```
Test2> cat tt.f
        implicit none
        real*8 a(3001),b(3002),c(3003)
        integer i

        do i=1,3001
          a(i)=i
          b(i+1)=i*i
          c(i+2)=i*i*i
        enddo

        write(*,*) aa,i,a(3001),b(3002),c(303)

        end
Test2> ifort -g -C -c tt.f
tt.f(11): error #6404: This name does not have a type, and must have
an explicit type.   [AA]
        write(*,*) aa,i,a(3001),b(3002),c(303)
-------------------^
compilation aborted for tt.f (code 1)
Test2>
```

Scientific Programming

LECTURE 3: COMPILING AND LINKING

SOURCE CODE MAY BE STORED IN A SINGLE OR MULTIPLE FILES

- Simple and fully self contained programs are good to keep in a single file. E.g.format conversion:
  marcs2krz <input_files> <output_files>

- Useful subroutines are good to keep in separate files so that many program can re-use them. E.g. spline interpolation:
  spline.cpp containing *spinit* and *spinter*

COMPILER

- Compiler is a program that translates the source text into something called object file.
- Object file consists of machine command mnemonics, register references, memory references relative to the starting address of each subroutine and calls to external subroutines.
- The compiler also may introduce modifications to the source code to improve performance (optimization) or simplify debugging/profiling
- Object file cannot be executed! It must be further processed (segment address arithmetic, explicit command codes, attaching the missing subroutines). This is done by the linker
- Typically compiler is invoked from command line:
  f77 <flags> file1.f file2.f main.f file3.f
- In Unix, Linux and Mac OS X flags are prefixed with minus: -c
- In Windows flags are prefixed with a slash: /comp

COMPILER

- Individual files can be compiled separately or together but the sequence matters.
- If you have subroutines located in file1.f90 that use module located in file2.f90, file2.f90 must be compiled first. During this compilation a special file (file2.MOD) describing module properties to the outside world is created.
- Other languages are more explicit. E.g. C/C++ require header files describing calling sequences for subroutines

USING HEADER FILE

stdio.h:

```
typedef struct __sFILE {
        unsigned char *_p;      /* current position in (some)
        buffer */
        int     _r;             /* read space left for getc() */
        int     _w;             /* write space left for putc() */
        short   _flags;         /* flags, below; this FILE is free if 0
        */
        short   _file;          /* fileno, if Unix descriptor, else -1 */
        …
        /* operations */
        void    *_cookie;       /* cookie passed to io functions */
        int     (*_close)(void *)
        /* separate buffer for long sequences of ungetc() */
        struct  __sbuf _ub;     /* ungetc buffer */
        …
} FILE;
…
int      fgetc(FILE *);
char     *fgets(char *, int, FILE *);
FILE     *fopen(const char *, const char *);
int      fprintf(FILE *, const char *, ...);
int      fputc(int, FILE *);
…
```

Using stdio:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int n, char *nm[])
{
  int c, c_old, ierr;
  char *nm_out;

  if(n>2)
  {
    FILE *fi, *fo;

    c_old=0;
    if(!strcmp(nm[1],nm[2]))
    {
      nm_out=(char *)malloc(strlen(nm[2])+5);
      strcpy(nm_out, nm[2]);
      strcpy(nm_out+strlen(nm[2]),".TMP");
    }
    fi=fopen(nm[1],"rt"); if(fi==NULL) return 16;
```

HEADER FILES IN OTHER LANGUAGES

- Header files can be useful in other languages
- Suppose many of your FORTRAN subroutines work on the same multi-dimensional grid. Grid size must be known to multiple layers of subroutines but passing it through parameter list might be tedious.
- An alternative is to have declarations of dimensions in a separate file that is included in all subroutines where it is needed.
- Example:

SIZES.EOS

```
      INTEGER IONSIZ
      PARAMETER (IONSIZ=6)
C
C Parameters used by the Equation of State (EOS) code.
C
      INTEGER ELEDIM,SPCHAR,SPLSIZ
      PARAMETER (ELEDIM=120,SPCHAR=8,SPLSIZ=650)
```

eos.f

```
      integer function
     *   eqcount(elemen,spname,ion,nlines,nlist,ELESIZ)
      INCLUDE 'SIZES.EOS'

      integer nlines,nlist,ELESIZ
      character*(3) elemen(ELESIZ)
      character*2 tmp
      character*(SPCHAR) spname(nlines)
      character*(SPCHAR) tmplist(SPLSIZ),chname
      integer ion(nlines),ionmax
      real a(IONSIZ)
      double precision b(IONSIZ)
```

DEBUGGING

- One of the important functions implemented by the compiler is preparing your code for debugging
- Debugging means that a set of additional instructions is inserted in the code that
  - Instruct the CPU to react to various exceptions
  - Explicitly declared dimensions are stored in special hidden variables and whenever those variables are accessed the address is compared with the address of the first and last elements
  - Special labels are inserted in the object module marking the beginning instructions corresponding to a given line in the source code (the label is usually the source code line number). This allows tracing the error back to the source code but this also prevents advanced optimization that reshuffles the source text
  - Important flags are: -g –C -fpe
- Running a code compiled with those flags may or may not be sufficient for finding the problem. Using a debugger program may be more efficient: you can set break points, look at variable values etc.
- Good products for different platforms are DDE (often comes with compilers on Unix platforms), GDB (free GNU), TotalView (best but expensive)

PROFILING

- Similar to debugging but the inserted instructions collect statistics that is writing to the statistics file.
- Usually the relevant switch is -p
- Each subroutine records the times of the start and the end of each call.
- On a special request the time each loop execution is marked as well.
- A special profiler program reads the file and compiles the statistics.
- Profiling is very useful for identifying performance bottlenecks.
- In fact, some compiler can do profiling-based optimization

OPTIMIZATION

Optimization speeds up the code

- Dead code is removed
- Expressions involving only constants are evaluated at compilation time
- Variables that are assigned/modified inside loops but not affected by those loops are taken out
- Short loops are unrolled
- If-statements are evaluated whenever possible
- Loop indexing is replaced by a down counter

OPTIMIZATION

- Long loops are modified for using multiple pipelines:

```
do i=1,n
  a(i)=b(i)**2+log(c(i))
enddo
```

```
nn = n – mod(n, 4)
do i=1,nn,4
  a(i) = b(i)**2 + log(c(i))
  a(i+1) = b(i+1)**2 + log(c(i+1))
  a(i+2) = b(i+2)**2 + log(c(i+2))
  a(i+3) = b(i+3)**2 + log(c(i+3))
enddo
do i=nn+1,n
  a(i) = b(i)**2 + log(c(i))
enddo
```

- Relatively short subroutines are compiled and inserted in to the text of the caller (in-lining, requires the caller and the subroutine to be present in the same file).
- Cross-file optimization

LINKING

- Linking is performed by a separate program called linker (ld on Unix, Linux, Mac OS X and link on Windows)
- Linker is primarily designed for two actions:
  1. Creating a ready-to-go executable
  2. Creating a library for future linking
- An executable needs to have all the external links resolved before it can run. These are resolved by combining one or more object modules and libraries

LINKING OBJECT FILES AND LIBRARIES

- A typical structure for a linking command for Linux is:
  ld –o prog file1.o file2.o … -llib1 –llib2 –L<path> -llib3
- Of course, for a given language most of the libraries are well known and always the same
- With this in mind people created wrappers for compiler and linkers. E.g. f90 will produce an executable with the following command:
  f90 –O –o prog file1.f file2.o … –L<path> -llib3

MORE ABOUT LIBRARIES

- Libraries are collections of object modules with address calculation based on an undefined segment address.
- Libraries also have a list of names and the offset of the entry point (the first executable command in a given module relative to the start of the library) – this is the library catalog.
- Libraries come in two flavors: *static* and *dynamic*.

STATIC AND DYNAMIC LIBRARIES

- Most of the standard libraries come in both flavors
- The difference is:
  - Dynamic library module is loaded at the time it is needed during the execution. The linking process consists of including the path to the library into the executable. Thus executable is much smaller and loads/starts faster.
  - Linking with static libraries makes executable larger but fully portable.

CREATING A LIBRARY

- Compiler has to use fake segment address for address arithmetic. This is conventionally known as Position Independent Code (PIC)
- To create your own library all the source files must be compiled with a flag –fpic or –fPIC.
- The linker must be told that the output is a library. This is done with another special flag (usually –shared or –dynamic, for dynamic library or –static for a static library).

USING SCIENCE LIBRARIES

- A number of useful science/math libraries are available for free or come with compilers
- For example, BLAS and LAPACK (linear algebra and linear equations)
- Nothing beats the LAPACK LU decomposition solver for SLE …
- … except if you can afford a commercial science library like NAG or IMSL. In this case you get also optimization, non-linear equations, sparse matrices, statistics etc.

MIXING PROGRAMMING LANGUAGES

- During this course you may try different programming languages and discover that some are better suited for particular tasks (English is the language for business, French – politics, German – money, Italian – love etc.)
- Suppose you need to do some complex data manipulation, which occasionally require number crunching muscles. This calls for an object oriented language for a main program linked with powerful FORTRAN subroutines.
- How to make this:

```
f77 –O –c sub1.f sub2.f
cc –O –c prog.c
f77 –O –o prog prog.o sub1.o sub2.o –lm -lc
```

WELL, SOME LANGUAGES DO NOT NEED COMPILERS ☺

- IDL is an interpreter language. This means that one line of code is compiled and executed
- The beauty is that you can stop any time, do other things and then continue. No need for separate debugger.
- The downside is the performance
- Excellent for prototyping and graphics
- Python is a funny combination of a highly object oriented interpreter which is then translated into C and compiled. The Python debugger is free.

NEXT LECTURE: CODE DEVELOPMENT/DEBUGGING

Lecture is on September 23rd

You have over one week to do the home work

The results should be presented in the class

This may require 2-4 hours

First round: September 19th

Second round: September 21st