

Imperativ och objektorienterad programmeringsmetodik

Föreläsning 8 av många

Tobias Wrigstad

Bottom-up vs. Top-down, lagertänkande, läsbar kod

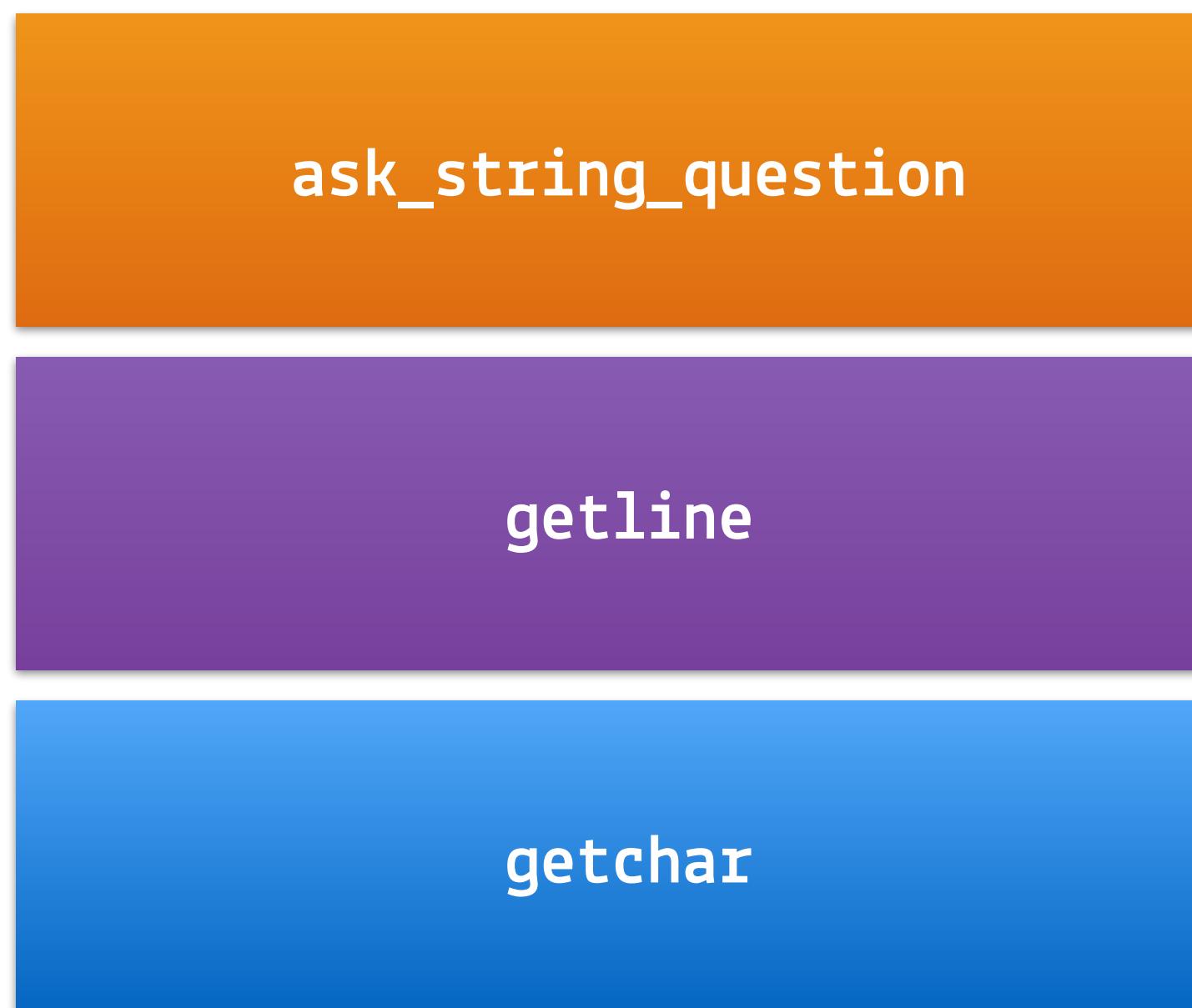


Att tänka i lager

Närmare domänen



Närmare maskinen



`ask_string_question`

`getline`

`getchar`

Ställ fråga, läs strängsvar

Läs en rad

Läs ett tecken

Att tänka i lager

`ask_string_question`

För varje funktion/strukt – vilket lager tillhör den?

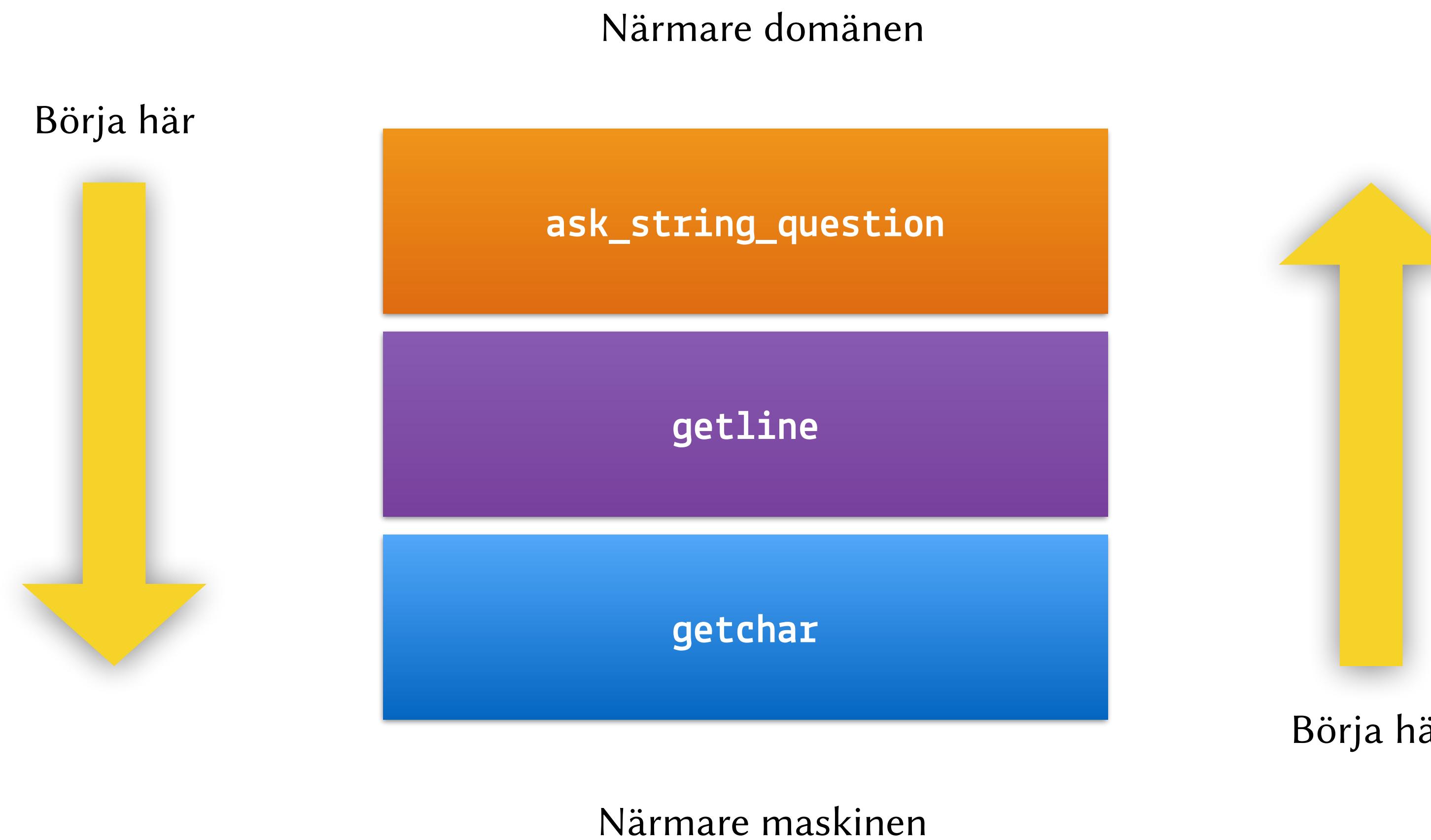
`getline`

Vad bör den (inte) vara beroende av?

`getchar`

Vilka lager ”behöver” jag åtkomst till för att lösa mitt problem?

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 känna 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 shall 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 under bootstrapplabbarna

- Implement the functions `ask_question_string` and `ask_question_char`

Asks a question and reads a string in response

Asks a question and reads a char in response

- Recap

Cheat/Fuska — låtsas att det existerar en funktion som utför ett aktuellt delsteg

Dodge/Skarv — förenkla problemet t.ex. genom att utgå från att vissa fall aldrig uppkommer

Implementation av frågor under bootstrapbarna

- Implement the functions ask_question_string and ask_question_char

- Asks a question and reads a string in response

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

- Asks a question and reads a char in response

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

Implementation av frågor under bootstrapplabbarna

- Implement the functions ask_question_string and ask_question_char

- Asks a question and reads a string in response

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

- Asks a question and reads a char in response

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

Fusk!

Implementation av frågor under bootstrapplabbarna

- Implement the functions `ask_question_string` and `ask_question_char`

Asks a question and reads a string in response

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



```
/// TODO: implement
char *read_string()
{
    return "TODO!";
}
```

Asks a question and reads a char in response

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



```
/// TODO: implement
int read_int()
{
    return 42;
}
```

Implementation av frågor under bootstrapplabbarna

- Implement the functions ask_question_string and ask_question_char
 - Asks a question and reads a string in response

```
/// TODO: implement
char *read_string()
{
    return "TODO!";
}
```

Fusk!

- Asks a question and reads a char in response

```
/// 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;
}
```

- Skarv: anta att input alltid är valida heltal

Fixa sista fusket

- 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 alla plattformar)

Implementation av frågor, bottom-up [som på bootstrapplabbarna]

- 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`.

Top-down eller bottom-up? Gör på det sätt du själv känner att det är lättast att tänka

Nästa steg: ta bort skarvana

```
/// 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;
}
```

- ✓ Lösning: användaren får välja om strängen skall ha kvar newline eller ej!

Nästa steg: ta bort skarvana



```
/// 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!

(vi låtsas att det finns en
funktion is_number som
löser vårt problem)

Nästa steg: ta bort skarvorna



```
/// 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!

”Ha alltid ett körbart program”

Nästa steg: ta bort skarvorna

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

Nu när behovet finns: ny variant av `read_string`

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 vissa platfformar...
    }
    return buf;
}
```

Skarv: utgår från att newline är ett enda tecken

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_number(buf) == false);

    return atol(buf);
}
```



Photo by Dex Ezekiel on Unsplash

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_number(buf) == false);

    return atol(buf);
}
```

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, STRIP_NEWLINE);  
}
```

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

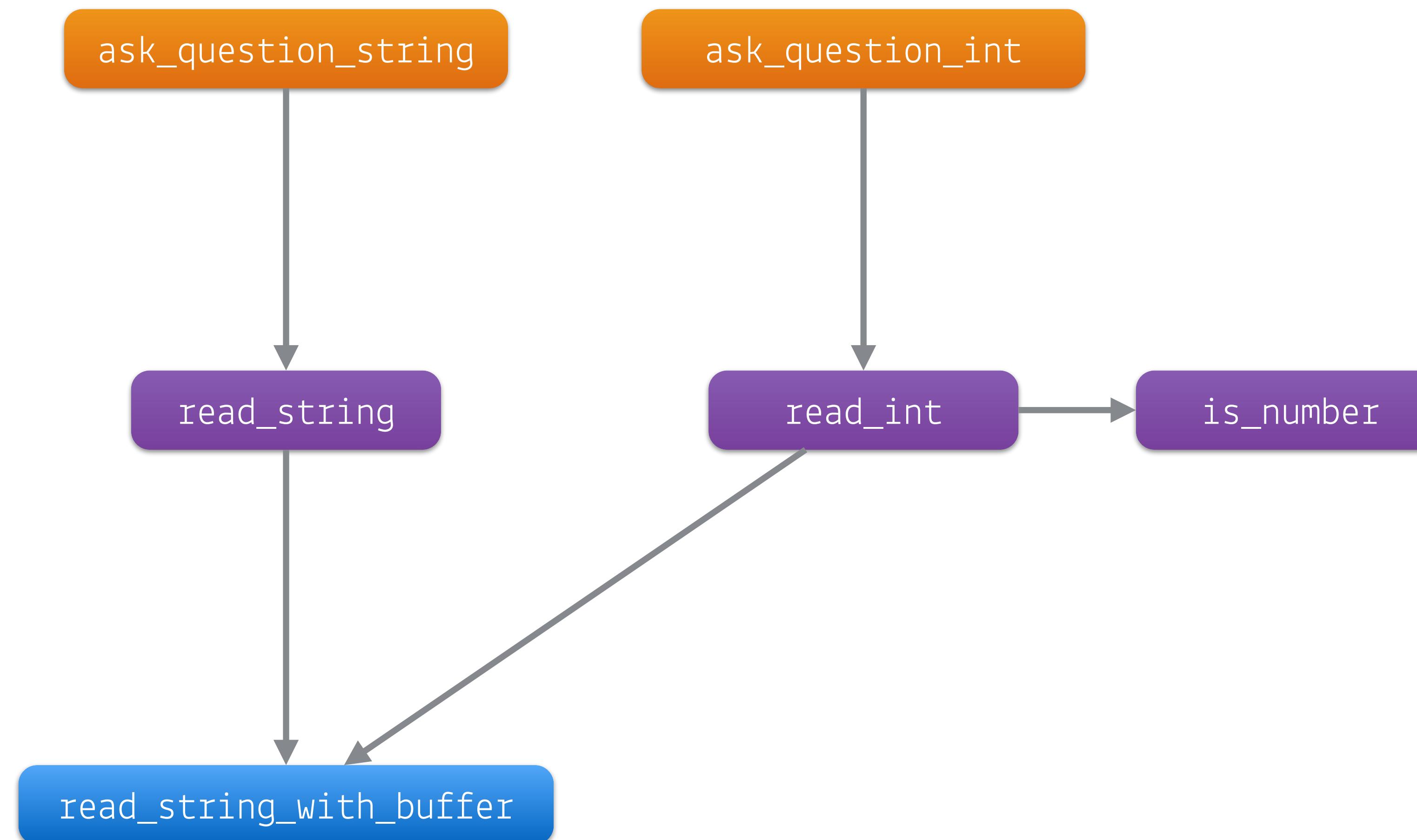
Undvik onödig upprepning

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()  
{  
    char *buffer = NULL;  
    size_t initial_buffer_size = 0;  
    return read_string_with_optional_buffer(buffer, initial_buffer_size, 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()
{
    return (shelf_t) {
        .name    = ask_question_string("Mata in ett tecken")[0],
        .number = ask_question_int("Mata in ett tal")
    };
}
```

Skarv: förutsätter valitt indata

Skarv: ask_question_string läcker minne?

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);
    free(name);

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

    return shelf;
}
```



Upprepning!

”Bad smell”: upprepning

```
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);
    free(name);

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

    return shelf;
}
```

- 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

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
 - Koden blir mer kompllicerad
 - 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 är ett sätt att parameterisera beteende

```
/// 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 *);  
  
/// Konstruktör 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 *(*c_f)(char *);
```

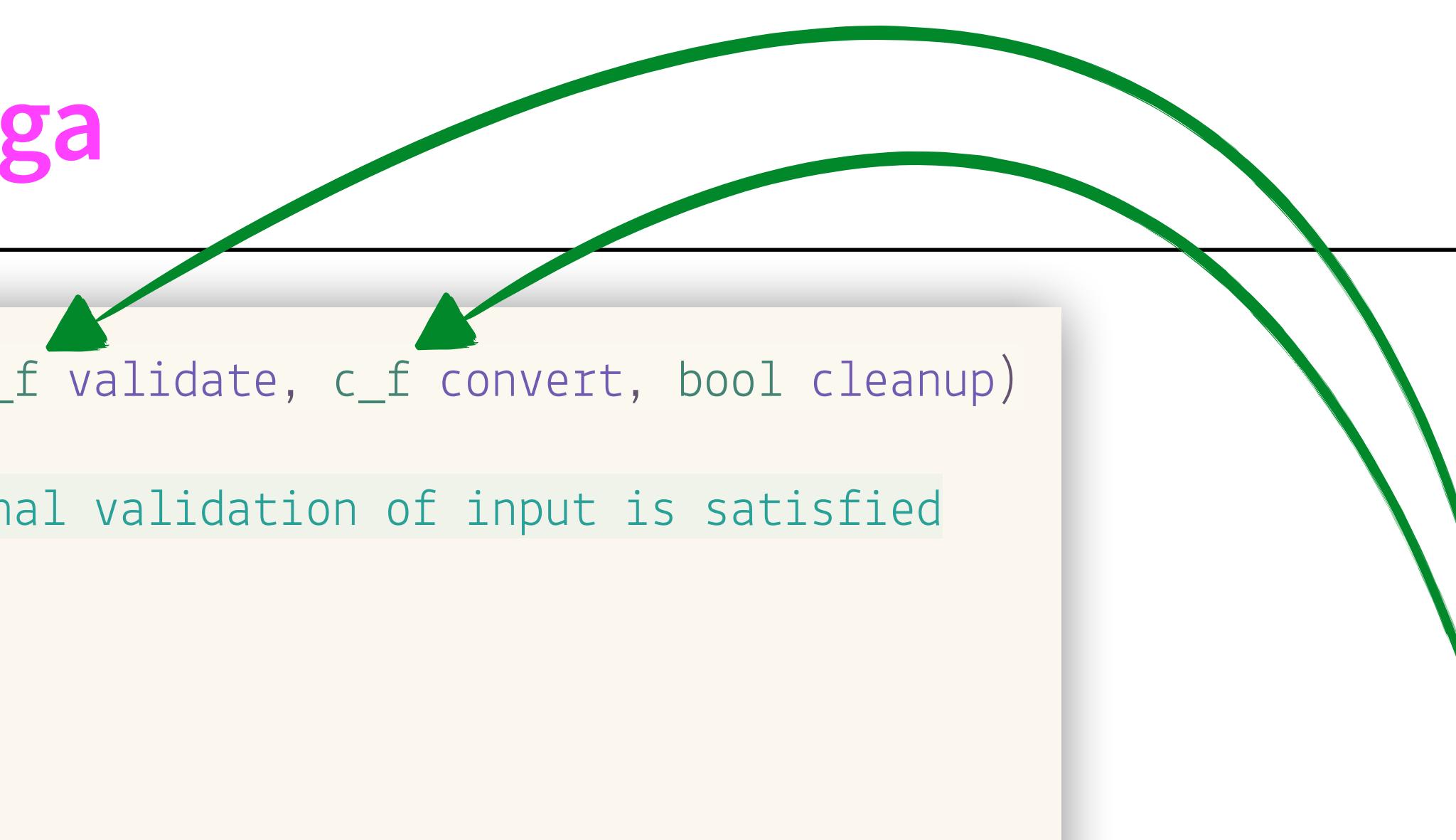
(Naturligtvis med bättre namn på typerna!)

Generaliserad fråga

```
void *ask_question(char *q, v_f validate, c_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;
}
```



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

Validera data, skapa data — hyllversionen

```
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 = toupper(input[0]);
    shelf->number = atol(input+1);
    return shelf;
}
```

Slutlig ask_shelf/string_question

```
shelf_t *ask_shelf_question()
{
    return (shelf_t *)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 * => answer_t i ask_question och m_f
typedef union answer answer_t;

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

answer_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)

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

```
answer_t str_to_str(char *s)
{
    return (answer_t) { .string_value = s };
}
```

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

x = foo()
return x.bar;

är samma som

return foo().bar;



Till slut

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

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

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

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!

Straightline code – läsbart?

```
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;
}
```

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 (strcmp("Jj", answer) == false);

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

Fusk!

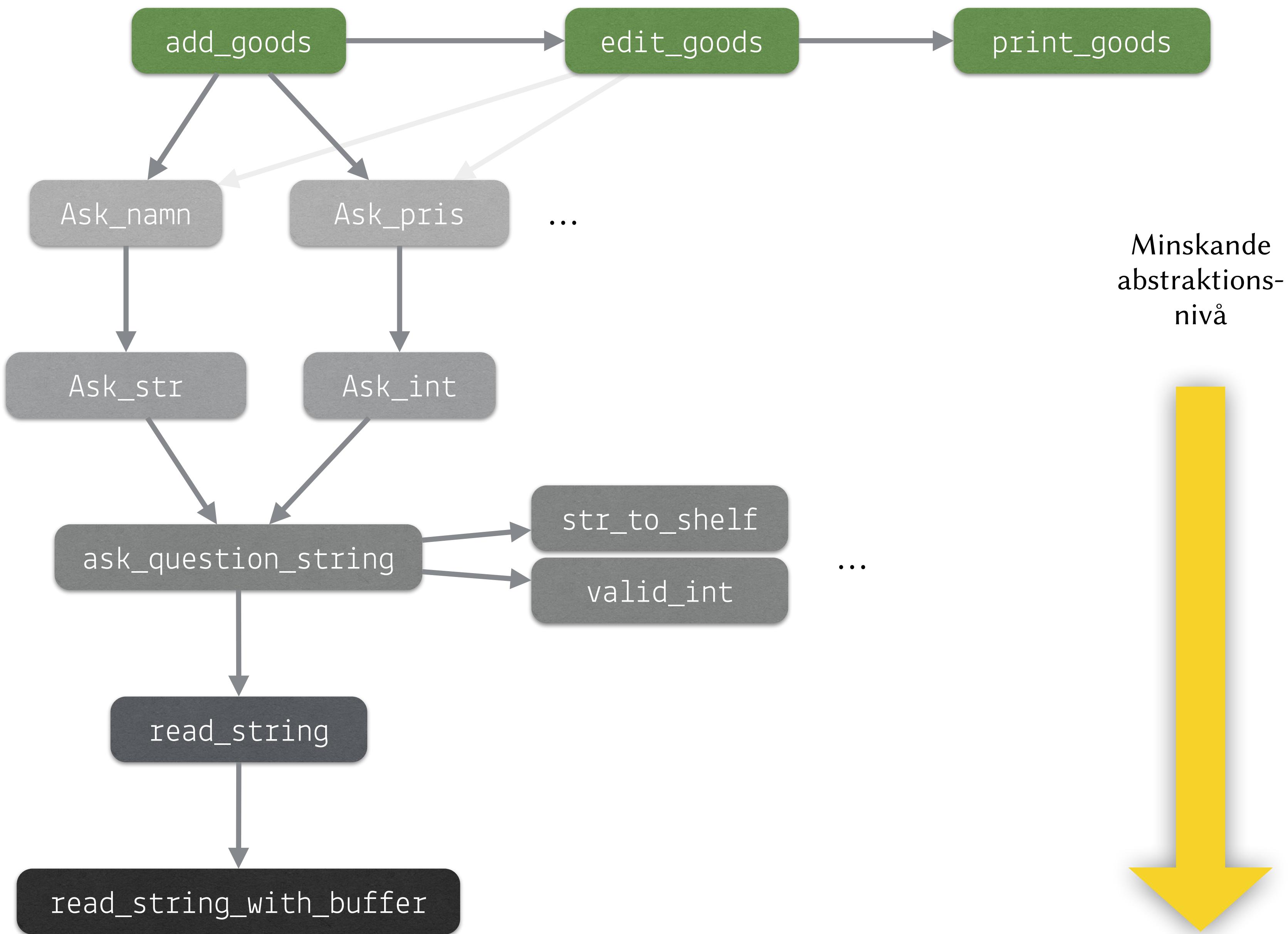
Skarv: följer inte specen (inget redigera-val)

Återanvändning i edit_goods

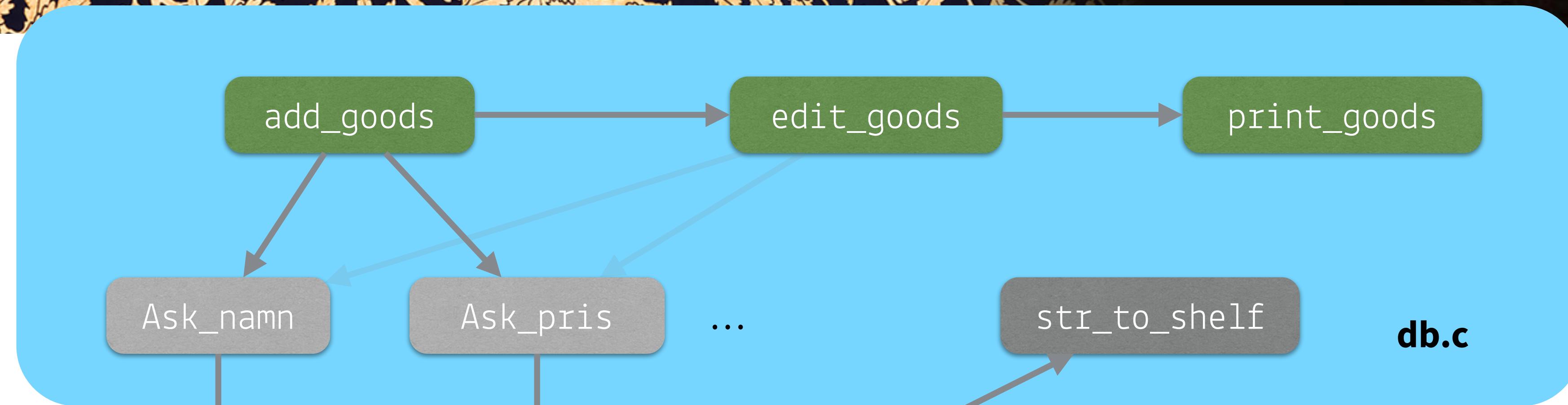
```
void edit_goods(db_t *db, goods_t *g)
{
    char answer = Ask_char("...")  
    ↗  
    goodt_g copy = *g;  
  
    switch (answer) {  
        case 'N':  
        case 'n': copy.name = Ask_namn(); break;  
        // etc.  
        case 'P':  
        case 'p': copy.price = Ask_pris(); break;  
    }  
  
    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"

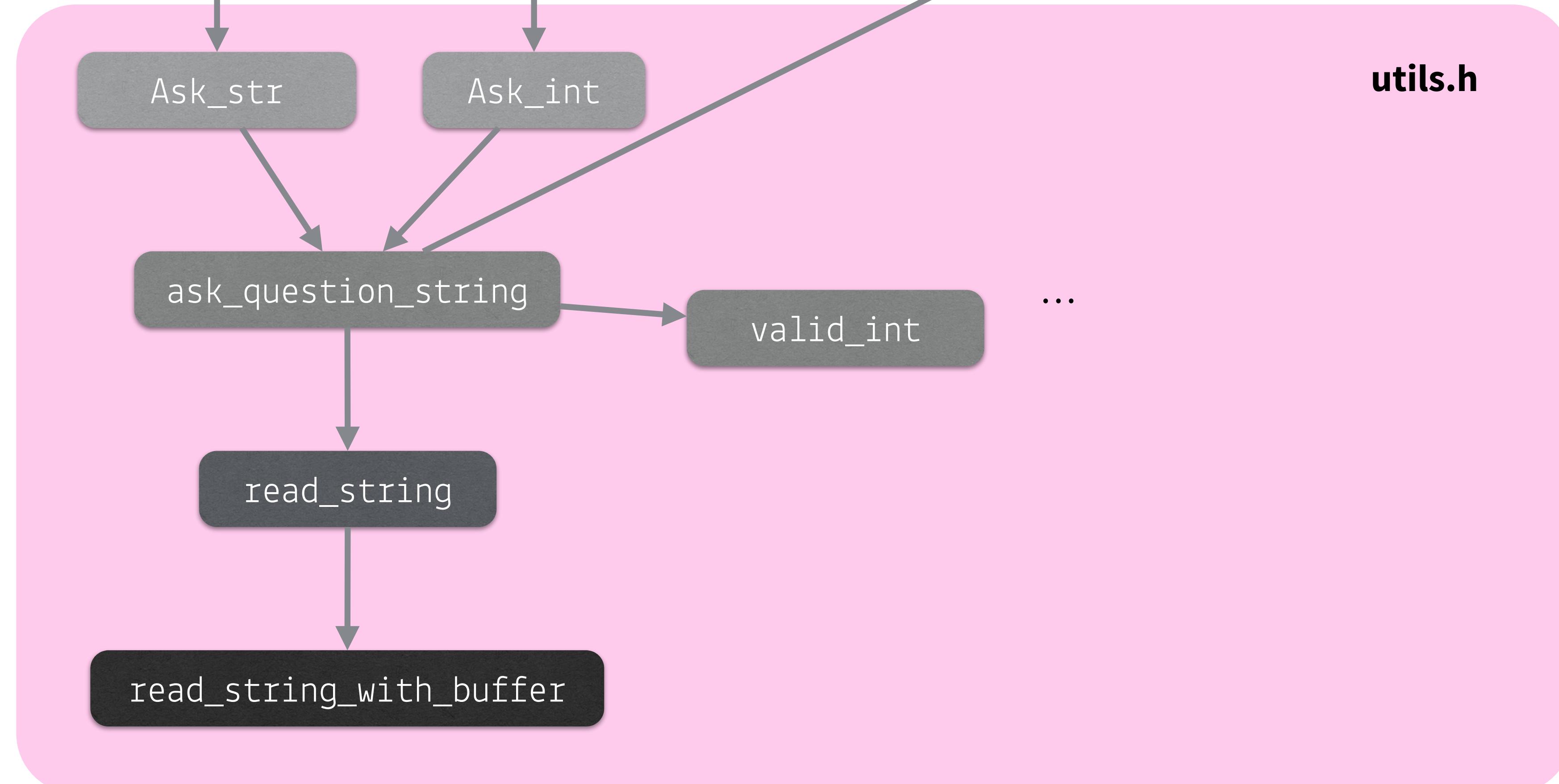
Kanske skulle detta ges en egen
funktion som också anropades i
add_goods?



Specifikt för DB-programmete



Generella rutiner för ”alla” program





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
- Programma nära domänen
- Göm tekniska komplexiteter ”längre ned”

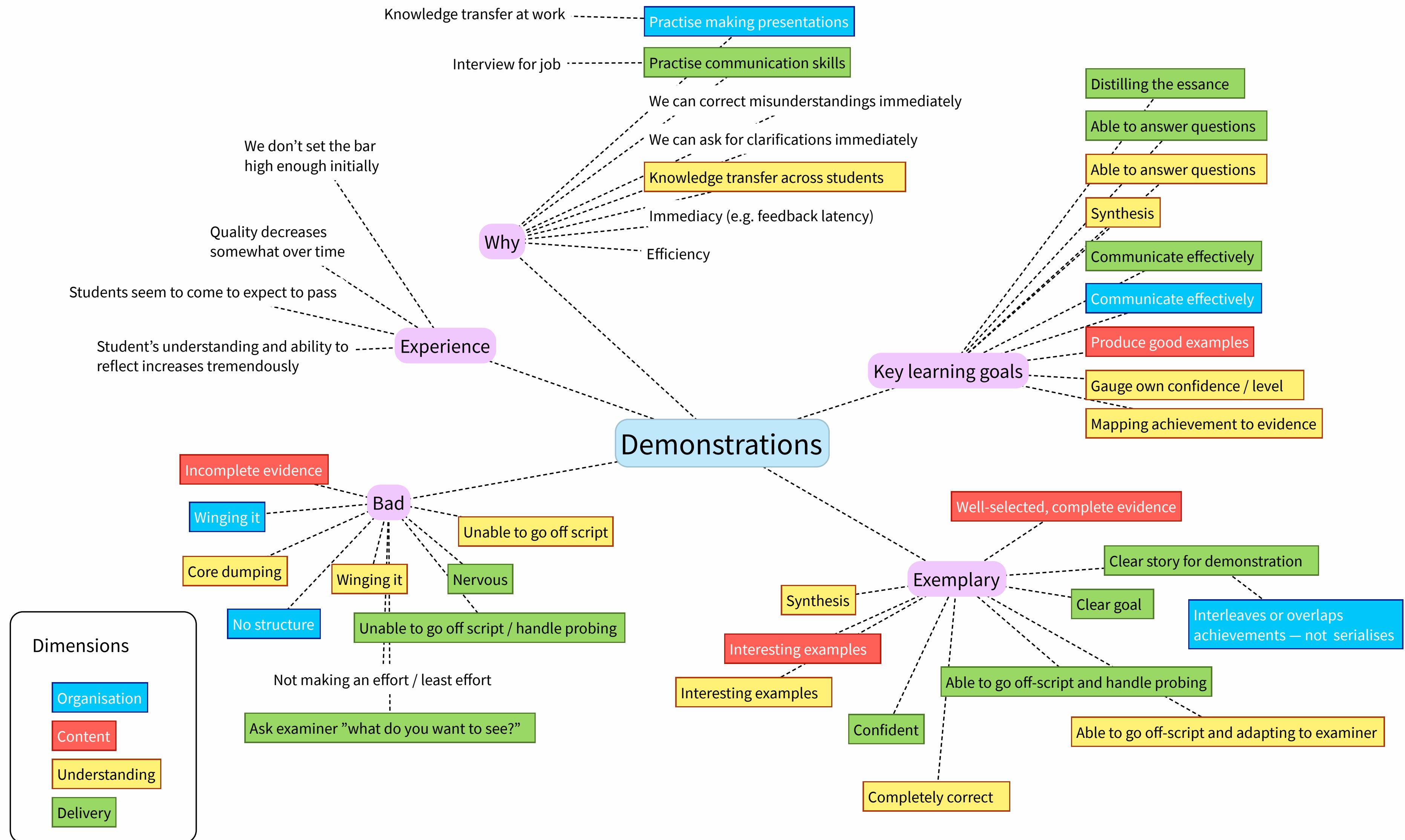
Läsbar kod

- 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.

Demonstrationer på IOOPM





Hur en redovisning skall gå till

- When the examiner arrives, you must pitch your demonstration, which means:
 1. Stating what achievements you wish to demonstrate
 2. Explaining why/how it makes sense to demonstrate them together
 3. Explaining what evidence you will use in the demonstration

If the examiner is not satisfied with 1 or 2, she will explain why and reject the request.

- By evidence, we mostly mean artefacts developed in the course that will form the basis of the demonstration. For example, “we profiled the binary search tree that we developed in Assignment 2”. The key idea with evidence is that you relate the achievements to the programming assignments (and programming project) of the course. Expect to be asked questions that force you to go “off-script”, e.g., what would happen if X is changed for Y, or the examiner asks you to change some lines of code, or introduces a bug etc.
- If the examiner finds the answer to 1–3 satisfactory, she will hand over the running of the demonstration to you! Different examiners approach this task differently. Some will interrupt and ask questions. Some delay all questions to the end.