Slides from FYS3150/4150 Programming aspects

Morten Hjorth-Jensen[1,2]

Department of Physics, University of Oslo[1]

Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University[2]

Fall 2015

Extremely useful tools, strongly recommended

and discussed at the lab sessions the first two weeks.

- GIT for version control (see webpage)
- ipython notebook
- QTcreator for editing and mastering computational projects (for C++ codes, see webpage of course)
- Armadillo as a useful numerical library for C++, highly recommended
- Unit tests, see also webpage
- Devilry for handing in projects

A structured programming approach

- Before writing a single line, have the algorithm clarified and understood. It is crucial to have a logical structure of e.g., the flow and organization of data before one starts writing.
- Always try to choose the simplest algorithm. Computational speed can be improved upon later.
- Try to write a as clear program as possible. Such programs are easier to debug, and although it may take more time, in the long run it may save you time. If you collaborate with other people, it reduces spending time on debuging and trying to understand what the codes do. A clear program will also allow you to remember better what the program really does!

A structured programming approach

- The planning of the program should be from top down to bottom, trying to keep the flow as linear as possible. Avoid jumping back and forth in the program. First you need to arrange the major tasks to be achieved. Then try to break the major tasks into subtasks. These can be represented by functions or subprograms. They should accomplish limited tasks and as far as possible be independent of each other. That will allow you to use them in other programs as well.
- Try always to find some cases where an analytical solution exists or where simple test cases can be applied. If possible, devise different algorithms for solving the same problem. If you get the same answers, you may have coded things correctly or made the same error twice or more.

Compiling and linking without QTcreator.

In order to obtain an executable file for a C++ program, the following instructions under Linux/Unix can be used
c++ -c -Wall myprogram.cpp c++ -o myprogram myprogram.o
where the compiler is called through the command c++/g++. The compiler option -Wall means that a warning is issued in case of non-standard language. The executable file is in this case `myprogram`. The option `-c` is for compilation only, where the program is translated into machine code, while the `-o` option links the produced object file `myprogram.o` and produces the executable `myprogram` .
For Fortran2008 we use the Intel compiler, replace c++ with `ifort`. Also, to speed up the code use compile options like
c++ -O3 -c -Wall myprogram.cpp

Makefiles and simple scripts

Under Linux/Unix it is often convenient to create a so-called makefile, which is a script which includes possible compiling commands.

Comment lines General makefile for c - choose PROG = name of given program Here we define compiler option, libraries and the target CC= g++ -Wall PROG= myprogram this is the math library in C, not necessary for C++ LIB = -lm Here we make the executable file *PROG* :PROG.o *CC*PROG.o *LIB* − *o*PROG whereas here we create the object file *PROG*.*o* :PROG.c *CC* − *c*PROG.c If you name your file for `makefile`, simply type the command `make` and Linux/Unix executes all of the statements in the above makefile. Note that C++ files have the extension `.cpp`.

Hello world

The C encounter.

Here we present first the C version.
/* comments in C begin like this and end with */ include ¡stdlib.h¿ /* atof function */ include ¡math.h¿ /* sine function */ include ¡stdio.h¿ /* printf function */ int main (int argc, char* argv[]) double r, s; /* declare variables */ r = atof(argv[1]); /* convert the text argv[1] to double */ s = sin(r); printf("Hello, World! sin(return 0; /* success execution of the program */

Hello World

Dissection I.

The compiler must see a declaration of a function before you can call it (the compiler checks the argument and return types). The declaration of library functions appears in so-called "header files" that must be included in the program, e.g.,
include ¡stdlib.h¿ /* atof function */ We call three functions (atof, sin, printf) and these are declared in three different header files. The main program is a function called main with a return value set to an integer, int (0 if success). The operating system stores the return value, and other programs/utilities can check whether the execution was successful or not. The command-line arguments are transferred to the main function through
int main (int argc, char* argv[])

Dissection II.

The command-line arguments are transferred to the main function through
int main (int argc, char* argv[]) The integer `argc` is the no of command-line arguments, set to one in our case, while `argv` is a vector of strings containing the command-line arguments with `argv[0]` containing the name of the program and `argv[1]`, `argv[2]`, ... are the command-line args, i.e., the number of lines of input to the program. Here we define floating points, see also below, through the keywords `float` for single precision real numbers and `double` for double precision. The function `atof` transforms a text (`argv[1]`) to a float. The sine function is declared in math.h, a library which is not automatically included and needs to be linked when computing an executable file.
With the command `printf` we obtain a formatted printout. The `printf` syntax is used for formatting output in many C-inspired languages (Perl, Python, Awk, partly C++).

Hello World

Now in C++.

Here we present first the C++ version.
// A comment line begins like this in C++ programs // Standard ANSI-C++ include files using namespace std include ¡iostream¿ // input and output int main (int argc, char* argv[]) // convert the text argv[1] to double using atof: double r = atof(argv[1]); double s = sin(r); cout ¡¡ "Hello, World! sin(" ¡¡ r ¡¡ ")=" ¡¡ s ¡¡ "; // success return 0;

C++ Hello World

Dissection I.

We have replaced the call to `printf` with the standard C++ function `cout`. The header file `<iostream.h>` is then needed. In addition, we don't need to declare variables like `r` and `s` at the beginning of the program. I personally prefer however to declare all variables at the beginning of a function, as this gives *me* a feeling of greater readability.

Brief summary

C/C++ program.

- A C/C++ program begins with include statements of header files (libraries,intrinsic functions etc)
- Functions which are used are normally defined at top (details next week)
- The main program is set up as an integer, it returns 0 (everything correct) or 1 (something went wrong)
- Standard `if`, `while` and `for` statements as in Java, Fortran, Python...
- Integers have a very limited range.

Brief summary

Arrays.

- A C/C++ array begins by indexing at 0!
- Array allocations are done by size, not by the final index value.If you allocate an array with 10 elements, you should index them from $0, 1, \ldots, 9$.
- Initialize always an array before a computation.

Serious problems and representation of numbers

Integer and Real Numbers.

- Overflow
- Underflow
- Roundoff errors
- Loss of precision

Limits, you must declare variables

C++ and Fortran declarations.

type in C/C++ and Fortran2008	*bits*	*range*
int/INTEGER (2)	*16*	*-32768 to 32767*
unsigned int	*16*	*0 to 65535*
signed int	*16*	*-32768 to 32767*
short int	*16*	*-32768 to 32767*
unsigned short int	*16*	*0 to 65535*
signed short int	*16*	*−32768 to 32767*
int/long int/INTEGER (4)	*32*	*-2147483648 to 2147483*
signed long int	*32*	*-2147483648 to 2147483*
float/REAL(4)	*32*	3.4×10^{-44} *to* $3.4 \times 10^{+}$
double/REAL(8)	*64*	1.7×10^{-322} *to* $1.7 \times 10^{+}$
long double	*64*	1.7×10^{-322} *to* $1.7 \times 10^{+}$

From decimal to binary representation

How to do it.

$$a_n 2^n + a_{n-1} 2^{n-1} + a_{n-2} 2^{n-2} + \cdots + a_0 2^0.$$

In binary notation we have thus $(417)_{10} = (110110001)_2$ since we have

$$(110100001)_2 = 1\times 2^8 + 1\times 2^7 + 0\times 2^6 + 1\times 2^5 + 0\times 2^4 + 0\times 2^3 + 0\times 2^2 + 0\times 2^2 +$$

To see this, we have performed the following divisions by 2

417/2=208	*remainder 1*	*coefficient of* 2^0 *is 1*
208/2=104	*remainder 0*	*coefficient of* 2^1 *is 0*
104/2=52	*remainder 0*	*coefficient of* 2^2 *is 0*
52/2=26	*remainder 0*	*coefficient of* 2^3 *is 0*
26/2=13	*remainder 1*	*coefficient of* 2^4 *is 0*
13/2= 6	*remainder 1*	*coefficient of* 2^5 *is 1*
6/2= 3	*remainder 0*	*coefficient of* 2^6 *is 0*

From decimal to binary representation

Integer numbers.

using namespace std; include ¡iostream¿ int main (int argc, char* argv[]) int i; int terms[32]; // storage of a0, a1, etc, up to 32 bits int number = atoi(argv[1]); // initialise the term a0, a1 etc for (i=0; i ¡ 32 ; i++) terms[i] = 0; for (i=0; i ¡ 32 ; i++) terms[i] = numbernumber /= 2;
// write out results cout ¡¡ "Number of bytes used= " ¡¡ sizeof(number) ¡¡ endl; for (i=0; i ¡ 32 ; i++) cout ¡¡ " Term nr: " ¡¡ i ¡¡ "Value= " ¡¡ terms[i]; cout ¡¡ endl;
return 0;

From decimal to binary representation

Integer numbers Fortran.

PROGRAM
binary¡*nteger**IMPLICITNONEINTEGERi*, *number*, *terms*(0 :
31)!*storageofa*0, *a*1, *etc*, *upto*32*bits*
WRITE(*,*) 'Give a number to transform to binary notation'
READ(*,*) number ! Initialise the terms a0, a1 etc terms = 0 !
Fortran takes only integer loop variables DO i=0, 31 terms(i) =
MOD(number,2) number = number/2 ENDDO ! write out results
WRITE(*,*) 'Binary representation ' DO i=0, 31 WRITE(*,*)'
Term nr and value', i, terms(i) ENDDO
END PROGRAM binary¡*nteger*

Integer Numbers

Possible Overflow for Integers.

// A comment line begins like this in C++ programs // Program to calculate 2**n // Standard ANSI-C++ include files */ using namespace std include ¡iostream¿ include ¡cmath¿ int main() int int1, int2, int3; // print to screen cout ¡¡ "Read in the exponential N for $2^N =$ "; //*readfromscreencin* >> *int*2; *int*1 = (*int*)*pow*(2., (*double*)*int*2); *cout* << "$2^N * 2^N =$ " << *int*1 ∗ *int*1 << ""; *int*3 = *int*1 − 1; *cout* << "$2^N * (2^N - 1) =$ " << *int*1 ∗ *int*3 << ""; *cout* << "$2^N - 1 =$ " << *int*3 << ""; *return*0; // End: program main()

Loss of Precision

Machine Numbers.

In the decimal system we would write a number like 9.90625 in what is called the normalized scientific notation.

$$9.90625 = 0.990625 \times 10^1,$$

and a real non-zero number could be generalized as

$$x = \pm r \times 10^n, \tag{1}$$

with r a number in the range $1/10 \leq r < 1$. In a similar way we can use represent a binary number in scientific notation as

$$x = \pm q \times 2^m, \tag{2}$$

with q a number in the range $1/2 \leq q < 1$. This means that the mantissa of a binary number would be represented by the general formula

Loss of Precision

Machine Numbers.

In a typical computer, floating-point numbers are represented in the way described above, but with certain restrictions on q and m imposed by the available word length. In the machine, our number x is represented as

$$x = (-1)^s \times \text{mantissa} \times 2^{\text{exponent}}, \tag{4}$$

where s is the sign bit, and the exponent gives the available range. With a single-precision word, 32 bits, 8 bits would typically be reserved for the exponent, 1 bit for the sign and 23 for the mantissa.

Loss of Precision

Machine Numbers.

A modification of the scientific notation for binary numbers is to require that the leading binary digit 1 appears to the left of the binary point. In this case the representation of the mantissa q would be $(1.f)_2$ and $1 \leq q < 2$. This form is rather useful when storing binary numbers in a computer word, since we can always assume that the leading bit 1 is there. One bit of space can then be saved meaning that a 23 bits mantissa has actually 24 bits. This means explicitely that a binary number with 23 bits for the mantissa reads

$$(1.a_{-1}a_{-2}\ldots a_{-23})_2 = 1\times 2^0 + a_{-1}\times 2^{-1} + + a_{-2}\times 2^{-2} + \cdots + a_{-23}\times 2^{-23}. \tag{5}$$

As an example, consider the 32 bits binary number

$$(10111110111101000000000000000000)_2,$$

where the first bit is reserved for the sign, 1 in this case yielding a

Loss of Precision

Machine Numbers.

However, since the exponent has eight bits, this means it has $2^8 - 1 = 255$ possible numbers in the interval $-128 \leq m \leq 127$, our final exponent is $125 - 127 = -2$ resulting in 2^{-2}. Inserting the sign and the mantissa yields the final number in the decimal representation as

$$-2^{-2}\left(1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5}\right) =$$

$$(-0.4765625)_{10}.$$

In this case we have an exact machine representation with 32 bits (actually, we need less than 23 bits for the mantissa).
If our number x can be exactly represented in the machine, we call x a machine number. Unfortunately, most numbers cannot and are thereby only approximated in the machine. When such a number occurs as the result of reading some input data or of a

Loss of Precision

Machine Numbers.

A floating number x, labelled $fl(x)$ will therefore always be represented as

$$fl(x) = x(1 \pm \epsilon_x), \tag{6}$$

with x the exact number and the error $|\epsilon_x| \leq |\epsilon_M|$, where ϵ_M is the precision assigned. A number like $1/10$ has no exact binary representation with single or double precision. Since the mantissa

$$(1.a_{-1}a_{-2}\ldots a_{-n})_2$$

is always truncated at some stage n due to its limited number of bits, there is only a limited number of real binary numbers. The spacing between every real binary number is given by the chosen machine precision. For a 32 bit words this number is approximately $\epsilon_M \sim 10^{-7}$ and for double precision (64 bits) we have $\epsilon_M \sim 10^{-16}$, or in terms of a binary base as 2^{-23} and 2^{-52} for single and double precision, respectively.

Loss of Precision

Machine Numbers.

In the machine a number is represented as

$$fl(x) = x(1+\epsilon) \tag{7}$$

where $|\epsilon| \leq \epsilon_M$ and ϵ is given by the specified precision, 10^{-7} for single and 10^{-16} for double precision, respectively. ϵ_M is the given precision. In case of a subtraction $a = b - c$, we have

$$fl(a) = fl(b) - fl(c) = a(1+\epsilon_a), \tag{8}$$

or

$$fl(a) = b(1+\epsilon_b) - c(1+\epsilon_c), \tag{9}$$

meaning that

$$fl(a)/a = 1 + \epsilon_b \frac{b}{a} - \epsilon_c \frac{c}{a}, \tag{10}$$

and if $b \approx c$ we see that there is a potential for an increased error

Loss of Precision

Machine Numbers.

Define the absolute error as

$$|fl(a) - a|, \tag{11}$$

whereas the relative error is

$$\frac{|fl(a) - a|}{a} \leq \epsilon_a. \tag{12}$$

The above subraction is thus

$$\frac{|fl(a) - a|}{a} = \frac{|fl(b) - fl(c) - (b - c)|}{a}, \tag{13}$$

yielding

$$\frac{|fl(a) - a|}{a} = \frac{|b\epsilon_b - c\epsilon_c|}{a}. \tag{14}$$

The relative error is the quantity of interest in scientific work. Information about the absolute error is normally of little use in the

Loss of numerical precision

Suppose we wish to evaluate the function

$$f(x) = \frac{1 - \cos(x)}{\sin(x)},$$

for small values of x. Five leading digits. If we multiply the denominator and numerator with $1 + \cos(x)$ we obtain the equivalent expression

$$f(x) = \frac{\sin(x)}{1 + \cos(x)}.$$

If we now choose $x = 0.007$ (in radians) our choice of precision results in

$$\sin(0.007) \approx 0.69999 \times 10^{-2},$$

and

Loss of numerical precision

The first expression for $f(x)$ results in

$$f(x) = \frac{1 - 0.99998}{0.69999 \times 10^{-2}} = \frac{0.2 \times 10^{-4}}{0.69999 \times 10^{-2}} = 0.28572 \times 10^{-2},$$

while the second expression results in

$$f(x) = \frac{0.69999 \times 10^{-2}}{1 + 0.99998} = \frac{0.69999 \times 10^{-2}}{1.99998} = 0.35000 \times 10^{-2},$$

which is also the exact result. In the first expression, due to our choice of precision, we have only one relevant digit in the numerator, after the subtraction. This leads to a loss of precision and a wrong result due to a cancellation of two nearly equal numbers. If we had chosen a precision of six leading digits, both expressions yield the same answer.

Loss of numerical precision

If we were to evaluate $x \sim \pi$, then the second expression for $f(x)$ can lead to potential losses of precision due to cancellations of nearly equal numbers.

This simple example demonstrates the loss of numerical precision due to roundoff errors, where the number of leading digits is lost in a subtraction of two near equal numbers. The lesson to be drawn is that we cannot blindly compute a function. We will always need to carefully analyze our algorithm in the search for potential pitfalls. There is no magic recipe however, the only guideline is an understanding of the fact that a machine cannot represent correctly *all* numbers.

Loss of precision can cuae serious problems

Real Numbers.

- **Overflow**: When the positive exponent exceeds the max value, e.g., 308 for `DOUBLE PRECISION` (64 bits). Under such circumstances the program will terminate and some compilers may give you the warning `OVERFLOW`.
- **Underflow**: When the negative exponent becomes smaller than the min value, e.g., -308 for `DOUBLE PRECISION`. Normally, the variable is then set to zero and the program continues. Other compilers (or compiler options) may warn you with the `UNDERFLOW` message and the program terminates.

Loss of precision, real numbers

Roundoff errors. A floating point number like

$$x = 1.2345678911121314 68 = 0.12345678911121314 68 \times 10^1 \tag{15}$$

may be stored in the following way. The exponent is small and is stored in full precision. However, the mantissa is not stored fully. In double precision (64 bits), digits beyond the 15th are lost since the mantissa is normally stored in two words, one which is the most significant one representing 123456 and the least significant one containing 789111213. The digits beyond 3 are lost. Clearly, if we are summing alternating series with large numbers, subtractions between two large numbers may lead to roundoff errors, since not all relevant digits are kept. This leads eventually to the next problem, namely

More on loss of precision

Real Numbers.

- **Loss of precision**: When one has to e.g., multiply two large numbers where one suspects that the outcome may be beyond the bonds imposed by the variable declaration, one could represent the numbers by logarithms, or rewrite the equations to be solved in terms of dimensionless variables. When dealing with problems in e.g., particle physics or nuclear physics where distance is measured in fm (10^{-15} m), it can be quite convenient to redefine the variables for distance in terms of a dimensionless variable of the order of unity. To give an example, suppose you work with single precision and wish to perform the addition $1 + 10^{-8}$. In this case, the information containing in 10^{-8} is simply lost in the addition. Typically, when performing the addition, the computer equates first the exponents of the two numbers to be added. For 10^{-8} this has however catastrophic consequences since in order to obtain an exponent equal to 10^0, bits in the mantissa are shifted to the right. At the end, all bits in the mantissa are zeros.

A problematic case

Three ways of computing e^{-x}.

Brute force:

$$\exp(-x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^n}{n!}$$

Recursion relation for

$$\exp(-x) = \sum_{n=0}^{\infty} s_n = \sum_{n=0}^{\infty} (-1)^n \frac{x^n}{n!}$$

$$s_n = -s_{n-1} \frac{x}{n},$$

$$\exp(x) = \sum_{n=0}^{\infty} s_n$$

$$\exp(-x) = \frac{1}{\exp(x)}$$

Program to compute $\exp(-x)$

Brute Force.

// Program to calculate function exp(-x) // using straightforward summation with differing precision using namespace std include ¡iostream¿ include ¡cmath¿ // type float: 32 bits precision // type double: 64 bits precision define TYPE double define PHASE(a) (1 - 2 * (abs(a) define TRUNCATION 1.0E-10 // function declaration TYPE factorial(int);

Program to compute $\exp(-x)$

Still Brute Force.

int main() int n; TYPE x, term, sum; for(x = 0.0; x ¡ 100.0; x += 10.0) sum = 0.0; //initialization n = 0; term = 1; while(fabs(term) ¿ TRUNCATION) term = PHASE(n) * (TYPE) pow((TYPE) x,(TYPE) n) / factorial(n); sum += term; n++; // end of while() loop

Program to compute $\exp(-x)$

Oh it never ends!

printf("= number of terms = x, exp(-x), sum, n); // end of for() loop
printf(""); // a final line shift on output return 0; // End: function main() // The function factorial() // calculates and returns n! TYPE factorial(int n) int loop; TYPE fac; for(loop = 1, fac = 1.0; loop ¡= n; loop++) fac *= loop;
return fac; // End: function factorial()

Results $\exp(-x)$

What is going on?

x	$\exp(-x)$	*Series*	*Number of terms in seri*
0.0	*0.100000E+01*	*0.100000E+01*	*1*
10.0	*0.453999E-04*	*0.453999E-04*	*44*
20.0	*0.206115E-08*	*0.487460E-08*	*72*
30.0	*0.935762E-13*	*-0.342134E-04*	*100*
40.0	*0.424835E-17*	*-0.221033E+01*	*127*
50.0	*0.192875E-21*	*-0.833851E+05*	*155*
60.0	*0.875651E-26*	*-0.850381E+09*	*171*
70.0	*0.397545E-30*	*NaN*	*171*
80.0	*0.180485E-34*	*NaN*	*171*
90.0	*0.819401E-39*	*NaN*	*171*
100.0	*0.372008E-43*	*NaN*	*171*

Program to compute $\exp(-x)$

```
// program to compute exp(-x) without exponentials using
namespace std include ¡iostream¿ include ¡cmath¿ define
TRUNCATION 1.0E-10
int main()  int loop, n; double x, term, sum; for(loop = 0; loop ¡=
100; loop += 10)  x = (double) loop; // initialization sum = 1.0;
term = 1; n = 1;
```

Program to compute $\exp(-x)$

Last statements.

```
while(fabs(term) ¿ TRUNCATION)  term *= -x/((double) n); sum
+= term; n++;  // end while loop cout ¡¡ "x = " ¡¡ x ¡¡ " exp =
" ¡¡ exp(-x) ¡¡"series = " ¡¡ sum ¡¡ " number of terms =" ¡¡ n ¡¡
"";  // end of for() loop
cout ¡¡ ""; // a final line shift on output
/* End: function main() */
```

Results $\exp(-x)$

More Problems.

x	$\exp(-x)$	*Series*	*Number of te*
0.000000	*0.10000000E+01*	*0.10000000E+01*	*1*
10.000000	*0.45399900E-04*	*0.45399900E-04*	*4*
20.000000	*0.20611536E-08*	*0.56385075E-08*	*7*
30.000000	*0.93576230E-13*	*-0.30668111E-04*	*10*
40.000000	*0.42483543E-17*	*-0.31657319E+01*	*12*
50.000000	*0.19287498E-21*	*0.11072933E+05*	*15*
60.000000	*0.87565108E-26*	*-0.33516811E+09*	*18*
70.000000	*0.39754497E-30*	*-0.32979605E+14*	*20*
80.000000	*0.18048514E-34*	*0.91805682E+17*	*23*
90.000000	*0.81940126E-39*	*-0.50516254E+22*	*26*
100.000000	*0.37200760E-43*	*-0.29137556E+26*	*29*

Most used formula for derivatives

3 point formulae.

First derivative ($f_0 = f(x_0)$, $f_{-h} = f(x_0 - h)$ and $f_{+h} = f(x_0 + h)$

$$\frac{f_h - f_{-h}}{2h} = f_0' + \sum_{j=1}^{\infty} \frac{f_0^{(2j+1)}}{(2j+1)!} h^{2j}.$$

Second derivative

$$\frac{f_h - 2f_0 + f_{-h}}{h^2} = f_0'' + 2\sum_{j=1}^{\infty} \frac{f_0^{(2j+2)}}{(2j+2)!} h^{2j}.$$

Error Analysis

$$\epsilon = log_{10}\left(\left|\frac{f''_{\text{computed}} - f''_{\text{exact}}}{f''_{\text{exact}}}\right|\right),$$

$$\epsilon_{\text{tot}} = \epsilon_{\text{approx}} + \epsilon_{\text{ro}}.$$

For the computed second derivative we have

$$f_0'' = \frac{f_h - 2f_0 + f_{-h}}{h^2} - 2\sum_{j=1}^{\infty} \frac{f_0^{(2j+2)}}{(2j+2)!} h^{2j},$$

and the truncation or approximation error goes like

$$\epsilon_{\text{approx}} \approx \frac{f_0^{(4)}}{12} h^2.$$

Error Analysis

If we were not to worry about loss of precision, we could in principle make h as small as possible. However, due to the computed expression in the above program example

$$f_0'' = \frac{f_h - 2f_0 + f_{-h}}{h^2} = \frac{(f_h - f_0) + (f_{-h} - f_0)}{h^2},$$

we reach fairly quickly a limit for where loss of precision due to the subtraction of two nearly equal numbers becomes crucial.
If $(f_{\pm h} - f_0)$ are very close, we have $(f_{\pm h} - f_0) \approx \epsilon_M$, where $|\epsilon_M| \leq 10^{-7}$ for single and $|\epsilon_M| \leq 10^{-15}$ for double precision, respectively.
We have then

$$|f_0''| = \left| \frac{(f_h - f_0) + (f_{-h} - f_0)}{h^2} \right| \leq \frac{2\epsilon_M}{h^2}.$$

Error Analysis

Our total error becomes

$$|\epsilon_{\text{tot}}| \leq \frac{2\epsilon_M}{h^2} + \frac{f_0^{(4)}}{12} h^2.$$

It is then natural to ask which value of h yields the smallest total error. Taking the derivative of $|\epsilon_{\text{tot}}|$ with respect to h results in

$$h = \left(\frac{24\epsilon_M}{f_0^{(4)}} \right)^{1/4}.$$

With double precision and $x = 10$ we obtain

$$h \approx 10^{-4}.$$

Beyond this value, it is essentially the loss of numerical precision which takes over.

Error Analysis

Due to the subtractive cancellation in the expression for f'' there is a pronounced detoriation in accuracy as h is made smaller and smaller.
It is instructive in this analysis to rewrite the numerator of the computed derivative as

$$(f_h - f_0) + (f_{-h} - f_0) = (e^{x+h} - e^x) + (e^{x-h} - e^x),$$

as

$$(f_h - f_0) + (f_{-h} - f_0) = e^x(e^h + e^{-h} - 2),$$

since it is the difference $(e^h + e^{-h} - 2)$ which causes the loss of precision.

Error Analysis

x	$h = 0.01$	$h = 0.001$	$h = 0.0001$	$h = 0.0000001$	
0.0	*1.000008*	*1.000000*	*1.000000*	*1.010303*	*1*
1.0	*2.718304*	*2.718282*	*2.718282*	*2.753353*	*2*
2.0	*7.389118*	*7.389057*	*7.389056*	*7.283063*	*7*
3.0	*20.085704*	*20.085539*	*20.085537*	*20.250467*	*2*
4.0	*54.598605*	*54.598155*	*54.598151*	*54.711789*	*5*
5.0	*148.414396*	*148.413172*	*148.413161*	*150.635056*	*14*

Error Analysis

The results for $x = 10$ are shown in the Table

h	$e^h + e^{-h}$	$e^h + e^{-h} - 2$
10^{-1}	*2.0100083361116070*	$1.0008336111607230 \times 10^{-2}$
10^{-2}	*2.0001000008333358*	$1.0000083333605581 \times 10^{-4}$
10^{-3}	*2.0000010000000836*	$1.0000000834065048 \times 10^{-6}$
10^{-5}	*2.0000000099999999*	$1.0000000050247593 \times 10^{-8}$
10^{-5}	*2.0000000001000000*	$9.9999897251734637 \times 10^{-11}$
10^{-6}	*2.0000000000010001*	$9.9997787827987850 \times 10^{-13}$
10^{-7}	*2.0000000000000098*	$9.9920072216264089 \times 10^{-15}$
10^{-8}	*2.0000000000000000*	$0.0000000000000000 \times 10^{0}$
10^{-9}	*2.0000000000000000*	$1.1102230246251565 \times 10^{-16}$
10^{-10}	*2.0000000000000000*	$0.0000000000000000 \times 10^{0}$

Week 35

Overview of week 35

- Monday: Repetition from last week
- Numerical differentiation
- C/C++ programming details, pointers, read/write to/from file
- Tuesday: Intro to linear Algebra and presentation of project 1.
- Matrices in C++ and Fortran2008
- Gaussian elimination and discussion of project 1.
- Computer-Lab: start project 1. Reading asssignments and preparation for project 1: sections 2.5 and 3.1 for general C++ and Fortran features. Sections 6.1-6.4 (till page 182) are relevant for project 1.

Technical Matter in C/C++: Pointers

A pointer specifies where a value resides in the computer's memory (like a house number specifies where a particular family resides on a street).
A pointer points to an address not to a data container of any kind!
Simple example declarations:
using namespace std; // note use of namespace int main() // what are the differences? int var; cin ¿¿ var; int *p, q; int *s, *t; int * a new[var]; // dynamic memory allocation delete [] a;

Technical Matter in C/C++: Pointer example I

```
using namespace std; // note use of namespace int main()  int var;
int *p; p = var; var = 421; printf("Address of integer variable var :
printf("Its value:  printf("Value of integer pointer p :  printf("The
value p points at :  printf("Address of the pointer p :  return 0;
```

Dissection: Pointer example I

Discussion.

int main() int var; // Define an integer variable var int *p; // Define a pointer to an integer p = var; // Extract the address of var var = 421; // Change content of var printf("Address of integer variable var : printf("Its value: printf("Value of integer pointer p : // The content of the variable pointed to by p is *p printf("The value p points at : // Address where the pointer is stored in memory printf("Address of the pointer p : return 0;

Pointer example II

int matr[2]; int *p; p = matr[0]; matr[0] = 321; matr[1] = 322; printf("of matrix element matr[1]: printf("of the matrix element matr[1]; printf("of matrix element matr[2]: printf("of the matrix element matr[2]: printf("of the pointer p: printf("value p points to: printf("value that (p+1) points to printf("of pointer p :

Dissection: Pointer example II

int matr[2]; // Define integer array with two elements int *p; // Define pointer to integer p = matr[0]; // Point to the address of the first element in matr matr[0] = 321; // Change the first element matr[1] = 322; // Change the second element printf("of matrix element matr[1]: printf("of the matrix element matr[1]; printf("of matrix element matr[2]: printf("of the matrix element matr[2]: printf("of the pointer p: printf("value p points to: printf("value that (p+1) points to printf("of pointer p :

Output of Pointer example II

Address of the matrix element matr[1]: 0xbfffef70 Value of the matrix element matr[1]; 321 Address of the matrix element matr[2]: 0xbfffef74 Value of the matrix element matr[2]: 322 Value of the pointer: 0xbfffef70 The value pointer points at: 321 The value that (pointer+1) points at: 322 Address of the pointer variable : 0xbfffef6c

File handling; C-way

using namespace std; include ¡iostream¿ int main(int argc, char *argv[]) FILE *in$_f ile, *out_f ile; if (argc <
3)printf (" Theprogramshasthefollowingstructure : "); printf (" writeinthena
fopen(argv[1]," r"); //returnspointertotheinputfileif (in$_f ile ==
NULL)//NULLmeansthatthefileismissingprintf (" Can′ tfindtheinputfileexit(

File handling; C way cont.

$$\text{out}_f ile =$$
$$fopen(argv[2], "w"); //returnsapointertotheoutputfileif(out_f ile ==$$
$$NULL)//can'tfindthefileprintf("Can'tfindtheoutputfileexit(0); fclose(in_f ile$$

File handling, C++-way

include ¡fstream¿

// input and output file as global variable ofstream ofile; ifstream ifile;

File handling, C++-way

int main(int argc, char* argv[]) char *outfilename; //Read in output file, abort if there are too //few command-line arguments if(argc ¡= 1) cout ¡¡ "Bad Usage: " ¡¡ argv[0] ¡¡ " read also output file on same line" ¡¡ endl; exit(1); else outfilename=argv[1]; ofile.open(outfilename); ..... ofile.close(); // close output file

File handling, C++-way

void output(double r$_m$*in*, *double*r$_m$*ax*, *int*max$_s$*tep*, *double* ∗
d)*inti*; *ofile* << "*RESULTS* : " << *endl*; *ofile* << *setiosflags*(*ios* :: *showp*

// end of function output

File handling, C++-way

int main(int argc, char* argv[]) char *infilename; // Read in input file, abort if there are too // few command-line arguments if(argc ¡= 1) cout ¡¡ "Bad Usage: " ¡¡ argv[0] ¡¡ " read also input file on same line" ¡¡ endl; exit(1); else infilename=argv[1]; ifile.open(infilename); .... ifile.close(); // close input file

File handling, C++-way

const char* filename1 = "myfile"; ifstream ifile(filename1); string filename2 = filename1 + ".out" ofstream ofile(filename2); // new output file ofstream ofile(filename2, ios$_b$*ase* :: *app*); //*append*

// Read something from the file:

double a; int b; char c[200]; ifile ¿¿ a ¿¿ b ¿¿ c; // skips white space in between

// Can test on success of reading:

if (!(ifile ¿¿ a ¿¿ b ¿¿ c)) ok = 0;

Call by value and reference

int main(int argc, char argv[]) int a: int *b; a = 10; b = new int[10]; for (i = 0; i ¡ 10; i++) b[i] = i; func(a, b); delete [] b; return 0;

Call by value and reference

Morten: Too complicated LaTeX code for computer code to be decoded....

Call by value and reference

- Lines 1,2: Declaration of two variables `a` and `b`. The compiler reserves two locations in memory. The size of the location depends on the type of variable. Two properties are important for these locations: the address in memory and the content in the location. The value of `a` is `a`. The address of `a` is `&a`. The value of `b` is `*b`. The address of `b` is `&b`.
- Line 3: The value of `a` is now 10.
- Line 4: Memory to store 10 integers is reserved. The address to the first location is stored in b. Address to element number 6 is given by the expression (b + 6).
- Line 5: All 10 elements of `b` are given values: `b[0] = 0`, `b[1] = 1`, ..., `b[9] = 9`
- line 7: here we deallocate the variable `b`.

Call by value and reference

- Line 6: The `main()` function calls the function `func()` and the program counter transfers to the first statement in `func()`. With respect to data the following happens. The content of `a` (= 10) and the content of `b` (a memory address) are copied to a stack (new memory location) associated with the function `func()`
- Line 7: The variable `x` and `y` are local variables in `func()`. They have the values `x = 10`, `y` is the address of the first element in `b` in `main()`.
- Line 8: The local variable `x` stored in the stack memory is changed to 17. Nothing happens with the value `a` in `main()`.

Call by value and reference

- Line 9: The value of y is an address and the symbol `*y` means the position in memory which has this address. The value in this location is now increased by 10. This means that the value of `b[0]` in the main program is equal to 10. Thus func() has modified a value in `main()`.
- Line 10: This statement has the same effect as line 9 except that it modifies the element `b[6]` in `main()` by adding a value of 10 to what was there originally, namely 5.
- Line 11: The program counter returns to `main()`, the next expression after `func(a,b)`. All data on the stack associated with `func()` are destroyed.

Call by value and reference

- The value of a is transferred to `func()` and stored in a new memory location called `x`. Any modification of `x` in `func()` does not affect in any way the value of `a` in `main()`. This is called *transfer of data by value*.
- On the other hand the next argument in `func()` is an address which is transferred to `func()`. This address can be used to modify the corresponding value in `main()`. In the C language it is expressed as a modification of the value which `y` points to, namely the first element of `b`. This is called *transfer of data by reference* and is a method to transfer data back to the calling function, in this case `main()`.

Call by value and reference

C++ allows however the programmer to use solely call by reference (note that call by reference is implemented as pointers). To see the difference between C and C++, consider the following simple examples. In C we would write

int n; n =8; func(n); /* n is a pointer to n */ .... void func(int *i) *i = 10; /* n is changed to 10 */ ....

whereas in C++ we would write

int n; n =8; func(n); // just transfer n itself .... void func(int i) i = 10; // n is changed to 10 .... The reason why we emphasize the difference between call by value and call by reference is that it allows the programmer to avoid pitfalls like unwanted changes of variables. However, many people feel that this reduces the readability of the code.

Call by value and reference, F90/95

In Fortran we can use `INTENT(IN)`, `INTENT(OUT)`, `INTENT(INOUT)` to let the program know which values should or should not be changed.

SUBROUTINE coulomb$_i$*ntegral*(*np*, *lp*, *n*, *l*, *coulomb*)*USEeffective*$_i$*nteraction*$_d$*eclarUSEen* *n*, *l*, *np*, *lpINTEGER* :: *iREAL*(*KIND* = 8), *INTENT*(*INOUT*) :: *coulombREAL*(*KIND* = 8) :: $z_r el, oscl_r, sum_c oulomb$... This hinders unwanted changes and increases readability.

Object orientation

Why object orientation?

- Three main topics: objects, class hierarchies and polymorphism
- The aim here is to be to be able to write a more general code which can easily be tailored to new situations.
- **Polymorphism** is a term used in software development to describe a variety of techniques employed by programmers to create flexible and reusable software components. The term is Greek and it loosely translates to "many forms". Strategy: try to single out the variables needed to describe a given system and those needed to describe a given solver. !split

Object orientation

In programming languages, a polymorphic object is an entity, such as a variable or a procedure, that can hold or operate on values of differing types during the program's execution. Because a polymorphic object can operate on a variety of values and types, it can also be used in a variety of programs, sometimes with little or

Programming classes

In Fortran a vector or matrix start with 1, but it is easy to change a vector so that it starts with zero or even a negative number. If we have a double precision Fortran vector which starts at -10 and ends at 10, we could declare it as `REAL(KIND=8) :: vector(-10:10)`. Similarly, if we want to start at zero and end at 10 we could write `REAL(KIND=8) :: vector(0:10)`. We have also seen that Fortran allows us to write a matrix addition $\mathbf{A} = \mathbf{B} + \mathbf{C}$ as `A = B + C`. This means that we have overloaded the addition operator so that it translates this operation into two loops and an addition of two matrix elements $a_{ij} = b_{ij} + c_{ij}$.

The way the matrix addition is written is very close to the way we express this relation mathematically. The benefit for the programmer is that our code is easier to read. Furthermore, such a way of coding makes it more likely to spot eventual errors as well.

In Ansi C and C++ arrays start by default from $i = 0$. Moreover, if we wish to add two matrices we need to explicitely write out the two loops as

```
for(i=0 ; i ¡ n ; i++)  for(j=0 ; j ¡ n ; j++)  a[i][j]=b[i][j]+c[i][j]
```

Programming classes

However, the strength of C++ is the possibility to define new data types, tailored to some particular problem. Via new data types and overloading of operations such as addition and subtraction, we can easily define sets of operations and data types which allow us to write a matrix addition in exactly the same way as we would do in Fortran. We could also change the way we declare a C++ matrix elements a_{ij}, from $a[i][j]$ to say $a(i,j)$, as we would do in Fortran. Similarly, we could also change the default range from $0 : n-1$ to $1 : n$.

To achieve this we need to introduce two important entities in C++ programming, classes and templates.

Programming classes

The function and class declarations are fundamental concepts within C++. Functions are abstractions which encapsulate an algorithm or parts of it and perform specific tasks in a program. We have already met several examples on how to use functions. Classes can be defined as abstractions which encapsulate data and operations on these data. The data can be very complex data structures and the class can contain particular functions which operate on these data. Classes allow therefore for a higher level of abstraction in computing. The elements (or components) of the data type are the class data members, and the procedures are the class member functions.

Programming classes

Classes are user-defined tools used to create multi-purpose software which can be reused by other classes or functions. These user-defined data types contain data (variables) and functions operating on the data.

A simple example is that of a point in two dimensions. The data could be the x and y coordinates of a given point. The functions we define could be simple read and write functions or the possibility to compute the distance between two points.

Programming classes

C++ has a class complex in its standard template library (STL). The standard usage in a given function could then look like

// Program to calculate addition and multiplication of two complex numbers using namespace std; include ¡iostream¿ include ¡cmath¿ include ¡complex¿ int main() complex¡double¿ x(6.1,8.2), y(0.5,1.3); // write out x+y cout ¡¡ x + y ¡¡ x*y ¡¡ endl; return 0;

where we add and multiply two complex numbers $x = 6.1 + \imath 8.2$ and $y = 0.5 + \imath 1.3$ with the obvious results $z = x + y = 6.6 + \imath 9.5$ and $z = x \cdot y = -7.61 + \imath 12.03$.

Programming classes

We proceed by splitting our task in three files.

We define first a header file complex.h which contains the declarations of the class. The header file contains the class declaration (data and functions), declaration of stand-alone functions, and all inlined functions, starting as follows

ifndef
$Complex_H defineComplex_H //variousincludestatementsanddefinitionsinclud$
$iostream > //StandardANSI - C++includefilesinclude < new >$
$include....$

class Complex ... definition of variables and their character ; //
declarations of various functions used by the class ... endif

Programming classes

Next we provide a file complex.cpp where the code and algorithms of different functions (except inlined functions) declared within the class are written. The files `complex.h` and `complex.cpp` are normally placed in a directory with other classes and libraries we have defined.

Finally, we discuss here an example of a main program which uses this particular class. An example of a program which uses our complex class is given below. In particular we would like our class to perform tasks like declaring complex variables, writing out the real and imaginary part and performing algebraic operations such as adding or multiplying two complex numbers.

Programming classes

include "Complex.h" ... other include and declarations int main () Complex a(0.1,1.3); // we declare a complex variable a Complex b(3.0), c(5.0,-2.3); // we declare complex variables b and c Complex d = b; // we declare a new complex variable d cout ¡¡ "d=" ¡¡ d ¡¡ ", a=" ¡¡ a ¡¡ ", b=" ¡¡ b ¡¡ endl; d = a*c + b/a; // we add, multiply and divide two complex numbers cout ¡¡ "Re(d)=" ¡¡ d.Re() ¡¡ ", Im(d)=" ¡¡ d.Im() ¡¡ endl; // write out of the real and imaginary parts

Programming classes

We include the header file complex.h and define four different complex variables. These are $a = 0.1 + i1.3$, $b = 3.0 + i0$ (note that if you don't define a value for the imaginary part this is set to zero), $c = 5.0 - i2.3$ and $d = b$. Thereafter we have defined standard algebraic operations and the member functions of the class which allows us to print out the real and imaginary part of a given variable.

Programming classes

class Complex private: double re, im; // real and imaginary part public: Complex (); // Complex c; Complex (double re, double im = 0.0); // Definition of a complex variable; Complex (const Complex c); // Usage: Complex c(a); // equate two complex variables Complex operator= (const Complex c); // c = a; // equate two complex variables, same as previous ....

Programming classes

Complex () // destructor double Re () const; // double
real$_p$art = a.Re(); doubleIm()const; //doubleimag$_p$art =
a.Im(); doubleabs()const; //doublem =
a.abs(); //modulusfriendComplexoperator +
(constComplexa, constComplexb); friendComplexoperator −
(constComplexa, constComplexb); friendComplexoperator ∗
(constComplexa, constComplexb); friendComplexoperator/(constComplex

Programming classes

The class is defined via the statement `class Complex`. We must first use the key word `class`, which in turn is followed by the user-defined variable name `Complex`. The body of the class, data and functions, is encapsulated within the parentheses `{...}`.

Programming classes

Data and specific functions can be private, which means that they cannot be accessed from outside the class. This means also that access cannot be inherited by other functions outside the class. If we use `protected` instead of `private`, then data and functions can be inherited outside the class.

Programming classes

The key word `public` means that data and functions can be accessed from outside the class. Here we have defined several functions which can be accessed by functions outside the class. The declaration `friend` means that stand-alone functions can work on privately declared variables of the type `(re, im)`. Data members of a class should be declared as private variables.

Programming classes

The first public function we encounter is a so-called constructor, which tells how we declare a variable of type `Complex` and how this variable is initialized. We have chose three possibilities in the example above:

A declaration like `Complex c;` calls the member function `Complex()` which can have the following implementation

Complex:: Complex () re = im = 0.0;

meaning that it sets the real and imaginary parts to zero. Note the way a member function is defined. The constructor is the first function that is called when an object is instantiated.

Programming classes

Another possibility is

Complex:: Complex () which means that there is no initialization of the real and imaginary parts. The drawback is that a given compiler can then assign random values to a given variable.

A call like `Complex a(0.1,1.3);` means that we could call the member function 'Complex(double, double)'as

Complex:: Complex (double re$_a$, *doubleim*$_a$)$re = re_a; im = im_a;$

Programming classes

The simplest member function are those we defined to extract the real and imaginary part of a variable. Here you have to recall that these are private data, that is they invisible for users of the class. We obtain a copy of these variables by defining the functions

double Complex:: Re () const return re; // getting the real part
double Complex:: Im () const return im; // and the imaginary part Note that we have introduced the declaration `const`. What does it mean? This declaration means that a variabale cannot be changed within a called function.

Programming classes

If we define a variable as `const double p = 3;` and then try to change its value, we will get an error when we compile our program. This means that constant arguments in functions cannot be changed.

// const arguments (in functions) cannot be changed: void myfunc (const Complex c) c.re = 0.2; /* ILLEGAL!! compiler error... */
If we declare the function and try to change the value to 0.2, the compiler will complain by sending an error message.

Programming classes

If we define a function to compute the absolute value of complex variable like

double Complex:: abs () return sqrt(re*re + im*im); without the constant declaration and define thereafter a function `myabs` as

double myabs (const Complex c) return c.abs(); // Not ok because c.abs() is not a const func. the compiler would not allow the c.abs() call in myabs since `Complex::abs` is not a constant member function.

Programming classes

Constant functions cannot change the object's state. To avoid this we declare the function `abs` as

```
double Complex:: abs () const  return sqrt(re*re + im*im);
```

Programming classes

C++ (and Fortran) allow for overloading of operators. That means we can define algebraic operations on for example vectors or any arbitrary object. As an example, a vector addition of the type $\mathbf{c} = \mathbf{a} + \mathbf{b}$ means that we need to write a small part of code with a for-loop over the dimension of the array. We would rather like to write this statement as `c = a+b;` as this makes the code much more readable and close to eventual equations we want to code. To achieve this we need to extend the definition of operators.

Programming classes

Let us study the declarations in our complex class. In our main function we have a statement like `d = b;`, which means that we call `d.operator= (b)` and we have defined a so-called assignment operator as a part of the class defined as

Complex Complex:: operator= (const Complex c) re = c.re; im = c.im; return *this;

With this function, statements like `Complex d = b;` or `Complex d(b);` make a new object *d*, which becomes a copy of *b*. We can make simple implementations in terms of the assignment

Complex:: Complex (const Complex c) *this = c; which is a pointer to "this object", `*this` is the present object, so `*this = c;` means setting the present object equal to *c*, that is `this->operator= (c);`.

The meaning of the addition operator + for Complex objects is defined in the function

```
Complex operator+ (const Complex& a, const Complex& b); //
```

The compiler translates `c = a + b;` into `c = operator+ (a, b);`. Since this implies the call to function, it brings in an additional overhead. If speed is crucial and this function call is performed inside a loop, then it is more difficult for a given compiler to perform optimizations of a loop.

The solution to this is to inline functions. We discussed inlining in chapter 2 of the lecture notes. Inlining means that the function body is copied directly into the calling code, thus avoiding calling the function. Inlining is enabled by the inline keyword

inline Complex operator+ (const Complex a, const Complex b) return Complex (a.re + b.re, a.im + b.im); Inline functions, with complete bodies must be written in the header file complex.h.

Programming classes

Consider the case `c = a + b;` that is, `c.operator= (operator+ (a,b));` If `operator+`, `operator=` and the constructor `Complex(r,i)` all are inline functions, this transforms to

c.re = a.re + b.re; c.im = a.im + b.im; by the compiler, i.e., no function calls

Programming classes

The stand-alone function `operator+` is a friend of the Complex class

class Complex ... friend Complex operator+ (const Complex a, const Complex b); ... ; so it can read (and manipulate) the private data parts *re* and *im* via

inline Complex operator+ (const Complex a, const Complex b) return Complex (a.re + b.re, a.im + b.im);

Programming classes

Since we do not need to alter the re and im variables, we can get the values by Re() and Im(), and there is no need to be a friend function

inline Complex operator+ (const Complex a, const Complex b)
return Complex (a.Re() + b.Re(), a.Im() + b.Im());

Programming classes

The multiplication functionality can now be extended to imaginary numbers by the following code

inline Complex operator* (const Complex a, const Complex b)
return Complex(a.re*b.re - a.im*b.im, a.im*b.re + a.re*b.im);

It will be convenient to inline all functions used by this operator.

Programming classes

To inline the complete expression `a*b;`, the constructors and `operator=` must also be inlined. This can be achieved via the following piece of code

inline Complex:: Complex () re = im = 0.0; inline Complex:: Complex (double re, *doubleim*)...*inlineComplex* :: *Complex*(*constComplexc*)...*inlineComplex* :: *operator* = (*constComplexc*)...

Programming classes

// e, c, d are complex e = c*d; // first compiler translation: e.operator= (operator* (c,d)); // result of nested inline functions // operator=, operator*, Complex(double,double=0): e.re = c.re*d.re - c.im*d.im; e.im = c.im*d.re + c.re*d.im; The definitions `operator-` and `operator/` follow the same set up.

Programming classes

Finally, if we wish to write to file or another device a complex number using the simple syntax `cout << c;`, we obtain this by defining the effect of $<<$ for a Complex object as

ostream operator¡¡ (ostream o, const Complex c) { o ¡¡ "(" ¡¡ c.Re() ¡¡ "," ¡¡ c.Im() ¡¡ ")"; return o; }

Programming classes, templates

What if we wanted to make a class which takes integers or floating point numbers with single precision? A simple way to achieve this is copy and paste our class and replace `double` with for example `int`.

C++ allows us to do this automatically via the usage of templates, which are the C++ constructs for parameterizing parts of classes. Class templates is a template for producing classes. The declaration consists of the keyword `template` followed by a list of template arguments enclosed in brackets.

Programming classes

We can therefore make a more general class by rewriting our original example as

template¡class T¿ class Complex private: T re, im; // real and imaginary part public: Complex (); // Complex c; Complex (T re, T im = 0); // Definition of a complex variable; Complex (const Complex c); // Usage: Complex c(a); // equate two complex variables Complex operator= (const Complex c); // c = a; // equate two complex variables, same as previous

Programming classes

We can therefore make a more general class by rewriting our original example as

Complex () // destructor T Re () const; // T real$_p$*art* = *a*.*Re*(); *TIm*()*const*; //*Timag*$_p$*art* = *a*.*Im*(); *Tabs*()*const*; //*Tm* = *a*.*abs*(); //*modulusfriendComplexoperator* + (*constComplexa*, *constComplexb*); *friendComplexoperator* − (*constComplexa*, *constComplexb*); *friendComplexoperator* ∗ (*constComplexa*, *constComplexb*); *friendComplexoperator*/(*constComplex*

Programming classes

What it says is that `Complex` is a parameterized type with T as a parameter and T has to be a type such as double or float. The class complex is now a class template and we would define variables in a code as

Complex¡double¿ a(10.0,5.1); Complex¡int¿ b(1,0);

Programming classes

Member functions of our class are defined by preceding the name of the function with the `template` keyword. Consider the function we defined as `Complex:: Complex (double re_a, double im_a)`. We would rewrite this function as

template¡class T¿ Complex¡T¿:: Complex (T re$_a$, Tim_a)$re = re_a$; $im = im_a$; The member functions are otherwise defined following ordinary member function definitions.

Programming classes

Here follows a very simple first class in the file squared.h

// Not all declarations here // Class to compute the square of a number template¡class T¿ class Squared public: // Default constructor, not used here Squared()

// Overload the function operator() T operator()(T x)return x*x;

;

Programming classes

and we would use it as

include ¡iostream¿ include "squared.h" using namespace std;

int main() Squared¡double¿ s; cout ¡¡ s(3) ¡¡ endl;

Unit Testing

Unit Testing is the practice of testing the smallest testable parts, called units, of an application individually and independently to determine if they behave exactly as expected. Unit tests (short code fragments) are usually written such that they can be preformed at any time during the development to continually verify the behavior of the code. In this way, possible bugs will be identified early in the development cycle, making the debugging at later stage much easier. There are many benefits associated with Unit Testing, such as

- It increases confidence in changing and maintaining code. Big changes can be made to the code quickly, since the tests will ensure that everything still is working properly.
- Since the code needs to be modular to make Unit Testing possible, the code will be easier to reuse. This improves the code design.
- Debugging is easier, since when a test fails, only the latest changes need to be debugged.
    - Different parts of a project can be tested without the need to

Simple example of unit test

Look up the guide on how to install unit tests for c++ at course webpage. This is the version with classes. include ¡unittest++/UnitTest++.h¿
class MyMultiplyClass public: double multiply(double x, double y)
return x * y; ;
TEST(MyMath) MyMultiplyClass my;
CHECK$_E$*QUAL*(56, *my*.*multiply*(7, 8));
int main() return UnitTest::RunAllTests();

Simple example of unit test

And without classes include ¡unittest++/UnitTest++.h¿ double multiply(double x, double y) return x * y; TEST(MyMath) CHECK$_E$*QUAL*(56, *multiply*(7, 8)); int main() return UnitTest::RunAllTests(); For Fortran users, the link at http://sourceforge.net/projects/fortranxunit/ contains a similar software for unit testing.

The report: how to write a good scienfitic/technical report

What should it contain? A typical structure.

- An introduction where you explain the aims and rationale for the physics case and what you have done. At the end of the introduction you should give a brief summary of the structure of the report
- Theoretical models and technicalities. This is the methods section.
- Results and discussion

The report

What should I focus on? Methods sections.

- Describe the methods and algorithms
- You need to explain how you implemented the methods and also say something about the structure of your algorithm and present some parts of your code
- You should plug in some calculations to demonstrate your code, such as selected runs used to validate and verify your results. The latter is extremely important!! A reader needs to understand that your code reproduces selected benchmarks and reproduces previous results, either numerical and/or well-known closed form expressions.

The report

What should I focus on? Results.

- Present your results
- Give a critical discussion of your work and place it in the correct context.
- Relate your work to other calculations/studies
- An eventual reader should be able to reproduce your calculations if she/he wants to do so. All input variables should be properly explained.
- Make sure that figures and tables should contain enough information in their captions, axis labels etc so that an eventual reader can gain a first impression of your work by studying figures and tables only.

The report

What should I focus on? Conclusions.

- State your main findings and interpretations
- Try as far as possible to present perspectives for future work
- Try to discuss the pros and cons of the methods and possible improvements

The report

What should I focus on? additional material.

- Additional calculations used to validate the codes
- Selected calculations, these can be listed with few comments
- Listing of the code if you feel this is necessary

You can consider moving parts of the material from the methods section to the appendix. You can also place additional material on your webpage.

The report

What should I focus on? References.

- Give always references to material you base your work on, either scientific articles/reports or books.
- Refer to articles as: name(s) of author(s), journal, volume (boldfaced), page and year in parenthesis.
- Refer to books as: name(s) of author(s), title of book, publisher, place and year, eventual page numbers