C++ Fokus på sproget #04

in C++ Fokus with 0 Comments

Valg af datatype i Arduino

I en tidligere post har vi beskæftiget os med variable og hvorledes man navngiver, så ens kode bliver let at læse. Det er vigtigt at huske på, at man i C++ skelner mellem store og små bogstaver. På engelsk kalder man fænomenet 'case sensitive'. To variable, som hedder henholdsvis 'MyVariable' og 'myvariable', er i C++ forskellige og kan anvendes i en sketch på samme tid. Imidlertid er det bad code, og man vil nok aldrig navngive sådan i virkeligheden.

Fik jeg nævnt, at du så vidt muligt bør undgå at bruge forkortelser når du navngiver. Det kan godt være, at du til evig tid kan huske, at variablerne OTemp og ITemp gemmer informationer om temperaturen henholdsvis ude og inde, men jeg kan garantere dig, at andre meget hurtigere kan gennemskue din kode, hvis du i stedet skriver OutsideTemperature og InsideTemperature.

Man kan erklære, (engelsk: declare) en variabel som i dette eksempel:

int MyVariable=0;

Når du erklærer dine variable, skal du angive, hvilken type den pågældende variabel tilhører. Det er nødvendigt for at compileren kan vide, hvorledes den skal håndtere variablen. En Arduino sketch kan anvende alle datatyper fra C++. Nogle af de mest hyppige datatyper, du kommer til at anvende, er heltal eller integers, som det hedder på engelsk. Nogle gange har man behov for at kunne arbejde med negative tal og andre gange bruger man kun de positive, hele tal. De sidstnævnte er tal uden fortegn og benævnes 'unsigned'. En meget anvendt type er 'unsigned int'. Den fylder 2 bytes, 16 bits og kan rumme tal fra 0 til 65 535, hvilket er det samme som 2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2-1=2^16-1). 16 pladser tildelt værdien 2 og ganget med hinanden, minus 1. Typen 'int' fylder ligeledes 2 bytes, men da den har fortegn, kan den rumme værdier fra -32 768 til 32 767. Når man anvender integers skal man med andre ord gøre sig klart, om man har behov for negative tal eller om værdierne altid er positive.

int MyVariable=0;

og

unsigned int MyVariable=0;

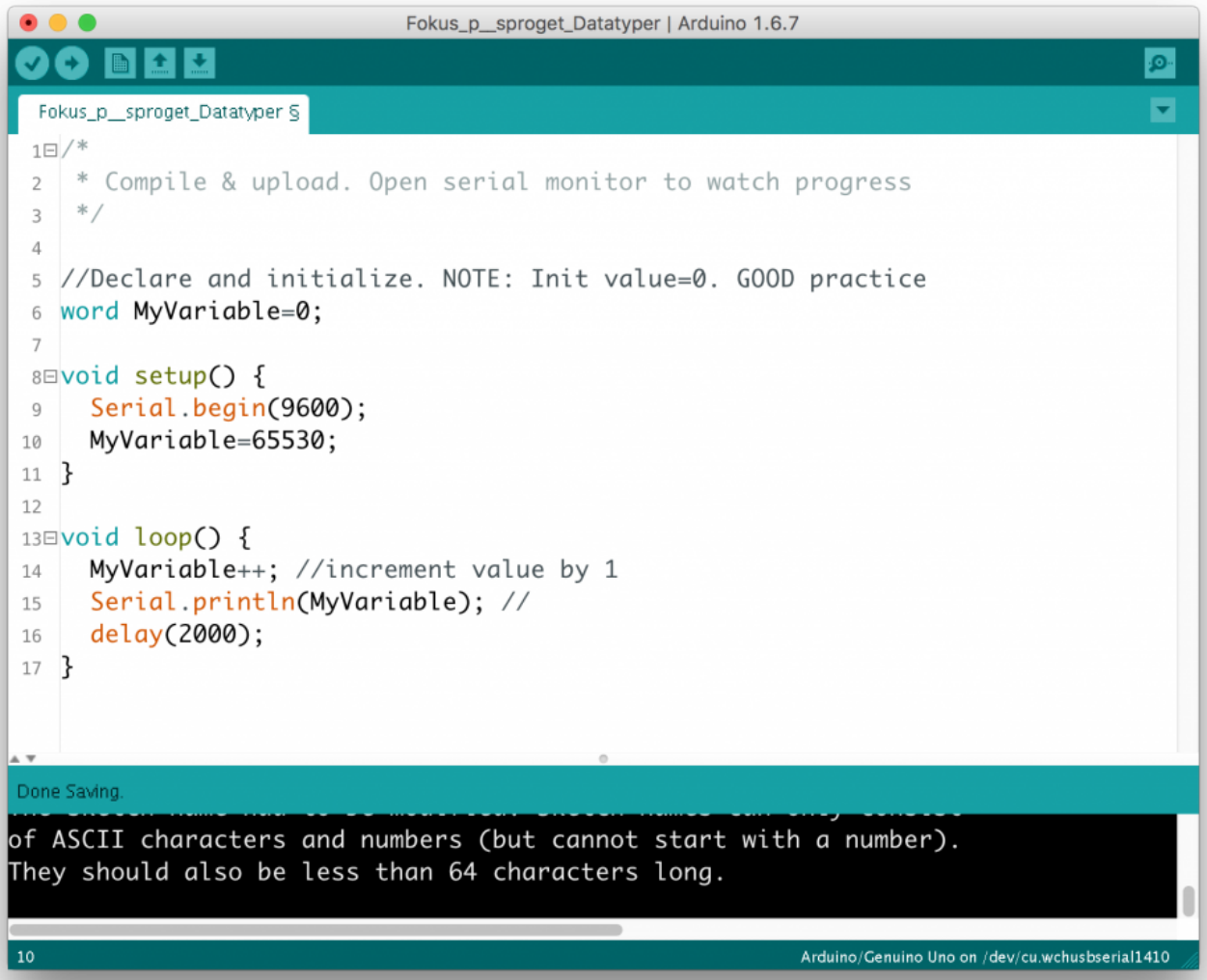
Når man læser koden til mange af de sketches, som er produceret til Arduino, kan man se, at mange programmører lidt tankeløst vælger typen int til deres integer variabler. Der er imidlertid et par rigtig gode grunde til at tænke lidt mere over valg af datatyper. Hvis man vælger dem, så ens variabler optager mindst mulig plads i memory, kan man frigøre værdifuld plads. Arbejder man f. eks. med Blynk, kommer man let i en situation, hvor hver eneste byte tæller, fordi Blynk library sammen med Ethernet library tilsammen bruger løs af memory og ikke efterlader meget memory til resten af din sketch. Bruger man Uno, Nano eller Yún, som alle er baseret på Atmega328 kommer man hurtigt i bekneb for plads. Hvis man f. eks. skal bruge en variabel til at lagre et pin-nummer, er der ingen grund til at anvende int. Typen 'byte' vil her være det korrekte valg: den fylder kun én byte og rummer værdier op til og med 255. Alt rigeligt til de pins, der er på en Arduino. (Typen byte er i sig selv unsigned.) Den anden gode grund til at være omhyggelig med at vælge den rette datatype finder vi ved at betragte, hvad der sker, hvis en variabel "løber over". Hvad sker der, hvis vi lægger 1 til en unsigned int, som i forvejen har nået

max og nu rummer tallet 65 536? Altså følgende tilfælde:

unsigned int MyNumber = 65 535;

MyNumber = MyNumber + 1;

Prøv at indtaste følgende kode og prøv at køre den:



Det viser sig, at MyNumber derved får værdien 0. Den starter så at sige forfra. It rolls over.

Noget tilsvarende sker der, hvis vi erklærer en variabel af typen int og trækker 1 fra, når den har værdien – 32 768. Når Arduinoen regner på det, bliver resultatet 32 767. Har den værdien 32 767 og vi lægger 1 til ender vi på -32 768. Se det er noget som kan skabe ravage i ens sketch og give nogle resultater, som er totalt mærkelige. Det undgår man, hvis man altid kritisk og velovervejet tager stilling til datatyper. Så man skal være opmærksom på, at man både kan vælge typer, som er alt for rummelige og typer som ikke kan lagre så store tal, som man har brug for.

Af og til vil du støde på tal, som er skrevet på denne måde:

```
Addr=0x10;
```

AddrWrite=0xB8;

Det kan se lidt kryptisk ud, men der er i virkeligheden blot tale om integers, som er skrevet i 16-talssystemet. Sætter man 0x foran sit tal, fortæller man compileren, at vi arbejder hexadecimalt eller base 16, som man også kan sige det. Så tallet ox10 betyder i virkeligheden tallet '10' i base 16. Og man udtaler det 'et – nul'. Oversat til vores eget talsystem betyder det 1 16'er og 0 enere. Dvs. 0x10 = 16. I vores talsystem har vi i sagens natur 10 cifre, men i base 16 har man brug for 16. Derfor svarer A til 11, B til 12 osv. Så 'B – otte' svarer til 8 16'ere og otte enere, hvilket netop er tallet 136. Dvs. 0xB8 = 136 Jeg er oftest stødt på disse tal i forbindelse med adresser på I2C enheder o.l.

Mange gange har man brug for at der er komma i tallet. Så har man brug for variable af typen 'float' eller 'double'. I C++ er der forskel på de to typer, men i Arduino er de ens og fylder 4 bytes. De rummer tal fra 3.4028235E+38 ned til -3.4028235E+38. Det er jo nogle voldsomt store og nogle voldsomt små tal og umiddelbart ser det ud til, at vi er dækket godt ind, for hvem har brug for større tal? Det er jo mere end en trillion trilliarder! Hvem har brug for mere? Svaret er, at det har vi, radioamatørerne. For bag de svimlende tal gemmer sig en meget dårlig præcision. Det forholder sig nemlig sådan, at af de 32 bits som udgør et tal defineret som float, er nogle af dem reserveret til at rumme eksponenten. Helt præcist er 9 bits afsat til det, hvilket efterlader 23 bits til betydende cifre, – cifre man kan stole på. Eller sagt på mere almindeligt dansk. Du kan kun stole på 6-7 cifre i dit tal. Det er ofte tilstrækkeligt, men hvis man nu har lavet en VFO med en DDS, skulle man jo eksempelvis gerne kunne indstille frekvensen 21 355 000 Hz. Hvis man ikke tager sine forholdsregler, kan man ikke stole på de to sidste cifre, og frekvensen kan komme til at ligge op til 100 Hz forkert. Denne unøjagtighed bliver forværret, hvis man har behov for at gange eller dividere for at nå frem til det endelige tal. Hver gang tallet indgår i en beregning, bliver unøjagtigheden større. Vi radioamatører kommer typisk ud for denne problematik i forbindelse med VFO- og GPS-konstruktioner. For at komme om ved dette, holder man sig i disse tilfælde til at erklære sine variable som 'long' eller 'long long'. Men det har en pris, idet det tager ekstra lang tid for controlleren at udføre beregninger med disse typer. En long fylder 4 bytes og kan være signed eller unsigned. En unsigned long kan rumme tal fra 0 til 4 294 967 295, dvs. 9-10 betydende cifre. Den skal erklæres på følgende måde:

long MyVariable = 4123123123L

MyVariable kan naturligvis initialiseres med andre startværdier, men læg lige mærke til 'L' til slut.

At arbejde med long eller long long har dog sin pris. Mens de bearbejdes fylder de meget i memory og det koster tid.

Hele denne problematik med ringe nøjagtighed på decimaltal skyldes den måde, som vores Atmega-controller har fået implementeret sine instrukser på. Skriver man kode i C++ til en Mac eller en PC kommer man langt ved at anvende 'double' eller 'long double' idet de almindeligvis arbejder med 64 bit henholdsvis 128 bit, men også her kan programmører være tvunget til at finde særlige løsninger, hvis der er behov for mange cifres nøjagtighed.

Char giver sig selv, men da C++ ikke default har 'string' som datatype, skal man anvende særlige løsninger, hvis man skal arbejde med tekststrenge. Der findes imidlertid gode løsninger på dette. Se f. eks. her. Er du ved at nå grænsen for, hvor stor din sketch kan være er strings et af de steder, hvor man kan gøre sin kode effektiv. Tænk på, at hvert eneste tegn fylder en byte, så hvis man har mange meddelelser i sin sketch, fylder de godt til i memory. En af måderne at håndtere sine "faste tekster" på, består i at lagre dem i EEPROM, men der findes andre snedige måder at optimere sin kode på dette punkt. Prøv at Google: optimize code string arduino. Det kunne lyde som om det er i modstrid min tidligere opfordring til at skrive rigeligt med kommentarer. Kommentarer er jo også tekst, men kommentarer bliver IKKE compileret og fylder derfor ikke op i den compilerede sketch.

Her er de så, datatyperne:

Datatyper i Arduino

Type	Længde i bytes	Range
byte	1	0 til 255
int	2	-32 768 til 32 767
unsigned int	2	0 til 65 535
word	2	0 til 65 535
long	4	-2 147 483 648 til 2 147 483 647
unsigned long	4	0 til 4 294 967 295
float	4	-3,4028235E+38 til 3,4028235E+38
double	4	-3,4028235E+38 til 3,4028235E+38
char	1	-128 til 127
unsigned char	1	0 til 255
string	?	se kommentaren ovenfor i denne post
boolean	1	true eller false

In short

Sørg for at dine variabler fylder så lidt som muligt.Der er f. eks. ingen grund til at vælge int som type, hvis du blot skal lagre et pin-nummer. Her er typen ‘byte’ tilstrækkelig, da den rummer værdier fra 0 til 255. Den fylder kun en byte mens en int fylder 2 bytes. Vælg en unsigned type, hvis du ikke har brug for negative tal. Tænk dig godt om, hvis du har behov for stor præcision. Får du problemer med mangel på memory, er det værd at gennemgå sine datatyper og specielt være opmærksom på, hvorledes man håndterer tekst. Kommentarer, som er markeret med // eller /* og */ fylder ikke, da compileren ignorerer dem. Så skriv bare løs!