Lecture 1

- Basics review -- part of previous course TND012
- Streams
  - stream insertion **`operator>>`**
  - stream extraction **`operator<<`**
  - automatic conversion of a stream to **`bool`**
  - NEW – Re-direction of standard input/output -- *[pag. 72]*
    (*omdirigering av* ***`cin`***/***`cout`***)
  - to read a text file
  - NEW – in-memory I/O (string stream processing)

The basics: built in data types

```
int age = 5;

double sum = 0;

const int MAX_GRADE = 5;
```

Declaration of variables and constants

Variables have types

Values	Type	Number of bytes
Characteres	`char`	1 byte

Integers	`int`
	`long int`
	`long long int`
		Machine dependent
		(`int` usually 4 bytes)

Reals	`float`
	`double`
	`long double`
		Machine dependent
		(`double` usually 8 bytes)

Booleans	`bool`	1 byte

Variables (constants)

```
int age;
age = 4;
```

Each variable has some bytes in the memory reserved for it

same as

```
int age = 4;
```
-- declare and initialize

- All variables have
  - A name **age**
  - A type **int**
  - A size in bytes 4 bytes
  - A memory address 5
  - A value **4**

Main memory

0	00001011
1	10101011
2	11000001
3	11110011
4	10101010
5	00000000
6	00000000
7	00000000
8	00000100
9	00001011
10	00101011
11	00101011

age

The basics

- Variables can only store values withing a certain range
  - Why?
  - Finite number of bytes used to represent an **int** (**double**)
  - What´s the largest(smallest) **int** (**double**) representable on my machine?
  - Can all real numbers in [1,2] be represented?

Libraries **<climits>** and **<cfloat>** can help us to answer these questions

See **TNG033first.cpp**

First example

- Write a program that displays the sum of a user given sequence of integers
  - End of input indicated by a non-numeric valued, e.g. "**STOP**"

See **sum.cpp**

Input and Output

```
#include  <iostream>
using namespace std;
```

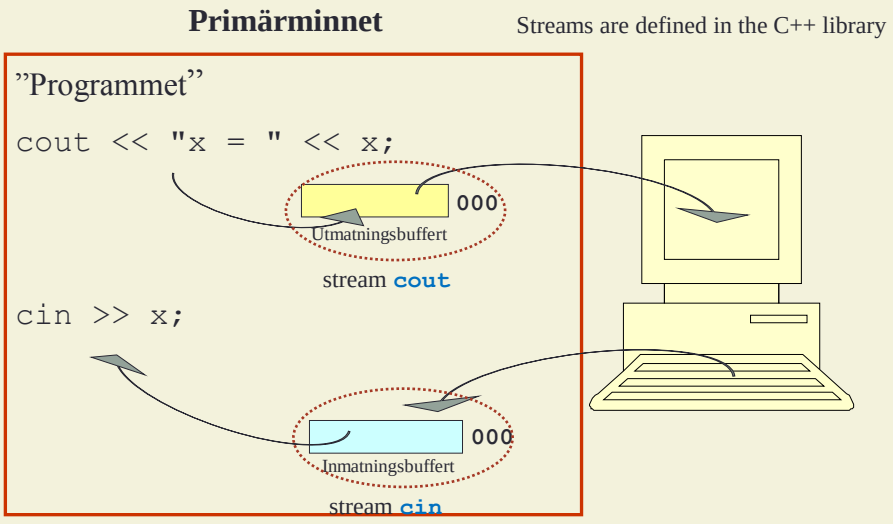
- The following stream objects are declared
  - **cin** -- input stream connected to the keyboard

    cin >> i; -- **stream extraction operator**

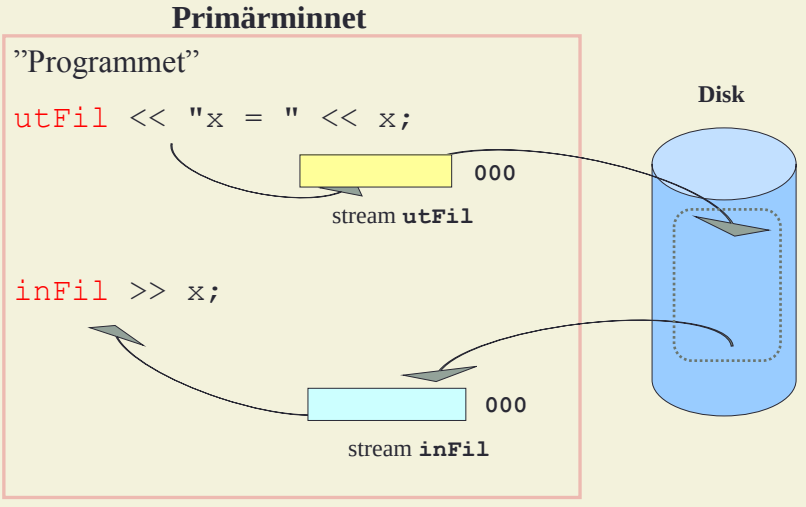
  - **cout** -- output stream connected to the screen

    cout << sum; -- **stream insertion operator**

Streams: cin and cout



Streams and Files



C++ streams philosophy

- What can be done with streams (available operations)?
  - **Open** a stream: connect a stream with external device (e.g. file)
  - **Close** the stream: disconnect the stream from the external device
  - **Read** from the stream
  - **Write** to the stream
  - **Test** if the stream is in "*good state*"

- If an error occurs during open/read/write then one of the stream's error bits is set to "**1**" -- stream is in "*bad state*"
- Stream in "*bad state*" is automatically converted to **`false`** when used as test condition in a loop or **`if`**-satsen
- Stream in "*good state*" is automatically converted to **`true`** when used as test condition in a loop or **`if`**-satsen

Stream insertion operator

```
//member functions of class istream
istream& operator>> (int& val);
istream& operator>> (double& val);

//global functions
istream& operator>> (istream& is, string& str);
…
```

The stream used for reading is returned

Reading with `operator>>`

```
int i;
cin >> i;
```

What happens if the user types a word "**C++**" or an integer larger than **MAX_INT**?

Possible input errors

- value of wrong type is given by the user
- overflow (underflow)

stream is set to *bad state*

- Program continues executing, even if the stream (**cin**) is set to *bad state*
- Not possible to read from a stream in *bad state*

Reading loop

```
double i;
double sum = 0;
while ( cin >> i )
{
   sum += i;
}
cout << "Sum = " << sum;
```

Read + test

Is the stream in "good state"?

Is the input valid?

```
double i;
double sum = 0;
cin >> i;       //read
while ( cin )   //test
{
   sum += i;
   cin >> i;    //read again
}
cout << "Sum = " << sum;
```

Read a sequence of numbers

End of the input sequence is indicated by a non-numeric value, e.g. '**stop**'

Reading with `getline`

Read pag. 77-78

```
string name;
cout << "Vad heter du? " << endl;
cin >> name;
```

```
Vad heter du?
Aida Nordman\n
```

Reads until white space

```
string name;
cout << "Vad heter du? " << endl;
cin >> ws;
getline(cin, name);
```

Extract and ignore (leading) white spaces

- Reads all characters until end of the line ('`\n`')
- '`\n`' is extracted from the stream's buffer but not added to `name`

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

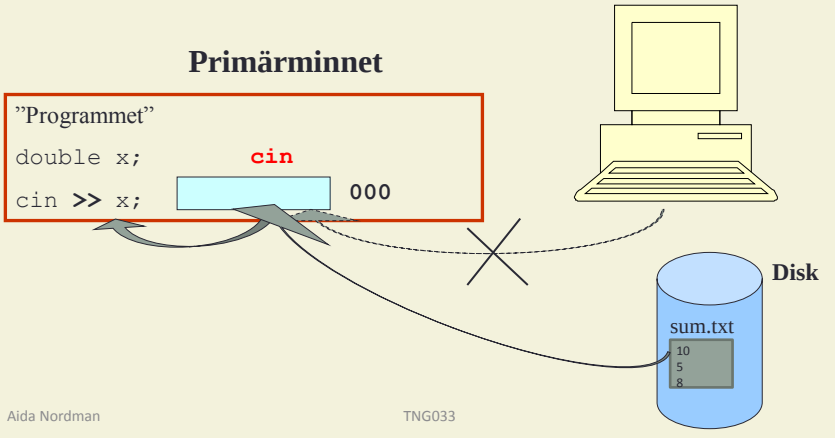

Redirection of input and output

NEW

- **Redirection of input** (DOS and Unix)

  `sum < sum.txt`

  reads input from file `sum.txt`



Redirection of input and output NEW

Read pag. 72-73

- **Redirection of input** (DOS and Unix)

  `sum < input.txt`

  reads input from file `input.txt`

- **Redirection of output**

  `sum > output.txt`

  writes output to file `output.txt`

- **Redirection of input and output**

  `sum < input.txt > output.txt`

There is an executable **`sum.exe`**

See instructions in the appendix of **Lab1**

End of Part I

Second example

- Write a program that displays the sum of the integers given in a text file

See **sum_file.cpp**

Open files for reading

Library for working with files
ifstream defined in **<fstream>**

1. **Declare stream variable**, `inFil`
2. **Open the stream**: associate the stream with file to be read

```
#include <fstream>
string file_name;
cout << "File name: ";
cin >> file_name;
ifstream inFil(file_name);        //C++11
if ( !inFil )
      cout << "File could not be opened!!";
else
    //read the file;
```

See Fö12, TND012

Test if the it was possible to connect the stream to the text file

C++ streams philosophy

- If an error occurs during open/read/write then one of the stream's error bits is set to "**1**" -- stream is in "*bad state*"
  - Attempting to connect an **`ifstream`** with a non-existent file sets stream to "*bad state*"
  - Attempting **to read** when end-of-file (**`EOF`**) has been reached sets stream to "*bad state*"

Reading a text file

Note: your program must also work for an **empty text file**

Test if last reading operation suceeded (check if the stream is in "*good state*")
Stream is converted to a **`bool`** (i.e. **`true`** or **`false`**)

```
//Read first
while ( inFil )
{
        …; //do something with the data read
        …; //read again
}
```

If **`EOF`** the stream **`inFil`** is set to "*bad state*"

Text file (stream's buffer):

111	*222*	*333*	**EOF**

String stream processing (in memory I/O)

NEW

- Streams are usually connected to external devices, e.g. keyboard, file
- But, streams can also be connected to strings
  - **`istreamstring`**-- read data from a string
  - **`ostreamstring`**-- write data to a string
  - **`stringstream`** -- read and write data to a string

Read sec. 11.9

istringstream

```
#include <sstream>
string s = "34.5  12";
istringstream is(s);        -- declaration+initialization
double d;
is >> d;                    -- convert text to double, d ← 34.5
string s1;
getline(is,s1);             -- s1 ← ”  12”
if ( is.eof() )             -- the end of the string is interpreted by
                               the istringstream object as end-of-file
is.clear()                  -- re-set all error flags to zero
```

Lab 1, exercise2

istringstream

- Usages
  - check that input data has the correct format
  - easy to do type conversions from text

Validation function:
- Scrutinizes the contents of input data (line)
- Repairs the data, if necessary

```
getline(inFil, line);
while ( inFil )
{
      if ( !line_validation(line) )
      {
        //Correct the data, if necessary
      }
      else
      {
        //Do something with the data
      }
      getline(inFil, line);  //read another line
}
```


Exercise

- Write a program that reads every line of a text file and does the following steps
  1. Test if the line has the format "*n1 op n2*" -- e.g. **4 + 6**
     -- *op* can be "**+**", "**-**", "*****", and "**/**"
  1. If correct format then perform the operation and write the result to text file **results.txt** -- e.g. "**4 + 6 = 10**"
  2. If not correct format then write the line number and the line to text file **errors.txt**
- Note operands can be any number, **int** or **double**, in scientific or fixed point notation
- Use string stream processing
- Repeat the exercise and do not use string string streams (**istringstream**)

See **streamStrings_1.cpp**

ostringstream

```
#include <sstream>
double d = 189;
ostringstream os;            -- declaration
//Get the text format of d in scientific notation
os << scientific << d;
string s1 = os.str();
```

get the content of the stream buffer as a string – "**1.890000e+002**"

- Usages
  - to convert to text format
  - take advantage of the powerful formatting capabilites of C++ streams

Exercise

- Write a function named **convert** that given a number (**double**) returns a string corresponding to the number in scientific notation

```
convert(189) → "1.890000e+002"
```

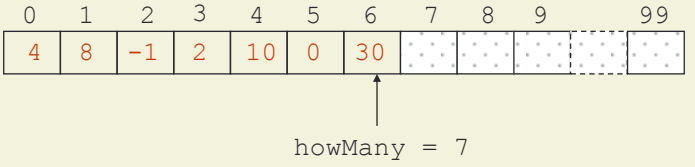
```
string convert(double d)
{
    ostringstream os;

    os << scientific << d;

    return (os.str());
}
```

End of Part II

Arrays (fält)




```
const int SIZE = 99;
int seq[SIZE] = {0};
int howMany = 0;
```

Number of array slots used

Array size must be an integer **constant**

The number of slots in the array must be known at compile time

Make sure in your code that

`howMany ≤ SIZE`

Exercise

- Write a program that performs the following steps
  1. Read a sequence of numbers until the user enters a non-numeric value (e.g. "**STOP**")
  2. Display the sequence, 5 values per line
  3. Display the smallest and largest values in the sequence
  4. Display the sum of the values in the sequence

See **array.cpp**

Read a Sequence into an Array

```
const int SIZE = 100;
double seq[SIZE] = {0};
int howMany = 0;

while ( cin >> seq[howMany] )
{
        howMany++;
        if (howMany == SIZE) break;
}
```

Read + Test

Does the array have any *free* slots?

Exit the loop

End of the input is indicated by a non-numeric value

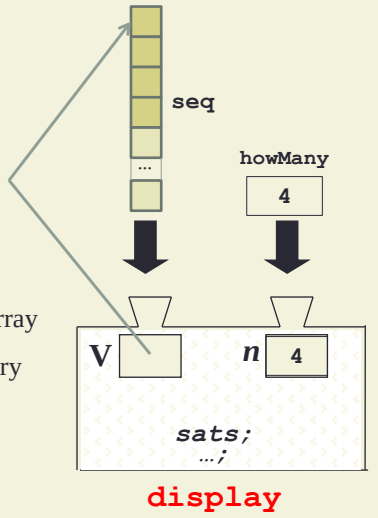
Lab 1, exercise 1

Call-by-reference *versus* call-by-value

```
void display(int V[], int n)
{
  for(int i = 0; i < n; i++)
       cout << V[i];
}
```

An extra parameter is needed for the number of slots being used

```
display(seq,howMany);
```



Arrays are passed by reference to functions

Reason is **efficiency**

V has access to the array

V contains the memory address of **seq**

V[0] = 10; modifies array **seq**

Constant arrays as function arguments

Safer code: array **V** is treated as a constant inside the function

```
void display(const int V[], int n)
{
  for(int i = 0; i < n; i++)
       cout << V[i];
}
```

Good programming practice

Combine the efficiency of call-by-reference with the safety of call-by-value

Next …

- Fö 2
  - Defining new data types (**`struct`**) *[sec.15.3]*
  - Functions overloading *[end of sec. 4.3]*
  - NEW Tables *[sec. 5.10.1]*
    (flerdimensionella fält)
- Lab 1
  - Start exercise 1
  - Review ”*uppdelning av en C++ program*” *[sec. 4.5]*
  - TND012 course web page
    - Login: **`TND012`**
    - Password: **`TND012ht2_12`**