ARDUINO – příručka programátora

Slovo úvodem

Tato publikace je určena začátečníkům, kteří se chtějí seznámit se základním programováním dnes oblíbené platformy Arduino. Není to učebnice programování a není to ani „úplná učebnice Arduina“. Je to text, jehož cílem je dát základní popis, rady a návod, jak to celé uchopit, abyste se hned na začátku neztratili. Proto budeme často věci zjednodušovat a často i zamlčovat nějaké pokročilé detaily, které by začátečníka spíš mátly, než mu pomohly[1]. Doufáme, že tento text čtenářům ukáže, že programování pro Arduino je zvládnutelné a že čtenáře přiměje k dalšímu experimentování a hledání dalších informací.

Mírně v rozporu s úvodním odstavcem si dovolíme uvést pár upřesňujících detailů. Co to vlastně je Arduino? Často se někde píše nebo říká něco jako „Napsal jsem to v Arduinu,“ „Nainstaluješ si do počítače Arduino,“ „Žaluzie mi řídí Arduino,“ ale moc se nerozlišuje, co konkrétně je předmětem diskuse, protože to nakonec z kontextu obvykle vyplývá. Arduino totiž není jen jedna věc, jeden program nebo jeden programovací jazyk, ale celý „ekosystém“, který zahrnuje jak programovací jazyk (kterým vyjadřujete, co má program dělat), tak aplikaci v počítači – vývojové prostředí (kde se píše a ladí program), a také i hardware – skutečný modul s elektronikou (kde program nakonec běží). O všem dohromady a i jednotlivě o každé části se obvykle mluví jako o Arduinu[2].

Často také uvidíte někde napsané „programovací jazyk Arduino“. Není to úplně přesné, ale nakonec to vlastně docela vystihuje to, že je to něco, co je určeno pro psaní programů pro Arduino. Ve skutečnosti je základem programování pro Arduino jazyk C++, přičemž program napsaný člověkem se před překladem do strojového kódu srozumitelného elektronickému modulu ještě upraví a doplní, aby se člověk nemusel starat o všechny detaily. Moduly s elektronikou, prodávané jako Arduino, je možné programovat i v jiných programovacích jazycích jako je třeba Python nebo assembler nebo v grafických blokových jazycích typu Scratch nebo Blockly, ale v této publikaci se ale budeme věnovat výhradně tomu základnímu způsobu, tedy „programovacímu jazyku Arduino“ založenému na C++.

Také je vhodné zmínit, že ty skutečné moduly s elektronikou můžou být velmi rozdílné, různě rychlé i různě výkonné a přesto se k nim můžeme chovat při psaní programů v prostředí Arduina velmi podobně. Základní programy se budou (když se napíšou správně) chovat stejně na malém a relativně pomalém osmibitovém modulu za pade stejně jako na velkém a rychlém modulu za tři litry.

---

[1] *A v poznámkách pod čarou, jako je třeba tahle, budeme uvádět nějaká upřesňující vysvětlení, která by hloubavému čtenáři měla pomoci pochopit některé podivnosti a souvislosti.*

[2] *Takže výše uvedené výkřiky se vztahují k (postupně) hardwaru, vývojovému prostředí a programovacímu jazyku.*

Struktura programu

Základní struktura programu pro Arduino je poměrně jednoduchá, v nejmenší variantě se skládá z příkazů napsaných do pouhých dvou funkcí, pojmenovaných setup a loop.

Funkce `setup()` je přípravná a používá se pro inicializaci toho, co je na začátku běhu programu potřeba. Například k nastavení pinů Arduina na vstup nebo výstup, nastavení parametrů sériové komunikace a podobných jednorázových akcí.
Funkce `loop()` je výkonná a používá se k běžnému chodu aplikace. Tělo této funkce obsahuje programový kód, který se má opakovaně provádět stále dokola, typicky čtení vstupů a podle toho nastavování výstupů. Pokud funkce skončí, systém ji opět hned znovu zavolá. Tato funkce je jádrem všech programů Arduina a vykonává většinu činností.

Důležité!

Vždy je v programu nutné uvést obě tyto funkce, i kdyby měly být prázdné (jinak se program ani nepovede přeložit).

setup()

Funkce setup(), jak již bylo řečeno výše, se volá pouze jednou při spuštění programu. Používá se k jednorázové inicializaci toho, co je potřeba – nastavení režimu jednotlivých pinů, nastavení sériové komunikace apod. Tato funkce musí být v programu obsažena vždy, i když žádné inicializační příkazy neobsahuje.

Příklad:

```
void setup()
{
  pinMode(pin, OUTPUT); // nastav 'pin' jako výstupní
  Serial.begin(115200); // otevři seriovou linku a nastav její rychlost na 115200 Bd
}
```

loop()

Po dokončení funkce setup() se začne neustále dokola volat a provádět funkce loop(), jak její název (loop = smyčka) ostatně napovídá. I tato funkce musí být v programu obsažena vždy, i kdyby byla prázdná a žádné příkazy neobsahovala[3]. Pokud funkce skončí, bude zavolána znovu.

[3] *Není obvyklé, že je funkce loop prázdná, ale stát se to může v situaci, kdy se veškeré dění odehraje ve funkci setup, třeba nějaký jednorázový výpočet, jehož výsledek sdělíme uživateli a tím program skončí.*

Příklad:

```
void loop()
{
  digitalWrite(pin, HIGH);  // nastav 'pin' na log.1
  delay(1000);  // čekej jednu sekundu
  digitalWrite(pin, LOW);  // nastav 'pin' na log.0
  delay(1000);  // čekej jednu sekundu
}
```

Pokud k danému pinu připojíme LED, bude s ní tento program do nekonečna blikat s vteřinovou frekvencí[4].

Základní syntaxe

Syntaxe jazyka pro Arduino je až na naprosté výjimky totožná se syntaxí programovacího jazyka C++. Pro snadné uchopení uživateli, kteří nejsou zrovna odborníci na programování, uvádíme v této kapitole ty nejzákladnější důležité věci.

Jako v jiných jazycích, základní „stavební jednotkou" je příkaz. Může to být například přiřazovací příkaz, který něco počítá (`obvod = prumer * 3.14;`) nebo zavolání funkce (`zalejKytky();`). Každý příkaz je ukončen středníkem `;`. Na formátování textu pomocí mezer, odsazení a řádkování nezáleží, ale velmi doporučujeme text formátovat pro přehlednost tak, že obsah každého bloku bude odsazen a že na řádku bude vždy jen jeden příkaz. Na velikosti písmen (malá/velká) naopak záleží, proměnné `pocet`, `POCET` a `PoCeT` jsou různé.

Příkazy je možné sdružovat do bloků, se kterými se pak zachází tak, jako bychom na tom samém místě místo jednoho příkazu zapsali celý ten blok. To se hodí například při vyhodnocování podmínek – je-li splněna, chceme často vykonat nějakou posloupnost příkazů a ne jen jeden. Bloky příkazů jsou ohraničeny složenými závorkami `{` `}`.

Pokud bychom příkazy za podmínkou složenými závorkami do bloku neuzavřeli, podmínka by ovlivňovala vykonání pouze prvního z nich a ostatní by se vykonaly vždy.

[4] *Tedy skoro vteřinovou, není to naprosto přesné, ale to v tuto chvíli není podstatné. Budík si takhle ale radši neprogramujte.*

Příklad:

```
A=0;B=0;C=0;
if (podminka)
{
  A=5;
  B=5;
}
C=5;
```

Pokud je podmínka splněna, A i B budou nastaveny na 5, jinak budou obě 0. C bude nastaveno na 5 vždy.

Příklad:

```
D=0;E=0;F=0;
if(podminka)
  D=5;
  E=5;
F=5;
```

Pokud je podmínka splněna, D bude nastaveno na 5, pokud ne, D bude 0. V každém případě však E a F budou nastaveny na 5, i když to může na první pohled vypadat jinak.

Celé kusy kódu můžeme pojmenovat a vytvořit tak funkci[5]. Funkce má své jméno, pomocí kterého ji můžeme zavolat. Můžeme jí předat parametry a může vracet nějakou hodnotu.

Program pro Arduino se skládá z jednotlivých funkcí, které se mezi sebou volají podle potřeby tak, jak to programátor napíše – v té minimální variantě aspoň funkce setup a loop.

{ .. } složené závorky

Složené závorky definují začátek a konec bloku kódu. Používají se ve funkcích, v podmínkách i ve smyčkách.

Příklad:

```
void nejakaFunkce()
{
  digitalWrite(pin,HIGH);
  while(digitalRead(jinyPin)==LOW)
  {
    Serial.println("Ještě dole.");
    delay(250);
  }
```

[5] *V jiných jazycích se tomu může říkat například podprogram, subrutina, procedura...*

```
  Serial.println("Už nahoře!");
  digitalWrite(pin,LOW);
}
```

Každá otevírací složená závorka { musí mít svou závorku uzavírací }. Proto se často uvádí, že složené závorky musí být párovány. Samostatně umístěné závorky (otevírací bez uzavírací a naopak) často vedou k záhadným, špatně dohledatelným chybám při překladu.

Programové prostředí Arduino obsahuje praktickou funkci pro kontrolu párování složených závorek. Stačí vybrat závorku, dojet k ní kurzorem nebo kliknout myší bezprostředně před nebo za ni a související závorka bude zvýrazněna.

Také je možné schovat celý blok uzavřený ve složených závorkách kliknutím na čtvereček s minusem na začátku řádky a později ho zase ukázat (minus se po schování změní na plus).

; středník

Středníkem musí být ukončen každý příkaz, ať už to je deklarace proměnné nebo výkonný příkaz ve funkci nebo smyčce.

Příklad:

```
int x = 13;  // deklaruje proměnnou 'x' jako datový typ integer a dává jí hodnotu 13
```

Poznámka:

Pokud zapomenete příkaz středníkem ukončit, dojde k chybě při překladu. Z chybového hlášení může být zřejmé, že se jedná o zapomenutý středník, ovšem také nemusí. Pokud se objeví hlášení o záhadné nebo zdánlivě nelogické chybě, zkontrolujte nejprve, zda v zápisu programu nechybí středník v blízkosti místa, kde kompilátor chybu ohlásil (obvykle těsně před).

// jednořádkové komentáře

Do programu je velmi vhodné vkládat komentáře, vysvětlující jeho činnost[6]. Komentář může být jednořádkový nebo blokový. Jednořádkový komentář začíná dvěma po sobě jdoucími lomítky // a končí automaticky na konci řádku.

Příklad:

```
// toto je jednořádkový komentář
```

Jednořádkové komentáře jsou často používány za příkazy k vysvětlení jejich funkce nebo jako poznámka pro další použití:

[6] *Zkušenost ukazuje, že ani autor programu po nějakém čase sám neví, co a proč jeho program vlastně přesně dělá. Komentáře hodně pomůžou.*

```
hodnota = analogRead(pin);  // přečteme ze vstupu hodnotu
```

Mohou být také použity k dočasnému znefunkčnění kódu programu pro účely ladění:

```
Serial.print("Začínám... ");
// delay(5000);
Serial.println("... končím");
```

Vše od `//` až do konce řádku bude překladač ignorovat, takže v tomto kódu nebude mezi dvěma výpisy prodleva 5 sekund, jak by mohlo na první pohled vypadat.
Vzhledem k tomu, že komentáře jsou programem ignorovány a nezabírají žádný paměťový prostor, mohou a měly by tedy být hojně používány.

/* .. */ blokové komentáře

Blokové komentáře, nebo víceřádkové komentáře jsou oblasti textu, které jsou v programu ignorovány. Jsou používány pro obsažnější komentování kódu nebo poznámky, které pomohou pochopit ostatním význam částí programu. Blokové komentáře začínají dvojicí znaků lomítko a hvězdička `/*` *a končí těmito znaky v opačném pořadí* `*/`.

Příklad:

```
/*
Toto je blokový komentář, nezapomeňte ho ukončit.
Znaky pro jeho začátek a konec musí být vždy v páru!
*/
```

Do blokového komentáře můžeme také zavřít i část kódu, který nechceme nechat vykonávat (například pro ladění). V blokovém komentáři mohou být i jednořádkové komentáře (uvozené `//`), ale není možno do blokového komentáře vložit další blokový komentář.

Blokový komentář nemusí začínat jen na začátku a končit na konci řádku, může to být kdekoli.

Proměnné

Proměnná je způsob pojmenování a uložení číselné hodnoty pro pozdější použití v programu. Jak označení „proměnná“ naznačuje, jejich hodnota může být průběžně měněna, na rozdíl od konstant, jejichž hodnota se nikdy nemění. Proměnná musí být

deklarována a volitelně jí lze už při deklaraci přiřadit hodnotu, která do ní má být uložena.
Proměnná může být pojmenována libovolným jménem, které nepatří mezi klíčová slova v jazyce Arduino.

Příklad:

```
int vstup;  // deklaruje proměnnou jako celé číslo
vstup = analogRead(2);  // přiřadí proměnné hodnotu podle analogové hodnoty pinu 2
```

V tomto příkladu je na prvním řádku deklarována proměnná jménem '*vstup*'. První řádek kódu deklaruje, že proměnná '*vstup*' bude obsahovat hodnotu s datovým typem int (zkratka pro název integer, celé číslo). Druhý řádek nastaví hodnotu proměnné podle hodnoty aktuálního napětí na pinu 2.

Jakmile byla proměnné přiřazena nebo změněna hodnota, můžete od té chvíle dále použít její hodnotu přímo nebo testovat, zda tato hodnota splňuje určité podmínky, aniž by záleženo na tom odkud se tato hodnota vzala, zdroj hodnoty se také může změnit, ale proměnné se to už dál netýká[7].

Před prvním použitím hodnoty proměnné musí vždy být proměnné nějaká hodnota určena. Často je to nějakým výpočtem, ale je také možné proměnné přiřadit iniciální hodnotu už při deklaraci:

```
// deklaruje proměnnou jako celé číslo a dává jí počáteční hodnotu 50
int pocet = 50;
```

Následující příklad testuje, zda hodnota proměnné *vstup* je menší než 100. Je-li to pravda, pak hodnotu proměnné *vstup* přiřazením změní na 100. V každém případě nakonec na základě hodnoty proměnné *vstup* způsobí program prodlevu, která je nyní díky podmínce vždy minimálně 100 milisekund:

```
if (vstup < 100)  // testuje, zda je hodnota proměnné méně než 100
{
  vstup = 100;  // je-li to pravda, je jí přiřazena hodnota 100
}
delay(vstup);  // použije hodnotu proměnné jako parametr funkce delay
```

Názvy proměnných

Proměnným a námi psaným funkcím bychom měli pro lepší orientaci v programovém kódu dávat popisné názvy. Jména proměnných, jako je ‚hustota' nebo ‚barva' pomáhají programátorovi i komukoli jinému při čtení kódu snadněji poznat, co proměnná znamená, na rozdíl od třeba ‚a' a ‚b', která mohou znamenat cokoli a na

[7] *I kdyby se například napětí na pinu 2 v příkladu po přečtení změnilo, v proměnné bude pořád ta hodnota z chvíle, kdy se ze vstupu pomocí funkce analogRead četla.*

první pohled to není vidět[8]*. Volba názvu je na programátorovi, avšak je potřeba dodržovat několik jednoduchých pravidel.*

Název proměnné ani funkce nesmí obsahovat mezeru, takže pokud chceme proměnnou nazvat víceslovně, musíme slova spojit. Ve světě Arduina je zaběhlý tzv. „camelCase“ systém, kdy jednotlivá slova jsou psána malým písmenem, ale jejich první písmeno je velké (až na první slovo)[9], například analogRead nebo digitalWrite, která jsme již použili výše, anebo třeba prumernaObsazenostAutobusu, pokud bychom si tak chtěli nějakou proměnnou nazvat. V názvu proměnné smí být všechny znaky anglické abecedy (malá i velká písmena a-z a A-Z), číslice (0-9) a podtržítko (_) a nesmí začínat číslicí. Na velikosti písmen záleží.

Deklarace proměnných

Všechny proměnné musí být deklarovány ještě před jejich prvním použitím. Deklarace proměnné znamená, že definujete její typ (int, long, float, atd.), přiřadíte jí jméno a případně i počáteční hodnotu. Deklarovat proměnnou budete v programu jen jednou; hodnotu proměnné můžete v programu měnit podle potřeby.

Následující příklad deklaruje, že proměnná ‚*vstup*‘ je typu int neboli integer, a že její počáteční hodnota je rovna nule.

Příklad:

```
int vstup = 0;
```

Rozsah platnosti proměnných

Proměnná může být deklarována mimo jakoukoli funkci, uvnitř funkce nebo uvnitř nějakého bloku. Místo, kde je proměnná deklarována, určuje její použitelnost pro jednotlivé části programu.

Globální proměnná je ta, která je deklarována vně jakékoli funkce. K takové proměnné mají přístup a mohou ji používat všechny příkazy ve všech funkcích v programu, deklarace proměnné však musí být uvedena před („nad“) jejím použitím.
Lokální proměnná je ta, která je deklarována uvnitř funkce nebo bloku uzavřeného ve složených závorkách. Je dostupná a může být použita pouze příkazy uvnitř funkce (nebo

[8] *V našich příkladech jsou ale naopak obvykle použita krátká nicneříkající jména proměnných (například vstup nebo pocet), protože při vysvětlování programovacího jazyka na jejím významu nezáleží.*

[9] *„camelCase“ je podle anglického názvu camel pro velblouda, jehož hrby to má připomínat a grafické znázornění s písmenem c prvním malým a druhým velkým pak říká, že název má začínat malým písmenem a až další slova velkým.*

bloku), kde byla deklarována[10]. Standardně je lokální proměnná vytvořena při volání funkce vždy znovu a se skončením funkce zaniká. To, že přístup ke svým lokálním proměnným má pouze daná funkce, zjednodušuje program a snižuje možnost programátorských chyb, zároveň to, že lokální proměnná vzniká a zaniká s voláním funkce také zmenšuje objem potřebné paměti, protože tato proměnná tak potřebuje v paměti místo jen pokud zrovna běží daná funkce. Navíc se lokální proměnné v různých blocích nebo funkcích mohou jmenovat stejně, aniž by se navzájem ovlivňovaly, což také přispívá k čistotě programů a snížení nebezpečí chyb.

Pokud se více různých proměnných jmenuje stejně a funkce, ve kterých jsou tyto proměnné deklarovány, se navzájem volají, pracuje se vždy s proměnnou definovanou „nejvíc uvnitř“ a proměnné stejného jména v nadřazených (volajících) funkcích se tím neovlivní.

Následující příklad ukazuje, jakým způsobem můžeme deklarovat různé typy proměnných a předvádí viditelnost každé z proměnných v programu:

```
int pin;  // proměnná 'pin' je globální, tj. viditelná pro všechny funkce

void setup()
{
  pin = 5;  // globální proměnné přiřadíme hodnotu
}

void loop()
{
  // proměnná 'i' je viditelná jen uvnitř smyčky 'for'
  for (int i=0; i<20; i++)
  {
    // proměnná 'hodnota' je viditelná jen uvnitř smyčky
    // pin je globální proměnná a proto je vidět i tady,
    // i když tu nebyla deklarována
    int hodnota = analogRead(pin);
    Serial.print(i);
    Serial.println(hodnota);
  }
  // proměnná 'f' je viditelná jen uvnitř funkce 'loop', a to jen odsud dále
  float f;
  f=2.5;
  Serial.println(f);
}
```

[10] *Výjimkou je proměnná, kterou deklarujeme jako řídící proměnnou cyklu 'for' tak jako v následujícím příkladu proměnná 'i'. Je platná v tomto cyklu, ale de fakto jsme ji deklarovali před tím blokem.*

Konstanty

Konstanty slouží pro pojmenování hodnot, které se v průběhu vykonávání programu nijak nemění. Slouží k tvorbě přehlednějších programů. Arduino má několik předdefinovaných konstant, uživatel si může vytvořit další.

TRUE / FALSE

TRUE a FALSE jsou konstanty, které definují základní logické hodnoty splněno / nesplněno nebo pravda / nepravda. Obvykle jsou definovány jako TRUE=1 a FALSE=0. Při vyhodnocování podmínek se jako nesplněná podmínka vyhodnotí konstanta FALSE a výraz, jehož číselná hodnota je 0. Konstanta TRUE a libovolná číselná hodnota různá od 0 se vyhodnotí jako splněná podmínka. Běžné matematické operace porovnávání se chovají podle očekávání.

Příklad:

```
if (a == FALSE)
{
  příkazy;  // příkazy se provedou, pokud a je FALSE nebo 0
}
else
{
  // příkazy se provedou, pokud a je TRUE nebo pokud a je číslo různé od 0
  příkazy;
}

if (b >= 42)
{
  příkazy;  // příkazy se provedou, je-li b větší nebo rovno číslu 42.
}
```

HIGH / LOW

Tyto konstanty definují napěťovou úroveň pinů Arduina jako vysokou nebo nízkou a jsou používány při čtení nebo zápisu na digitální piny.
HIGH je definována jako logická úroveň 1, ON, zapnuto, resp. tolik voltů, jaké je napájení Arduina[11], zatímco LOW je logická úroveň 0, OFF, vypnuto, nebo 0 voltů.

[11] *Obvykle 5 V, ale dnes často 3,3 V – záleží na konkrétním hardwaru.*

Příklad:

```
digitalWrite(13, HIGH);  // nastaví pin číslo 13 na logickou úroveň 1 (ON)
```

INPUT / INPUT_PULLUP / OUTPUT

Tyto konstanty se používají ve funkci pinMode() pro přepnutí funkce digitálního pinu na vstup (INPUT) nebo výstup (OUTPUT). Konstanta INPUT_PULLUP přepne digitální pin na vstup a navíc zapojí interní rezistor mezi napájecí napětí a pin (tzv. „zdvíhací" nebo anglicky „pull-up" rezistor).

Příklad:

```
pinMode(11, INPUT);  // nastaví pin číslo 11 jako vstupní
// nastaví pin číslo 12 jako vstupní a zapojí zdvihací rezistor
pinMode(12, INPUT_PULLUP);
pinMode(13, OUTPUT);  // nastaví pin číslo 13 jako výstupní
```

Použití konstanty `INPUT_PULLUP` zajistí stabilní hodnotu i v nezapojeném stavu (a to vysokou), je tedy obzvláště vhodná pro použití, když je na vstup zapojeno tlačítko nebo spínač. Stačí pak, aby spínač byl svým druhým kontaktem připojen k zemi (0 V) – pokud je sepnutý, bude na pinu 0 (nízká úroveň), pokud rozepnutý, bude tam díky zdvíhacímu rezistoru 1 (vysoká úroveň)[12].

Uživatelské konstanty

Uživatelské konstanty slouží ke zpřehlednění kódu, kdy si programátor nadefinuje pojmenování pro nějakou číselnou hodnotu, aby v kódu nemusel pořád psát toto číslo.

Uživatelské konstanty se dají definovat dvěma způsoby:

A. připojením klíčového slova `const` při deklaraci proměnné spolu s přiřazením její hodnoty:

```
int const DELKA = 13;  // konstanta DELKA bude mít hodnotu 13
const int PIN_LED = 7; // konstanta PIN_LED bude mít hodnotu 7
```

(je lhostejné, jestli const napíšete před nebo za typ, v obou případech bude hodnota proměnné zafixována a nepůjde později změnit)

B. pomocí konstrukce `#define jmeno hodnota` (a to bez rovnítka a středníku[13]):

```
#define PIN_TLACITKO 6 // konstanta PIN_TLACITKO bude mít hodnotu 6
```

[12] *Pokud by byl pin nastaven pouze jako INPUT, tak by v rozepnutém stavu nebylo definováno, jaká je hodnota pinu a za napřiznivých okolností by mohla být někdy vyhodnocena jako nízká (0) a někdy jako vysoká (1).*

[13] *Tak to diktuje definice jazyka C++ a způsobu překladu zdrojového programu do spustitelného kódu. Zejména si všimněte, že tam není středník. Pokud by tam byl uveden,můžete si tím způsobit velmi těžce nalezitelnou chybu v programu.*

Poznámka:

V řadě programů pro Arduino se používá druhý způsob zápisu, ale velmi doporučujeme používat způsob první. Říkáme v něm kromě jména a hodnoty i typ této konstanty, čímž usnadňujeme práci překladači, který tak může při použití konstanty informaci o typu využít (například zkontrolovat, že na daném místě se smí tento typ použít), navíc se na takovou konstantu vztahují pravidla rozsahu platnosti uvedená výše u proměnných. Při použití druhého způsobu zápisu překladač tuto informaci nemá a může tak konstantu za určitých okolností použít jinak, než uživatel zamýšlel.

Jména konstant mají povolené stejné znaky jako jména proměnných. Na rozdíl od nich se ve světě Arduina často konstanty píší verzálkami (velkými písmeny) a slova se oddělují podtržítkem. Není to ale povinné.

Výhodou použití konstant proti konkrétním číslům je situace, kdy shodou okolností dvě naprosto rozdílné věci mají zrovna stejnou hodnotu, ale logicky se jedná o něco zcela jiného. Třeba velikost bot a oblíbený počet kostek cukru v kafi. Pokud chceme někdy později jednu z nich změnit, tak v případě pojmenované konstanty změníme její hodnotu na jednom místě (tam, kde je konstanta nadefinovaná), protože v celém zbytku programu už používáme její jméno a ne konkrétní číslo. Pokud ale nemáme konstantu pojmenovanou a chceme ji změnit (protože nám třeba vyrostla noha), musíme projít celý program a všude, kde vidíme číslo 10, se musíme rozhodnout, jestli to máme změnit na 11, nebo jestli těch 10 musí zůstat 10, aby nám kafe chutnalo. Bohužel, bez pojmenovaných konstant programátor často nezmění všechno co měl a změní i co neměl, takže pak má zásobu malých bot a nechutně sladkého kafe…

Datové typy

byte

Datový typ byte ukládá hodnoty jako osmibitové číselné hodnoty bez desetinných míst. Ukládá hodnotu v rozsahu 0 až 255.

Příklad:

```
// deklaruje proměnnou 'nejakeCislo' jako datový typ byte
byte nejakeCislo = 180;
```

Poznámka:

Celočíselná proměnná přeteče (podteče), pokud dojde k překročení její maximální nebo minimální hodnoty. V obou případech se hodnota „zatočí“ na druhý konec, například, je-li

byte x = 255 a následná operace zvětší hodnotu o 1, (x = x + 1 nebo x ++), hodnota x bude po přetečení rovna 0.

int

Celočíselný datový typ integer je nejběžnějším způsobem ukládání celých čísel. Ukládá hodnotu jako šestnáctibitovou v rozsahu -32 768 až 32 767.

Příklad:

```
// deklaruje proměnnou 'nejakeCislo' jako datový typ integer
int nejakeCislo = 1500;
```

Poznámka:

Celočíselná proměnná přeteče (podteče), pokud dojde k překročení její maximální nebo minimální hodnoty. Například, je-li x = 32 767 a následná operace zvětší hodnotu o 1, (x = x + 1 nebo x ++), hodnota x je po přetečení rovna -32 768. Obdobně naopak, snížení hodnoty -32 768 o 1 bude mít za výsledek +32 767.

long

Datový typ long (integer) je určen pro velká čísla bez desetinných míst. Ukládá hodnotu jako třicetidvoubitovou v rozsahu -2 147 483 648 až 2 147 483 647.

Příklad:

```
// deklaruje proměnnou 'nejakeCislo' jako datový typ long
long nejakeCislo = 90000;
```

float

Datový typ float je určen pro operace s čísly s plovoucí desetinnou čárkou. Čísla s plovoucí desetinnou čárkou mají větší rozlišení než celá čísla a jsou uložena jako třicetidvoubitové hodnoty s rozsahem -3,4028235 x 10^{38} až 3,4028235 x 10^{38}.

Příklad:

```
// deklaruje proměnnou 'nejakeCislo' jako datový typ float
float nejakeCislo = 3.14;
```

Pokud používáme číselné konstanty zachycující necelá čísla, pak je nutné pro Arduino používat desetinnou tečku, tj. například `x = 5.005;` jak se používá ve světě, nikoli `x = 5,005;` jak píšeme podle platných českých norem.

Poznámka:

Čísla s plovoucí desetinnou čárkou mají velmi omezenou přesnost, kvůli níž může dojít k problémům při porovnávání dvou takových čísel. Matematické operace s plovoucí desetinnou čárkou jsou také mnohem pomalejší než celočíselná aritmetika a na Arduinu je, pokud to jen trochu jde, velmi vhodné se jim vyhýbat. Často lze použít jednoduchý „trik": místo zaznamenání čísla s desetinnou částí pomocí proměnné typu float použijeme celočíselný typ, do kterého uložíme číslo s desetinnou čárkou posunutou o vhodný počet míst.

Například místo výše uvedeného příkladu bychom někdy mohli použít záznam čísla 100x většího, například:

```
// pro jistotu si v názvu proměnné poznamenám vynásobení 100x
int pi100 = 314;
```

a tak s ním dále pracovat, ovšem samozřejmě stále musíme vést v patrnosti tento posun, například:

```
int obvodKruznice = 2 * pi100 * prumerKruznice / 100;
```

double

Datový typ double je stejně jako float určen pro operace s čísly s plovoucí desetinnou čárkou. Na některých Arduinech (například Uno, Mini, Micro) je totožný s typem float, na některých jiných (například Due) je uložen ve dvojnásobně velkém prostoru, tj. v 64 bitech.

Příklad:

```
// deklaruje proměnnou 'jineCislo' jako datový typ double
double jineCislo = 3.14;
```

Poznámka:

Stejně jako typu float, i typu double je na Arduino lepší se vyhnout. Pokud ho nutně potřebujete, dobře si zkontrolujte, jak je tento typ pro vaše Arduino implementován.

char

Typ char se používá pro uložení jednoho písmene.

I když se do proměnných tohoto typu ukládají písmena, ve skutečnosti je v paměti uložen kód příslušného písmene a program s ním pracuje jako s číslem.

Příklad:

```
char pismeno = 'A';  // proměnnou inicializujeme jako písmeno A
```

Písmeno bude v paměti představováno číslem 65 a můžeme s ním dělat vše, co s jinými čísly. Platí to i naopak: všude, kde můžeme uvést nějaké číslo, můžeme místo toho uvést písmeno s tímto kódem. Pouze musíme překladači sdělit, že to je písmeno, a to provedeme uzavřením do apostrofů (').
Tento způsob již byl použit v deklaraci výše, je možné ale dělat postatně zajímavější věci:

```
if (pismeno >='a' && pismeno <= 'z')
    Serial.println("Male pismeno.");
```

Nebo:

```
pismeno = 'B'+3;  // bude to E
```

Poznámka:

Písmena jsou kódována pomocí systému ASCII, který ale neobsahuje všechny používané znaky, zejména ne všechny znaky s diakritikou. Dá se říci, že obsahuje jen znaky, které najdete na anglické nebo americké klávesnici. Celou tabulku naleznete například v dokumentaci Arduina (a na mnoha dalších místech).
Jednotlivá písmena se uzavírají do apostrofů ('), nikoli do uvozovek ("). Uzavřením do uvozovek by vzniknul řetězec znaků, i kdyby to byl znak jediný ('a' není "a").

pole

Polem se nazývá množina prvků, ke kterým je možno přistupovat prostřednictvím indexace. Libovolná hodnota v poli může být zpřístupněna uvedením názvu pole a indexu hodnoty.

Pole jsou indexována od nuly, přičemž první hodnota v poli má index 0.
Pokud deklarujeme pole bez počátečního obsahu, musíme uvést jeho velikost a hodnoty na pozice dané indexem mu přiřadit později:

Příklad:

```
// deklaruje pole jménem 'mojePole' se šesti prvky typu integer
int mojePole[5];
```

Prvkům pole je ale možné přiřadit hodnoty už při deklaraci obdobně jako obyčejným proměnným.
Pokud neuvedeme velikost pole, bude mít tolik prvků, kolik jich při inicializaci uvedeme.

Příklad:

Tato deklarace vytvoří pole o šesti prvcích:

```
int mojePole[] = {9, 50, 42, 161, -75, 200};
```

Tato deklarace vytvoří pole o stu prvcích, z nichž prvních šest je inicializováno zde uvedenými hodnotami a zbylých 94 jsou nuly:

```
int velkePole[100] = {9, 50, 42, 161, -75, 200};
```

Chcete-li použít hodnotu prvku z pole, například přiřadit prvku pole hodnotu nebo naopak hodnotu prvku přiřadit do obyčejné proměnné, pak použijete název pole a index pozice, uzavřený do hranatých závorek[14]:

```
mojePole[3] = 10;  // přiřadí čtvrtému prvku pole hodnotu 10
x = mojePole[3];  // 'x' se nyní rovná 10
```

Pole jsou často používána v cyklech for, kde je hodnota čítače zároveň použita jako index pozice pro každý prvek pole.

V následujícím příkladu definujeme pole pro nastavování různé úrovně svícení LED. Použijeme smyčku, jejíž čítač začíná na 0. Hodnota na pozici 0 v poli 'jas' (v tomto případě 180) se předá funkci analogWrite, která ji použije k nastavení analogové úrovně[15] na pinu daném konstantou PIN_LED (zde 10). Poté počká 200 ms a pak se smyčka znovu opakuje, vybere se další prvek pole podle hodnoty proměnné čítače atd.

```
int const PIN_LED = 10;  // určení, ke kterému pinu je připojená LED

byte jas[] = {180, 30, 255, 200, 10, 90, 150, 60};
// vytvořeno pole s osmi prvky různých hodnot

void setup()
{
  // není potřeba nic nastavovat
}

void loop()
{
  // necháme smyčku běžet pro všechna čísla od 0 do 6, tj. všechny indexy v poli
  for(int i=0; i<7; i++)
  {
    analogWrite(PIN_LED, jas[i]);  // zapiš na výstup hodnotu z pole
    delay(200);  // pauza 200 ms
  }
}
```

[14] *Pozor, hodnota indexu se nijak nekontroluje, musíte sami vědět, že index je uvnitř pole a ne až za koncem! Obvyklá chyba je, že pole s 5 prvky indexujete čísly 1 až 5 (tj. vynecháte index 0 a indexem 5 sáhnete až „za" pole; správně je 0 až 4).*

[15] *Přesněji řečeno PWM modulace s daným plněním, viz popis funkce analogWrite v dokumentaci Arduino.*

Celá čísla se znaménkem / bez znaménka

Typy, které jsme vypsali dosud, umožňují zachytit jak záporná, tak kladná čísla (například int -32 768 až +37 767). Často se zápornými čísly pracovat nepotřebujeme, a k tomu se hodí právě čísla bez znaménka.

U včech celočíselných typů můžeme uvést ještě klíčové slovo unsigned. To pak změní typ na právě typ bez znaménka. Na rozdíl od stejného typu se znaménkem může obsahovat jen čísla od nuly výše, například unisgned int má rozsah 0 až 65 535.

Různě dlouhá celá čísla

Výše popsané typy představují základní sadu používaných typů v Arduinu. Jejich velikost (rozsah možných hodnot) jsme u těchto typů uvedli, v obecnosti však na různých platformách nemusí být přesně tato – například datový typ int může být ne 32, ale 64 bitů dlouhý apod. To je dáno vlastnostmi a možnostmi jazyků C a C++. Překladač, použitý v Arduinu, ale podporuje i řadu dalších typů, které se velmi hodí při potřebě typu s nějakou určenou velikostí tak, aby na všech podporovaných platformách byla tato velikost stejná.

Tyto další typy, které podporuje překladač použitý také v Arduinu, je možné rozdělit do čtyř skupin:

a) typy s přesně danou velikostí:

```
int8_t, int16_t, int32_t, int64_t
```

celé číslo, které v paměti zabírá 8, 16, 32 nebo 64 bitů a může nabývat záporných i kladných hodnot.

```
uint8_t, uint16_t, uint32_t, uint64_t
```

celé číslo bez znaménka o zadané velikosti, může nabývat pouze kladných hodnot.

b) typy, pro které je definována minimalní velikost (ale mohou být i větší):

```
int_least8_t, int_least16_t, int_least32_t, int_least64_t
uint_least8_t, uint_least16_t, uint_least32_t, uint_least64_t
```

c) typy, pro které je definována minimalní velikost (ale mohou být i větší) a zároveň je jejich zpracování co nejrychlejší:

```
int_fast8_t, int_fast16_t, int_fast32_t, int_fast64_t
uint_fast8_t, uint_fast16_t, uint_fast32_t, uint_fast64_t
```

Typy v této skupině nejsou příliš často používány, uvádíme je pro úplnost. Na některých platformách totiž může být rychlejší pracovat například s celým 64bitovým prvkem místo jediného bytu, takže int_fast8_t by tam mohl měl 64 bitů. To ale na běžných Arduinech nenajdete.

d) Typy s co největším rozsahem, ať už je jaký chce:

```
intmax_t, uintmax_t
```

Ani tyto typy nejsou běžně používány a hodí se jen při opravdu velmi speciálních případech.

Poznámka:

S klidem zapomeňte na typy ve skupinách b), c) a d), ale naučte se rozumět typům ze skupiny a) a používat je. Velmi často je naleznete v knihovnách a cizích programech, kde se velmi dobře hodí jak z hlediska správnosti program, tak i čitelnosti, protože je na první pohled jasné, jaké hodnoty takový typ může obsahovat. Kromě toho může použití správných typů velmi pomoci (zejména na Arduinech, která mají málo paměti, nebo kde Váš program bude mít velké požadavky) – a naopak, použití nesprávných typů může vést kromě pomalého běhu až i k nemožnosti program plně vytvořit.

Jeden příklad za všechny: při procházení prvků pole s pěti prvky pomocí smyčky for potřebujeme řídící proměnnou cyklu, ve které bude index 0 do 4. Není tedy potřeba typ int (-32 768 až +32 767), stačí uint8_t (0 až 255). Výsledný program bude o něco menší i rychlejší, protože není potřeba pracovat s dvěma byty, ale jen s jedním, což je základní jednotka na malých Arduinech. Jakýkoli větší typ (dva a více bytů) se musí rozvinout do práce s jednotlivými byty.

Aritmetické operace

Základní aritmetické operátory jsou: sčítání, odčítání, násobení a dělení. Vrací součet, rozdíl, součin nebo podíl (v uvedeném pořadí) dvou operandů.

Příklad:

```
y = y + 3;
x = x - 7;
i = j * 6;
r = r / 5;
```

Pokud jsou operandy celočíselné, bude i výsledek celočíselný. Například 9/4 poskytne výsledek 2 místo 2,25, protože operandy 9 a 4 jsou celá čísla, a tedy i dělení se bude provádět v celých číslech. Obdobně podíl 1/2 vyjde při celočíselném dělení 0.

Je-li některý z operandů typu float, bude vypočet probíhat v typu float, i kdyby všechny ostatní byly celočíselné. Takže při dělení 1,0/2 se zjistí, že výraz obsahuje aspoň jedno číslo typu float (v zápisu 1.0 je desetinná tečka, takže to je float) a tedy výsledek vyjde 0,5, dělení 9/4,0 vyjde 0,25 atd.

Pro své proměnné zvolte vždy takový datový typ, který dokáže uložit dostatečně velká čísla jako výsledky vašich výpočtů. Mějte na paměti, že proměnná může přetéct, pokud se do jejího prostoru výsledná hodnota nevejde. Například pokud k proměnné typu byte (která má rozsah 0 až 255) s aktuální hodnotou 255 přičteme 1, stane se z ní 0, protože 256 se do jednobytové proměnné nevejde. A naopak, pokud od této nuly zase jedničku odečteme, dostaneme 255, protože menší číslo než 0 se do jednoho bytu uložit také nedá. U typů, které mají i záporné hodnoty, to platí stejně: proměnná typu int přeteče z 32767 po přičtení jedničky na nejmenší číslo, tedy -32768 a když od -32768 jedničku odečteme, dostaneme +32767. Pro matematiku, která vyžaduje práci s reálnými čísly nebo zlomky, zvolte proměnné typu float, ale buďte si vědomi jejich nedostatků: zabírají více místa v paměti, výpočet je pomalejší a výsledek nemusí být přesně.

Poznámka:

Pomocí operátoru přetypování, kdy před výraz uvedeme do kulatých závorek jméno typu, lze převést jeden typ na jiný, aniž by bylo nutné ukládat přetypovanou hodnotu do proměnné. Například pokud je mujFloat typu float, tak výraz `(int) mujFloat` *bude celkově typu int. Nebo přiřazení* `i = (int) 3.6;` *nastaví i rovno 3.*

Složené přiřazení

Složené přiřazení kombinuje aritmetické operace s přiřazením hodnoty proměnné.

Mezi nejčastější složená přiřazení patří:

```
x++;  // je totéž jako x = x + 1; tedy zvětšení hodnoty x o 1
x--;  // je totéž jako x = x - 1; tedy zmenšení hodnoty x o 1
x += y;  // je totéž jako x = x + y; tedy zvětšení hodnoty x o hodnotu y
x -= y;  // je totéž jako x = x - y; tedy zmenšení hodnoty x o hodnotu y
x *= y;  // je totéž jako x = x * y; tedy vynásobení x hodnotou y
x /= y;  // je totéž jako x = x / y; tedy vydělení x hodnotou y
```

Příklad:

```
// ztrojnásobí původní hodnotu x a znovu jí přiřadí výsledné hodnotě x
x *= 3;
// vydělí proměnnou y hodnotou proměnné z a výsledek uloží zpátky do y
y /= z;
```

Relační operátory

Porovnání se často používá v testech uvnitř příkazu if, kdy se vyhodnocuje, zda uvedená podmínka platí.
K dispozici jsou následující základní typy podmínek:

```
x == y  // x je rovno y
x != y  // x není rovno y
```

```
x < y  // x je menší než y
x > y  // x je větší než y
x <= y  // x je menší nebo rovno y
x >= y  // x je větší nebo rovno y
```

x a y mohou být jak proměnná, tak konstanta, konkrétní číslo nebo hodnota nějaké zavolané funkce.

Logické operátory

Logické operátory jsou nástroj ke spojování více výrazů do jedné podmínky (často například v testu ‚if'). K dispozici jsou tři logické operátory, OR, AND a NOT:

Logické OR:

```
// vrací hodnotu TRUE, pokud je splněna aspoň jedna podmínka
if (x > 0 || y > 0)
```

Logické AND:

```
// vrací hodnotu TRUE pouze pokud jsou splněny obě podmínky
if (x > 0 && x < 5)
```

Logické NOT:

```
// vrací opak výsledku podmínky, tj. hodnotu TRUE, pokud základní
// podmínka splněna není
if (!(x > 0))
```

Pozor na priority těchto operátorů. Například výraz `if (A || B && C)` se bude vyhodnocovat, jako kdybychom napsali `if (A || (B && C))`, protože logické AND má vyšší prioritu, než OR. Zejména operátor NOT (vykřičník) může způsobit potíže, protože má velmi vysokou prioritu – větší, než obvyklá porovnání na rovnost nebo nerovnost[16].

Aby nedošlo k nechtěnému vyhodnocení, je potřeba výraz správně uzávorkovat.

Jednoduchá pomůcka je: priority OR, AND a NOT jsou ve stejném pořadí, jako se sčítáním, násobením a znaménkem pro záporné číslo.

Ještě jednodušší pravidlo ale zní: když nevím, závorkuji, protože pár závorek navíc na správném místě nevadí.

[16] *I proto jsou v ukázce výše kolem porovnání x > 0 závorky; kdyby tam nebyly, tak by se nejprve vyhodnotilo !x a výsledek negace by se porovnal s nulou.*

Řízení toku programu

if

Příkaz if testuje, zda byla splněna určitá podmínka. Pokud ano (TRUE), vykoná všechny příkazy v bloku za if. Pokud podmínka splněna není (FALSE), program příkazy v bloku za if přeskočí.

Formát příkazu if je:

```
if (podmínka)
{
  příkazy;
}
```

přičemž podmínka může být jak jednoduché porovnání, tak i komplikovanější podmínka složená z více výrazů pomocí logických operátorů.

Poznámka:

Dejte si pozor na náhodné použití přiřazení '=' místo porovnání '==='. V příkazu if (x = 10) dojde místo porovnání hodnoty x s číslem 10 k přiřazení čísla do proměnné. Navíc hodnotou každého přiřazení je číslo, které se přiřadilo, tj. zde 10, což se zde vyhodnotí jako pravda (TRUE), protože jako TRUE se chápe cokoli, co není nula. Místo '=' je v porovnání nutno použít dvě rovnítka za sebou, '==', tedy if(x == 10).

if .. else

Operace 'if .. else' umožňuje rozhodování stylem 'buď .. a nebo' a větvení programu podle výsledku operace.

Například, pokud chcete otestovat stav digitálního vstup a podle toho, zda je vstupní pin ve stavu HIGH provést jednu činnost a když je hodnota LOW (neboli není HIGH) tak naopak provést něco jiného, můžete to zapsat tímto způsobem:

```
if (inputPin == HIGH)
{
  udelejA;
}
else
{
  udelejB;
}
```

Takové testy můžeme spojovat do celého řetězce za sebe, pokud testujeme více vzájemně se vylučujících podmínek. Pamatujte si však, že vždy se spustí v závislosti na podmínkách testů jen jedna část kódu:

```
if (inputPin < 500)
{
  udelejA;
}
else if (inputPin >= 1000)
{
  udelejB;
}
else
{
  udelejC;
}
```

for

Příkaz for se používá k opakování bloku příkazů, uzavřených do složených závorek.

Záhlaví smyčky se skládá ze tří částí, oddělených středníkem(;):

```
for (inicializace; podmínka; postup)
{
  příkazy;
}
```

Nejprve se provede inicializace (obvykle deklarace a inicializace řídící proměnné cyklu). Za inicializací následuje podmínka, která se testuje při každém průchodu smyčkou. Pokud je podmínka splněna, jsou provedeny příkazy v bloku za příkazem for („vlastní tělo cyklu“) a následně se vykoná část označená jako postup (obvykle změna řídící proměnné cyklu, abychom postoupili na další prvek). Pak se opět testuje podmínka a pokud je splněna, vykonají se opět příkazy v bloku za if a část pojmenovaná postup atd. až do chvíle, kdy se při testování podmínky zjistí, že již splněna není. Tehdy se smyčka ukončí a pokračuje se příkazy zapsanými za celou smyčkou.

Následující příklad proběhne smyčky for funguje takto: na začátku je vytvořena celočíselná proměnná 'i' a je inicializována číslem 0. V každém kroku se otestuje, zda je její hodnota menší než 20 a je-li to pravda, provedou se příkazy, uzavřené ve složených závorkách a pak se zvětší hodnota proměnné 'i' o 1:

```
// deklaruje proměnnou 'i', testuje zda je menší než 20 a
// pokud ano, vykoná následující blok příkazů
// a pak zvětší hodnotu i o 1
for (int i=0; i<20; i++)
{
  digitalWrite(13, HIGH);  // nastaví pin 13 na HIGH
```

```
  delay(250);  // čeká 250 ms
  digitalWrite(13, LOW);  // nastaví pin 13 na LOW
  delay(250);  // čeká 250 ms
}
```

Poznámka:

Programovací jazyk C má smyčky mnohem flexibilnější, než jsou podobné smyčky v jiných programovacích jazycích, včetně BASICu. Některé z prvků záhlaví nebo i všechny tři mohou být vynechány, středníky ale zůstávají povinné a pro inicializaci, podmínku i výraz může být použit jakýkoli platný příkaz jazyka C. Správně definované záhlaví příkazu for může nabídnout elegantní řešení některých komplikovaných nebo neobvyklých programových problémů.

while

Smyčka while se bude cyklicky opakovat, dokud podmínka uvnitř závorek bude pravdivá (TRUE). Jakmile podmínka splněna nebude (FALSE), smyčka skončí. Pokud podmínka není splněna hned při prvním testování, tělo smyčky se vůbec neprovede a smyčka skončí, naopak pokud bude podmínka splněna vždy, program smyčku while neopustí a bude ji vykonávat pořád dokola.

Příklad:

```
while (podmínka)
{
  příkazy;
}
```

Následující příklad testuje, zda je hodnota proměnné 'nejakeCislo' menší než 200 a je-li to pravda, provede uvedené příkazy a zvětší hodnotu proměnné o 1. Smyčka se bude opakovat tak dlouho, dokud je hodnota proměnné 'nejakeCislo' menší než 200, neboli až do doby, kdy proměnná 'nejakeCislo' nabyde hodnoty 200 nebo větší.

```
// smyčka poběží, dokud bude proměná menší než 200
while (nejakeCislo < 200)
{
  příkazy; // vykonání příkazů
  nejakeCislo++;  // zvětšení proměnné o 1
}
```

do .. while

Tato smyčka pracuje podobně jako smyčka while, jen s tím rozdílem, že podmínka je testována nikoli na začátku, ale až na konci smyčky, takže smyčka proběhne vždy nejméně jednou.

Příklad:

```
do
}
  příkazy;
}
while (podmínka);
```

Následující příklad přiřadí proměnné 'x' výsledek funkce readSensor()[17], zastaví činnost programu na 50 milisekund a poté bude smyčku opakovat, dokud nebude 'x' větší nebo rovno 100:

```
do  // začátek smyčky; podmínka je na konci
{
  x = readSensor();  // přiřaď proměnné 'x' výsledek funkce readSensors()
  delay (50);  // čekej 50 milisekund
}
while (x < 100);  // znovu na začátek smyčky, dokud je 'x' menší než 100.
                  // Až bude 'x' 100 nebo víc, smyčka skončí.
```

Funkce

Funkce je část kódu, který je pojmenován a který ve svém těle obsahuje blok příkazů, které jsou provedeny, když je tato funkce zavolána (spuštěna). Funkce obvykle vykonávají opakující se úkony a slouží také k udržení přehlednosti programu.

Funkce musí být před použitím nejprve deklarována, čili musí být uvedeno, jaké jsou její parametry a jakou hodnotu vrací („jakého je typu“). To je datový typ hodnoty, kterou bude funkce vracet jako výsledek své činnosti, například 'int' (zkratka pro integer), pokud funkce vrací celočíselnou hodnotu. Při deklaraci funkce se jako první uvede její typ (pokud se nemá vracet žádná hodnota, deklarujeme ji s typem void, česky prázdno, nic). Po uvedení typu následuje název funkce a za ním v kulatých závorkách seznam parametrů, předávaných funkci, s uvedením jejich typu. Pokud funkce nemá parametry, budou kulaté závorky prázdné, nebo do nich napíšeme void, aby bylo explicitně jasné, že tam nic není.

Deklarace funkce:

```
typ jmenoFunkce (parametry);
```

[17] *To je jen jakási funkce, která vrací nějakou hodnotu. Její přesná činnost je teď nedůležitá.*

Zároveň s deklarováním funkce často rovnou i definujeme, co funkce dělá, a to tak, že uvedeme i její tělo:

Definice funkce:

```
typ jmenoFunkce (parametry)
{
  příkazy;
}
```

Při volání funkce se uvede její jméno a do závorek parametry, tentokrát už bez uvedení datových typů. Pokud funkce něco vrací a chceme tuto hodnotu využít, pracujeme s ní tak, jako kdybychom na příslušném místě tuto hodnotu zapsali, například:

```
// vrati největší ze 4 zadaných prvků
maximum = najdiNejvetsiPrvek(a,b,c,d);
delkaVcentimetrech = delkaVpalcich(predmet) * 2.54;
```

Pokud nás výsledná hodnota nezajímá, pak ji nikam nepřiřadíme ani ji nevyužijeme pro výpočet:

```
vypniTopeni(obyvak);
```

Pokud funkce nemá žádné parametry, uvedou se při volání prázdné závorky (nesmějí se vypustit), například:

```
zapniMotor();
if(tlacitkoZmacknuto())
{
  pocetZmacknuti++;
}
```

Příklad definice funkce:

Následující funkce readDelayVal() typu integer má sloužit v programu ke zjištění velikosti zpoždění čtením hodnoty napětí z potenciometru a přepočítáním na požadovaný rozsah, řekněme od 0 do 255. Nejprve deklarujeme lokální proměnnou 'v'. Dalším příkazem do ní vložíme hodnotu, přečtenou z potenciometru pomocí funkce analogRead[18], která vrací hodnotu v rozsahu od 0 do 1023. Pro získání hodnoty v rozsahu od 0 do 255 tuto hodnotu v dalším příkazu vydělíme 4. Nakonec pomocí příkazu return vrátíme výsledek tomu, kdo si tuto funkci zavolal.

```
int readDelayVal()
{
  int v;   // vytvoří lokální proměnnou 'v'
  v = analogRead(pot);  // přečte hodnotu potenciometru
```

[18] *Jedna z knihovních funkcí pro Arduino*

```
  v /= 4;  // konvertuje číslo z rozsahu 0 až 1023 na 0 až 255
  return v;  // ukončí funkci a vrátí hodnotu proměnné 'v' jako hodnotu této funkce
}
```

Téhož výsledku bychom mohli dosáhnout i zkráceným zápisem bez použití proměnné:

```
int readDelayVal()
{
  // přečte hodnotu potenciometru, vydělí 4 a vrátí jako výsledek funkce
  return analogRead(pot) / 4;
}
```

Příkazem return se funkce ukončí, což můžeme využít kdekoli uvnitř funkce, pokud zjistíme, že už nepotřebujeme, aby dále běžela, například:

```
void zatop(int tepelnaPohoda)
{
  if (prectiTeplotu() > tepelnaPohoda)
    return;
  zavriOkno();
  zapniTopeni();
}
```

V tomto případě funkce ani nic nevrací, takže příkaz return bude uveden samotný, bez nějaké hodnoty. Funkce, která nic nevrací, nemusí return vůbec mít – když se příkazy vykonají až do konce funkce, ta prostě skončí. Pokud by ale funkce měla něco vracet, tak samozřejmě příkaz return musíme použít a něco pomocí něj vrátit.

Digitální vstupy a výstupy

pinMode(pin, režim)

Tato funkce se používá pro nastavení určitého pinu na vstup (INPUT) nebo výstup (OUTPUT).

Nastavení pinu na výstup se provede s použitím hodnoty OUTPUT:

```
pinMode(pin, OUTPUT);  // nastav 'pin' jako výstupní
```

Nastavení pinu na vstupní se provede s použitím hodnoty INPUT:

```
pinMode(pin, INPUT);  // nastav 'pin' jako vstupní
```

Digitální piny Arduina jsou standardně nastaveny jako vstupní, takže je není nutno do tohoto stavu na začátku běhu programu pomocí funkce `pinMode()` nastavovat. Chování pinů ale můžeme v průběhu programu měnit podle potřeby tam a zpět, pokud to připojená elektronika umožňuje. Nastavení pinu do stejného režimu, v jakém již je, nevadí.

Piny, nakonfigurované jako vstupní, je nutno vždy připojit na správnou logickou úroveň (LOW nebo HIGH). Tu zajistí buď připojené externí zařízení, nebo k nim mohou být programově připojeny zdvihací (pull-up) rezistory o velikosti 20 až 50 kOhm, obsažené v interní struktuře mikrokontroléru. Ty pinům zajistí bezpečnou logickou úroveň HIGH i v případě, že nejsou nikam připojeny. Ve výchozím nastavení mikrokontroléru jsou tyto zdvihací rezistory odpojeny; připojují se použitím parametru `INPUT_PULLUP` při nastavování pracovního módu:

```
// nastav 'pin' jako vstupní a zapni pull-up rezistor
pinMode(pin, INPUT_PULLUP);
```

Pull-up rezistory se běžně používají například pro jednoduché připojení spínače.

Poznámka:

Proud, který je možno z pinu odebírat, je omezený. Například Arduino UNO může na pinech, nakonfigurovaných jako výstup, poskytnout proud až 40 mA do jiných zařízení či obvodů. Takový proud postačí pro jasné rozsvícení LED (nezapomeňte na předřadný rezistor), ale není to dostatečný proud pro sepnutí většiny relé, solenoidů nebo motorů.

Zkraty na pinech Arduina nebo jejich zatížení nadměrným proudem může obvod výstupního pinu uvnitř mikrokontroleru poškodit nebo zničit, případně poškodit nebo zničit celé Arduino. Proto je vhodné místo přímého propojení pinů použít rezistor s odporem o přibližné hodnotě 220 Ω, zapojený mezi pin Arduina a připojované zařízení.

digitalRead(pin)

Funkce digitalRead(pin) přečte stav určeného digitálního pinu. Vrací hodnotu HIGH nebo LOW podle toho, jestli je na pinu vysoká nebo nízká úroveň.

Příklad:

```
stav = digitalRead(pin);  // uloží do proměnné 'stav' aktuální stav pinu 'pin'
if (digitalRead(pin) == HIGH)
  Serial.println("Zapnuto");  // je-li na pinu 'pin' vysoká úroveň, napíšeme to.
```

Číslo pinu je číselná hodnota 0 až 13 (pro základní Arduina, na jiných to může být jinak). Doporučujeme si jednotlivé piny pojmenovat (viz kap. Konstanty na str. 10), výrazně se tím zlepší čitelnost programu.

digitalWrite(pin, hodnota)

Nastaví zadaný digitální pin na vysokou (HIGH) nebo nízkou (LOW) úroveň.

Příklad:

```
digitalWrite(pin, HIGH);  // nastav 'pin' na HIGH
```

Číslo pinu je číselná hodnota 0 až 13 (pro základní Arduina, na jiných to může být jinak). Doporučujeme si jednotlivé piny pojmenovat (viz kap. Konstanty na str. 10), výrazně se tím zlepší čitelnost programu.

Následující program čte stav tlačítka, připojeného na digitální vstup a pokud je sepnuté, rozsvítí LED, připojenou k digitálnímu výstupu.

```
const int PIN_LED = 13;  // LED je připojená na pin 13
const int PIN_TLACITKO = 7;  // tlačítko je připojené na pin 7

void setup()
{
  pinMode(PIN_LED, OUTPUT);  // nastav pin s LED jako výstup
  pinMode(PIN_TLACITKO, INPUT);  // nastav pin s tlačítkem jako vstup
}

void loop()
{
  int stav;  // proměnná pro uložení přečtené hodnoty
  stav = digitalRead(PIN_TLACITKO);  // vlož do proměnné 'value' stav vstupního pinu
  digitalWrite(PIN_LED, stav);  // nastav LED podle stavu tlačítka
}
```

Analogové vstupy a výstupy

analogRead(pin)

Přečte hodnotu napětí z určeného analogového pinu v 10-bitovém rozlišením. Tato funkce pracuje pouze s analogovými piny (A0 až A5). Výsledná hodnota je celočíselná s rozsahem od 0 do 1023.

Příklad:

```
// nastav hodnotu proměnné 'value' podle napětí na pinu 'pin'
value = analogRead(pin);
```

Poznámka:

Analogové piny, na rozdíl od těch digitálních, nemusí být nejprve deklarovány jako vstupní nebo výstupní (použitím funkce analogRead se přepnou do režimu pro čtení analogové hodnoty automaticky). Piny A0 – A5 můžeme použít i jako digitální, v takovém případě se nastavují obdobným způsobem, jako ostatní vstupně-výstupní piny pomocí funkce pinMode a čtou či mění funkcemi digitalRead a digitalWrite.

analogWrite(pin, value)

Zapíše pseudo-analogovou hodnotu, generovanou pomocí hardwarově řízené pulsní šířkové modulace (PWM) na výstupní pin. Na Arduino Uno a podobných můžeme použít piny 3, 5, 6, 9, 10 a 11.

Příklad:

```
analogWrite(pin, hodnota);  // zapíše 'value' analogově na 'pin'
```

Hodnota 0 zaslaná jako parametr 'hodnota' nastavuje na zadaném výstupním pinu napětí 0 voltů, hodnota 255 nastavuje na zadaném výstupním pinu napětí, které odpovídá napájecímu napětí modulu Arduino (obvykle 5 voltů nebo 3,3 voltů; v dalším textu budeme používat označení *max*).

Aby bylo možno dosáhnout hodnot napětí mezi 0 a *max*, střídá se na výstupním pinu logická hodnota 0 a 1 neboli napětí 0 V a *max* V. Poměr tohoto střídání je určen parametrem 'hodnota' (0 až 255) – čím vyšší hodnota, tím déle je na pinu vysoká logická úroveň. Například, pro hodnotu 64, tj. ¼ z 256, je na pinu úroveň 0 tři čtvrtiny času, a 1 čtvrtinu času, pro hodnotu 128 bude na pinu úroveň 0 polovinu času a 1 druhou polovinu, a hodnota 192 znamená, že na pinu bude hodnota 0 čtvrtinu času a 1 tři čtvrtiny času.

Generování PWM signálu je podporováno přímo hardwarovým jádrem mikrokontroleru, takže pro běh stačí nastavit požadovanou hodnotu jednou. Až do příštího volání funkce analogWrite (nebo do zavolání funkce digitalRead či digitalWrite na stejném pinu) se bude tento signál generovat „sám“.

Poznámka:

Analogové piny – na rozdíl od těch digitálních, nemusí být nejprve deklarovány jako vstupní nebo výstupní.

Následující příklad čte analogovou hodnotu ze vstupního pinu, upraví její hodnotu vydělením čtyřmi a nastaví na výstup PWM signál s odpovídající hodnotou. Připojená LED bude podle toho svítit více nebo méně jasně[19]:

```
const int PIN_LED = 10;  // LED s rezistorem 220 ohm na pinu 10
const int PIN_POTENCIOMETR = 0;  // potenciometr na analogový pin 0
```

[19] *Ve skutečnosti bude blikat se zadaným poměrem na frekvenci cca 490 Hz, ale setrvačnost lidského oka vyvolá dojem jasnějšího nebo méně jasného svitu.*

```
void setup()
{
  // žádné nastavení není potřebné
}

void loop()
{
  int hodnota;  // proměnná k uložení aktuální hodnoty z potenciometru
  // přečti a ulož do proměnné 'hodnota'
  hodnota = analogRead(PIN_POTENCIOMETR);
  hodnota /= 4;  // konvertuj rozsah 0-1023 na 0-255
  // vyšli výstupní PWM signal na pin s připojenou LED
  analogWrite(PIN_LED, value);
}
```

Časování

delay(milisecond)

Pozastaví program na dobu určenou v milisekundách, jedné sekundě tedy odpovídá hodnota 1 000.

Příklad:

```
delay(500);  // čekej půl sekundy
delay(1000);  // čekej jednu sekundu
```

delayMicroseconds(microsecond)

Pozastaví program na dobu určenou v mikrosekundách, jedné sekundě tedy odpovídá hodnota 1 000 000.

Příklad:

```
delayMicroseconds(1000);  // čekej jednu milisekundu
delayMicroseconds(500000);  // čekej půl sekundy
```

millis()

Vrací počet milisekund od startu aktuálního programu (od zapnutí napájení, posledního resetu nebo nahrání programu).

Příklad:

```
unsigned long cas;  // deklaruje proměnnou 'cas' typu unsigned long
cas = millis();  // uloží do proměnné 'cas' času od startu (v milisekundách)
```

Poznámka:

Typ unsigned long může zachytit číslo od 0 do 4 294 967 295, takže hodnota této proměnné přeteče (přetočí se zpět na nulu) přibližně po 49 dnech nepřetržitého běhu Arduina. Pokud bychom použili typ int, došlo by k přetečení už za cca 9 hodin (a hodnota by pak byla záporná, protože typ int je celé číslo mezi -32 768 a +32 767).

micros()

Vrací počet mikrosekund od okamžiku rozběhu aktuálního programu (od zapnutí napájení, posledního resetu nebo nahrání programu).

Příklad:

```
// deklaruje proměnnou 'mikroCas' typu unsigned long
unsigned long mikroCas;
// uloží do proměnné 'mikroCas' čas od startu (v mikrosekundách)
mikroCas = micros();
```

Poznámka:

Typ unsigned long může zachytit číslo od 0 do 4 294 967 295, takže hodnota této proměnné přeteče (přetočí se zpět na nulu) přibližně po 71 minutách běhu Arduina.
Rozlišení funkce je u 16 MHz procesorů 4 mikrosekundy a u 8 MHz 8 mikrosekund, výstupem funkce tedy budou násobky čtyř, nebo osmi.

Matematické funkce

min(a,b)

Vrátí menší ze zadaných dvou čísel. Používá se například k omezení hodnot ze senzoru, aby nedošlo k překročení určité meze.

Příklad:

```
value = min(value, 100);  // nastaví 'value' na menší z hodnot 'value' nebo 100,
                          // čímž zajistí, že proměnná 'value' nepřekročí 100
```

Vzhledem k tomu, jak je tato funkce implementována, nepoužívejte jako její argument návratovou hodnotu funkce. Používejte ji pouze na konstanty nebo proměnné.

```
// ŠPATNĚ:
x = min(nejakaFunkce(), cokoli);

// SPRÁVNĚ:
x = nejakaFunkce();
x = min(x, cokoli);
```

max(a,b)

Vrátí větší ze zadaných dvou čísel.

Příklad:

```
value = max(value, 100);  // nastaví 'value' na větší z hodnot 'value' nebo 100,
                          // čímž zajistí, že proměnná 'value' neklesne pod 100
```

Vzhledem k tomu, jak je tato funkce implementována, nepoužívejte jako její argument návratovou hodnotu funkce. Používejte ji pouze na konstanty nebo proměnné.

```
// ŠPATNĚ:
x = max(nejakaFunkce(), cokoli);

// SPRÁVNĚ:
x = nejakaFunkce();
x = max(x, cokoli);
```

abs(a)

Vrátí absolutní hodnotu čísla. Vstupem je číslo a výstupem jeho absolutní hodnota.

Příklad:

```
x = abs(150);  // x bude 150
x = abs(-150);  // x bude taky 150
```

Vzhledem k tomu, jak je tato funkce implementována, používejte ji pouze na konstanty nebo proměnné, nikoli na výrazy nebo návratovou hodnotu funkce.

```
// ŠPATNĚ:
x = abs(nejakaFunkce());

// SPRÁVNĚ:
x = nejakaFunkce();
x = abs(x);
```

Poznámka:

Absolutní hodnota čísla x je nezáporné číslo. Pokud je x >= 0, abs(x) = x. Pokud je x < 0, potom abs(x) = -x. Laicky řečeno je absolutní hodnota vzdálenost čísla od nuly na číselné ose.

constrain(x,a,b)

Tato funkce je kombinací funkcí min() a max() s vhodnými parametry. Slouží k omezení rozsahu proměnné jak shora, tak zdola. Má tři parametry: vstupní (upravovanou) hodnotu, dolní mez a horní mez. Pokud vstupní číslo klesne pod spodní mez, výstupem je hodnota spodní meze. Překročí-li horní mez, výslednou hodnotou je hodnota horní mez. Pokud je hodnota v mezích, výstupem funkce bude stejná hodnota, jako vstupní.

Příklad:

```
x = constrain(upravovanaHodnota, dolniMez, horniMmez);

x = constrain(1,10,100);  // x bude 10
x = constrain(150,10,100);  // x bude 100
x = constrain(50,10,100);  // x bude 50
```

map(x,a,b,A,B)

Stejně jako funkce constrain(), slouží i funkce map() k úpravě rozsahu proměnné. Na rozdíl od předchozí funkce však nedochází k oříznutí hodnot, ale k rovnoměrnému „roztažení“ stupnice.

Dá se použít například k úpravě hodnoty v procentech (0 až 100) pro použití ve funkci analogWrite, která pracuje s hodnotami od 0 do 255.

Příklad:

```
void setup()
{
  Serial.begin(115200);
}

void loop()
{
  for(int procent=0;i<=100;i++)
  {
    analogWrite(PIN_LED, map(procent,0,100,0,255));
    delay(50);
  }
}
```

Upozornění:

Funkci je možné použít i pro „stlačení“, tj. převod z většího rozsahu na menší, ale nebude to zcela rovnoměrné (čím větší nepoměr mezi velikostí vstupního a výstupního rozsahu, tím hůř).

Následující příklad čte hodnotu z potenciometru a podle toho nastavuje svit LED. Protože funkce analogRead vrací číslo v rozsahu od 0 do 1023, je nutné přepočítat na rozsah, který vyžaduje funkce analogWrite, tj. od 0 do 255:

```
const int PIN_LED = 10;
const int PIN_POTENCIOMETR = A3;

void setup()
{
  // není nic potřeba
}

void loop()
{
  int hodnota = analogRead(PIN_POTENCIOMETR);
  hodnota = map(hodnota, 0, 1023, 0, 255);
  analogWrite(PIN_LED, hodnota);
}
```

pow(zaklad,exponent)

Funkce pro umocnění čísla. Vstupními parametry jsou číslo a požadovaná mocnina.

Příklad:

```
x = pow(10,3);  // x bude 1000
```

```
y = pow(10,4);  // y bude 10000
z = pow(2,5);  // x bude 32;

int a = 10;
int b = 3;
float c;

void setup()
{
  Serial.begin(115200);
}

void loop()
{
  c = pow(a,b);
  Serial.println(c);
  delay(1000);
}
```

sqrt(cislo)

Funkce vrací druhou odmocninu vstupního čísla.

Příklad:

```
x = sqrt(25);  // x bude 5
y = sqrt(256);  // y bude 16
z = sqrt(2);  // z bude cca 1,41
```

Náhodná čísla

randomSeed(zaklad)

Nastavuje výchozí bod[20] pro funkci random().

Příklad:

```
randomSeed(value);  // použij hodnotu proměnné 'value' jako semínko
```

[20] *Anglicky seed = „semínko“.*

Arduino není schopno vytvořit skutečně náhodné číslo. Místo toho se vytváří tzv. pseudonáhodné číslo, jehož hodnota se odvíjí od nějaké počáteční hodnoty. Funkce randomSeed umožňuje tuto počáteční hodnotu nastavit, což pomáhá vytvořit „náhodnější náhodná čísla“. Existuje celá řada možností, jak nastavit semínko, včetně funkce analogRead(), která může číst elektrický šum z analogového pinu. Při použití stejného semínka bude generátor při každém zapnutí Arduina produkovat stejnou sekvenci pseudonáhodných čísel. I čtení aktuálního času pomocí funkce millis může být pořád stejné, pokud to bude první věc, kterou po startu Arduina uděláte (protože ten čas od startu bude vždy stejný). Tato funkce však bude poskytovat pěkné semínko, když ji použijete až po nějakém čekání na uživatelský vstup (člověk nikdy nezmáčkne tlačítko ve stejný čas).

random(max) a random(min, max)

Funkce random vrací pseudonáhodná čísla v rozsahu 0 až max, nebo min až max.

Příklad:

```
znamka = random(1,5);  // náhodně oznámkuj písemku
// ulož do proměnné 'nahoda' náhodné sudé číslo od 0 do 200
nahoda = 2 * random(100);
```

Poznámka:

Abyste se vyvarovali stejné sekvenci náhodných čísel při každém spuštění Arduina, použijte vždy funkci randomSeed().

Následující příklad vytvoří náhodnou hodnotu mezi 0 až 255 a použije ji pro řízení úrovně PWM signálu na pinu 10. Bude-li připojena LED, bude se její jas náhodně měnit 2x za vteřinu:

```
const int PIN_LED = 10;  // LED s rezistorem 220 ohm na pin 10
const int PIN_ANTENA = A1;  // analogový vstup

void setup()
{
  // nic se nenastavuje
}

void loop()
{
  int nahodneCislo;  // proměnná pro uložení náhodného čísla
  randomSeed(analogRead(PIN_ANTENA));  // použij analogový vstup jako semínko
  nahodneCislo = random(255);  // ulož náhodné číslo v rozsahu 0 až 255 do proměnné
  analogWrite(PIN_LED, nahodneCislo);  // zapiš hodnotu proměnné na výstup
  delay(500);  // čekej 0,5 sekundy
}
```

Komunikace

Serial.begin(rate)

Otevře sériový port a nastaví komunikační rychlost pro přenos dat. Pro přenos je v řadě příkladů pro Arduino použita rychlost 9600 baudů[21]. Otevření seriové linky a nastavení rychlosti se nejčastěji provádí ve funkci setup.

Příklad:

```
void setup()
{
  // otevři sériový port a nastav přenosovou rychlost na 9600 bps
  Serial.begin(115200);
}
```

Poznámky:

Při použití sériové komunikace nelze použít piny 0 (RX) a 1 (TX) současně jako digitální.

Nezapomeňte nastavit na druhé straně komunikace (PC, jiné Arduino...) tutéž přenosovou rychlost.

Serial.print(data) a Serial.println(data)

Funkce odesílají data pomocí sériové linky. Funkce print odešle data tak jak jsou, funkce println navíc odřádkuje (to může být čitelnější, pokud zobrazujeme data v programu Serial Monitor vývojového prostředí Arduino IDE).

Pomocí těchto příkazů můžeme poslat přes seriovou linku22 libovolný datový typ. Pro sledování v počítači můžete použít „Seriový monitor“, který je součástí prostředí Arduino[23], nebo můžete použít libovolný program, který umí po seriové lince komunikovat[24]. V Seriovém monitoru Arduina nezapomeňte nastavit rychlost, kterou jste použili pro inicializaci funkcí `Serial.begin` (výběrem ze seznamu v pravém dolním rohu).

---

[21] Běžné Arduino (a to i levné neoriginální klony) obvykle zvládnou i rychlosti podstatně vyšší, například 115200, ne vždy však zcela libovolnou rychlostí. Pro komunikaci s počítačem je potřeba zvolit některou ze standardních „normovaných“ rychlostí: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, 115200, 128000, 250000, 256000 nebo podle možností Vaší aplikace. Čím vyšší číslo, tím rychlejší přenos, ale také tím náchylnější k poruchám.

[22] I když dnešní počítače seriový port už obvykle nemají, většina Arduin má přímo v sobě převodník z USB a po připojení do počítače vytvoří „virtuální“ seriový port.

[23] Pro použití Seriového monitoru nejprve vyberte správný port v menu Nástroje/Port a pak otevřete monitor z menu Nástroje/Seriový monitor, klávesová zkratka Ctrl+Shift+M

[24] Je jich hodně. Na Windows třeba PuTTY, RealTerm, na Mac OS screen, Serial, na linuxu cu, screen, putty atd.

Příklad:

```
Serial.print("Ne, nevyhral jsi. Myslel jsem si cislo ");  // vypiš text
Serial.println(random(123));  // vypiš náhodné číslo mezi 0 a 123.
```

Následující jednoduchý příklad jednou za sekundu přečte napětí z analogového pinu 0 a odešle hodnotu do počítače.

```
const int PIN_VSTUP = A0;  // chceme například číst z analogového vstupu 0

void setup()
{
  Serial.begin(115200);  // nastav komunikační rychlost na 115200 bps
}

void loop()
{
  Serial.println(analogRead(PIN_VSTUP));  // odešli analogovou hodnotu
  delay(1000);  // čekej 1 sekundu
}
```

Poznámka:

Pro přenos dat po seriové lince existuje řada možností. Další informace o použití funkcí Serial.println() a Serial.print() naleznete na webových stránkách Arduino.cc.