Lecture 6: Classes

Jonas Kusch and Martina Prugger

University of Innsbruck

March 24, 2023

Last goals: You are able to

- ☑ use functions
- ☑ use structs

Last goals: You are able to

- ☑ use functions
- ☑ use structs

Today's learning goals: You will be able to

- ☐ start using classes

Last goals: You are able to

- ☑ use functions
- ☑ use structs

Today's learning goals: You will be able to

- ☐ start using classes

Ask questions any time!

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`.

Exercise

Close the loop: Make sure that the last entry will point to the first entry and vice-versa.

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 (data encapsulation)
- ...

Aim: Readability and extendability

- structure your code as objects that can interact (like object list which has objects entries; object solver which has object time integrator, object mesh, ...)
- data encapsulation

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

```cpp
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.

The constructor

- Every class has a constructor, i.e., a function which is called when an object is created.

```
class Entry{
    ...
public:
    Entry(long data);
};

Entry::Entry(long data): _data(data), _publicData(data), _next(0), _previous(0) {
    std::cout<<"Constructor called." << std::endl;
}
```

What happens if we create an object `Entry tmp`?

The constructor

```
class Entry{
    ...
public:
    Entry(double data);
private:
    Entry(){}
};
```

- Making the default constructor private will remove the option to call it.

Data management

```
class Entry{
    double* _data;
public:
    Entry(double data): _data(new double){ *_data = data;}

};

int main(){
    if(true) Entry tmp(1);
    return 0;
}
```

- What behaviour do you expect?

Data management

```
class Entry{
    double* _data;
public:
    Entry(double data): _data(new double){ *_data = data;}

};

int main(){
    if(true) Entry tmp(1);
    return 0;
}
```

- What behaviour do you expect?
- Dynamic memory will not be deleted by default. This has to be done via the destructor.

Data management

```
class Entry{
    double* _data;
public:
    Entry(double data): _data(new double){ *_data = data;}
    ~Entry(){delete _data;}
};

int main(){
    if(true) Entry tmp(1);
    return 0;
}
```

- What behaviour do you expect?
- Dynamic memory will not be deleted by default. This has to be done via the destructor.
- Commonly an object which allocates dynamic memory has to deallocate it.

Do I need a destructor?

```
class Entry{
    double* _data;
public:
    Entry(double data): _data(&data){}
};

int main(){
    double data = 2;
    Entry tmp(data);
    return 0;
}
```

Do I need a destructor?

```
class Entry{
    double* _data;
public:
    Entry(double data): _data(&data){}
};

int main(){
    double* data = new double;
    Entry tmp(data);
    return 0;
}
```

Do I need a destructor?

```
class Entry{
    double* _data;
public:
    Entry(double data): _data(NULL){
        _data = new double [10];
        _data[0] = data;
    }
};

int main(){
    double* data = new double;
    Entry tmp(data);
    delete data;
    return 0;
}
```

What does this code do?

```
class Entry{
    double* _data;
public:
    Entry(double data): _data(&data){}
    ~Entry(double data){delete data;}
};

int main(){
    double* data = new double;
    Entry tmp(data);
    delete data;
    return 0;
}
```

What does this code do?

```
class Entry{
    double* _data;
public:
    Entry(double data): _data(new double){ *_data = data;}
    double* GetData(){return _data;}
};

int main(){
    double* d;
    if(true){
        Entry* tmp = new Entry(1.0);
        d = tmp->GetData();
    }
    std::cout<< *d <<std::endl;
    return 0;
}
```

What does this code do?

```
class Entry{
    double* _data;
public:
    Entry(double data): _data(new double){ *_data = data;}
    double* GetData(){return _data;}
};

int main(){
    double* d;
    if(true){
        Entry tmp(1.0);
        d = tmp.GetData();
    }
    std::cout<< *d <<std::endl;
    return 0;
}
```

What does this code do?

```
class Entry{
    double* _data;
public:
    Entry(double data): _data(new double){ *_data = data;}
    double* GetData(){return _data;}
    ~Entry(){std::cout<<"Removing data..."<<std::endl;}
};

int main(){
    double* d;
    if(true){
        Entry tmp(1.0);
        d = tmp.GetData();
    }
    std::cout<< *d <<std::endl;
    return 0;
}
```

What does this code do?

```
class Entry{
    double* _data;
public:
    Entry(double data): _data(new double){ *_data = data;}
    double* GetData(){return _data;}
    ~Entry(){delete _data;}
};

int main(){
    double* d;
    if(true){
        Entry tmp(1.0);
        d = tmp.GetData();
    }
    std::cout<< *d <<std::endl;
    return 0;
}
```

Last one...

```
double* GetData(Entry& e){
    return e._data;
}

class Entry{
    double* _data;
public:
    Entry(double data): _data(new double){ *_data = data;}
    double* GetData(){return _data;}
    ~Entry(){delete _data;}
};

int main(){
    Entry tmp(1.0);
    std::cout<< *GetData(tmp) <<std::endl;
    return 0;
}
```

...works like this

```
class Entry{
    double* _data;
public:
    Entry(double data): _data(new double){ *_data = data;}
    double* GetData(){return _data;}
    ~Entry(){delete _data;}
    friend double* GetData(Entry& e);
};

double* GetData(Entry& e){return e._data;}

int main(){
    Entry tmp(1.0);
    std::cout<< *GetData(tmp) <<std::endl;
    return 0;
}
```

What was the problem?

- Class undefined when used in function (forward declaration can help)
- Function accessed private data (declare as friend)

Data protection:

- Ensure that data is not manipulated.
- Think of all the errors we generated by giving a pointer to the main!

What was the problem?

- Class undefined when used in function (forward declaration can help)
- Function accessed private data (declare as friend)

Data protection:

- Ensure that data is not manipulated.
- Think of all the errors we generated by giving a pointer to the main!

Exercise

Write a class `List` which has a pointer to the first and last `Entry` in the list. Moreover, it incorporates a function `Add` which creates a new entry (in dynamic memory) and adds it to the back of the list. Provide a function `Print` which prints out all values in the list. Do not forget the destructor!

Entry

```
class Entry{
    double _data;
    Entry* _next;
    Entry* _previous;
    Entry(double d): _data(data), _previous(NULL), _next(NULL) {}
    Entry(double d, Entry* prev);
    double GetData(){return _data;}
    Entry* GetNext(){return _next;}
    Entry* GetPrevious(){return _previous;}
    friend List;
};

Entry::Entry(double d, Entry* prev): _data(d), _previous(prev), _next(0) {
    previous->_next = this;
}
```

- this points to the object itself.

Copy constructor

```
class A{
public:
    double* _d;
    A(double d): _d(new double) {*_d = d;}
    ~A() {delete _d;}
};

void foo(A aFoo){}

int main(){
    A a(1.234);
    foo(a);
    std::cout<< *a._d <<std::endl;
}
```

- What is the output?

Copy constructor

- Function `foo` creates a new object `aFoo` and copies all values from `a`.
- We did not specify how this new class `A` is copied!
- By default all variables are simply copied. I.e., `aFoo._d = a._d`.
- `aFoo` is deallocated when leaving function (static memory!)
- Destructor of `aFoo` deallocates `aFoo._d` and thereby `a._d`.

→ Good thing we understand pointers ;)

Copy constructor

- Function `foo` creates a new object `aFoo` and copies all values from `a`.
- We did not specify how this new class `A` is copied!
- By default all variables are simply copied. I.e., `aFoo._d = a._d`.
- `aFoo` is deallocated when leaving function (static memory!)
- Destructor of `aFoo` deallocates `aFoo._d` and thereby `a._d`.

→ Good thing we understand pointers ;)

Way out: Define our own copy constructor.

Copy constructor

```
class A{
public:
    double* _d;
    A(double d): _d(new double) {*_d = d;}
    A(const A& a){
        _d = new double;
        *_d = *a._d;
    }
    ~A() {delete _d;}
};
void foo(A a){} \\ calls copy constructor

int main(){
    A a(1.234);
    if(true) A b(a); \\ calls copy constructor
    foo(a);
    std::cout<< *a._d <<std::endl;
}
```