

## **An Introduction to C++**

---

### **Part 1**

A review of basic C++

## **Why C++?**

---

- ◆ Generic high level programming
  - ◆ Shorter development times
  - ◆ Smaller error rate
  - ◆ Easier debugging
  - ◆ Better software reuse
- ◆ Efficiency
  - ◆ As fast or faster than FORTRAN
  - ◆ Faster than C, Pascal, ...
- ◆ Job skills
  - ◆ We all need to find a job some day...

## Generic programming

---

- ◆ Print a sorted list of all words used by [Shakespeare](#)

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>

using namespace std;
int main()
{
    vector<string> data(istream_iterator<string>(cin),istream_iterator<string>());
    sort(data.begin(), data.end());
    unique_copy(data.begin(), data.end(),ostream_iterator<string>(cout,"\\n"));
}
```

## Efficiency

---

- ◆ Using efficient C++ techniques
  - ◆ Templates
  - ◆ Expression templates
  - ◆ Template meta programs
  - ◆ “light objects” and inlining
- ◆ Achieve performance
  - ◆ As fast as FORTRAN in normal codes
  - ◆ Faster than FORTRAN in some cases
  - ◆ See <http://www.oonumerics.org/blitz/benchmarks/>

## Job offer from a recent listing

---

- ◆ Position: Quantitative Analyst
- ◆ Skills: Strong academic background in mathematics, theoretical physics, computer science or engineering. Must have excellent computational (C, C++) and number crunching ability essential.
- ◆ Location: New York, USA
- ◆ Start: ASAP
- ◆ Salary: Dep on Experience
  
- ◆ Salary comparison:
  - ◆ Good C++ programmer >100k\$
  - ◆ MIT assistant professor 60k\$

## Which compiler?

---

- ◆ In this course we will learn modern programming techniques in C++ for scientific computing.
  - ◆ We need to use the full C++ standard features
    - ◆ not yet fully supported by most compilers
  - ◆ We need highly-optimizing compilers
  
- ◆ We will use the KAI C++ compiler ([www.kai.com](http://www.kai.com))
  - ◆ best optimizing C++ compiler
  - ◆ nearly full standard support
  - ◆ License available on all workstations/PCs at ETH
  - ◆ KAI will give you a test license for your Linux PC at home
    - ◆ Unlimited ETH-student license available for 50 US\$
  
- ◆ GNU gcc v. 2.95.2 is not sufficiently standard-conforming and optimizing!

## Introduction

---

- ◆ Programming styles
  - ◆ Procedural Programming
  - ◆ Modular Programming
  - ◆ Object Oriented Programming
  - ◆ Generic Programming
  
- ◆ Programming languages
  - ◆ FORTRAN, BASIC
  - ◆ C, Pascal, Modula, FORTRAN 90
  - ◆ C++, Java, Smalltalk, Oberon
  
- ◆ Needs
  - ◆ Generic programming at a high abstraction level
  - ◆ High efficiency
  - ◆ Is this a contradiction?

## Why C++?

---

	C++	C	Java	FORTRAN	FORTRAN 95
Efficiency	√√	√	×	√√	√
Modular Programming	√	√	√	×	√
Object Oriented Programming	√	×	√	×	√
Generic Programming	√	×	×	×	×

## Overview

---

- ◆ This week
  - ◆ A quick overview over procedural programming in C/C++
- ◆ Following weeks
  - ◆ Classes
  - ◆ Inheritance and object oriented programming
  - ◆ Templates and generic programming
  - ◆ Standard C++ library
- ◆ Later in the course
  - ◆ Expression templates
  - ◆ Optimization for scientific simulations

## A first C++ program

---

```
/* A first program for RGP I
   written by Matthias Troyer */
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    cout << "Hello ETH students.\n";
```

```
    // std::cout without the using declaration
```

```
    return 0;
```

```
}
```

- ◆ /\* and \*/ are the delimiters for comments
- ◆ includes declarations of I/O streams
- ◆ declares that we want to use the standard library ("std")
- ◆ the main program is always called "main"
- ◆ "cout" is the standard output stream.
- ◆ "<<" is the operator to write to a stream
- ◆ statements end with a ;
- ◆ // starts one-line comments
- ◆ A return value of 0 means that everything went OK

## A first calculation

---

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
    cout << "The square root of 5 is"
    << sqrt(5.) << "\n";
    return 0;
}
```

- ◆ `<cmath>` is the header for mathematical functions
- ◆ Output can be connected by `<<`
- ◆ Expressions can be used in output statements
- ◆ What are these constants?
  - ◆ `5.`
  - ◆ `0`
  - ◆ `"\n"`

## C++built-in data types

---

- ◆ **Integral**
    - ◆ `short, unsigned short`
    - ◆ `int, unsigned int`
    - ◆ `long, unsigned long`
  - ◆ **Floating point**
    - ◆ `float`
    - ◆ `double`
    - ◆ `long double`
  - ◆ **Character**
    - ◆ `char, unsigned char, signed char`
    - ◆ `wchar`
  - ◆ **Boolean (logical)**
    - ◆ `bool`
- ◆ unsigned means non-negative
  - ◆ the number of bits used by the types is  
`short <= int <= long`  
`float <= double <= long double`
  - ◆ `char` is a 1-byte representations of a character
  - ◆ `wchar` is a multi-byte representation of non-roman characters
  - ◆ `bool` can be true or false

## Literal constants

---

- ◆ Characters
  - ◆ 'a', 'b', ....
  - ◆ special characters:
    - ◆ '\n' ... newline
    - ◆ '\t' ... tab
    - ◆ '\0' ... null character
    - ◆ '\\' ... \
    - ◆ '\'' ... '
    - ◆ '\"' ... "
    - ◆ and more
- ◆ C-style character strings
  - ◆ "Hello"
  - ◆ arrays of chars ending with '\0'
  - ◆ in C++ only used for literal constants
  - ◆ For variables the standard string class is better
- ◆ Integer numbers
  - ◆ 0, -3, ....
  - ◆ long: 0L, 5L, ...
  - ◆ unsigned: 3u, 7U, ...
  - ◆ unsigned long: 1uL, 9Lu, 6UL, ...
- ◆ Floating point numbers
  - ◆ double precision (double)
    - ◆ 3.1415927, 0.
  - ◆ single precision (float)
    - ◆ 4.562f, 3.0F
  - ◆ extended precision (long double)
    - ◆ 6.54498467494849849489L
- ◆ Boolean
  - ◆ true, false

## A more useful program

---

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    cout << "Enter a number:\n";
    double x;
    cin >> x;
    cout << "The square root of "
    << x << " is "
    << sqrt(x) << "\n";
    return 0;
}
```

- ◆ a variable named 'x' of type 'double' is declared
- ◆ a double value is read and assigned to x
- ◆ The square root is printed

## Variable declarations

---

- ◆ have the syntax: `type variablelist;`
  - ◆ `double x;`
  - ◆ `int i,j,k; // multiple variables possible`
  - ◆ `bool flag;`
- ◆ can appear anywhere in the program
 

```
int main() {
...
double x;
}
```
- ◆ can have initializers, can be constants
  - ◆ `int i=0; // C-style initializer`
  - ◆ `double r(2.5); // C++-style constructor`
  - ◆ `const double pi=3.1415927;`

## Implementation-specific properties of numeric types

---

- ◆ defined in header `<limits>`
- ◆ `numeric_limits<T>::is_specialized // is true if information available`
- ◆ most important values for integral types
  - ◆ `numeric_limits<T>::min() // minimum (largest negative)`
  - ◆ `numeric_limits<T>::max() // maximum`
  - ◆ `numeric_limits<T>::digits // number of bits ( digits base 2)`
  - ◆ `numeric_limits<T>::digits10 // number of decimal digits`
  - ◆ and more: `is_signed, is_integer, is_exact, ...`
- ◆ most important values for floating point types
  - ◆ `numeric_limits<T>::min() // minimum (smallest nonzero positive)`
  - ◆ `numeric_limits<T>::max() // maximum`
  - ◆ `numeric_limits<T>::digits // number of bits ( digits base 2)`
  - ◆ `numeric_limits<T>::digits10 // number of decimal digits`
  - ◆ `numeric_limits<T>::epsilon() // the floating point precision`
  - ◆ and more: `min_exponent, max_exponent, min_exponent10, max_exponent10, is_integer, is_exact`
- ◆ first example of templates, use e.g.:
 

```
std::numeric_limits<double>::epsilon()
```

## Advanced types

---

- ◆ **Enumerators** are integer which take values only from a certain set

```
enum trafficlight {red, orange, green};
enum occupation {empty=0, up=1, down=2, updown=3};
trafficlight light=green;
```

- ◆ **Arrays** of size n

```
int i[10]; double vec[100]; float matrix[10][10];
```

- ◆ indices run from 0 ... n-1! (FORTRAN: 1...n)
- ◆ last index changes fastest (opposite to FORTRAN)

- ◆ Complex types can be given a new name

```
typedef double[10] vector10;
vector10 v={0,1,4,9,16,25,36,49,64,81};
vector10 mat[10]; // actually a matrix!
```

## Expressions and operators

---

- ◆ Arithmetic

- ◆ multiplication:  $a * b$
- ◆ division:  $a / b$
- ◆ remainder:  $a \% b$
- ◆ addition:  $a + b$
- ◆ subtraction:  $a - b$
- ◆ negation:  $-a$

- ◆ Increment and decrement

- ◆ pre-increment:  $++a$
- ◆ post-increment:  $a++$
- ◆ pre-decrement:  $--a$
- ◆ post-decrement:  $a--$

- ◆ Logical (result bool)

- ◆ logical not:  $!a$
- ◆ less than:  $a < b$
- ◆ less than or equal:  $a <= b$
- ◆ greater than:  $a > b$
- ◆ greater than or equal:  $a >= b$
- ◆ equality:  $a == b$
- ◆ inequality:  $a != b$
- ◆ logical and:  $a \&\& b$
- ◆ logical or:  $a \|\| b$

- ◆ Conditional:  $a ? b : c$

- ◆ Assignment:  $a = b$

## Bitwise operations

---

- ◆ Bit operations
  - ◆ bitwise not: `~a`
    - ◆ inverts all bits
  - ◆ left shift: `a << n`
    - ◆ shifts all bits to higher positions, fills with zeros, discards highest
  - ◆ right shift: `a >> n`
    - ◆ shifts to lower positions
  - ◆ bitwise and: `a & b`
  - ◆ bitwise xor: `a ^ b`
  - ◆ bitwise or: `a | b`
- ◆ The **bitset** class implements more functions. We will use it later in one of the exercises.
- ◆ Interested students should refer to the recommended C++ books
- ◆ The shift operators have been redefined for I/O streams:
  - ◆ `cin >> x;`
  - ◆ `cout << "Hello\n";`
- ◆ The same can be done for all new types: "operator overloading"
- ◆ Example: **matrix operations**
  - ◆ `A+B`
  - ◆ `A-B`
  - ◆ `A*B`

## Compound assignments

---

- ◆ `a *= b`
- ◆ `a /= b`
- ◆ `a %= b`
- ◆ `a += b`
- ◆ `a -= b`
- ◆ `a <<= b`
- ◆ `a >>= b`
- ◆ `a &= b`
- ◆ `a ^= b`
- ◆ `a |= b`
- ◆ `a += b` equivalent to `a=a+b`
- ◆ allow for simpler codes and better optimizations

## Special operators

---

- ◆ scope operators: `::`
- ◆ member selectors
  - ◆ `.`
  - ◆ `->`
- ◆ subscript `[]`
- ◆ function call `()`
- ◆ construction `()`
- ◆ `typeid`
- ◆ casts
  - ◆ `const_cast`
  - ◆ `dynamic_cast`
  - ◆ `reinterpret_cast`
  - ◆ `static_cast`
- ◆ `sizeof`
- ◆ `new`
- ◆ `delete`
- ◆ `delete[]`
- ◆ pointer to member select
  - ◆ `.*`
  - ◆ `->*`
- ◆ `throw`
- ◆ comma `,`
- ◆ all these will be discussed later

## Operator precedences

---

- ◆ Are listed in detail in all reference books
- ◆ Arithmetic operators follow usual rules
  - ◆ `a+b*c` is the same as `a+(b*c)`
- ◆ Otherwise, *when in doubt use parentheses*

## Statements

---

- ◆ simple statements

- ◆ `;` // null statement
- ◆ `int x;` // declaration statement
- ◆ `typedef int index_type;` // type definition
- ◆ `cout << "Hello world";` // all simple statements end with `;`

- ◆ compound statements

- ◆ more than one statement, enclosed in curly braces

```
{
  int x;
  cin >> x;
  cout << x*x;
}
```

## The if statement

---

- ◆ Has the form

```
if (condition)
  statement
```

- ◆ or

```
if (condition)
  statement
else
  statement
```

- ◆ can be chained

```
if (condition)
  statement
else if(condition)
  statement
else
  statement
```

- ◆ Example:

```
if (light == red)
  cout << "STOP!";
else if (light == orange)
  cout << "Attention";
else {
  cout << "Go!";
  go();
}
```

## The switch statement

---

- ◆ can be used instead of deeply nested if statements:

```
switch (light) {
  case red:
    cout << "STOP!";
    break;
  case orange:
    cout << "Attention";
    break;
  case green:
    cout << "Go!";
    go();
    break;
  default:
    cerr << "illegal color";
    abort();
}
```

- ◆ do not forget the `break`!
- ◆ always include a default!
  - ◆ the telephone system of the US east coast was once disrupted completely for several hours because of a missing default!
- ◆ also multiple labels possible:

```
switch(ch) {
  case 'a':
  case 'e':
  case 'i':
  case 'o':
  case 'u':
    cout << "vowel";
  default:
    cout << "other character";
}
```

## The for loop statement

---

- ◆ has the form
 

```
for ( init-statement ; condition ; expression )
  statement
```

- ◆ example:

```
◆ for (int i=0;i<10;++i)
  cout << i << "\n";
```

- ◆ can contain more than one statement in `for(;;)`:

```
◆ double fac;
  int k;
  for (k=1, fac=1 ; k<50 ; ++k, fac*=I) {
    cout << k << "! = " << fac << "\n";
  }
```

- ◆ attention: variables declared in `for(;;)` are valid only within the for loop! difference to old-style C!

## The while statement

---

- ◆ is a simpler form of a loop:

```
while (condition)
    statement
```

- ◆ example:

```
while (trafficlight()==red) {
    cout << "Still waiting\n";
    sleep(1);
}
```

## The do-while statement

---

- ◆ is similar to the while statement

```
do
    statement
while (condition);
```

- ◆ Example

```
do {
    cout << "Working\n";
    work();
} while (work_to_do());
```

## The break and continue statements

---

- ◆ **break** ends the loop immediately and jumps to the next statement following the loop
- ◆ **continue** starts the next iteration immediately
- ◆ An example:

```
while (true) {
    if (light()==red)
        continue;
    start_engine();
    if(light()==orange)
        continue;
    drive_off();
    break;
}
```

## A loop example: what is wrong?

---

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Enter a number: ";
    unsigned int n;
    cin >> n;

    for (int i=1;i<=n;++i)
        cout << i << "\n";

    int i=0;
    while (i<n)
        cout << ++i << "\n";

    i=1;
    do
        cout << i++ << "\n";
    while (i<=n);

    i=1;
    while (true) {
        if(i>n)
            break;
        cout << i++ << "\n";
    }
}
```

## The goto statement

---

- ◆ will not be discussed as it should not be used
- ◆ included only for machine produced codes, e.g. FORTRAN -> C translators
- ◆ can always be replaced by one of the other control structures
- ◆ **we will not allow any goto in the exercises!**

## Dynamic allocation

---

- ◆ Automatic allocation
  - ◆ `float x[10];` // allocates memory for 10 numbers
- ◆ Dynamic allocation
  - ◆ `unsigned int n; cin >> n; float x[n];` // will not work
  - ◆ The compiler has to know the number!
  - ◆ Solution: dynamic allocation
    - ◆ `float *x=new float[n];` // allocate some memory for an array
    - ◆ `x[0]=...;...` // do some work with the array x
    - ◆ `delete[] x;` // delete the memory for the array. x[], \*x now undefined!
    - ◆ `int *ip=new int;` // get a new integer
    - ◆ `*ip=n;` // assign a value
    - ◆ `delete ip;` // discard the integer; \*ip now undefined!
- ◆ Don't confuse
  - ◆ `delete`, used for simple variables
  - ◆ `delete[]`, used for arrays

## What is a pointer?

---

- ◆ a variable referring to the address of another
- ◆ how is it declared?
  - ◆ `int *ip; // pointer to an integer`
- ◆ how is it initialized?
  - ◆ `int i=0;`
  - ◆ `ip = &i; // & takes the address of a variable`
  - ◆ `float* fp=new float; // allocates a float, stores it address in fp`
- ◆ how is it used?
  - ◆ `*ip = 1; // * dereferences the pointer:`  
    `// *ip is the variable the pointer points to`
  - ◆ `*fp = 3.1415927;`

## Arrays and pointers

---

- ◆ are very similar, but different!
  - ◆ see these examples!
- ```

int array[10];
for (int i=0;i < 10; ++i)
    array[i]=i;

int* p = array;
// same as &array[0]
for (int i=0;i < 10; ++i)
    cout << *p++;

int* pointer=new int[10];
for (int i=0;i < 10; ++i)
    pointer[i]=i;

int* p = pointer;
for (int i=0;i < 10; ++i)
    cout << *p++;

delete[] pointer;
  
```

## Pointer arithmetic

---

- ◆ for any pointer `T *p`; the following holds:
  - ◆ `p[n]` is the same as `*(p+n)`;
- ◆ Adding an integer `n` to a pointer increments it by the `n` times the size of the type:
  - ◆ `reinterpret_cast<unsigned long>(p+1) - reinterpret_cast<unsigned long>(p) == sizeof(T)`
  - ◆ is always true!
- ◆ Increment `++` and decrement `--` increase/decrease by one element
- ◆ Take care: `int *p`; does not allocate memory!
- ◆ Be sure to only use valid pointers
  - ◆ initialize them
  - ◆ do not use them after the object has been deleted!
  - ◆ catastrophic errors otherwise

## Another difficult topic: references

---

- ◆ are aliases for other variables:

```
float very_long_variabe_name_for_number=0;

float &x=very_long_variabe_name_for_number;
// x refers to the same memory location

x=5; // sets very_long_variabe_name_for_number to 5;

float y=2;
x=y; // sets very_long_variabe_name_for_number to 2;
// does not set x to refer to y!
```

## A more flexible program

---

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

```
float square(float x) {
    return x*x;
}
```

- ◆ a function “square” is defined
  - ◆ return value is float
  - ◆ parameter x is float

```
int main() {
    cout << "Enter a number:\n";
    float x;
    cin >> x;
    cout << x << " " <<
        square(x) << "\n";
    return 0;
}
```

- ◆ and used in the program

## Function call syntax

---

- ◆ syntax:
 

```
returntype functionname
    (parameters )
{
    functionbody
}
```

- ◆ *returntype* is “void” if there is no return value:

```
void error(char[] msg) {
    cerr << msg << "\n";
}
```

- ◆ There are several kinds of parameters:

- ◆ pass by value
- ◆ pass by reference
- ◆ pass by const reference
- ◆ pass by pointer

- ◆ Advanced topics to be discussed later:

- ◆ inline functions
- ◆ default arguments
- ◆ function overloading
- ◆ template functions

## Pass by value

---

- ◆ The variable in the function is a copy of the variable in the calling program:

```
void f(int x) {  
    x++; // increments x but not the variable of the calling program  
    cout << x;  
}  
  
int main() {  
    int a=1;  
    f(a);  
    cout << a; // is still 1  
}
```

- ◆ Copying of variables time consuming for large objects like matrices

## Pass by reference

---

- ◆ The function parameter is an alias for the original variable:

```
void increment(int& n) {  
    n++;  
}  
  
int main() {  
    int x=1; increment(x); // x now 2  
    increment(5); // will not work since 5 is literal constant!  
}
```

- ◆ avoids copying of large objects:
  - ◆ `vector eigenvalues(Matrix &A);`
- ◆ but allows unwanted modifications!
  - ◆ the matrix A might be changed by the call to eigenvalues!

## Pass by const reference

---

- ◆ Problem:
  - ◆ `vector eigenvalues(Matrix &A);` // allows modification of A
  - ◆ `vector eigenvalues(Matrix A);` // involves copying of A
- ◆ how do we avoid copying and prohibit modification?
  - ◆ `vector eigenvalues (const Matrix &A);`
  - ◆ now a reference is passed -> no copying
  - ◆ the parameter is const -> cannot be modified

## Pass by pointer

---

- ◆ Another method to pass an object without copying is to pass its address
- ◆ Used mostly in C
- ◆ `vector eigenvalues(Matrix *m);`
- ◆ disadvantages:
  - ◆ The parameter must always be dereferenced: `*m;`
  - ◆ In the function call the address has to be taken:  

```
Matrix A;  
cout << eigenvalues(&A);
```
- ◆ often unnecessary in C++

## A swap example

---

- ◆ Five examples for swapping number

- ◆ `void swap1 (int a, int b) { int t=a; a=b; b=t; }`
- ◆ `void swap2 (int& a, int& b) { int t=a; a=b; b=t;}`
- ◆ `void swap3 (const int & a, const int& b)`  
`{ int t=a; a=b; b=t;}`
- ◆ `void swap4 (int *a, int *b) { int *t=a; a=b; b=t;}`
- ◆ `void swap5 (int* a, int* b) {int t=*a; *a=*b; *b=t;}`

- ◆ Which will compile?

- ◆ What is the effect of:

- ◆ `{ int a=1; b=2; swap1(a,b); cout << a << " " << b << "\n";}`
- ◆ `{ int a=1; b=2; swap2(a,b); cout << a << " " << b << "\n";}`
- ◆ `{ int a=1; b=2; swap3(a,b); cout << a << " " << b << "\n";}`
- ◆ `{ int a=1; b=2; swap4(&a,&b); cout << a << " " << b << "\n";}`
- ◆ `{ int a=1; b=2; swap5(&a,&b); cout << a << " " << b << "\n";}`