# TDDE18 & 726G77 Functions & struct

**Christoffer Holm** 

Department of Computer and information science



# Last lecture we learned most things needed to actually write a functioning program.

From now on we only aim to make it easier for us to program!



- 2 More on variables
- 3 More on functions
- 4 Operator Overloading
- 5 Stream flags
- 6 File separation
- 7 Testing
- 8 Time lab



Blocks

{ // start of block

// body of block

} // end of block



Blocks

- In C++, a block is a chunk of code surrounded by { and }.
- It represents a series of statements that are related.
- Good to use for creating sections.
- Is also closely related to a concept called *scope*...



Scope

```
int x{}; // global scope
int main()
{
    int y{}; // local scope
}
```



Scope

- A scope defines when and where a named entity (example: a variable) is accessible
- Two main categories of scope
- global scope is all entities that are accessible everywhere
- *local scope* is when the entity is only accessible in a part of the code (i.e. not everywhere)



```
int x{0};
{
    int x{1};
    {
        cout << x << " ";
        int x{2};
        cout << x << " ";
    }
    cout << x << " ";
}
cout << x << " ";
</pre>
```



Scope & Blocks

```
int x{0};
{
    int x{1};
    {
        cout << x << " ";
        int x{2};
        cout << x << " ";
    }
    cout << x << " ";
}
cout << x << " ";
</pre>
```

\$ ./a.out 1 2 1 0



- A block always opens a new local scope
- Each named entity created inside the scope only exists until the end of the block
- All named entites defined outside of the scope is available as long as no other *more local* named entity has the same name
- Things not created inside a block is automatically defined in the *global scope*



```
int x{0}; // global
int main()
{
    int y{1};
    {
        int z{2};
        cout << x << ' ' << y ' ' << z << endl;
    }
}</pre>
```



- In the previous example:
- x is available in the entire program
- y is available in the entire main block
- z is only available inside the inner block



#### What is the difference between x and y in the previous example? Aren't both available in exactly the same parts of the program?



#### What is the difference between x and y in the previous example? Aren't both available in exactly the same parts of the program?

#### They are not, because of functions!



Functions

- A function is a named block that can be executed (called) in other parts of the program.
- Is only executed when called.
- Used to reduce repetition in the code.



(Tedious) Example

```
#include <iostream>
using namespace std;
int main()
{
   string name1;
   string name2;
   cout << "Person 1, your name: ";
   cin >> name1;
   cout << "Person 2, your name: ";
   cin >> name2;
}
```



(Tedious) Example

- In the previous example we are prompting people to enter their names.
- The code is almost identical for both people.
- Imagine that there are 100 people instead, now this would be annoying to write.
- Even worse: imagine you want to change something in the functionality? **Nightmare**



What are functions?

```
return_type function_name(parameters)
{
   // statements
   return result;
}
```



What are functions?

- A function has a *return type*, a name and *parameters*
- It takes data through the *parameters* and gives us a result by *returning* it
- Through the parameters we specify what we want to send to the function (what data types the values should be, how many values there are etc.)
- The return type specifies what type of data we get back from the function



```
int main()
{
   string name1;
   string name2;
   name1 = read_name(1);
   name2 = read_name(2);
   return 0;
}
```



- A lot to unpack here.
- i is a parameter to read\_name, which means it is a local variable that is only available inside the function.
- We call read\_name by writing its name and then specifying what value i should have inside the function.
- name1 and name2 are only available in main while i and result are only available inside read\_name.



- read\_name gives back a string when it has been called.
- At the end of the function we specify which value should be handed back by *returning* a specific value (in this case whatever happens to be stored inside result).
- This is done with the return keyword.
- In main we are assigning the result of *different* calls to read\_name to the variables name1 and name2.



#### 18/106

#### Functions

- Whenever a function is called, the execution of the program jumps into the called function and executes each line there.
- Once the function returns it takes the value with it and jumps back to the point where the function was called and continue from there.
- main is a function that the operating system calls when you start your program. The value returned from it is code to the operating system that signals how it went (0 if everything went as expected).



Procedure

```
void foo()
{
   cout << "a procedure" << endl;
}</pre>
```



Procedure

- A function that doesn't return a value.
- Doesn't need a return statement.
- Has void as return-type.
- void is not a type we can assign to variables.



Declaration and definition

```
void function(); // declaration
// ...
void function()
{
   // ...
}
```



Declaration and definition

- C++ is processed by the compiler from top to bottom.
- We can tell the compiler that we intend to use a function before we definie it.
- This is done by *declaring* the function.
- Giving the function a body later on is called the functions *definition*.
- This allows us to separate our code.



Declaration and definition

```
void hello(); // declaration
int main()
{
    hello();
}
void hello() // definition
{
    cout << "hello" << endl;
}</pre>
```



















- When passing a value to a function C++ will *copy* that value into the function.
- This means that inside the function you are free to modify the parameter without it changing the value outside of the function.



#### 2 More on variables

- 3 More on functions
- 4 Operator Overloading
- 5 Stream flags
- 6 File separation
- 7 Testing
- 8 Time lab



#### More on variables

Data types

- built-in types
- Object types
- Pointers



#### More on variables

Data types

- built-in types
  - int
  - double
  - bool
  - etc.
- Object types
- Pointers



#### More on variables

Data types

- built-in types
- Object types
  - string
  - struct (today!)
  - class (later)
- Pointers


Data types

- built-in types
- Object types
- Pointers
  - Comes later on!





- This is a perfectly fine example of us storing information about a person!
- However, it might get a bit annoying if we want to store information about more people.



```
string name1{};
string name2{};
int age1{};
int age2{};
cout << "Person 1, enter your name and age: ";
cin >> name1 >> age1;
cout << "Person 2, enter your name and age: ";
cin >> name2 >> age2;
```



# Now imagine 100 people!



31/106



```
string name{};
int age{};
name = "Christoffer";
age = 30;
```









```
struct Person
{
   string name{};
   int age{};
};
Person p;
p.name = "Christoffer";
p.age = 30;
```



Compound data type

Person p1 {"Christoffer", 30};
Person p2 {"Oskar", 31};



Compound data type

Person p1 {"Christoffer", 30};
Person p2 {"Oskar", 31};





Compound data type

```
Person p1 {"Christoffer", 30};
Person p2 {"Oskar", 31};
```

p1.age++;





Compound data type

```
Person p1 {"Christoffer", 30};
Person p2 {"Oskar", 31};
```

p1.age++;





#### 34/106

#### More on variables

- A struct defines a type.
- You can create several variables of a struct that has its own collection of values.
- Such a variable is called an *object*.
- You can access each variable (field) inside the object with the . operator.
- You can also modify these fields as you do with normal variables.



Сору

```
Person teacher{"Christoffer", 30};
Person copied_teacher{teacher};
copied_teacher.age++;
cout << teacher.age << endl;</pre>
```



Сору

- It is possible to copy variables, including structs.
- This will create a new instance which has the same values as the original.
- However, changing the copy will leave the original intact and likewise vice versa.



References

```
string word{"hello"};
string& greeting{word};
greeting = "hi";
cout << word << endl;</pre>
```



References

```
string word{"hello"};
string& greeting{word};
greeting = "hi";
cout << word << endl;</pre>
```

What will be printed?



References

- We can create an *alias* to a variable through references.
- An alias is a different name to the same entity; so in the example we have two names for the same variable: word and greeting (where greeting is the alias).
- This is quite powerful when used together with functions (as we will see later)!



```
string word{"hello"};
string const& greeting{word};
word = "hi"; // works
greeting = "hello"; // Compilation error
```



- Constant references are *aliases* which disallows changes through them.
- This means that we can modify the value through the original variable but not through the alias.
- Useful if we want to have a read-only variant of a variable.



# Rule of thumb: Always add const, and remove it only if you have to modify the value!



- **1** Functions
- 2 More on variables
- 3 More on functions
- 4 Operator Overloading
- 5 Stream flags
- 6 File separation
- 7 Testing
- 8 Time lab



```
void read_name(string& name)
{
    cout << "Your name: ";
    cin >> name;
}
int main()
{
    string my_name;
    read_name(my_name);
    cout << my_name << endl;
}</pre>
```







```
void read_name(string& name)
{
    cout << "Your name: ";
    cin >> name;
}
int main()
{
    string my_name;
    read_name(my_name);
    cout << my_name << endl;
}</pre>
```





```
void read_name(string& name)
{
    cout << "Your name: ";
    cin >> name;
}
int main()
{
    string my_name;
    read_name(my_name);
    cout << my_name << endl;
}</pre>
```





- If a parameter is declared as a reference then it becomes an alias for a variable from outside the scope of the function.
- This means that we can read and modify my\_name from inside the read\_name function by just modifying the name alias.



```
void print(string message)
{
   cout << message << endl;
}
int main()
{
   string my_msg{"Long message!"};
   print(my_msg);
}</pre>
```





```
void print(string message)
{
   cout << message << endl;
}
int main()
{
   string my_msg{"Long message!"};
   print(my_msg);
}</pre>
```





```
void print(string message)
{
   cout << message << endl;
}
int main()
{
   string my_msg{"Long message!"};
   print(my_msg);
}</pre>
```





```
void print(string const& message)
{
   cout << message << endl;
}
int main()
{
   string my_msg{"Long message!"};
   print(my_msg);
}</pre>
```





```
void print(string const& message)
{
   cout << message << endl;
}
int main()
{
   string my_msg{"Long message!"};
   print(my_msg);
}</pre>
```





- Some types, for example string are quite expensive to copy.
- string must copy each character, and if it is a long text that will be quite a lot of copying.
- In that case it might be better to share a variable with a function instead of copying.



- However, it should not be a normal reference since we do not want to accidentally overwrite or change the value of the original variable.
- In that case it is good to use const&.
- **Rule of thumb:** if it is a non-builtin type you should never pass it as a copy, use const& instead.



Function overloading

```
int main()
{
   // will call version 1
   add(1, 2);
   // will call version 2
   add(3.4, 5.6);
}
```



Function overloading

- Functions can have the same name in C++.
- But then the compiler must be able to determine which version should be called.
- This means that the parameters matter.



Function overloading

- The compiler will pick version 1 if we pass in int as parameters and version 2 if we pass in double.
- Each overload must have a unique set of parameter types.


Function overloading

```
// version 1
int add(int a, int b)
{
  return a + b;
}
// version 2
double add(double a,
           double b)
{
  return a + b;
}
```

• Note: the compiler cannot distinguish the return type of the function so the compiler doesn't take it into consideration.



Which version?

triangle\_area(1, 1, 1); triangle\_area(1, 1); triangle\_area(1, 1.0, 1.0); triangle\_area(1, 1, 1.0);



Which version?

triangle\_area(1, 1, 1); // b
triangle\_area(1, 1); // a
triangle\_area(1, 1.0, 1.0); // d
triangle\_area(1, 1, 1.0); // c



#### 49/106

#### More on functions

Which version?

- Note that the compiler looks at the amount of parameters and the types.
- The compiler deduces this information based on the values passed into the function when we are calling it.



Default-parameters

```
void ignore(int n, char stop)
{
    cin.ignore(n, stop);
}
```

ignore(100, ':');



Default-parameters

```
void ignore(int n)
{
    ignore(n, '\n');
}
```

ignore(100, ':');
ignore(100);



Default-parameters

```
void ignore()
{
    ignore(1024);
}
```

ignore(100, ':');
ignore(100);
ignore();



Default-parameters

```
void ignore(int n = 1024, char stop = '\n')
{
    cin.ignore(n, stop);
}
```

ignore(100, ':');
ignore(100);
ignore();



Default-parameters

- Sometimes we want optional parameters.
- Useful if there are some default-values we can assign to these parameters, but still want the caller to be able to give their own values.
- One way we can do this is to create different overloads where some parameters are missing.
- However this gets tedious pretty quickly.
- Therefore we can use something called *default-parameters*.



Default-parameters

- default-parameters must be at the end of the parameter list.
- They are declared by assigning a default value to the parameter in the parameter list.
- The compiler will match the parameters from left to right, meaning we can only have optional parameters in a sequence at the end of the list.
- Default parameters should only be in the declaration, not the definition.



Default-parameters

```
void ignore(int n = 1024, char stop = '\n');
int main()
{
  ignore(100, ':');
  ignore(100);
  ignore();
}
void ignore(int n, char stop)
{
  cin.ignore(n, stop);
```



- 1 Functions
- 2 More on variables
- 3 More on functions

- 5 Stream flags
- 6 File separation
- 7 Testing
- 8 Time lab



Example

```
struct Person
{
   string first_name;
   string last_name;
};
```



Example

```
int main()
{
    Person p1{"Christoffer", "Holm"};
    Person p2{"Klas", "Arvidsson"};
    if (p1.first_name < p2.first_name)
    {
        cout << p1.first_name << " "
            << p1.last_name << endl;
    }
}</pre>
```



Easier way

```
int main()
{
    Person p1{"Christoffer", "Holm"};
    Person p2{"Klas", "Arvidsson"};
    if (p1 < p2)
    {
        cout << p1 << endl;
    }
}</pre>
```



Easier way

- We can define how the normal operators are supposed to work with our struct.
- This allows us to create code that is easier to understand,
- since now we can specify for example what is means to see if one person is < another.</li>
- This is useful to determine how these objects could be sorted.



To make it work

```
bool operator<(Person const& p1, Person const& p2)
{
   return p1.first_name < p2.first_name;
}</pre>
```



To make it work

- Not all operators can be overloaded.
- Here is a list: https://en.cppreference.com/ w/cpp/language/operators
- An operator overload is just a function with a special name.
- Every operator is defined with the name operator followed by the operator you wish to overload.
- For example operator+, operator==, operator<< etc.



To make it work

- The type of the first parameter to the operator is usually the object you want to overload this operator for.
- The return type and the rest of the arguments depend on the specific operator.
- Two types of operator: Unary and Binary.



How does it work?



How does it work?

if (operator<(p1, p2))
{
 // ...
}</pre>



How does it work?

- When the compiler sees an expression involving an operator it will look for a function with the special operator name.
- If it exists, the compiler will the translate the operator expression to a function call to that special operator function.
- Note that normal function rules apply to operators as well.



Binary operator

Му_Туре	a;
My_Type	b;
a+b;	
a <b;< td=""><td></td></b;<>	
a==b;	



Binary operator

My\_Type a; My\_Type b; a+b; a<b; a==b;

My\_Type a; My\_Type b; operator +(a, b); operator <(a, b); operator==(a, b);



Binary operator

- Binary operators are those operators that involves two values (a and b in the previous example).
- These operators take two parameters: the first corresponds to the value to the left of the operator while the second corresponds to the value right of the operator.
- The return type can be whatever you want, but it should make sense!



#### Unary operator

Му_Туре	a;
-a;	
++a;	
a++;	



Unary operator

My\_Type a; -a; ++a; a++; My\_Type a; operator-(a); operator++(a); operator++(a);



#### Unary operator

#### My\_Type a; -a; ++a; a++;

- ++a and a++ are not the same expression.
- So their operator-overload should be different.
- But how?
- C++ has a solution.



Unary operator

My\_Type a; -a; ++a; a++; My\_Type a; operator-(a); operator++(a); operator++(a, 0);



Unary operator

Му_Туре	a;
-a;	
++a;	
a++;	

- The compiler adds a 0 as the second parameter to the postfix-version of all increment and decrement operators.
- This is only so that the operator overloading between these versions can be distinguished.
- The 0 does not mean anything.



```
struct My_Int
{
    int data;
};
My_Int& operator++(My_Int& i);
My_Int operator++(My_Int& i, int);
```



```
My_Int& operator++(My_Int& i)
{
    ++i.data;
    return i;
}
```



```
My_Int operator++(My_Int& i, int)
{
    My_Int tmp{i};
    ++i;
    return tmp;
}
```



- Prefix increment (and decrement) will increment (or decrement) the value and then return the new value.
- C++ dictates that the return type should be a reference in this case, as to reduce copies.
- Postfix increment (and decrement) will increment (or decrement) the value and then return the previous value.
- Therefore we must return a copy of the object since the original object has changed value.



**Operator Overloading** 

- It is a good idea to add your logic in as few operators as possible and then reuse these operators to implement the others.
- In the previous example we implemented the increment logic in the prefix-increment operator and then in the postfix-increment operator we simply use the prefix-version.
- This way if we have to change the behaviour it is enough to do it in one place.



Overloading printing operators

Person p1{"Christoffer Holm"};
cout << p1 << endl;</pre>


Overloading printing operators

Person p1{"Christoffer Holm"}; ((cout << p1) << endl);</pre>



Overloading printing operators

Person p1{"Christoffer Holm"}; (operator<<(cout, p1)) << endl);</pre>



Overloading printing operators

Person p1{"Christoffer Holm"};
operator<<(operator<<(cout, p1), endl);</pre>



Overloading printing operators

Person p1{"Christoffer Holm"};
operator<<(operator<<(cout, p1), endl);</pre>

#### What should our operator << return to make it work?



What is cout?

- The type of cout is ostream.
- We need to capture cout and return it in our operator<<.</li>



Overloading printing operators

```
ostream& operator<<(ostream& os, Person const& p)
{
    os << p.first_name << " " << p.last_name;
    return os;
}</pre>
```



This is called chaining

Person p1{"Christoffer Holm"}; cout << p1 << endl;</pre>



This is called chaining

Person p1{"Christoffer Holm"}; ((cout << p1) << endl);</pre>



This is called chaining

Person p1{"Christoffer Holm"}; (operator<<(cout, p1)) << endl);</pre>



This is called chaining

Person p1{"Christoffer Holm"};
cout << endl;</pre>



Overloading reading operator

Person p; int x; cin >> p >> x;



Overloading reading operator

Person p; int x; ((cin >> p) >> x);



Overloading reading operator

Person p; int x; ((operator>>(cin, p)) >> x);



Overloading reading operator

Person p; int x; operator>>((operator>>(cin, p), x);



Overloading reading operator

- cin is of type istream.
- Just as with the printing operator, we want chaining for our operator.
- We are reading into variables, so every parameter should be a reference.



Overloading reading operator

```
istream& operator>>(istream& is, Person& p)
{
    is >> p.first_name >> p.last_name;
    return is;
}
```



#### 1 Functions

- 2 More on variables
- 3 More on functions
- 4 Operator Overloading
- 5 Stream flags
- 6 File separation
- 7 Testing
- 8 Time lab



What happens?

```
int x;
string word;
cout << "Enter int: ";
cin >> x;
cout << x << endl;
cout << "Enter word: ";
cin >> word;
cout << word << endl;</pre>
```



What happens?

```
int x;
string word;
cout << "Enter int: ";
cin >> x;
cout << x << endl;
cout << "Enter word: ";
cin >> word;
cout << word << endl;</pre>
```

```
Enter int: 5
5
Enter word: hello
hello
```



What happens?

```
int x;
string word;
cout << "Enter int: ";
cin >> x;
cout << x << endl;
cout << "Enter word: ";
cin >> word;
cout << word << endl;</pre>
```

Enter int: a 0 Enter word:



What happens?

```
int x;
string word;
cout << "Enter int: ";
cin >> x;
cout << x << endl;
cout << "Enter word: ";
cin >> word;
cout << word << endl;</pre>
```

Why does this happen?



Why does this happen?

- If an operation fails; in this case trying to read an int but finding the letter 'a' instead,
- then an error flag is raised inside the stream,
- as long as this flag is raised the operations will immediately fail meaning nothing will happen.



What flags are there?

- fail Stream operation failed
- eof device has reached the end
- bad irrecoverable stream error
- good no errors



What flags are there?

- Multiple flags can be set at once,
- except good; it is set when no other flag is set.
- This means that several errors can occur at once
- Do note that these flags are set *after* a stream operation fails.
- The stream does not magically detect an error if no operation has been performed.



So how do we fix it?

```
int x;
string word;
cin >> x;
cin.clear();
cin >> word;
```



So how do we fix it?

- cin.clear() will clear any and all flags that are raised.
- You can also check if a specific flag is raised:



Checking for specific flag

```
if (cin.fail())
{
    // the fail flag
}
if (cin.eof())
{
    // the eof flag
}
if (cin.bad())
{
    // the bad flag
}
```



Setting the flags

cin.setstate(ios\_base::failbit); cin.setstate(ios\_base::eofbit); cin.setstate(ios\_base::badbit); cin.setstate(ios\_base::goodbit);



Setting the flags

- When creating your own operator>> you may want to set error flags.
- This is done with cin.setstate.
- You can set the flags by passing in specific values called failbit, eofbit, badbit and goodbit.



#### **1** Functions

- 2 More on variables
- 3 More on functions
- 4 Operator Overloading
- 5 Stream flags
- 6 File separation
- 7 Testing
- 8 Time lab



Modular thinking

- Related functions can be gathered inside a file.
- This is called a *module* (Not to be confused with the new *C++20 Modules*).
- Modules can be compiled separately from the main program, this will result in an *object file*.
- The declaration of functions that should be available in your module are placed in a *header file*.



#### 86/106

#### File separation

Modular thinking

- The actual definition of these functions are placed in an *implementation file*.
- Header files and implementation should have the same name, except for the file extension.
- Then all modules can be compiled together with your program to create an *executable file*.



- Implementation files (.cc)
- Executable files



- Implementation files (.cc)
- Executable files
- Header files (.h)



- Implementation files (.cc)
- Executable files
- Header files (.h)
- Object file (.o)



- Implementation files contains definitions.
- Header files contains declarations.
- Executable files are the actual programs the computer can run.
- Object files are smaller parts of the program that has been precompiled. They cannot run on their own, but can be combined together to create an executable file.


#### Example

#### test.h

#ifndef TEST\_H
#define TEST\_H
void test(int x = 0); // declaration
#endif/TEST\_H

#### test.cc

```
#include "test.h"
#include <iostream>
using namespace std;
void test(int x) // definition
{
    cout << x << endl;
}</pre>
```

#### main.cc



#### terminal

- \$ g++ test.cc main.cc
  \$ ./a.out
  0
- 1



Example

- test.h has what is known as a header-guard.
- #ifndef TEST\_H means: if the symbol TEST\_H is not defined, then compile everything, otherwise skip to the #endif and continue compilation from there.
- #define TEST\_H creates the symbol TEST\_H.
- This is done to ensure that we doesn't accidentally declare the same things twice. Which happens if we accidentally include the same file more than once.



Example

- #include "test.h" is replaced with the content of the
   test.h file.
- This is how importing modules works, the compiler will recieve a list of declarations (i.e. functions, structs and other things that will be available).
- Note: these things are only declared, not defined.



#### 90/106

#### File separation

Example

- The definition of these things are done inside a separate implementation file (test.cc in this example).
- We also have the main.cc file which contains our program that uses the test module.
- Both of these files includes the header file (test.h) but only one of them gives the definition of the things declared.



Dependency graph





Dependancy graph

- In the previous image we can see how it all comes together.
- test.cc includes test.h
- main.cc includes test.h and iostream
- These cc-files can then be compiled together to create an executable file a.out
- We have two modules: test and main



#### 1 Functions

- 2 More on variables
- 3 More on functions
- 4 Operator Overloading
- 5 Stream flags
- 6 File separation
- 7 Testing
- 8 Time lab



- When writing modules we should test them before we use them.
- This is to make sure that they work in all cases.
- This can be done manually by creating a main-program that allows the user to enter inputs that tests each functionality.
- However this is tedious work which we probably can do better, and more accurately.



Testing modules

```
#include "Person.h"
#include <iostream>
using namespace std;
int main()
{
    Person p1{"a", "a"};
    Person p2{"b", "b"};
    Person p3{"a", "a"};
    if (p1 < p2)
    {
        cout << "operator< works!" << endl;
     }
    if (p1 == p3 && p1 != p2)
    {
        cout << "operator== works!" << endl;
    }
}</pre>
```



Testing modules

- We should always test all functionality in our modules.
- This can be done by writing alot examples where we test various functions and functionality.



```
#include "Person.h"
#include <iostream>
using namespace std;
int main()
{
    Person ans{"Christoffer", "Holm"};
    Person p;
    cout << "Enter 'Christoffer Holm': ";
    cin >> p;
    if (p == ans)
    {
        cout << "operator>> works!" << endl;
    }
}</pre>
```



```
#include "Person.h"
#include <iostream>
winclude <sstream>
using namespace std;
int main()
{
    Person ans{"Christoffer", "Holm"};
    Person p;
    istringstream iss{"Christoffer Holm"};
    iss >> p;
    if (p == ans)
    {
        cout << "operator>> works!" << endl;
    }
}</pre>
```



- You could test stream operators by letting the user enter appropriate data and see that it works.
- However, after a while this gets tedious since we have to test all cases over and over again, writing the same things over and over again into the terminal.
- What we can do then is to simulate cin and cout to automatically test operator>> and operator<<.



- cin can be simulated with istringstream.
- It is defined in <sstream>.
- It works exactly like cin, with the difference that you specify what has been entered into the simulated cin when you create istringstream.



```
#include <iostream>
#include <iostream>
winclude <sstream>
using namespace std;
int main()
{
    Person p{"Christoffer", "Holm"};
    ostringstream oss{};
    oss << p;
    if (oss.str() == "Christoffer Holm")
    {
        cout << "operator<< works!" << endl;
    }
}</pre>
```



- You can simulate cout with ostringstream.
- It works exactly like cout with the exception that it prints to a string instead of to the terminal.
- This allows us to retrieve the written output with oss.str() and to check that it was correct.



```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
TEST_CASE("testing < and ==")
{
    Person p1{"a", "a"};
    Person p2{"b", "b"};
    Person p3{"a", "b"};
    CHECK(p1 == p3);
    CHECK(p1 == p2);
    CHECK_FALSE(p1 == p2);
    CHECK_FALSE(p2 < p1);
}</pre>
```



```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
TEST_CASE("testing < and ==")
{
    Person p1{"a", "a"};
    Person p2{"b", "b"};
    Person p3{"a", "b"};
    REQUIRE(p1 == p3);
    REQUIRE(p1 <= p2);
    REQUIRE(p1 <= p2);
    REQUIRE_FALSE(p2 < p1);
}</pre>
```



- catch.hpp is a testing framework (a module) that we will use in this course.
- It makes it a lot easier for us to test our modules.
- The framework will create a good main-function for you with tests and nice output messages.
- All you have to do is create the test cases.



- To start using catch.hpp you have to download the header file catch.hpp from here: https://github.com/catchorg/Catch2
- Place the file in the same directory as your program.
- Define the symbol CATCH\_CONFIG\_MAIN and then include catch.hpp.



- To create a test case you use the command TEST\_CASE which takes a string that is supposed to contain a short description of what this test case is testing.
- Then you open a block and everything inside this block will be bundled together as one testcase.
- Then you add conditional statements which you surround with either the keyword CHECK or REQUIRE.



- CHECK simply tests whether the statement inside is true (there is a CHECK\_FALSE version that tests if it is false).
   Either way, once it has been tested it will move on to the next test.
- REQUIRE works the same way, except that it will stop all tests if this one fails.
- That is all the basic tools you need to use catch.hpp.



#### 1 Functions

- 2 More on variables
- 3 More on functions
- 4 Operator Overloading
- 5 Stream flags
- 6 File separation
- 7 Testing
- 8 Time lab



## Time lab

Lab 2

- The Time lab (lab2) covers everything we have discussed in these lectures: struct, functions, operator overloading, testing etc.
- It is a lot harder than lab 1 so plan your time accordingly.
- The goal of the lab is to create your own module.
- That means you are not creating a program but instead a smaller part of a program that can be used for many different programs.
- Testing is a part of the lab so we expect you to write good testcases.



#### Time lab

Labs

- Lab1 Deadline: September 10th
- Lab2 Deadline: September 25th
- Complementary work



# Time lab

Teaching session

- First teaching session: September 13th at 08:15-10:00
- There will be one session in English (always given in the highest numbered room in TimeEdit)
- Content will be about lab 2





