

Föreläsning 10

Tobias Wrigstad

Preprocessorn



Nyckelordet Const



Nyckelordet **const**

Används för att definiera ett konstant värde

```
const int v1 = 42;    // v1 innehåller alltid 42
```

```
int const v2 = 42;    // v2 innehåller alltid 42
```

```
int const *p1 = &v1; // p1 pekar på ett konstant värde
```

```
int const *const p2 = &v; // konstant pekare till konstant värde
```

```
person_t const *p3;
```

Ytterligare exempel

const avser alltid det som står till vänster, precis som *

`int const` — ett konstant heltal

`int const *` — en pekare till ett konstant heltal

`int * const` — en konstant pekare till ett heltal

`int * const *` — en pekare till en konstant pekare till ett heltal

`int const * *` — en pekare till en pekare till ett konstant heltal

`int const * const *` — en pekare till en konstant pekare till ett konstant heltal

Konstanta strängar

- `char *foo = "Hello, world";` är i regel en konstant (`char const *`)
- Kompilatorn kan placera tecknen i ROM
- Försök att ändra dem kompilerar, men leder till hårda kraschar
- *"passing char ** to parameter of type const char ** discards qualifiers in nested pointer types"*

Kräver en explicit typomvandling (`char const **`)

Const eller inte const?

- Använd alltid **const** om det går!
- Man kan ”typomvandla bort” const

```
(int *) p; // när p : int * const
```

- Gör endast detta när koden som använder resultatet kunde ha annoteras med const

Annars kan saker gå sönder (varför?)

- Helt OK att typomvandla åt andra hållet, om man inte tänker ändra sitt data parallellt

```
(char const *) s; // när s är char *
```

Const är grund (shallow)

```
struct person
{
    char *name;
    int age;
};

typedef struct person person_t;

person_t const *p = ...;
p->age = 42; // kompileringsfel!
strcpy(p->name, "Ivor"); // OK
```

Preprocessor



CPP — The C PreProcessor

- Textersättning som sker innan kompilatorn
- Vi har sett några exempel tidigare:

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



Inkludering av annan fil

```
#ifndef symbol  
#define symbol  
...
```



If-sats som körs vid kompileringen

```
#endif
```



```
#define INT_MAX 2147483647
```



Definition av konstant (eg. makro)

- CPP — körs innan kompilatorn automatiskt (körs sällan eller aldrig enskilt)

Preprocessor”operatorer”

- Makron som spänner över flera rader \
- Göra om kod till strängar #
- Konkatenera uttryck ##
- Man kan även stänga av eller sätta flaggor vid kompilering

- Otroligt primitivt

Man kan göra **fantastiska** saker med dem

T.ex. definiera hela datastrukturer, e.g. Define_List(int)

```

#define DECLARE_LIST(name, elem) \
    typedef struct name##_t name##_t; \
    typedef bool (*name##_cmp_fn)(elem* a, elem* b); \
    typedef elem* (*name##_map_fn)(elem* a, void* arg); \
    typedef void (*name##_free_fn)(elem* a); \
    name##_t* name##_pop(name##_t* list, elem** data); \
    name##_t* name##_push(name##_t* list, elem* data); \
    name##_t* name##_append(name##_t* list, elem* data); \
    name##_t* name##_next(name##_t* list); \
    name##_t* name##_index(name##_t* list, ssize_t index); \
    elem* name##_data(name##_t* list); \
    elem* name##_find(name##_t* list, elem* data); \
    ssize_t name##_findindex(name##_t* list, elem* data); \
    bool name##_subset(name##_t* a, name##_t* b); \
    bool name##_equals(name##_t* a, name##_t* b); \
    name##_t* name##_map(name##_t* list, name##_map_fn f, void* arg); \
    name##_t* name##_reverse(name##_t* list); \
    size_t name##_length(name##_t* list); \
    void name##_free(name##_t* list); \

```



Preprocessormakron

- För robusthet — sätt parenteser runt varje arguments användande, och hela makrot

```
#define Max(a,b) ( (a) < (b) ? (b) : (a) )
```

- För längre makron, använd följande form

```
do { ... } while (0);
```

Robusta preprocessormakron?

- Hitta felen!

```
#define sqr_a(x) (x)*(x)
```

```
#define sqr_b(x) (x)*(x)
```

```
#define sqr_c(x) x*x
```

```
int a = 9;
```

```
int b = sqr_a(++a);
```

```
int c = sqr_b(a)+10;
```

```
int b = sqr_c(a-10);
```

Preprocessormakron

- Textuell ersättning, fungerar inte ”som C”:

```
#define Max(a,b) a < b ? b : a
```

```
Max(x + y, z);           // x + y < z ? z : x + y
```

```
Max(++x, z);             // ++x < z ? z : ++x
```

Robusta preprocessormakron?

- Byt plats på två värden utan en tredje "slaskvariabel", mha xor

```
#define Swap(a,b) { \
    a ^= b; \
    b ^= a; \    // Illustrerar 5-radersmakro
    a ^= b; \
}
```

- Funkar bra

```
if (x < y)
    Swap(x, y);
```

Robusta preprocessormakron?

- Byt plats på två värden utan en tredje ”slaskvariabel”, mha xor

```
#define Swap(a,b) a ^= b; b ^= a; a ^= b;
```

- Funkar inte så bra (varför?!)

```
if (x < y)
    Swap(x, y);
```

- Kompilerar inte ens

```
if (x < y)
    Swap(x, y);
else
    Swap(x, z);
```


Printline debugging

- I lämplig headerfil

```
#define Debug
```

- Överallt i resten av koden

```
#ifdef Debug  
printf("..."); // Spårutskrift  
#endif
```

- Detta används ofta också för andra villkor i kod, t.ex. plattform, OS, etc.

```

#ifndef __min_assert_h__
#define __min_assert_h__

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

#define __minute__assert(kind, msg, b) \
    if (b) { printf("Assertion failed: %s(%s), file %s, line %d\n", \
        kind, msg, __FILE__, __LINE__); exit(EXIT_FAILURE); } \

#define Assert_true(arg) \
    do { __minute__assert("assertTrue", #arg, !(arg)); } while (0); \
#define Assert_false(arg) \
    do { __minute__assert("assertFalse", #arg, (arg)); } while (0); \

#endif

```

massert.h



```

#ifndef __min_assert_h__
#define __min_assert_h__

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

#define __minute__assert(kind, msg, b)
    if (b) { printf("Assertion failed: %s(%s), file %s, line %d\n",
        kind, msg, __FILE__, __LINE__); exit(EXIT_FAILURE); } \

#define assertTrue(arg)
    do { __minute__assert("assertTrue", #arg, !(arg)); } while (0); \
#define assertFalse(arg)
    do { __minute__assert("assertFalse", #arg, (arg)); } while (0); \

#endif

```

massert.h



```
#include "massert.h"

int main(int argc, char *argv[])
{
    Assert_true(argc > 1);
    return 0;
}
```

test.c

```
$ gcc -Wall -g -o test test.c
```

```
$ ./test
```

```
Assertion failed: Assert_true(argc > 1), file test.c, line 5
```



Iteratorer



Kompilering & Länkning



Gå igenom

gcc

gcc -o

gcc -g

gcc -Wall

gcc .o .o -o foo

nm bar.o

Aliasering



Exemplifiera

inläsning i en buffer flera gånger så alla ord i trädet blir samma

Modularisering



Coupling och cohesion

- En moduls cohesion är ett mått på utsträckningen i vilken dess åtaganden tillsammans ”ger mening” — ju högre desto bättre
 - Låg cohesion betyder att en modul har väldigt många olika åtaganden
 - Anti-pattern: ”god classes” (god modules)
 - En modul med bara en funktion som utför en sak har maximal cohesion
- Coupling mellan moduler är ett mått på deras ömsesidiga beroende av varandra — lägre är bättre
 - Hög coupling betyder att det är svårt att isolera förändringar
- Designprincip: öka cohesion och minska coupling!

De värsta sorternas coupling och cohesion

- Coincidental Cohesion

Modulens beståndsdelar är helt orelaterade

- Content/Pathological Coupling

När en metod använder eller förändrar data inuti en annan modul direkt, utan att gå via dess funktioner

```
commodity_t book = {};  
book.cost = -12,50;  
int cost_of_book = book.cost;
```

```
commodity_t book = mk_book(...);  
set_cost_of_book(book, 12,50);
```

mylogging_system.c

```
void searchMessages(char* msg) { ... }

File openFile(char* fileName) { ... }

char *readFromFile(File file, int size) { ... }

void closeLogFile() { ... }

void flushLogs(char* msg) { ... }

int writeToFile(File file, char *bytes) { ... }

void logMessage(char* msg) { ... }

void deleteMessage(char* msg) { ... }

void openLogFile() { ... }

void setLogFileName(char *fileName) { ... }
```



loggning

filhantering

mylogging_system.c

```
void searchMessages(char* msg) { ... }
```

```
File openFile(char* fileName) { ... }
```

```
char *readFromFile(File file, int size) { ... }
```

```
void closeLogFile() { ... }
```

```
void flushLogs(char* msg) { ... }
```

```
int writeToFile(File file, char *bytes) { ... }
```

```
void logMessage(char* msg) { ... }
```

```
void deleteMessage(char* msg) { ... }
```

```
void openLogFile() { ... }
```

```
void setLogFileName(char *fileName) { ... }
```

loggning

filhantering



logging.c | h

```
void searchMessages(char *msg) { ... }  
void closeLogFile() { ... }  
void flushLogs(char *msg) { ... }  
void logMessage(char *msg) { ... }  
void deleteMessage(char *msg) { ... }  
void openLogFile() { ... }  
void setLogFileName(char *fileName) { ... }
```

```
File openFile(char *fileName) { ... }  
char *readFromFile(File file, int size) { ... }  
int writeToFile(File file, char *bytes) { ... }
```

file_handling.c | h



Informationsgömning

- En moduls implementationsdetaljer skall inte vara möjliga att observera utifrån

- Designprincip:

Göm föränderliga detaljer bakom ett stabilt gränssnitt

- Inkapsling är en term som ofta används synonymt med informationsgömning

Man kan se inkapsling som en teknik, informationsgömning som en princip

Sammanfattning

- Att ett program är korrekt och effektivt är bara två egenskaper av många som ett bra program skall ha
- Använd alltid lämpliga abstraktioner för att göra program överskådliga och enklare att ändra (kontrollabstraktion och dataabstraktion, t.ex.)
- Designprincipen modularisering är ett viktigt verktyg för att bryta ned ett problem i (allt) mindre beståndsdelar som blir enklare att lösa

Dela alltid upp era program i moduler

- Designprincipen informationsgömning är viktig för att skydda abstraktioner

Ge en modul ett stabilt gränssnitt (i en .h-fil i C)

- Inkapsling är en viktig teknik för informationsgömning

Exponera aldrig interna funktioner eller struktdefinitioner i gränssnittet (.h-filen)

Bindning



Olika typer av bindning

Statisk bindning — avgörs vid kompileringstillfället

`puts(a)`

`funktions_pekare(42)`

Dynamisk bindning

`funktions_pekare(42)`

`a.puts()`