Lecture 5: Functions and Classes

Jonas Kusch and Martina Prugger

University of Innsbruck

March 21, 2023

Last goals: You are able to

- ☑ generate dynamic and static arrays
- ☑ understand pointer arithmetics

Last goals: You are able to

- ☑ generate dynamic and static arrays
- ☑ understand pointer arithmetics

Today's learning goals: You will be able to

- □ use functions
- □ start using classes

Last goals: You are able to

- ☑ generate dynamic and static arrays
- ☑ understand pointer arithmetics

Today's learning goals: You will be able to

- □ use functions
- □ start using classes

Ask questions any time!

Pointer arithmetics

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

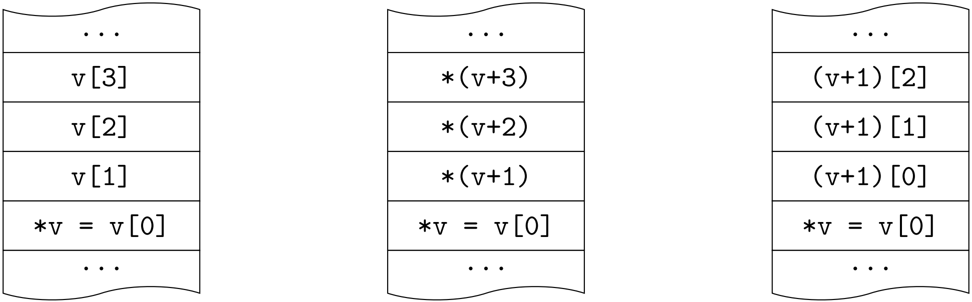
```cpp
#include <iostream>

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[2]
v[1]
*v = v[0]
...
```

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

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

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

```
#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!

```
#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?

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?

```cpp
#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 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?

```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 missing?

What is the output?

```cpp
#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?

```cpp
#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.

Static variables

- `static` variables remain in memory when function is returned

```cpp
#include <iostream>

void f(){
    static long counter = 0;
    counter++;
    std::cout<<"Function called "<<counter<<" times."<<std::endl;
}

int main(){
    f();
    f();
    return 0;
}
```

Last goals: You are able to

- ☑ generate dynamic and static arrays
- ☑ understand pointer arithmetics

Today's learning goals: You will be able to

- ☑ use functions
- □ start using classes

Last goals: You are able to

- ☑ generate dynamic and static arrays
- ☑ understand pointer arithmetics

Today's learning goals: You will be able to

- ☑ use functions
- □ start using classes

Structs

- Sometimes you want to generate your own data types.
- Example: Pair of two doubles.

```
#include <iostream>

struct Pair{
    double first;
    double second;
};

int main(){
    Pair p;
    p.first = 1.0;
    p.second = 2.0;
    return 0;
}
```

What does this code do?

```cpp
#include <iostream>
struct entry{
    long data;
    entry* next;
    entry* previous;
};
int main(){
    entry* previous= NULL;
    for( long i = 0; i < 10; ++i ){
        entry* current = new entry;
        current->data = i;
        current->previous = previous;
        if(previous) previous->next = current;
        previous = current;
    }
    previous->next = NULL;
    return 0;
}
```

Your turn

Exercise

Print all entries in reverse and forward order by running through the created objects of type `entry`.

Exercise

Delete all created objects of type `entry`.

Classes

Classes can be seen as fancy structs which are equipped with

- constructors that take care of initialization
- destructors that take care of deletion
- (copy) operations
- functions
- hierarchies
- protection of variables and functions
- . . .

Classes - Syntax

```
class class_name{
    private:
        \\ private variables and functions
    public:
        \\ public variables and functions
};
```

Example:

```
class Entry{
    private:
        long _data;
        Entry* _next;
        Entry* _previous;
    public:
        Entry(long data): _data(data) {}
};
```

Classes - Functions

```
class Entry{
    private:
        long _data;
        Entry* _next;
        Entry* _previous;
    public:
        Entry(long data): _data(data) {}
        void Print(){std::cout<<_data<<std::endl;}
};

Entry first(2);
first.print();
```

Classes - Functions

```
class Entry{
    private:
        long _data;
        Entry* _next;
        Entry* _previous;
    public:
        Entry(long data): _data(data) {}
        void Print();
};

void Entry::Print(){
    std::cout<<_data<<std::endl;
}
```

Classes - private and public

```
class Entry{
    private:
        long _data;
        Entry* _next;
        Entry* _previous;
    public:
        long _publicData;
        Entry(long data): _data(data), _publicData(data) {}
};

Entry first(2);
std::cout<<first._data;
std::cout<<first._publicData;
```

- `private` data/functions are protected from modification
- `public` data/functions are accessible
- note that the `Print` function *can* access `private` data

Your turn

Exercise

Write a class `ODESolver` which stores all needed variables. The class 1) provides a void function `Solve(endTime)` which stores the solution inside the class and 2) provides a void function `Write(fileName)` which writes an outputfile with the solution at every time point.