Om pekare och minneshantering i C, relaterat till operativsystem och särskilt konstruktionen `fork()-execvp()`.

Detta extra material är tänkt att dels fördjupa och förtydliga pekarbegreppet från *C* och ge ett stöd till systemprogrammeringen där vi behöver skicka en array av strängar till systemanropet `execvp()`.

Vi börjar först med att studera hur *C* organiserar lagringen av innehållet i arrayer. Vi studerar därför följande program:

```
main()
{
  int a[10] = { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 };
  int i;
  char str[20] = "Olle";

  printf("\n\na: %d &a: %d\n\n", a, &a);

  for(i=0;i<=9;i++)printf("a[%d]: %d &a[%d]: %d\n", i, a[i], i, &(a[i]));

  printf("\n");

  printf("\n\nstr: %d &str: %d\n\n", str, &str);
  for(i=0;i<=18;i++)
    if(str[i]!=0)
      printf("str[%d]: %c &str[%d]: %d\n", i, str[i], i, &(str[i]));

  printf("\n");

  printf("&i: %d\n\n", &i);
}
```

Det har följande körning:

```
a: 2686704 &a: 2686704

a[0]: 10 &a[0]: 2686704
a[1]: 20 &a[1]: 2686708
a[2]: 30 &a[2]: 2686712
a[3]: 40 &a[3]: 2686716
a[4]: 50 &a[4]: 2686720
a[5]: 60 &a[5]: 2686724
a[6]: 70 &a[6]: 2686728
a[7]: 80 &a[7]: 2686732
a[8]: 90 &a[8]: 2686736
a[9]: 100 &a[9]: 2686740

str: 2686656 &str: 2686656

str[0]: O &str[0]: 2686656
str[1]: l &str[1]: 2686657
str[2]: l &str[2]: 2686658
str[3]: e &str[3]: 2686659

&i: 2686700
```

Av detta kan vi dra slutsatsen att minnesplatser hörande till samma array placeras ut i minnesceller som ligger intill varandra, adresserna till `a` går från `&a[0]: 2686704` till `&a[9]: 2686740` och motsvarande gäller för `str`. Vi ser även att ett heltal (int) tar 4 byte att lagra så stegen mellan lagringsplatserna är 4 för integer-arrayen och 1 för char-arrayen (`str`).

Det är då viktigt att veta att så fort vi har en variabel i *C* med ett värde så har vi också en lagringsplats, i datorns minne, denna plats har en adress och det är den adressen vi refererar till med hjälp av adressoperatorn `&`.

Vi kan bara lagra heltal i datorns minne. Att vi verkar kunna lagra text är för att varje tecken har en heltalskod och därmed kan representeras som ett heltal. Om vi ska vara ännu noggrannare kan vi säga att vi egentligen bara kan lagra två tal, 0 eller 1, att vi verkar kunna lagra fler heltal beror på att dessa heltal kodas som följder av nollor och ettor, så kallade *bitar*. I exemplet ovan (som är ganska allmängiltigt) har en `int` 32 bitar som blir 4 byte och en `char` har 8 bitar som blir 1 byte. Minnespositionerna anges i byte som vi ser på sifforna. Varje byte i datorns primärminne har alltså en adress som anger dess position i minnet. (Observera att med minnesadress här avses det logiska minnet, som en process upplever det, eftersom operativsystem idag har virtuell minneshantering är den verkliga fysiska adressen en annan.)

Då är vi redo för det här med pekare: En pekare är en heltalsvariabel som kan lagra en adress till minnesposition. Den deklarareras precis som vanliga datatyper, men med en stjärna. Således är deklarationerna

```
int *ptr;
char *str;
float *fptr;
```

deklarationer av variablerna `ptr`, `str` och `fptr` som då alltså är pekare till minnesutrymmen där man kan lagra `int`, `char`, respektive `float`. Observera att dessa variabler alltså inte lagrar heltal (`int`), tecken (`char`) eller flyttal (`float`), utan adresser till minnesutrymmen där man *kan* lagra heltal, tecken respektive flyttal.

När vi programmerar med pekare får vi skriva

```
ptr = &a;
```

om vi vill lagra adressen till `int`-variabeln `a` i `ptr`. Det är helt regelvidrigt att skriva `ptr = a;` om `a` är ett heltal, vi ska ju lagra en adress till en heltalsvariabeln i `a`, inte värdet på a själv. Det är vanligt med sådana sammanblandningar och kompilatorn ger då en varning. Denna varning är då att betrakta som ett fel, den beror på en begreppssammanblandning vilket mycket troligt handlar om att programmet är felskrivet. Vi måste hitta orsaken till felet och åtgärda det om vi ska ha ett solitt fungerande program.

Dessa saker är troligtvis så långt som ni kommit i kursen i grundläggande programmering. Nu ska fördjupa det här och se på den speciella tillämpningen då vi ska omvandla en sträng till en array av strängar och organisera den så att vi kan skicka den till `execvp()`.

UNIX-kommandon som ges vid en kommandoprompt ges som en inknappad textsträng där de olika argumenten separeras med mellanslag, till exempel:

```
$ ls -l /home
```

ger alla filer i under katalogen `/home`. Detta kommando består av delsträngarna "`ls`", "`-l`" och "`/home`" sammanslagna till en sträng men separerade av mellanslag. (Dollartecknet "`$`" innan symboliserar prompten som bara betyder att skalet är redo att ta emot ett kommando.)

Ett problem i laboration 2 är att skapa ett eget skal som alltså tar en kommandosträng från användaren, delar upp den i kommandoradsargument och skickar till systemkommandot `execvp()`. Därför uppstår problemet att bryta upp en sträng innehållandes ett kommando som ovan och lägga in de olika delsträngarna i en array av arrayer. Vi ska se på en variant av hur vi kan lösa detta här. Nedanstående program är skrivet som en utvidgning av `fork-exec.c` från kapitel 3 i *Advanced Linux Programming*. Det ursprungliga programmet i ALP hade inte förmågan att hantera allmänna argumentsträngar utan kunde bara köras med en hårdkodad variant av strängarray, `char *arg_list[]`, programmet nedan råder bot på detta problem genom att införa en allmän parameterarray, `char ** gen_arg_list` som deklararas straxt efter `arg_list`. Det är vår målsättning att går igenom programmet och förklara rad för rad så att ni kan tillämpa detta.

```
 1 /**************************************************************************
 2 * Code listing from "Advanced Linux Programming," by CodeSourcery LLC  *
 3 * Copyright (C) 2001 by New Riders Publishing                           *
 4 * See COPYRIGHT for license information.                                *
 5 **************************************************************************/

 6 /**************************************************************************
 7 * Expanded by Johnny Panrike to allow for a general command string,     *
 8 * (C) 2011                                                              *
 9 **************************************************************************/

10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <sys/types.h>
13 #include <unistd.h>
14 #include <sys/wait.h>
15 #include <string.h>

16 /* Spawn a child process running a new program.  PROGRAM is the name
17    of the program to run; the path will be searched for this program.
18    ARG_LIST is a NULL-terminated list of character strings to be
19    passed as the program's argument list.  Returns the process id of
20    the spawned process.  */

21 int spawn (char* program, char** arg_list)
22 {
23   pid_t child_pid;

24   /* Duplicate this process.  */
25   child_pid = fork ();
26   if (child_pid != 0)
27     /* This is the parent process.  */
28     return child_pid;
29   else {
30     /* Now execute PROGRAM, searching for it in the path.  */
31     execvp (program, arg_list);
32     /* The execvp function returns only if an error occurs.  */
33     fprintf (stderr, "an error occurred in execvp\n");
34     abort ();
35   }
36 }
```

```
37 int main ()
38 {
39   /* The argument list to pass to the "ls" command.  */
40   char* arg_list[] = {
41     "ls",     /* argv[0], the name of the program.  */
42     "-l",
43     "/",
44     NULL      /* The argument list must end with a NULL.  */
45   };

46   char ** gen_arg_list;

47   char command_str[256];
48   int i, current_arg, length, num_spaces = 0, num_args;

49   /* Spawn a child process running the "ls" command.  Ignore the
50      returned child process id.  */
51   spawn (arg_list[0], arg_list);
52   wait(0);

53   printf("Command: ");
54   gets(command_str);
55   length = strlen(command_str);
56   i=0;
57   while(i++<length)if(command_str[i]==' '){command_str[i]='\0';num_spaces++;}

58   num_args = num_spaces + 1;

59   //printf("Num args: %d.\n", num_args);

60   gen_arg_list = (char **)malloc((num_args + 1)*sizeof(char **));

61   gen_arg_list[num_args+1-1]=NULL;
62
63   gen_arg_list[0]=&(command_str[0]);
64   i=0;
65   current_arg=1;
66   while(i++<length-1 /*suppress last position*/ )
67     if(command_str[i]=='\0')
68       gen_arg_list[current_arg++]=&(command_str[i+1]);

69   //for(i=0;i<num_args+1;i++)printf("Arg %d: %s.\n", i, gen_arg_list[i]);

70   spawn(gen_arg_list[0],gen_arg_list);
71   wait(0);

72   free(gen_arg_list);

73   printf ("done with main program\n");

74   return 0;
75 }
```

Rad 1-36 innehåller inget principiellt nytt utan är bara en upprepning av det från ALP. Ra 40-45 innehåller den hårdkodade varianten av parameterlista och rad 46 innehåller en deklaration av en generell array av arrayer, "`char ** gen_arg_list`". Observera att vi har två stjärnor, en array är egentligen en pekare och vi vill här arbeta med arrayer av arrayer, därmed får vi två stjärnor i deklararationen.

Det finns en viktig skillnad mellan deklarationerna av `gen_arg_list` och `arg_list`, de är båda av den generella typen `char **` men deklarationen av `arg_list` reserverar också minnesutrymmena som pekas på av pekarna i `arg_list`. Deklararationen "`char ** gen_arg_list`" reserverar däremot inget minnesutrymme alls och det är inte meningsfullt att göra det för vi vet inte hur många platser vi ska ha. Det som blir dynamiskt här är alltså att programmet anpassar storleken av `gen_arg_list` efter hur mycket som behövs. Det är därför den heter `gen_arg_list`, "*General Argument List*".

Vi går vidare och förklarar flera rader, rad 47 och 48 innehåller några olika hjälpvariabler, strängen `command_str` ska lagra den kommandosträng som användaren ska mata in. Rad 49-52 innehåller bara en testkörning av `spawn()` på den hårdkodade `arg_list`. Det är rad 53 till 69 som är extra intressant för oss, det är detta moment som många haft problem med, vi ska därför förklara det i detalj.

Det som sker i rad 53-57 är en inmatning av kommandosträngen `command_str`, vi räknar antalet mellanslag och byter ut dem mot nollor, alltså strängavslutningar. Tanken här är att bygga upp arrayen `gen_arg_list` som en array av pekare som indikerar startpositioner på strängar och därvid använda att strängen `command_str` ju redan har dessa tecken lagrade.

Vi inför nu även variabeln `num_args` som räknar antalet argument, det blir helt enkelt bara antal mellanslag + 1. Det kan tyckas vara onödigt att införa en sådan variabel, men jag inför den ändå för att vårt program ska bli mer lättläsligt.

På rad 60 vet vi hur många argumenten är, det är stycken, då kan vi skapa lagringsutrymmen för alla pekare som vi behöver. Det gör vi med systemanropet `malloc()` som ger oss de minnespositioner vi behöver. Vi anänder `sizeof`-operatorn för att veta exakt storlek på det minne vi behöver. Observera att vi också reserverar ytterligare ett minnesutrymme för den sista pekaren i som måste vara NULL. Därför skriver vi (`num_args + 1`) i anropet till `malloc()`. När rad 60 är körd så innehåller arrayen `gen_arg_list` precis så många pekare som behövs för att kunna peka ut alla kommandoradsargument i arrayen `command_str`.

Nästa steg är att fylla `gen_arg_list` med de värden den ska ha, vi har hittills bara skapat lagringsplatserna som den ska innehålla. Den sista platsen sätts till NULL på rad 61 och raderna 63-68 lägger in alla adresser till alla förstatecken till delsträngarna i kommandosträngen in till pekarvariablerna i `gen_arg_list`. Efter denna loop är klar kan vi anropa `spawn()` som förut.

På ra 71 inväntar vi att barnprocessen som skapades i `spawn()` ska avslutas och sedan gör vi något väldigt viktigt: vi frigör det minne som vi allokerade för `gen_arg_list`. Visserligen avslutas programmet, men jag väljer ändå att illustrera detta, vi måste alltid frigöra det minne som vi allokerar dynamiskt, *C* gör det inte åt oss.

Nu är det tänkt att ni ska gå igenom programkoden, skriva in den och provköra den och använda någon liknande teknik för att fullborda shell-uppgiften. Jag har nu visat hur man knäcker just stränghanteringen i denna uppgift, så då är det rimligt att kräva att ni hanterar avslut av ert ska på ett bra sätt, alltså att man kan skriva `exit` och då avslutar skalet, vidare också att man kan ha en prompt som återkommer mellan kommandona som man ger.

Lycka till!