Lecture 5: Classes

Johannes Gerstmayr and Markus Walzthöni

Material: Jonas Kusch and Martina Prugger
University of Innsbruck

November 10, 2023

Structs

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

```cpp
#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?

```
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 forward and reverse 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 with constructor:

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

Classes - Functions

→ add a print function

```
#include <iostream >

class Entry{
    private:
        long _data;
        Entry* _next;
        Entry* _previous;
    public:
        Entry(long data): _data(data) {}
        void Print() {std::cout << _data << std::endl;}
        //public, inline member function has access to private _data
};

int main()
{
    Entry first(2);
    first.Print();
}
```

Classes - Functions

```
class Entry{
    private:
        long _data;
        Entry* _next;
        Entry* _previous;
    public:
        Entry(long data): _data(data) {}
        void Print(); //declaration of member function
};

//function implementation (may be very long; causes compile costs; may be defined
void Entry::Print(){
    std::cout << _data << std::endl;
}
```

Classes – Declaration / Implementation

Implementation inside class (`.h` file)

- in case of short code ("return 0;")
- (+) for implicit inline; faster code
- (+) readability: see how it works at one place
- (-) longer compilation time
- (-) no clear separation between declaration and implementation (interface)

Implementation outside class (`.cpp` file)

- (+) reduces compile costs if many headers are imported
- (+) most people do not need to know details of implementation
- (+) parallel compilation of .cpp files

Classes – Declaration / Implementation

Implementation inside class (`.h` file)

- in case of short code ("return 0;")
- (+) for implicit inline; faster code
- (+) readability: see how it works at one place
- (-) longer compilation time
- (-) no clear separation between declaration and implementation (interface)

Implementation outside class (`.cpp` file)

- (+) reduces compile costs if many headers are imported
- (+) most people do not need to know details of implementation
- (+) parallel compilation of .cpp files

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._publicData; //ok
std::cout << first._data;       //boom!
```

- private data/functions are protected from access
- public data/functions are accessible

Classes – What is it?

A class ...

- is a user-defined type
- includes a class-specifier and class name
- requires semicolon after member specification `{ ...};`

members are:

- data members (`int x;`)
- member functions (`int GetX() return x;`)
- nested types (classes, enumerations, typedef)
- enumerators
- member templates (advanced)

Classes – What is it?

A class ...

- is a user-defined type
- includes a class-specifier and class name
- requires semicolon after member specification `{ ...};`

members are:

- data members (`int x;`)
- member functions (`int GetX() return x;`)
- nested types (classes, enumerations, typedef)
- enumerators
- member templates (advanced)

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

```cpp
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;
}
```

`const` qualifier (function arguments)

- arguments may be passed as reference in functions – avoid copying of data
- `const` qualifier ensures no modification inside function

```
//copy a; possibly large data in A (e.g. 1MB):
void f(A a) {
  std::cout << a;
  a = 2*a; //changes a locally (no affect outside)
}
//pass reference to a, no copy:
void f(A& a) {
  std::cout << a;
  a = 2*a; //changes a in code of called function
}
//pass const reference to a, no copy:
void f(const A& a) {
  std::cout << a;
  a = 2*a; //boom! compiler error
}
```

`const` qualifier (class member functions)

- member functions may have `const` after declaration
- `const` functions may not change data members
- `const` functions may not call non-`const` member functions
- **idea**: split up in functionality which changes class data, and code which preserves data
- **2 Ways:**
  - Always `const` / non-`const`
  - no `const` at all (usually no way to switch back to `const` in larger codes)
- `mutable` used to break this rule locally (avoid!!!)

const qualifier (classes)

```cpp
#include <iostream>

class MyClass {
private:
    int value;
public:
    MyClass(int v) : value(v) {}
    // Non-const member function
    void SetValue(int v) {value = v;}
    // Const member function
    int GetValue() const {return value;}
};
int main() {
    MyClass obj(10);
    std::cout << "Initial value: " << obj.GetValue() << std::endl;
    obj.SetValue(20);
    std::cout << "New value: " << obj.GetValue() << std::endl;
}
```

Excercise?

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 for List example

Use the following class `Entry` for the List:

```
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;
}
```

Simple Vector 2D class

Let's look at some more relevant example

- See file `Vector2D.cpp`
- Includes constructor, copy constructor, typical operators

Homework

Extend `Vector2D.cpp`:

- unitary $-$
- L2-Norm
- `operator*` for two vectors
- `operator*`: scalar with vectors
- remove default initialization

Classes: outlook to lecture 6

Next time we will se:

- More constructors
- Typical Operators
- Default constructor, copy operator, move mechanism, etc.
- Inheritance, overloading
- Templates