Effektiv Kode med C og C++

Forelesning 3, vår 2015

Alfred Bratterud

Agenda:

* Ulike skop
* Typer av minne og objekters levetid
* Stack
* Minnelekkasjer
* Referanser

Repetisjon

* Er en peker en datatype?
  * Hvorfor?
  * Hvor mange typer pekere?
* Er pekere trygge? Eksempler?
* Når er de nødvendige?

Minne

Viktig viktig pensum til prøven
...Få det både inn i hodet og i fingrene

Minnetilgang = mye kraft + mye ansvar

* Hva skjer her?

```
void* verySmart(){
        int arr[5000000];
        //Do some stuff.
        return (void*)arr;
}
```

* Initialisering? Allokering? Type? Størrelse?
* Vi trenger noen begreper for å si hva som skjer.

Fem typer skop steder der navn gjelder

* "Global scope" - alt som ikke er med i et annet skop
* "Namespace scope" - definert i et "namespace", men ikke dypere
* "Local scope" - inni en funksjon
* "Class scope" - medlemmer av en klasse
* "Statement scope" - inni et uttrykk, som feks. `while(true){int i=count};`

Fem typer skop

Global scope

Namespace scope

Local scope

Class scope

Statement scope

```cpp
#include <iostream>
using namespace std;

const double inch_cm = 2.54;

namespace metric{

  float to_cm(float inch){
    return inch * inch_cm;
  }
}

class patient{
  float _height_inch = 66;
public:
  patient(float h) : _height_inch(h){};
  float height_cm(){
    return metric::to_cm(_height_inch);
  }
};

int main(){
  patient p(68.2);
  for(int i = 0; i<100; i++){
    if(i>=5){
      float h = p.height_cm();
      cout << h << endl;
    }
  }
}
```

Fem typer skop

Global scope

Namespace scope

Local scope

Class scope

Statement scope

```cpp
#include <iostream>
using namespace std;

const double inch_cm = 2.54;

namespace metric{
  float to_cm(float inch){
    return inch * inch_cm;
  }
}

class patient{
  float _height_inch = 66;
public:
  patient(float h) : _height_inch(h){};
  float height_cm(){
    return metric::to_cm(_height_inch);
  }
};

int main(){
  patient p(68.2);
  for(int i = 0; i<100; i++){
    if(i>=5){
      float h = p.height_cm();
      cout << h << endl;
    }
  }
}
```

Fem typer skop

Global scope

Namespace scope

Local scope

Class scope

Statement scope

```cpp
#include <iostream>
using namespace std;

const double inch_cm = 2.54;

namespace metric{
  float to_cm(float inch){
    return inch * inch_cm;
  }
}

class patient{
  float _height_inch = 66;
public:
  patient(float h) : _height_inch(h){};
  float height_cm(){
    return metric::to_cm(_height_inch);
  }
};

int main(){
  patient p(68.2);
  for(int i = 0; i<100; i++){
    if(i>=5){
      float h = p.height_cm();
      cout << h << endl;
    }
  }
}
```

Navn gjelder i skopet der de ble definert (det innerste) og i alle nøstede skop innenfor

Når vi snakker om minnehåndtering i C++ er vi bare interessert i navn på data. Funksjoner ligger på reserverte steder - vi bryr oss aldri om hvor.

Fem typer skop

Global scope

Namespace scope

Local scope

Class scope

Statement scope

```cpp
#include <iostream>
using namespace std;

const double inch_cm = 2.54;

namespace metric{

  float to_cm(float inch){
    return inch * inch_cm;
  }
}

class patient{
  float _height_inch = 66;
public:
  patient(float h) : _height_inch(h){};
  float height_cm(){
    return metric::to_cm(_height_inch);
  }
};

int main(){
  patient p(68.2);
  for(int i = 0; i<100; i++){
    if(i>=5){
      float h = p.height_cm();
      cout << h << endl;
    }
  }
}
```

Navn gjelder i skopet der de ble definert (det innerste) og i alle nøstede skop innenfor

Når vi snakker om minnehåndtering i C++ er vi bare interessert i navn på data. Funksjoner ligger på reserverte steder - vi bryr oss aldri om hvor.

Tre typer minne «Storage classes»

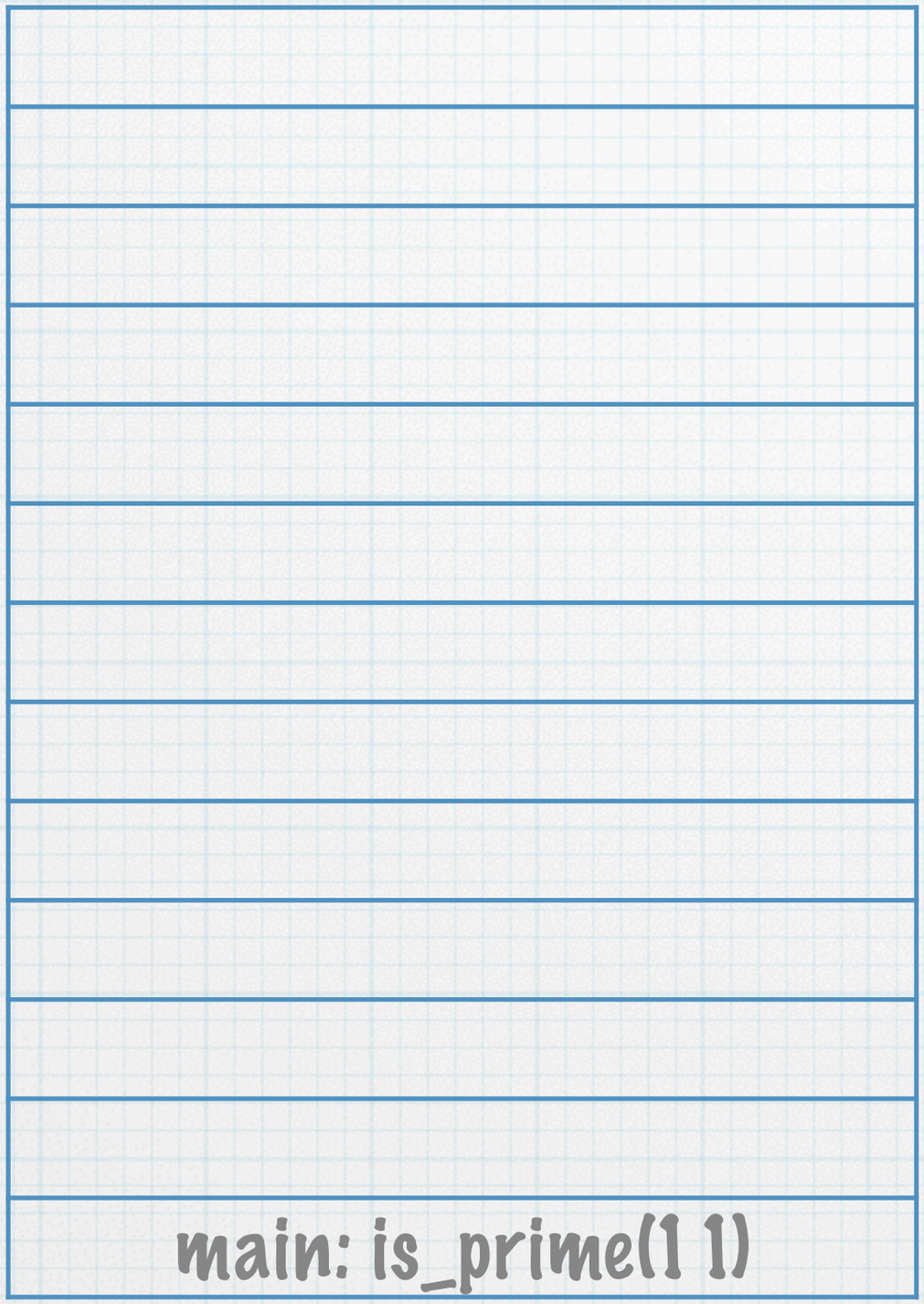
Fra boken §A.4.2 og §A.4.3 (Language summary)

* «Automatic storage»: «Local scope» + «statement scope»
  * > Allokeres - av ditt program - når CPU'en «går inn i deres skop».
  * < Frigjøres automatisk - av ditt program - når de går ut av skop.
    * Kompilator bygger det inn i din kode!
* «Static storage»: data i namespace- og globalt skop + «static»
  * > Allokeres (av OS'et) når prosessen starter
  * < Frigjøres (av OS'et) når prosessen avslutter
* Dynamic storage, "Free store (heap)": data i RAM du styrer
  * > Allokeres når du bruker ordet «new»
  * < Blir borte når DU gir beskjed (evt. når prosessen avslutter)

Automatic Storage

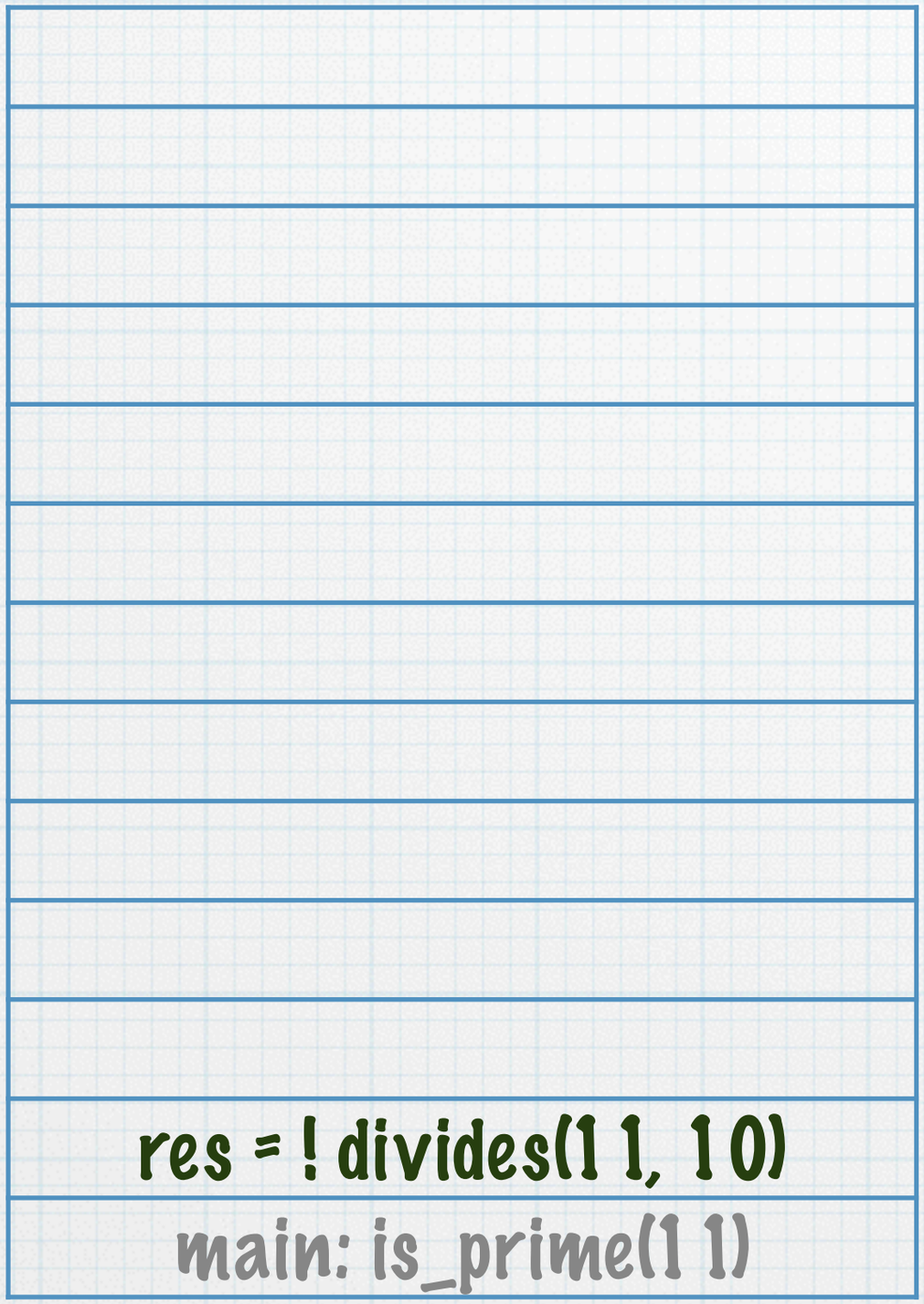
* Kalles gjerne «Stack»- fordi hardware implementerer det som en «stabel»
  * En ny «stack frame» legges på stabelen, pr. funksjonskall
  * Denne «poppes» bort når funksjonen returnerer
    * ...skop inn, skop ut
      * ...Din kode gjør det! Kompilatoren bygger det inn for deg.
* Variable definert inni funksjoner - og parametre - legges her
* Kan gjenbrukes av andre funksjoner, når kallet har returnert
* Mange "kopier" kan eksistere, hvis funksjonen kaller seg selv - eller hvis flere tråder.
  * Alltid (minst) en stack pr. tråd
* Har svært begrenset i størrelse (feks.10 MB), bestemt av OS'et

Stack (Forts.)



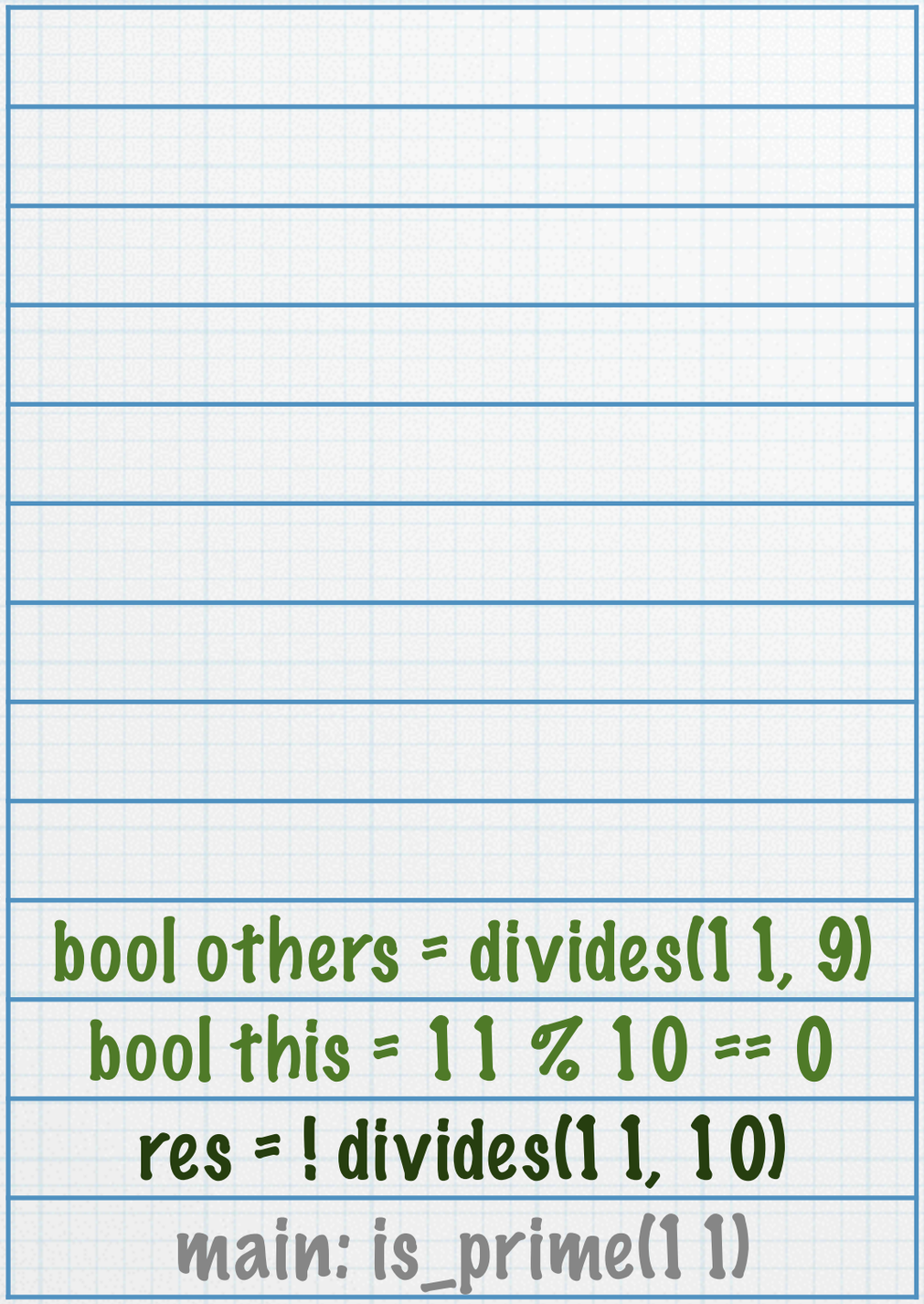
* OS'et leverer en stack til prosessen, som den får skrive til. Main nederst.
* Hver funksjon tar selv ansvar for å sette av nok plass på stack til sine egne "automatiske variabler" (Den flytter bare stack pointer). Kompilatoren fikser dette for oss.
* Hvert funksjonskall dytter sp videre opp...
* Blir det er fullt - Stack overflow!
* Return...? Håper vi lagret forrige kalls stack pointer!
* C++: Stack unwind. Alle destructorer kalles

Stack (Forts.)



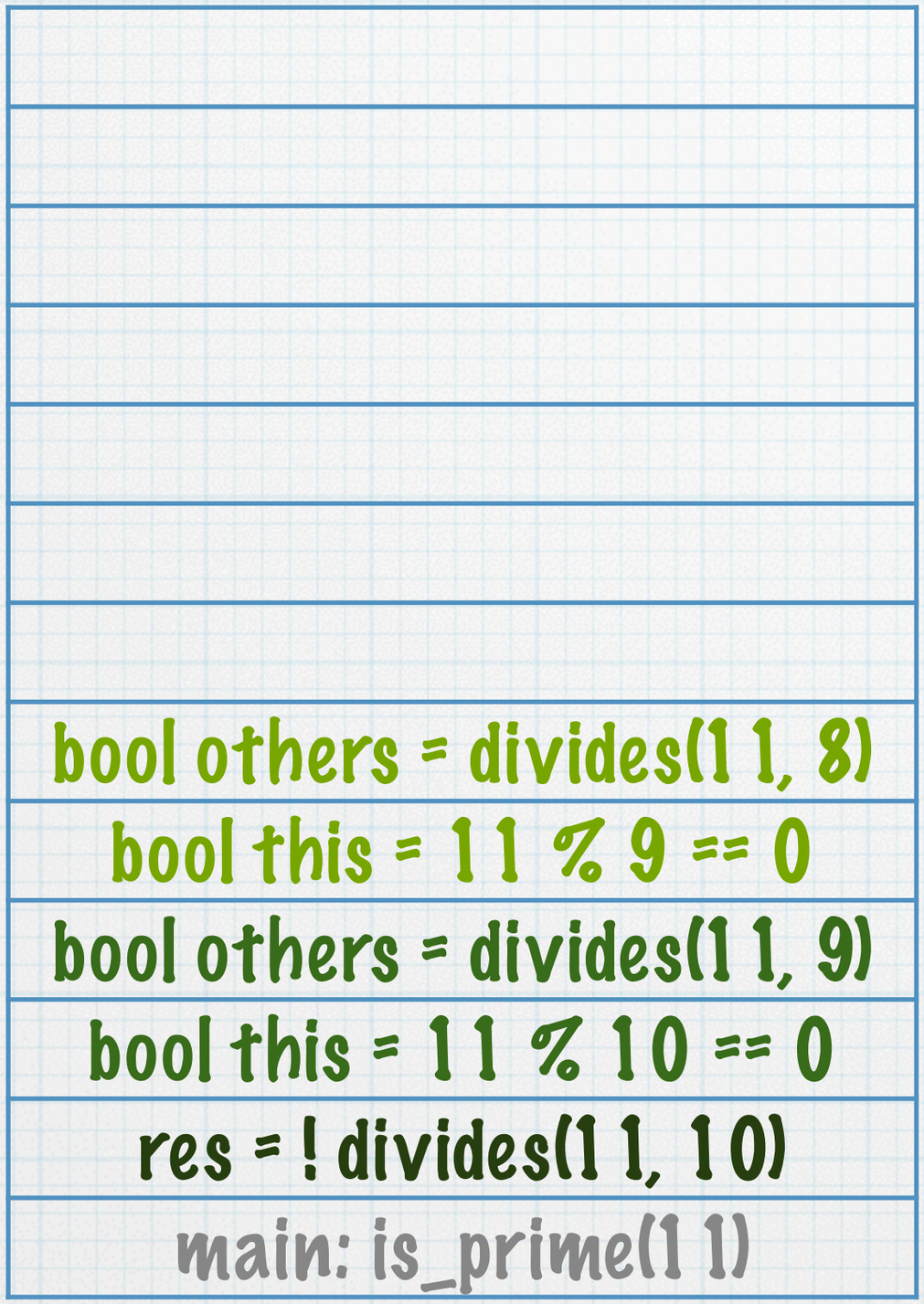
* OS'et leverer en stack til prosessen, som den får skrive til. Main nederst.
* Hver funksjon tar selv ansvar for å sette av nok plass på stack til sine egne "automatiske variabler" (Den flytter bare stack pointer). Kompilatoren fikser dette for oss.
* Hvert funksjonskall dytter sp videre opp...
* Blir det er fullt - Stack overflow!
* Return...? Håper vi lagret forrige kalls stack pointer!
* C++: Stack unwind. Alle destructorer kalles

Stack (Forts.)



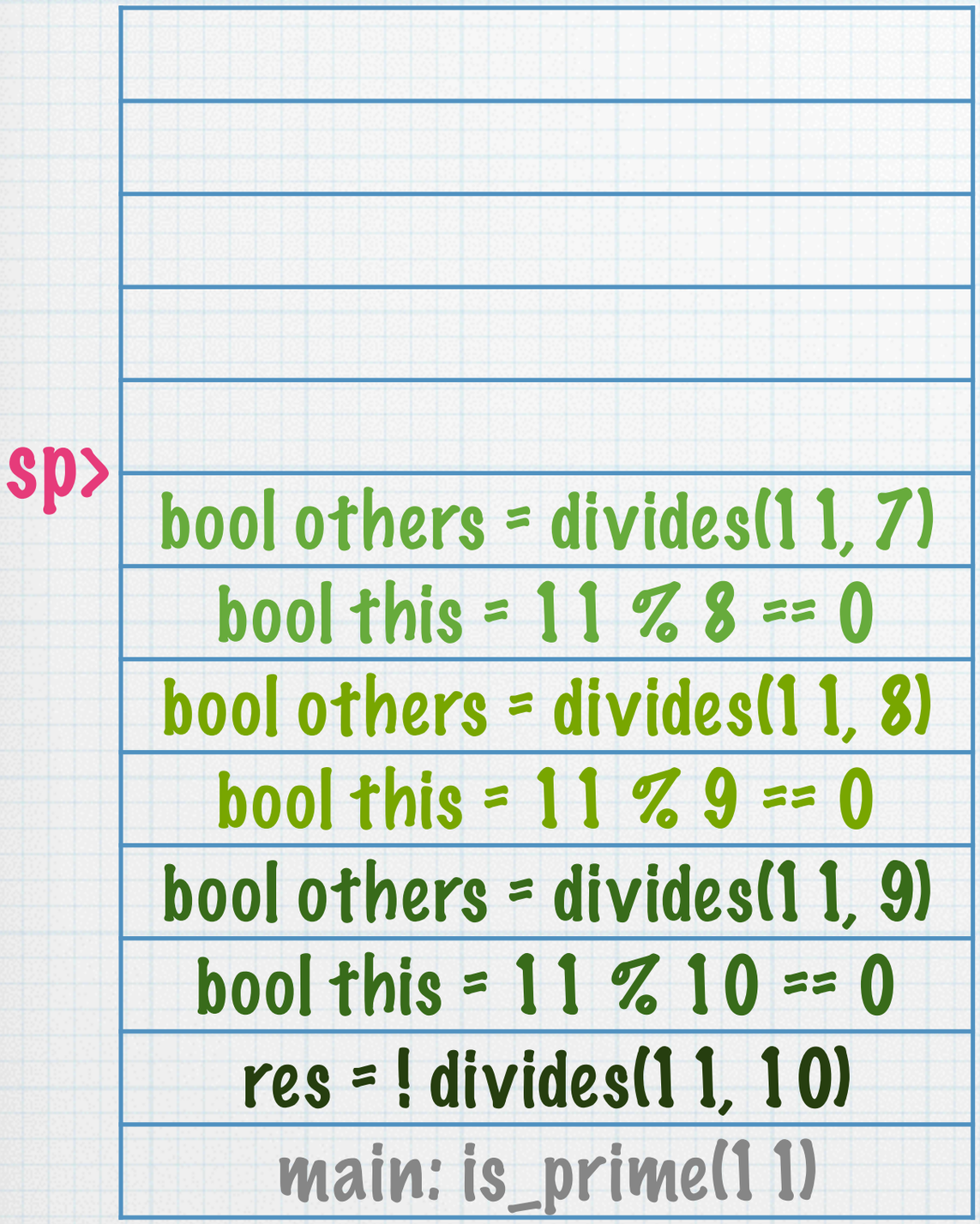
* OS'et leverer en stack til prosessen, som den får skrive til. Main nederst.
* Hver funksjon tar selv ansvar for å sette av nok plass på stack til sine egne "automatiske variabler" (Den flytter bare stack pointer). Kompilatoren fikser dette for oss.
* Hvert funksjonskall dytter sp videre opp...
* Blir det er fullt - Stack overflow!
* Return...? Håper vi lagret forrige kalls stack pointer!
* C++: Stack unwind. Alle destructorer kalles

Stack (Forts.)



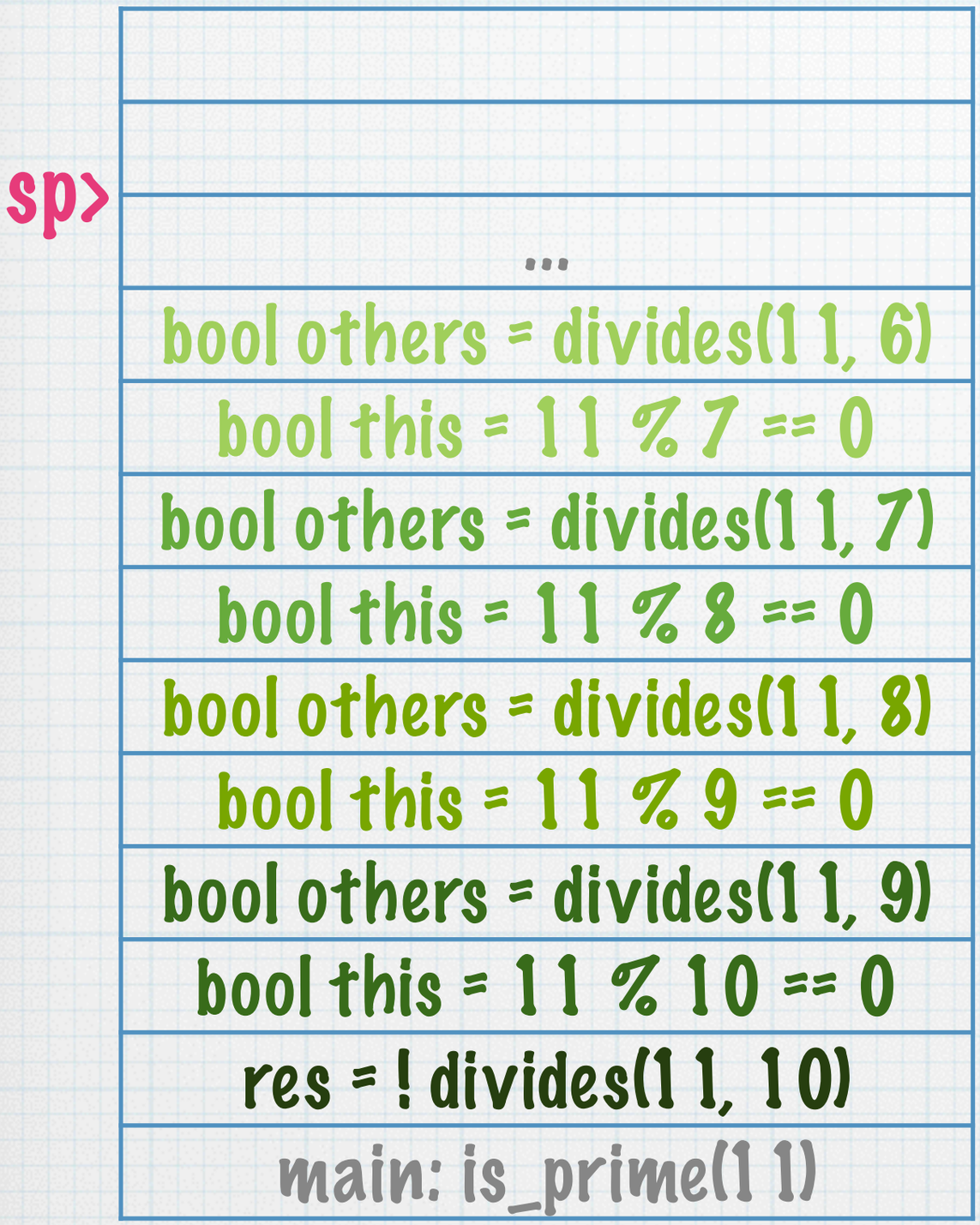
* OS'et leverer en stack til prosessen, som den får skrive til. Main nederst.
* Hver funksjon tar selv ansvar for å sette av nok plass på stack til sine egne "automatiske variabler" (Den flytter bare stack pointer). Kompilatoren fikser dette for oss.
* Hvert funksjonskall dytter sp videre opp...
* Blir det er fullt - Stack overflow!
* Return...? Håper vi lagret forrige kalls stack pointer!
* C++: Stack unwind. Alle destructorer kalles

Stack (Forts.)



* OS'et leverer en stack til prosessen, som den får skrive til. Main nederst.
* Hver funksjon tar selv ansvar for å sette av nok plass på stack til sine egne "automatiske variabler" (Den flytter bare stack pointer). Kompilatoren fikser dette for oss.
* Hvert funksjonskall dytter sp videre opp...
* Blir det er fullt - Stack overflow!
* Return...? Håper vi lagret forrige kalls stack pointer!
* C++: Stack unwind. Alle destructorer kalles

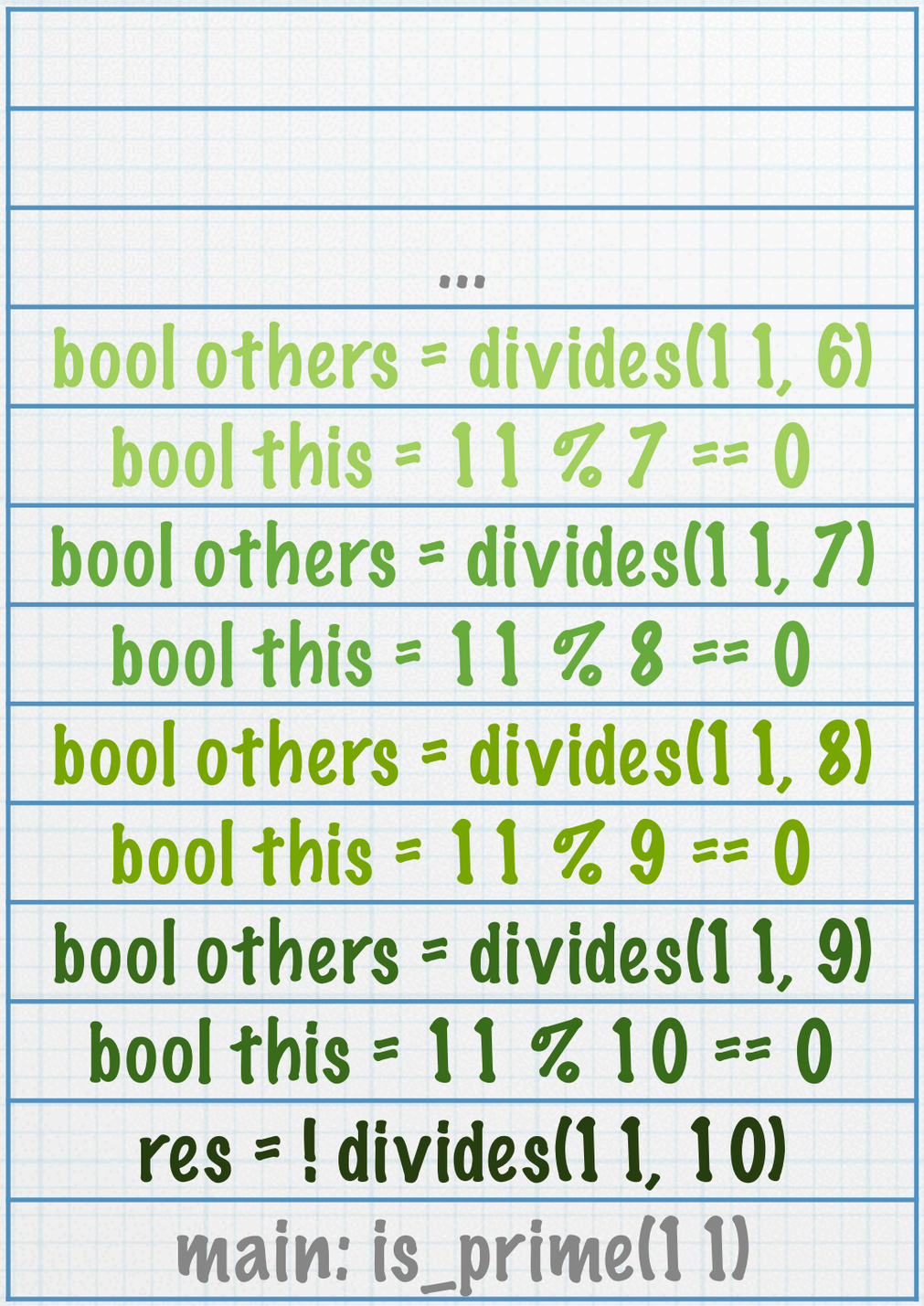
Stack (Forts.)



* OS'et leverer en stack til prosessen, som den får skrive til. Main nederst.
* Hver funksjon tar selv ansvar for å sette av nok plass på stack til sine egne "automatiske variabler" (Den flytter bare stack pointer). Kompilatoren fikser dette for oss.
* Hvert funksjonskall dytter sp videre opp...
* Blir det er fullt - Stack overflow!
* Return...? Håper vi lagret forrige kalls stack pointer!
* C++: Stack unwind. Alle destructorer kalles

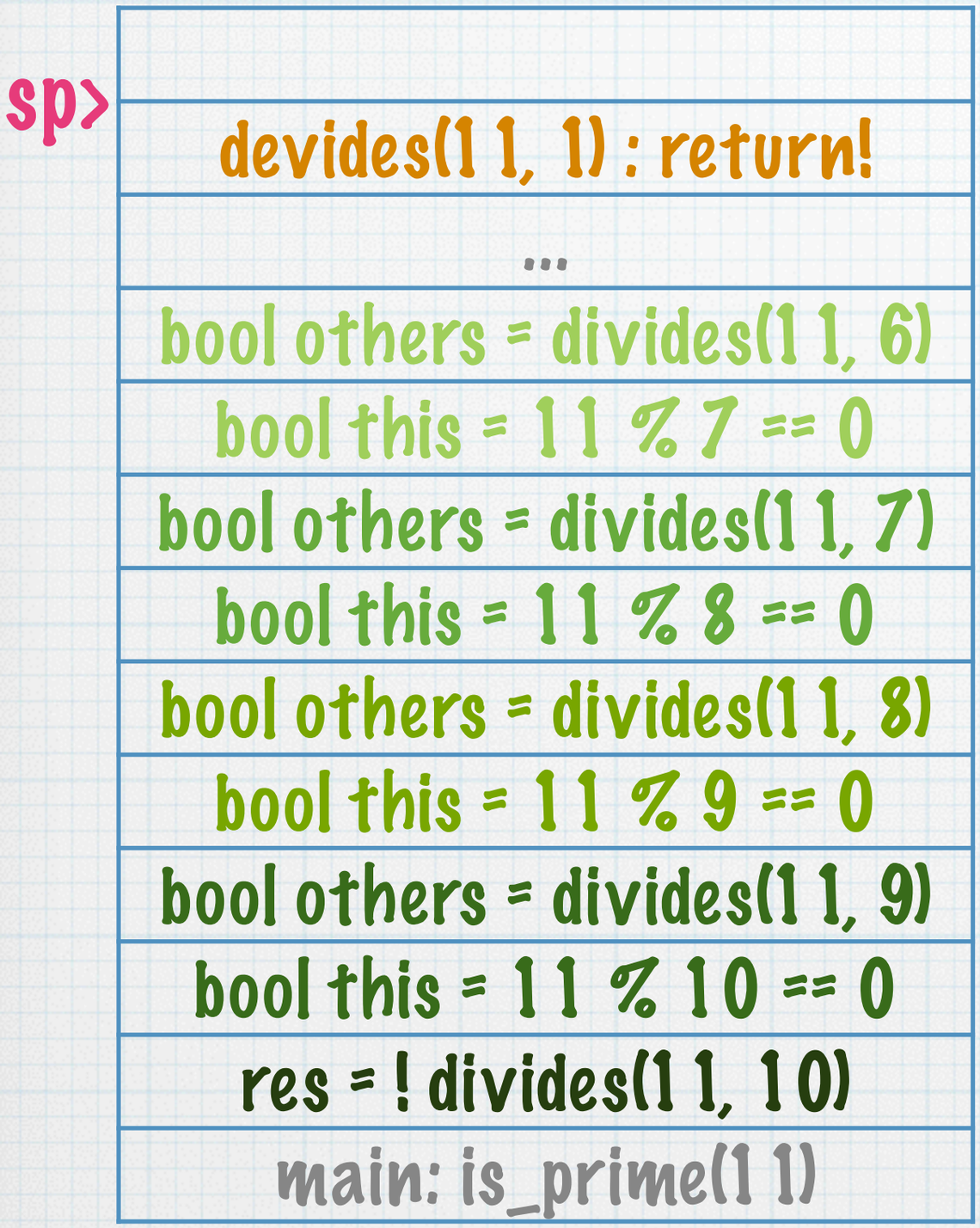
Stack (Forts.)

sp> STACK OVERFLOW!



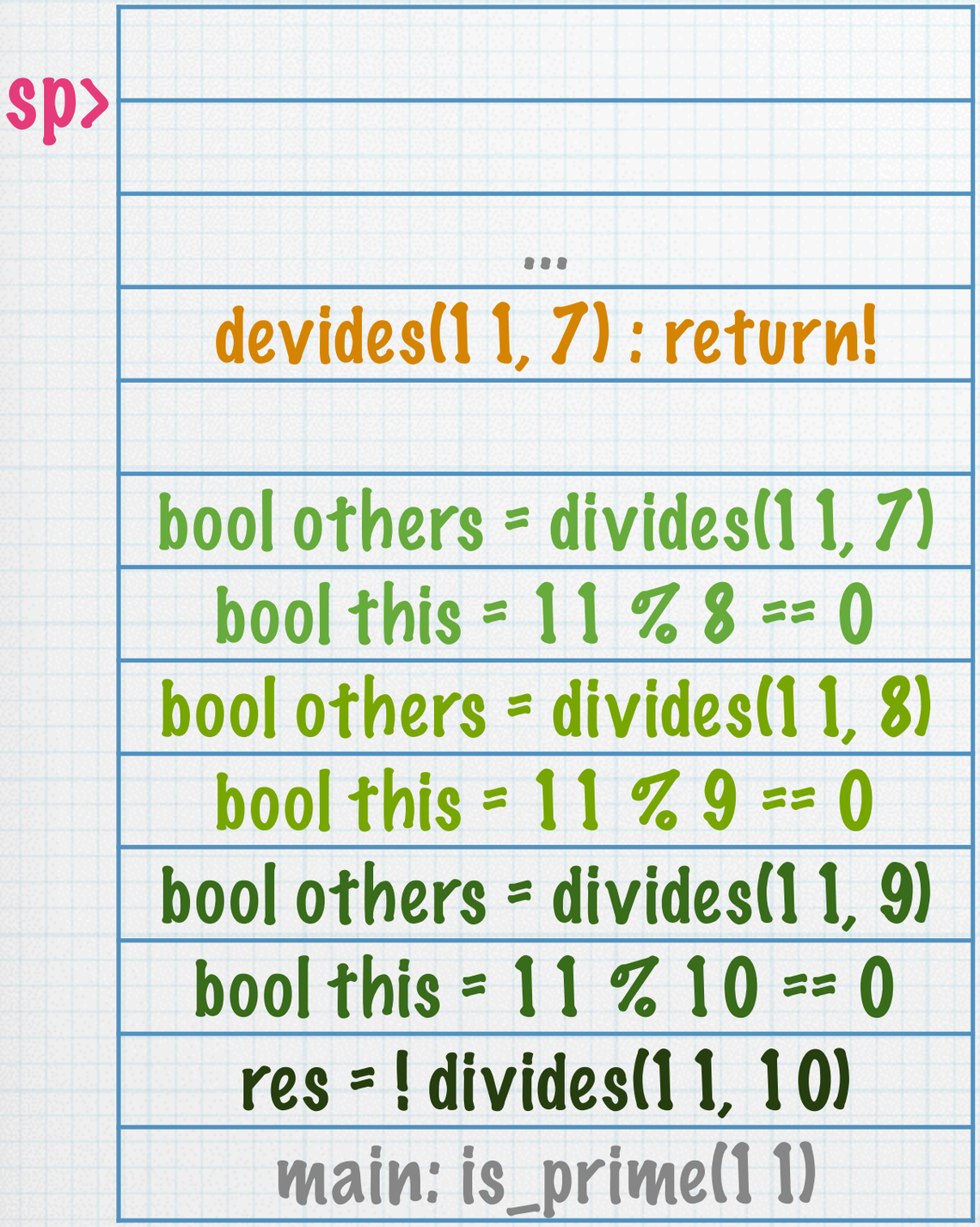
* OS'et leverer en stack til prosessen, som den får skrive til. Main nederst.
* Hver funksjon tar selv ansvar for å sette av nok plass på stack til sine egne "automatiske variabler" (Den flytter bare stack pointer). Kompilatoren fikser dette for oss.
* Hvert funksjonskall dytter sp videre opp...
* Blir det er fullt - Stack overflow!
* Return...? Håper vi lagret forrige kalls stack pointer!
* C++: Stack unwind. Alle destructorer kalles

Stack (Forts.)



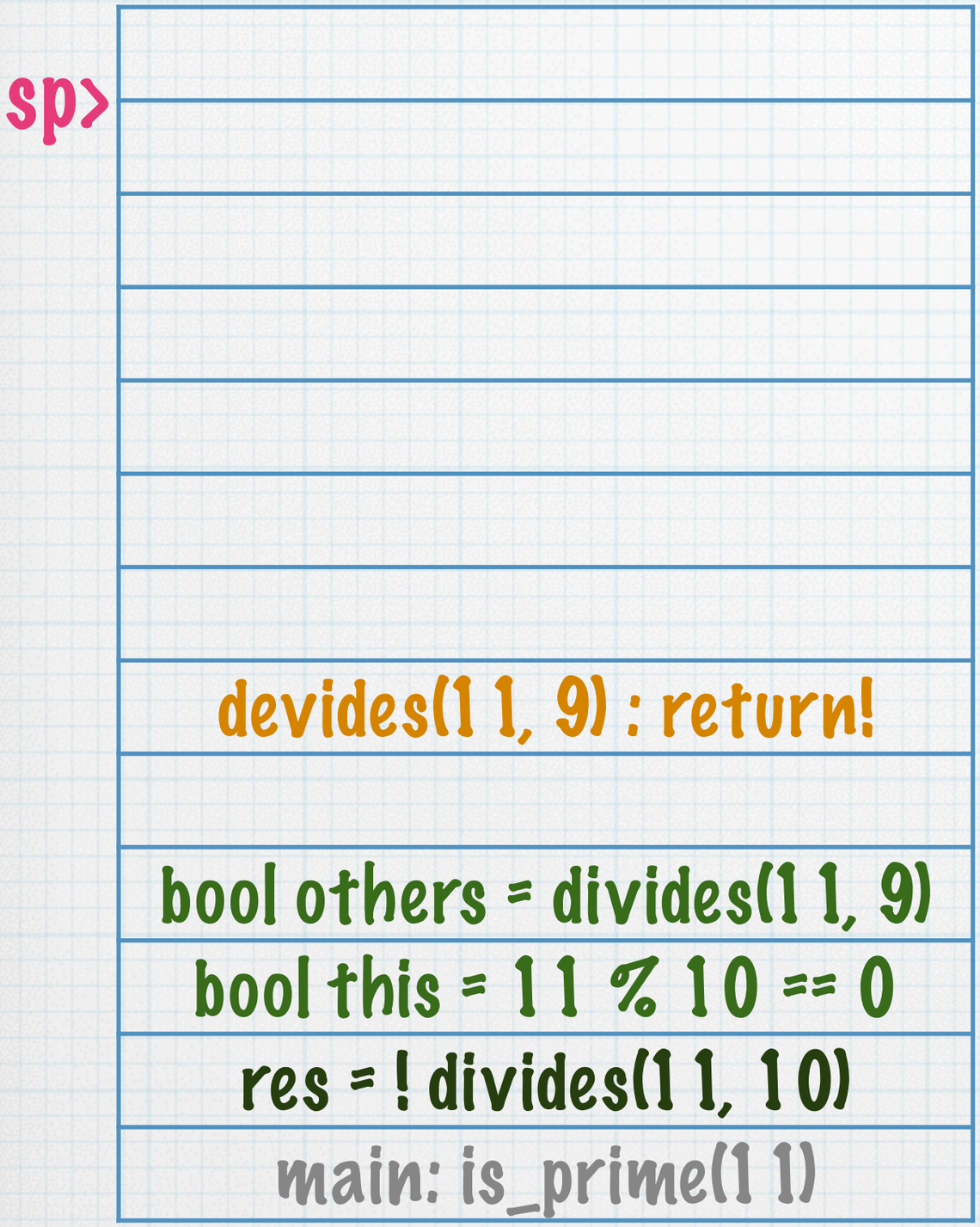
* OS'et leverer en stack til prosessen, som den får skrive til. Main nederst.
* Hver funksjon tar selv ansvar for å sette av nok plass på stack til sine egne "automatiske variabler" (Den flytter bare stack pointer). Kompilatoren fikser dette for oss.
* Hvert funksjonskall dytter sp videre opp...
* Blir det er fullt - Stack overflow!
* Return...? Håper vi lagret forrige kalls stack pointer!
* C++: Stack unwind. Alle destructorer kalles

Stack (Forts.)



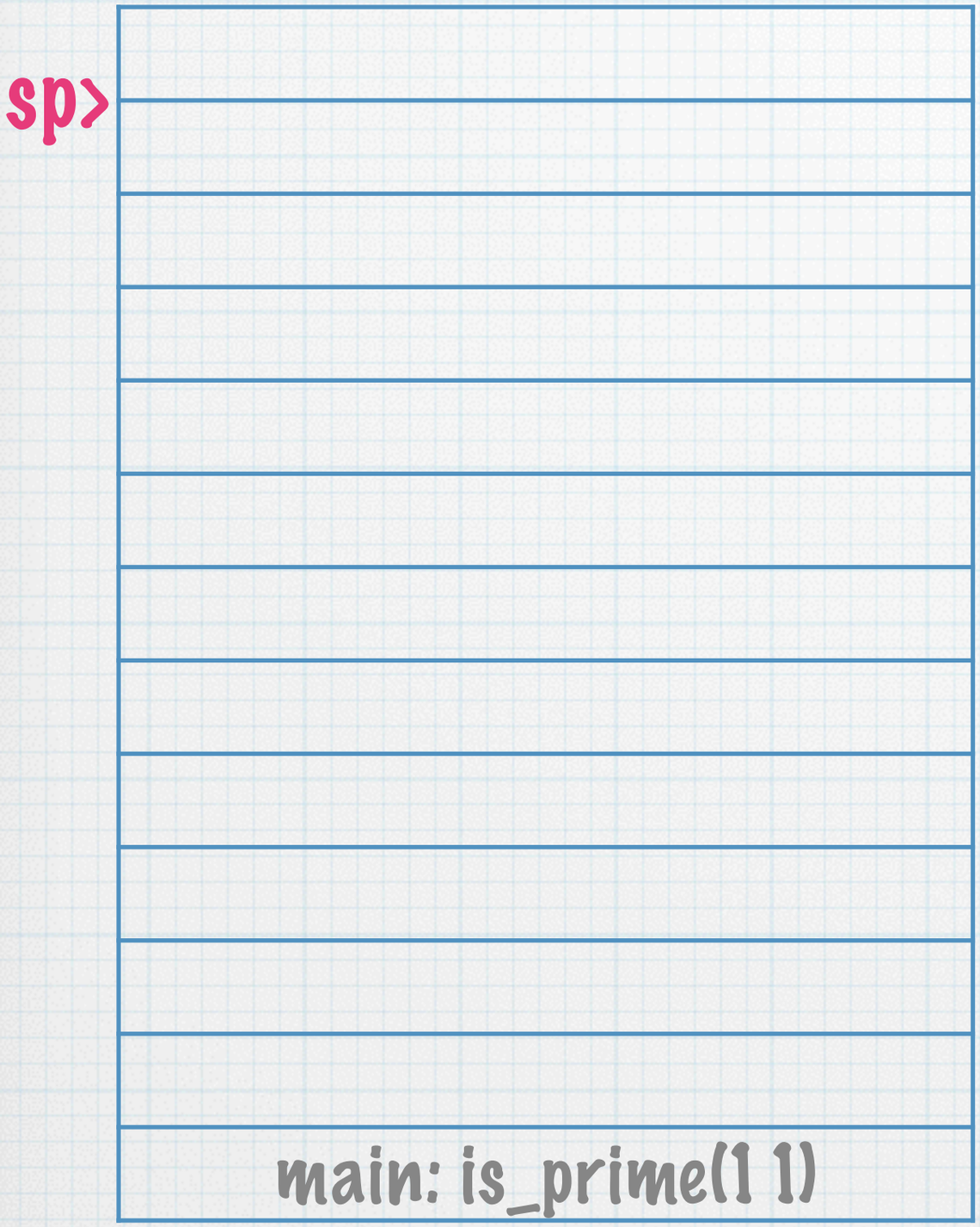
* OS'et leverer en stack til prosessen, som den får skrive til. Main nederst.
* Hver funksjon tar selv ansvar for å sette av nok plass på stack til sine egne "automatiske variabler" (Den flytter bare stack pointer). Kompilatoren fikser dette for oss.
* Hvert funksjonskall dytter sp videre opp...
* Blir det er fullt - Stack overflow!
* Return...? Håper vi lagret forrige kalls stack pointer!
* C++: Stack unwind. Alle destructorer kalles

Stack (Forts.)



* OS'et leverer en stack til prosessen, som den får skrive til. Main nederst.
* Hver funksjon tar selv ansvar for å sette av nok plass på stack til sine egne "automatiske variabler" (Den flytter bare stack pointer). Kompilatoren fikser dette for oss.
* Hvert funksjonskall dytter sp videre opp...
* Blir det er fullt - Stack overflow!
* Return...? Håper vi lagret forrige kalls stack pointer!
* C++: Stack unwind. Alle destructorer kalles

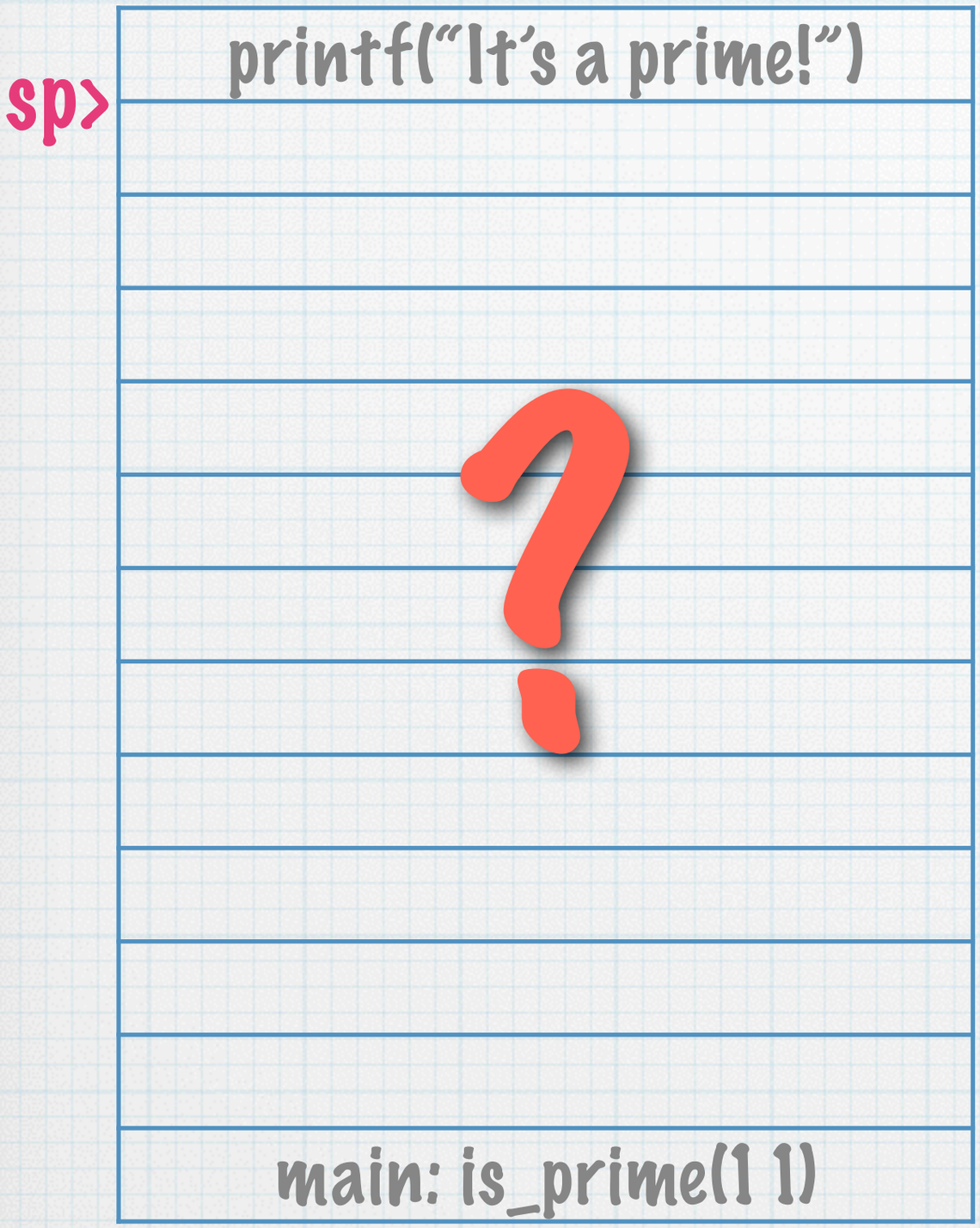
Stack (Forts.)



main: is_prime(1 1)

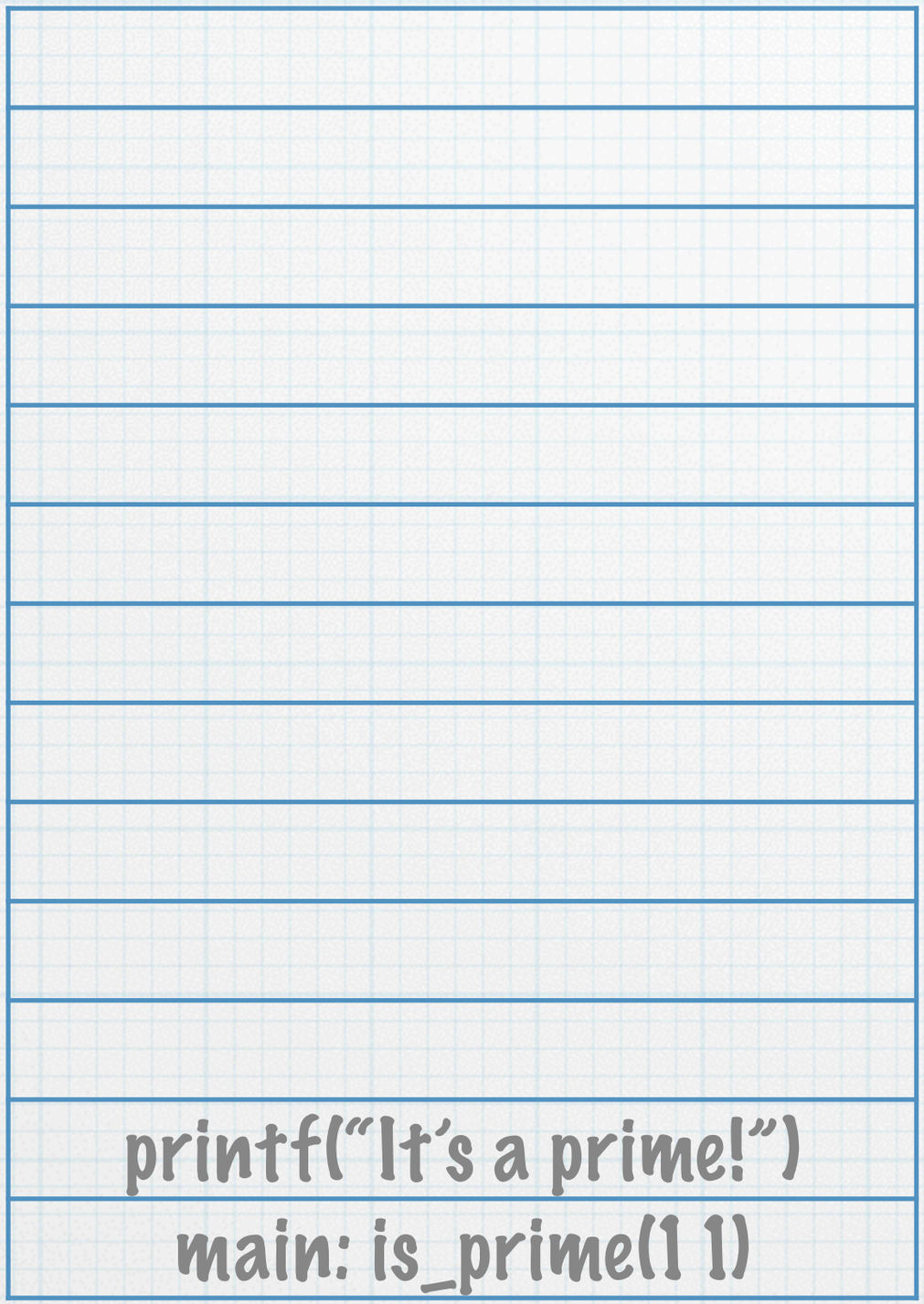
* OS'et leverer en stack til prosessen, som den får skrive til. Main nederst.
* Hver funksjon tar selv ansvar for å sette av nok plass på stack til sine egne "automatiske variabler" (Den flytter bare stack pointer). Kompilatoren fikser dette for oss.
* Hvert funksjonskall dytter sp videre opp...
* Blir det er fullt - Stack overflow!
* Return...? Håper vi lagret forrige kalls stack pointer!
* C++: Stack unwind. Alle destructorer kalles

Stack (Forts.)



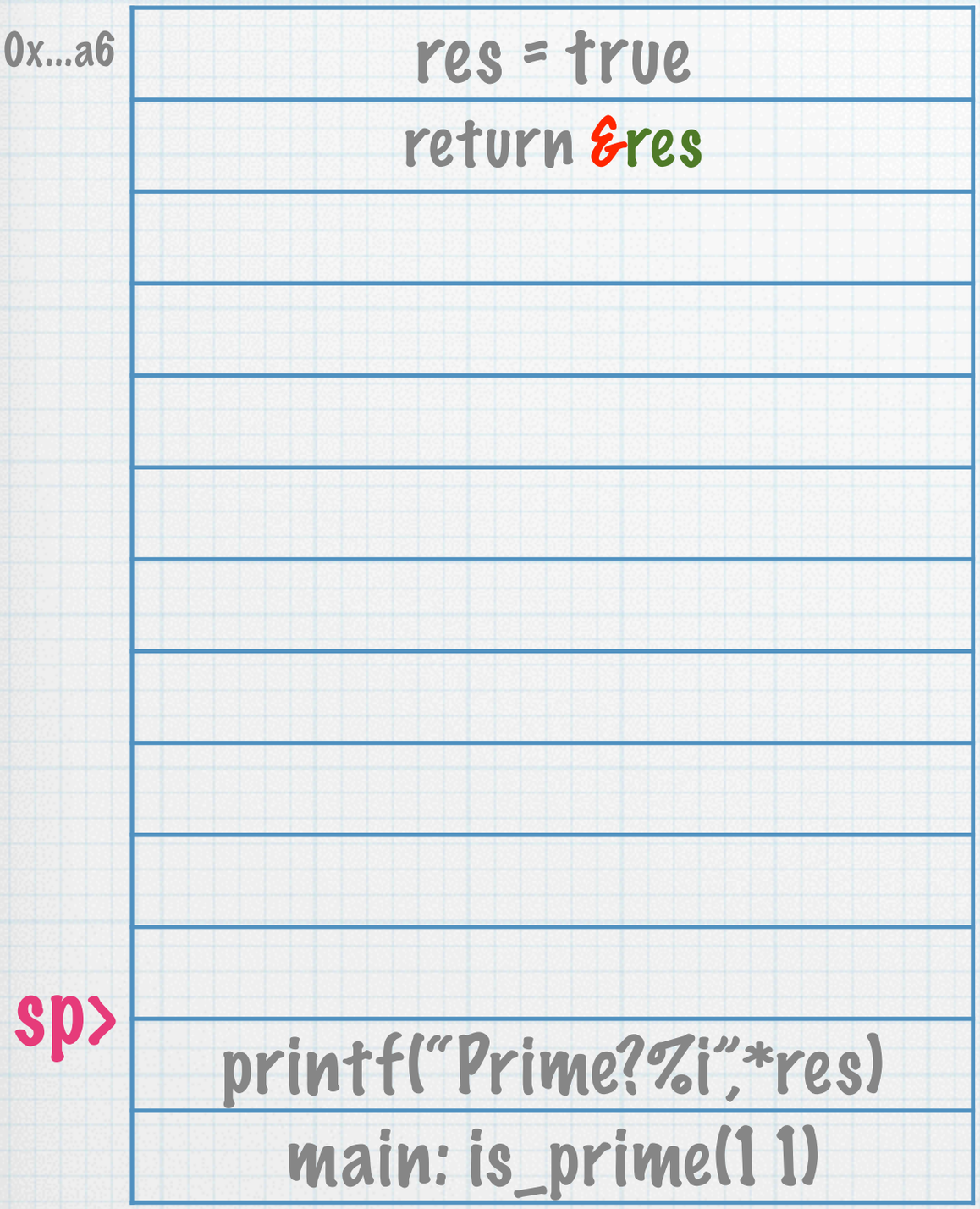
* Vi har glemt å flytte stack-pointer... Gjør det noe?
* Ja - ellers vokser stacken evig
* Heldigvis husker programmet alltid hvor forrige kalls stack-pointer var (ligger i «base-pointer»)
* Det å «returnere» fra en funksjon betyr nettopp å flytte stack-pointer tilbake til «base-pointer», og så «hoppe» til koden som kjørte sist (adressen ligger på stack)
* Returverdi ligger gjerne i register A (eax), argumentene sendes på stack.

Stack (Forts.)



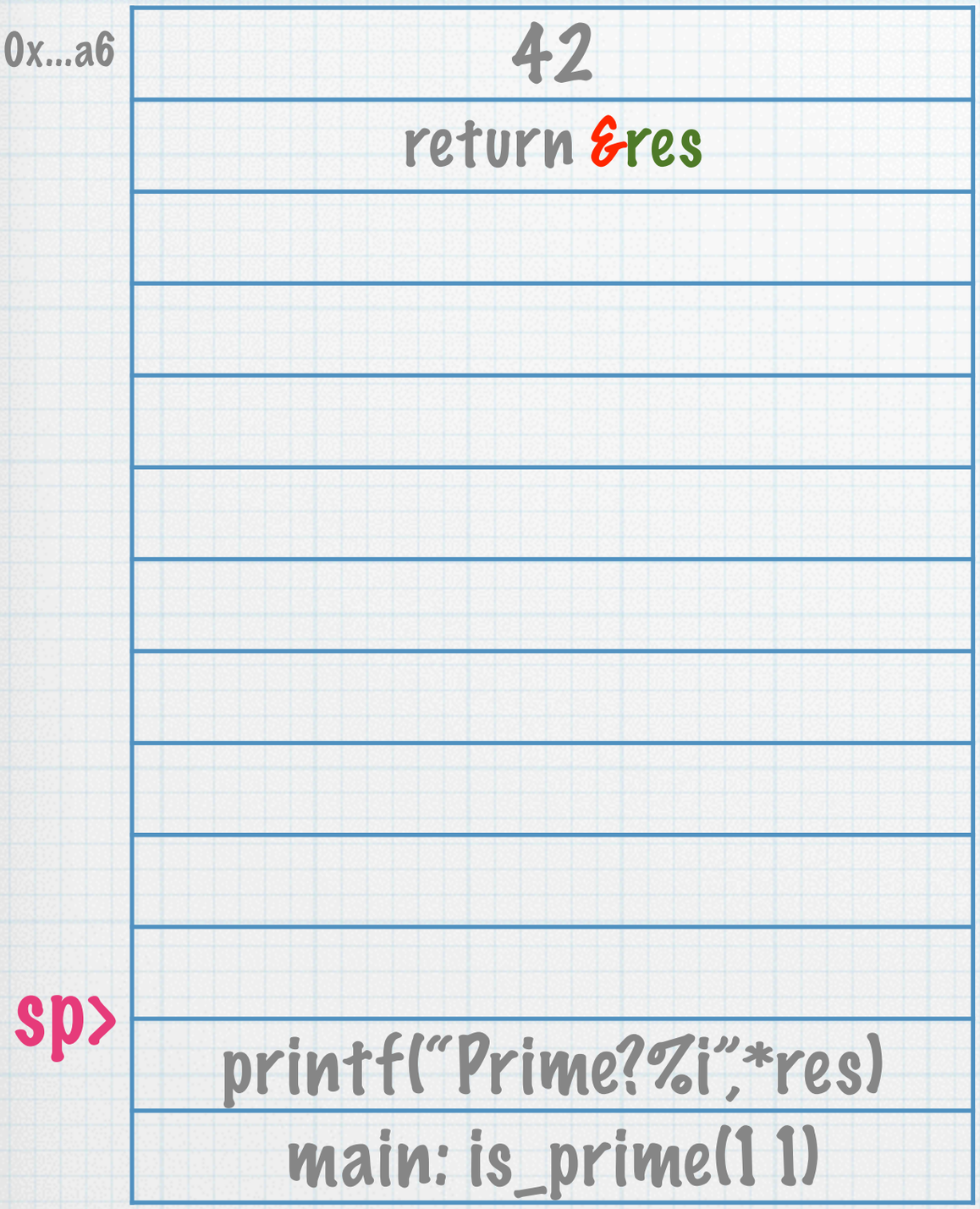
* Stack flyttet tilbake, resultater returnert.
* ...Men det ble mye kopiering på veien ned da?!
  * Hva med å bare returnere en peker?

Stack (Forts.)



* Returnerer man pekere, til lokale variable...
* kan man ha flaks

Stack (Forts.)



* Returnerer man pekere, til lokale variable...
* kan man ha flaks
* ...med mindre det har skjedd en ny serie av funksjonskall i mellomtiden

Automatic Storage (forts.)

* Kalles gjerne "Stack" - men det sier ikke alt
* Inkluderer all minnehåndtering **kompilatoren** kan gjøre for oss, (som å flytte **sp**), men også å lagre mellomregninger, som `i=10*sqrt(2)+5`
* Er svært begrenset i størrelse (feks.10 MB). Hvor stor er den hos deg? Prøv!
* Hvilken feilmelding vil jeg få hvis jeg skriver over stacken? Prøv!

Static Storage

* Det som er deklarert i "global scope" og "namespace scope" ligger her
* Kan ikke endre størrelse- må settes "link-time", før kjøring
* Variable i klasser og funksjoner som er eksplisitt definert "static" ligger også her

Dynamic memory / Free Store (heap)

* Det som er allokert med "new" (C++) legges i free store
    * Må frigjøres med "delete"
* Det som er allokert med "malloc" (C) legges på heap. Kan være samme sted, kan være annet.
    * Må frigjøres med "free"
* Dette er eneste måten å gjøre "manuell" "dynamisk allokering" på, dvs. "runtime".

Free Store (forts.)

* malloc(n) returnerer en void* til n bytes utenfor stack, i "trygt område".... bra?
  * Det tvinger oss til å gjøre en eksplisitt cast (vi kan ikke bruke void* i beregninger)
  * Vi må (noen ganger) huske hvor stor n var
* new er typet, og returnerer peker av riktig type - og dermed riktig størrelse. (int* x=new int(9))
* Men: begge lagrer data i "free store", og DU må ta ansvar for at dette ryddes opp. Ellers?

Minnelekkasjer

* ALDRI kall **malloc** (punktum, spesielt ikke) uten **free**
* ALDRI kall **new** uten **delete**
* I C++ trenger du (nesten) bare å bruke new, når du lager datastrukturer
  (og når du bruker C-biblioteker)
  * STL-containere kan opprettes fritt, uten å tenke på dette. De gjør det for deg.
* Usikker på om du har en lekkasje? Installer og kjør programmet ditt med valgrind: http://valgrind.org/

Nå: Demo!

stackoverflow.cpp
memleak.cpp