned int, unsigned long, float, double `base` (optional): the base in which to format an integral value `decimalPlaces` (**only if val is float or double**): the desired decimal places

Returns

an instance of the String class.

Example Code

All of the following are valid declarations for Strings.

```
String stringOne = "Hello String";
// using a constant String
String stringOne =  String('a');
// converting a constant char into a String
String stringTwo =  String("This is a string");                 //
converting a constant string into a String object
String stringOne =  String(stringTwo + " with more"); //
concatenating two strings
String stringOne =  String(13);
// using a constant integer
String stringOne =  String(analogRead(0), DEC);          // using an
int and a base
String stringOne =  String(45,
HEX);                                // using an int and a base
(hexadecimal)
String stringOne =  String(255,
BIN);                               // using an int and a base
(binary)
String stringOne =  String(millis(), DEC);                        //
using a long and a base
String stringOne =  String(5.698,
3);                                // using a float and the decimal
places
```

array

[Data Types]

Description

An array is a collection of variables that are accessed with an index number. Arrays in the C programming language, on which Arduino is based, can be complicated, but using simple arrays is relatively straightforward.

Creating (Declaring) an Array

All of the methods below are valid ways to create (declare) an array.

```
  int myInts[6];
  int myPins[] = {2, 4, 8, 3, 6};
  int mySensVals[6] = {2, 4, -8, 3, 2};
  char message[6] = "hello";
```

You can declare an array without initializing it as in myInts.
In myPins we declare an array without explicitly choosing a size. The compiler counts the elements and creates an array of the appropriate size.
Finally you can both initialize and size your array, as in mySensVals. Note that when declaring an array of type char, one more element than your initialization is required, to hold the required null character.

Accessing an Array

Arrays are zero indexed, that is, referring to the array initialization above, the first element of the array is at index 0, hence

`mySensVals[0] == 2, mySensVals[1] == 4,` and so forth.

It also means that in an array with ten elements, index nine is the last element. Hence:

```
int myArray[10]={9,3,2,4,3,2,7,8,9,11};
     // myArray[9]    contains 11
     // myArray[10]   is invalid and contains random information
(other memory address)
```

For this reason you should be careful in accessing arrays. Accessing past the end of an array (using an index number greater than your declared array size - 1) is reading from memory that is in use for other purposes. Reading from these locations is probably not going to do much except yield invalid data. Writing to random memory locations is definitely a bad idea and can often lead to unhappy results such as crashes or program malfunction. This can also be a difficult bug to track down.

Unlike BASIC or JAVA, the C compiler does no checking to see if array access is within legal bounds of the array size that you have declared.

To assign a value to an array:

```
mySensVals[0] = 10;
```

To retrieve a value from an array:

```
x = mySensVals[4];
```

Arrays and FOR Loops

Arrays are often manipulated inside for loops, where the loop counter is used as the index for each array element. For example, to print the elements of an array over the serial port, you could do something like this:

```
int i;
for (i = 0; i < 5; i = i + 1) {
  Serial.println(myPins[i]);
}
```

Example Code

For a complete program that demonstrates the use of arrays, see the (Knight Rider example) from the (Tutorials).

Knight Rider Example a come in next page

Knight Rider

We have named this example in memory to a TV-series from the 80's where the famous David Hasselhoff had an AI machine driving his Pontiac. The car had been augmented with plenty of LEDs in all possible sizes performing flashy effects.

Thus we decided that in order to learn more about sequential programming and good programming techniques for the I/O board, it would be interesting to use the **Knight Rider** as a metaphor.

This example makes use of 6 LEDs connected to the pins 2 - 7 on the board using 220 Ohm resistors. The first code example will make the LEDs blink in a sequence, one by one using only **digitalWrite(pinNum,HIGH/LOW)** and **delay(time)**. The second example shows how to use a **for(;;)** construction to perform the very same thing, but in fewer lines. The third and last example concentrates in the visual effect of turning the LEDs on/off in a more softer way.

Example for Hasselhoff's fans

Knight Rider 1

```
/* Knight Rider 1
 * --------------
 *
 * Basically an extension of Blink_LED.
 *
 *
 * (cleft) 2005 K3, Malmo University
 * @author: David Cuartielles
 * @hardware: David Cuartielles, Aaron Hallborg
 */

int pin2 = 2;
int pin3 = 3;
int pin4 = 4;
int pin5 = 5;
int pin6 = 6;
int pin7 = 7;
int timer = 100;

void setup(){
  pinMode(pin2, OUTPUT);
  pinMode(pin3, OUTPUT);
  pinMode(pin4, OUTPUT);
  pinMode(pin5, OUTPUT);
  pinMode(pin6, OUTPUT);
  pinMode(pin7, OUTPUT);
}

void loop() {
   digitalWrite(pin2, HIGH);
   delay(timer);
   digitalWrite(pin2, LOW);
   delay(timer);

   digitalWrite(pin3, HIGH);
   delay(timer);
   digitalWrite(pin3, LOW);
   delay(timer);

   digitalWrite(pin4, HIGH);
   delay(timer);
   digitalWrite(pin4, LOW);
   delay(timer);

   digitalWrite(pin5, HIGH);
   delay(timer);
   digitalWrite(pin5, LOW);
   delay(timer);

   digitalWrite(pin6, HIGH);
   delay(timer);
   digitalWrite(pin6, LOW);
   delay(timer);

   digitalWrite(pin7, HIGH);
   delay(timer);
   digitalWrite(pin7, LOW);
   delay(timer);

   digitalWrite(pin6, HIGH);
   delay(timer);
   digitalWrite(pin6, LOW);
   delay(timer);

   digitalWrite(pin5, HIGH);
   delay(timer);
   digitalWrite(pin5, LOW);
   delay(timer);

   digitalWrite(pin4, HIGH);
   delay(timer);
   digitalWrite(pin4, LOW);
   delay(timer);

   digitalWrite(pin3, HIGH);
   delay(timer);
   digitalWrite(pin3, LOW);
   delay(timer);
}
```

Knight Rider 2

```
/* Knight Rider 2
 * --------------
 *
 * Reducing the amount of code using for(;;).
 *
 *
 * (cleft) 2005 K3, Malmo University
 * @author: David Cuartielles
 * @hardware: David Cuartielles, Aaron Hallborg
 */

int pinArray[] = {2, 3, 4, 5, 6, 7};
int count = 0;
int timer = 100;
```

```
void setup(){
  // we make all the declarations at once
  for (count=0;count<6;count++) {
    pinMode(pinArray[count], OUTPUT);
  }
}

void loop() {
  for (count=0;count<6;count++) {
   digitalWrite(pinArray[count], HIGH);
   delay(timer);
   digitalWrite(pinArray[count], LOW);
   delay(timer);
  }
  for (count=5;count>=0;count--) {
   digitalWrite(pinArray[count], HIGH);
   delay(timer);
   digitalWrite(pinArray[count], LOW);
   delay(timer);
  }
}
```

Knight Rider 3

```
/* Knight Rider 3
 * --------------
 *
 * This example concentrates on making the visuals fluid.
 *
 *
 * (cleft) 2005 K3, Malmo University
 * @author: David Cuartielles
 * @hardware: David Cuartielles, Aaron Hallborg
 */

int pinArray[] = {2, 3, 4, 5, 6, 7};
int count = 0;
int timer = 30;

void setup(){
  for (count=0;count<6;count++) {
    pinMode(pinArray[count], OUTPUT);
  }
}

void loop() {
  for (count=0;count<5;count++) {
   digitalWrite(pinArray[count], HIGH);
   delay(timer);
   digitalWrite(pinArray[count + 1], HIGH);
   delay(timer);
   digitalWrite(pinArray[count], LOW);
   delay(timer*2);
  }
  for (count=5;count>0;count--) {
   digitalWrite(pinArray[count], HIGH);
   delay(timer);
   digitalWrite(pinArray[count - 1], HIGH);
   delay(timer);
   digitalWrite(pinArray[count], LOW);
   delay(timer*2);
  }
}
```

bool

[Data Types]

Description

A `bool` holds one of two values, `true` or `false`. (Each `bool` variable occupies one byte of memory.)

Example Code

This code shows how to use the `bool` datatype.

```
int LEDpin = 5;       // LED on pin 5
int switchPin = 13;   // momentary switch on 13, other side connected
to ground

bool running = false;

void setup()
{
  pinMode(LEDpin, OUTPUT);
  pinMode(switchPin, INPUT);
  digitalWrite(switchPin, HIGH);      // turn on pullup resistor
}

void loop()
{
  if (digitalRead(switchPin) == LOW)
  {  // switch is pressed - pullup keeps pin high normally
    delay(100);                        // delay to debounce switch
    running = !running;                // toggle running variable
    digitalWrite(LEDpin, running);     // indicate via LED
  }
}
```

boolean

[Data Types]

Description

`boolean` is a non-standard type alias for bool defined by Arduino. It’s recommended to instead use the standard type `bool`, which is identical.

byte

[Data Types]

Description

A byte stores an 8-bit unsigned number, from 0 to 255.

char

[Data Types]

Description

A data type that takes up 1 byte of memory that stores a character value. Character literals are written in single quotes, like this: 'A' (for multiple characters - strings - use double quotes: "ABC").

Characters are stored as numbers however. You can see the specific encoding in the ASCII chart. This means that it is possible to do arithmetic on characters, in which the ASCII value of the character is used (e.g. 'A' + 1 has the value 66, since the ASCII value of the capital letter A is 65). See `Serial.println` reference for more on how characters are translated to numbers.

The char datatype is a signed type, meaning that it encodes numbers from -128 to 127. For an unsigned, one-byte (8 bit) data type, use the *byte* data type.

Example Code

```
  char myChar = 'A';
  char myChar = 65;      // both are equivalent
```

double

[Data Types]

Description

Double precision floating point number. On the Uno and other ATMEGA based boards, this occupies 4 bytes. That is, the double implementation is exactly the same as the float, with no gain in precision.

On the Arduino Due, doubles have 8-byte (64 bit) precision.

Notes and Warnings

Users who borrow code from other sources that includes double variables may wish to examine the code to see if the implied precision is different from that actually achieved on ATMEGA based Arduinos.

float

[Data Types]

Description

Datatype for floating-point numbers, a number that has a decimal point. Floating-point numbers are often used to approximate analog and continuous values because they have greater resolution than integers. Floating-point numbers can be as large as 3.4028235E+38 and as low as -3.4028235E+38. They are stored as 32 bits (4 bytes) of information.

Floats have only 6-7 decimal digits of precision. That means the total number of digits, not the number to the right of the decimal point. Unlike other platforms, where you can get more precision by using a double (e.g. up to 15 digits), on the Arduino, double is the same size as float.

Floating point numbers are not exact, and may yield strange results when compared. For example 6.0 / 3.0 may not equal 2.0. You should instead check that the absolute value of the difference between the numbers is less than some small number.

Floating point math is also much slower than integer math in performing calculations, so should be avoided if, for example, a loop has to run at top speed for a critical timing function. Programmers often go to some lengths to convert floating point calculations to integer math to increase speed.

If doing math with floats, you need to add a decimal point, otherwise it will be treated as an int. See the Floating point constants page for details.

Syntax

`float var=val;`

`var` - your float variable name `val` - the value you assign to that variable

Example Code

```
  float myfloat;
  float sensorCalbrate = 1.117;

  int x;
  int y;
  float z;

  x = 1;
  y = x / 2;            // y now contains 0, ints can't hold
fractions
  z = (float)x / 2.0;   // z now contains .5 (you have to use 2.0,
not 2)
```

long

[Data Types]

Description

Long variables are extended size variables for number storage, and store 32 bits (4 bytes), from -2,147,483,648 to 2,147,483,647.

If doing math with integers, at least one of the numbers must be followed by an L, forcing it to be a long. See the Integer Constants page for details.

Syntax

`long var = val;`

`var` - the long variable name `val` - the value assigned to the variable

Example Code

```
  long speedOfLight = 186000L;   // see the Integer Constants page
for explanation of the 'L'
```

short

[Data Types]

Description

A short is a 16-bit data-type.

On all Arduinos (ATMega and ARM based) a short stores a 16-bit (2-byte) value. This yields a range of -32,768 to 32,767 (minimum value of -2^15 and a maximum value of (2^15) - 1).

Syntax

```
short var = val;
```

`var` - your short variable name `val` - the value you assign to that variable

Example Code

```
 short ledPin = 13
```

unsigned char

[Data Types]

Description

An unsigned data type that occupies 1 byte of memory. Same as the byte datatype.

The unsigned char datatype encodes numbers from 0 to 255.

For consistency of Arduino programming style, the byte data type is to be preferred.

Example Code

```
unsigned char myChar = 240;
```

void

[Data Types]

Description

The `void` keyword is used only in function declarations. It indicates that the function is expected to return no information to the function from which it was called.

Example Code

The code shows how to use `void`.

```
// actions are performed in the functions "setup" and "loop"
// but  no information is reported to the larger program

void setup()
{
  // ...
}

void loop()
{
  // ...
}
```

word

[Data Types]

Description

A word stores a 16-bit unsigned number, from 0 to 65535. Same as an unsigned int.

Example Code

```
  word w = 10000;
```

unsigned long

[Data Types]

Description

Unsigned long variables are extended size variables for number storage, and store 32 bits (4 bytes). Unlike standard longs unsigned longs won’t store negative numbers, making their range from 0 to 4,294,967,295 (2^32 - 1).

Syntax

```
unsigned long var = val;
```

`var` - your long variable name `val` - the value you assign to that variable

Example Code

```
unsigned long time;

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

```
void loop()
{
  Serial.print("Time: ");
  time = millis();
  //prints time since program started
  Serial.println(time);
  // wait a second so as not to send massive amounts of data
  delay(1000);
}
```

unsigned int

[Data Types]

Description

On the Uno and other ATMEGA based boards, unsigned ints (unsigned integers) are the same as ints in that they store a 2 byte value. Instead of storing negative numbers however they only store positive values, yielding a useful range of 0 to 65,535 ((2^16) - 1).

The Due stores a 4 byte (32-bit) value, ranging from 0 to 4,294,967,295 (2^32 - 1).

The difference between unsigned ints and (signed) ints, lies in the way the highest bit, sometimes referred to as the "sign" bit, is interpreted. In the Arduino int type (which is signed), if the high bit is a "1", the number is interpreted as a negative number, and the other 15 bits are interpreted with (2’s complement math).

Syntax

`unsigned int var = val;` `var` - your unsigned int variable name `val` - the value you assign to that variable

Example Code

```
  unsigned int ledPin = 13;
```

Notes and Warnings

When unsigned variables are made to exceed their maximum capacity they "roll over" back to 0, and also the other way around:

```
unsigned int x;
   x = 0;
   x = x - 1;       // x now contains 65535 - rolls over in neg
direction
   x = x + 1;       // x now contains 0 - rolls over
```

Math with unsigned variables may produce unexpected results, even if your unsigned variable never rolls over.

The MCU applies the following rules:

The calculation is done in the scope of the destination variable. E.g. if the destination variable is signed, it will do signed math, even if both input variables are unsigned.

However with a calculation which requires an intermediate result, the scope of the intermediate result is unspecified by the code. In this case, the MCU will do unsigned math for the intermediate result, because both inputs are unsigned!

```
unsigned int x=5;
unsigned int y=10;
int result;

   result = x - y; // 5 - 10 = -5, as expected
   result = (x - y)/2; // 5 - 10 in unsigned math is 65530!  65530/2
= 32765

   // solution: use signed variables, or do the calculation step by
step.
   result = x - y; // 5 - 10 = -5, as expected
   result = result / 2; //  -5/2 = -2 (only integer math, decimal
places are dropped)
```

Why use unsigned variables at all?

- The rollover behaviour is desired, e.g. counters
- The signed variable is a bit too small, but you want to avoid the memory and speed loss of long/ float.

Constants

Integer Constants

[Constants]

Description

Integer constants are numbers that are used directly in a sketch, like 123. By default, these numbers are treated as int but you can change this with the U and L modifiers (see below).

Normally, integer constants are treated as base 10 (decimal) integers, but special notation (formatters) may be used to enter numbers in other bases.

Base	Example	Formatter	Comment
10 (decimal)	123	none	
2 (binary)	B1111011	leading 'B'	only works with 8 bit values (0 to 255) characters 0&1 valid
8 (octal)	0173	leading "0"	characters 0-7 valid
16 (hexadecimal)	0x7B	leading "0x"	characters 0-9, A-F, a-f valid

Decimal (base 10)

This is the common-sense math with which you are acquainted. Constants without other prefixes are assumed to be in decimal format.

Example Code:
```
n = 101;     // same as 101 decimal   ((1 * 10^2) + (0 * 10^1) + 1)
```

Binary (base 2)

Only the characters 0 and 1 are valid.

Example Code:
```
n = B101;    // same as 5 decimal   ((1 * 2^2) + (0 * 2^1) + 1)
```

The binary formatter only works on bytes (8 bits) between 0 (B0) and 255 (B11111111). If it is convenient to input an int (16 bits) in binary form you can do it a two-step procedure such as:

```
myInt = (B11001100 * 256) + B10101010;    // B11001100 is the high
byte`
```

Octal (base 8)

Only the characters 0 through 7 are valid. Octal values are indicated by the prefix "0" (zero).

Example Code:

```
n = 0101;    // same as 65 decimal   ((1 * 8^2) + (0 * 8^1) + 1)
```

It is possible to generate a hard-to-find bug by (unintentionally) including a leading zero before a constant and having the compiler unintentionally interpret your constant as octal.

Hexadecimal (base 16)

Valid characters are 0 through 9 and letters A through F; A has the value 10, B is 11, up to F, which is 15. Hex values are indicated by the prefix "0x". Note that A-F may be syted in upper or lower case (a-f).

Example Code:

```
n = 0x101;   // same as 257 decimal   ((1 * 16^2) + (0 * 16^1) + 1)
```

Notes and Warnings

U & L formatters:

By default, an integer constant is treated as an int with the attendant limitations in values. To specify an integer constant with another data type, follow it with:

- a 'u' or 'U' to force the constant into an unsigned data format. Example: 33u
- a 'l' or 'L' to force the constant into a long data format. Example: 100000L
- a 'ul' or 'UL' to force the constant into an unsigned long constant. Example: 32767ul

Floating Point Constants

[Constants]

Description

Similar to integer constants, floating point constants are used to make code more readable. Floating point constants are swapped at compile time for the value to which the expression evaluates.

Example Code

```
n = 0.005;  // 0.005 is a floating point constant
```

Notes and Warnings

Floating point constants can also be expressed in a variety of scientific notation. 'E' and 'e' are both accepted as valid exponent indicators.

floating-point constant	evaluates to:	also evaluates to:
10.0	10	
2.34E5	2.34 * 10^5	234000
67e-12	67.0 * 10^-12	0.000000000067

constants

[Constants]

Description

Constants are predefined expressions in the Arduino language. They are used to make the programs easier to read. We classify constants in groups:

Defining Logical Levels: true and false (Boolean Constants)

There are two constants used to represent truth and falsity in the Arduino language: `true`, and `false`.

false

`false` is the easier of the two to define. false is defined as 0 (zero).

true

`true` is often said to be defined as 1, which is correct, but true has a wider definition. Any integer which is non-zero is true, in a Boolean sense. So -1, 2 and -200 are all defined as true, too, in a Boolean sense.

Note that the `true` and `false` constants are typed in lowercase unlike `HIGH`, `LOW`, `INPUT`, and `OUTPUT`.

Defining Pin Levels: HIGH and LOW

When reading or writing to a digital pin there are only two possible values a pin can take/be-set-to: `HIGH` and `LOW`.

HIGH

The meaning of `HIGH` (in reference to a pin) is somewhat different depending on whether a pin is set to an `INPUT` or `OUTPUT`. When a pin is configured as an `INPUT` with pinMode(), and read with digitalRead(), the Arduino (ATmega) will report `HIGH` if:

- a voltage greater than 3.0V is present at the pin (5V boards)
- a voltage greater than 2.0V volts is present at the pin (3.3V boards)

A pin may also be configured as an INPUT with `pinMode()`, and subsequently made HIGH with digitalWrite(). This will enable the internal 20K pullup resistors, which will *pull up* the input pin to a `HIGH` reading unless it is pulled `LOW` by external circuitry. This is how `INPUT_PULLUP` works and is described below in more detail.

When a pin is configured to OUTPUT with `pinMode()`, and set to `HIGH` with `digitalWrite()`, the pin is at:

- 5 volts (5V boards)
- 3.3 volts (3.3V boards)

In this state it can source current, e.g. light an LED that is connected through a series resistor to ground.

LOW

The meaning of `LOW` also has a different meaning depending on whether a pin is set to `INPUT` or `OUTPUT`. When a pin is configured as an `INPUT` with `pinMode()`, and read with `digitalRead()`, the Arduino (ATmega) will report LOW if:

- a voltage less than 1.5V is present at the pin (5V boards)
- a voltage less than 1.0V (Approx) is present at the pin (3.3V boards)

When a pin is configured to `OUTPUT` with `pinMode()`, and set to `LOW` with `digitalWrite()`, the pin is at 0 volts (both 5V and 3.3V boards). In this state it can sink current, e.g. light an LED that is connected through a series resistor to +5 volts (or +3.3 volts).

Defining Digital Pins modes: INPUT, INPUT_PULLUP, and OUTPUT

Digital pins can be used as `INPUT`, `INPUT_PULLUP`, or `OUTPUT`. Changing a pin with `pinMode()` changes the electrical behavior of the pin.

Pins Configured as INPUT

Arduino (ATmega) pins configured as `INPUT` with `pinMode()` are said to be in a *high-impedance* state. Pins configured as `INPUT` make extremely small demands on the circuit that they are sampling, equivalent to a series resistor of 100 Megohms in front of the pin. This makes them useful for reading a sensor.

If you have your pin configured as an `INPUT`, and are reading a switch, when the switch is in the open state the input pin will be "floating", resulting in unpredictable results. In order to assure a proper reading when the switch is open, a pull-up or pull-down resistor must be used. The purpose of this resistor is to pull the pin to a known state when the switch is open. A 10 K ohm resistor is usually chosen, as it is a low enough value to reliably prevent a floating input, and at the same time a high enough value to not not draw too much current when the switch is closed. See the Digital Read Serial tutorial for more information.

If a pull-down resistor is used, the input pin will be `LOW` when the switch is open and `HIGH` when the switch is closed.

If a pull-up resistor is used, the input pin will be `HIGH` when the switch is open and `LOW` when the switch is closed.

Pins Configured as INPUT_PULLUP

The ATmega microcontroller on the Arduino has internal pull-up resistors (resistors that connect to power internally) that you can access. If you prefer to use these instead of external pull-up resistors, you can use the `INPUT_PULLUP` argument in `pinMode()`.

See the Input Pullup Serial tutorial for an example of this in use.

Pins configured as inputs with either `INPUT` or `INPUT_PULLUP` can be damaged or destroyed if they are connected to voltages below ground (negative voltages) or above the positive power rail (5V or 3V).

Pins Configured as OUTPUT

Pins configured as `OUTPUT` with `pinMode()` are said to be in a *low-impedance* state. This means that they can provide a substantial amount of current to other circuits. ATmega pins can source (provide current) or sink (absorb current) up to 40 mA (milliamps) of current to other devices/circuits. This makes them useful for powering LEDs because LEDs typically use less than 40 mA. Loads greater than 40 mA (e.g. motors) will require a transistor or other interface circuitry.

Pins configured as outputs can be damaged or destroyed if they are connected to either the ground or positive power rails.

Defining built-ins: LED_BUILTIN

Most Arduino boards have a pin connected to an on-board LED izn series with a resistor. The constant `LED_BUILTIN` is the number of the pin to which the on-board LED is connected. Most boards have this LED connected to digital pin 13.

Conversion

byte()

[Conversion]

Description

Converts a value to the byte data type.

Syntax

`byte(x)`

Parameters

`x`: a value of any type

Returns

`byte`

char()

[Conversion]

Description

Converts a value to the char data type.

Syntax

`char(x)`

Parameters

`x`: a value of any type

Returns

`char`

float()

[Conversion]

Description

Converts a value to the float data type.

Syntax

`float(x)`

Parameters

`x`: a value of any type

Returns

`float`

Notes and Warnings

See the reference for float for details about the precision and limitations of floating point numbers on Arduino.

int()

[Conversion]

Description

Converts a value to the int data type.

Syntax

`int(x)`

Parameters

`x`: a value of any type

Returns

`int`

long()

[Conversion]

Description

Converts a value to the long data type.

Syntax

```
long(x)
```

Parameters

`x`: a value of any type

Returns

```
long
```

word()

[Conversion]

Description

Converts a value to the word data type.

Syntax

```
word(x)
word(h, l)
```

Parameters

`x`: a value of any type

`h`: the high-order (leftmost) byte of the word

`l`: the low-order (rightmost) byte of the word

Returns

```
word
```

Variable Scope & Qualifiers

Const

[Variable Scope & Qualifiers]

Description

The `const` keyword stands for constant. It is a variable *qualifier* that modifies the behavior of the variable, making a variable "*read-only*". This means that the variable can be used just as any other variable of its type, but its value cannot be changed. You will get a compiler error if you try to assign a value to a `const` variable.

Constants defined with the `const` keyword obey the rules of variable scoping that govern other variables. This, and the pitfalls of using `#define`, makes the `const` keyword a superior method for defining constants and is preferred over using `#define`.

Example Code

```
const float pi = 3.14;
float x;

// ....

x = pi * 2;    // it's fine to use consts in math

pi = 7;        // illegal - you can't write to (modify) a constant
```

Notes and Warnings

`#define` or `const`

You can use either `const` or `#define` for creating numeric or string constants. For arrays, you will need to use `const`. In general `const` is preferred over `#define` for defining constants.

scope

[Variable Scope & Qualifiers]

Description

Variables in the C programming language, which Arduino uses, have a property called scope. This is in contrast to early versions of languages such as BASIC where every variable is a *global* variable.

A global variable is one that can be seen by every function in a program. Local variables are only visible to the function in which they are declared. In the Arduino environment, any variable declared outside of a function (e.g. setup(), loop(), etc.), is a *global* variable.

When programs start to get larger and more complex, local variables are a useful way to insure that only one function has access to its own variables. This prevents programming errors when one function inadvertently modifies variables used by another function.

It is also sometimes handy to declare and initialize a variable inside a `for` loop. This creates a variable that can only be accessed from inside the for-loop brackets.

Example Code

```
int gPWMval;  // any function will see this variable

void setup()
{
  // ...
}

void loop()
{
  int i;    // "i" is only "visible" inside of "loop"
  float f;  // "f" is only "visible" inside of "loop"
  // ...

  for (int j = 0; j <100; j++){
  // variable j can only be accessed inside the for-loop brackets
  }

}
```

static

[Variable Scope & Qualifiers]

Description

The `static` keyword is used to create variables that are visible to only one function. However unlike local variables that get created and destroyed every time a function is called, static variables persist beyond the function call, preserving their data between function calls.

Variables declared as static will only be created and initialized the first time a function is called.

Example Code

```
/* RandomWalk
* Paul Badger 2007
* RandomWalk wanders up and down randomly between two
* endpoints. The maximum move in one loop is governed by
* the parameter "stepsize".
* A static variable is moved up and down a random amount.
* This technique is also known as "pink noise" and "drunken walk".
*/

#define randomWalkLowRange -20
#define randomWalkHighRange 20
int stepsize;

int thisTime;
int total;

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

void loop()
{        //  test randomWalk function
  stepsize = 5;
  thisTime = randomWalk(stepsize);
  Serial.println(thisTime);
   delay(10);
}

int randomWalk(int moveSize){
  static int  place;     // variable to store value in random walk - declared
static so that it stores
                         // values in between function calls, but no other
functions can change its value

  place = place + (random(-moveSize, moveSize + 1));

  if (place < randomWalkLowRange){                              // check lower and
upper limits
    place = randomWalkLowRange + (randomWalkLowRange - place);  // reflect number
back in positive direction
  }
  else if(place > randomWalkHighRange){
    place = randomWalkHighRange - (place - randomWalkHighRange);  // reflect number
back in negative direction
  }

  return place;
}
```

volatile

[Variable Scope & Qualifiers]

Description

`volatile` is a keyword known as a variable *qualifier*, it is usually used before the datatype of a variable, to modify the way in which the compiler and subsequent program treats the variable.

Declaring a variable volatile is a directive to the compiler. The compiler is software which translates your C/C++ code into the machine code, which are the real instructions for the Atmega chip in the Arduino.

Specifically, it directs the compiler to load the variable from RAM and not from a storage register, which is a temporary memory location where program variables are stored and manipulated. Under certain conditions, the value for a variable stored in registers can be inaccurate.

A variable should be declared volatile whenever its value can be changed by something beyond the control of the code section in which it appears, such as a concurrently executing thread. In the Arduino, the only place that this is likely to occur is in sections of code associated with interrupts, called an interrupt service routine.

int or long volatiles

If the volatile variable is bigger than a byte (e.g. a 16 bit int or a 32 bit long), then the microcontroller can not read it in one step, because it is an 8 bit microcontroller. This means that while your main code section (e.g. your loop) reads the first 8 bits of the variable, the interrupt might already change the second 8 bits. This will produce random values for the variable.

Remedy:

While the variable is read, interrupts need to be disabled, so they can’t mess with the bits, while they are read. There are several ways to do this:

1. noInterrupts
2. use the ATOMIC_BLOCK macro. Atomic operations are single MCU operations - the smallest possible unit.

Example Code

```
// toggles LED when interrupt pin changes state

int pin = 13;
volatile byte state = LOW;

void setup()
{
  pinMode(pin, OUTPUT);
  attachInterrupt(0, blink, CHANGE);
}

void loop()
{
  digitalWrite(pin, state);
}

void blink()
{
  state = !state;
}
```

```
#include <util/atomic.h> // this library includes the ATOMIC_BLOCK macro.
volatile int input_from_interrupt;

  ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
     // code with interrupts blocked (consecutive atomic operations will not get
interrupted)
     int result = input_from_interrupt;
   }
```

Utilities

PROGMEM

[Utilities]

Description

Store data in flash (program) memory instead of SRAM. There’s a description of the various types of memory available on an Arduino board.

The `PROGMEM` keyword is a variable modifier, it should be used only with the datatypes defined in pgmspace.h. It tells the compiler "put this information into flash memory", instead of into SRAM, where it would normally go.

PROGMEM is part of the pgmspace.h library. It is included automatically in modern versions of the IDE, however if you are using an IDE version below 1.0 (2011), you’ll first need to include the library at the top your sketch, like this:

```
#include <avr/pgmspace.h>
```

Syntax

const dataType variableName[] PROGMEM = {data0, data1, data3…};

`dataType` - any variable type
`variableName` - the name for your array of data

Note that because PROGMEM is a variable modifier, there is no hard and fast rule about where it should go, so the Arduino compiler accepts all of the definitions below, which are also synonymous. However experiments have indicated that, in various versions of Arduino (having to do with GCC version), PROGMEM may work in one location and not in another. The "string table" example below has been tested to work with Arduino 13. Earlier versions of the IDE may work better if PROGMEM is included after the variable name.

```
const dataType variableName[] PROGMEM = {}; // use this form
const PROGMEM dataType variableName[] = {}; // or this one
const dataType PROGMEM variableName[] = {}; // not this one
```

While `PROGMEM` could be used on a single variable, it is really only worth the fuss if you have a larger block of data that needs to be stored, which is usually easiest in an array, (or another C data structure beyond our present discussion).

Using `PROGMEM` is also a two-step procedure. After getting the data into Flash memory, it requires special methods (functions), also defined in the pgmspace.h library, to read the data from program memory back into SRAM, so we can do something useful with it.

Example Code

The following code fragments illustrate how to read and write unsigned chars (bytes) and ints (2 bytes) to PROGMEM.

```
// save some unsigned ints
const PROGMEM  uint16_t charSet[]  = { 65000, 32796, 16843, 10, 11234};

// save some chars
const char signMessage[] PROGMEM  = {"I AM PREDATOR,  UNSEEN COMBATANT. CREATED BY 
THE UNITED STATES DEPART"};

unsigned int displayInt;
int k;    // counter variable
char myChar;


void setup() {
  Serial.begin(9600);
  while (!Serial);  // wait for serial port to connect. Needed for native USB

  // put your setup code here, to run once:
  // read back a 2-byte int
  for (k = 0; k < 5; k++)
  {
    displayInt = pgm_read_word_near(charSet + k);
    Serial.println(displayInt);
  }
  Serial.println();

  // read back a char
  for (k = 0; k < strlen_P(signMessage); k++)
  {
    myChar =  pgm_read_byte_near(signMessage + k);
    Serial.print(myChar);
  }

  Serial.println();
}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Arrays of strings

It is often convenient when working with large amounts of text, such as a project with an LCD display, to setup an array of strings. Because strings themselves are arrays, this is in actually an example of a two-dimensional array.

These tend to be large structures so putting them into program memory is often desirable. The code below illustrates the idea.

```
/*
 PROGMEM string demo
 How to store a table of strings in program memory (flash),
 and retrieve them.

 Information summarized from:
 http://www.nongnu.org/avr-libc/user-manual/pgmspace.html

 Setting up a table (array) of strings in program memory is slightly complicated, 
but
 here is a good template to follow.

 Setting up the strings is a two-step process. First define the strings.
*/

#include <avr/pgmspace.h>
const char string_0[] PROGMEM = "String 0";   // "String 0" etc are strings to 
store - change to suit.
const char string_1[] PROGMEM = "String 1";
const char string_2[] PROGMEM = "String 2";
const char string_3[] PROGMEM = "String 3";
const char string_4[] PROGMEM = "String 4";
const char string_5[] PROGMEM = "String 5";
```

```
// Then set up a table to refer to your strings.

const char* const string_table[] PROGMEM = {string_0, string_1, string_2, string_3,
string_4, string_5};

char buffer[30];    // make sure this is large enough for the largest string it
must hold

void setup()
{
  Serial.begin(9600);
  while(!Serial); // wait for serial port to connect. Needed for native USB
  Serial.println("OK");
}


void loop()
{
  /* Using the string table in program memory requires the use of special functions
to retrieve the data.
     The strcpy_P function copies a string from program space to a string in RAM
("buffer").
     Make sure your receiving string in RAM  is large enough to hold whatever
     you are retrieving from program space. */


  for (int i = 0; i < 6; i++)
  {
    strcpy_P(buffer, (char*)pgm_read_word(&(string_table[i]))); // Necessary casts
and dereferencing, just copy.
    Serial.println(buffer);
    delay( 500 );
  }
}
```

Notes and Warnings

Please note that variables must be either globally defined, OR defined with the static keyword, in order to work with PROGMEM.

The following code will NOT work when inside a function:

```
const char long_str[] PROGMEM = "Hi, I would like to tell you a bit about myself.\
n";
```

The following code WILL work, even if locally defined within a function:

```
const static char long_str[] PROGMEM = "Hi, I would like to tell you a bit about
myself.\n"
```

The `F()` macro

When an instruction like :

```
Serial.print("Write something on  the Serial Monitor");
```

is used, the string to be printed is normally saved in RAM. If your sketch prints a lot of stuff on the Serial Monitor, you can easily fill the RAM. If you have free FLASH memory space, you can easily indicate that the string must be saved in FLASH using the syntax:

```
Serial.print(F("Write something on the Serial Monitor that is stored in FLASH"));
```

sizeof()

[Utilities]

Description

The sizeof operator returns the number of bytes in a variable type, or the number of bytes occupied by an array.

Syntax

`sizeof(variable)`

Parameters

`variable`: any variable type or array (e.g. int, float, byte)

Returns

The number of bytes in a variable or bytes occupied in an array. (size_t)

Example Code

The `sizeof` operator is useful for dealing with arrays (such as strings) where it is convenient to be able to change the size of the array without breaking other parts of the program.

This program prints out a text string one character at a time. Try changing the text phrase.

```
char myStr[] = "this is a test";
int i;

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

void loop() {
  for (i = 0; i < sizeof(myStr) - 1; i++){
    Serial.print(i, DEC);
    Serial.print(" = ");
    Serial.write(myStr[i]);
    Serial.println();
  }
  delay(5000); // slow down the program
}
```

Notes and Warnings

Note that `sizeof` returns the total number of bytes. So for larger variable types such as ints, the for loop would look something like this. Note also that a properly formatted string ends with the NULL symbol, which has ASCII value 0.

```
for (i = 0; i < (sizeof(myInts)/sizeof(int)); i++) {
  // do something with myInts[i]
}
```