Användning av argument till program

Denna skrift visar hur man skriver program som kan använda argument på kommandoraden.

1 Deklarera `main()` korrekt

Det finns lite olika sätt att deklarera `main()`, beroende på om man vill använda argument till programmet eller inte.

Så här kan man skriva när man inte tänker använda argument:

```
int main()
```

Observera att `main()` *alltid* ska deklareras `int` och *aldrig* som någonting annat (som t ex `void`). `main()` är en funktion precis som alla andra, men koden som anropar den väntar sig ett värde tillbaka.

Om du inte gillar meddelandet från kompilatorn om att returvärde saknas, lägg till ett `return 0` då!

För att använda argument skriver man så här:

```
int main(int argc, char *argv[])
```

Nu kommer argumenten till programmet att finnas tillgängliga i `main()` via dess argument `argc` och `argv`.

2 Hur argumenten representeras

Själva namnen `argc` och `argv` spelar egentligen ingen roll, men de är standard och betyder *argument counter* respektive *argument vector*.

`argv` är en array av strängar, där varje element (varje sträng) är ett av argumenten som programmet startades med.

`argc` är ett tal som säger hur många strängar som finns i argumentvektorn.

Elementen i `argv` är numrerade 0 ... `argc`-1, dvs på vanligt C-manér.

`argv[0]` innehåller normalt programnamnet[1]. Om t ex `testprog` startas utan argument gäller:

```
argc == 1
argv[0] == "testprog"
```

När programmet startas med argument hamnar de i `argv` i samma ordning som de angavs på kommandoraden. T ex om vi skriver `testprog foo gurka` gäller följande:

[1] En något modifierad sanning. Vad som finns i `argv[0]` är systemberoende, och kan t ex vara programnamnet med eller utan hela sökvägen, `NULL`, eller kanske någonting annat.

```
argc == 3
argv[0] == "testprog"
argv[1] == "foo"
argv[2] == "gurka"
```

Följande program itererar över sina argument och skriver ut dem:

```
int main(int argc, char **argv)
{
  int i;

  printf("argc = %d\n",argc);

  for (i=0; i<argc; i++)
    printf("argv[%d]: \"%s\"\n",i,argv[i]);

  return 0;
}
```

3 Kontrollera antalet argument

Det är fel att försöka använda element i `argv` med index större än `argc`-1. Att göra så orsakar med stor sannolikhet segmenteringsfel. Därför är det viktigt att alltid kontrollera att `argc` innehåller rätt värde innan man försöker använda något av argumenten.

Normalt väntas ett visst antal argument av programmet, och det är lätt att kontrollera `argc` mot detta tal och skriva ut ett meddelande när det inte stämmer. Vanligt är att man skriver ut en rad med ett exempel på hur programmet borde startats, så slipper användaren gissa vid andra försöket:

```
$ mkdir
usage: mkdir directory ...
$ mkdir lab3
```

Så här kan programmet tänkas se ut:

```
int main(int argc, char *argv[])
{
   int i;

   if (argc < 2 ) {
      fprintf(stderr,"usage: mkdir directory ...\n");
      exit(1);
   }
   else {
      for (i=1; i<argc; i++)
         if (mkdir(argv[i],0775) == -1)
            perror("mkdir");
   }
   return 0;
}
```

4 Annat än strängar

Hittills har alla exempel använt strängar. Detta är naturligtvis lättast, därför att argumenten skickas till `main()` i form av strängar.

Men ofta vill man ha en annan representation, som t ex `int`. I så fall måste man konvertera från strängrepresentationen till något mer lämpligt innan man försöker använda argumenten.

Funktioner för att göra sådana konverteringar är t ex `atoi()`, `atol()` eller `atof()` (konverterar till `int`, `long` resp `float`). T ex:

```
int main(int argc, char *argv[])
{
   int m,n;

   if (argc != 3) {
      fprintf(stderr,"usage: add m n");
      exit(1);
   }
   else {
      n=atoi(argv[1]);
      m=atoi(argv[2]);
      printf("%d\n",m+n)
   }

   return 0;
}
```

5 Wildcards

Man är van att kunna skriva vissa särskilda tecken ibland i stället för en lista med enstaka filnamn, vanligtvis `*`, `?`, `[]` med mera, samt kombinationer därav. Dessa tecken kallas för "wildcards" eller ibland "jokertecken".

Det är viktigt att förstå att programmet aldrig ser dessa tecken! Det är i stället kommandotolken (bash, tcsh, osv) som ersätter dessa tecken med motsvarande fillista innan programmet startas[2]:

```
$ mv *.[ch] ~/slask
$ rm *
```

Programmen får alltså (potentiellt) långa argumentlistor, vars innehåll beror på hur shellet tolkar `*.[ch]` resp `*` i den aktuella omgivningen, dvs vilka filer de byts ut mot.

Resultatet är att vi aldrig behöver ta hänsyn till sådana specialtecken.

[2] Gäller "riktiga" operativsystem. Standardbeteendet hos t ex MS-DOS är att programmet självt får hantera wildcards, med resultat att olika program hanterar dem på lite olika sätt.