Lecture 4: Pointers and functions

Jonas Kusch and Martina Prugger
University of Innsbruck

March 17, 2023

Last goals: You are able to

- ☑ understand memory management
- ☑ start to use pointers
- ☑ generate dynamic and static arrays

Last goals: You are able to

- ☑ understand memory management
- ☑ start to use pointers
- ☑ generate dynamic and static arrays

Today's learning goals: You will be able to

- ☐ understand pointer arithmetics
- ☐ use functions

Last goals: You are able to

- ☑ understand memory management
- ☑ start to use pointers
- ☑ generate dynamic and static arrays

Today's learning goals: You will be able to

- ☐ understand pointer arithmetics
- ☐ use functions

Ask questions any time!

Why?

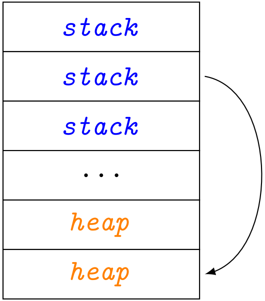
- Pointers give control over memory.
- Pass data to a different program part without copying it (`functions`,`classes`,...).
- Control when to delete data (dynamic vs. static memory).

stack
stack
stack
...
heap
heap

- **static** memory managed by compiler (*stack*)
- **dynamic** memory managed by user (*heap*)
- dynamic memory can be accessed with pointers (stored in stack)
- address space in heap is accessed with `new`

Why?

- Pointers give control over memory.
- Pass data to a different program part without copying it (`functions`,`classes`,...).
- Control when to delete data (dynamic vs. static memory).



- **static** memory managed by compiler (*stack*)
- **dynamic** memory managed by user (*heap*)
- dynamic memory can be accessed with pointers (stored in stack)
- address space in heap is accessed with `new`

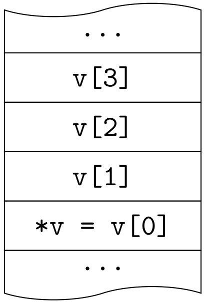
Code example

```
#include <iostream>

int main(){
    double dStatic = 0.1;
    double *dDynamic = new double; // allocate memory in heap
    *dDynamic = 0.1;

    std::cout<<dStatic<<" "<<*dDynamic<<std::endl;

    delete dDynamic; // free memory

    std::cout<<dStatic<<" "<<*dDynamic<<std::endl;

    return 0;
}
```

Arrays in heap

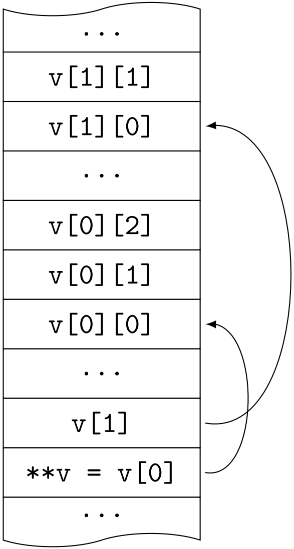
```
#include <iostream>

int main(){
    double *v = new double [2]; // allocate array of size 2 in heap
    v[0] = 0.1;
    v[1] = 0.12;

    delete [] v; // free memory of entire array

    return 0;
}
```

Arrays in heap



- We store address of `v[0]` on `v`.
- Since `v[1]`,... neighbor `v[0]` we also know their addresses.
- More about this when talking about pointer arithmetics.

Multi-dimensional arrays in heap



Multi-dimensional arrays in heap

Multi-dimensional arrays in heap

```cpp
#include <iostream>

int main(){
    int n = 3, m = 4;
    double** v = new double* [n]; // allocate array of pointers

    for( long i = 0; i < n; ++i)
        v[i] = new double [m]; // allocate double array for every v[i]

    v[0][1] = 0.1;

    for( long i = 0; i < n; ++i)
        delete [] v[i]; // delete array of doubles for every v[i]

    delete [] v; // delete array of pointers

    return 0;
}
```

Multi-dimensional arrays in heap

Task

Implement a 3-dimensional array a with dimension $n_1 = 2, n_2 = 3, n_3 = 4$. Fill the array with numbers $a_{ijk} = i + j + k$. Do not forget to free your memory before the program terminates.

What does the code do? What happens in memory?

```cpp
#include <iostream>

int main(){
    double* d = new double;
    *d = 0.1;
    double* p = d;

    delete p;

    std::cout<<*d<<std::endl;

    return 0;
}
```

What does the code do? What happens in memory?

```cpp
#include <iostream>

int main(){
    bool condition = true;

    if( condition ){
        double* d = new double;
        *d = 0.1;
    }

    std::cout<<*d<<std::endl;

    return 0;
}
```

What does the code do? What happens in memory?

```cpp
#include <iostream>

int main(){
    bool condition = true;
    double* d;

    if( condition ){
        d = new double;
        *d = 0.1;
    }

    std::cout<<*d<<std::endl;

    return 0;
}
```

Last goals: You are able to

- ☑ understand memory management
- ☑ start to use pointers
- ☑ generate dynamic and static arrays

Today's learning goals: You will be able to

- ☐ understand pointer arithmetics
- ☐ use functions

Last goals: You are able to

- ☑ understand memory management
- ☑ start to use pointers
- ☑ generate dynamic and static arrays

Today's learning goals: You will be able to

- ☐ understand pointer arithmetics
- ☐ use functions

Pointer arithmetics

- Arithmetics on pointers allowed.
- `d[i]` equivalent to $= {}^{*}(d + i)$

```
#include <iostream>

int main(){
    double* d;
    d = new double [4];
    d[0] = 0.0; d[1] = 0.1; d[2] = 0.2;

    std::cout<< *d << " " << *(d + 1) <<std::endl;

    return 0;
}
```

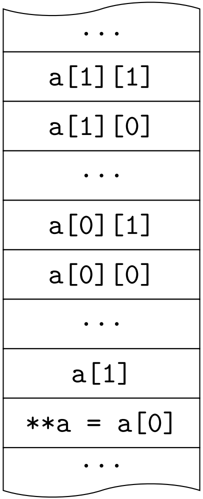
Pointer arithmetics

...	...	...
`v[3]`	`*(v+3)`	`(v+1)[2]`
`v[2]`	`*(v+2)`	`(v+1)[1]`
`v[1]`	`*(v+1)`	`(v+1)[0]`
`*v = v[0]`	`*v = v[0]`	`*v = v[0]`
...	...	...

What's the output?

```cpp
#include <iostream>

int main(){
    long** a = new long* [2];
    a[0] = new long [2];
    a[1] = new long [2];

    for( long i = 0; i < 2; ++i )
        for( long j = 0; j < 2; ++j )
            a[i][j] = i + j;

    std::cout<< *(a[1]+1) << " " << (*a - 1)[2] <<std::endl;
    std::cout<< *((a + 1)[0] + 1) <<std::endl;
    std::cout<< (a + 2)[0][2] << std::endl;

    return 0;
}
```

What's the output?



→ `*(a[1]+1)`

→ `(*a - 1)[2]`

→ `*((a + 1)[0] + 1)`

→ `(a + 2)[0][2]`

Last goals: You are able to

- ☑ understand memory management
- ☑ start to use pointers
- ☑ generate dynamic and static arrays

Today's learning goals: You will be able to

- ☑ understand pointer arithmetics
- ☐ use functions

Last goals: You are able to

- ☑ understand memory management
- ☑ start to use pointers
- ☑ generate dynamic and static arrays

Today's learning goals: You will be able to

- ☑ understand pointer arithmetics
- ☐ use functions

Functions

- Is everyone familiar with functions in programming languages?

```
<return_data_type> function_name( <input_1>, <input_2>,... ){
    \\ function body
    return <return_value>
}
```

Functions

- Is everyone familiar with functions in programming languages?

```
<return_data_type> function_name( <input_1>, <input_2>,... ){
    \\ function body
    return <return_value>
}
```

```cpp
#include <iostream>

double add(double a, double b){
    double c = a + b;
    return c;
}

int main(){
    std::cout << add(1,2) <<std::endl;
    return 0;
}
```

Your turn

Exercise

Rewrite your ODE solver as a function which takes start time and time grid as input and returns the solution at each time point as output. Use another function to define the right-hand-side of your ODE.

Function Overloading

Function Overloading

```cpp
#include <iostream>

double add(double a, double b){
    std::cout<<"double"<<std::endl;
    return a + b;
}

int add(int a, int b){
    std::cout<<"int"<<std::endl;
    return a + b;
}

int main(){
    std::cout<<add(1,2)<<std::endl;
    std::cout<<add(1.0,2.0)<<std::endl;
    return 0;
}
```

Function Overloading

Function Overloading

```cpp
#include <iostream>

double add(double a, double b){
    std::cout<<"double"<<std::endl;
    return a + b;
}

int main(){
    float a = 1.2, b = 2.2;
    std::cout<<add(a,b)<<std::endl;
    char c = 'c';
    long i = 1;
    std::cout<<add(c,i)<<std::endl;
    return 0;
}
```

Main

- `main` is also a function
- input to `main` are command line arguments

```cpp
#include <iostream>

int main(int argc, char** argv) {
    std::cout << "number inputs: " << argc << ", arguments are:" << std::endl;
    for (int i = 0; i < argc; ++i) {
        std::cout << argv[i] << std::endl;
    }
}
```

Main

- `main` is also a function
- input to `main` are command line arguments

```cpp
#include <iostream>

int main(int argc, char** argv) {
    std::cout << "number inputs: " << argc << ", arguments are:" << std::endl;
    for (int i = 0; i < argc; ++i) {
        std::cout << argv[i] << std::endl;
    }
}
```

Task

Rewrite your ODE solver to read in the initial condition.

Memory management

- What happens in memory? Let's check!

```
#include <iostream>

void print_address(double a){
    std::cout<<"Address in function "<<&a<<std::endl;
}

int main(){
    double a = 1.2;
    std::cout<<"Address in function "<<&a<<std::endl;
    print_address(a);
    return 0;
}
```

- Per default, the input is copied to a new location in memory.
- Advantage: Data is save from modification inside function.
- Disadvantage?

Memory management

- What happens in memory? Let's check!

```cpp
#include <iostream>

void print_address(double a){
    std::cout<<"Address in function "<<&a<<std::endl;
}

int main(){
    double a = 1.2;
    std::cout<<"Address in function "<<&a<<std::endl;
    print_address(a);
    return 0;
}
```

- Per default, the input is copied to a new location in memory.
- Advantage: Data is save from modification inside function.
- Disadvantage?

Memory management

- What happens in memory? Let's check!

```
#include <iostream>

void print_address(double a){
    std::cout<<"Address in function "<<&a<<std::endl;
}

int main(){
    double a = 1.2;
    std::cout<<"Address in function "<<&a<<std::endl;
    print_address(a);
    return 0;
}
```

- Per default, the input is copied to a new location in memory.
- Advantage: Data is save from modification inside function.
- Disadvantage?

Call by reference

```cpp
#include <iostream>

void print_address(double& a){
    std::cout<<"Address in function "<<&a<<std::endl;
}

int main(){
    double a = 1.2;
    std::cout<<"Address in function "<<&a<<std::endl;
    print_address(a);
    return 0;
}
```

- & operator ensures data is not copied (pointers!)
- Disadvantage: Data is not save from modification inside function.
- Advantage: Data can be modified from within function, performance

Call by reference

```
#include <iostream>

void print_address(double& a){
    std::cout<<"Address in function "<<&a<<std::endl;
}

int main(){
    double a = 1.2;
    std::cout<<"Address in function "<<&a<<std::endl;
    print_address(a);
    return 0;
}
```

- & operator ensures data is not copied (pointers!)
- Disadvantage: Data is not save from modification inside function.
- Advantage: Data can be modified from within function, performance

Call by reference

```cpp
#include <iostream>

void print_address(const double& a){
    std::cout<<"Address in function "<<&a<<std::endl;
}

int main(){
    double a = 1.2;
    std::cout<<"Address in function "<<&a<<std::endl;
    print_address(a);
    return 0;
}
```

- & operator ensures data is not copied (pointers!)
- Disadvantage: Data is not save from modification inside function.
- Advantage: Data can be modified from within function, performance

What is the output?

```
#include <iostream>

void foo(double a){
    a = 0.123;
}

int main(){
    double a = 1.2;
    foo(a);
    std::cout<<a<<std::endl;
    return 0;
}
```

What is the output?

```
#include <iostream>

void foo(double& a){
    a = 0.123;
}

int main(){
    double a = 1.2;
    foo(a);
    std::cout<<a<<std::endl;
    return 0;
}
```

What is the output?

```cpp
#include <iostream>

void foo(double& a){
    a = 0.123;
}

int main(){
    double* p = new double;
    *p = 1.0;
    foo(*p);
    std::cout<<*p<<std::endl;
    return 0;
}
```

What is the output?

```
#include <iostream>

void foo(double& a){
    a = 0.123;
}

int main(){
    double* p = new double;
    *p = 1.0;
    foo(*p);
    std::cout<<*p<<std::endl;
    return 0;
}
```

What is missing?

What is the output?

```
#include <iostream>

void foo(double* a){
    a[0] = 1.234;
}

int main(){
    double* p = new double [3];
    p[0] = 0; p[1] = 1; p[2] = 2;
    foo(p);
    std::cout<<*p<<std::endl;
    return 0;
}
```

What is the output?

```
#include <iostream>

void foo(double* a){
    a = a + 1;
}

int main(){
    double* p = new double [3];
    p[0] = 0; p[1] = 1; p[2] = 2;
    foo(p);
    std::cout<<*p<<std::endl;
    return 0;
}
```

Your turn

Exercise

Write a function which takes a dynamic array of type `double` called `x` as input and as well as an output array `y`. The function then stores `sin(x[i])` on the output array. Make sure that `x` is copied efficiently and cannot be modified inside the function. The output `y` is available outside the function after it has been called.

Now it's up to you...

Current learning goals: After homework and self-study

- ☑ understand memory management
- ☑ start to use pointers
- ☑ generate dynamic and static arrays
- ☑ understand pointer arithmetics
- ☑ use functions

Any questions / remarks ? :) {*jonas.kusch, martina.prugger*}*@uibk.ac.at*

Now it's up to you...

Current learning goals: After homework and self-study

- ☑ understand memory management
- ☑ start to use pointers
- ☑ generate dynamic and static arrays
- ☑ understand pointer arithmetics
- ☑ use functions

Any questions / remarks ? :) {*jonas.kusch, martina.prugger*}*@uibk.ac.at*

Next learning goals:

- ☐ continue using functions
- ☐ start working with classes