Erlends Guide Til C

Intro

Hei. Det viktige man må lære for å skjønne C er ikke det som er likt, men det som er annerledes. Likevel kommer jeg til å prøve å sammenligne ting til andre programmeringsspråk, for enkelhetens skyld.

Typer

I C er alle "vanlige typer"primitive typer. Det vil si at vi kun har

```
int
```

og ikke f.eks. Java sin

```
Integer
```

Der dette har aller mest å si er strings. I C er en string *NESTEN* kun et array av chars. Det kan skrives på formen

```
char string[1024] = "String ellerno"
```

eller

```
char *string = "String ellerno"
```

Det er forskjell mellom disse to, den første er kun et array av chars på formen:

```
{'S', 't', 'r', 'i', ...(osv), 'n', 'o', '\0'}
```

imens `char *string` er en pointer(mer info i delen om pointers) til en "string literal". Det viktigste du trenger å vite nå er at disse ikke skal redigeres, og helst kun brukes når du har en string som er permanent, f.eks. en path til en fil.

```
int // vanlig int, minst 16 bits
float // desimal, stort sett 32 bits
long // større int
long long // enda større int
char // char,
```

Output

For å gjøre output i C, bruker man som regel `printf`, som tar formatet for strengen og eventuelle variabler. Å printe en streng som:

```
char string[1024] = "En string!";
printf(string);
```

er sett på som dårlig praksis, siden man ikke spesifiserer format. En bedre måte:

```
char string[1024] = "En string!";
printf("%s\n", string);
```

I dette printf-kallet, er det vi faktisk printer ut, det som er innenfor "%s", men %s blir byttet ut med variabelen vi gir etter. Her er en liste av de mest relevante formatene:

```
%d // desimaltall
%f // floattall
%s // streng
%x // hextall
%o // oktaltall
```

Enda et par eksempeler:

```
int num = 5;
printf("Tallet er: %d!\n", num);
```

som printer ut `Tallet er: 5` og:

```
int num1 = 5;
int num2 = 8;

for(int i = 0; i < 3; i++) {
    printf("%d%6d", num1, num2);
}
```

Det siste eksempelet printer ut dette:

```
5     8
5     8
5     8
```

siden vi satte 6 foran d, som betyr at den skal ha plass for seks siffer.

Man kan også bruke funksjonen `putchar`, som tar en verdi og skriver den ut som ASCII.

printf og putchar skriver til stdout, som er den vanligste streamen for output. Man kan også bruke fprintf for å

Input

For å ta inn input fra bruker, anbefaler jeg `fgets`. Denne funksjonen tar tre argumenter: bufferet man skal ha dataen i, størrelsen på bufferet og en stream.

Et eksempel fra ett program jeg har skrevet:

```
    while (1) {
        printf("%s %s  $  ", state.user, state.curr_dir);
        fgets(buf, sizeof(buf), stdin);
        if (strcmp(buf, up) != 0) {
            argument_parser(buf, &state);
        } else {
            printf("%s", state.tail->command);
```

```
        }
    }
```

Her ser vi at fgets tar fra stdin, som er den mest vanlige streamen å bruke. For å få størrelsen til buf, bruker vi sizeof(). Den tar en variablen eller array og gir størrelsen i bytes, så en array med 10 32-bit ints vil være 40 bytes langt.

Headers

Stort sett er det første man ser i en C-fil disse ordere:

```
#include <stdio.h>
```

Dette er en headerfil som er inkludert på systemet ditt, og har en del funksjoner for standard I/O (derav navnet). `printf`, `fprintf` og `fgets` er alle i denne headeren. En header er bare en fil man legger over siden egen kode, som inneholder funksjoner og structs(objekter, er en egen del om dem lenger ned.).

Andre nyttige headers:

```
#include <stdlib.h> //malloc(), free() osv
#include <stdint.h> // uint32_t, uint8_t og andre int-typer
#include <errno.h> // errorhåndtering av syscalls sine error returns
#include <string.h> // memcpy(), memset(), strncpy() og andre string/minne funk
#include <stdbool.h> // bool-typen, som kan settes som true eller false
#include <time.h> // tidshåndtering
```

Pointers

En pointer er en peker til en adresse i minne. Her er et eksempelprogram med en `int` pointer:

```
#include <stdio.h>

int main(void) {
    int count = 5;
    int *pointer = &count;
    (*pointer)++;
    (*pointer)++;
    printf("%d", *pointer);
    return 0;
}
```

Forlaring av de forskjellige nye variablene:

```
int count = 5 //variabel
int *pointer = &count /* initialiserer pointer til å være en peker til
addressen av count */
*pointer /* dereferencer pointer for å gi verdien som ligger på adressen den
```

```
peker på */
&count // adressen til count, kan sendes inn i funksjon som en pointer
```

Det er stor forskjell på når vi skal initialisere pointeren eller aksessere den, men dessverre bruker man samme symbol til begge. Så om du ser typen først, er det en initialisering, og om du kun ser * og så variabelen, så aksesserer man det variabelen peker på.

```
printf("%d", *pointer)
// eller
*pointer += 5
```

Først har vi den primitive `int`, som vi er kjent med. Så tar vi adressen til den med `*` og initialiserer en pointer som heter `pointer`. Dette er en variabel som bare peker til minneadressen til `count`. Denne kan nå sendes inn til f.eks. en funksjon sånn som dette:

```
#include <stdio.h>

void multiply_val_in_adress(int *p, int mult) {
    *p *= mult;
}

int main(void) {
    int count = 5;
    int mult = 3;
    int *pointer = &count;
    printf("Val in pointer: %d\n", *pointer);
    printf("Val in variable: %d\n", count);
    multiply_val_in_adress(pointer, mult);
    printf("New val in pointer: %d\n", *pointer);
    printf("New val in variable: %d\n", count);
}
```

Her ser du at vi har en funksjon som returnerer void, tar en peker til en int og en int den skal gange med inten som blir pekt til med. Jeg har med både pointer og variablen, så man skal se at de egentlig er det samme. Vi har endret det som ligger på minneadressen vi først ga til count. Dette blir output:

```
Val in pointer: 5
Val in variable: 5
New val in pointer: 15
New val in variable: 15
```

Vi kan titte nærmere på funksjonssignaturen til funksjonen i forrige kodeeksempel. `int *p` betyr at funksjonen tar inn en pointer til en `int`, dvs. en variabel, som er en adresse, hvor det ligger en `int`.

```
void multiply_val_in_adress(int *p, int mult) {
    *p *= mult;
```

```
}
```

Man kan også bruke `&` operatoren vi så tidligere, som henter adressen til variabelen og skrive dette om man bare vil endre verdien uten å lage en pointer:

```
func_that_takes_a_pointer(&count, value, string);
```

Structs

En `struct` er måten man lager et "objekt" i C. Det er en kombinasjon av typer, slik som dette:

```
struct coordinates {
    int x;
    int y;
}
```

Etter å initialisere en struct, kan man aksessere medlemene av structen på ulike måter, avhengig om det er en peker til en struct, eller et faktisk structobjekt:

```
#include <stdio.h>

struct coordinates {
    int x;
    int y;
};

void change_coords(struct coordinates *coords, int new_x, int new_y) {
    coords->x = new_x;
    coords->y = new_y;
}

int main(void) {
    struct coordinates coords = { 1, 1 };
    printf("x: %d, y: %d\n", coords.x, coords.y);
    change_coords(&coords, 3, 4);
    printf("new x: %d, new y: %d\n", coords.x, coords.y);
    return 0;
}
```

Dette returnerer:

```
x: 1, y: 1
new x: 3, new y: 4
```

Om man behandler et faktisk struct object, aksesserer man medlemmet direkte, ved å bruke '.', som man kan se i print statementet. Funksjonen change_coords tar adressen til structen, og vil da endre verdiene i den direkte. Om man sender en struct inn i en funksjon med kun et structobjekt, vil du kun få en kopi av de

verdiene som er i structen nå, og alle eventuelle endringer vil kun gjelde om du returnerer structen.

Typedef

Å måtte skrive

```
struct coordinates coords = { 0 };
change_coords(struct coordinates *coords, int x, int y);
```

hver gang man skal initialisere eller legge en struct er slitsomt. Det vi kan gjøre i stedet for er å typedeffe en struct:

```
#include <stdio.h>

typedef struct weapon {
    int durability;
    int damage;
    float weight;
} weapon;

typedef struct armor {
    int durability;
    int defense;
    float weight;
} armor;

typedef struct player {
    int hp;
    int strength;
    weapon *sword;
    armor *chestplate;
} player;

void new_weapon(player *hero, weapon *greatsword) {
    hero->sword = greatsword;
}

int main(void) {
    player hero = { 10, 2, 0, 0 };
    weapon greatsword = { 100, 5, 3.4 };
    new_weapon(&hero, &greatsword);
    printf("Our hero's new weapon has %d damage!\n", hero.sword->damage);
    return 0;
}
```

Her har vi laget en enkel struct for en spillerkarakter i et videospill, samt to structs for våpen og rustning.

Filer

En fil i C er en pointer til et filobjekt. fopen()

```
FILE *file = fopen("file.txt", "w+");
```

`fopen(const char *path, const char *mode)` returnerer en filpointer som vi kan bruke for å interagere med filen, blant annet lese fra den og skrive til den. "w+" er modusen filen er i. "w" betyr at vi tillater å skrive til filen, og + betyr at vi tillater "det andre", i dette tilfellet lesing. Om vi kun vil ha lesing, kan vi skrive "r". Lesing og skriving kan også skrives som "r+".

`fread(void *ptr, size_t size, size_t num_items, FILE *stream` tar num_items antall items, med størrelse size, fra stream til ptr.

`fwrite(const void *ptr, size_t size, size_t num_items, FILE *stream` gjør det samme som fread(), bare at den gjør det *fra* ptr til stream.

`fclose(FILE *stream)` lukker filen. Det er veldig viktig å lukke en fil etter man har åpnet den, når man har gjort det man skal med den, for å hindre udefinert oppførsel.

`fcloseall(void)` lukker alle streams som er åpne. Kan være nyttig om man jobber med veldig mange filer.

Minnehåndtering

Når man initialiserer en variabel vanlig, så allokerer man den på *stack*-en, slikt som dette:

```
int array[1024] = { 0 }
```

Pro tip: `= { 0 }` null-initialiserer hele minnet du initialiserer, så du kan være sikker på at ingen verdier er noe annet enn null. Dette gjelder også for structs, hvor man kan ha forskjellige typer verdier.

Stacken er fast blokk med minne man får ved hvert program. Hvor stor stacken er, avhenger av arkitektur og andre ting som ikke er viktig her. Det viktigste du trenger å vite er at stort sett burde du gi litt ekstra minne om du vet ca. hvor mye du skal ha. Om du *ikke* vet hvor mye minne du skal, må du allokere dette minnet manuelt med malloc eller calloc.

Malloc

`malloc`(*size_t size*) er en funksjon i headeren stdlib.h. Den gjør dynamisk minneallokering, altså tar allokerer minne fra heapen. Dette er det samme objekter i Java gjør, bare at de gjør det for deg, og når objektet ikke brukes lengre, frigjør garbage collecteren minne for deg. I C må du selv bestemme når du skal allokere minnet, og alt minne du har allokert dynamisk må du frigjøre selv. Et lett eksempel er at du tar brukerinput, og skal lage et array

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    char buf[1024];

    /* fgets tar en buffer og legger til input, her stdin, inn i bufferet */
    fgets(buf, sizeof(buf), stdin);
    printf("%s", buf);

    /* gjør om brukerinputet fra en string til en int */
    int size = atoi(buf);
    printf("%d", size);

    /* allokerer minne til en int array med størrelsen bruker ba om */
    int *array = malloc(size * sizeof(int));
    /* gjør noe med arrayet */

    /* frigjør alltid allokert minne */
    free(array);

    return 0;
}
```

`calloc`(*size_t count, size_t size*) gjør det same som malloc, men du gir den et spesifikt antall elementer, samt null-initialiserer alt minne. Et eksempel på et calloc-kall er:

```c
int size = 1024;
char array = calloc(size, sizeof(char));
```

Dette gir et array med 1024 `char` hvor alle har verdien 0.

Free

Om du har allokert minne, *må* du frigjøre det også. Dette gjøres med funksjonen `free(void *ptr*)` som tar en pointer av vilkårlig type. Etter dette kan du ikke bruke denne pointeren lenger: Dangling pointer(wikipedia-lenke).

TODO:

Pointer aritmetikk + pointers = arrays Yrjar tips