Lecture 2

- Basics review -- part of previous courses
  - Structures ***(struct)*** *[sec. 15.3]*
  - Functions overloading *[end of sec. 4.3]*
  - *Preparation for lab 1, exercise 2*

Structures *(posttyp)*

- Programmer can define new data types: structures and classes
- Structures (and classes) are used to store data of different types as a unit

```
struct Salesman
{
    string name;
    double sales; //total of sales
};
```

```
Salesman S = {"Tim XX", 1500};
cout << S.name << " " << S.sales;
```

S

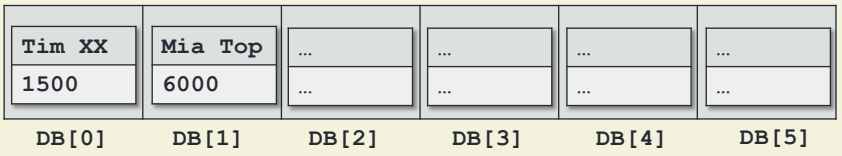
name	Tim XX
sales	1500

Sec. 15.3
about **struct**

Structures *(posttyp)*

```
struct Salesman
{
    string name;
    double sales; //total of sales
};
```

```
Salesman DB[6];
DB[0].name = "Tim XX";
DB[0].sales = 1500;
DB[1].name = "Mia Top";
DB[1].sales = 6000;
```



Available operations

- Assignment

```
Salesman S1 = {"Tim Covenant", 1000};
Salesman S2;
S2 = S1;      //ok!
```

- Comparison operators: **==**, **>**, **<**, *osv*
  - Not available!!
  - But, these operations can be programmed …

```
//lexicographical comparison of names
if (S1 < S2)  //does not compile!
  … ;
```

Which functions do we want to have for `Salesman`?

- **To read** a salesman's data
- **To write** (display) a given salesman's data
- **To compare** two salesmans by name

Operator overloading

```
int i, k;
cin >> i >> k;
cout << i << " " << k;
if (i > k) …
```

`cin >> i;` translated to `operator>>(cin, i)`

`cout << i;` translated to `operator<<(cout, i)`

```
Salesman S1, S2;
cin >> S1 >> S2;
cout << S1 << endl << S2;
if (S1 > S2) …
```

`Salesman` resembles an existing type of C++ (e.g. an **`int`**)

Operators are functions whose name is a special symbol like

`>>`	`<<`	`*`	`=`
`+`	`+=`	`-`	`-=`
`[]`	`>`	`<`	`==`

Functions overloading

- Functions with same name -- (doing the same conceptual task)
- Different types of arguments

```
void display(int V[], int n)
{ …; }
void display(const Salesman& S)
{ …; }
```

```
int A[10] = {0};
display(A, 10);
Salesman tim = {"Tim", 100};
display(tim);
```

Compiler looks at the arguments types to decide wich function to call

Functions overloading

Standard library offers many definitions for the same functions

```
bool operator> ( const string& lhs, const string& rhs );
bool operator> ( const char* lhs, const string& rhs );
bool operator> ( const string& lhs, const char* rhs );
```

S1 (**S2**) cannot be changed by the function

S1 (**S2**) passed by reference -- **eficiency**

```
bool operator>(const Salesman& S1, const Salesman& S2)
{
   return (S1.name > S2.name); //alphabetical order
}
```

We can define our own comparison operator for **Salesman**

Bubble sort

```
void bubbelsort(Salesman V[], int n)
{
    Salesman temp;
    for ( int pass = 0; pass <= n - 2; pass++ )
        for (int i = 0; i <  n - 1; i++ )   // one pass
        {
            if (V[i] > V[i + 1])  //compare
            {
                //swap
                temp = V[ i ];
                V[ i ] = V[ i + 1 ];
                V[ i + 1 ] = temp;
            }
        }
}
```


Operator/functions overloading

```
//member functions of class istream
istream& operator>> (int& val);
istream& operator>> (double& val);
//global functions
istream& operator>> (istream& is, string& str );
…
```

istream::getline

```
istream& getline (char s[], streamsize n );
istream& getline (char s[], streamsize n, char delim );
```

Global functions part of the **Standard Library**

```
istream& getline (istream& is, string& str );
istream& getline (istream& is, string& str, char delim );
```

- Functions with same name but different arguments

Overloading `operator>>`

```
string s;
cin >> s;
```

Translated to
`operator>>(cin, s);`

- **`istream`** class defines streams for reading input
- **`cin`** is an object of the standard class **`istream`**

Function returns the stream after reading

stream objects must be passed by reference

```
istream& operator>>(istream& in, string& str)
{
  //read from stream in to string str
  return in;
}
```


Overloading `operator>>` for `Salesman`

```
Salesman S1;
cin >> S1;
```

Translated to
`operator>>(cin, S1);`

Input

```
1500      Tim Tiberland\n
700        Erik Nord\n
20.5674    Carl Sharif Ole\n
```

```
istream& operator>>(istream& in, Salesman& S);
```

Now, we just need to implement the function **`operator>>`** like any other function

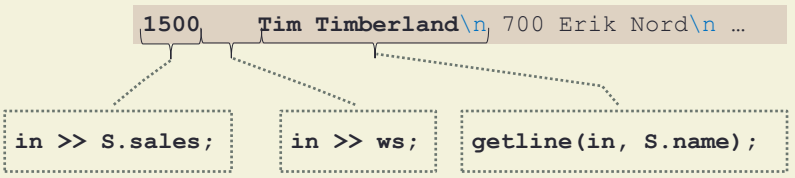
Overloading `operator>>` for `Salesman`

Define a stream extraction operator for **`Salesman`**

```
istream& operator>>(istream& in, Salesman& S)
{
    in >> S.sales; //read sales
    in >> ws;
    getline(in, S.name); //read name

    return in;  //return the stream
}
```

Input stream's buffer



Overloading `operator>>`

```
Type x;
cin >> x;
```

Translated to

```
operator>>(cin, x);
```

An overloaded declaration for **`operator>>`** should look like

```
istream& operator>>(istream& in, Type& x);
```

Reading **Salesman** data from **cin**

Define a stream extraction operator for **Salesman**

```
istream& operator>>(istream& in, Salesman& S)
{
    in >> S.sales >> ws; //read sales
    getline(in, S.name); //read name
    return in;
}
```

```
const int SIZE = 100;
Salesman DB[SIZE];
int howMany = 0;
while ( howMany < SIZE && (cin >> DB[howMany]) )
      ++howMany;
```

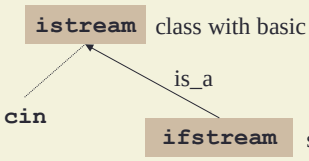
are there free slots in the array?

Reading **Salesman** data

Define a stream extraction operator for **Salesman**

```
istream& operator>>(istream& in, Salesman& S)
{
    in >> S.sales >> ws; //read sales
    getline(in, S.name); //read name
    return in;
}
```

See **http://www.cplusplus.com/reference/iolibrary/**



Reading **Salesman** data from a text file

```
#include <fstream>
const int SIZE = 100;
Salesman DB[SIZE];
int howMany = 0;
ifstream file("sales_data.txt");
if ( !file )
{
   cout << "Data file not found!!" << endl;
   return 0;
}
while ( (howMany < SIZE) && (file >> DB[howMany]) )
        ++howMany;
```

Create a stream and connect it to the file

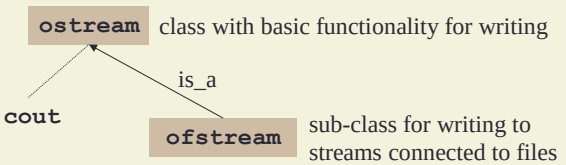
Test if connecting stream to file succeeded

ifstream is a sub-class of **istream**

Overloading **operator<<** for **Salesman**

```
Salesman S1;
cout << S1;
```

Translated to
`operator<<(cout, S1);`



Define our own stream insertion operator for **Salesman**

```
ostream& operator<<(ostream& out, const Salesman& S);
```

One program, several files

- In **C++**, a program can be divided by several files

1. Header files `salesman.h`

It contains function declarations (no implementation)

It is an *specification* for the new data type

Cannot be compiled

2. Definition files `salesman.cpp`

It contains the definition of the functions declared in the header file

No **`main()`** function

Can be compiled separately

3. Main program file `sales.cpp`

It is a program with a **`main()`** function

It may contain other functions

Can be compiled and executed

Uppdelning av Program

Study
`salesman.h`
`salesman.cpp`
`sales.cpp`
Create a project in Code::Blocks

Useful for Lab 1

Read sec. 4.5

Binary search is used in the program **`sales.cpp`**
Binary search can only be used for sorted arrays

Binary search
See pag. 112

Next…

- Fö 3
  - Pointers *(pekare)* *[sec. 5.4.1 – 5.4.3]*
    - Pointers and arrays
    - Pointer arithmetic

- Do Lab 1
  - Study **`salesman_V1.cpp`** for exerc. 2