

Inga globala variabler och statiska arrayer

Tobias Wrigstad

*Hur man refaktorerar
bort dem i sitt program*



Globala variabler är dåligt

- Låt `int g;` vara en global variabel i nedanstående (icke-fullständiga) program — vad skrivs ut i printf-satsen?


```
int g; // global

int main(void)
{
    g = 42;

    foo();

    printf("%d\n", g);

    return 0;
}
```



Globala variabler är dåligt

- Låt `int g`; vara en global variabel i nedanstående (icke-fullständiga) program — vad skrivs ut i printf-satsen?

```
int g; // global

int main(void)
{
    g = 42;

    foo();

    printf("%d\n", g);

    return 0;
}
```



VI HAR INGEN ANING!

Globala variabler är dåligt

- Låt `int g`; vara en global variabel i nedanstående (icke-fullständiga) program — vad skrivs ut i printf-satsen?


```
int g; // global

int main(void)
{
    g = 42;

    foo();

    printf("%d\n", g);

    return 0;
}
```



VI HAR INGEN ANING!

Eller — mer korrekt, det beror på vad `foo()` gör, och alla andra funktioner som `foo()` anropar, direkt eller indirekt

Globala variabler är dåligt

- Globala variabler kan skrivas och läsas av alla funktioner som kan se dem
- Konsekvenser

Det är svårt att resonera om deras värden

Varje uppdatering av en global variabel får stora konsekvenser — påverkar alla som kan se den/använder den

```
int g; // global

int main(void)
{
    g = 42;

    foo();

    printf("%d\n", g);

    return 0;
}
```

Förbättra datatyperna i ditt program

- Antag att vi har skrivit ett program med en databas i en global variabel
- Två problem:
 - DB är en global variabel
 - DB.goods är en statisk array
- Vi attackerar dem i denna ordning

```
typedef struct db db_t;

struct db
{
    goods_t goods[1024];
    int total;
};

db_t DB; // global
```

Recept för att bli av med en global variabel g av typen T

Steg 1 — flytta deklARATIONEN av $T\ g$ från global scope in i `main`

Kompilerera om. Kompileringsfel på alla ställen där variabeln används.

Steg 2 — för varje funktion f som använder g , lägg till $T\ *g$ som en parameter

OBS! Notera $T\ *g$ och inte $T\ g$. **Pekarsemantik.**

Nu måste du också ändra g till $*g$ i koden, t.ex. $g = 27$ blir $*g = 27$.

Kompilerera om. Kompileringsfel på alla ställen funktionerna anropas — för få argument (g skickas ju inte med). Ändra överallt utom i `main`.

Steg 3 — lägg till g som argument till alla funktionsanrop där g nu saknas

Kompilerera om. Alla kompileringsfel som är kvar är i `main`. Lös detta genom att skicka `in &g` som argument till funktionerna. **KLART.**

Se nästa bild för om T är en pekartyp eller g endast läses

Specialfall i föregående recept

- Om T är en pekartyp behöver du inte den extra pekarindirektionen i steg 2, och inte heller använda &g i steg 3 — **såvida inte**

Du tilldelar till din globala variabel — alltså du pekar om den

- Om g endast läses och inte skrivs behöver du inte skicka in g som en pekare i steg 2

Exempel

```
void foo()
{
    g = 24;
}

int g; // global

int main(void)
{
    g = 42;

    foo();

    printf("%d\n", g);

    return 0;
}
```

Exempel [steg 1]

```
void foo()
{
    g = 24;
}

int main(void)
{
    int g; // local

    g = 42;

    foo();

    printf("%d\n", g);

    return 0;
}
```

```
globals.c: In function 'foo':
globals.c:5:3: error: 'g' undeclared (first use in this function)
    g = 24;
    ^
```

Exempel [steg 2]

```
void foo(int *g)
{
    *g = 24;
}

int main(void)
{
    int g; // local

    g = 42;

    foo();

    printf("%d\n", g);

    return 0;
}
```

```
globals.c: In function 'main':
globals.c:14:3: error: too few arguments to function 'foo'
    foo();
    ^
globals.c:3:6: note: declared here
void foo(int *g)
    ^
```

Exempel [steg 3]

```
void foo(int *g)
{
    *g = 24;
}

int main(void)
{
    int g; // local

    g = 42;

    foo(&g);

    printf("%d\n", g);

    return 0;
}
```

Inga kompileringsfel

Programmen sida vid sida

```
void foo()
{
    g = 24;
}

int g; // global

int main(void)
{
    g = 42;

    foo();

    printf("%d\n", g);

    return 0;
}
```

```
void foo(int *g)
{
    *g = 24;
}

int main(void)
{
    int g; // local

    g = 42;

    foo(&g);

    printf("%d\n", g);

    return 0;
}
```

Ett problem kvarstår: DB.goods är statisk array

- Två problem:

DB.goods är $1024 * \text{sizeof}(\text{goods_t})$ — tar upp en stor del av stacken

Vad händer om 1024 är för lite?

- Lösning till det **första** problemet:

Flytta arrayen till heapen

- Lösning till det **både det första och det andra** problemet: **(Bättre!)**

Använd en dynamisk datastruktur istället, t.ex. en länkad lista

Recept för att bli av med en statisk array i posten `arr` i strukten `S`

Steg 1 — ändra typen på `arr` från `T[n]` till `T *` i `S`

Kompilera om. Du får inga kompileringsfel eftersom pekare och arrayer i stora stycken är samma sak.

Steg 2 — för varje plats `p` som skapar en ny `S`, allokerar `arr` med hjälp av `calloc`

`T arr[n]` ursprungligen ger `calloc(n, sizeof(T))` för att få en pekare till en array med `n` element av rätt storlek för `T`. Ex.: `mystrukt.arr = calloc(n, sizeof(T));`

Kompilera om. Fortfarande inga kompileringsfel.

Steg 3 — där `S`-strukturer förstörs, frigör minnet i `arr` explicit med `free`

Ex.: `free(mystrukt->arr)`. **KLART.**

Det svåra är i steg 3 — att avgöra var strukturerna förstörs.

Tips för ren kod som är lätt att jobba med

- För varje strukt `S` i ditt program, ha alltid en funktion `S_new` som skapar element av den strukten, och en `S_free` som frigör dem.

Då medför steg 2 i föregående recept endast två ändringar i programmet, en i `S_new` och en i `S_free`

- Anropa `S_new` och `S_free` överallt där struktarna behövs
- Använd valgrind för att hitta där du glömmer att anropa free

Notera

- **char** name[30] är också en statisk array som i ärlighetens namn (förmodligen) borde tas bort och ersättas med en **char ***
- Det är viktigast att fokusera på

Stora arrayer vars element är stora

Arrayer vars längd är godtyckliga och riskerar att inte passa

Arrayer som är slösaktiga (t.ex. 1024 element men använder normalt 5)

- Även om det är bättre att **allt är bra**TM, så är det viktigaste att man kan demonstrera förståelse för principen!

...men se även nästa bild

Strängar på heapen

- Det är krångligt att programmera med statiska arrayer — framför allt när man hanterar strängar. Det blir mycket `strcpy` eller `strncpy` överallt.
- Om man har en inläsningsfunktion som läser in strängar på heapen slipper man:

```
strcpy(mystrukt->arr, my_new_str);
```

och kan istället göra

```
mystrukt->arr = my_new_str;
```

- **OBS!** Om `mystrukt->arr` redan pekar på en sträng så borde man förmodligen göra `free` på den, såvida den inte används någon annanstans i programmet. Alltså:

```
if (mystrukt->arr) free(mystrukt->arr);
```

```
mystrukt->arr = my_new_str;
```

Recept för att göra om db->goods till ett träd (eller liknande) istället för en array

Steg 1 — skapa funktioner för att stoppa in och ta ut element ur db->goods

Du kan skapa dem från din kod enkelt, t.ex. `db->goods[db->total++] = g` ändras till ett anrop till `add_good(db, g)` som vars kod är `db->goods[db->total++] = g`.

Du är klar med detta steg när de enda funktioner som läser eller skriver direkt till db->goods är de funktioner som du just har skrivit! Gör detta funktion för funktion och verifiera att programmet fungerar hela tiden.

Steg 2 — ändra db->goods från T^* till en lämplig dynamisk datastruktur (t.ex. ett träd)

Förmodligen behöver du inte någon db->total längre.

Ändra den funktion som skapar databasen så att den initierar ett tomt träd. Gör motsvarande för funktionen som förstör databasen.

(fortsättning på nästa sida)

Recept för att göra om db->goods till ett träd (eller liknande) istället för en array

Steg 2 — (fortsättning)

Gå igenom alla funktioner som du skrev ovan och ändra dem så att de använder trädets `add_good(db, g)` blir t.ex. kanske `tree_insert(db->goods, g)`;

Du kommer eventuellt att behöva ändra vissa funktioner så att de tar in andra parametrar. Låt säga att du hade en `get_good(db, index)` — den är inte vettig längre eftersom det inte finns `index` i trädet. Kanske är det rätt att söka ut varan efter dessa namn:

```
good_t *get_good(db_t *db, char *name)
{
    return tree_find(db->goods, name);
}
```

Detta kan vara knepigt att göra stegvis — börja med funktioner som skapar, sedan de som tar bort, och sist de som ändrar.