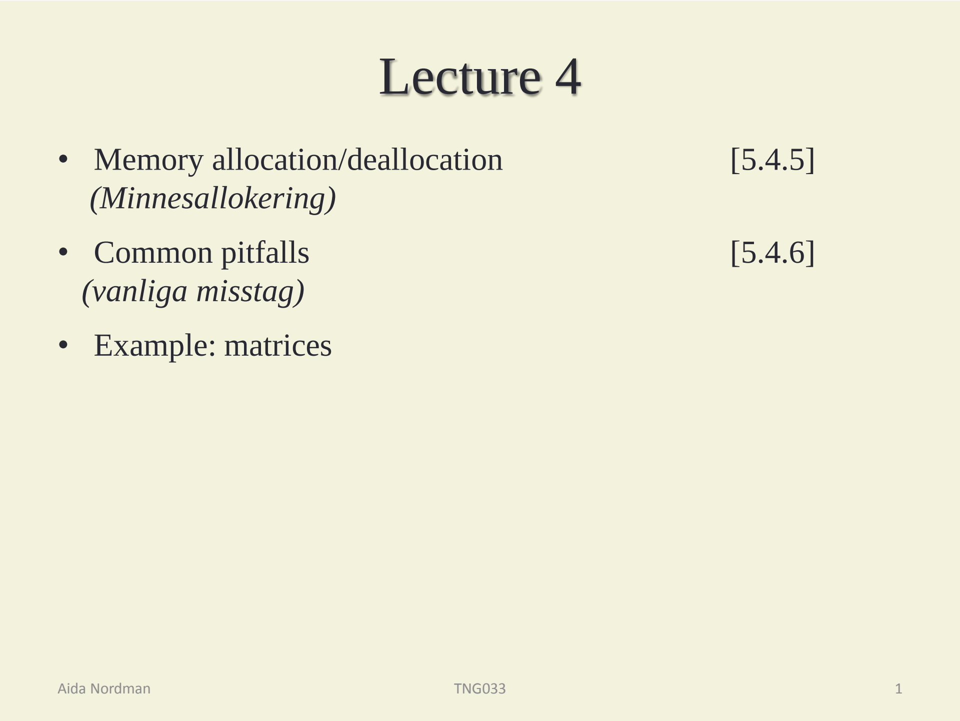
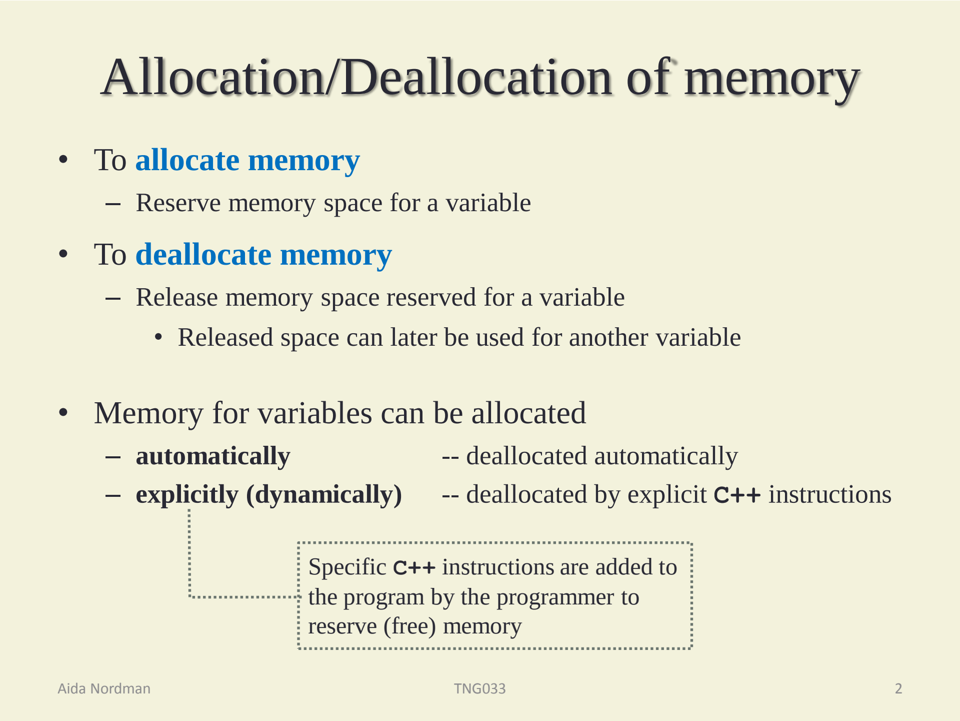


Lecture 4

- Memory allocation/deallocation [5.4.5]
  (Minnesallokering)
- Common pitfalls [5.4.6]
  (vanliga misstag)
- Example: matrices



Allocation/Deallocation of memory

- To **allocate memory**
  - Reserve memory space for a variable
- To **deallocate memory**
  - Release memory space reserved for a variable
    - Released space can later be used for another variable
- Memory for variables can be allocated
  - **automatically** -- deallocated automatically
  - **explicitly (dynamically)** -- deallocated by explicit **C++** instructions

Specific **C++** instructions are added to the program by the programmer to reserve (free) memory

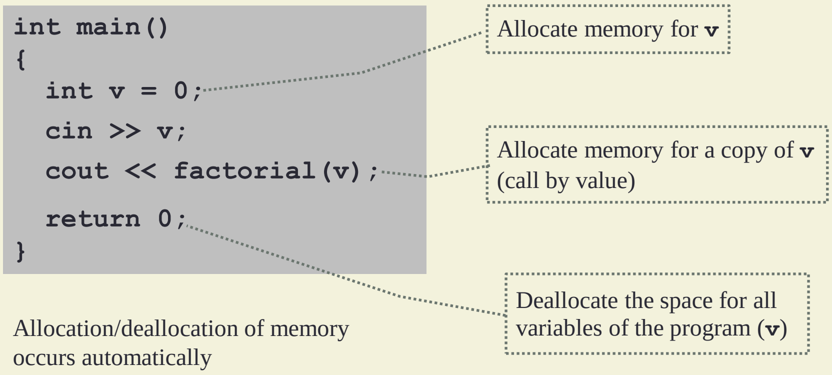
Automatic memory allocation/deallocation

- Memory for the *usual* variables is allocated/deallocate automatically
  - Local variables
  - Functions parameters
  - Variables declared in the `main()`

```
int age = 5;
```

Programmer just declares a variable and the compiler automatically reserves memory for the variabel

Automatic memory allocation/deallocation



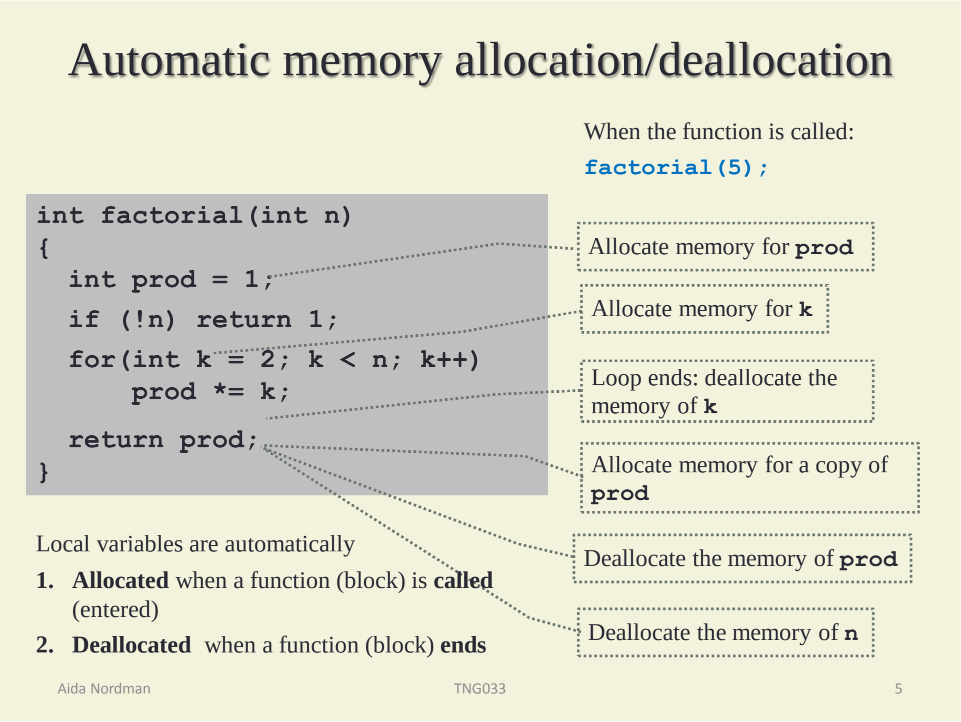
```
int main()
{
  int v = 0;
  cin >> v;
  cout << factorial(v);

  return 0;
}
```

Allocate memory for **v**

Allocate memory for a copy of **v** (call by value)

Deallocate the space for all variables of the program (**v**)

Allocation/deallocation of memory occurs automatically



Automatic memory allocation/deallocation

When the function is called:

`factorial(5);`

```
int factorial(int n)
{
  int prod = 1;
  if (!n) return 1;
  for(int k = 2; k < n; k++)
      prod *= k;
  return prod;
}
```

Allocate memory for **`prod`**

Allocate memory for **`k`**

Loop ends: deallocate the memory of **`k`**

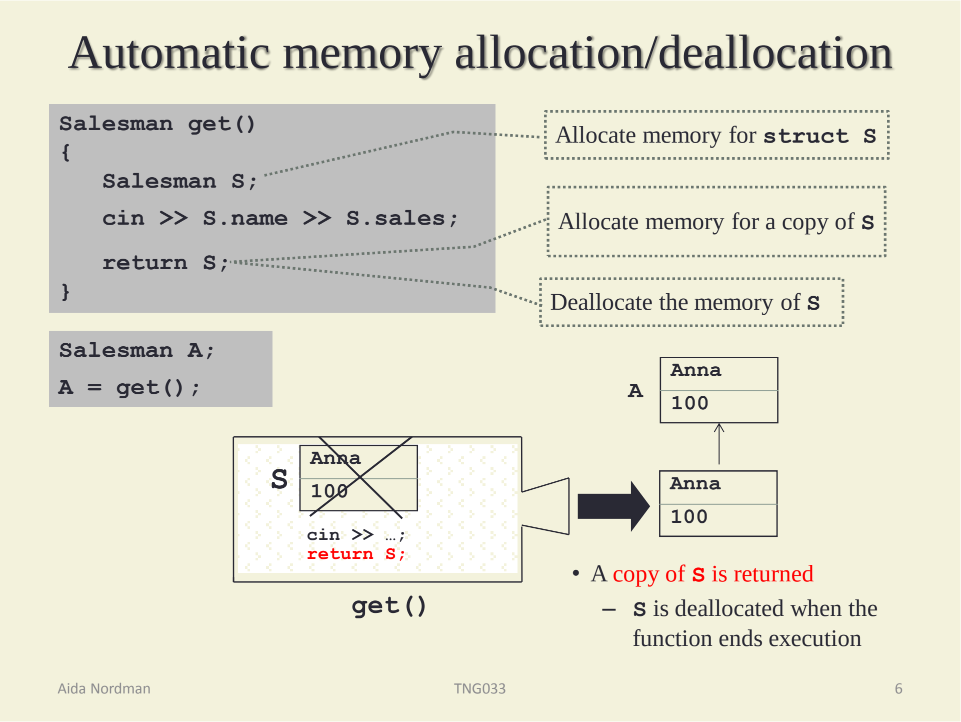
Allocate memory for a copy of **`prod`**

Deallocate the memory of **`prod`**

Deallocate the memory of **`n`**

Local variables are automatically

1. **Allocated** when a function (block) is **called** (entered)
2. **Deallocated** when a function (block) **ends**



Automatic memory allocation/deallocation

```
Salesman get()
{
   Salesman S;
   cin >> S.name >> S.sales;

   return S;
}
```

Allocate memory for **`struct S`**

Allocate memory for a copy of **`S`**

Deallocate the memory of **`S`**

```
Salesman A;
A = get();
```

- A copy of **`S`** is returned
  - **`S`** is deallocated when the function ends execution

Allocation/Deallocation of memory

- What if the programmer has no idea of how big an array should be?
  - Most of the times about 10 items, other times about 250.000
    - If `int`s are stored, it means **1MB** in the worst case
- Solution **A**
  1. Ask the user how many items to be stored
  2. Allocate dynamically the memory for an array
- Solution **B** -- Fö 5, lab 2
  - Use a dynamic data structure
    - No need to ask in advance to the user how many items
    - New items can be added and removed one at time

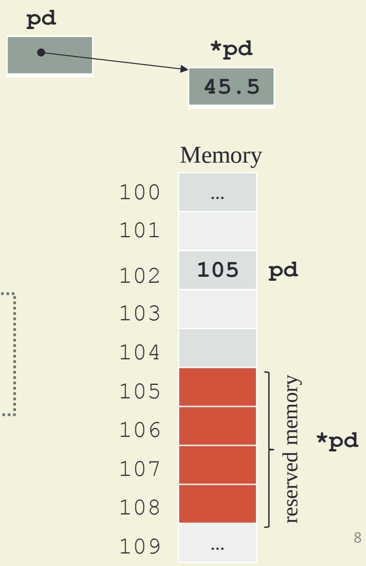
Memory Allocation: **new**

Declarations of functions used to manage dynamic storage in C++

```
#include <new>
double *pd = new double;
*pd = 45.5;
```

1. Allocate memory space for a double
2. Return the memory address of the first byte of the allocated memory, i.e. a pointer

Note: There's no way to access the allocated memory, but through the pointer

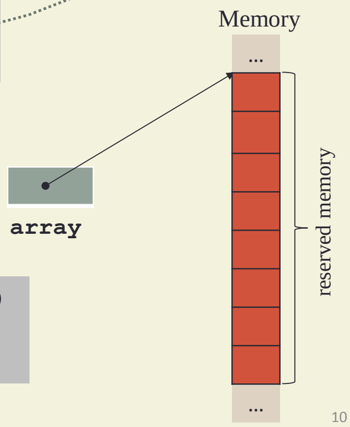


- Solution **A**
  1. Ask the user how many items (**`howMany`**) to be stored
  2. Allocate dynamically the memory for an array
     - Create an array that has **`howMany`** slots

Memory Allocation: **`new`**

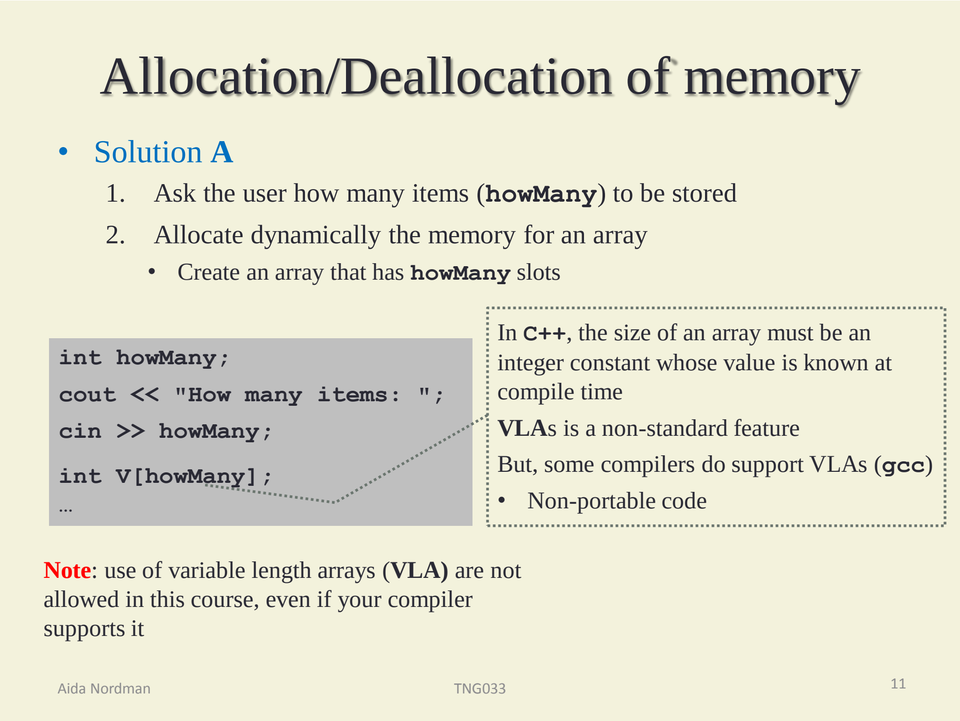
```
#include <new>
int howMany;
cout << "How many items: ";
cin >> howMany;
int *array = new int[howMany];
```

Allocate space for an array of **`int`**s



Read values and store them into the array

```
for(int i = 0; i < howMany; i++)
  cin >> array[i];
 //cin >> *(array+i)
```



Allocation/Deallocation of memory

- Solution **A**
  1. Ask the user how many items (**`howMany`**) to be stored
  2. Allocate dynamically the memory for an array
     - Create an array that has **`howMany`** slots

```
int howMany;
cout << "How many items: ";
cin >> howMany;

int V[howMany];
…
```

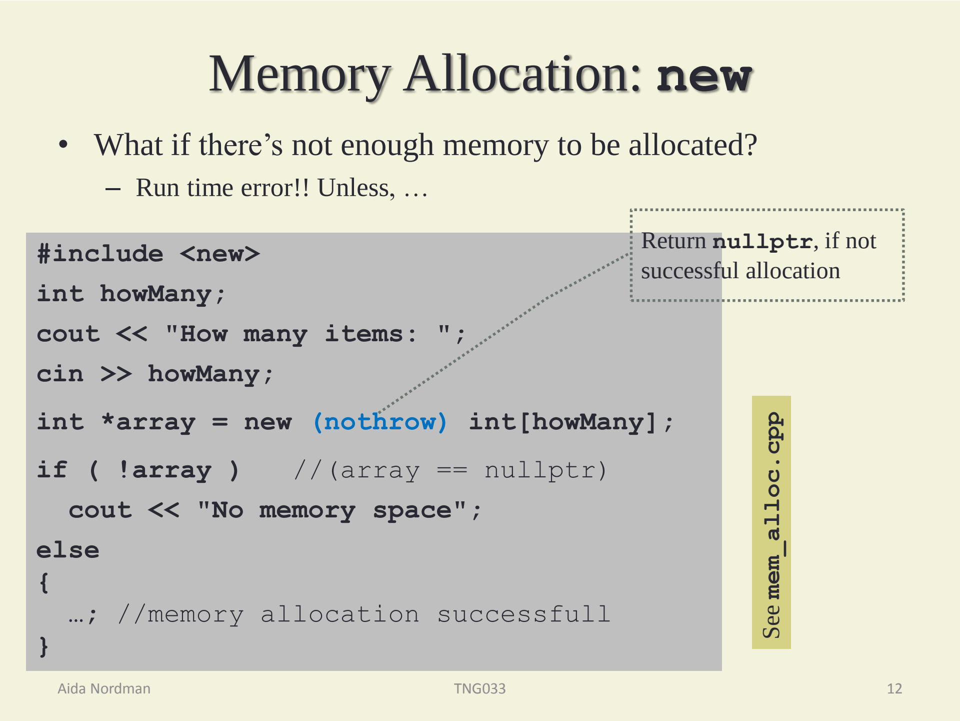
In **`C++`**, the size of an array must be an integer constant whose value is known at compile time

VLAs is a non-standard feature

But, some compilers do support VLAs (**`gcc`**)

- Non-portable code

Note: use of variable length arrays (**VLA)** are not allowed in this course, even if your compiler supports it



Memory Allocation: `new`

- What if there's not enough memory to be allocated?
  - Run time error!! Unless, …

```
#include <new>
int howMany;
cout << "How many items: ";
cin >> howMany;
int *array = new (nothrow) int[howMany];
if ( !array )    //(array == nullptr)
  cout << "No memory space";
else
{
  …; //memory allocation successfull
}
```

Return **`nullptr`**, if not successful allocation

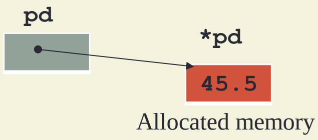
See **`mem_alloc.cpp`**

Memory Deallocation

- Memory allocated with **`new`** is reserved until the program ends
  - Unless, the programmer deallocates it explicitly
- Programs may need to deallocate memory of ”*old*” variables before are able to allocate new memory

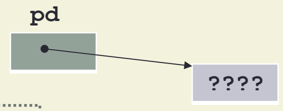
Memory Deallocation: **`delete`**

```
#include <new>
double *pd = new double(45.5);
…;
```



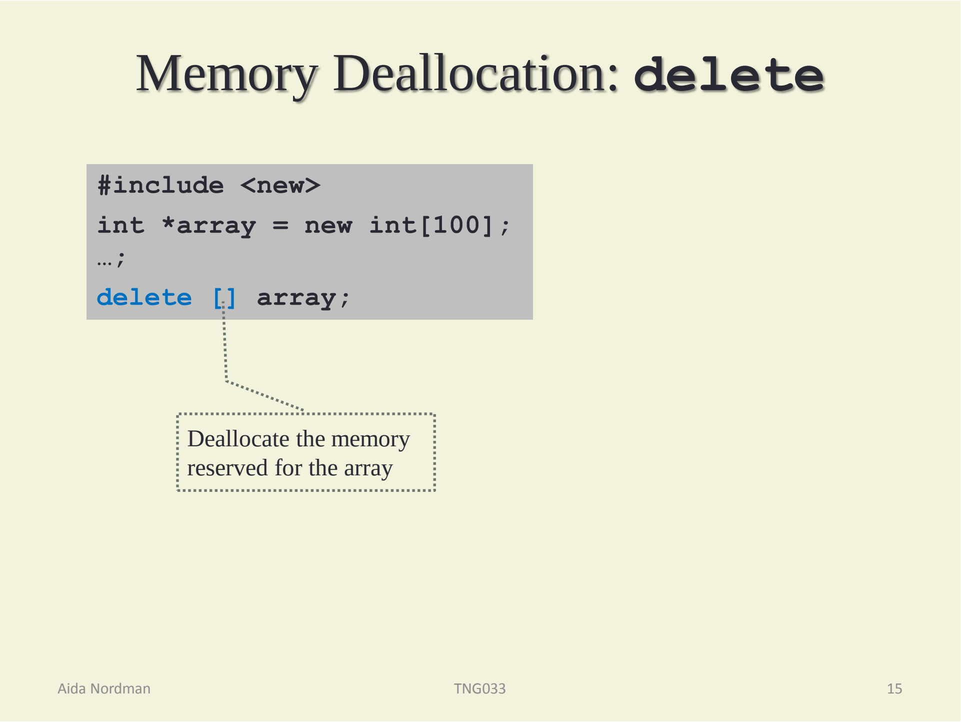
```
//*pd is not needed
delete pd;
```

Deallocate the memory pointed by pointer **`pd`**



```
cout << *pd;
```

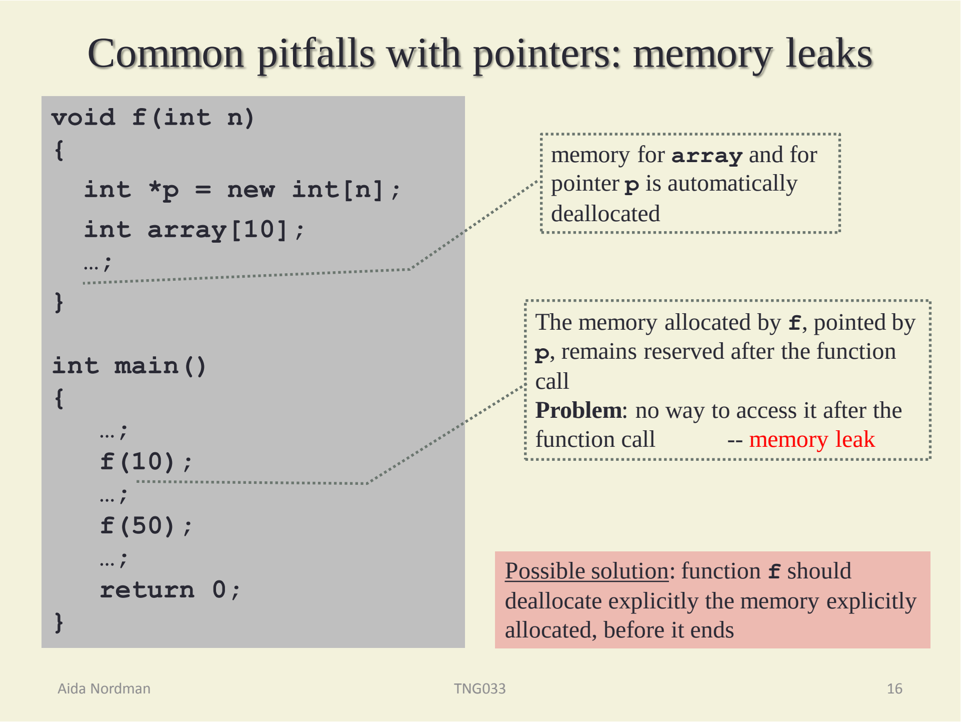
Do not dereference a pointer pointing to deallocated memory Program *may* crash!!



Memory Deallocation: **delete**

```
#include <new>
int *array = new int[100];
…;
delete [] array;
```

Deallocate the memory reserved for the array



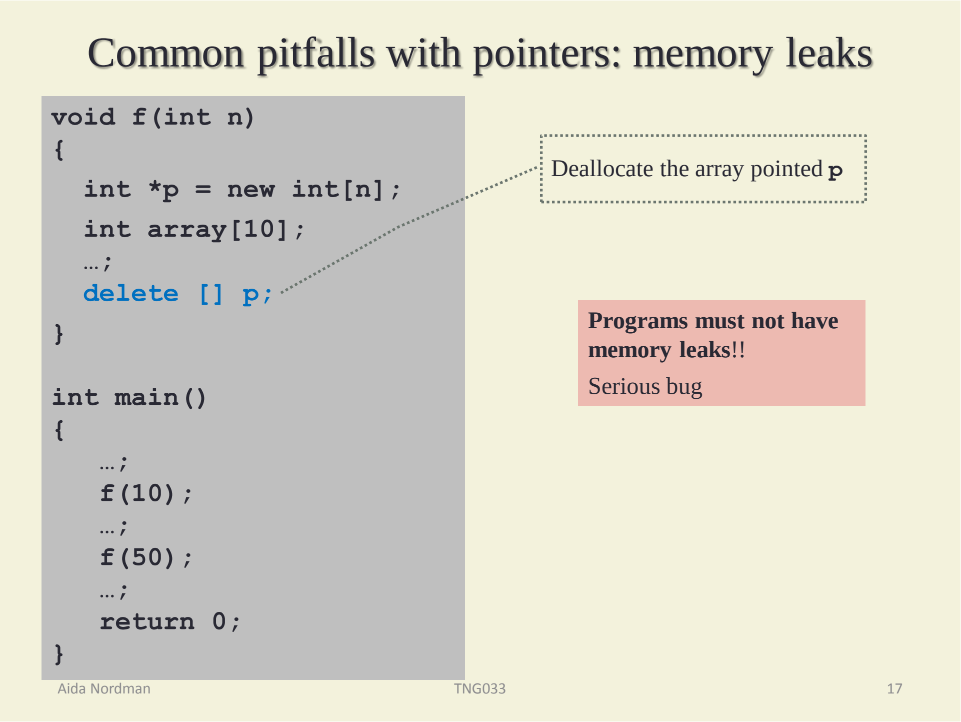
Common pitfalls with pointers: memory leaks

```
void f(int n)
{
  int *p = new int[n];
  int array[10];
  …;
}

int main()
{
   …;
   f(10);
   …;
   f(50);
   …;
   return 0;
}
```

memory for **array** and for pointer **p** is automatically deallocated

The memory allocated by **f**, pointed by **p**, remains reserved after the function call
Problem: no way to access it after the function call -- memory leak

Possible solution: function **f** should deallocate explicitly the memory explicitly allocated, before it ends



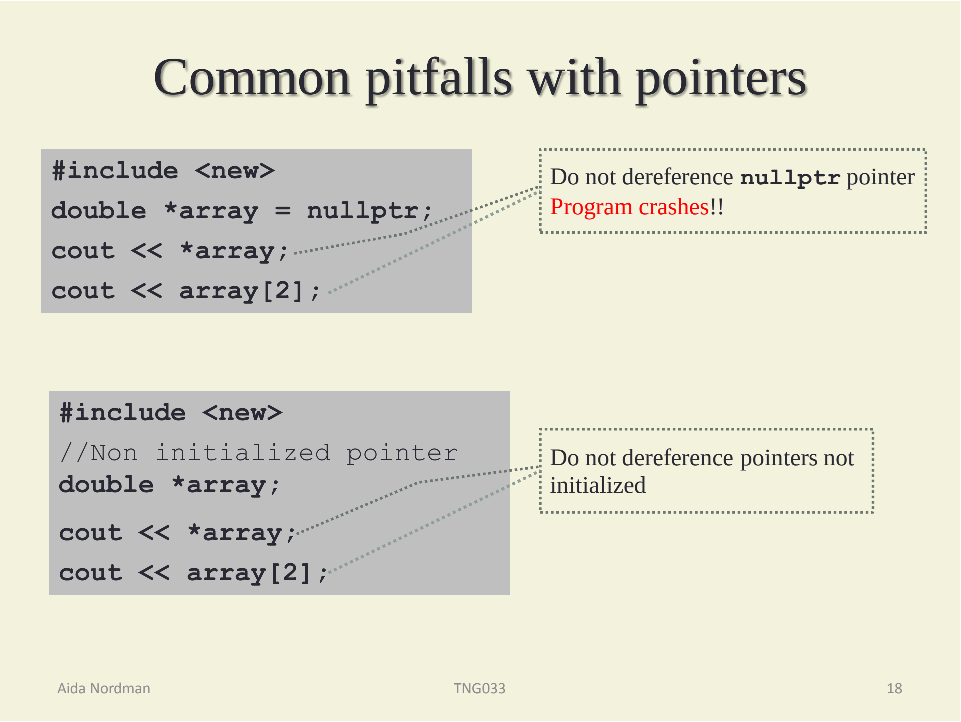
Common pitfalls with pointers: memory leaks

```
void f(int n)
{
  int *p = new int[n];
  int array[10];
  …;
  delete [] p;
}

int main()
{
   …;
   f(10);
   …;
   f(50);
   …;
   return 0;
}
```

Deallocate the array pointed **p**

Programs must not have memory leaks!!

Serious bug



Common pitfalls with pointers

```
#include <new>
double *array = nullptr;
cout << *array;
cout << array[2];
```

Do not dereference **nullptr** pointer
Program crashes!!

```
#include <new>
//Non initialized pointer
double *array;
cout << *array;
cout << array[2];
```

Do not dereference pointers not initialized

Common pitfalls with pointers

- Functions must not return pointers to (memory) local variables that have been allocated automatically

See `DonDo.cpp`

```
int* read_seq()
{
   int seq[10];
   for(int i = 0; i < 10; i++)
        cin >> seq[i];
   return seq;
}
```

name of array (**seq**) is converted to a pointer **int***

```
int main()
{
  int *ptr = read_seq();
  …;
  return 0;
}
```

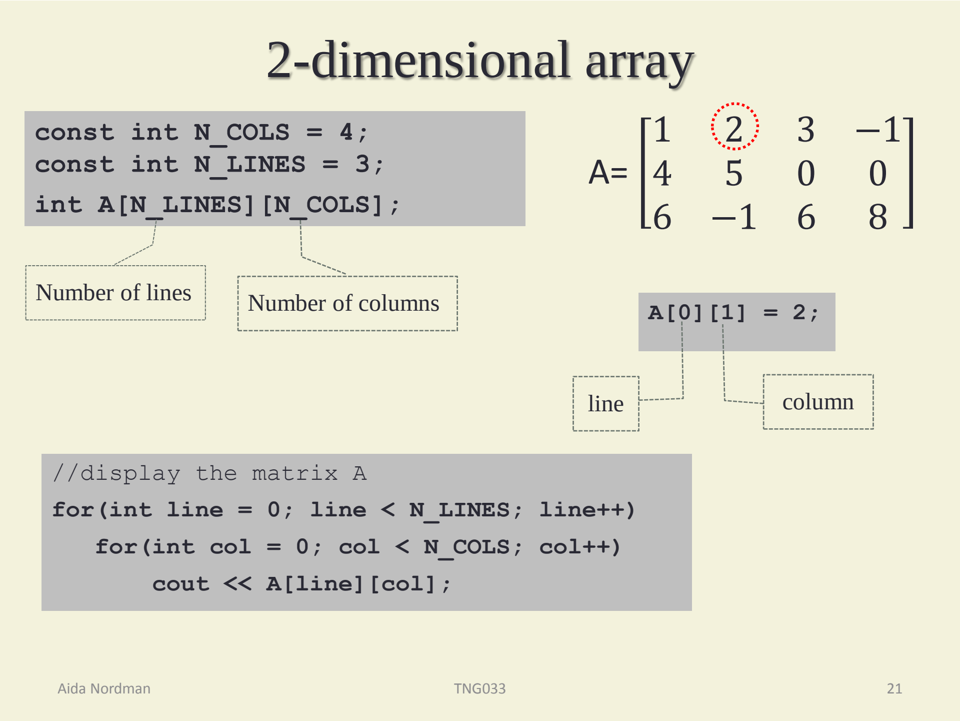
ptr points to memory that has already been (automatically) deallocated!!

Read sec. 5.4.6
Example on pag. 171

Example: matrices

- How can a matrix A be represented in C++?

$$A = \begin{array}{cc} \xrightarrow{\text{4 columns}} & \\ \begin{bmatrix} 1 & 2 & 3 & -1 \\ 4 & 5 & 0 & 0 \\ 6 & -1 & 6 & 8 \end{bmatrix} & \Big\downarrow \text{3 lines} \end{array}$$



2-dimensional array

```
const int N_COLS = 4;
const int N_LINES = 3;
int A[N_LINES][N_COLS];
```

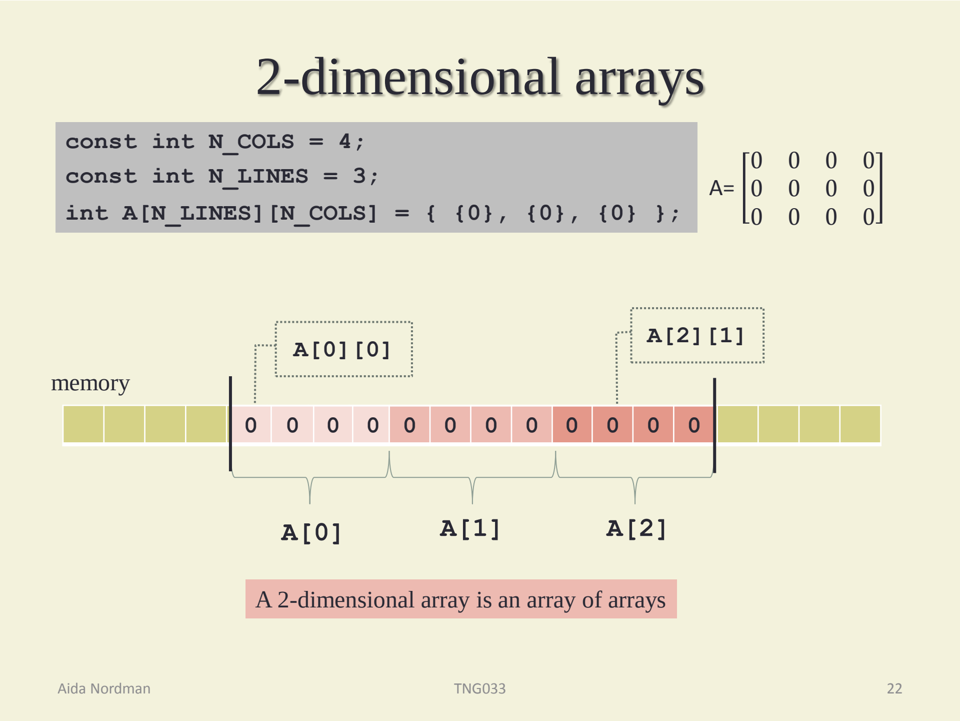
Number of lines
Number of columns

$$A=\begin{bmatrix}1 & 2 & 3 & -1\\ 4 & 5 & 0 & 0\\ 6 & -1 & 6 & 8\end{bmatrix}$$

```
A[0][1] = 2;
```

line
column

```
//display the matrix A
for(int line = 0; line < N_LINES; line++)
   for(int col = 0; col < N_COLS; col++)
       cout << A[line][col];
```

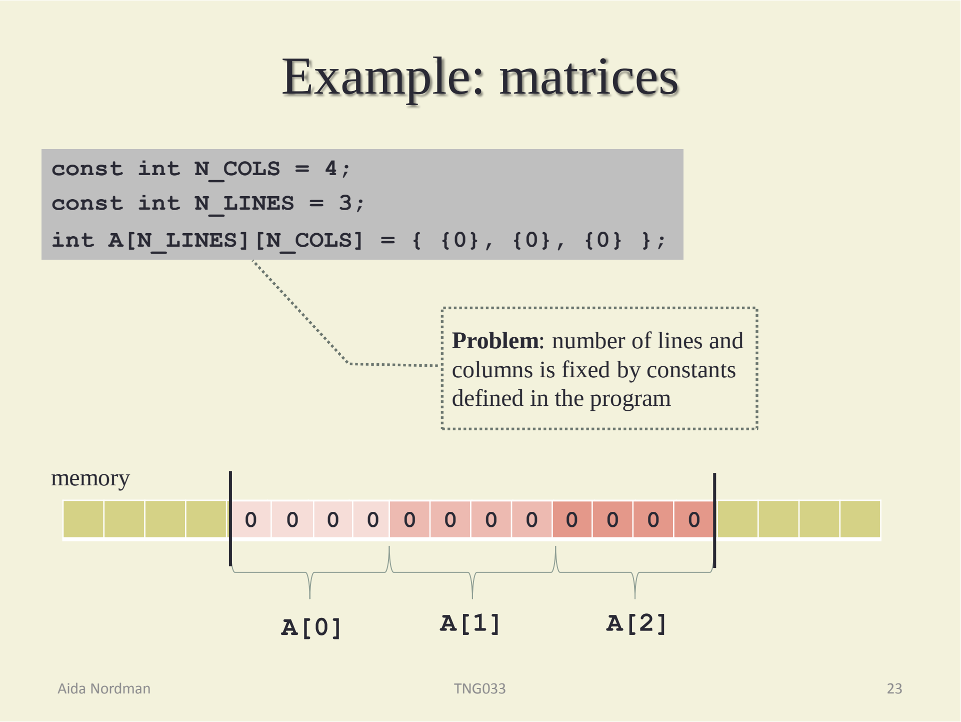


2-dimensional arrays

```
const int N_COLS = 4;
const int N_LINES = 3;
int A[N_LINES][N_COLS] = { {0}, {0}, {0} };
```

$$A=\begin{bmatrix}0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\end{bmatrix}$$

A 2-dimensional array is an array of arrays



Example: matrices

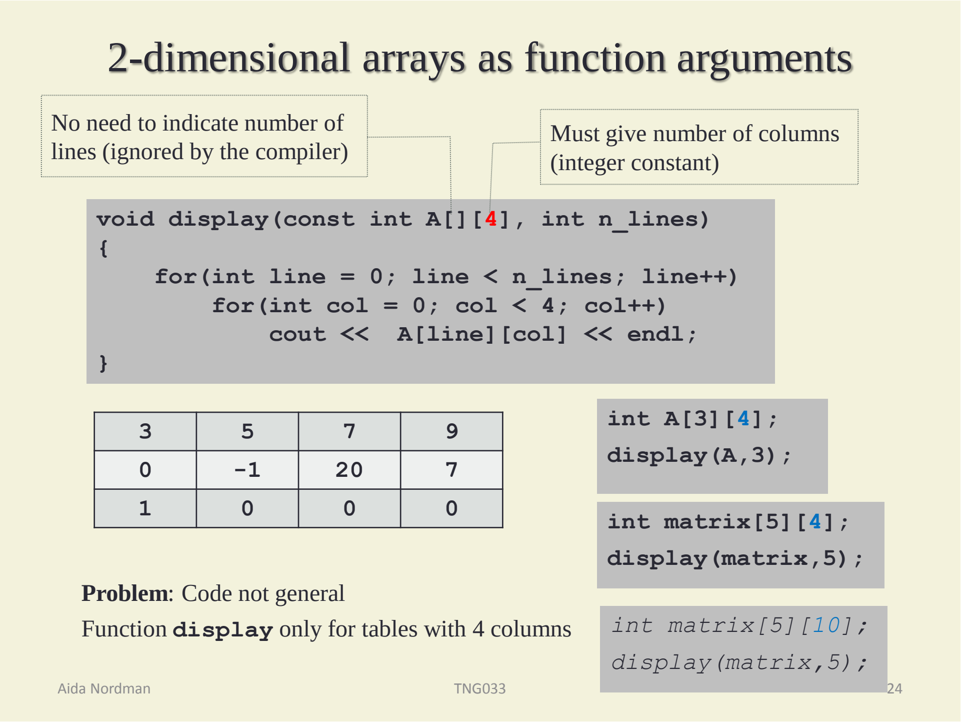
```
const int N_COLS = 4;
const int N_LINES = 3;
int A[N_LINES][N_COLS] = { {0}, {0}, {0} };
```

Problem: number of lines and columns is fixed by constants defined in the program

memory

				0	0	0	0	0	0	0	0	0	0	0	0				

Aida Nordman TNG033 23



2-dimensional arrays as function arguments

No need to indicate number of lines (ignored by the compiler)

Must give number of columns (integer constant)

```
void display(const int A[][4], int n_lines)
{
    for(int line = 0; line < n_lines; line++)
        for(int col = 0; col < 4; col++)
            cout <<  A[line][col] << endl;
}
```

3	5	7	9
0	-1	20	7
1	0	0	0

```
int A[3][4];
display(A,3);
```

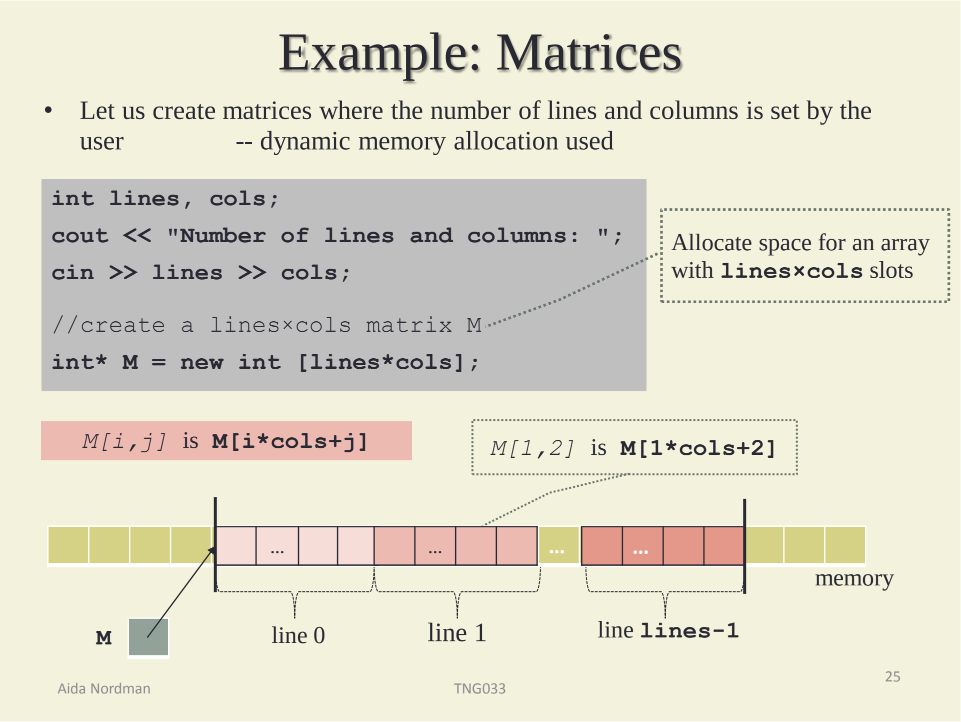
```
int matrix[5][4];
display(matrix,5);
```

```
int matrix[5][10];
display(matrix,5);
```

Problem: Code not general

Function **display** only for tables with 4 columns

Aida Nordman TNG033 24



Example: Matrices

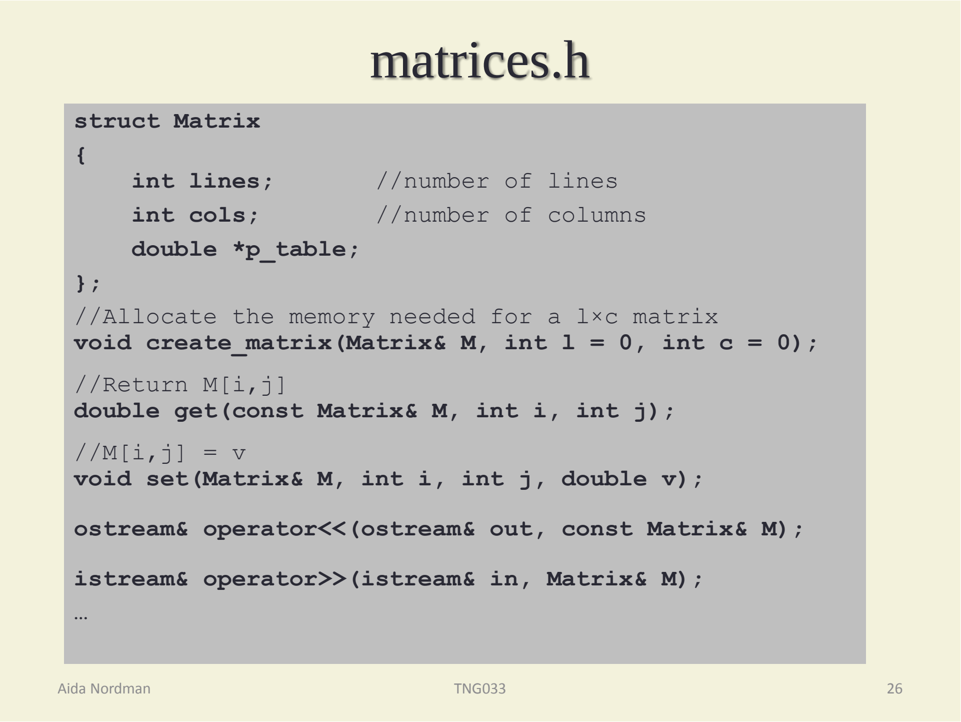
- Let us create matrices where the number of lines and columns is set by the user -- dynamic memory allocation used

```
int lines, cols;
cout << "Number of lines and columns: ";
cin >> lines >> cols;

//create a lines×cols matrix M
int* M = new int [lines*cols];
```

Allocate space for an array with **lines×cols** slots

M[i,j] is **M[i*cols+j]**

M[1,2] is **M[1*cols+2]**

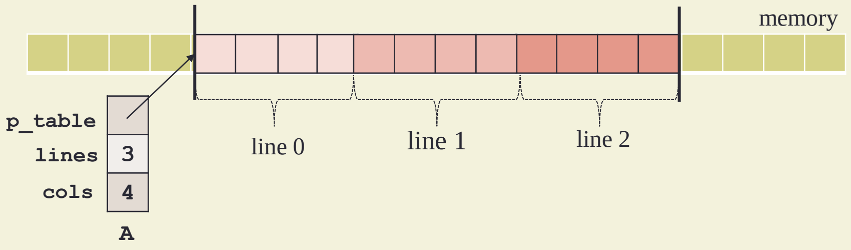


matrices.h

```
struct Matrix
{
    int lines;        //number of lines
    int cols;         //number of columns
    double *p_table;
};
//Allocate the memory needed for a l×c matrix
void create_matrix(Matrix& M, int l = 0, int c = 0);

//Return M[i,j]
double get(const Matrix& M, int i, int j);

//M[i,j] = v
void set(Matrix& M, int i, int j, double v);

ostream& operator<<(ostream& out, const Matrix& M);

istream& operator>>(istream& in, Matrix& M);

…
```

```
void create_matrix(Matrix& M, int l, int c)
{
    if (!(l && c))   //create an empty matrix
    {
        M.lines = M.cols = 0;
        M.p_table = nullptr;
        return;
    }
    M.lines = l; //set number of lines
    M.cols = c;  //set number of columns
    M.p_table = new double [l*c]; //allocate memory
}
```

```
Matrix A;
int lines, cols;
cout << "Line and columns? ";
cin >> lines >> cols;
create_matrix(A, lines, cols);
```



Example: matrices

Download the files **`matrices.h`**, **`matrices.cpp`**, **`matrices_test.cpp`** and create a project

Study the example

Next …

- Fö 5
  - Dynamic data structures: singly-linked lists [13.1.1]
  - Very important for Lab 2
- Lesson 1 -- exercises
  - Do exercise 1 and 2
  - Do exercise 3, after Fö 5