

Föreläsning 9

Tobias Wrigstad

”Hur man tänker”



”Hur man tänker”



Att tänka i lager

Närmare domänen



Närmare maskinen

`ask_string_question`

Ställ fråga, läs strängsvar

`getline`

Läs en rad

`getchar`

Läs ett tecken

Att tänka i lager

`ask_string_question`

För varje funktion/strukt — vilket lager tillhör den?

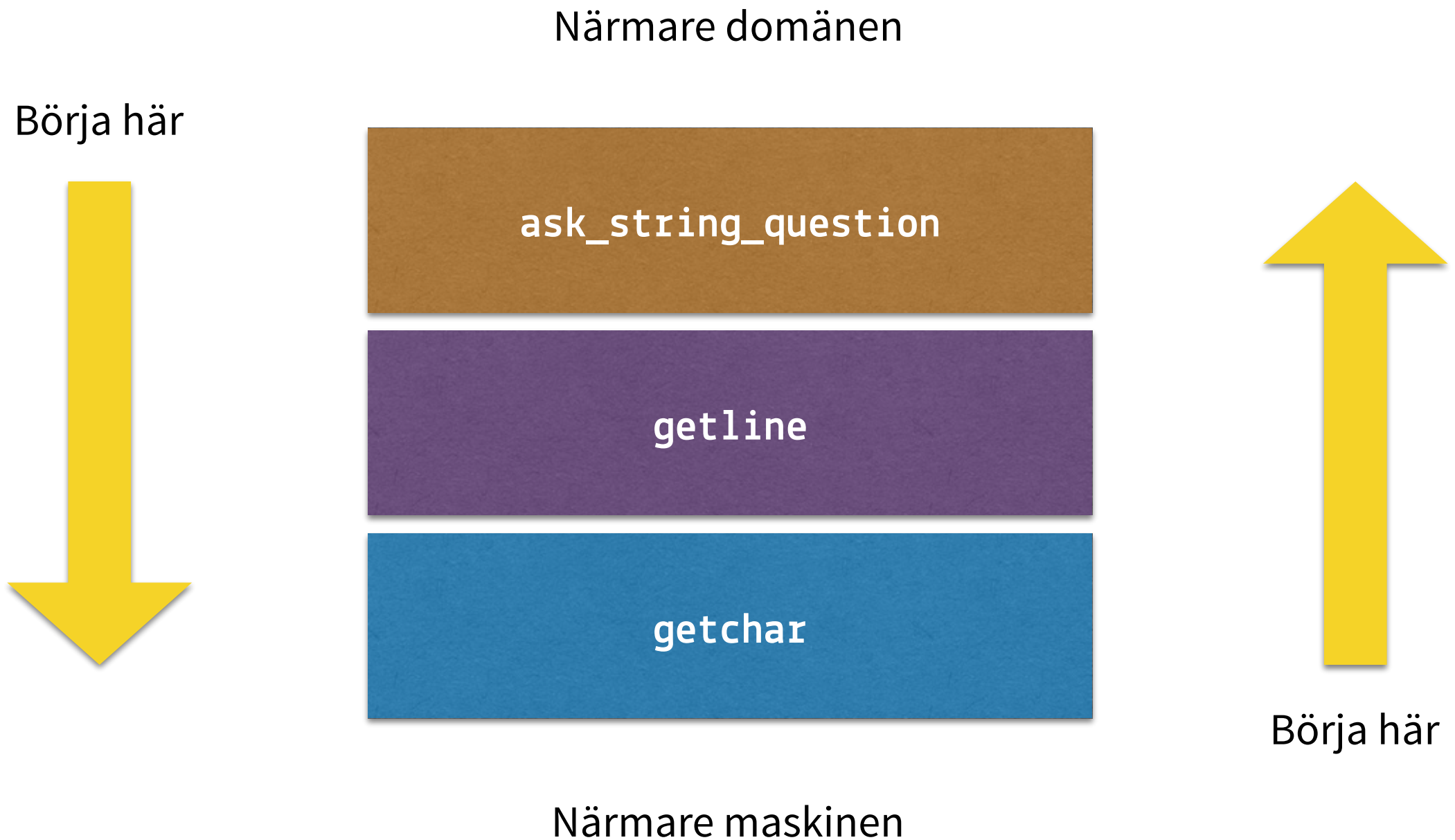
`getline`

Vad bör den (inte) vara beroende av?

`getchar`

Vilka lager ”behöver” jag?

Top-down eller bottom up?



Att arbeta bottom-up

- Börja på den ”lägsta nivån”

Högre teknisk komplexitet, ofta helt orelaterade till domänen

- Nackdelar/risker:

Att du gör något som inte behövs senare

Det kan kännas motigt att vara så långt borta från domänen/specifikationen

- Konsekvenser:

Mindre fuskande eftersom varje funktion bygger på de som vi just implementerat

Att arbeta top-down

- Börja på den ”högsta nivån”

Omfattar i regel mest logik som hör till specifikationen/domänen

- Nackdelar/risker:

Man kan fatta dumma designbeslut på grund av tekniska omständigheter som man inte känner till ännu

Kan vara svårt att ha ett körande program hela tiden

- Konsekvenser:

Ofta mer fuskande eftersom man ”knuffar funktionalitet framför sig”

Vilket skall jag välja?

- Det är litet beroende på person

Om du inte direkt vet vilket som är bäst för dig — prova båda och utvärdera!

- **Min rekommendation:** top-down är bättre för den som känner sig osäker på C

Implementation av frågor i lagerhanteraren

```
/// Asks a question and reads a string in response
char *ask_string_question(char *question)
{
    puts(question);
    return read_string();
}
```

```
/// Asks a question and reads a char in response
int ask_int_question(char *question)
{
    puts(question);
    return read_int();
}
```

Implementation av frågor i lagerhanteraren

```
/// Asks a question and reads a string in response
char *ask_string_question(char *question)
{
    puts(question);
    return read_string();
}
```

```
/// Asks a question and reads a char in response
int ask_int_question(char *question)
{
    puts(question);
    return read_int();
}
```

Fusk!



Implementation av frågor i lagerhanteraren

Löser fusk 2 från föregående bild!

```
/// Reads a line from the keyboard and converts it to an int
int read_int()
{
    char *buf = read_string();
    int result = atol(buf);

    free(buf);
    return result;
}
```

Fusk!

Skarv: anta att input alltid är valida heltal

Implementation av frågor i lagerhanteraren

Löser resterande fusk från föregående bilder!

```
/// Reads a line from the keyboard, puts it on the heap and returns a  
/// pointer to it  
char *read_string()  
{  
    char *buf = NULL;  
    size_t len = 0;  
    ssize_t read = getline(&buf, &len, stdin);  
    buf[read-1] = '\\0'; // Skarv 1 & 2  
    return buf;  
}
```

***Skarv 1:** anta att vi aldrig vill ha newline kvar!*

***Skarv 2:** anta att newline == '\\n' (stämmer ej på Windows)*

Implementation av frågor, bottom-up

- I stort sett som top down, fast baklänges

Först `read_string`

Sedan `read_int` ovanpå `read_string`

Sedan `ask_int_question` ovanpå `read_int`

- Det är ett visst avstånd som måste överbryggas från vad programmet vill (`ask_int_question`) och `read_string`.

Gör på det sätt du själv känner att det är lättast att tänka

Nästa steg: ta bort skarvarna

```
/// Reads a line from the keyboard, puts it on the heap and returns a
/// pointer to it
char *read_string(bool strip_newline)
{
    char *buf = NULL;
    size_t len = 0;
    ssize_t read = getline(&buf, &len, stdin);
    if (strip_newline && read > 0) buf[read-1] = '\\0'; // Skarv 2 är kvar
    return buf;
}
```

Användaren får välja om strängen skall ha kvar newline!

Nästa steg: ta bort skarvarna

```
/// Reads a line from the keyboard, puts it on the heap and returns a
/// pointer to it
int read_int(bool repeat_until_valid_int)
{
    char *buf = NULL;

    do {
        if (buf) free(buf);

        buf = read_string(true);

    } while (repeat_until_valid_int && is_number(buf) == false);

    int result = atol(buf);
    free(buf);
    return result;
}
```

Fusk!



Nästa steg: ta bort skarvarna

”Ha alltid ett körbart program”

```
/// Returns true if a string only has digits
bool is_number(char *str)
{
    return true;
}
```

*Gör så att vi kan testa
programmet, men
funkar förstås inte
för icke-valid input!*

Nästa steg: ta bort skarvarna

Löser fusket från föregående bild!

```
/// Returns true if a string only has digits
bool is_number(char *str)
{
    bool valid_int = true;

    for (char *c = str; *c && valid_int; ++c)
    {
        valid_int = isdigit(*c);
    }

    return valid_int;
}
```

*Loopa igenom
strängen och kolla
att varje char är en
siffra*

`read_string` lägger en sträng på heapen

- Inte alltid rätt, t.ex. i `read_int`
- Tänk om jag vill läsa in direkt i en `char`-array i databasen?
- Logiken är densamma oavsett var i minnet man läser in strängen

Bra idé: bryt ut detta ur funktionen så blir den mer generell

Lättare att återanvända

Lättare att testa

- Sedan kan vi bygga en ekvivalent `read_string` ovanpå den generella, som sparar en sträng på heapen

Nästa steg: ta bort skarvarna

En mer generell funktion för att läsa in strängar!

```
/// Reads a line from the keyboard, puts it in the len-sized
/// memory space pointed to by buf, and optionally removed newlines
char *read_string_with_buffer(char *buf, size_t len, bool strip_newline)
{
    ssize_t read = getline(&buf, &len, stdin);
    if (read > 0 && strip_newline)
    {
        buf[read-1] = '\\0'; // -2 på Windows...
    }
    return buf;
}
```

Skarv: utgår från att vi inte kör på Windows...

Undvik onödig upprepning

Vi kan enkelt implementera om `read_string()` i termer av `read_string_with_buffer()`

```
/// Reads a line from the keyboard, removes newlines,  
/// puts on the heap and returns a pointer to it  
char *read_string()  
{  
    return read_string_with_buffer(NULL, 0, true);  
}
```



Oftast behöver man inte newline — vill man ha det får man använda `read_string_with_buffer`

Inga magiska konstanter (läsbarhet)

```
/// Uses the improved read_string_with_buffer
int read_int(bool repeat_until_valid_int)
{
    char *buf = alloca(16); // 16 is a big number
    int len = 16;

    do
    {
        buf = read_string_with_buffer(buf, len, true);

    } while (repeat_until_valid_int && is_valid_int(buf) == false);

    return atol(buf);
}
```

?! 

Inga magiska konstanter (läsbarhet)

```
#define STRIP_NEWLINE true

/// Uses the improved read_string_with_buffer
int read_int(bool repeat_until_valid_int)
{
    char *buf = alloca(16); // 16 is a big number
    int len = 16;

    do {

        buf = read_string_with_buffer(buf, len, STRIP_NEWLINE);

    } while (repeat_until_valid_int && is_valid_int(buf) == false);

    return atol(buf);
}
```

Inga magiska konstanter (läsbarhet)

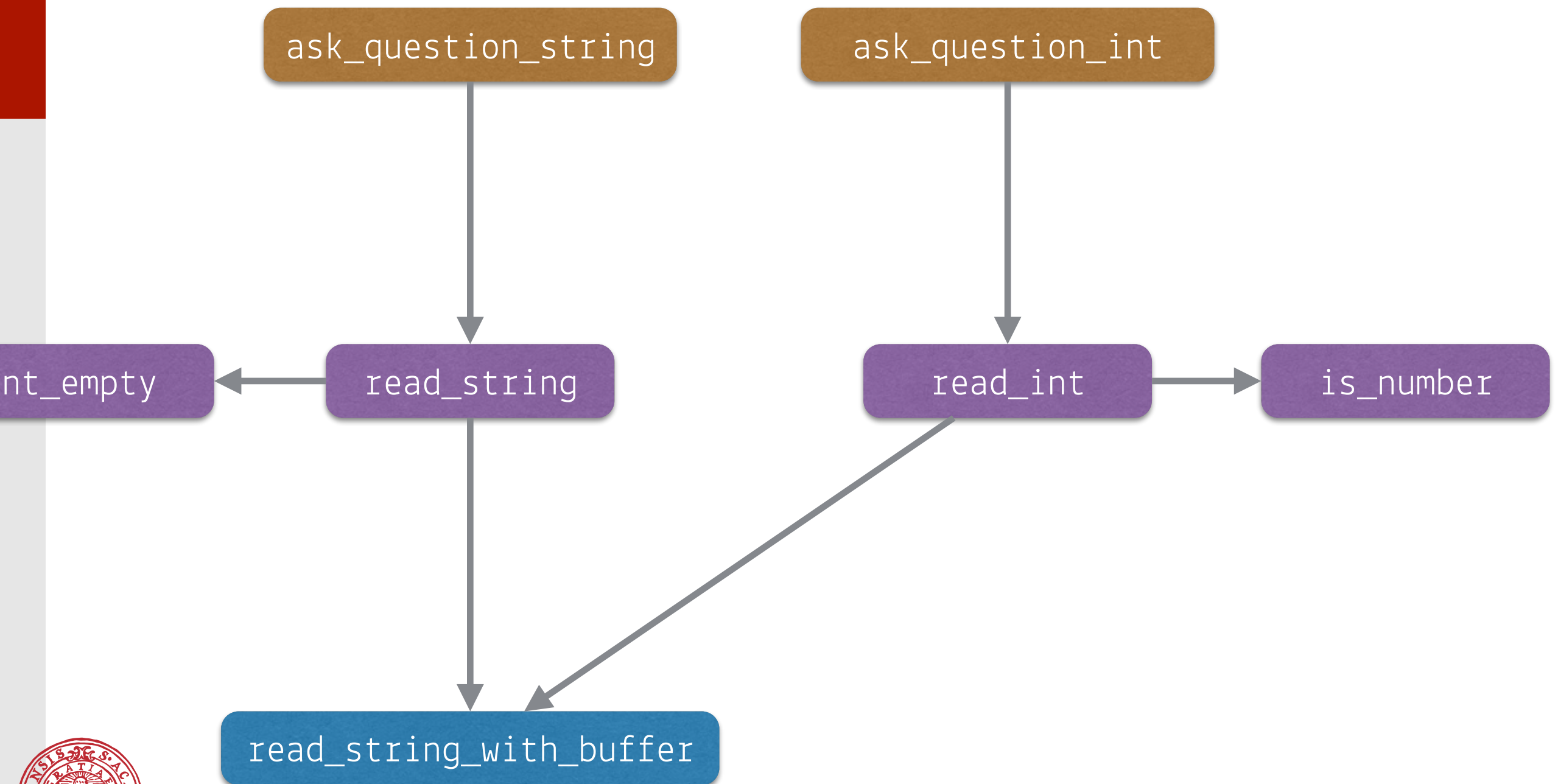
Vi kan enkelt implementera om `read_string()` i termer av `read_string_with_buffer()`

```
/// Reads a line from the keyboard, removes newlines,  
/// puts on the heap and returns a pointer to it  
char *read_string()  
{  
    return read_string_with_buffer(NULL, 0, STRIP_NEWLINE);  
}
```



Oftast behöver man inte newline — vill man ha det får man använda `read_string_with_buffer`

Beroenden



Ytterligare påbyggnad

- Med hjälp av våra två ask kan vi enkelt bygga en ask shelf (t.ex. A25)

```
/// Reads the necessary data for a shelf_t, constructs a  
/// shelf_t, and returns it  
shelf_t ask_shelf_question()  
{  
    shelf_t shelf;  
    shelf.name    = ask_string_question("Mata in ett tecken")[0];  
    shelf.number = ask_int_question("Mata in ett tal");  
    return shelf;  
}
```

Skarv: förutsätter valitt indata

Fixa skarven: validera

```
shelf_t ask_shelf_question()
{
    shelf_t shelf; // har char name; int number;

    char *name = NULL;
    do {
        if (name) free(name);
        name = ask_string_question("Mata in ett tecken");
        shelf.name = name[0];
    } while (strlen(name) != 1);

    long number = 0;
    do {
        number = ask_int_question("Mata in ett tal 0--99");
        shelf.number = number;
    } while (0 <= number && number < 100);

    return shelf;
}
```

Repetition!

”Bad smell: upprepning”

- Det finns ett mönster som upprepas

Läs in data

Validera

(Ta bort temporära data)

Skapa efterfrågad struktur

- Samma mönster, men olika beteende för olika data

```
shelf_t ask_shelf_question()
{
    shelf_t shelf; // har char name; int number;

    char *name = NULL;
    do {
        if (name) free(name);
        name = ask_string_question("Mata in ett tecken");
        shelf.name = name[0];
    } while (strlen(name) != 1);

    long number = 0;
    do {
        number = ask_int_question("Mata in ett tal 0--99");
        shelf.number = number;
    } while (0 <= number && number < 100);

    return shelf;
}
```

Generalisering

- Kan vi skapa en funktion som följer mönstret men som gör rätt sak för rätt data?
- Försök ett: vi skickar in "flaggor" för att tala om vad för data etc. skall läsas in
 - + Löser problemet
 - Kodan blir väldigt komplicerad
 - Går inte att utöka för data som vi inte känner till
- Försök två: bryt ut beteende med hjälp av funktionspekare
 - + Löser problemet
 - + Framtidssäker eftersom logiken tillhandahålls av användaren

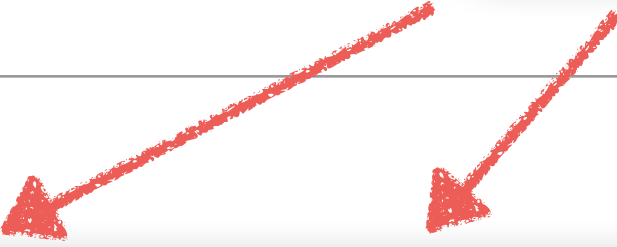
Funktionspekare

```
/// Valideringsfunktion tar emot en pekare till
/// data och kontrollerar om datat kan omvandlas
/// till avsedd typ
/// Exempel:
///   - valid_int
///   - valid_shelf
typedef bool  (*v_f)(char *);

/// Konstruktor som tar emot en sträng med validerat
/// data och omvandlar det till avsedd typ, returnerar
/// en pekare till det nya datat
/// Exempel:
///   - str_to_int
///   - str_to_shelf
typedef void *(*m_f)(char *);
```

Generaliserad fråga

```
typedef bool  (*v_f)(char *);  
typedef void *(*m_f)(char *);
```



```
void *ask_question(char *q, v_f validate, m_f convert, bool cleanup)  
{  
    // Ask question until optional validation of input is satisfied  
    char *input = NULL;  
    do {  
        puts(q);  
        if (input) free(input);  
        input = read_string();  
    } while (validate && validate(input) == false);  
  
    // If a conversion function was specified, convert input  
    void *result = convert ? convert(input) : input;  
  
    if (cleanup) free(input);  
    return result;  
}
```

Validera data, skapa data

```
bool valid_shelf(char *input)
{
    return strlen(input) == 3 && isalpha(input[0]) && valid_int(input+1);
}

shelf_t *str_to_shelf(char *input)
{
    shelf_t *shelf = malloc(sizeof(shelf_t));
    shelf->name = input[0];
    shelf->number = atol(input+1);
    return shelf;
}
```

Slutlig ask_shelf/string_question

```
shelf_t ask_shelf_question()
{
    return *ask_question("Mata in en hyllplats (tecken, följt av siffra 0-99)",
                          valid_shelf,
                          str_to_shelf,
                          true);
}
```

```
char *ask_string_question()
{
    return *ask_question("Mata in en sträng",
                          NULL,
                          NULL,
                          false);
}
```

*Skarv: fungerar för heltal,
men "fult"*

Förbättring: unioner

```
// ändra void * => result_t i ask_question och m_f
typedef union result result_t;

union result
{
    void *ptr;
    long int_value;
    char char_value;
};

result_t str_to_shelf(char *input)
{
    shelf_t *shelf = malloc(sizeof(shelf_t));
    shelf->name = input[0];
    shelf->number = atol(input+1);
    return (result_t) { .ptr = shelf };
};
```

Lösning med hjälp av unioner (nästan samma)

```
result_t str_to_int(char *s)
{
    return (result_t) { .int_value = atol(s) };
}
```

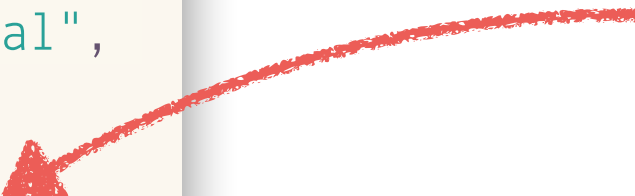
```
result_t str_to_str(char *s)
{
    return (result_t) { .ptr = s };
}
```

```
int ask_int_question()
{
    return ask_question("Mata in ett heltal",
                        valid_int,
                        str_to_int,
                        true).int_value;
}
```

x = foo()
return x.bar;

är samma som

return foo().bar;



Till slut

```
shelf_t *ask_shelf_question()
{
    return ask_question("Mata in en hyllplats (tecken, följt av siffra 0-99)",
                        ok_shelf,
                        make_shelf,
                        true).ptr;
}
```

```
char *ask_string_question()
{
    return ask_question(
        "Mata in en sträng",
        NULL,
        str_to_str,
        false).ptr;
}
```

```
int ask_int_question()
{
    return ask_question(
        "Mata in ett heltal",
        valid_int,
        str_to_int,
        true).int_value;
}
```

Ännu bättre: exponering för programmet med hjälp av makron

```
/// Grundläggande funktioner
#define Ask_int(q)          ask_question(q, valid_int, str_to_int, true)
#define Ask_str(q)          ask_question(q, NULL, str_to_str, false)

/// Funktion för specifika återkommande frågor
#define Ask_namn()          Ask_str("Namn:")
#define Ask_beskrivning( ) Ask_str("Beskrivning:")
#define Ask_pris()          Ask_int("Pris:")
#define Ask_lagerhylla()    ask_question("...", valid_shelf, str_to_shelf, true)
#define Ask_antal()         Ask_int("Antal:")
```

OBS! Detta kan man alltså göra även utan funktionspekare och unioner!

Titta nu på add_goods — hur ”ren” den blir

```
void add_goods(db_t *db)
{
    goods_t g;

    g.name     = Ask_namn();
    g.desc     = Ask_beskrivning();
    g.price    = Ask_pris();
    g.shelf    = Ask_lagerhylla();
    g.amount   = Ask_antal();

    db->goods[db->total] = g;
    ++db->total;
}
```

Skarv: följer inte specen (inget val spara/redigera)

Ta bort skarven — fortfarande snyggt

```
void add_goods(db_t *db)
{
    goods_t g;

    do {
        g.name    = Ask_namn();
        g.desc    = Ask_beskrivning();
        g.price   = Ask_pris();
        g.shelf   = Ask_lagerhylla();
        g.amount  = Ask_antal();

        char answer = Ask_char("Spara? (ja/nej)");
    } while (strchr("Jj", answer) == false);

    db->goods[db->total] = g;
    ++db->total;
}
```

Fusk!

Skarv: följer inte specen (inget redigera-val)

Återanvändning i edit_goods

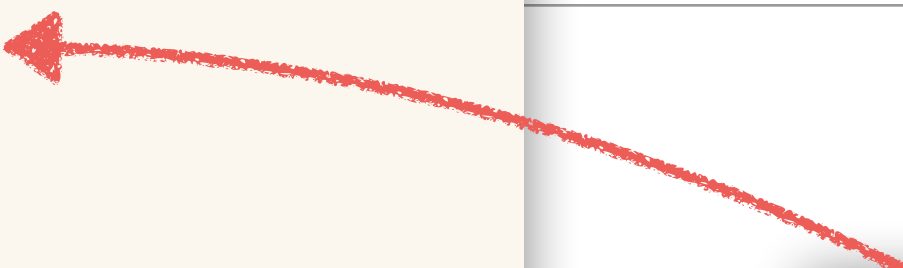
```
void edit_goods(db_t *db, goods_t *g)
{
    char answer = Ask_char();

    goodt_g copy = *g;


    switch (answer) {
    case 'N':
    case 'n': copy.name = Ask_namn(); break;
    // etc.
    case 'P':
    case 'p': copy.price = Ask_pris(); break;
    }

    print_goods(copy); // fusk!

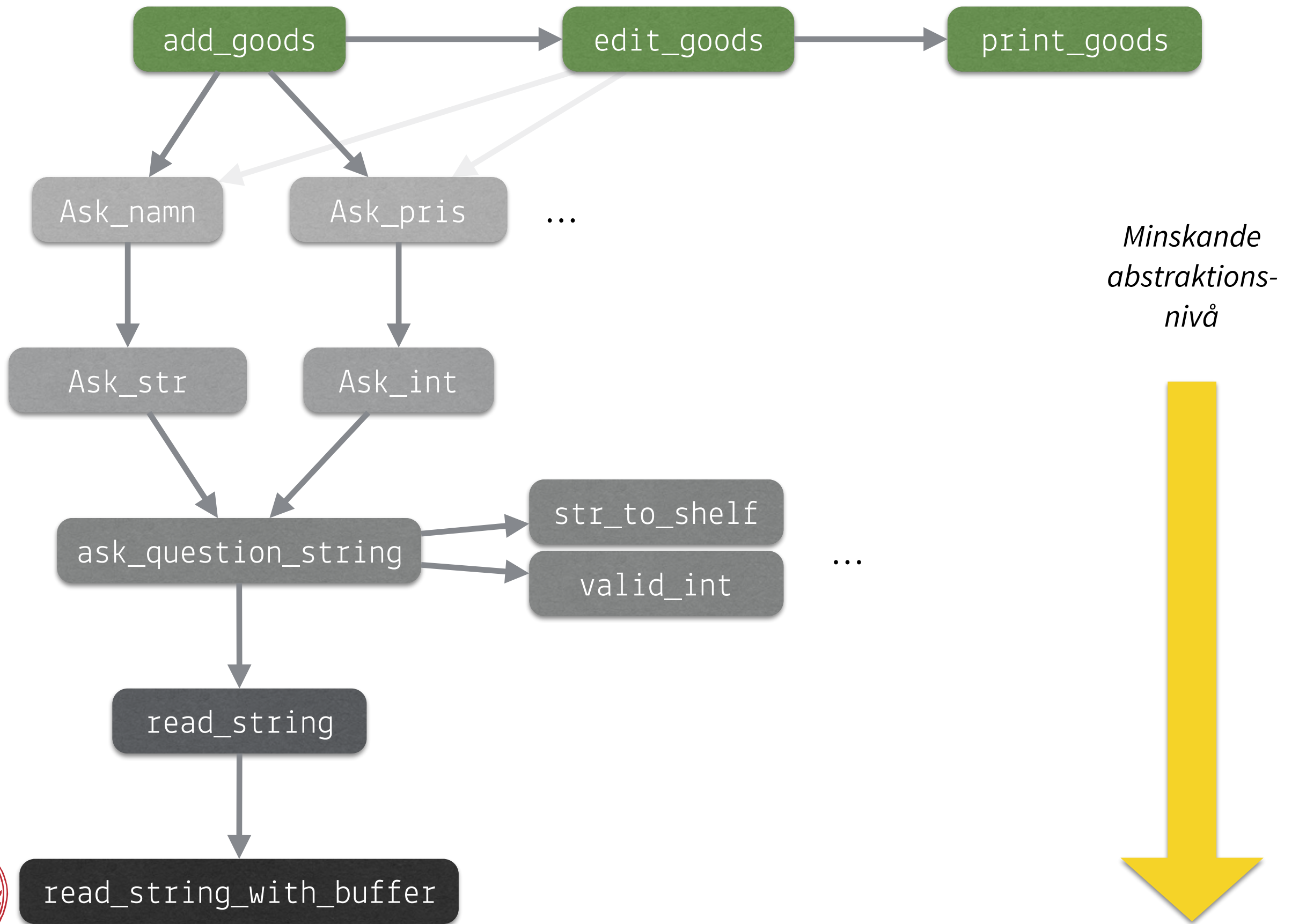
    char answer = Ask_char("Spara? (ja/nej)");
    if (strchr("Jj", answer) == false)
    {
        *g = copy;
    }
}
```

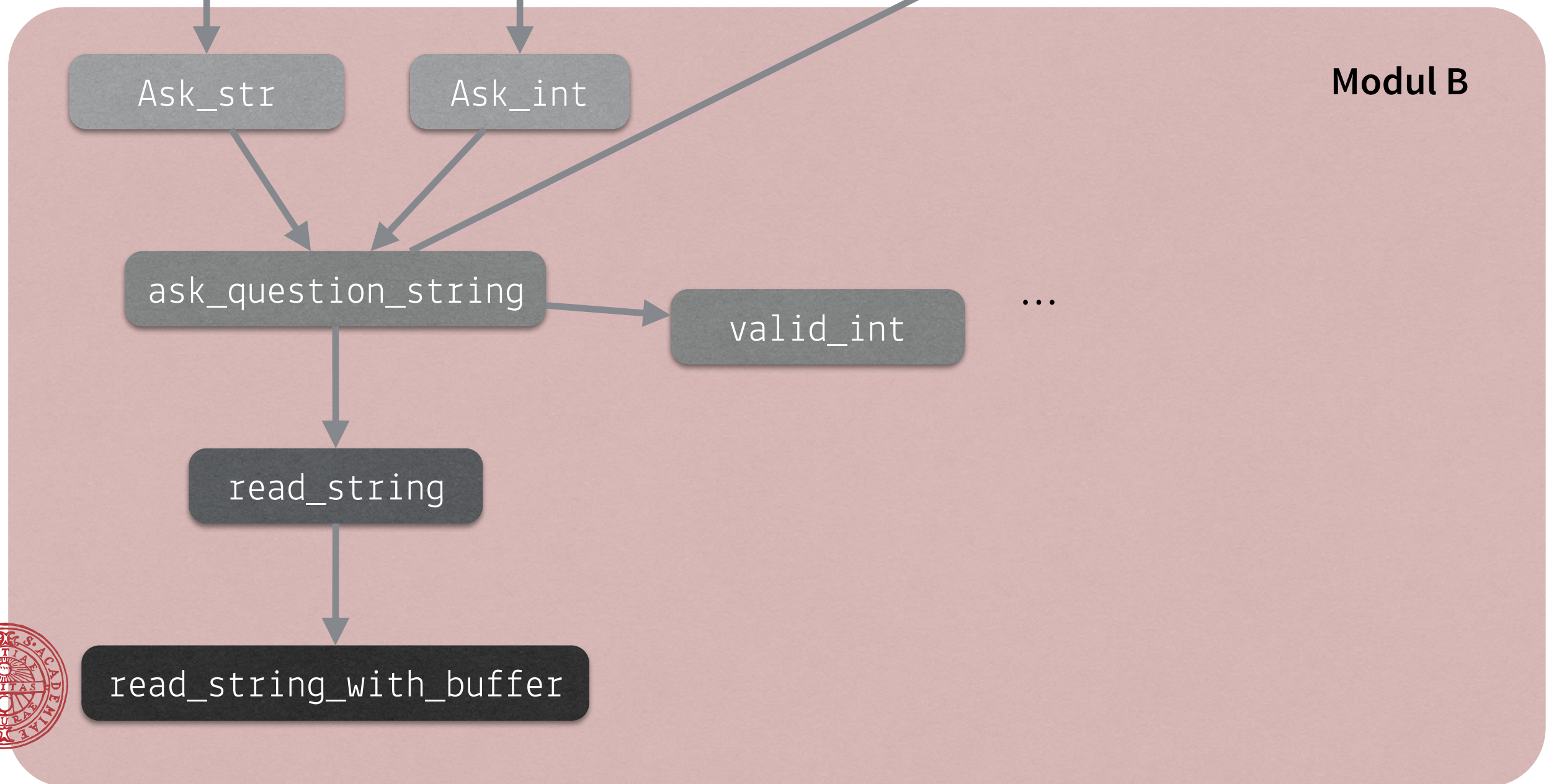
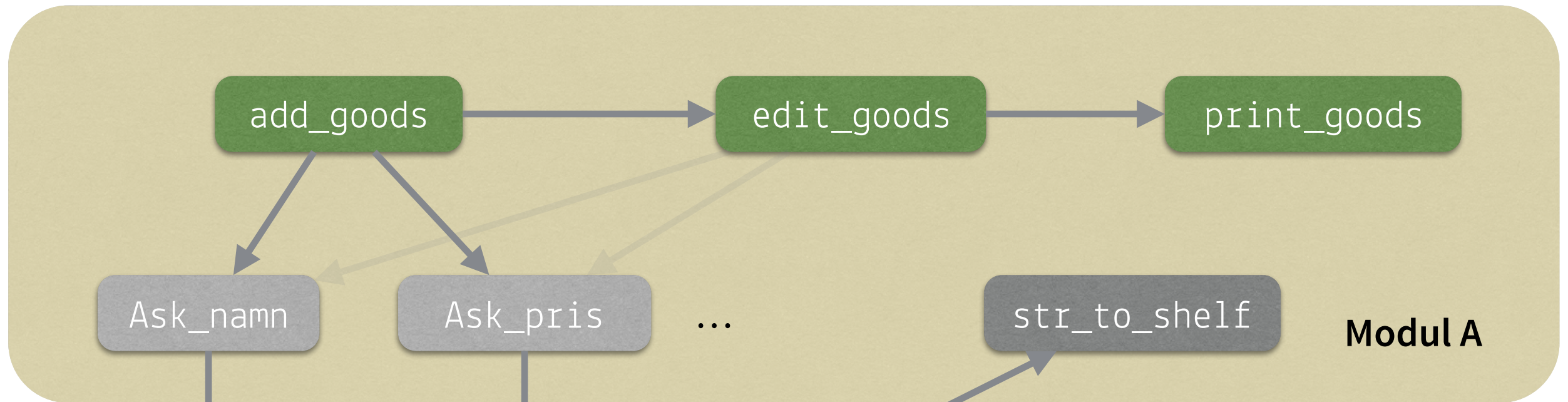


```
"Vad vill du redigera?\n"
"[N]amn\n"
// etc
"[P]ris"
```



Kanske skulle detta ges en egen funktion som också anropades i add_goods?





Krångligare med så många mellansteg?

- Se på koden för `add_goods`, den är **tydlig** eftersom den slipper bry sig så mycket om tekniska detaljer

Procedurell abstraktion: tydligt vad varje funktion gör, även utan insyn

Endast ett anrop till `getline` i all kod

- I många andra funktioner, t.ex. `edit`, kunde jag återanvända `Ask_-`funktionerna och därigenom få lika fin och ren kod som i `add_goods` — ”gratis”
- Observera att man måste inte ha ”supergenerella” funktioner i botten

Man kan ha separata `read`-funktioner utan funktionspekare etc.

Sammanfattning

- Top-down eller bottom-up

Vad är rätt för dig?

- Lagertänkande

Bygger abstraktioner bit för bit

Lager är **inte** detsamma som moduler

- Generella byggstenar kan återanvändas
- Programmera nära domänen
- Göm tekniska komplexiteter ”längre ned”

Läsbar kod

- Notera att den längsta funktionen här är ~20 rader (`edit_goods`) — den är för lång!
- De flesta funktionerna är ca 5–6 rader — en bra längd
- Funktioner skall helst bara göra en sak
- Om de har för många rader så blir det svårt att överblicka vad de gör

Svårt att se att de är korrekta

Svårt att underhålla, förstå, etc.