The elements of Arduino (C++) code

https://www.arduino.cc/reference/en/

Contents

Sketch

setup()

The `setup()` function is called when a sketch starts. Use it to initialize variables, pin modes, start using libraries, etc. The `setup()` function will only run once, after each powerup or reset of the Arduino board.

Example Code

```
int buttonPin = 3;

void setup()
{
  Serial.begin(9600);
  pinMode(buttonPin, INPUT);
}

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

loop()

After creating a setup() function, which initializes and sets the initial values, the `loop()` function does precisely what its name suggests, and loops consecutively, allowing your program to change and respond. Use it to actively control the Arduino board.

Example Code

```
int buttonPin = 3;

// setup initializes serial and the button pin
void setup()
{
  Serial.begin(9600);
  pinMode(buttonPin, INPUT);
}

// loop checks the button pin each time,
// and will send serial if it is pressed
void loop()
{
  if (digitalRead(buttonPin) == HIGH)
    Serial.write('H');
  else
    Serial.write('L');

  delay(1000);
}
```

Control Structure

break

`break` is used to exit from a for, while or do…while loop, bypassing the normal loop condition. It is also used to exit from a switch case statement.

Example Code

In the following code, the control exits the `for` loop when the sensor value exceeds the threshold.

```
for (x = 0; x < 255; x ++)
{
    analogWrite(PWMpin, x);
    sens = analogRead(sensorPin);
    if (sens > threshold){      // bail out on sensor detect
       x = 0;
       break;
    }
    delay(50);
}
```

continue

The `continue` statement skips the rest of the current iteration of a loop (for, while, or do… while). It continues by checking the conditional expression of the loop, and proceeding with any subsequent iterations.

Example Code

The following code writes the value of 0 to 255 to the `PWMpin`, but skips the values in the range of 41 to 119.

```
for (x = 0; x <= 255; x ++)
{
    if (x > 40 && x < 120){      // create jump in values
        continue;
    }

    analogWrite(PWMpin, x);
    delay(50);
}
```

do...while

The `do…while` loop works in the same manner as the while loop, with the exception that the condition is tested at the end of the loop, so the do loop will always run at least once.

Syntax

```
do
{
    // statement block
} while (condition);
```

The `condition` is a boolean expression that evaluates to `true` or `false`.

Example Code

```
do
{
  delay(50);          // wait for sensors to stabilize
  x = readSensors();  // check the sensors

} while (x < 100);
```

If-else

The `if…else` allows greater control over the flow of code than the basic if statement, by allowing multiple tests to be grouped together. An `else` clause (if at all exists) will be executed if the condition in the `if` statement results in `false`. The `else` can proceed another `if` test, so that multiple, mutually exclusive tests can be run at the same time.

Each test will proceed to the next one until a true test is encountered. When a true test is found, its associated block of code is run, and the program then skips to the line following the entire if/else construction. If no test proves to be true, the default `else` block is executed, if one is present, and sets the default behavior.

Note that an `else if` block may be used with or without a terminating `else` block and vice versa. An unlimited number of such `else if` branches is allowed.

Syntax

```
if (condition1)
{
  // do Thing A
}
else if (condition2)
{
  // do Thing B
}
else
{
  // do Thing C
}
```

Example Code

Below is an extract from a code for temperature sensor system

```
if (temperature >= 70)
{
  //Danger! Shut down the system
}
else if (temperature >= 60 && temperature < 70)
{
  //Warning! User attention required
}
else
{
  //Safe! Continue usual tasks...
}
```

for

The `for` statement is used to repeat a block of statements enclosed in curly braces. An increment counter is usually used to increment and terminate the loop. The `for` statement is useful for any repetitive operation, and is often used in combination with arrays to operate on collections of data/pins.

Syntax

```
for (initialization; condition; increment) {
        //statement(s);
}
```

The **initialization** happens first and exactly once. Each time through the loop, the **condition** is tested; if it’s `true`, the statement block, and the **increment** is executed, then the **condition** is tested again. When the **condition** becomes `false`, the loop ends.

Example Code

```
// Dim an LED using a PWM pin
int PWMpin = 10; // LED in series with 470 ohm resistor on pin 10

void setup()
{
  // no setup needed
}

void loop()
{
   for (int i=0; i <= 255; i++){
      analogWrite(PWMpin, i);
      delay(10);
   }
}
```

Notes and Warnings

The C `for` loop is much more flexible than `for` loops found in some other computer languages, including BASIC. Any or all of the three header elements may be omitted, although the semicolons are required. Also the statements for initialization, condition, and increment can be any valid C statements with unrelated variables, and use any C datatypes including floats. These types of unusual `for` statements may provide solutions to some rare programming problems.

For example, using a multiplication in the increment line will generate a logarithmic progression:

```
for(int x = 2; x < 100; x = x * 1.5){
println(x);
}
```

Generates: 2,3,4,6,9,13,19,28,42,63,94

Another example, fade an LED up and down with one `for` loop:

```
void loop()
{
   int x = 1;
   for (int i = 0; i > -1; i = i + x){
      analogWrite(PWMpin, i);
      if (i == 255) x = -1;             // switch direction at peak
      delay(10);
   }
}
```

goto

Transfers program flow to a labeled point in the program

Syntax

```
label:

goto label; // sends program flow to the label
```

Example Code

```
for(byte r = 0; r < 255; r++){
    for(byte g = 255; g > -1; g--){
        for(byte b = 0; b < 255; b++){
            if (analogRead(0) > 250){ goto bailout;}
            // more statements ...
        }
    }
}

bailout:
```

Notes and Warnings

The use of `goto` is discouraged in C programming, and some authors of C programming books claim that the `goto` statement is never necessary, but used judiciously, it can simplify certain programs. The reason that many programmers frown upon the use of goto is that with the unrestrained use of `goto` statements, it is easy to create a program with undefined program flow, which can never be debugged.

With that said, there are instances where a `goto` statement can come in handy, and simplify coding. One of these situations is to break out of deeply nested for loops, or if logic blocks, on a certain condition.

return

Terminate a function and return a value from a function to the calling function, if desired.

Syntax

```
return;

return value; // both forms are valid
```

Parameters

`value': any variable or constant type

Example Code

A function to compare a sensor input to a threshold

```
 int checkSensor(){
    if (analogRead(0) > 400) {
        return 1;
    }
    else{
        return 0;
    }
}
```

The return keyword is handy to test a section of code without having to "comment out" large sections of possibly buggy code.

```
void loop(){

// brilliant code idea to test here

return;

// the rest of a dysfunctional sketch here
// this code will never be executed
}
```

switch...case

Like if statements, switch case controls the flow of programs by allowing programmers to specify different code that should be executed in various conditions. In particular, a switch statement compares the value of a variable to the values specified in case statements. When a case statement is found whose value matches that of the variable, the code in that case statement is run.

The break keyword exits the switch statement, and is typically used at the end of each case. Without a break statement, the switch statement will continue executing the following expressions ("falling-through") until a break, or the end of the switch statement is reached.

Syntax

```
switch (var) {
  case label1:
    // statements
    break;
  case label2:
    // statements
    break;
  default:
    // statements
}
```
Parameters

`var`: a variable whose value to compare with various cases. **Allowed data types:** int, char
`label1`, `label2`: constants. **Allowed data types:** int, char

Returns

Nothing

Example Code

```
  switch (var) {
    case 1:
      //do something when var equals 1
      break;
    case 2:
      //do something when var equals 2
      break;
    default:
      // if nothing else matches, do the default
      // default is optional
      break;
  }
```

while

A `while` loop will loop continuously, and infinitely, until the expression inside the parenthesis, () becomes false. Something must change the tested variable, or the while loop will never exit. This could be in your code, such as an incremented variable, or an external condition, such as testing a sensor.

Syntax

```
while(condition){
  // statement(s)
}
```

The `condition` is a boolean expression that evaluates to `true` or `false`.

Example Code

```
var = 0;
while(var < 200){
  // do something repetitive 200 times
  var++;
}
```

Further Syntax

#define

`#define` is a useful C component that allows the programmer to give a name to a constant value before the program is compiled. Defined constants in arduino don't take up any program memory space on the chip. The compiler will replace references to these constants with the defined value at compile time.

This can have some unwanted side effects though, if for example, a constant name that had been #defined is included in some other constant or variable name. In that case the text would be replaced by the #defined number (or text).

In general, the const keyword is preferred for defining constants and should be used instead of #define.

Syntax

```
#define constantName value
```

Note that the # is necessary.

Example Code

```
#define ledPin 3
// The compiler will replace any mention of ledPin with the value 3 at
compile time.
```

Notes and Warnings

There is no semicolon after the #define statement. If you include one, the compiler will throw cryptic errors further down the page.

```
#define ledPin 3;    // this is an error
```

Similarly, including an equal sign after the #define statement will also generate a cryptic compiler error further down the page.

```
#define ledPin  = 3  // this is also an error
```

#include

`#include` is used to include outside libraries in your sketch. This gives the programmer access to a large group of standard C libraries (groups of pre-made functions), and also libraries written especially for Arduino.

The main reference page for AVR C libraries (AVR is a reference to the Atmel chips on which the Arduino is based) is here.

Note that `#include`, similar to `#define`, has no semicolon terminator, and the compiler will yield cryptic error messages if you add one.

Example Code

This example includes a library that is used to put data into the program space *flash* instead of *ram*. This saves the ram space for dynamic memory needs and makes large lookup tables more practical.

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

prog_uint16_t myConstants[] PROGMEM = {0, 21140, 702  , 9128,  0, 25764, 
8456,
0,0,0,0,0,0,0,0,29810,8968,29762,29762,4500};
```

/* */ (block comment)

// (single line comment)

; (semicolon)

{} (curly braces)

Arithmetic Operators

% (modulo)

* (multiplication)

+ (addition)

- (subtraction)

/ (division)

= (assignment operator)

Comparison Operators

!= (not equal to)

< (less than)

<= (less than or equal to)
== (equal to)
> (greater than)
>= (greater than or equal to)

Boolean Operators

! (logical not)
&& (logical and)
|| (logical or)

Pointer Access Operators

& (reference operator)
* (dereference operator)

Bitwise Operators

& (bitwise and)
<< (bitshift left)
>> (bitshift right)
^ (bitwise xor)
| (bitwise or)
~ (bitwise not)

Compound Operators

&= (compound bitwise and)
*= (compound multiplication)
++ (increment)
+= (compound addition)
-- (decrement)
-= (compound subtraction)

/= (compound division)
|= (compound bitwise or)