Introduction to C++ (and C) Programming

Hans Petter Langtangen[1,2]

Simula Research Laboratory[1]

Dept. of Informatics, Univ. of Oslo[2]

January 2006

Outline

Contents

Required background

- Programming experience with either Java or Fortran/Matlab
- Interest in numerical computing with C++
- Interest in low-level details of the computer
- Knowledge of some C is advantageous (but not required)

About learning C++

- C++ is a complicated computer language
- It takes time to master C++ – one year is the rule of thumb
- Four days can only give a taste of C++
- You need to work intensively with C++ in your own projects to master the language
- C++ exposes you to lots of “low-level details” – these are hidden in languages like Java, Matlab and Python
- Hopefully, you will appreciate the speed and flexibility of C++

Teaching philosophy

Intensive course:

- Lectures 9-12
- Hands-on training 13-16
- Learn from dissecting examples
- Get in touch with the dirty work
- Get some overview of advanced topics
- Focus on principles and generic strategies
- Continued learning on individual basis

This course just gets you started - use textbooks, reference manuals and software examples from the Internet for futher work with projects

Recommended attitude

- Dive into executable examples
- Don't try to understand everything
- Try to adapt examples to new problems
- Look up technical details in manuals/textbooks
- Learn on demand
- Stay cool

Why do you need to learn “old” compiled languages?

- Because C, C++, and Fortran (77/95) are the most efficient existing tools for intensive numerical computing
- Because tons of fast and well-tested codes are available in Fortran, C/C++
- Newer languages have emphasized simplicity and reliability – at the cost of computational efficiency
- To get speed, you need to dive into the details of compiled languages, and this course is a first, gentle step

C

- C is a dominating language in Unix and Windows environments
- The C syntax has inspired lots of popular languages (Awk, C++, Java, Perl, Python, Ruby)
- Numerous tools (numerical libraries, e.g., MPI) are written in C; interfacing them requires C knowledge
- C is extremely portable; "all" machines can compile and run C programs
- C is very low level and close to the machine
- Unlimited possibilities; one can do anything in C
- Programmers of high-level languages often get confused by strange/unexpected errors in C

C++

C++ extends C with

- nicer syntax:
  - declare variables wherever you want
  - in/out function arguments use references (instead of pointers)
- classes for implementing user-defined data types
- a standard library (STL) for frequently used data types (list, stack, queue, vector, hash, string, complex, ...)
- object-oriented programming
- generic programming, i.e., parameterization of variable types via *templates*
- exceptions for error handling

C is a subset of C++

C versus other languages

- Fortran 77 is more primitive but more reliable
- Matlab is as simple/primitive as Fortran 77, but with many more high-level commands (= easy to use)
- C++ is a superset of C and much richer/higher-level/reliable
- Java is simpler and more reliable than C++
- Python is even more high-level, but potentially slow
- Fortran 90/95 is simpler than Java/C++ and a good alternative to C

Speed of C versus speed of other languages

- C is regarded as very fast
- Fortran 77 may yield slightly faster code
- C++ and Fortran 90/95 are in general slower, but C++ is very close to C in speed
- Java is normally considerably slower

Some guidelines

- C programmers need to be concerned with low-level details that C++ (and Java or Fortran) programmers can omit
- Don't use C unless you have to - use C++ instead
- The best solution is often to combine languages: Python to administer user interfaces, I/O and computations, with intensive numerics implemented in C++ or Fortran

High vs low level programs

- Goal: make a window on the screen with the text “Hello World”
- Implementations in
  1. C and the X11 library
  2. C++ and the Qt library
  3. Python

C/X11 implementation (1)

```
#include <stdio.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>


#define STRING   "Hello, world"
#define BORDER   1
#define FONT     "fixed"

XWMHints        xwmh = {
    (InputHint|StateHint),      /* flags */
    False,                      /* input */
    NormalState,                /* initial_state */
    0,                          /* icon pixmap */
    0,                          /* icon window */
    0, 0,                       /* icon location */
    0,                          /* icon mask */
    0,                          /* Window group */
};
```

C/X11 implementation (2)

```
main(argc,argv)
    int argc;
    char **argv;
{
    Display    *dpy;            /* X server connection */
    Window      win;            /* Window ID */
    GC          gc;             /* GC to draw with */
    XFontStruct *fontstruct;    /* Font descriptor */

    unsigned long fth, pad;     /* Font size parameters */
    unsigned long fg, bg, bd;   /* Pixel values */
    unsigned long bw;           /* Border width */
    XGCValues   gcv;            /* Struct for creating GC */
    XEvent      event;          /* Event received */
    XSizeHints  xsh;            /* Size hints for window manager */
    char       *geomSpec;       /* Window geometry string */
    XSetWindowAttributes xswa;  /* Temp. Set Window Attr. struct */

    if ((dpy = XOpenDisplay(NULL)) == NULL) {
        fprintf(stderr, "%s: can't open %s\en", argv[0],
                XDisplayName(NULL));
        exit(1);
    }
```

C/X11 implementation (3)

```
if ((fontstruct = XLoadQueryFont(dpy, FONT)) == NULL) {
    fprintf(stderr, "%s: display %s doesn't know font %s\en",
            argv[0], DisplayString(dpy), FONT);
    exit(1);
}
fth = fontstruct->max_bounds.ascent +
      fontstruct->max_bounds.descent;

bd = WhitePixel(dpy, DefaultScreen(dpy));
bg = BlackPixel(dpy, DefaultScreen(dpy));
fg = WhitePixel(dpy, DefaultScreen(dpy));

pad = BORDER;
bw = 1;

xsh.flags = (PPosition | PSize);
xsh.height = fth + pad * 2;
xsh.width = XTextWidth(fontstruct, STRING,
                       strlen(STRING)) + pad * 2;
xsh.x = (DisplayWidth(dpy,DefaultScreen(dpy))-xsh.width)/2;
xsh.y = (DisplayHeight(dpy,DefaultScreen(dpy))-xsh.height)/2;

win = XCreateSimpleWindow(dpy, DefaultRootWindow(dpy),
                          xsh.x, xsh.y, xsh.width, xsh.height,
                          bw, bd, bg);
```

C/X11 implementation (4)

```
XSetStandardProperties(dpy, win, STRING, STRING, None,
                       argv, argc, &xsh);
XSetWMHints(dpy, win, &xwmh);

xswa.colormap = DefaultColormap(dpy, DefaultScreen(dpy));
xswa.bit_gravity = CenterGravity;
XChangeWindowAttributes(dpy, win,
        (CWColormap | CWBitGravity), &xswa);

gcv.font = fontstruct->fid;
gcv.foreground = fg;
gcv.background = bg;
gc = XCreateGC(dpy, win,
        (GCFont | GCForeground | GCBackground), &gcv);
XSelectInput(dpy, win, ExposureMask);

XMapWindow(dpy, win);
```

C/X11 implementation (5)

```
    /*
     * Loop forever,  examining each event.
     */
    while (1) {
        XNextEvent(dpy, &event);
        if (event.type == Expose && event.xexpose.count == 0) {
            XWindowAttributes xwa;
            int            x, y;
            while (XCheckTypedEvent(dpy, Expose, &event));
            if (XGetWindowAttributes(dpy, win, &xwa) == 0)
                break;
            x = (xwa.width - XTextWidth(fontstruct, STRING,
                                        strlen(STRING))) / 2;
            y = (xwa.height + fontstruct->max_bounds.ascent
                 - fontstruct->max_bounds.descent) / 2;
            XClearWindow(dpy, win);
            XDrawString(dpy, win, gc, x, y, STRING, strlen(STRING));
        }
    }
    exit(1);
}
```

C++/Qt implementation

```
#include <qapplication.h>
#include <qlabel.h>

int main(int argc, char* argv[])
{
  QApplication a(argc, argv);
  QLabel hello("Hello world!", 0);
  hello.resize(100, 30);
  a.setMainWidget(&hello);
  hello.show();
  return a.exec();
}
```

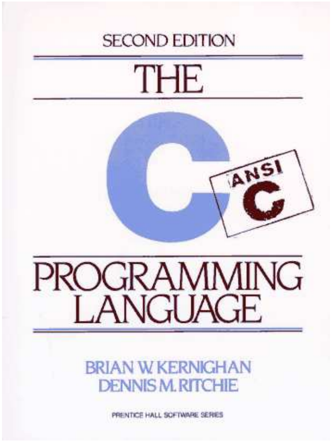
Point: C++ offers abstractions, i.e., complicated variables that hide lots of low-level details. Something similar is offered by Java.

Python implementation

```
#!/usr/bin/env python
from Tkinter import *
root = Tk()
Label(root, text='Hello, World!',
      foreground="white", background="black").pack()
root.mainloop()
```

Similar solutions are offered by Perl, Ruby, Scheme, Tcl

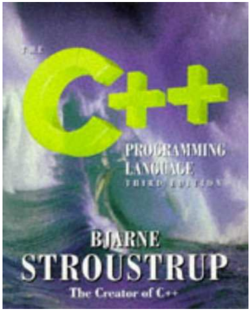
THE textbook on C

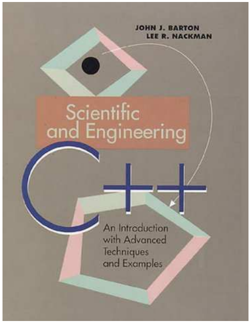


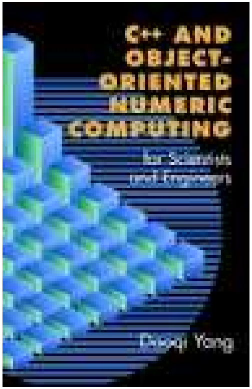
Kernighan and Ritchie: The C Programming Language

Recommended C++ textbooks

Stroustrup, Barton & Nackman, or Yang:




JOHN J. BARTON
LEE R. NACKMAN
Scientific and Engineering
C++
An Introduction with Advanced Techniques and Examples


C++ AND OBJECT-ORIENTED NUMERIC COMPUTING
for Scientists and Engineers
Daoqi Yang

More books reviewed:
http:://www.accu.org/
http://www.comeaucomputing.com/booklist/

The first C++ encounter

Learning by doing:

- Scientific Hello World: the first glimpse of C++
- Data filter: reading from and writing to files, calling functions
- Matrix-vector product: arrays, dynamic memory management, for-loops, subprograms

We mainly teach C++ – the C version specialities are discussed at the end of each example (in this way you learn quite some C with little extra effort)

Scientific Hello World in C++

- Usage:

```
./hw1.app 2.3
```

- Output of program `hw1.app`:

```
Hello, World! sin(2.3)=0.745705
```

- What to learn:
  1. store the first command-line argument in a floating-point variable
  2. call the sine function
  3. write a combination of text and numbers to standard output

The code

```
#include <iostream> // input/output functionality
#include <math.h>   // the sine function
#include <stdlib.h> // the atof function

int main (int argc, char* argv[])
{
  // convert the text argv[1] to double using atof:
  double r = atof(argv[1]);
  // declare variables wherever needed:
  double s = sin(r);
  std::cout << "Hello, World! sin(" << r << ")=" << s << '\n';
  return 0;  /* success */
}
```

File: src/C++/hw/hw1.cpp (C++ files have extension .cpp, .C or .cxx)

Dissection (1)

- 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 “header files” that must be included in the program:

  ```
  #include <math.h> // the sine function
  ```

- We use three functions (`atof`, `sin`, and `std::cout <<`; these are declared in three different header files
- Comments appear after `//` *on a line* or between `/*` and `*/` (anywhere)
- On some systems, including stdlib.h is not required because iostream includes stdlib.h
- Finding the right header files (.h) is always a challenge

Dissection (2)

- The main program is a function called `main`
- The command-line arguments are transferred to the main function:

```
int main (int argc, char* argv[])
```

- `argc` is the no of command-line arguments + 1
- `argv` is a vector of strings containing the command-line arguments
- `argv[1]`, `argv[2]`, ... are the command-line args
- `argv[0]` is the name of the program

Dissection (3)

- Floating-point variables in C and C++:
  1. float: single precision
  2. double: double precision
- atof: transform a text (argv[1]) to float
- Automatic type conversion: double = float
- The sine function is declared in math.h (note: math.h is not automatically included)
- Formatted output is possible, but easier with `printf`
- The return value from `main` is an `int` (0 if success)
- The operating system stores the return value, and other programs/utilities can check whether the execution was successful or not

An interactive version

- Let us ask the user for the real number instead of reading it from the command line

```
std::cout << "Give a real number:";
double r;
std::cin >> r;  // read from keyboard into r
double s = sin(r);
// etc.
```

Scientific Hello World in C

```
#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 in the beginning */
  r = atof(argv[1]);   /* convert the text argv[1] to double */
  s = sin(r);
  printf("Hello, World! sin(%g)=%g\n", r, s);
  return 0;            /* success execution of the program */
}
```

File: src/C/hw/hw1.c (C files have extension .c)

Differences from the C++ version

- C uses stdio.h for I/O and functions like `printf` for output; C++ can use the same, but the official tools are in iostream (and use constructions like `std::cout << r`)
- Variables can be declared anywhere in C++ code; in C they must be listed in the beginning of the function

How to compile and link C++ programs

- One step (compiling and linking):

  ```
  unix> g++ -Wall -O3 -o hw1.app hw1.cpp -lm
  ```

  `-lm` can be skipped when using g++
  (but is otherwise normally required)

- Two steps:

  ```
  unix> g++ -Wall -O3 -c hw1.cpp    # compile, result: hw1.o
  unix> g++ -o hw1.app hw1.o -lm    # link
  ```

- Native C++ compiler on other systems:

  ```
  IBM AIX> xlC -O2 -c hw1.cpp
  IBM AIX> xlC -o hw1.app hw1.o -lm

  other unix> CC -O2 -c hw1.cpp
  other unix> CC -o hw1.app hw1.o -lm
  ```

  Note: -Wall is a g++-specific option

Collect compiler commands in a script

- Even for small test programs it is tedious to write the compilation and linking commands
- Automate with a script!

```
#!/bin/sh
g++ -Wall -O3 -c hw1.cpp
g++ -o hw1.app hw1.o -lm
```

or parameterize the program name:

```
#!/bin/sh
progname=$1
g++ -Wall -O3 -c $progname.cpp
g++ -o $progname.app $progname.o -lm
```

Running the script

- Suppose the name of the script is `compile.sh`
- Make the script executable:

```
unix> chmod a+x compile.sh
```

- Execute the script:

```
unix> ./compile.sh
```

or if it needs the program name as command-line argument:

```
unix> ./compile.sh hw1
```

The make.sh scripts in the course software

- Compiler name and options depend on the system
- Tip: make a script `make.sh` to set up suitable default compiler and options, and go through the compilation and linking
- With this course we have some `make.sh` scripts using environment variables in your start-up file (.bashrc, .cshrc):

  ```
  # C++ compiler and associated options:
  CPP_COMPILER
  CPP_COMPILER_OPTIONS
  ```

  If not defined, these are set according to the computer system you are on (detected by `uname -s`)

The make.sh script (1)

```
#!/bin/sh

# determine compiler options (check first if the environment
# variable CPP_COMPILER is set):
if [ ! -n "$CPP_COMPILER" ]; then
  # base CPP_COMPILER on the current machine type:
  case ‘uname -s‘ in
        Linux)
                CPP_COMPILER=g++
                CPP_COMPILER_OPTIONS="-Wall -O3"
                ;;
        AIX)
                CPP_COMPILER=xlC
                CPP_COMPILER_OPTIONS="-O"
                ;;
        SunOS)
                CPP_COMPILER=CC
                CPP_COMPILER_OPTIONS="-O3"
                ;;
        *)
                # GNU’s gcc is available on most systems...
                C_COMPILER=gcc
                C_COMPILER_OPTIONS="-Wall -O3"
                ;;
    esac
fi
```

The make.sh script

```
# fetch all C++ files:
files=`/bin/ls *.cpp`

for file in $files; do
  stem=`echo $file | sed 's/\.cpp$//'`
  echo $CPP_COMPILER $CPP_COMPILER_OPTIONS -I. -o $stem.app $file -lm
  $CPP_COMPILER $CPP_COMPILER_OPTIONS -I. -o $stem.app $file -lm
  ls -s $stem.app
done
```

How to compile and link C programs

- To use GNU's compiler: just replace `g++` by `gcc`
- On other systems:

```
IBM AIX> xlc -O2 -c hw1.c
IBM AIX> xlc -o hw1.app hw1.o -lm

other unix> cc -O2 -c hw1.c
other unix> cc -o hw1.app hw1.o -lm
```

How to compile and link in general

- We compile a bunch of Fortran, C and C++ files and link these with some libraries
- Compile each set of files with the right compiler:

```
unix> g77 -O3 -I/some/include/dir -c *.f
unix> gcc -O3 -I/some/other/include/dir -I. -c *.c
unix> g++ -O3 -I. -c *.cpp
Each command produces a set of corresponding object files
with extension .o
```

- Then link:

```
unix> g++ -o executable_file -L/some/libdir -L/some/other/libdir \
         *.o -lmylib -lyourlib -lstdlib
```

  Here, we link all `*.o` files with three libraries: `libmylib.a`, `libyourlib.so`, `libstdlib.so`, found in `/some/libdir` or `/some/other/libdir`
- Library type: `lib*.a`: static; `lib*.so`: dynamic

Executables vs. libraries

- A set of object files can be linked with a set of libraries to form an executable program, provided the object files contains one main program
- If the main program is missing, one can link the object files to a static or sheared library `mylib2`:

```
unix> g++ -shared -o libmylib2.so *.o
unix> g++ -static -o libmylib2.a  *.o
```

- If you write a main program in `main.cpp`, you can create the executable program by

```
unix> g++ -O -c main.cpp  # create main.o
unix> g++ -o executable_file main.o -L. -lmylib2
```

Makefiles

- Compiling and linking are traditionally handled by makefiles
- The `make` program executes the code in makefiles
- Makefiles have an awkward syntax and the make language is primitive for text processing and scripting
- The (old) important feature of make is to check time stamps in files and only recompile the required files
- I have stopped using makefiles – I prefer plain scripts

Things can easily go wrong in C

- Let's try a version of the program where we fail to include stdlib.h (i.e. the compiler does not see the declaration of `atof`)

  ```
  unix> gcc -o tmp -O3 hw-error.c
  unix> ./tmp 2.3
  Hello, World! sin(1.07374e+09)=-0.617326
  ```

  File: src/C/hw/hw-error.c
- The number 2.3 was not read correctly...
- argv[1] is the string `"2.3"`
- r is not 2.3 (!)
- The program compiled and linked successfully!

Remedy

- Use the C++ compiler, e.g.

```
unix> g++ -o tmp -O3 hw-error.c
hw-error.c: In function ‘int main(int, char **)’:
hw-error.c:9: implicit declaration of function ‘int atof(...)’
```

- or use gcc -Wall with gcc:

```
unix> gcc -Wall -o tmp -O3 hw-error.c
hw-error.c: In function ‘main’:
hw-error.c:9: warning: implicit declaration of function ‘atof’
```

The warning tells us that the compiler cannot see the declaration of atof, i.e., a header file with atof is missing

Example: Data transformation

- Suppose we have a file with xy-data:

  ```
  0.1 1.1
  0.2  1.8
  0.3 2.2
  0.4   1.8
  ```

  and that we want to transform the y data using some mathematical function f(y)
- Goal: write a C++ program that reads the file, transforms the y data and write new xy-data to a new file

Program structure

1. Read name of input and output files as command-line arguments
2. Print error/usage message if less than two command-line arguments are given
3. Open the files
4. While more data in the file:
    1. read x and y from the input file
    2. set y = myfunc(y)
    3. write x and y to the output file
5. Close the files

File: src/C++/datatrans/datatrans1.cpp

The C++ code (1)

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <math.h>

double myfunc(double y)
{
  if (y >= 0.0) {
    return pow(y,5.0)*exp(-y);
  } else {
    return 0.0;
  }
}

int main (int argc, char* argv[])
{
  char*  infilename;  char* outfilename;
  /* abort if there are too few command-line arguments */
  if (argc <= 2) {
    std::cout << "Usage: " << argv[0] << " infile outfile" << '\n';
    exit(1);
  } else {
    infilename = argv[1]; outfilename = argv[2];
  }
```

The C++ code (2)

```
  std::ifstream ifile( infilename);
  std::ofstream ofile(outfilename);
  std::cout << argv[0] << ": converting " << infilename << " to "
    << outfilename << '\n';
  double x, y;
  int ok = true;  // boolean variable for not end of file
  while (ok) {
    if (!(ifile >> x >> y)) ok = false;
    if (ok) {
      y = myfunc(y);
      ofile.unsetf(std::ios::floatfield);
      ofile << x << " ";
      ofile.setf(std::ios::scientific, std::ios::floatfield);
      ofile.precision(5);
      ofile << y << std::endl;
    }
  }
  ifile.close();  ofile.close();
  return 0;
}
```

We can avoid the prefix `std::` by writing

```
using namespace std; /* e.g.: cout now means std::cout */
```

C++ file opening

- File handling in C++ is implemented through classes
- Open a file for reading (`ifstream`):

  ```
  #include <fstream>
  const char* filename1 = "myfile";
  std::ifstream ifile(filename1);
  ```

- Open a file for writing (`ofstream`):

  ```
  std::string filename2 = filename1 + ".out"
  std::ofstream ofile(filename2);  // new output file
  ```

  or open for appending data:

  ```
  std::ofstream ofile(filename2, ios_base::app);
  ```

C++ file reading and writing

- 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;
```

- Print to file:

```
ofile << x << " " << y << '\n';
```

- Of course, C's I/O and file handling can be used

```
#include <cstdio>  // official C++ name for stdio.h
```

```
call ios::sync_with_stdio() if stdio/iostream are mixed
```

Formatted output with iostream tools

- To set the type of floating-point format, width, precision, etc, use member functions in the output object:

```
ofile.setf(std::ios::scientific, std::ios::floatfield);
ofile.precision(5);
```

- I find such functions tedious to use and prefer `printf` syntax instead

Formatted output with printf tools

- The iostream library offers comprehensive formatting control
- printf-like functions from C makes the writing faster (and more convenient?)
- Writing to standard output:

```
printf("f(%g)=%12.5e for i=%3d\n",x,f(x),i);
```

- There is a family of printf-like functions:
  1. `printf` for writing to standard output
  2. `fprintf` for writing to file
  3. `sprintf` for writing to a string
- Writing to a file: use `fprintf` and C-type files, or use C++ files with the `oform` tool on the next slide

A convenient formatting tool for C++

- Use the C function `sprintf` to write to a string with printf-like syntax:

```
char buffer[200];
sprintf(buffer, "f(%g)=%12.5e for i=%3d",x,f(x),i);
std::cout << buffer;
```

- This construction can be encapsulated in a function:

```
std::cout << oform("f(%g)=%12.5e for i=%3d",x,f(x),i);
```

```
char* oform (const char* fmt, ...) /* variable no of args! */
{
  va_list ap;  va_start(ap, fmt);
  static char buffer[999]; // allocated only once
  vsprintf (buffer, fmt, ap);
  va_end(ap);
  return buffer;
}
```

`static` variables in a function preserve their contents from call to call

The printf syntax

- The printf syntax is used for formatting output in many C-inspired languages (Perl, Python, awk, partly C++)
- Example: write

  ```
  i= 4, r=0.7854, s= 7.07108E-01, method=ACC
  ```

  i.e.
- i=[integer in a field of width 2 chars]
- r=[float/double written as compactly as possible]
- s=[float/double written with 5 decimals, in scientific notation, in a field of width 12 chars]
- method=[text]
- This is accomplished by

  ```
  printf("i=%2d, r=%g, s=%12.5e, method=%s\n", i, r, s, method);
  ```

More about I/O in C++

- General output object: `ostream`
- General input object: `istream`
- `ifstream` (file) is a special case of `istream`
- `ofstream` (file) is a special case of `ostream`
- Can write functions

  ```
  void print (ostream& os) { ... }
  void scan  (istream& is) { ... }
  ```

  These work for both cout/cin and ofstream/ifstream
- That is, one print function can print to several different media

What is actually the argv array?

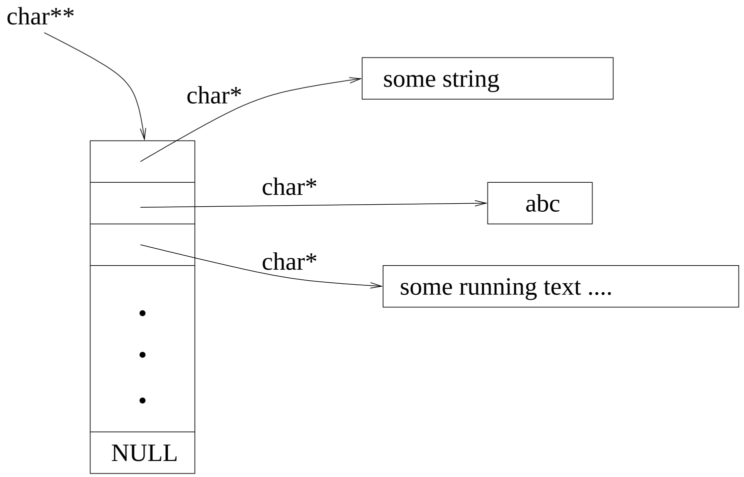
- argv is an array of strings

  ```
  # C/C++ declaration:
  char** argv;
  # or
  char* argv[];
  ```

- argv is a double pointer; what this means in plain English is that
  1. there is an array somewhere in memory
  2. argv points to the first entry of this array
  3. entries in this array are pointers to other arrays of characters (char*), i.e., strings

  Since the first entry of the argv array is a char*, argv is a pointer to to a pointer to char, i.e., a double pointer (char**)

The argv double pointer



Type conversion

- The `atof` function returns a float, which is then stored in a double

  ```
  r = atof(argv[1]);
  ```

- C/C++ transforms floats to doubles implicitly
- The conversion can be written explicitly:

  ```
  r = (double) atof(argv[1]);  /* C style */
  r = double(atof(argv[1]));   // C++ style
  ```

- Explicit variable conversion is a good habit; it is safer than relying on implicit conversions

Data transformation example in C

- Suppose we have a file with xy-data:

  ```
  0.1 1.1
  0.2  1.8
  0.3 2.2
  0.4   1.8
  ```

  and that we want to transform the y data using some mathematical function f(y)
- Goal: write a C program that reads the file, transforms the y data and write the new xy-data to a new file

Program structure

1. Read name of input and output files as command-line arguments
2. Print error/usage message if less than two command-line arguments are given
3. Open the files
4. While more data in the file:
    1. read x and y from the input file
    2. set y = myfunc(y)
    3. write x and y to the output file
5. Close the files

File: src/C/datatrans/datatrans1.c

The C code (1)

```
#include <stdio.h>
#include <math.h>

double myfunc(double y)
{
  if (y >= 0.0) {
    return pow(y,5.0)*exp(-y);
  } else {
    return 0.0;
  }
}
```

The C code (2)

```
int main (int argc, char* argv[])
{
  FILE *ifile;  /* input  file */
  FILE *ofile;  /* outout file */
  double x, y;
  char *infilename;
  char *outfilename;
  int  n;
  int  ok;

  /* abort if there are too few command-line arguments */
  if (argc < 3) {
    printf("Usage: %s infile outfile\n", argv[0]); exit(1);
  } else {
    infilename = argv[1]; outfilename = argv[2];
  }
  printf("%s: converting %s to %s\n",
         argv[0],infilename,outfilename);
  ifile = fopen( infilename, "r"); /* open for reading */
  ofile = fopen(outfilename, "w"); /* open for writing */
```

The C code (3)

```
  ok = 1;  /* boolean (int) variable for detecting end of file */
  while (ok) {
    n = fscanf(ifile, "%lf%lf", &x, &y); /* read x and y */
    if (n == 2) {
      /* successful read in fscanf: */
      printf("%g %12.5e\n", x, y);
      y = myfunc(y);
      fprintf(ofile, "%g %12.5e\n", x, y);
    } else { /* no more numbers */ ok = 0; }
  }
  fclose(ifile);  fclose(ofile);  return 0;
}
```

Major differences from the C++ version

- Use of `FILE*` pointers instead of `ifstream` and `ofstream`
- Use of `fscanf` and `fprintf` instead of

```
ifile >> object;
ofile << object;
```

- You can choose any of these two I/O tools in C++

C file opening

- Open a file:

```
FILE *somefile;
somefile = fopen("somename", "r" /* or "w" */);
if (somefile == NULL) {
   /* unsuccessful open, write an error message */
   ...
}
```

- More C-ish style of the if-test:

```
if (!somefile) { ... }
```

C file reading and writing

- Read something from the file:

```
double a; int b; char c[200];
n = fscanf(somefile, "%lf%d%s", &a, &b, c);

/* %lf means long float, %d means integer, %s means string */
/* n is the no of successfully converted items */

/* variables that are to be set inside the function, as in
   fscanf, must be preceeded by a &, except arrays (c is
   a character array - more about this later)
*/

/* fscanf returns EOF (predefined constant) when reaching
   the end-of-file mark
*/
```

- Print to file:

```
fprintf(ofile, "Here is some text: %g %12.5e\n", x, y);
```

Read until end of file

- Method 1: read until `fscanf` fails:

```
ok = 1;  /* boolean variable for not end of file */
while (ok) {
  n = fscanf(ifile, "%lf%lf", &x, &y); /* read x and y */
  if (n == 2) {
    /* successful read in fscanf: */ ... }
  } else {
    /* didn't manage to read two numbers, i.e.
       we have reached the end of the file
     */
    ok = 0;
  }
}
```

- Notice that `fscanf` reads structured input; errors in the file format are difficult to detect
- A more fool-proof and comprehensive approach is to read character by character and interpret the contents

Next example: matrix-vector product

- Goal: calculate a matrix-vector product
- Declare a matrix A and vectors x and b
- Initialize A
- Perform b = A*x
- Check that b is correct

What to learn

- How one- and multi-dimensional are created in C and C++
- Dynamic memory management
- Loops over array entries
- More flexible array objects in C++
- C and C++ functions
- Transfer of arguments to functions
- Pointers and references

Basic arrays in C and C++

- C and C++ use the same basic array construction
- These arrays are based on pointers to memory segments
- Array indexing follows a quickly-learned syntax:
  `q[3][2]` is the same as `q(3,4)` in Fortran, because
  1. C/C++ (multi-dimensional) arrays are stored row by row (Fortran stores column by column)
  2. base index is 0 (Fortran applies 1)

Declaring basic C/C++ vectors

- Basic C/C++ arrays are somewhat clumsy to define
- C++ has more high-level vectors in its Standard Template Library, or one can use third-party array objects or write one's own
- Declaring a fixed-size vector in C/C++ is very easy:

```
#define N 100

double x[N];
double b[50];
```

- Vector indices start at 0
- Looping over the vector:

```
int i;
for (i=0; i<N; i++) {
  x[i] = f(i) + 3.14;
}

double f(int i) { ... }  /* definition of function f */
```

Declaring basic C matrices

- Declaring a fixed-size matrix:

```
/* define constants N and M: */
#define N 100
#define M 100

double A[M][N];
```

- Array indices start at 0
- Looping over the matrix:

```
int i,j;
for (i=0; i<M; i++) {
  for (j=0; j<N; j++) {
     A[i][j] = f(i,j) + 3.14;
  }
}
```

Matrix storage scheme

- Note: matrices are stored row wise; the column index should vary fastest
- Recall that in Fortran, matrices are stored column by column
- Typical loop in Fortran (2nd index in outer loop):

```
for (j=0; j<N; j++) {
  for (i=0; i<M; i++) {
     A[i][j] = f(i,j) + 3.14;
  }
}
```

  But in C and C++ we now traverse `A` in jumps!

Dynamic memory allocation

- The length of arrays can be decided upon at run time and the necessary chunk of memory can be allocated while the program is running
- Such dynamic memory allocation is error-prone!
- You need to allocate *and deallocate* memory
- C++ programmers are recommended to use a library where dynamic memory management is hidden
- We shall explain some details of dynamic memory management; you should know about it, but not necessarily master the details

Dynamic memory allocation in C

- Static memory allocation (at compile time):

```
double x[100];
```

- Dynamic memory allocation (at run time):

```
double* x;
x = (double*) calloc(n, sizeof(double));
/* or: */
x = (double*) malloc(n*sizeof(double));
```

- `calloc`: allocate and initialize memory chunk (to zeros)
- `malloc`: just allocate a memory chunk
- Free memory when it is no longer used:

```
free(x);
```

Dynamic memory allocation in C++

- The ideas are as in C (allocate/deallocate), but
- C++ uses the functions `new` and `delete` instead of `malloc` and `free`

```
double* x = new double[n];   // same as malloc
delete [] x;                 // same as free(x)

// allocate a single variable:
double* p = new double;
delete p;
```

- *Never* mix `malloc`/`calloc`/`free` with `new`/`delete`!

```
double* x = new double[n];
...
free(x);  // dangerous
```

High-level vectors in C++

- C++ has a Standard Template Library (STL) with vector types, including a vector for numerics:

  ```
  std::valarray<double> x(n);  // vector with n entries
  ```

- It follows the subscripting syntax of standard C/C++ arrays:

  ```
  int i;
  for (i=0, i<N; i++) {
    x[i] = f(i) + 3.14;
  }

  // NOTE: with STL one often avoids for-loops
  // (more about this later)
  ```

- STL has no matrix type!

Storage of vectors

- A vector is actually just a pointer to the first element:

  ```
  double* x;     // dynamic vector
  double  y[N]; // vector with fixed size at compile time
  ```

  Note: one can write

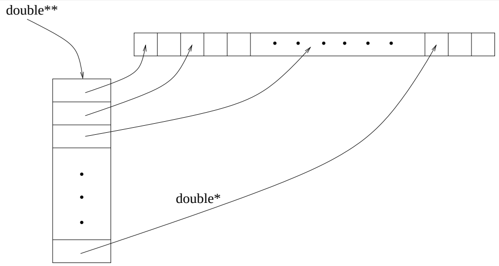
  ```
  double *x;
  /* or */
  double* x;
  ```

  (the first is C style, the second is C++ style...)

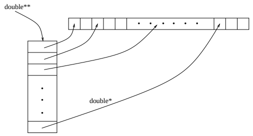
Storage of matrices

- A matrix is represented by a double pointer (e.g. `double**`) that points to a contiguous memory segment holding a sequence of `double*` pointers
- Each `double*` pointer points to a row in the matrix

```
double** A;    // dynamic matrix
A[i] is a pointer to the i+1-th row
A[i][j] is matrix entry (i,j)
```



Allocation of a matrix in C



- Allocate vector of pointers to rows:

```
A = (double**) calloc(n, sizeof(double*));
```

- Allocate memory for all matrix entries:

```
A[0] = (double*) calloc(n*n, sizeof(double));
```

- Set the row pointers to the correct memory address:

```
for (i=1; i<n; i++) A[i] = A[0] + n*i;
```

- C++ style allocation:

```
A = new double* [n]; A[0] = new double [n*n];
```

Deallocation of a matrix in C

- When the matrix is no longer needed, we can free/deallocate the matrix
- Deallocation syntax:

```
free(A[0]);  /* free chunk of matrix entries*/
free(A);     /* free array of pointers to rows */
```

- C++ style:

```
delete [] A[0];
delete [] A;
```

Warning: be careful with dynamic memory management!

- Working with pointers, malloc/calloc and free is notoriously error-prone!
- Avoid explicit memory handling if you can, that is, use C++ *libraries* with classes that hide dynamic memory management
- Tip: Stroustrup's `Handle` class offers a smart pointer (object with pointer-like behavior) that eliminates the need for explicit `delete` calls
- Source can be found in `src/C++/Wave2D/Handle.h`

A glimpse of the Handle class

```
template< typename T > class Handle
{
  T* pointer;  // pointer to actual object
  int* pcount; // the number of Handle's pointing to the same object

public:
  explicit Handle(T* pointer_)
     : pointer(pointer_), pcount(new int(1)) {}

  explicit Handle(const Handle<T>& r) throw()
     : pointer(r.pointer), pcount(r.pcount) { ++(*pcount); }

  ~Handle() throw()
      { if (--(*pcount) == 0) { delete pointer; delete pcount; } }

  T* operator->() { return pointer; }
  T& operator*()  { return *pointer; }

  Handle& operator= (const Handle& rhs) throw() {
      if (pointer == rhs.pointer) return *this;
      if (--(*pcount) == 0) {
            delete pointer; delete pcount;
      }
      pointer = rhs.pointer;
      pcount = rhs.pcount;
      ++(*pcount);
      return *this;
```

Using our own array type

- In C++ we can hide all the allocation/deallocation details in a new type of variable
- For convenience and educational purposes we have created the special type `MyArray`:

```
MyArray<double> x(n), A(n,n), b(n);

// indices start at 1:
for (i=1; i <=n; i++) {
  x(i) = ...;
  A(3,i) = ...;
}
```

- `MyArray` indexing is inspired by Fortran 77: data are stored column by column and the first index is 1 (not 0!)
- `MyArray` is a dynamic type with built-in new/delete
- `MyArray`'s internal storage: a plain C vector

Declaring and initializing A, x and b

```
MyArray<double> A, x, b;
int n;
if (argc >= 2) {
  n = atoi(argv[1]);
} else {
  n = 5;
}
A.redim(n,n);  x.redim(n);  b.redim(n);

int i,j;
for (j=1; j<=n; j++) {
  x(j) = j/2.0;
  for (i=1; i<=n; i++) {
    A(i,j) = 2.0 + double(i)/double(j);
  }
}
```

Matrix-vector product loop

- Computation:

```
double sum;
for (i=1; i<=n; i++) {
  sum = 0.0;
  for (j=1; j<=n; j++) {
    sum += A(i,j)*x(j);
  }
  b(i) = sum;
}
```

- Note: we traverse `A` column by column because `A` is stored (and indexed) in Fortran fashion

Complete code: src/C++/mv/mv2.cpp

The corresponding C version

- Explicit allocation/deallocation of vector/matrix
- The core loop is not that different:

```
for (i=0; i<n; i++) {
  x[i] = (i+1)/2.0;
  for (j=0; j<n; j++) {
    A[i][j] = 2.0 + (((double) i)+1)/(((double) j)+1);

    if (n < 10) { printf("A(%d,%d)=%g\t", i,j,A[i][j]); }
  }
  if (n < 10) { printf("  x(%d)=%g\n", i,x[i]); }
}
```

Subprograms in C++

- Subprograms are called *functions* in C++
- `void` as return type signifies subroutines in Fortran (no return value)
- A function with return value:

```
double f(double x) { return sin(x)*pow(x,3.2); }  // as in C
```

- Default transfer of arguments: “call by value”, i.e., in

```
x1 = 3.2;
q = f(x1)
```

  `f` takes a *copy* `x` of `x1`

Call by reference

Problem setting: How can changes to a variable inside a function be visible in the calling code?

- C applies pointers,

```
int n; n=8;
somefunc(&n);  /* &n is a pointer to n */

void somefunc(int *i)
{
  *i = 10; /* n is changed to 10 */
  ...
}
```

- Pointers also work in C++ (C is a subset of C++!), but in C++ it is standard to use *references*

```
int n; n=8;
somefunc(n);  /* just transfer n itself */

void somefunc(int& i)  // reference to i
{
  i = 10; /* n is changed to 10 */
  ...
}
```

Always use references for large objects

- This function implies a copy of x:

  ```
  void somefunc(MyArray<double> x)
  { ... }
  ```

  Copying is inefficient if x is large!!
- Here only a reference (kind of address) is transferred to the function:

  ```
  void somefunc(MyArray<double>& x)
  {
    // can manipulate the entries in x
    x(5) = 10;  // ok
  }
  ```
- Manipulation of the array can be avoided using the const keyword:

  ```
  void somefunc(const MyArray<double>& x)
  {
    // can NOT manipulate the entries in x
    x(5) = 10;  // illegal to assign new values
    r = x(1);   // ok to read array entries
  ```

A C++ function

- Initialize A and x in a separate function:

```
void init (MyArray<double>& A, MyArray<double>& x)
{
  const int n = x.size();
  int i,j;
  for (j=1; j<=n; j++) {
    x(j) = j/2.0;  /* or completely safe: double(j)/2.0 */
    for (i=1; i<=n; i++) {
      A(i,j) = 2.0 + double(i)/double(j);
    }
  }
}
```

- Notice that `n` is not transferred as in C and Fortran 77; `n` is a part of the `MyArray` object

Subprograms in C

- The major difference is that C has not references, only pointers
- Call by reference (change of input parameter) must use pointers:

```
void init (double **A, double *x, int n)
{
  int i,j;
  for (i=1; i<=n; i++) {
    x[i] = (i+1)/2.0;
    for (j=1; j<=n; j++) {
      A[i][j] = 2.0 + (((double) i)+1)/(((double) j)+1);
    }
  }
}
```

More about pointers

- A pointer holds the memory address to a variable

```
int* v;   /* v is a memory address */
int  q;   /* q is an integer */
q=1;
v = &q;   /* v holds the address of q */
*v = 2;   /* q is changed to 2 */
```

- In function calls:

```
int n; n=8;
somefunc(&n);

void somefunc(int *i)  /* i becomes a pointer to n */
{
  /* i becomes a copy of the pointer to n, i.e.,
     i also points to n.
  */
  *i = 10; /* n is changed to 10 */
  ...
}
```

Array arguments in functions

- Arrays are always transferred by pointers, giving the effect of call by reference
- That is, changes in array entries inside a function is visible in the calling code

```
void init (double** A, double* x, int n)
{
  /* initialize A and x ... */
}

init(A, x, n);
/* A and x are changed */
```

Pointer arithmetics

- Manipulation with pointers can increase the computational speed
- Consider a plain for-loop over an array:

  ```
  for (i=0; i<n; ++i) { a[i] = b[i]; }
  ```

- Equivalent loop, but using a pointer to visit the entries:

  ```
  double *astop, *ap, *bp;
  astop = &a[n - 1]; /* points to the end of a */
  for (ap=a, bp=b; a <= astop; ap++, bp++)  *ap = *bp;
  ```

  This is called pointer arithmetic
- What is the most efficient approach?

Preprocessor directives

- The compilation process consists of three steps (the first is implicit):
  1. run the preprocessor
  2. compile
  3. link
- The preprocessor recognices special *directives*:

  ```
  #include <math.h>  /* lines starting with #keyword */
  ```

  meaning: search for the file math.h, in /usr/include or directories specified by the -I option to gcc/cc, and copy the file into the program
- Directives start with #
- There are directives for file include, if-tests, variables, functions (macros)

Preprocessor if-tests

- If-test active at compile time:

```
for (i=0; i<n; i++) {
#ifdef DEBUG
  printf("a[%d]=%g\n",i,a[i])
#endif
```

  Compile with DEBUG defined or not:

```
unix> gcc -DDEBUG -Wall -o app mp.c   # DEBUG is defined
unix> gcc -UDEBUG -Wall -o app mp.c   # DEBUG is undefined
unix> gcc -Wall -o app mp.c           # DEBUG is undefined
```

Macros

- Macros for defining constants:

  ```
  #define MyNumber 5
  ```

  meaning: replace the text `MyNumber` by `5` anywhere

- Macro with arguments (a la text substitution):

  ```
  #define SQR(a) ((a)*(a))
  ```

  ```
  #define MYLOOP(start,stop,incr,body) \
     for (i=start; i<=stop; i=i+incr) \
       { body }
  ```

  ```
  r = SQR(1.2*b);
  MYLOOP(1,n,1, a[i]=i+n; a[i]=SQR(a[i]);)
  ```

How to examine macro expansions

- You can first run the preprocessor on the program files and then look at the source code (with macros expanded):

```
unix> g++ -E -c mymacros.cpp
```

Output will be in `mymacros.o`

```
r = ( ( 1.2*b )*( 1.2*b ) );
for (i= 1 ; i<= n ; i=i+ 1 )
{   a[i]=i+n; a[i]= ( a[i] )*( a[i] ) ;  }
```

A useful debug macro

```
void debugprint(char *str, int line, char *file)
{ printf("%s, line %6d: %s\n",file,line,str); }

#ifdef DEBUG
/* define debug as call to debugprint */
#define debug(s) debugprint(s,__LINE__,__FILE__)
/* __LINE__ and __FILE__ are predefined preprocessor macros */
#else
/* define debug as empty string */
#define debug(s)
#endif

debug("some debug line");  /* active/deactive; depends on DEBUG */
debug(oform("r=%g, b=%g, i=%d, a[0]=%f",r,b,i,a[0]));

output:
macros.c, line     35: r=21.538, b=3.86742, i=10, a[0]=100.0
```

Single vs double precision

- Can introduce a macro real:

  ```
  real myfunc(real x, real y, real t)
  { ... }
  ```

- Define real at compile time

  ```
  gcc -Dreal=double ...
  ```

  or in the code:

  ```
  #define real float
  ```

  (in some central header file)

- If hardcoded, using typedef is considered as a more fool-proof style:

  ```
  typedef double real;  /* define real as double */
  ```

Macros and C++

- Message in C++ books: avoid macros
- Macros for defining constants

  ```
  #define n 5
  ```

  are in C++ replaced by const variables:

  ```
  const int n = 5;
  ```

- Macros for inline functions

  ```
  #define SQR(a) (a)*(a)
  ```

  are in C++ replaced by *inline* functions:

  ```
  inline double sqr (double a) { return a*a; }
  ```

- Much less use of macros in C++ than in C

Requirements to solutions of exercises

- Write as clear and simple code as possible
- (Long and tedious code is hard to read)
- (Too short code is hard to read and dissect)
- Use comments to explain *ideas* or intricate details
- All exercises must have a test example, “proving” that the implementation works!
- Output from the test example must be included!

Exercise 1: Modify the C++ Hello World program

- Locate the first Hello World program
- Compile the program and test it (manually and with ../make.sh)
- Modification: write "Hello, World!" using cout and the sine-string using printf

Exercise 2: Extend the C++ Hello World program

- Locate the first Hello World program
- Read three command-line arguments: `start`, `stop` and `inc`
- Provide a "usage" message and abort the program in case there are too few command-line arguments
- For `r=start` step `inc` until `stop`, compute the sine of r and write the result
- Write an additional loop using a `while` construction
- Verify that the program works

Exercise 3: Integrate a function (1)

- Write a function

```
double trapezoidal(userfunc f, double a, double b, int n)
```

that integrates a user-defined function function `f` between `a` and `b` using the Trapezoidal rule with `n` points:

$$\int_a^b f(x)dx \approx h\left(\frac{f(a)}{2} + \frac{f(b)}{2} + \sum_{i=1}^{n-2} f(a+ih)\right), \quad h = \frac{b-a}{n-1}.$$

The user-defined function is specified as a *function pointer*:

```
typedef double (*userfunc)(double x);
```

Exercise 3: Integrate a function (2)

- Any function taking a `double` as argument and returning `double`, e.g.,

  ```
  double myfunc(double x) { return x + sin(x); }
  ```

  can now be used as a `userfunc` type, e.g.,

  ```
  integral_value = trapezoidal(myfunc, 0, 2, 100);
  ```

- Verify that `trapezoidal` is implemented correctly (hint: linear functions should be integrated exactly)

Binary format

- A number like π can be represented in ASCII format as `3.14` (4 bytes) or `3.14159E+00` (11 bytes), for instance
- In memory, the number occupies 8 bytes (a `double`), this is the binary format of the number
- The binary format (8 bytes) can be stored directly in files
- Binary format (normally) saves space, and input/output is much faster since we avoid translatation between ASCII chars and the binary repr.
- The binary format varies with the hardware and occasionally with the compiler version
- Two types of binary formats: little and big endian
- Motorola and Sun: big endian; Intel and Compaq: little endian

Exercise 4: Work with binary data in C (1)

- Scientific simulations often involve large data sets and binary storage of numbers saves space in files
- How to write numbers in binary format in C:

```
/* f is some FILE* pointer */

/* r is some double, n is some int */
fwrite((void*) &r, sizeof(r), 1, f);
fwrite((void*) &n, sizeof(n), 1, f);

/* a is some double* array of length n */
fwrite((void*) a, sizeof(double), n, f);
```

- `fwrite` gets `r` as an array of bytes (rather than array of `double`s), and the sequence of bytes is dumped to file
- Reading binary numbers follow the same syntax; just replace `fwrite` by `fread`

Exercise: Work with binary data in C (2)

- Create `datatrans2.c` (from `datatrans1.c`) such that the input and output data are in binary format
- To test the `datatrans2.c`, we need utilities to create and read binary files
  1. make a small C program that generates n xy-pairs of data and writes them to a file in binary format (read n from the command line),
  2. make a small C program that reads xy-pairs from a binary file and writes them to the screen

  With these utilities you can create input data to `datatrans2.c` and view the file produced by `datatrans2.c`

Exercise: Work with binary data in C (3)

- Modify the `datatrans2.c` program such that the x and y numbers are stored in one long (dynamic) array
- The storage structure should be x1, y1, x2, y2, ...
- Read and write the array to file in binary format using one `fread` and one `fwrite` call
- Try to generate a file with a huge number (10 000 000?) of pairs and use the Unix `time` command to test the efficiency of reading/writing a single array in one `fread`/`fwrite` call compared with reading/writing each number separately

Exercise 5: Work with binary data in C++

- Do the C version of this exercise first!
- How to write numbers in binary format in C++:

```
/* os is some ofstream object */

/* r is some double, n is some int */
os.write((char*) &r, sizeof(double));
os.write((char*) &n, sizeof(int));

/* a is some double* array of length n */
os.write((char*) a, sizeof(double)*n);

/* is is some std::ifstream object */
is.read((char*) &r, sizeof(double));
is.read((char*) &n, sizeof(int));
is.read((char*) a, sizeof(double)*n);
```

- Modify the `datatrans1.cpp` program such that it works with binary input and output data (use the C utilities in the previous exercise to create input file and view output file)

Exercise 6: Efficiency of dynamic memory allocation (1)

- Write this code out in detail as a stand-alone program:

```
#define NREPETITIONS 1000000
int i,n;
n = atoi(argv[1]);
for (i=1; i<=NREPETITIONS; i++)
{
  // allocate a vector of n doubles
  // deallocate the vector
}
```

Exercise 6: Efficiency of dynamic memory allocation (2)

- Write another program where each vector entry is allocated separately:

```
int i,j;
for (i=1; i<=NREPETITIONS; i++)
{
  // allocate each of the doubles separately:
  for (j=1; j<=n; j++)
  {
    // allocate a double
    // free the double
  }
}
```

Exercise: Efficiency of dynamic memory allocation (3)

- Measure the CPU time of vector allocations versus allocation of individual entries:

```
unix> time myprog1
unix> time myprog2
```

- Adjust NREPETITIONS such that the CPU time of the fastest program is of order 10 seconds (CPU measurements should last a few seconds, so one often adapts problem parameters to get CPU times of this order)

Traditional programming

Traditional procedural programming:

- subroutines/procedures/functions
- data structures = variables, arrays
- data are shuffled between functions

Problems with procedural approach:

- Numerical codes are usually large, resulting in lots of functions with lots of arrays (and their dimensions)
- Too many visible details
- Little correspondence between mathematical abstraction and computer code
- Redesign and reimplementation tend to be expensive

Programming with objects (OOP)

Programming with objects makes it easier to handle large and complicated codes:

- Well-known in computer science/industry
- Can group large amounts of data (arrays) as a single variable
- Can make different implementations look the same for a user
- Not much explored in numerical computing (until late 1990s)

Example: programming with matrices

Mathematical problem:

- Matrix-matrix product: $\mathbf{C} = \mathbf{MB}$
- Matrix-vector product: $\mathbf{y} = \mathbf{Mx}$

Points to consider:

- What is a matrix?
- a well defined mathematical quantity, containing a table of numbers and a set of legal operations
- How do we program with matrices?
- Do standard arrays in any computer language give good enough support for matrices?

A dense matrix in Fortran 77

Fortran syntax (or C, conceptually)

```
      integer p, q, r
      double precision M(p,q), B(q,r), C(p,r)
      double precision y(p), x(q)

C     matrix-matrix product: C = M*B
      call prodm(M, p, q, B, q, r, C)

C     matrix-vector product: y = M*x
      call prodv(M, p, q, x, y)
```

Drawback with this implementation:

- Array sizes must be explicitly transferred
- New routines for different precisions

Working with a dense matrix in C++

```
// given integers p, q, j, k, r
MatDense M(p,q);           // declare a p times q matrix
M(j,k) = 3.54;             // assign a number to entry (j,k)

MatDense B(q,r), C(p,r);
Vector   x(q), y(p);       // vectors of length q and p
C=M*B;                     // matrix-matrix product
y=M*x;                     // matrix-vector product
M.prod(x,y);               // matrix-vector product
```

Observe that

- we hide information about array sizes
- we hide storage structure (the underlying C array)
- the computer code is as compact as the mathematical notation

A dense matrix class

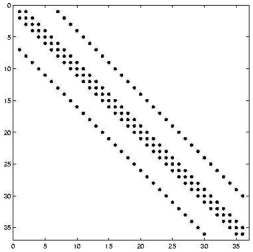
```
class MatDense
{
private:
  double** A;    // pointer to the matrix data
  int      m,n;  // A is an m times n matrix
public:
  //    ---  mathematical interface ---
  MatDense (int p, int q);                   // create pxq matrix
  double& operator () (int i, int j);        // M(i,j)=4; s=M(k,l);
  void operator = (MatDense& B);             // M = B;
  void prod (MatDense& B, MatDense& C);  // M.prod(B,C); (C=M*B)
  void prod (Vector& x, Vector& z);          // M.prod(y,z); (z=M*y)
  MatDense  operator * (MatDense& B);        // C = M*B;
  Vector    operator * (Vector& y);          // z = M*y;
  void size (int& m, int& n);                // get size of matrix
};
```

Notice that the storage format is hidden from the user

What is this object or class thing?

- A class is a collection of data structures and operations on them
- An object is a realization (variable) of a class
- The MatDense object is a good example:
  1. data: matrix size + array entries
  2. operations: creating a matrix, accessing matrix entries, matrix-vector products,..
- A class is a new type of variable, like reals, integers etc
- A class can contain other objects;
  in this way we can create complicated variables that are easy to program with

Extension to sparse matrices



- Matrix for the discretization of $-\nabla^2 u = f$.
- Only $5n$ out of n^2 entries are nonzero.
- Store only the nonzero entries!
- Many iterative solution methods for $\mathbf{Au} = \mathbf{b}$ can operate on the nonzeroes only

How to store sparse matrices (1)

$$\mathbf{A} = \begin{pmatrix} a_{1,1} & 0 & 0 & a_{1,4} & 0 \\ 0 & a_{2,2} & a_{2,3} & 0 & a_{2,5} \\ 0 & a_{3,2} & a_{3,3} & 0 & 0 \\ a_{4,1} & 0 & 0 & a_{4,4} & a_{4,5} \\ 0 & a_{5,2} & 0 & a_{5,4} & a_{5,5} \end{pmatrix}.$$

- Working with the nonzeroes only is important for efficiency!

How to store sparse matrices (2)

- The nonzeroes can be stacked in a one-dimensional array
- Need two extra arrays to tell where a row starts and the column index of a nonzero

$$\begin{aligned}
\texttt{A} &= (a_{1,1}, a_{1,4}, a_{2,2}, a_{2,3}, a_{2,5}, \ldots \\
\texttt{irow} &= (1, 3, 6, 8, 11, 14), \\
\texttt{jcol} &= (1, 4, 2, 3, 5, 2, 3, 1, 4, 5, 2, 4, 5).
\end{aligned}$$

$\Rightarrow$ more complicated data structures and hence more complicated programs

Sparse matrices in Fortran

Code example for $\mathbf{y} = \mathbf{Mx}$

```
integer p, q, nnz
integer irow(p+1), jcol(nnz)
double precision M(nnz), x(q), y(p)
...
call prodvs (M, p, q, nnz, irow, jcol, x, y)
```

Two major drawbacks:

- Explicit transfer of storage structure (5 args)
- Different name for two functions that perform the same task on two different matrix formats

Sparse matrix as a C++ class (1)

```
class MatSparse
{
  private:
    double* A;      // long vector with the nonzero matrix entries
    int*    irow;   // indexing array
    int*    jcol;   // indexing array
    int     m, n;   // A is (logically) m times n
    int     nnz;    // number of nonzeroes
  public:
    // the same functions as in the example above
    // plus functionality for initializing the data structures

    void prod (Vector& x, Vector& z);  // M.prod(y,z); (z=M*y)
};
```

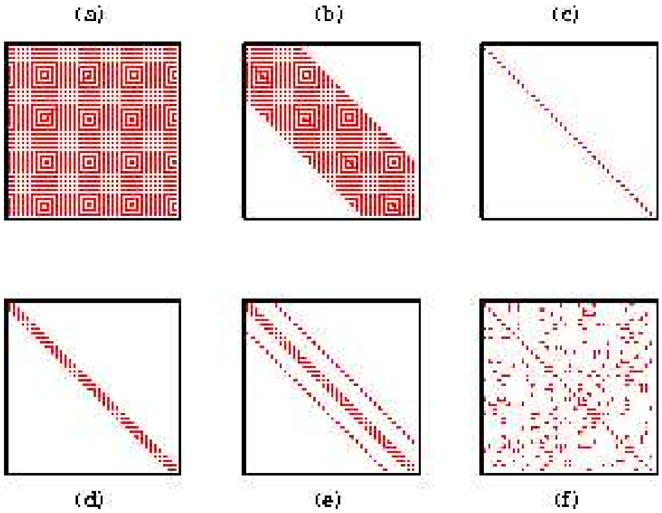
Sparse matrix as a C++ class (2)

- What has been gained?
- Users cannot see the sparse matrix data structure
- Matrix-vector product syntax remains the same
- The usage of MatSparse and MatDense is the same
- Easy to switch between MatDense and MatSparse

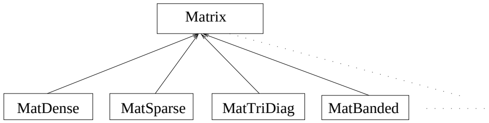
The jungle of matrix formats

- When solving PDEs by finite element/difference methods there are numerous advantageous matrix formats:
  - dense matrix
  - banded matrix
  - tridiagonal matrix
  - general sparse matrix
  - structured sparse matrix
  - diagonal matrix
  - finite difference stencil as matrix
- The efficiency of numerical algorithms is often strongly dependent on the matrix storage scheme
- Goal: hide the details of the storage schemes

Different matrix formats



The matrix class hierarchy



- Generic interface in *base class* Matrix
- Implementation of storage and member functions in the *subclasses*
- Generic programming in user code:

  ```
  Matrix& M;
  M.prod(x,y);   // y=M*x
  ```

  i.e., we need not know the structure of `M`, only that it refers to some concrete subclass object;
  C++ keeps track of *which* subclass object!
- `prod` must then be a *virtual* function

Object-oriented programming

- Matrix = object
- Details of storage schemes are hidden
- Common interface to matrix operations
- Base class: define operations, no data
- Subclasses: implement specific storage schemes and algorithms
- It is possible to program with the base class only!

Bad news...

- Object-oriented programming do wonderful things, but might be *inefficient*
- Adjusted picture:
  When indexing a matrix, one needs to know its data storage structure because of efficiency
- In the rest of the code one can work with the generic base class and its virtual functions

$\Rightarrow$ Object-oriented numerics: balance between efficiency and OO techniques

A simple class example

- We may use C++ classes to encapsulate C code and make C functions easier to use
- Example: a tool for measuring CPU time in programs
- We "wrap" a class around basic C library calls

Simple clock; C function interface

- `time.h` has a function `clock` for measuring the CPU time
- Basic usage:

```
#include <time.h>
clock_t t0 = clock();  // read CPU time
// do tasks ...
clock_t t1 = clock();
double cpu_time = (t1 - t0)/CLOCKS_PER_SEC;
```

More info; C function interface

- sys/times.h has a struct (class without functions) tms
- tms gives info about user time and system time of the current and all children processes
- tms is a C struct with data attributes

```
tms_utime    :   user time   (this process)
tms_stime    :   system time (this process)
tms_cutime   :   user time,   child process
tms_cstime   :   system time, child process
```

Example on using tms

- Basic usage (GNU/Linux):

```
#include <sys/times.h>  /* tms */
#include <unistd.h>     /* for clock ticks per sec */
tms t1, t2;
times(&t1);
/* perform operations... */
times(&t2);
tms diff;
// user time:
diff.tms_utime  = t2.tms_utime  - t1.tms_utime;
// system time:
diff.tms_stime  = t2.tms_stime  - t1.tms_stime;
// user time, children processes:
diff.tms_cutime = t2.tms_cutime - t1.tms_cutime;
// system time, children processes:
diff.tms_cstime = t2.tms_cstime - t1.tms_cstime;
double ticks = sysconf(_SC_CLK_TCK);
double cpu_time;
cpu_time = double(diff.tms_utime + diff.tms_stime)/ticks;
```

Desired easy-to-use C++ function interface

```
#include <CPUclock.h>

CPUclock clock;
clock.init();
// perform tasks ...
double cpu_time = clock.getCPUtime();
...

// perform more tasks
...
double cpu_time2 = clock.getCPUtime();

// perform even more tasks
...
double cpu_time3 = clock.getCPUtime();
```

`clock.getCPUtime()` returns the CPU time since the last call to the function

class CPUclock; simplest approach

```
class CPUclock
{
private:
  clock_t t0;
public:
  void init () { t0 = clock(); }
  double getCPUtime() {
    double t0_end = clock();
    double cpu = double((t0_end - t0)/CLOCKS_PER_SEC)
    t0 = clock_t(t0_end);
    return cpu;
  }
};
```

class CPUclock with tms struct

```
#ifndef CPUclock_H
#define CPUclock_H
#include <time.h>          // clock function

#ifdef HAS_TMS
#include <sys/times.h>     // tms struct
#endif

class CPUclock
{
private:
  clock_t t0;

#ifdef HAS_TMS
  tms t1, diff;
  double cpu_time, child_cpu_time;
#endif

public:
  void init ();
  double getCPUtime();
};
#endif
```

CPUclock.cpp (1)

```
#include <CPUclock.h>
#ifdef HAS_TMS
#include <unistd.h>
#endif

void CPUclock:: init ()
{
  t0 = clock();
#ifdef HAS_TMS
  times(&t1);
#endif
}
```

Note: the implementation may differ between platforms (e.g. Linux, SunOS, Windows)

CPUclock.cpp (2)

```
double CPUclock:: getCPUtime ()
{
  double t0_end = clock();
  double cpu_time_clock = double((t0_end - t0)/CLOCKS_PER_SEC);

#ifdef HAS_TMS
  tms t2;
  times(&t2);
  diff.tms_utime  = t2.tms_utime  - t1.tms_utime;
  diff.tms_stime  = t2.tms_stime  - t1.tms_stime;
  diff.tms_cutime = t2.tms_cutime - t1.tms_cutime;
  diff.tms_cstime = t2.tms_cstime - t1.tms_cstime;
  double clock_ticks_per_sec = sysconf(_SC_CLK_TCK);  // Linux
  cpu_time_clock = double(diff.tms_utime + diff.tms_stime) \
             /clock_ticks_per_sec;
  child_cpu_time = \
    double(diff.tms_cutime + diff.tms_cstime)/clock_ticks_per_sec;

  // update t1 such that next getCPUtime() gives new difference:
  times(&t1);
#endif
  t0 = clock_t(t0_end);

  return cpu_time_clock;
}
```

Why do we need classes to do this?

- We could have made a plain function interface, e.g.,

  ```
  CPUclock_init();
  // perform tasks ...
  double cpu_time = CPUclock_getCPUtime();
  ```

  to hide the original (long) C code
- Problem: we need to store `t0` and `t1` as a global variables
- The class solution is cleaner, easier to extend (e.g., return user time, system time, user time of child process, etc.)
- When functions need to remember a state (like `t0`), one is better off with a class

Extension

- Offer a function for system time:

```
double CPUclock:: getSystemTime()
{
#ifdef HAS_TMS
  return double(diff.tms_stime)/sysconf(_SC_CLK_TCK);
#endif
}
```

Complex arithmetic in C++

- Making a class for complex numbers is a good educational example
- Note: C++ already has a class `complex` in its standard template library (STL) – use that one for professional work

  ```
  #include <complex>
  std::complex<double> z(5.3,2.1), y(0.3);
  std::cout << z*y + 3;
  ```

- However, writing our own class for complex numbers is a very good exercise for novice C++ programmers!

Usage of our Complex class

```
#include "Complex.h"

void main ()
{
  Complex a(0,1);   // imaginary unit
  Complex b(2), c(3,-1);
  Complex q = b;

  std::cout << "q=" << q << ", a=" << a << ", b=" << b << "\n";

  q = a*c + b/a;

  std::cout << "Re(q)=" << q.Re() << ", Im(q)=" << q.Im() << "\n";
}
```

Basic contents of class Complex

- Data members: real and imaginary part
- Member functions:
  1. construct complex numbers

     ```
     Complex a(0,1);   // imaginary unit
     Complex b(2), c(3,-1);
     ```

  2. Write out complex numbers:

     ```
     std::cout << "a=" << a << ", b=" << b << "\n";
     ```

  3. Perform arithmetic operations:

     ```
     q = a*c + b/a;
     ```

Declaration of class Complex

```
class Complex
{
private:
  double re, im;  // real and imaginary part
public:
  Complex  ();                              // Complex c;
  Complex  (double re, double im = 0.0);  // Complex a(4,3);
  Complex  (const Complex& c);              // Complex q(a);
 ~Complex  () {}
  Complex& operator= (const Complex& c);  // a = b;
  double   Re () const;          // double real_part = a.Re();
  double   Im () const;          // double imag_part = a.Im();
  double   abs () const;         // double m = a.abs(); // modulus

  friend Complex operator+ (const Complex& a, const Complex& b);
  friend Complex operator- (const Complex& a, const Complex& b);
  friend Complex operator* (const Complex& a, const Complex& b);
  friend Complex operator/ (const Complex& a, const Complex& b);
};
```

`friend` means that stand-alone functions can work on private parts (`re`, `im`)

The simplest member functions

- Extract the real and imaginary part (recall: these are private, i.e., invisible for users of the class; here we get a copy of them for reading)

```
double Complex:: Re () const { return re; }
double Complex:: Im () const { return im; }
```

- What is `const`? see next slide...
- Computing the modulus:

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

The const concept (1)

- `const` variables cannot be changed:

```
const double p = 3;
p = 4; // ILLEGAL!! compiler error...
```

- `const` arguments (in functions) cannot be changed:

```
void myfunc (const Complex& c)
{ c.re = 0.2; /* ILLEGAL!! compiler error... */  }
```

- `const Complex` arguments can only call `const` member functions:

```
double myabs (const Complex& c)
{ return c.abs(); }   // ok because c.abs() is a const func.
```

The const concept (2)

- Without `const` in

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

  the compiler would not allow the `c.abs()` call in `myabs`

  ```
  double myabs (const Complex& c)
  { return c.abs(); }
  ```

  because `Complex::abs` is not a `const` member function
- `const` functions cannot change the object's state:

  ```
  void Complex::myfunc2 () const
  { re = 0.0; im = 0.5;  /* ILLEGAL!! compiler error... */ }
  You can only read data attributes and call \emp{const} functions
  ```

Overloaded operators

- C++ allows us to define `+ - * /` for arbitrary objects
- The meaning of `+` for `Complex` objects is defined in the function

  ```
  Complex operator+ (const Complex& a, const Complex& b); // a+b
  ```
- The compiler translates

  ```
  c = a + b;
  ```

  into

  ```
  c = operator+ (a, b);
  ```

  i.e., the overhead of a function call
- If the function call appears inside a loop, the compiler cannot apply aggressive optimization of the loop! That is why the next slide is important!

Inlined overloaded operators

- 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 compliete bodies, must be written in the .h (header) file

Consequence of inline

- Consider

  ```
  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
- More about this later

Friend functions (1)

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

  ```
  class Complex
  {
    ...
    friend Complex operator+ (const Complex& a, const Complex& b);
    ...
  };
  ```

  so it can read (and manipulate) the private data parts `re` and `im`:

  ```
  inline Complex operator+ (const Complex& a, const Complex& b)
  { return Complex (a.re + b.re, a.im + b.im); }
  ```

Friend functions (2)

- 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()); }
```

- `operator-`, `operator*` and `operator/` follow the same set up

Constructors

- Constructors have the same name as the class
- The declaration statement

  ```
  Complex q;
  ```

  calls the member function `Complex()`
- A possible implementation is

  ```
  Complex:: Complex ()  { re = im = 0.0; }
  ```

  meaning that declaring a complex number means making the number (0,0)
- Alternative:

  ```
  Complex:: Complex ()  {}
  ```

  Downside: no initialization of `re` and `im`

Constructor with arguments

- The declaration statement

  ```
  Complex q(-3, 1.4);
  ```

  calls the member function `Complex(double, double)`
- A possible implementation is

  ```
  Complex:: Complex (double re_, double im_)
  { re = re_;  im = im_; }
  ```

The assignment operator

- Writing

  ```
  a = b
  ```

  implies a call

  ```
  a.operator= (b)
  ```

  – this is the definition of assignment
- We implement `operator=` as a part of the class:

  ```
  Complex& Complex:: operator= (const Complex& c)
  {
    re = c.re;
    im = c.im;
    return *this;
  }
  ```
- If you forget to implement `operator=`, C++ will make one (this can be dangerous, see class MyVector!)

Copy constructor

- The statements

  ```
  Complex q = b;
  Complex q(b);
  ```

  makes a new object `q`, which becomes a copy of `b`
- Simple implementation in terms of the assignment:

  ```
  Complex:: Complex (const Complex& c)
  { *this = c; }
  ```

- `this` is a pointer to “this object”, `*this` is the present object, so `*this = c` means setting the present object equal to c, i.e., `this->operator= (c)`

Output function

- Output format of a complex number: (re,im), i.e., `(1.4,-1)`
- Desired user syntax:

```
std::cout << c;
any_ostream_object << c;
```

- The effect of << for a Complex object is defined in

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

- The input operator (`operator>>`) is more complicated (need to recognize parenthesis, comma, real numbers)

The multiplication operator

- First attempt:

```
inline Complex operator* (const Complex& a, const Complex& b)
{
  Complex h;     // Complex()
  h.re = a.re*b.re - a.im*b.im;
  h.im = a.im*b.re + a.re*b.im;
  return h;      // Complex(const Complex&)
}
```

- Alternative (avoiding the `h` variable):

```
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);
}
```

Inline constructors

- To inline the complete expression `a*b`, the constructors and `operator=` must also be inlined!

```
inline Complex:: Complex ()  { re = im = 0.0; }
inline Complex:: Complex (double re_, double im_)
{ ... }
inline Complex:: Complex (const Complex& c)
{ ... }
inline Complex:: operator= (const Complex& c)
{ ... }
```

Behind the curtain

```
// 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;
```

Benefit of inlined operators in loops

- Consider this potentially very long loop:

  ```
  Complex s, a;
  // initialize s and a...
  for (i = 1; i <= huge_n; i++) {
    s = s + a;
    a = a*3.0;
  }
  ```

- Without inlining `operator=`, `operator+`, `operator*`, and the constructors, we introduce several (how many??) function calls inside the loop, which prevent aggressive optimization by the compiler

The “real” name of C++ functions (1)

- C++ combines the name of the function and the type of arguments; this name is seen from the operating system
- This allows for using the same function name for different functions if only the arguments differ
- Examples (g++ generated names):

```
Complex:: Complex()
_ZN7ComplexC1Ev

Complex:: Complex(double re_, double im_)
_ZN7ComplexC1Edd

void Complex:: abs()
_ZN7Complex5absEv

void Complex:: write(ostream& o)
_ZN7Complex5writeERSo

Complex operator+ (const Complex& a, const Complex& b)
_ZplRK7ComplexS1_
```

The “real” name of C++ functions (2)

- You need to know the “real” name of a C++ function if you want to call it from C or Fortran
- You can see the “real” name by running `nm` on the object file:

```
unix> nm Complex.o
```

- It takes some effort to get used to reading the output from `nm`

Header file

- We divide the code of class Complex into a header file `Complex.h` and a file `Complex.cpp` with the body of the functions
- The header file contains the class declaration (data and functions), declaration of stand-alone functions, and *all inline functions with bodies*

```
#ifndef Complex_H
#define Complex_H

#include <...>

class Complex
{...};

std::ostream operator<< (std::ostream& o, const Complex& c);
std::istream operator>> (const Complex& c, std::istream& i);

// inline functions with bodies:
inline Complex operator+ (const Complex& a, const Complex& b)
{ return Complex(a.re + b.re, a.im + b.im); }
...
#endif
```

Other files

- `Complex.cpp` contains the bodies of the non-inline functions in class Complex
- Test application (with main program): any filename with extension .cpp, e.g., `main.cpp`
- `Complex.cpp` can be put in a library (say) `mylib.a` together with many other C++ classes
- `Complex.h` (and other header files for the library) are put in an include directory `$HOME/mysoft/include`
- Compile `main.cpp` and link with the library (you must notify the compiler about the include dir and where the library is)

```
g++ -I$HOME/mysoft/include -c main.cpp
g++ -o myexecutable -L$HOME/mysoft/lib main.o -lmylib -lm
```

Example: class MyVector

- Class `MyVector`: a vector
- Data: plain C array
- Functions: subscripting, change length, assignment to another vector, inner product with another vector, ...
- This examples demonstrates many aspects of C++ programming
- Note: this is mainly an educational example; for professional use one should use a ready-made vector class (`std::valarray` for instance)

MyVector functionality (1)

- Create vectors of a specified length:

  ```
  MyVector v(n);
  ```

- Create a vector with zero length:

  ```
  MyVector v;
  ```

- Redimension a vector to length n:

  ```
  v.redim(n);
  ```

- Create a vector as a copy of another vector w:

  ```
  MyVector v(w);
  ```

- Extract the length of the vector:

  ```
  const int n = v.size();
  ```

MyVector functionality (2)

- Extract an entry:

```
double e = v(i);
```

- Assign a number to an entry:

```
v(j) = e;
```

- Set two vectors equal to each other:

```
w = v;
```

- Take the inner product of two vectors:

```
double a = w.inner(v);
```

  or alternatively

```
a = inner(w,v);
```

MyVector functionality (3)

- Write a vector to the screen:

  ```
  v.print(std::cout);
  ```

- Arithmetic operations with vectors:

  ```
  // MyVector u, y, x; double a
  u = a*x + y;   // 'DAXPY' operation
  ```

- The proposed syntax is defined through functions in class `MyVector`
- Class `MyVector` holds both the data in the vector, the length of the vector, as well as a set of functions for operating on the vector data
- `MyVector` objects can be sent to Fortran/C functions:

  ```
  // v is MyVector
  call_my_F77_function (v.getPtr(), v.size(), ...)
  //                     array       length
  ```

The MyVector class

```
class MyVector
{
private:
  double* A;                   // vector entries (C-array)
  int     length;
  void    allocate (int n);    // allocate memory, length=n
  void    deallocate();        // free memory
public:
  MyVector ();                 // MyVector v;
  MyVector (int n);            // MyVector v(n);
  MyVector (const MyVector& w); // MyVector v(w);
 ~MyVector ();                 // clean up dynamic memory

  bool redim (int n);                         // v.redim(m);
  MyVector& operator= (const MyVector& w);// v = w;
  double  operator() (int i) const;           // a = v(i);
  double& operator() (int i);                 // v(i) = a;

  void print (std::ostream& o) const;         // v.print(cout);
  double inner (const MyVector& w) const; // a = v.inner(w);
  int size () const { return length; }        // n = v.size();
  double* getPtr () { return A; }  // send v.getPtr() to C/F77
};
```

Functions declared in the MyVector header file

- These appear after the class `MyVector` declaration:

```
// operators:
MyVector operator* (double a, const MyVector& v);    // u = a*v;
MyVector operator* (const MyVector& v, double a);    // u = v*a;
MyVector operator+ (const MyVector& a, const MyVector& b); // u = a
```

- The reason why these are declared outside the class, that the functions take two arguments: the left and right operand
- An alternative is to define the operators in the class, then the left operand is the class (`this` object) and the argument is the right operand
- We recommend to define binary operators outside the class with explicit left and right operand

Constructors (1)

- Constructors tell how we declare a variable of type `MyVector` and how this variable is initialized

```
MyVector v;   // declare a vector of length 0

// this actually means calling the function

MyVector::MyVector ()
{ A = NULL; length = 0; }
```

Constructors (2)

- More constructors:

```
MyVector v(n);  // declare a vector of length n
```

```
// means calling the function
```

```
MyVector::MyVector (int n)
{ allocate(n); }
```

```
void MyVector::allocate (int n)
{
  length = n;
  A = new double[n];  // create n doubles in memory
}
```

Destructor

- A `MyVector` object is created (dynamically) at run time, but must also be destroyed when it is no longer in use. The destructor specifies how to destroy the object:

```
MyVector::~MyVector ()
{
  deallocate();
}

// free dynamic memory:
void MyVector::deallocate ()
{
  delete [] A;
}
```

The assignment operator

- Set a vector equal to another vector:

```
// v and w are MyVector objects
v = w;
```

means calling

```
MyVector& MyVector::operator= (const MyVector& w)
// for setting v = w;
{
  redim (w.size()); // make v as long as w
  int i;
  for (i = 0; i < length; i++)  { // (C arrays start at 0)
    A[i] = w.A[i];
  }
  return *this;
}

// return of *this, i.e. a MyVector&, allows nested
// assignments:
u = v = u_vec = v_vec;
```

Redimensioning the length

- Change the length of an already allocated `MyVector` object:

```
v.redim(n);  // redimension v to length n
```

- Implementation:

```
bool MyVector::redim (int n)
{
  if (length == n)
    return false;  // no need to allocate anything
  else {
    if (A != NULL) {
      // "this" object has already allocated memory
      deallocate();
    }
    allocate(n);
    return true;   // the length was changed
  }
}
```

The copy constructor

- Create a new vector as a copy of an existing one:

  ```
  MyVector v(w);  // take a copy of w
  ```

  ```
  MyVector::MyVector (const MyVector& w)
  {
    allocate (w.size());  // "this" object gets w's length
    *this = w;            // call operator=
  }
  ```

- `this` is a pointer to the current (“this”) object, `*this` is the object itself

The const concept (1)

- `const` is a keyword indicating that a variable is not to be changed

```
const int m=5;  // not allowed to alter m
```

```
MyVector::MyVector (const MyVector& w)
// w cannot be altered inside this function
// & means passing w by _reference_
// only w's const member functions can be called
// (more about this later)
```

```
MyVector::MyVector (MyVector& w)
// w can be altered inside this function, the change
// is visible from the calling code
```

```
bool MyVector::redim (int n)
// a local _copy_ of n is taken, changing n inside redim
// is invisible from the calling code
```

The const concept (2)

- `const` member functions, e.g.,

  ```
  void MyVector::print (std::ostream& o) const
  ```

  means that the functions do not alter any data members of the class

Essential functionality: subscripting

- a and v are MyVector objects, want to set

  ```
  a(j) = v(i+1);
  ```

- The meaning of a(j) and v(i+1) is defined by

  ```
  inline double& MyVector::operator() (int i)
  {
    return A[i-1];
    // base index is 1 (not 0 as in C/C++)
  }
  ```

More about the subscription function

- Why return a `double` *reference*?

  ```
  double& MyVector::operator() (int i) { return A[i-1]; }
  ```

- Because the reference ("pointer") gives access to the memory location of `A[i-1]` so we can modify its contents (assign new value)
- Returning just a `double`,

  ```
  double MyVector::operator() (int i) { return A[i-1]; }
  ```

  gives access to a *copy* of the value of `A[i-1]`

Inlined subscripting

- Calling `operator()` for subscripting implies a function call
- Inline `operator()`: function body is copied to calling code, no overhead of function call
- Note: inline is just a hint to the compiler; there is no guarantee that the compiler really inlines the function
- With inline we hope that `a(j)` is as efficient as `a.A[j-1]`
- Note: inline functions and their bodies must be implemented in the .h (header) file!

More about inlining

- Consider this loop with vector arithmetics:

```
// given MyVector a(n), b(n), c(n);
for (int i = 1; i <= n; i++)
  c(i) = a(i)*b(i);
```

- Compiler inlining translates this to:

```
for (int i = 1; i <= n; i++)
  c.A[i-1] = a.A[i-1]*b.A[i-1];
// or perhaps
for (int i = 0; i < n; i++)
  c.A[i] = a.A[i]*b.A[i];
```

- More optimizations by a smart compiler:

```
double* ap = &a.A[0]; // start of a
double* bp = &b.A[0]; // start of b
double* cp = &c.A[0]; // start of c
for (int i = 0; i < n; i++)
  cp[i] = ap[i]*bp[i];         // pure C!
```

Add safety checks

- New version of the subscripting function:

```
inline double& MyVector::operator() (int i)
{
#ifdef SAFETY_CHECKS
  if (i < 1 || i > length)
    std::cerr <<   // or write to std::cout
    "MyVector::operator(), illegal index, i=" << i;
#endif

  return A[i-1];
}
```

- In case of a false ifdef, the C/C++ preprocessor physically removes the if-test before the compiler starts working
- To define safety checks:

```
g++ -DSAFETY_CHECKS -o prog prog.cpp
```

More about const (1)

Const member functions cannot alter the state of the object:

- Return access to a vector entry and allow the object to be changed:

  ```
  double& operator() (int i) { return A[i-1]; }
  ```

  ```
  a(j) = 3.14;  // example
  ```

- The same function with a const keyword can only be used for reading array values:

  ```
  double c = a(2);  // example
  ```

  ```
  double  operator() (int i) const
   { return A[i-1]; }
  ```

  (return `double`, i.e., a copy, not `double&`)

More about const (2)

- Only const member functions can be called from const objects:

```
void someFunc (const MyVector& v)
{
  v(3) = 4.2;  // compiler error, const operator() won't work
}

void someFunc (MyVector& v)
{
  v(3) = 4.2;  // ok, calls non-const operator()
}
```

Two simple functions: print and inner

```
void MyVector::print (std::ostream& o) const
{
  int i;
  for (i = 1; i <= length; i++)
    o << "(" << i << ")=" << (*this)(i) << '\n';
}

double a = v.inner(w);

double MyVector::inner (const MyVector& w) const
{
  int i; double sum = 0;
  for (i = 0; i < length; i++)
    sum += A[i]*w.A[i];
  // alternative:
  // for (i = 1; i <= length; i++) sum += (*this)(i)*w(i);
  return sum;
}
```

Operator overloading (1)

- We can easily define standard C++ output syntax also for our special `MyVector` objects:

```
// MyVector v
std::cout << v;
```

- This is implemented as

```
std::ostream& operator<< (std::ostream& o, const MyVector& v)
{
  v.print(o); return o;
}
```

- Why do we return a reference?

```
// must return std::ostream& for nested output operators:
std::cout << "some text..." << w;

// this is realized by these calls:
operator<< (std::cout, "some text...");
operator<< (std::cout, w);
```

Operator overloading (2)

- We can redefine the multiplication operator to mean the inner product of two vectors:

```
double a = v*w;  // example on attractive syntax
```

```
// global function:
double operator* (const MyVector& v, const MyVector& w)
{
   return v.inner(w);
}
```

Operator overloading (3)

```
// have some MyVector u, v, w; double a;

u = v + a*w;

// global function operator+
MyVector operator+ (const MyVector& a, const MyVector& b)
{
  MyVector tmp(a.size());
  for (int i=1; i<=a.size(); i++)
    tmp(i) = a(i) + b(i);
  return tmp;
}

// global function operator*
MyVector operator* (const MyVector& a, double r)
{
  MyVector tmp(a.size());
  for (int i=1; i<=a.size(); i++)
    tmp(i) = a(i)*r;
  return tmp;
}

// symmetric operator: r*a
MyVector operator* (double r, const MyVector& a)
{ return operator*(a,r); }
```

Limitations due to efficiency

- Consider this code segment:

```
MyVector u, x, y;  double a;
u = y + a*x;       // nice syntax!
```

- What happens behind the curtain?

```
MyVector temp1(n);
temp1 = operator* (a, x);
MyVector temp2(n);
temp2 = operator+ (y, temp1);
u.operator= (temp2);
```

$\Rightarrow$ Hidden allocation - undesired for large vectors

Alternative to operator overloading

- Avoid overloaded operators and their arithmetics for large objects (e.g., large arrays) if efficiency is crucial
- Write special function for compound expressions, e.g., `u = y + a*x` could be computed by

  ```
  u.daxpy (y, a, x)
  ```

  which could be implemented as

  ```
  void MyVector:: daxpy (const MyVector& y, double a,
                         const MyVector& x)
  {
     for (int i = 1; i <= length; i++)
       A[i] = y.A[i] + a*x.A[i];
  }
  ```

Another implementation of daxpy

- Having specialized expressions such as `a*x+y` as member functions, may “pollute” the vector class
- Here is a stand-alone function (outside the class):

```
void daxpy (MyVector& u, const MyVector& y,
            double a, const MyVector& x)
{
   for (int i = 1; i <= y.size(); i++)
     u(i) = a*x(i) + y(i);
}

// usage:
daxpy(u, y, a, x);
```

Yet another implementation of daxpy

- The result is returned:

```
MyVector daxpy (const MyVector& y, double a, const MyVector& x)
{
   MyVector r(y.size());  // result
   for (int i = 1; i <= y.size(); i++)
     r(i) = a*x(i) + y(i);
   return r;
}

// usage:
u = daxpy(y, a, x);
```

- What is the main problem wrt efficiency here?

Vectors of other entry types

- Class `MyVector` is a vector of doubles
- What about a vector of floats or ints?
- Copy and edit code...?
- No, this can be done automatically by use of *macros* or *templates*
- Templates is the recommended C++ approach

Macros for parameterized types (1)

- Substitute double by `Type`:

```
class MyVector(Type)
{
private:
  Type* A;
  int length;
public:
  ...
  Type& operator() (int i) { return A[i-1]; }
  ...
};
```

- Define `MyVector(Type)` through a macro:

```
#define concatenate(a,b) a ## b
#define MyVector(X) concatenate(MyVector_,X)
```

- Store this declaration in a file (say) `MyVector.h`
- The preprocessor translates `MyVector(double)` to `MyVector_double` before the code is compiled

Macros for parameterized types (2)

- Generate real C++ code in other files:

```
// in MyVector_double.h, define MyVector(double):
#define Type double
#include <MyVector.h>
#undef  Type

// MyVector_float.h, define MyVector(float):
#define Type float
#include <MyVector.h>
#undef  Type

// MyVector_int.h, define MyVector(int):
#define Type int
#include <MyVector.h>
#undef  Type
```

Templates

- Templates are the native C++ constructs for parameterizing parts of classes
- `MyVector.h`:

```
template<typename Type>
class MyVector
{
  Type* A;
  int length;
public:
  ...
  Type& operator() (int i) { return A[i-1]; }
  ...
};
```

- Declarations in user code:

```
MyVector<double> a(10);
MyVector<int> counters;
```

Subscripting in parameterized vectors

- Need a const and a non-const version of the subscripting operator:

```
      Type& operator()       { return A[i-1]; }
const Type& operator() const { return A[i-1]; }
```

- Notice that we return a const reference and not just

```
Type operator() const { return A[i-1]; }
```

- Why?
  returning Type means taking a copy of A[i-1], i.e., calling the copy constructor, which is very inefficient if Type is a large object (e.g. when we work with a vector of large grids)

Note

- We have used int for length of arrays, but size_t (an unsigned integer type) is more standard in C/C++:

```
double* A;
size_t n;  // length of A
```

About doing exercises

- We strongly recommend to go through the exercises on the next pages, unless you are an experienced C++ class programmer
- The step from one exercise to the next is made sufficiently small such that you don't get too many new details to fight with at the same time
- Take the opportunity to consult teachers in the computer lab; doing the exercises there with expert help is efficient knowledge building – towards the more demanding compulsory exercises and projects in this course

Exercise 7: Get started with classes (1)

- Make a small program with the following code:

```
class X
{
private:
  int i,j;
public:
  X(int i, int j);
  void print() const;
};

X::X(int i_, int j_)
{ i = i_;  j = j_; }

void X::print() const
{
  std::cout << "i=" << i << " j=" << j << '\n';
}
```

plus a main program testing class X:

```
X x(3,9);  x.print();
```

Exercise 7: Get started with classes (2)

- Compile and run
- How can you change the class such that the following code is legal:

```
X myx; myx.i=5; myx.j=10; myx.print();
```

Exercise 8: Work with .h and .cpp files (1)

- Consider the program from the previous exercise
- Place the class declaration in a header file `X.h`:

```
#ifndef X_H
#define X_H

#include <...>

class X
{
  ...
};

// inline functions:
...

#endif
```

Exercise 8: Work with .h and .cpp files (2)

- Implement the constructor(s) and print function in an `X.cpp` file:

```
#include <X.h>

X::X(int i_, int j_)
{ ... }

void X::print()
{ ... }
```

- Place the main function in `main.cpp`

Exercise 8: Work with .h and .cpp files (3)

- Compile the two .cpp files:

```
g++ -I. -O2 -c X.cpp main.cpp
```

- Link the files with the libraries:

```
g++ -o Xprog X.o main.o -lm
```

Exercise 9: Represent a function as a class

- In exercise 3 we implemented a C/C++ function `userfunc` that used a function pointer for representing a user-defined function `f`
- As an alternative, `f` may be realized as a class,

```
class F : public FunctionClass
{
  double a;  // parameter in the function expression
public:
  F(double a_) { a = a_; }
  virtual double operator() (double x) const { return a*x; }
};
```

- The `trapezoidal` function now has the signature

```
double trapezoidal(FunctionClass& f, double a, double b, int n)
```

  Implement this function and verify that it works

Exercise 10: Implement class MyVector

- Type in the code of class `MyVector`
- Collect the class declaration and inline functions in `MyVector.h`

```
#ifndef MyVector_H
#define MyVector_H

class MyVector
{ ... };

inline double& operator() (int i)
{ ... }
...
#endif
```

- Write the bodies of the member functions in `MyVector.cpp`

```
#include <MyVector.h>
// other includes...

MyVector::MyVector () { A = NULL; length = 0; }
...
```

- Make a main program for testing: `main.cpp`

Exercise 11: DAXPY (1)

- The mathematical vector operation

$$u \leftarrow ax + y,$$

  where a is scalar and x and y are vectors, is often referred to as a DAXPY operation, because DAXPY is the Fortran subroutine name for this operation in the standardized BLAS1 library

- Make a C++ function

```
void daxpy (MyVector& u, double a, const MyVector& x,
            const MyVector& y)
{ ... }
```

  performing a loop over the array entries for computing u

Exercise 11: DAXPY (2)

- Make a C++ function

  ```
  void daxpy_op (MyVector& u, double a, const MyVector& x,
                 const MyVector& y)
  {
    u = a*x + y;
  }
  ```

  using overloaded operators in the `MyVector` class
- Compare the efficiency of the two functions (hint: run 10^p daxpy operations with vectors of length 10^q, e.g., with $p = 4$ and $q = 6$)
- Compare the efficiency with a tailored Fortran 77 subroutine

Exercise 12: Communicate with C

- Say you want to send a `MyVector` object to a Fortran or C routine
- Fortran and C understand pointers only: `double*`
- `MyVector` has an underlying pointer, but it is private
- How can class `MyVector` be extended to allow for communication with Fortran and C?
- Test the procedure by including a C function in the main program, e.g.,

```
void printvec(double* a, int n)
{
  int i;
  for (i=0; i<n; i++) { printf("entry %d = %g\n",i,a[i]); }
}
```

Exercise 13: Communicate with Fortran

- Consider the previous exercise, but now with a `printvec` routine written in Fortran 77:

```
      SUBROUTINE PRINTVEC77(A,N)
      INTEGER N,I
      REAL*8 A(N)
      DO 10 I=1,N
        WRITE(*,*) 'A(',I,')=',A(I)
 10   CONTINUE
      RETURN
      END
```

- C/C++ wrapper function (i.e., the F77 routine as viewed from C/C++):

```
extern "C" {
  void printvec77_ (double* a, const int& n);
}
```

- Compile and link the F77 and C++ files (sometimes special Fortran libraries like libF77.a must be linked)

Exercise 14: Extend MyVector (1)

- Extend class `MyVector` with a `scan` function
- `scan` reads an ASCII file with values of the vector entries
- The file format can be like this:

  ```
  n
  v1
  v2
  v3
  ...
  ```

  where n is the number of entries and v1, v2, and so on are the values of the vector entries
- Compile, link and test the code

Exercise 14: Extend MyVector (2)

- Make an alternative to `scan`:

  ```
  // global function:
  istream& operator>> (istream& i, MyVector& v)
  { ... }
  ```

  for reading the vector from some istream medium (test it with a file and standard input)

A more flexible array type

- Class `MyVector` is a one-dimensional array
- Extension: `MyArray`
- Basic ideas:
  1. storage as MyVector, i.e., a long C array
  2. use templates (entry type is `T`)
  3. offer multi-index subscripting:

     ```
     T& operator() (int i, int j);
     T& operator() (int i, int j, int k);
     ```
- `MyArray` may be sufficiently flexible for numerical simulation

Class MyArray

```
template <class T>
class MyArray
{
protected:
  T*      A;                          // vector entries (C-array)
  int     length;
public:
  MyArray ();                         // MyArray<T> v;
  MyArray (int n);                    // MyArray<T> v(n);
  MyArray (const MyArray& w);         // MyArray<T> v(w);
 ~MyArray ();                         // clean up dynamic memory

  int redim (int n);                             // v.redim(m);
  int size () const { return length; }           // n = v.size();

  MyArray& operator= (const MyArray& w);         // v = w;

        T  operator() (int i) const;             // a = v(i);
  const T& operator() (int i);                   // v(i) = a;

        T  operator() (int i, int j) const;      // a = v(p,q);
  const T& operator() (int i, int j);            // v(p,q) = a;

  void print (ostream& o) const;                 // v.print(cout);
};
```

The interior of MyArray

- The code is close to class `MyVector`
- The subscripting is more complicated
- (i,j) tuples must be transformed to a single address in a long vector
- Read the source code for details: src/C++/Wave2D/MyArray.h and src/C++/Wave2D/MyArray.cpp

Exercise 15: 3D MyArray

- `MyArray` works for one and two indices
- Extend `MyArray` such that it handles three indices as well:

  ```
  T& operator() (int i, int j, int k);
  ```

  A few other functions must be supplied

Memory-critical applications

- C++ gives you the possibility to have full control of dynamic memory, yet with a simple and user-friendly syntax
- Suppose you want to keep track of the memory usage
- Make a class `MemBoss` that manages a large chunk of memory
- Use `MemBoss` instead of plain new/delete for allocation and deallocation of memory

Outline of class MemBoss (1)

```
class MemBoss
{
private:
  char* chunk;  // the memory segment to be managed
  size_t size;  // size of chunk in bytes
  size_t used;  // no of bytes used
  std::list<char*>  allocated_ptrs;  // allocated segments
  std::list<size_t> allocated_size;  // size of each segment
public:
  MemBoss(int chunksize)
    { size=chunksize; chunk = new char[size]; used=0; }
 ~MemBoss() { delete [] chunk; }
  void* allocate(size_t nbytes)
    { char* p = chunk+used;
      allocated_ptrs.insert_front(p);
      allocated_size.insert_front(nbytes);
      used += nbytes;
      return (void*) p;
    }
  void deallocate(void* p); // more complicated
  void printMemoryUsage(std::ostream& o);
};
```

Outline of class MemBoss (2)

```
// memory is a global object:
MemBoss memory(500000000); // 500 Mb

// redefine new and delete:
void* operator new (size_t t)
{ return memory.allocate(t); }

void operator delete (void* v)
{ memory.deallocate(v); }

// any new and delete in your program will work with
// the new memory class!!
```

Local new and delete in a class

- A class can manage its own memory
- Example: list of 2D/3D points can allocate new points from a common chunk of memory
- Implement the member functions `operator new`, `operator delete`
- Any new or delete action regarding an object of this class will use the tailored new/delete operator

Lessons learned

- It is easy to use class `MyVector`
- Lots of details visible in C and Fortran 77 codes are hidden inside the class
- It is not easy to write class `MyVector`
- Thus: rely on ready-made classes in C++ libraries unless you really want to write develop your own code and you know what are doing

C++ programming is effective when you build your own high-level classes out of well-tested lower-level classes

Don't use MyVector - use a library

- Class `MyVector` has only one index (one-dim. array)
- Class `MyArray` (comes with this course) is a better alternative for numerical computing
- Even better: use a professional library
- One possible choice is Blitz++
  http://www.oonumerics.org/blitz/
  (works well under GNU's g++ compiler)

C++ (array) libraries

- Blitz++: high-performance C++ array library
- A++/P++: serial and parallel array library
- Overture: PDE (finite difference/volume) on top of A++/P++
- MV++: template-based C++ array library
- MTL: extension of STL to matrix computations
- PETSc: parallel array and linear solver library (object-oriented programming in C)
- Kaskade: PDE (finite element) solver library
- UG: PDE solver library (in C)
- Diffpack: PDE (finite element) solver library w/arrays

The Standard Template Library

- STL = Standard Template Library
- STL comes with all C++ compilers
- Contains vectors, lists, queues, stacks, hash-like data structures, etc.
- Contains generic algorithms (functions) operating on the various data structures
- STL is a good example on C++ programming with templates, so called generic programming, an alternative to OOP
- In generic programming, data structures and algorithms are separated (algorithms are stand-alone functions, not member functions in data structures as in OOP)

Working with STL

STL has three basic ingredients:

- Containers (vector, list, ...)
- Iterators (generalized pointers to elements in containers)
- Algorithms (copy, sort, find, ...)

Each container has an associated iterator, and algorithms work on any container through manipulation with iterators

Container: vector

```
#include <vector>

std::vector<double> v(10, 3.2 /* default value */);
v[9] = 1001;  // indexing, array starts at 0
const int n = v.size();
for (int j=0; j<n; j++)
  std::cout << v[j] << " ";   // only one index is possible

// vector of user-defined objects:
class MyClass { ... };
std::vector<MyClass> w(n);
```

Container: string

```
#include <string>

std::string s1 = "some string";
std::string s2;
s2 = s1 + " with more words";
std::string s3;
s3 = s2.substr(12 /*start index*/, 16 /*length*/);
printf("s1=%s, s3=%s\n", s1.c_str(), s3.c_str());
// std::string's c_str() returns a char* C string
```

STL lists

- List:

```
#include <list>

std::list<std::string> slist;
slist.push_front("string 1");  // add at beginning
slist.push_front("string 2");
slist.push_back("string 3");   // add at end

slist.clear();                 // erase the whole list

// slist<std::string>::iterator p;  // list position
slist.erase(p);                // erase element at p
slist.insert(p, "somestr");    // insert before p
```

Iterators (1)

- Iterators replace "for-loops" over the elements in a container
- Here is a typical loop over a vector

  ```
  // have some std::vector<T> v;
  std::vector<T>::iterator i;
  for (i=v.begin(); i!=v.end(); ++i)
    std::cout << *i << " ";
  ```

  (i is here actually a `T*` pointer)
- ...and a similar loop over a list:

  ```
  std::list<std::string>::iterator s;
  for (s=slist.begin(); s!=slist.end(); ++s)
    std::cout << *s << '\n';
  ```

  (s is here more complicated than a pointer)

Iterators (2)

- All STL data structures are traversed in this manner,

```
some_iterator s;
// given some_object to traverse:
for (s=some_object.begin(); s!=some_object.end(); ++s)  {
  // process *s
}
```

- The user's code/class must offer `begin`, `end`, `operator++`, and `operator*` (dereferencing)

Algorithms

- Copy:

```
std::vector<T> v;
std::list<T> l;
...
// if l is at least as long as v:
std::copy(v.begin(), v.end(), l.begin());
// works when l is empty:
std::copy(v.begin(), v.end(), std::back_inserter(l));
```

- Possible implementation of copy:

```
template<class In, class Out>
Out copy (In first, In last, Out result)
{
  // first, last and result are iterators
  while (first != last) {
    *result = *first;   // copy current element
    result++; first++;  // move to next element
  }
  // or a more compact version:
  // while (first != last) *result++ = *first++;
  return result;
}
```

Specializing algorithms

- Note that `copy` can copy any sequence (vector, list, ...)
- Similar, but specialized, implementation for vectors of `doubles` (just for illustration):

```
double* copy(double* first, double* last,
             double* result)
{
  for (double* p = first; p != last; p++, result++) {
    *p = *result;
  }
  // or
  while (first != last) {
    *result = *first;
    result++; first++;
  }
  return result;
}
```

Some other algorithms

- `find`: find first occurence of an element
- `count`: count occurences of an element
- `sort`: sort elements
- `merge`: merge sorted sequences
- `replace`: replace element with new value

Exercise 16: List of points (1)

- Make a class for 2D points

```
class Point2D
{
  double x, y;  // coordinates
public:
  Point2D();
  Point2D(double x_, double y_);
  Point2D(const Point2D& p);
  void set(double x_, double y_);
  void get(double& x_, double& y) const;
  double getX() const;
  double getY() const;
  void scan (istream& is);  // read from e.g. file
  void print(ostream& os);
};
istream& operator>> (istream& is, Point2D& p);
ostream& operator<< (ostream& os, const Point2D& p);
```

Exercise 16: List of points (2)

- Make a list of 2D points:

  ```
  std::list<Point2D> plist;
  ```

- Fill the list with points
- Call the STL algorithm sort to sort the list of points (find some electronic STL documentation)
- Print the list using a for-loop and an iterator

STL and numerical computing

- std::valarray is considered superior to std::vector for numerical computing
- valarray does not support multi-index arrays
- Can use valarray as internal storage for a new matrix or multi-index array type
- Supports arithmetics on vectors

```
#include <valarray>

std::valarray<double> u1(7), u2(7), u3(7);
u1[6]=4;
u3 = 3.2*u1 + u2;

// no begin(), end() for valarray
for (j=0; j<7; j++)
  std::cout << u3[j] << " ";
```

STL and the future

- Many attractive programming ideas in STL
- For numerical computing one is normally better off with other libraries than STL and its valarray
- Template (generic) programming is more efficient than OOP since the code is fixed at compile time
- The template technology enables very efficient code (e.g. automatic loop unrolling controlled by a library)
- Blitz++: creative use of templates to optimize array operations
- MTL: extension of STL to matrix computations (promising!)
- Still portability problems with templates

Efficiency in the large

- What is efficiency?
- *Human efficiency* is most important for programmers
- *Computational efficiency* is most important for program users

Smith, Bjorstad and Gropp

"In the training of programming for scientific computation the emphasis has historically been on squeezing out every drop of floating point performance for a given algorithm. ...... This practice, however, leads to highly tuned racecarlike software codes: delicate, easily broken and difficult to maintain, but capable of outperforming more user-friendly family cars."

Premature optimization

- "Premature optimization is the root of all evil" (Donald Knuth)
- F77 programmers tend to dive into implementation and think about efficiency in every statement
- "80-20" rule: "80" percent of the CPU time is spent in "20" percent of the code
- Common: only some small loops are responsible for the vast portion of the CPU time
- C++ and F90 force us to focus more on design

Don't think too much about efficiency before you have a thoroughly debugged and verified program!

Some rules

- Avoid lists, sets etc, when arrays can be used without too much waste of memory
- Avoid calling small virtual functions in the innermost loop (i.e., avoid object-oriented programming in the innermost loop)
- Implement a working code with emphasis on design for extensions, maintenance, etc.
- Analyze the efficiency with a tool (profiler) to predict the CPU-intensive parts
- Attack the CPU-intensive parts after the program is verified

Some more rules

- Heavy computation with small objects might be inefficient, e.g., vector of class complex objects
- Virtual functions: cannot be inlined, overhead in call
- Avoid small virtual functions (unless they end up in more than (say) 5 multiplications)
- Save object-oriented constructs and virtual functions for the program management part
- Use C/F77-style in low level CPU-intensive code (for-loops working on plain C arrays)
- Reduce pointer-to-pointer-to....-pointer links inside for-loops

And even some more rules

- Attractive matrix-vector syntax like `y = b - A*x` has normally significant overhead compared to a tailored function with one loop
- Avoid implicit type conversion
  (use the explicit keyword when declaring constructors)
- Never return (copy) a large object from a function
  (normally, this implies hidden allocation)

Examples on inefficient constructions

- Code:

```
MyVector somefunc(MyVector v)  // copy!
{
  MyVector r;
  // compute with v and r
  return r;                    // copy!
}
```

  $\Rightarrow$ two unnecessary copies of possibly large `MyVector` arrays!

- More efficient code:

```
void somefunc(const MyVector& v, MyVector& r)
{
  // compute with v and r
}
```

- Alternative: use vectors with built-in reference counting such that `r=u` is just a copy of a reference, not the complete data structure

Hidden inefficiency

- Failure to define a copy constructor

```
class MyVector
{
  double* A; int length;
public:
  // no copy constructor  MyVector(const MyVector&)
};
```

C++ automatically generates a copy constructor with copy of data item by data item:

```
MyVector::MyVector(const MyVector& v)
{
  A = v.A;  length = v.length;
}
```

Why is this bad? What type of run-time failure can you think of? (Hint: what happens in the destructor of w if you created w by MyVector(u)?)

C++ versus Fortran 77

- F77 is normally hard to beat
- With careful programming, C++ can come close
- Some special template techniques can even beat F77 (significantly)
- C++ often competes well with F77 in complicated codes
- F77 might be considerably faster than C++ when running through large arrays (e.g., explicit finite difference schemes)
- If C++ is not fast enough: port critical loops to F77

Remark: F90 is also often significantly slower than F77

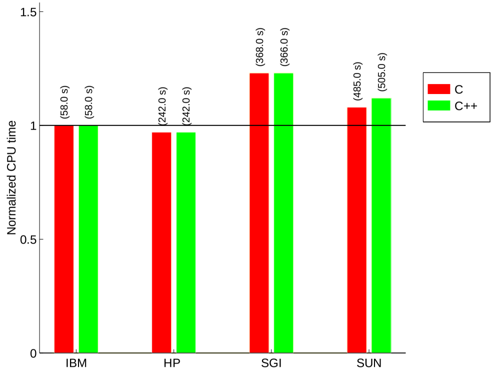
Efficiency tests

- Diffpack/C++ vs. C vs. FORTRAN 77
- Low-level linear algebra (BLAS)
- Full PDE simulators

Joint work with Cass Miller's group at the Univ. of North Carolina at Chapel Hill

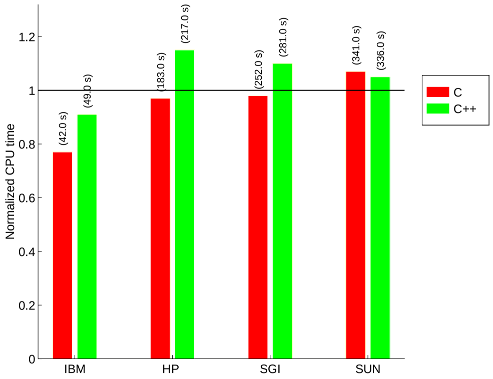
Test: DAXPY

$$y \leftarrow ax + y$$



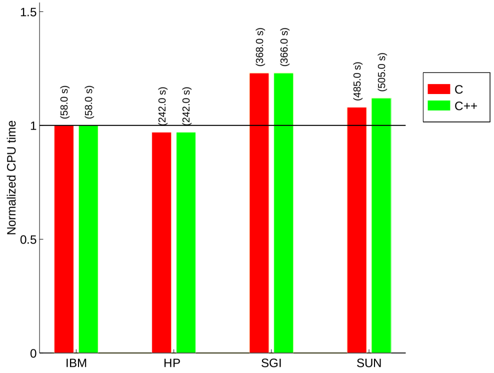
Test: DDOT

$$s \leftarrow (u, v)$$



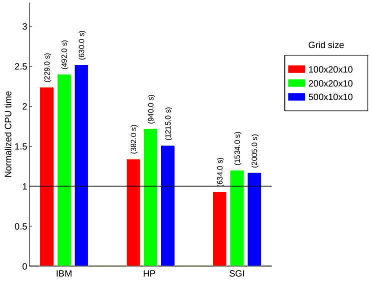
Test: DGEMV

$$x \leftarrow Ay$$



Test: linear convection-diffusion

- Model:
$$\frac{\partial u}{\partial t} + \vec{v} \cdot \nabla u = k \nabla^2 u \text{ in 3D}$$
- Tests iterative solution (BiCGStab w/Jacobi prec.) of linear systems

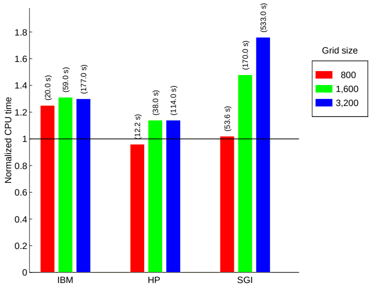


Test: Richards' equation

- Model:

$$\frac{\partial \theta}{\partial t} + S_s S \frac{\partial \psi}{\partial t} = \frac{\partial}{\partial z}\left[K\left(\frac{\partial \psi}{\partial z} + 1\right)\right] \quad \text{in 1D}$$

- Tests FE assembly w/advanced constitutive relations

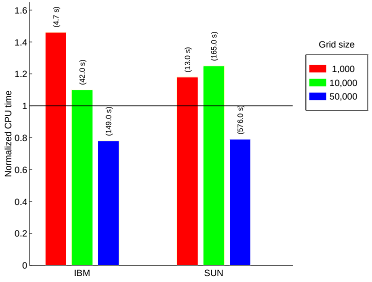


Test: convection-diffusion-reaction

- Model:

$$\text{convection-diffusion} + \alpha u^2 \text{ in 1D}$$

  by Newton's method
- Tests FE assembly



Strong sides of C++

- Rich language (over 60 keywords)
- Good balance between OO support and numerical efficiency
- Very widespread for non-numerical software
- Careful programming can give efficiency close to that of F77
- Well suited for large projects
- Compatibility with C
- The compiler finds many errors
- Good software development tools
- Good standard library for strings, lists, arrays, etc. (STL)

Weak sides of C++

- Lacks good standard libraries for numerics (STL is too primitive)
- Many possibilities for inefficient code
- Many ways of doing the same things (programming standard is important!)
- Supports ugly constructs
- The language is under development, which causes portability problems

An ideal scientific computing environment

Write numerical codes close to the mathematics and numerical algorithms!

- Write very high-level code for rapid prototyping
- Write lower-level code to control details
  – when needed
- Get efficiency as optimized Fortran 77 code

Recall: high-level codes are easier to read, maintain, modify and extend!

Application example

- Finite difference PDE solver for, e.g.,

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(H(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(H(x,y)\frac{\partial u}{\partial y}\right)$$

  on a rectangular grid
- Explicit 2nd-order finite difference scheme:

$$u_{i,j}^{\ell+1} = G(u_{i,j}^{\ell-1}, u_{i,j}^{\ell}, u_{i-1,j}^{\ell}, u_{i+1,j}^{\ell}, u_{i,j-1}^{\ell}, u_{i,j+1}^{\ell})$$

- Abstractions: 2D arrays, grid, scalar fields, FD operators, ...

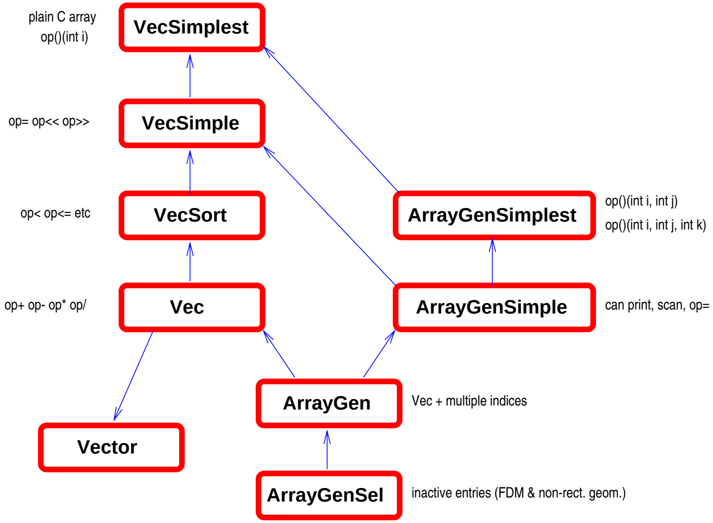
Typical features of a modern library

Layered design of objects:

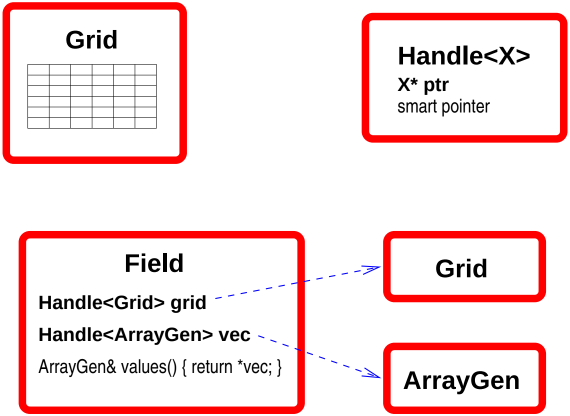
- smart pointers
  (automatic memory handling)
- arrays
- finite difference grid
- scalar field over the grid

Example here: Diffpack (www.diffpack.com)

Array classes



PDE classes


Grid
Handle<X>
X* ptr
smart pointer
Field
Handle<Grid> grid
Handle<ArrayGen> vec
ArrayGen& values() { return *vec; }
Grid
ArrayGen

Why all these classes?

- A simple scalar wave equation solver is easy to implement with just plain Fortran/C arrays
- The grid/field abstractions pay off in more complicated applications
- This application is (probably) a “worst case” example of using object-oriented programming; seemingly lots of overhead
- So: How much efficiency is lost?

Coding a scheme

- Traverse field values:

```
#define U(i,j) u.values()(i,j)

for (i=1; i<=in; i++) {
  for (j=1; j<=jn; j++) {
    U(i,j) = ... + U(i-1,j) + ...
```

- U(i,j) is a set of nested function calls:

```
  u.values() calls Handle<ArrayGen>::operator*
  (i,j)      calls ArrayGen::operator()
  operator() returns A[nx*(i-1)+j] with A[] in a
             virtual base class (i.e. ptr->A[])
```

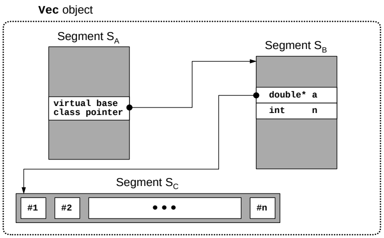
  $\Rightarrow$ 3 nested function calls

- All functions are inline, but does the compiler really see that the loop just operates on a 1D C array?
- The scheme is 1 page of code and consumes 90 percent of the CPU time of a wave simulator

Virtual base class

```
class Vec : public virtual VecSimplest
{
   public:
      Vec  (int length);
      ~Vec ();
      •••
}
```

```
class VecSimplest
{
   protected:
      double* a;
      int     n;
   public:
      VecSimplest  (int length);
      ~VecSimplest ();
      •••
}
```



Speeding up the code (1)

- Help the compiler; extract the array

  ```
  ArrayGen& U = u.values();

  for (i=1; i<=in; i++)
    for (j=1; j<=jn; j++)
      U(i,j) = ... + U(i-1,j) + ...
  ```

  $\Rightarrow$ one function call to inline operator()
- Almost 30 percent reduction in CPU time

Speeding up the code (2)

- Help the compiler; work with a plain C array

```
#ifdef SAFE_CODE
ArrayGen& U = u.values();
for (i=1; i<=in; i++)
  for (j=1; j<=jn; j++)
    U(i,j) = ... + U(i-1,j) + ...
#else

double* U = u.values().getUnderlyingCarray();

const int i0 = -nx-1;
for (i=1; i<=in; i++) {
  for (j=1; j<=jn; j++) {
    ic = j*nx + i + i0
    iw = ic - 1
    ...
    U[ic] = ... + U[iw] + ...
  }
}
#endif
```

- Almost 80 percent reduction in CPU time!

Speeding up the code (3)

- Do the intensive array work in F77

```
#ifdef SAFE_CODE
ArrayGen& U = u.values();
for (i=1; i<=in; i++) {
  for (j=1; j<=jn; j++) {
    U(i,j) = ... + U(i-1,j) + ...
#else

double* U = u.values().getUnderlyingCarray();

scheme77_ (U, ...);  // Fortran subroutine

#endif
```

- 65 percent reduction in CPU time (Fujitsu f95)
- 73 percent reduction in CPU time (GNU g77)

Speeding up the code (4)

- Lend arrays to a fast C++ array library
- Example: Blitz++
- Wrap a Blitz++ subscripting interface

```
double* ua = u.values().getUnderlyingCarray();

blitz::Array<real, 2> U(ua,
     blitz::shape(nx,ny),
     blitz::neverDeleteData,
     blitz::FortranArray<2>());

for (i=1; i<=in; i++)
  for (j=1; j<=jn; j++)
    U(i,j) = ... + U(i-1,j) + ...
```

- Note: same application code as for our ArrayGen object
- 62 percent reduction in CPU time

A note about compilers

- Main computational work in nested loops

```
for (i=1; i<=in; i++)
  for (j=1; j<=jn; j++)
    U(i,j) = ... + U(i-1,j) + ...
```

- GNU and Fujitsu compilers have been tested with numerous options (-O1, -O2, -O3, -ffast-math -funroll-loops)
- All options run at approx the same speed (!)
- Optimal optimization of the loop (?)

Lessons learned

- Exaggerated use of objects instead of plain arrays slows down the code
- The inner intensive loops can be recoded in C or F77 to get optimal performance
- The recoding is simple and quick human work
- The original, safe code is available for debugging
- The grid/field abstractions are very convenient for all work outside the intensive loops
  (large parts of the total code!)
- This was probably a worst case scenario

$\Rightarrow$ Program at a high level, migrate slow code to F77 or C. This is trivial in the Diffpack environment.

Object-based vs. -oriented programming

- Class `MyVector` is an example on programming with objects, often referred to as object-based programming (OBP)
- Object-oriented programming (OOP) is an extension of OBP
- OOP works with classes related to each other in a hierarchy
- OOP is best explained through an example

An OOP example: ODE solvers

- Topic: a small library for solving ordinary differential equations (ODEs)
$$\frac{dy_i}{dt} = f_i(y_1, \ldots, y_n, t), \quad y_i(0) = y_i^0,$$
for $i = 1, \ldots, n$
- Demonstrates OO design for a simple problem
- Introduces the basic OOP concepts in C++
- Principles are generic and apply to advanced numerics

ODE problems and methods

- Some vector $y_i(t)$ fulfills a 1st-order differential equation $dy_i/dt = f_i(y, t)$, where f_i is a vector
- Such mathematical models arise in physics, biology, chemestry, statistics, medicine, finance, ...
- Typical numerical solution method:
    1. start with some initial state y(0)
    2. at discrete points of time: compute new y(t) based on previously calcluated y values
- The simplest method (Forward Euler scheme):

$$y_i(t + \Delta t) = y_i(t) + \Delta t f_i(y(t), t)$$

  where Δt is a small time interval

Our problem framework

- There are numerous numerical solution methods for ODEs
- We want to
  1. implement a problem (i.e. f(y,t))
  2. easily access a range of solution methods
- A range of different problems (ODEs) must be easily combined with a range of solution methods

Design of a traditional F77 library

- Subroutines implementing various methods, e.g.

  ```
  SUBROUTINE RK4(Y,T,F,WORK1,N,TSTEP,TOL1,TOL2,...)
  ```

  for a 4th-order Runge-Kutta algorithm
- Y is the current solution (a vector)
- T is time
- F is a function defining the f values
- WORK1 is a scratch array
- N is the length of Y
- TSTEP is the time step (dt)
- TOL1, TOL2 are various parameters needed in the algorithm

User-given information

- Think of an ODE with lots of parameters C1, C2, ...
- Function F (user-given) defining f(y,t):

```
SUBROUTINE MYF(FVEC,Y,T,C1,C2,C3,C4,C5)
```

- Problem: MYF is to be called from a general RK4 routine; it does not know about the problem-dependent parameters C1, C2, C3, ...

```
CALL F(FVEC,Y,T)
```

- Problem-dependent parameters in MYF must be transferred through COMMON blocks

```
SUBROUTINE MYF(FVEC,Y,T)
...
COMMON /MYFPRMS/ C1, C2, C3, ...
...
```

Improvements

- Internal scratch arrays needed in algorithms should not be visible for the end-user
- All parameters needed in an algorithm must be specified as arguments; the user should only need to set a small set of parameters at run time, relying on sensible default values for the rest
- Ideally, the calling interface to all the ODE solvers is identical
- Problem-specific parameters in the definition of the equations to be solved should not need to be global variables
- All these goals can easily be reached by using C++ and object-oriented programming

The basic ideas of OO programming

- Create a base class with a generic interface
- Let the interface consist of virtual functions
- A hierarchy of subclasses implements various versions of the base class
- Work with a base class pointer only througout the code; C++ automatically calls the right (subclass) version of a virtual function
- This is the principle of object-oriented programming

The ODESolver hierarchy

- Create a base class for all ODE solver algorithms:

```
class ODESolver
{
  // common data needed in all ODE solvers
public:
  ...
  // advance the solution one step according to the alg.:
  virtual void advance(MyArray<double>& y,
                       double t, double dt);
};
```

- Implement special ODE algorithms as subclasses:

```
class ForwardEuler : public ODESolver
{
  ...
public:
  // the simple Forward Euler scheme:
  virtual void advance(MyArray<double>& y, double t, double dt);
};

class RungeKutta4 : public ODESolver
{ ... };
```

Working with ODE solvers

- Let all parts of the code work with ODE solvers through the common base class interface:

  ```
  void somefunc(ODESolver& solver, ...)
  {
    ...
    solver.advance(y,t,dt);
    ...
  }
  ```

  Here, solver will call the right algorithm, i.e., the `advance` function in the subclass object that solver actually refers to
- Result: All details of a specific ODE algorithm are hidden; we just work with a generic ODE solver

Some initial problem-dependent code is needed

- At one place in the code we must create the right subclass object:

  ```
  ODESolver* s= new RungeKutta4(...);
  ```

  ```
  // from now on s is sent away as a general ODESolver,
  // C++ remembers that the object is actually a Runge-Kutta
  // solver of 4th order:
  somefunc(*s, ...);
  ```

- Creation of specific classes in a hierarchy often takes place in what is called a *factory function*

User-provided functions

- The user needs to provide a function defining the equations
- This function is conveniently implemented as a class, i.e. in a problem class:

```
class Oscillator
{
  double C1, C2, C3, C4;
public:
  int  size() { return 2; }  // 2 ODEs to be solved
  void equation(MyArray<double>& f,
                const MyArray<double>& y, double t);
  void scan();  // read C1, C2, ... from some input
};
```

- Any ODESolver can now call the equation function of the problem class to evaluate the f vector

Generalizing

- Problem: The problem class type (Oscillator) cannot be visible from an ODESolver (if so, the solver has hardcoded the name of the problem being solved!)
- Remedy: all problem classes are subclasses of a common base class with a generic interface to ODE problems

Base class for all problems

- Define

```
class ODEProblem
{
  // common data for all ODE problems
public:
  virtual int  size();
  virtual void equation(MyArray<double>& f,
                        const MyArray<double>& y, double t);
  virtual void scan();
};
```

- Our special problem is implemented as a subclass:

```
class Oscillator : public ODEProblem
{
  ...
public:
  virtual int  size() { return 2; }
  virtual void equation(MyArray<double>& f,
                        const MyArray<double>& y, double t);
  virtual void scan();  // read C1, C2, ...
};
```

Implementing class Oscillator (1)

- ODE model:

$$\ddot{y} + c_1(\dot{y} + c_2\dot{y}|\dot{y}|) + c_3(y + c_4y^3) = \sin\omega t$$

Rewritten as a 1st order system (advantageous when applying numerical schemes):

$$\begin{aligned}\dot{y}_1 &= y_2 \equiv f_1\\ \dot{y}_2 &= -c_1(y_2 + c_2y_2|y_2|) - c_3(y_1 + c_4y_1^3) + \sin\omega t \equiv f_2\end{aligned}$$

Implementing class Oscillator (2)

```
class Oscillator : public ODEProblem
{
protected:
  real c1,c2,c3,c4,omega;  // problem dependent paramters
public:
  Oscillator () {}

  // here goes our special ODE:
  virtual void equation (MyArray<double>& f,
                         const MyArray<double>& y, real t);

  virtual int  size () { return 2; }  // 2x2 system of ODEs
  virtual void scan ();
  virtual void print (Os os);
};

void Oscillator::equation (MyArray<double>& f,
                           const MyArray<double>& y, real t)
{
  f(1) = y(2);
  f(2) = -c1*(y(2)+c2*y(2)*abs(y(2))) - c3*(y(1)+c4*pow3(y(1)))
         + sin(omega*t);
}
```

ODESolvers work with ODEProblems

- All ODE solvers need to access a problem class:

```
class ODESolver
{
  ODEProblem* problem;
  ...
};

// in an advance function of a subclass:
problem->equation (f, y, t);
```

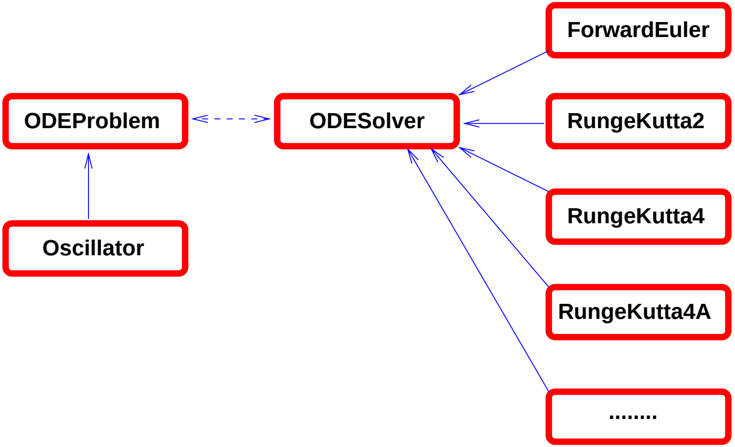
- Since equation is a virtual function, C++ will automatically call the equation function of our current problem class

Initially we need to make specific objects

```
ODEProblem* p = new Oscillator(...);
ODESolver* s = new RungeKutta4(..., p, ...);
somefunc(*s, ...);
```

From now on our program can work with a generic ODE solver and a generic problem

The class design



Solid arrows: inheritance (“is-a” relationship)
Dashed arrows: pointers (“has-a” relationship)

Functions as arguments to functions (1)

- In C: functions can be sent as argument to functions via function pointers

  ```
  typedef double (*funcptr)(double x, int i);
  ```

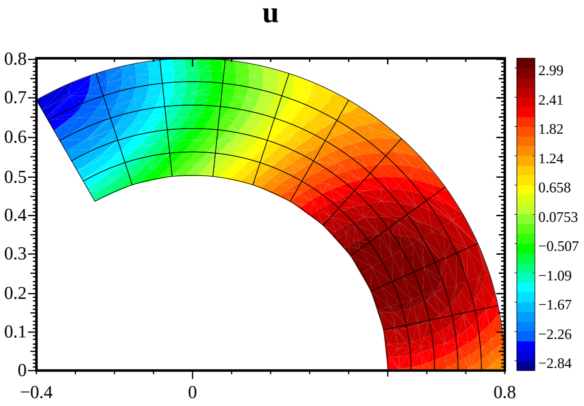
- In C++ one applies function objects (or functors)
- Idea: the function pointer is replaced by a base-class pointer/ref., and the function itself is a virtual function in a subclass

  ```
  class F : public FunctionClass
  {
  public:
      virtual double operator() (double x) const;
  };
  ```

PDE problems

- Partial differential equations (PDEs) are used to describe numerous processes in physics, engineering, biology, geology, meteorology, ...
- PDEs typically contain
  1. input quantities: coefficients in the PDEs, boundary conditions, etc.
  2. output quantities: the solution
- Input/output quantities are scalar or vector fields
- field = function defined over a 1D, 2D or 3D grid

Example: scalar field over a 2D grid



PDE codes

- PDEs are solved numerically by finite difference, finite element or finite volume methods
- PDE codes are often large and complicated
- Finite element codes can easily be x00 000 lines in Fortran 77
- PDE codes can be difficult to maintain and extend
- Remedy: program closer to the mathematics, but this requires suitable abstractions (i.e. classes)

A simple model problem

- 2D linear, standard wave equation with constant wave velocity c

$$\frac{\partial^2 u}{\partial t^2} = c^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)$$

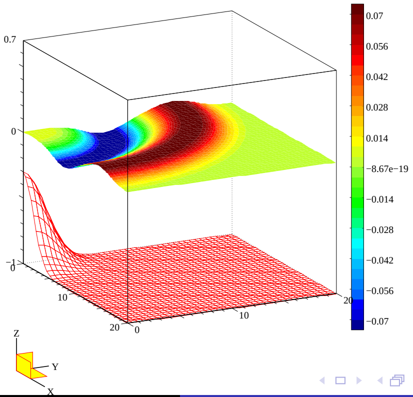
or variable wave velocity $c(x, y)$:

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x} \left(c(x, y)^2 \frac{\partial u}{\partial x} \right) + \frac{\partial}{\partial y} \left(c(x, y)^2 \frac{\partial u}{\partial y} \right)$$

- Vanishing normal derivative on the boundary
- Explicit finite difference scheme
- Uniform rectangular grid

Possible interpretation: water waves

u: water surface elevation; c^2: water depth



Basic abstractions

- Flexible array
- Grid
- Scalar field
- Time discretization parameters
- Smart pointers

References:

- Roeim and Langtangen: Implementation of a wave simulator using objects and C++
- Source code: src/C++/Wave2D

A grid class

Obvious ideas

- collect grid information in a grid class
- collect field information in a field class

Gain:

- shorter code, closer to the mathematics
- finite difference methods: minor
- finite element methods: important
- big programs: fundamental
- possible to write code that is (almost) independent of the number of space dimensions (i.e., easy to go from 1D to 2D to 3D!)

Grids and fields for FDM

Relevant classes in a finite difference method (FDM):

- Field represented by FieldLattice:
  1. a grid of type GridLattice
  2. a set of point values, `MyArray`
  3. `MyArray` is a class implementing user-friendly arrays in one and more dimensions
- Grid represented by GridLattice
  1. lattice with uniform partition in d dimensions
  2. initialization from input string, e.g.,

```
d=1 domain: [0,1], index [1:20]

d=3 [0,1]x[-2,2]x[0,10]
    indices [1:20]x[-20:20]x[0:40]
```

Working with the GridLattice class

Example of how we want to program:

```
GridLattice g;   // declare an empty grid
g.scan("d=2 [0,1]x[0,2] [1:10]x[1:40]");  // initialize g

const int i0 = g.getBase(1);   // start of first index
const int j0 = g.getBase(2);   // start of second index
const int in = g.getMaxI(1);   // end of first index
const int jn = g.getMaxI(2);   // end of second index
int i,j;
for (i = i0; i <= in; ++i) {
    for (j = i0; j <= jn; ++j) {
        std::cout << "grid point (" << i << ',' << j
                  << ") has coordinates (" << g.getPoint(1,i)
                  << ',' << g.getPoint(2,j) << ")\n";
    }
}
// other tasks:
const int nx = g.getDivisions(1);
const int ny = g.getDivisions(2);
const int dx = g.Delta(1);
const int dy = g.Delta(2);
```

The GridLattice class (1)

Data representation:

- Max/min coordinates of the corners, plus no of divisions

```
class GridLattice
{
    // currently limited to two dimensions
    static const int MAX_DIMENSIONS = 2;

    // variables defining the size of the grid
    double min[MAX_DIMENSIONS];   // min coordinate values
                                  //   in each dimension
    double max[MAX_DIMENSIONS];   // max coordinate values
                                  //   in each dimension
    int division[MAX_DIMENSIONS]; // number of points
                                  //   in each dimension
    int dimensions;               // number of dimensions
```

`static`: a common variable shared by all GridLattice objects

The GridLattice class (2)

Member functions:

- Constructors
- Initialization (through the scan function)
- Accessors (access to internal data structure)

```
public:
    GridLattice();
    GridLattice(int nx, int ny,
                double xmin_, double xmax_,
                double ymin_, double ymax_);
    void scan(const std::string& init_string);
              // scan parameters from init_string

    friend std::ostream& operator<<(std::ostream&, GridLattice&);

    int getNoSpaceDim () const;

    double xMin(int dimension) const;
    double xMax(int dimension) const;

    // get the number of points in each dimension:
    int getDivisions(int i) const;
```

The GridLattice class (3)

```
    ...
    // get total no of points in the grid:
    int getNoPoints() const;

    double Delta(int dimension) const;
    double getPoint(int dimension, int index);

    // start of indexed loops in dimension-direction:
    int getBase(int dimension) const;
    // end   of indexed loops in dimension-direction:
    int getMaxI(int dimension) const;
};
```

Mutators, i.e., functions for setting internal data members, are not implemented here. Examples could be `setDelta`, `setXmax`, etc.

The GridLattice class (4)

```
double GridLattice:: xMin(int dimension) const
{ return min[dimension - 1]; }

double GridLattice:: xMax(int dimension) const
{ return max[dimension - 1]; }

inline int GridLattice:: getDivisions(int i) const
{ return division[i-1]; }

int GridLattice:: getNoPoints() const
{
    int return_value = 1;
    for(int i = 0; i != dimensions; ++i)
        return_value *= division[i];

    return return_value;
}
```

The GridLattice class (5)

- Nested inline functions:

```
inline double GridLattice:: Delta(int dimension) const
{
   return (max[dimension-1] - min[dimension-1])
          / double(division[dimension-1]);
}

inline double GridLattice::
       getPoint(int dimension, int index)
{
   return min[dimension-1] +
          (Delta(dimension) * (index - 1));
}
```

- Some of today's compilers do not inline nested inlined functions

The GridLattice class (6)

- Remedy: can use a preprocessor macro and make our own tailored optimization:

```
inline double GridLattice:: getPoint
               (int dimension, int index)
{
#ifdef NO_NESTED_INLINES
   return min[dimension-1] +
          ((max[dimension-1]- min[dimension-1])
          / double(division[dimension-1]))*(index - 1);
#else
   return min[dimension-1] +
          (Delta(dimension) * (index - 1));
#endif
}
```

The GridLattice class (7)

- The scan function is typically called as follows:

```
// GridLattice g
g.scan("d=2 [0,1]x[0,2] [1:10]x[1:40]");
```

- To parse the string, use functionality in the C++ standard library:

```
void GridLattice:: scan(const string& init_string)
{
  using namespace std;  // allows dropping std:: prefix
  // work with an istream interface to strings:
  istringstream is(init_string.c_str());

  // ignore "d="
  is.ignore(1, 'd');  is.ignore(1, '=');

  // get the dimensions
  is >> dimensions;
  if (dimensions < 1 || dimensions > MAX_DIMENSIONS) {
     // write error message
     ...
  }
```

The GridLattice class (8)

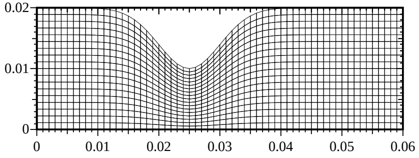
- Constructor with data for initialization:

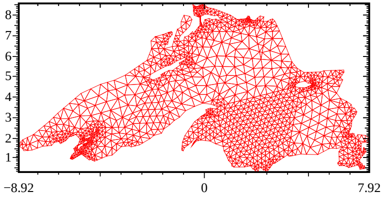
```
GridLattice:: GridLattice(int nx, int ny,
                          double xmin, double xmax,
                          double ymin, double ymax)
{
   dimensions = 2;
   max[0] = xmax;      max[1] = ymax;
   min[0] = xmin;      min[1] = ymin;
   division[0] = nx;  division[1] = ny;
}
```

- Constructor with no arguments:

```
GridLattice:: GridLattice()
{
   // set meaningful values:
   dimensions = 2;
   for (int i = 1; i <= MAX_DIMENSIONS; ++i) {
       min[i] = 0;  max[i] = 1;  division[i] = 2;
   }
}
```

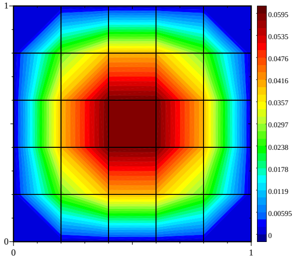
Various types of grids





More complicated data structures but the grid is still a single variable in the simulation code

The FieldLattice class (1)



Collect all information about a scalar finite difference-type field in a class with

- pointer to a grid (allows the grid to be shared by many fields)
- pointer to an array of grid point values
- optional: name of the field

The FieldLattice class (2)

```
class FieldLattice
{
public:
  Handle<GridLattice>        grid_lattice;
  Handle< MyArray<real> > grid_point_values;
  std::string                fieldname;

public:
  // make a field from a grid and a fieldname:
  FieldLattice(GridLattice& g,
               const std::string& fieldname);

  // enable access to grid-point values:
        MyArray<real>& values()
        { return *grid_point_values; }
  const MyArray<real>& values() const
        { return *grid_point_values; }

  // enable access to the grid:
        GridLattice& grid()       { return *grid_lattice; }
  const GridLattice& grid() const { return *grid_lattice; }

  std::string name() const        { return fieldname; }
};
```

The FieldLattice class (3)

```
FieldLattice:: FieldLattice(GridLattice& g,
                            const std::string& name_)
{
  grid_lattice.rebind(&g);
  // allocate the grid_point_values array:
  if (grid_lattice->getNoSpaceDim() == 1)
    grid_point_values.rebind(
        new MyArray<real>(grid_lattice->getDivisions(1)));
  else if (grid_lattice->getNoSpaceDim() == 2)
    grid_point_values.rebind(new MyArray<real>(
       grid_lattice->getDivisions(1),
       grid_lattice->getDivisions(2)));
  else
    ; // three-dimensional fields are not yet supported...
  fieldname = name_;
}
```

A few remarks on class FieldLattice

- Inline functions are obtained by implementing the function body inside the class declaration
- We use a parameter `real`, which equals `float` or `double` (by default)
- The `Handle<>` construction is a *smart pointer*, implementing reference counting and automatic deallocation (almost garbage collection)
- Using a `Handle<GridLattice>` object instead of a `GridLattice` object, means that a grid can be shared among several fields

C/C++ pointers cause trouble...

Observations:

- Pointers are bug no 1 in C/C++
- Dynamic memory demands pointer programming
- Lack of garbage collection (automatic clean-up of memory that is no longer in use) means that manual deallocation is required
- Every `new` must be paried with a `delete`
- Codes with memory leakage eat up the memory and slow down computations
- How to determine when memory is no longer in use? Suppose 5 fields point to the same grid, when can we safely remove the grid object?

Smart pointers with reference counting

Solution to the mentioned problems:

- Avoid explicit deallocation
- Introduce reference counting, i.e., count the number of pointer references to an object and perform a delete only if there are no more references to the object

Advantages:

- negligible overhead
- (kind of) automatic garbage collection
- several fields can safely share one grid

Smart pointers: usage

```
Handle<X> x;           // NULL pointer

x.rebind (new X());    // x points to new X object

someFunc (*x);         // send object as X& argument

// given Handle(X) y:
x.rebind (*y);         // x points to y’s object
```

Time discretization parameters

- Collect time discretization parameters in a class:
  1. current time value
  2. end of simulation
  3. time step size
  4. time step number

```
class TimePrm
{
  double time_;    // current time value
  double delta;    // time step size
  double stop;     // stop time
  int    timestep; // time step counter

public:
  TimePrm(double start, double delta_, double stop_)
  { time_=start; delta=delta_; stop=stop_; initTimeLoop(); }

  double time()    { return time_; }
  double Delta()   { return delta; }

  void initTimeLoop() { time_ = 0; timestep = 0; }

  bool finished()
     { return (time_ >= stop) ? true : false; }
```

Simulator classes

- The PDE solver is a class itself
- This makes it easy to
  1. combine solvers (systems of PDEs)
  2. extend/modify solvers
  3. couple solvers to optimization, automatic parameter analysis, etc.
- Typical look (for a stationary problem):

```
class MySim
{
protected:
  // grid and field objects
  // PDE-dependent parameters
public:
  void scan();          // read input and init
  void solveProblem();
  void resultReport();
};
```

Our wave 2D equation example

What are natural objects in a 2D wave equation simulator?

- GridLattice
- FieldLattice for the unknown u field at three consecutive time levels
- TimePrm
- Class hierarchy of functions:
    1. initial surface functions I(x,y) and/or
    2. bottom functions H(x,y)

Use smart pointers (Handles) instead of ordinary C/C++ pointers

Hierarchy of functions

- Class WaveFunc: common interface to all I(x,y) and H(x,y) functions for which we have explicit mathematical formulas

```
class WaveFunc
{
public:
  virtual ~WaveFunc() {}
  virtual real valuePt(real x, real y, real t = 0) = 0;
  virtual void scan() = 0;  // read parameters in depth func.
  virtual std::string& formula() = 0;  // function label
};
```

- Subclasses of WaveFunc implement various I(x,y) and H(x,y) functions, cf. the ODEProblem hierarchy

Example

```
class GaussianBell : public virtual WaveFunc
{
protected:
  real A, sigma_x, sigma_y, xc, yc;
  char fname; // I or H
  std::string formula_str;  // for ASCII output of function
public:
  GaussianBell(char fname_ = ' ');
  virtual real valuePt(real x, real y, real t = 0);
  virtual void scan();
  virtual std::string& formula();
};
```

Example cont.

```
inline real GaussianBell:: valuePt(real x, real y, real)
{
  real r = A*exp(-(sqr(x - xc)/(2*sqr(sigma_x))
               + sqr(y - yc)/(2*sqr(sigma_y))));
  return r;
}

GaussianBell:: GaussianBell(char fname_)
{ fname = fname_; }

std::string& GaussianBell:: formula()
{ return formula_str; }

void GaussianBell:: scan ()
{
  A = CommandLineArgs::read("-A_" + fname, 0.1);
  sigma_x = CommandLineArgs::read("-sigma_x_" + fname, 0.5);
  sigma_y = CommandLineArgs::read("-sigma_y_" + fname, 0.5);
  xc = CommandLineArgs::read("-xc_" + fname, 0.0);
  yc = CommandLineArgs::read("-yc_" + fname, 0.0);
}
```

Class CommandLineArgs is our local tool for parsing the command line

The wave simulator (1)

```
class Wave2D
{
  Handle<GridLattice>  grid;
  Handle<FieldLattice> up; // solution at time level l+1
  Handle<FieldLattice> u;  // solution at time level l
  Handle<FieldLattice> um; // solution at time level l-1
  Handle<TimePrm>      tip;
  Handle<WaveFunc>     I;  // initial surface
  Handle<WaveFunc>     H;  // bottom function
  // load H into a field lambda for efficiency:
  Handle<FieldLattice> lambda;

  void timeLoop();          // perform time stepping
  void plot(bool initial); // dump fields to file, plot later
  void WAVE(FieldLattice& up, const FieldLattice& u,
            const FieldLattice& um, real a, real b, real c);

  void setIC();             // set initial conditions
  real calculateDt(int func); // calculate optimal timestep
public:
  void scan();              // read input and initialize
  void solveProblem();      // start the simulation
};
```

The wave simulator (2)

```
void Wave2D:: solveProblem ()
{
  setIC();          // set initial conditions
  timeLoop();       // run the algorithm
}

void Wave2D:: setIC ()
{
  const int nx = grid->getMaxI(1);
  const int ny = grid->getMaxI(2);

  // fill the field for the current time period
  // with values from the appropriate function
  MyArray<real>& uv = u->values();
  for (int j = 1; j <= ny; j++)
      for (int i = 1; i <= nx; i++)
          uv(i, j) = I->valuePt(grid->getPoint(1, i),
                                grid->getPoint(2, j));

  // set the help variable um:
  WAVE (*um, *u, *um, 0.5, 0.0, 0.5);
}
```

The wave simulator (3)

```
void Wave2D:: timeLoop ()
{
  tip->initTimeLoop();
  plot(true);  // always plot initial condition (t=0)

  while(!tip->finished()) {
    tip->increaseTime();

    WAVE (*up, *u, *um, 1, 1, 1);
    // move handles (get ready for next step):
    tmp = um; um = u; u = up; up = tmp;

    plot(false);
  }
}
```

The wave simulator (4)

```
void Wave2D:: scan ()
{
  // create the grid...
  grid.rebind(new GridLattice());
  grid->scan(CommandLineArgs::read("-grid",
      "d=2 [-10,10]x[-10,10] [1:30]x[1:30]"));
  std::cout << *grid << '\n';

  // create new fields...
  up.     rebind(new FieldLattice(*grid, "up"));
  u.      rebind(new FieldLattice(*grid, "u"));
  um.     rebind(new FieldLattice(*grid, "um"));
  lambda.rebind(new FieldLattice(*grid, "lambda"));
```

The wave simulator (5)

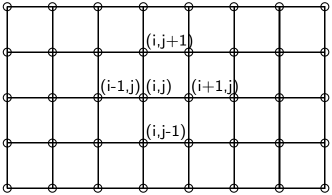
```
  // select the appropriate I and H
  int func = CommandLineArgs::read("-func", 1);
  if (func == 1) {
      H.rebind(new GaussianBell('H'));
      I.rebind(new GaussianBell('U'));
  }
  else {
      H.rebind(new Flat());
      I.rebind(new Plug('U'));
  }

  // initialize the parameters in the functions
  H->scan();
  I->scan();

  tip.rebind(new TimePrm(0, calculateDt(func),
      CommandLineArgs::read("-tstop", 30.0)));
}
```

The model problem

$$\begin{aligned}
\frac{\partial}{\partial x}\left(H(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(H(x,y)\frac{\partial u}{\partial y}\right) &= \frac{\partial^2 u}{\partial t^2}, \quad \text{in } \Omega \\
\frac{\partial u}{\partial n} &= 0, \quad \text{on } \partial\Omega \\
u(x,y,0) &= I(x,y), \quad \text{in } \Omega \\
\frac{\partial}{\partial t}u(x,y,0) &= 0, \quad \text{in } \Omega
\end{aligned}$$

Discretization (1)

Introduce a rectangular grid: $x_i = (i-1)\Delta x, \quad y_j = (j-1)\Delta y$



Seek approximation $u^{\ell}_{i,j}$ on the grid at discrete times $t_{\ell} = \ell\Delta t$

Discretization (2)

- Approximate derivatives by central differences

$$\frac{\partial^2 u}{\partial t^2} \approx \frac{u_{i,j}^{\ell+1} - 2u_{i,j}^{\ell} + u_{i,j}^{\ell-1}}{\Delta t^2}$$

  Similarly for the x and y derivatives.

- Assume for the moment that $H \equiv 1$, then

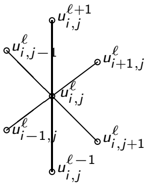
$$\frac{u_{i,j}^{\ell+1} - 2u_{i,j}^{\ell} + u_{i,j}^{\ell-1}}{\Delta t^2} = \frac{u_{i+1,j}^{\ell} - 2u_{i,j}^{\ell} + u_{i-1,j}^{\ell}}{\Delta x^2} + \frac{u_{i,j+1}^{\ell} - 2u_{i,j}^{\ell} + u_{i,j-1}^{\ell}}{\Delta y^2}$$

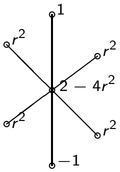
Discretization (3)

- Solve for $u_{i,j}^{\ell+1}$ (the only unknown quantity), simplify with $\Delta x = \Delta y$:

$$
\begin{aligned}
u_{i,j}^{\ell+1} &= 2u_{i,j}^{\ell} - u_{i,j}^{\ell-1} + \Delta t^2 [\Delta u]_{i,j}^{\ell} \\
[\Delta u]_{i,j}^{\ell} &= \Delta x^{-2}(u_{i+1,j}^{\ell} + u_{i-1,j}^{\ell} + \\
&\quad u_{i,j+1}^{\ell} + u_{i,j-1}^{\ell} - 4u_{i,j}^{\ell})
\end{aligned}
$$

Graphical illustration





Discretization (4)

A spatial term like $(Hu_y)_y$ takes the form

$$\frac{1}{\Delta y}\left(H_{i,j+\frac{1}{2}}\left(\frac{u^{\ell}_{i,j+1}-u^{\ell}_{i,j}}{\Delta y}\right)-H_{i,j-\frac{1}{2}}\left(\frac{u^{\ell}_{i,j}-u^{\ell}_{i,j-1}}{\Delta y}\right)\right)$$

Thus we derive

$$\begin{aligned}
u^{\ell+1}_{i,j} &= 2u^{\ell}_{i,j}-u^{\ell-1}_{i,j}\\
&\quad +r_x^2\left(H_{i+\frac{1}{2},j}(u^{\ell}_{i+1,j}-u^{\ell}_{i,j})-H_{i-\frac{1}{2},j}(u^{\ell}_{i,j}-u^{\ell}_{i-1,j})\right)\\
&\quad +r_y^2\left(H_{i,j+\frac{1}{2}}(u^{\ell}_{i,j+1}-u^{\ell}_{i,j})-H_{i,j-\frac{1}{2}}(u^{\ell}_{i,j}-u^{\ell}_{i,j-1})\right)\\
&= 2u^{\ell}_{i,j}-u^{\ell-1}_{i,j}+[\Delta u]^{\ell}_{i,j}
\end{aligned}$$

where $r_x=\Delta t/\Delta x$ and $r_y=\Delta t/\Delta y$

Algorithm (1)

- Define:
  - storage $u^{+}_{i,j}$, $u_{i,j}$, $u^{-}_{i,j}$ for $u^{\ell+1}_{i,j}$, $u^{\ell}_{i,j}$, $u^{\ell-1}_{i,j}$
  - whole grid: $(\bar{\infty}) = \{i = 1, \ldots, n_x, j = 1, \ldots, n_y\}$
  - inner points: $(\infty) = \{i = 2, \ldots, n_x - 1, j = 1, \ldots, n_y - 1\}$
- Set initial conditions

$$u_{i,j} = I(x_i, y_j), \quad (i,j) \in (\infty)$$

- Define $u^{-}_{i,j}$

$$u^{-}_{i,j} = u_{i,j} + [\Delta u]_{i,j}, \quad (i,j) \in (\infty)$$

Algorithm (2)

- Set $t = 0$
- While $t < t_{\text{stop}}$

  - $t = t + \Delta t$
  - Update all inner points

    $$u^{+}_{i,j} = 2u_{i,j} - u^{-}_{i,j} + [\Delta u]_{i,j}, \quad (i,j) \in (\infty)$$

  - Set boundary conditions ....
  - Initialize for next step

    $$u^{-}_{i,j} = u_{i,j}, \quad u_{i,j} = u^{+}_{i,j}, \quad (i,j) \in (\bar{\infty})$$

(without H)

Implementing boundary conditions (1)

We shall impose full reflection of waves like in a swimming pool

$$\frac{\partial u}{\partial n} \equiv \nabla u \cdot \mathbf{n} = 0$$

Assume a rectangular domain. At the vertical ($x =$constant) boundaries the condition reads:

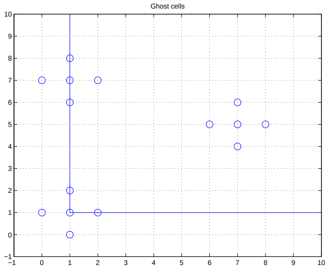
$$0 = \frac{\partial u}{\partial n} = \nabla u \cdot (\pm 1, 0) = \pm \frac{\partial u}{\partial x}$$

Similarly at the horizontal boundaries ($y =$constant)

$$0 = \frac{\partial u}{\partial n} = \nabla u \cdot (0, \pm 1) = \pm \frac{\partial u}{\partial y}$$

Implementing boundary conditions (2)

Applying the finite difference stencil at the left boundary ($i = 1$, $j = 1, \ldots, n_y$):



The computations involve cells outside our domain. This is a problem. The obvious answer is to use the boundary condition, e.g.,

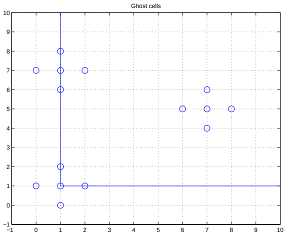
$$\frac{u_{2,j} - u_{0,j}}{2\Delta x} = 0 \qquad \Rightarrow \qquad u_{0,j} = u_{2,j}$$

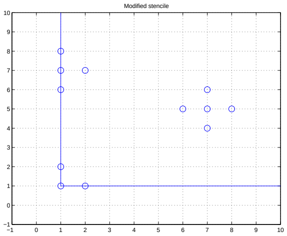
But how do we include this into the scheme..?

Implementing boundary conditions (3)

There are two ways to include boundary conditions:

- Add “ghost cells” at boundary with explicit updating of fictitious values outside the domain based upon values in the interior, e.g., $u_{0,j} = u_{2,j}$
- Modify stencil at boundary: $u_{xx} \longrightarrow \frac{u_{2,j} - 2u_{1,j} + u_{2,j}}{\Delta x^2}$





Updating of internal points

WAVE$(u^{+}, u, u^{-}, a, b, c)$

- UPDATE ALL INNER POINTS:

$$u^{+}_{i,j} = 2au_{i,j} - bu^{-}_{i,j} + c[\triangle u]_{i,j}, \quad (i,j) \in (\infty)$$

Updating of internal and boundary points

- UPDATE BOUNDARY POINTS:

$$
\begin{aligned}
i = 1, \quad & j = 2, \ldots, n_y - 1; \\
& u^+_{i,j} = 2au_{i,j} - bu^-_{i,j} + c[\triangle u]_{i,j:i-1\rightarrow i+1}, \\
i = n_x, \quad & j = 2, \ldots, n_y - 1; \\
& u^+_{i,j} = 2au_{i,j} - bu^-_{i,j} + c[\triangle u]_{i,j:i+1\rightarrow i-1}, \\
j = 1, \quad & i = 2, \ldots, n_x - 1; \\
& u^+_{i,j} = 2au_{i,j} - bu^-_{i,j} + c[\triangle u]_{i,j:j-1\rightarrow j+1}, \\
j = n_y, \quad & i = 2, \ldots, n_x - 1; \\
& u^+_{i,j} = 2au_{i,j} - bu^-_{i,j} + c[\triangle u]_{i,j:j-1\rightarrow j+1},
\end{aligned}
$$

Updating of corner points

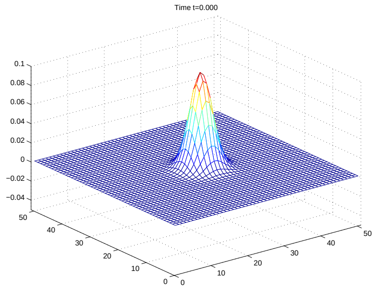
- UPDATE CORNER POINTS ON THE BOUNDARY:

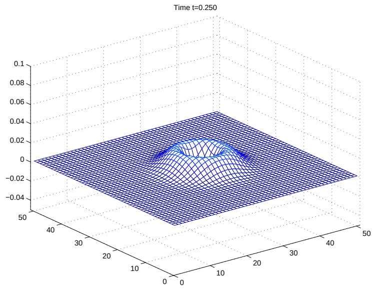
$$
\begin{aligned}
&i=1, && j=1;\\
& && u^+_{i,j} = 2au_{i,j} - bu^-_{i,j} + c[\triangle u]_{i,j:i-1\rightarrow i+1,j-1\rightarrow j+1}\\
&i=n_x, && j=1;\\
& && u^+_{i,j} = 2au_{i,j} - bu^-_{i,j} + c[\triangle u]_{i,j:i+1\rightarrow i-1,j-1\rightarrow j+1}\\
&i=1, && j=n_y;\\
& && u^+_{i,j} = 2au_{i,j} - bu^-_{i,j} + c[\triangle u]_{i,j:i-1\rightarrow i+1,j+1\rightarrow j-1}\\
&i=n_x, && j=n_y;\\
& && u^+_{i,j} = 2au_{i,j} - bu^-_{i,j} + c[\triangle u]_{i,j:i+1\rightarrow i-1,j+1\rightarrow j-1}
\end{aligned}
$$

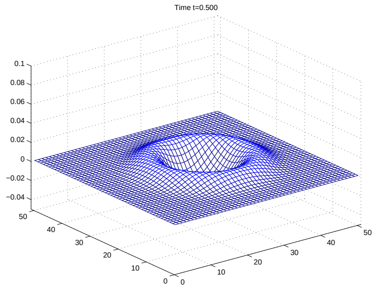
Modified algorithm

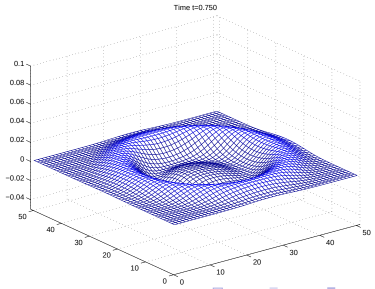
- DEFINITIONS: as above
- INITIAL CONDITIONS: $u_{i,j} = I(x_i, y_j), \quad (i,j) \in (\bar{\infty})$
- VARIABLE COEFFICIENT: set/get values for λ
- SET ARTIFICIAL QUANTITY $u^-_{i,j}$: WAVE$(u^-, u, u^-, 0.5, 0, 0.5)$
- Set $t = 0$
- While $t \le t_{\text{stop}}$
  - $t \leftarrow t + \Delta t$
  - (If λ depends on t: update λ)
  - UPDATE ALL POINTS: WAVE$(u^+, u, u^-, 1, 1, 1)$
  - INITIALIZE FOR NEXT STEP:
    $u^-_{i,j} = u_{i,j}, \quad u_{i,j} = u^+_{i,j}, \quad (i,j) \in (\infty)$

Visualizing the results









Ex: waves caused by earthquake (1)

- Physical assumption: long waves in shallow water

$$\frac{\partial^2 u}{\partial t^2} = \nabla \cdot \left[H(\mathbf{x})\nabla u\right]$$

- Rectangular domain $\Omega = (s_x, s_x + w_x) \times (s_y, s_y + w_y)$ with initial (Gaussian bell) function

$$I(x,y) = A_u \exp\left(-\frac{1}{2}\left(\frac{x - x_u^c}{\sigma_{ux}}\right)^2 - \frac{1}{2}\left(\frac{y - y_u^c}{\sigma_{uy}}\right)^2\right)$$

Ex: waves caused by earthquake (1)

- The equations model an initial elevation caused by an earthquake. The earthquake takes place near an underwater seamount

$$H(x,y) = 1 - A_H \exp\left(-\frac{1}{2}\left(\frac{x - x_H^c}{\sigma_{Hx}}\right)^2 - \frac{1}{2}\left(\frac{y - y_H^c}{\sigma_{Hy}}\right)^2\right)$$

- Simulation case inspired by the Gorringe Bank southwest of Portugal. Severe ocean waves have been generated due to earthquakes in this region.

Acknowledgements

This collection of slides on C++ and C programming has benefited greatly from corrections and additions suggested by

- Igor Rafienko
- Vetle Roeim
- Knut-Andreas Lie