```
 UW PICO(tm) 4.6                New Buffer                          Modified

program next_big_project

end program next_big_project






^G Get Help ^O WriteOut ^R Read File  ^Y Prev Pg   ^K Cut Text    ^C Cur Pos
^X Exit     ^J Justify  ^W Where is   ^V Next Pg   ^U UnCut Text  ^T To Spell
```

Scientific Programming

LECTURE 4: DESIGNING A NEW CODE

BEFORE WRITING THE FIRST LINE OF THE CODE

- Classify what you are doing
  - One-time effort (e.g. extract information from a text file, send the same email to 320 people)
  - A useful tool (e.g. send multiple emails to 320 people)
  - Prototype code (I do not recommend writing a production code from scratch)
- Write down the equations that you have to solve, algorithms that you will use and data manipulations that you need to perform.
- Make an inventory of existing (tested) subroutines relevant to your task.

NOW YOU CAN START WRITING YOUR CODE

- I will let you write a one-time code any way you like – I don’t care
- In the other two cases, well all scientific codes and most non-scientific ones consist of 4 parts:
  1. Input
  2. Initialization
  3. Processing
  4. Output

YOU ARE READY TO WRITE THE MAIN PROGRAM

```
int main(int nparam, char *param[])
{
  int iret;

  iret=input(nparam, param);  /* Do the input */
  if(iret)
  {
    printf("ERROR during input. Code: %d\n", iret);
  }

  iret=init();    /* Initialize things that do not */
                  /* need to be re-computed later  */
  if(iret) printf("ERROR during init. Code: %d\n", iret);

  if(process())   /* Process/compute */
  {
    printf("ERROR during processing. Code: %d\n", iret);
  }
  if(output())    /* Output/store the results */
    printf("ERROR during output. Code: %d\n", iret);
}
```

DON'T YOU BELIEVE ME?

```
PROGRAM OPAC_3D
   USE SHOW_VERSION
   USE INPUT_INIT_3D
   USE FORT_UNITS
   INTEGER,PARAMETER    :: WLGRID_SIZE=6000
   INTEGER              :: nWLGRID
   REAL(8)              :: WLGRID(WLGRID_SIZE)
   REAL(4)              :: dTemp,dPlog,OPAC(WLGRID_SIZE)

   CALL SHOW_VER()      ! Print version with authorization stamp
! Read line list, model names, abundances and T-P table resolution
   CALL INPUT(dTemp,dPlog) ! Initialize things for EOS and
                           ! opacity calculations

   CALL INIT(nWLGRID,WLGRID,WLGRID_SIZE) ! Initialize things for EOS and
                                         ! opacity calculations

   CALL OPACITY_3D(WLGRID,WLGRID_SIZE,nWLGRID, & ! Read model, create T-P table
                   dTemp,dPlog,OPAC)             ! solve EOS, compute opacities

   CALL OUTPUT_OPACITY(WLGRID,WLGRID_SIZE,nWLGRID, & ! Store results in a file
                       dTemp,dPlog,OPAC)
END PROGRAM OPAC_3D
```

WHAT SHALL WE DO NEXT?

- List available subroutines and the interfaces to them
- List additional tools/algorithms needed
- Is portability an issue?
  - *NO:* check local libraries for existing algorithms. If you have a choice go for the most advanced ones.
  - Y*ES:* still look for libraries but restrict yourself for the most common ones (BLAS, LAPACK) or those available as source code (Netlib, Num. Rec.)
- Finally, determine what needs to be programmed from scratch

NOW THE PART THAT YOU HAVE TO YOURSELF

- All (mathematical) algorithms should be made to subroutines so that you can test them separately
- You can write a simple driver for testing
- Example:

```
      subroutine rk4(h0,x1,x2,y1,y2,f)
      implicit none
      real x1,x2,y1,y2,h0,f
      external f
      real d1,d2,d3,d4,h,y
c
      h=h0
      d1=f(x1,y1)
      d2=f(x1+h*0.5,y1+d1*h*0.5)
      d3=f(x1+h*0.5,y1+d2*h*0.5)
      d4=f(x1+h,y1+d3*h)
      y2=y1+h*(d1+2.*(d2+d3)+d4)/6.
      ...
```

```
      implicit none
      real x1,x2,y1,y2,h0
      x1=0.
      x2=10.
      y1=33.
      h0=0.1
c
      call rk4(x1,x2,y1,y2,func)
      …
```

```
      real function func(x,y)
      implicit none
c
      real x,y
      …
```

COMMENTS: HEADER

- Subroutine functionality
- Parameters (in/out, type, dimensionality)
- History (date, what was modified, who did modifications)
- Example:

```
      subroutine rk4(h0,x1,x2,y1,y2,f)
C rk4 integrates an ordinary differential equation
C Parameters:
C h0 - (r*4, scalar, in) initial step size
C x1 - (r*4, scalar, in) starting point
C x2 - (r*4, scalar, in) final point
C y1 - (r*4, scalar, in) function value at x1
C y2 - (r*4, scalar, out) function value at x2
C f  - (r*4, function, in) derivative function
C History:
C 2009-09-28 NP Wrote
C ...
```

COMMENTS: TEXT

- Use comments
- The point is to remind you what is meant if you need to comeback to this part of the code
- Separate logical sections of the code by inserting a full line(s) comment
- Individual lines can be commented in-line (not in FORTRAN 77)
- Comments can be partially replaced by more meaningful variable names:
- Find your personal balance that keeps the code compact but clearly readable

```
      subroutine rk4(...)
      implicit none
      real x_start,x_end,
     *     func_start,func_end,
     *     step_init
      real deriv
      external deriv
      real deriv1,deriv2,deriv3,deriv4
      real step,func
      ...
```

WORKING YOUR WAY

- Use the “skeleton” model (like for the main program) to create the frame of the whole code (top to bottom approach, focus on functionality, information flow, I/O.
- Think which part(s) would take most computing, which parts/variable may require higher precision.
- For existing subroutines/library functions complete and double check the interface.
- Once the skeleton of the whole code is complete start writing the missing subroutines/functions.
- Take one at a time. Focus on their functionality and interface. Make sure that all combinations of the input parameters (even not allowed) are handled
- Debug and test each functional group of subroutines separately!
- Keep the test code in the same file – just comment it out. Use comments to remind yourself what test were performed and how to repeat them.

PRECISION

Question 1: How would you compute a first derivative of a function numerically? Would you get close to the true value with smaller step?

$$\frac{y_1 - y_2}{x_1 - x_2} \xrightarrow{x_2 \to x_1} \left.\frac{dy}{dx}\right|_{x_1}$$

Question 2: How would you compute a sum of all the elements in an array? Will the result be the same of the first or the last element is much larger than the rest?

$$S \equiv \sum_i f_i = \{\text{s=0.; for(i=0; i<n; i++) s=s+f[i];}\}$$

PRECISION: HANDLING PROBLEMS

- Identify variables that have large absolute values (e.g. energies in cm^{-1}) and use double precision.
- When taking numerical derivatives keep the ratio $\left|\Delta x / x\right|$ >10^{-5} for single precision and >10^{-10} for double. Smaller Δx will make the accuracy worse.
- When doing summation over a large dynamic range (min(abs)/max(abs) < 10^{-5} for single precision) split summations:

```
if(abs(f[i])>max(f)*0.5) s1=s1+f[i]; else s2=s2+f[i];
...
s=s1+s2;
```

TYPICAL PROGRAMMING MISTAKES

- The most difficult case is … when you write: **`a=b*2+3`** instead of **`a=b*3+2`**.
  It can only be traced by a dedicated testing when you know what the answer will be.
- Next most difficult mistake – uninitialized variables.
- Going down the list: inconsistent types of the true and formal parameter. This is found by compiler for more advanced languages.
- Going beyond boundaries of allocated memory. Again, special compiler flags should help (but not if pointers are involved).

INDIVIDUAL SUBROUTINE DEVELOPEMENT

- Write the code
- Compile it (correct syntactic errors)
- Design the test routine consisting of the main program that sets up a situation where the answer is known. E.g. in case of Runge-Kutta a differential equation that can be integrated analytically.
- Modify the test to simulate a realistic case close to what the subroutine will be doing as part of the large code.
- Verify that all possible parameter values are handled properly (e.g. negative initial step of the RK4).
- You may need to test even parts of the subroutine. For example, it is a good idea to verify that the numerical derivative function (called by RK4) produces sufficient accuracy as compared to the analytical derivatives.
- Always use analytical expressions when possible.

NEXT LECTURE: STYLE AND STRUCTURE

The lecture is scheduled on the 27th of September. I will be away ☹

We take the lecture on Monday, the 26th between 15:15 and 17:00 and use the 27th slot for the home work presentation.