An Introduction to C++

Part 1

A review of basic C and C++

Why C++?

- Generic high level programming
  - Shorter development times
  - Smaller error rate
  - Easier debugging
  - Better software reuse
- Efficiency
  - As fast or faster then 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>
#include <iterator>

using namespace std;

int main()
 {
vector<string> data;
copy(istream_iterator<string>(cin),istream_iterator<string>(),back_inserter(data));
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/

Why C++?

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

A first C++ program

```
/* A first program */

#include <iostream>

using namespace std;

int main()
{
  cout << "Hello 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

Getting the source by CVS: ETH D-PHYS machines

- Create a directory for your sources, e.g.

```
mkdir Lecture
cd Lecture
```

- Check out the sources for this week

```
export CVSROOT=/home/troyer/PT/AS08
cvs checkout PT
cd PT/week2
```

- Compile the program

```
g++ -o hello hello.C
```

- Run the program

```
./hello
```

Getting the source by CVS: your own machine with bash

- Create a directory for your sources, e.g.

```
mkdir Lecture
cd Lecture
```

- Check out the sources for this week

```
export CVSROOT=:ext:yourname@paris.ethz.ch:/home/troyer/PT/AS08
export CVS_RSH=ssh
cvs checkout PT
cd PT/week2
```

- Compile the program

```
c++ -o hello hello.C
```

- Run the program

```
./hello
```

Getting the source by CVS: your own machine with tcsh

- Create a directory for your sources, e.g.

```
mkdir Lecture
cd Lecture
```

- Check out the sources for this week

```
setenv CVSROOT :ext:yourname@paris.ethz.ch:/home/troyer/PT/AS08
setenv CVS_RSH ssh
cvs checkout PT
cd PT/week2
```

- Compile the program

```
c++ -o hello hello.C
```

- Run the program

```
./hello
```

More about namespaces

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Hello\n";
}
```

```
#include <iostream>
using std::cout;
int main()
{
  cout << "Hello\n";
}
```

```
#include <iostream>
int main()
{
  std::cout << "Hello\n";
}
```

- All these versions are equivalent
- Feel free to use any style in your program
- Do not use using statements in libraries though

A first calculation

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

using namespace std;

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

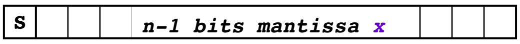
- `<cmath>` is the header for mathematical functions
- Output can be connected by `<<`
- Expressions can be used in output statements
- What are these constants?
  - `5.`
  - `0`
  - `"\n"`

Integral data types

- Signed data types
  - `short, int, long, long long`
  - Not yet standard: `int8_t, int16_t, int32_t, int64_t`
- Unsigned data types
  - `unsigned short`, `unsigned int, unsigned long, unsigned long long`
  - Not yet standard: `uint8_t, uint16_t, uint32_t, uint64_t`
- Are stored as binary numbers
  - `short`: usually 16 bit
  - `int`: usually 32 bit
  - `long`: usually 32 bit on 32-bit CPUs and 64 bit on 64-bit CPUs
  - `long long`: usually 64 bits

Integer representations

- An *n*-bit integer is stored in *n*/8 bytes
  - Little-endian: least significant byte first
  - Big-endian: most significant byte first
  - Exercise: write a program to check the format of your CPU
- Unsigned 

  - `x` just stored as *n* bits, values from 0 … 2^n-1
- Signed 

  - Stored as 2's complement, values from -2^{n-1} … $2^{n-1}-1$
  - Highest bit is sign **S**
  - `x` ≥ 0 : **S**=0, rest is `x`
  - `x` < 0 : **S**=1, rest is `~(-x -1)`
  - Advantage of this format: signed numbers can be added like unsigned

Integer constants

- Integer literals can be entered in a natural way
- Suffixes specify type (if needed)
  - int: `0`, `-3`, ….
  - unsigned int: `3u`, `7U` ,...
  - short: `0S`, `-5s`, ...
  - unsigned short: `1us`, `9su`, `6US`, ...
  - long: `0L`, `-5l`, ...
  - unsigned long: `1ul`, `9Lu`, `6Ul`, ...
  - long long: `0LL`, `-5ll`, ...
  - unsigned long long: `1ull`, `9LLu`, `6Ull`, ...

Characters

- Character types
  - Single byte: `char, unsigned char, signed char`
    - Uses ASCII standard
  - Multi-byte (e.g. for Japanese: 大): `wchar_t`
    - Unfortunately is not required to use Unicode standard
- Character literals
  - `'a', 'b', 'c', '1', '2',` …
  - `'\t'` … tabulator
  - `'\n'` … new line
  - `'\r'` … line feed
  - `'\0'` … byte value 0

Strings

- String type
  - C-style character arrays `char s[100]` should be avoided
  - C++ class `std::string` for single-byte character strings
  - C++ class `std::wstring` for multi-byte character strings
- String literals
  - `"Hello"`
  - Contain a trailing '\0', thus `sizeof("Hello")==6`

Boolean (logical) type

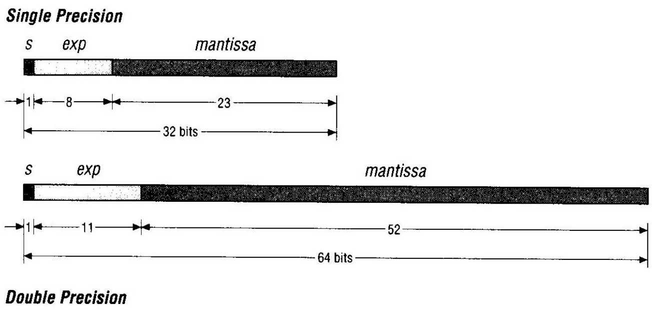
- Type
  - `bool`
- Literal
  - `true`
  - `false`

Floating point numbers

- Floating point types
  - single precision: `float`
    - usually 32 bit
  - double precision: `double`
    - Usually 64 bit
  - extended precision: `long double`
    - Often 64 bit (PowePC), 80 bit (Pentium) or 128 bit (Cray)
- Literals
  - single precision: `4.562f`, `3.0F`
  - double precision: `3.1415927`, `0.`
  - extended precision: `6.54498467494849849489L`

IEEE floating point representation

- The 32 (64) bits are divided into sign, exponent and mantissa



Type	Exponent	Mantissa	Smallest	Largest	Base 10 accuracy
`float`	8	23	1.2E-38	3.4E+38	6-9
`double`	11	52	2.2E-308	1.8E+308	15-17

Converting to/from IEEE representation

- Sign
  - Positive: 0, Negative: 1
- Mantissa
  - Left shifted until leftmost digit is 1, other digits are stored
- Exponent
  - Half of the range (127 for float, 1023 for double) is added

```
172.625                       Base 10
10101100.101 X 2 ** 0         Base 2
1.0101100101 X 2 ** 7         Base 2 Normalized
```

Add 127 for bias=134

```
0 10000110 01011001010000000000000
         1. Assumed bit and binary point
```

Floating point arithmetic

- Truncation can happen because of finite precision

```
  1.00000
  0.0000123
  ---------
  1.00001
```

- Machine precision e is smallest number such that 1+ e ≠1
  - Exercise: calculate e for `float`, `double` and `long double` on your machine
- Be very careful about roundoff
  - For example: sum numbers starting from smallest to largest
  - See examples provided

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 by replacing T above by the desired type:
  `std::numeric_limits<double>::epsilon()`

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;`

Advanced types

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

```
enum trafficlight {red=17, 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)
  - Should not be used in C++ anymore!!!

- 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 or look at http://www.cppreference.com/operator_precedence.html
- 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!";
}
```

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(;;), but this is very bad style!
  - ```
    double fac;
    int k;
    for (k=1,fac=1 ; k<50 ; ++k, fac*=k)
      cout << k << "! = " << fac << "\n";
    ```

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!**