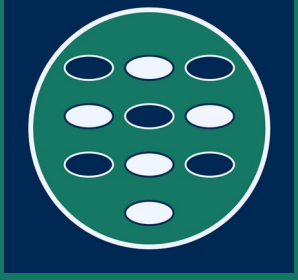
Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap



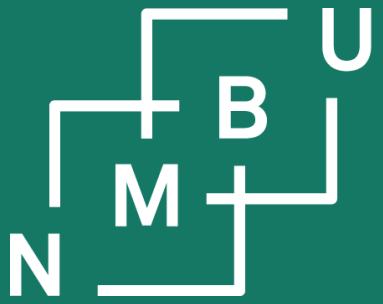
Digitalisering på Ås

INF205

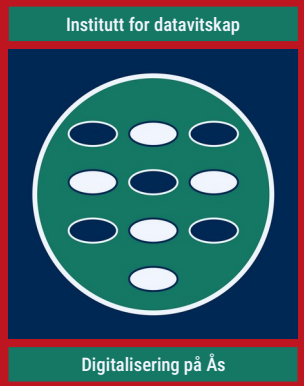
Ressurseffektiv programmering

2 Minnehandtering

2.1 Kallstakken og haugen



Noregs miljø- og biovitskaplege universitet



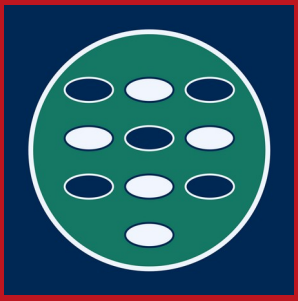
Eksempelproblem

INF205

13. februar 2026

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap



Digitalisering på Ås

2 Minnehandtering

2.1 Kallstakken og haugen

INF205

13. februar 2026

Ordliste

Allocation	Allokering
Call stack	Kallstakk
Deallocation	Deallokering
Garbage collection	Søppeltømming
Heap	Haug
Stack frame	Stakkramme

Memory on the stack vs. memory on the heap

Allocation: Reserve memory to store data.
Deallocation: Release the memory.

On the stack

The stack is already handled completely and safely by the compiler. **Memory on the stack** (local variables of functions) is **allocated** as part of a **stack frame when the function is called**. It is **deallocated** again **when the function returns**.

On the heap

Memory on the heap is managed independent of the stack, at runtime, subject to **explicit allocation and deallocation** instructions that must come from the programmer. There is no garbage collection in C++!

initialization to *i = 42

- **Allocation** is done with **new**. Example: **int* i = new int(42);**
- **Deallocation** is done with **delete**. Example: **delete i;**

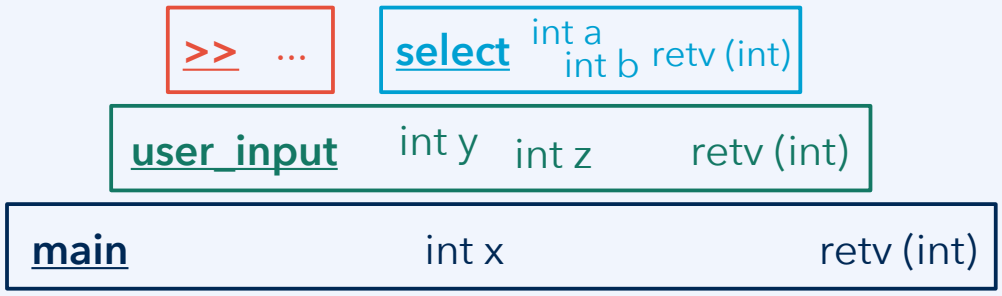
Functions and their stack frames

Stack-like memory management

When a function is called, a known amount of memory must be allocated for its variables (including parameters) "on top of the stack."

When the function returns, its memory can be released; the calling method and its variables become the top of the stack again.

The lifetime of local variables in a **stack frame** is limited to the function's runtime.




```
int select(int a, int b)
{
  if(a%2 == 0) return a;
  else return b;
}

int user_input()
{
  int y = 0, z = 0;
  std::cin >> y >> z;
  return select(y, z);
}

int main()
{
  int x = user_input();
}
```

Example file: three-functions.cpp; compile with "g++ -g3 -o …" and run using gdb.

Observations: Stack

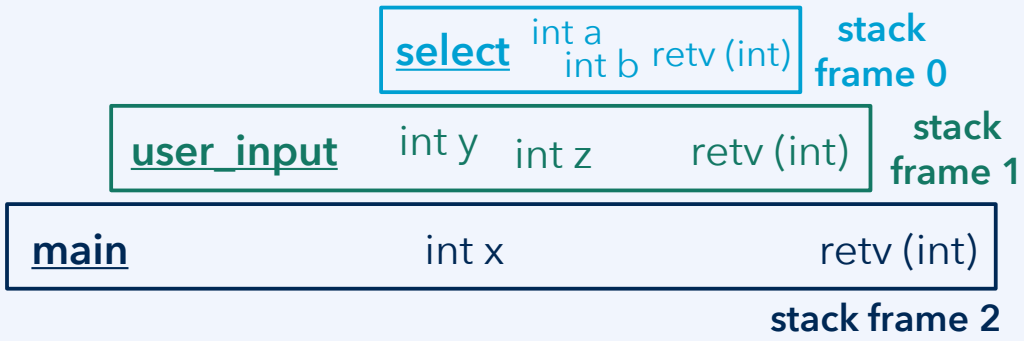
Backtrace and stack inspection using gdb

- Compile with "-g" or "-g3" option
- gdb three-functions
  - break three-functions.cpp:**6**
  - run

```
Breakpoint 1, select (a=4, b=3) at three-functions.cpp:6
6       if(a%2 == 0) return a;
```

  - bt ["backtrace"]

```
#0  select (a=4, b=3) at three-functions.cpp:6
#1  [...] user_input () at three-functions.cpp:14
#2  [...] main () at three-functions.cpp:19
```




```
1
2
3
4  int select(int a, int b)
5  {
6    if(a%2 == 0) return a;
7    else return b;
8  }
9
10 int user_input()
11 {
12   int y = 0, z = 0;
13   std::cin >> y >> z;
14   return select(y, z);
15 }
16
17 int main()
18 {
19   int x = user_input();
20 }
```

Summary: Allocation & deallocation (pointers)

How do we declare a pointer?

- Like any other variable. Its type is a pointer type; *e.g.*, int* my_int_pointer;

How do we initialize a pointer?

- Initialize to **nullptr** (pointer version of 0): int* my_int_pointer = **nullptr**;
- Initialize to **another variable's address**: int* my_int_pointer = &my_index;
- **Allocate memory** on the heap: int* my_int_pointer = **new** int(0);

How do we deallocate a variable if it is stored on the heap?

- Delete the pointer to it. Example: b = **new** BookIndex; …; **delete** b;

How to release the memory if it is a local variable that is stored on the stack?

- Don't do that! You can only call "delete" on memory allocated with "new".

What if we call **new**, but there is not enough free memory left on the system?

- **new** VeryBigObject may throw an exception (a high-level construct).
- **new(std::nothrow)** VeryBigObject may return **nullptr** (low-level construct).

Summary: Allocation & deallocation (arrays)

How do we declare a array?

- Give the size as constant expression in square brackets; *e.g.*, int values[6];
- Also possible: Just declare a pointer; *e.g.*, int* values;

How do we initialize an array?

- Explicitly give all the values: int values[ ] = {4, 2, 3, -7, 2, 3};
- Initialize to **all zeroes**, indicating the array size: int values[6] = { };
- **Allocate memory** with **default initialization**: int* values = **new** int[6]**()**;

How do we deallocate an array if it is stored on the heap?

- Use **delete[]**. Example: b = **new** BookIndex[100]**()**; …; **delete[]** b;
- **Pitfall:** If you use **delete** instead of **delete[]**, only b[0] will be deallocated!

What if we call **new**, but there is not enough free memory left on the system?

- **new** BigObject[100000]**()** may throw an exception.
- **new(std::nothrow)** BigObject[100000]**()** may return **nullptr**.

Glossar

Allokering

Definisjon: det å reservere minne på haugen for eit dataelement/objekt, datatabell eller annan datastruktur

- Vi kan òg seia at minnet på kallstakken blir «allokert», men det er alltid kompilatoren som tek vare på det;
- når programmeraren allokerer minne gjennom **new**, er det på haugen.

Glossar

Haug

Definisjon: minneområde der data blir allokert dynamisk, utan direkte tilknyting til kallstakken

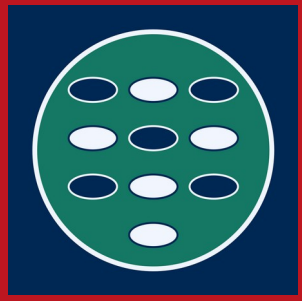
Glossar

Stakkramme

Definisjon: element av kallstakken, som inneheld alle parametrane og dei lokale variablane til ein funksjon

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap



Digitalisering på Ås

Samandrag / diskusjon

INF205

13. februar 2026

Fast copying

Example: **std-copy.cpp**

Element-wise copying

```
for(int i = 0;  i < num_copy;  i++)  target[i] = source[idx_start + i];
```

This is slow! Don't do this for large numbers of elements adjacent in memory. Also, note that Core Guidelines recommend "size_t" instead of int.

C-style fast copying (apply this only to a traditional C/C++ array)

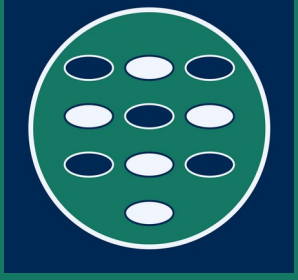
```
#include <cstring>
…
std::memcpy(target,  source + idx_start,  num_copy * sizeof(element_type));
```

Modern C++ style fast copying (can also be used for STL containers)

```
#include <algorithm>
…
std::copy(source + idx_start,  source + idx_start + num_copy,  target);
```

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap



Digitalisering på Ås

INF205

Ressurseffektiv programmering

2 Minnehandtering

2.1 Kallstakken og haugen