# Programming

Jan Thorbecke

## A perspective (Jack Dongorra)

- Programming is stuck
  - not dramatically changed since the 70's
- Change is more needed than ever
  - complexity is rising dramatically
    - highly parallel and distributed systems
    - multidisciplinary applications
- An HPC application lives much longer than the hardware
  - typical hardware life is typically 5 years
  - Fortran and C/C++ are the main programming models
- Software is a major cost component

## Contents

## Login

- Shells
  - Bash/sh
  - tcsh/csh
  - Korn
  - zsh
- Environment variables
  - .bashrc settings (source)

## What happens when you login?

- A Bash shell reads (in this order) during login:
  1. /etc/profile
  2. $HOME/.bash_profile
  3. $HOME/.bash_login
  4. $HOME/.profile
- Non a login shell (through rsh)
  5. $HOME/.bashrc
-

## Environment $PATH

- Colon separated list with directories the shell searches, for the commands the user may type.
- export PATH=.:$HOME/bin: # for Bourne, bash, related shells
  setenv PATH .:$HOME/bin: # for csh and related shells
- echo $PATH
- env (shows all environment variables)

## Environment $LD_LIBRARY_PATH

- list of directories the OS searches to include shared (dynamic) libraries which the executable need
- export LD_LIBRARY_PATH="/opt/intel/cc/10.1.015/lib";
- when you start your executable and see missing lib....so files it is very likely that your LD_LIBRARY_PATH is not set correctly

## Environment $HOME

> env (shows all defined variables)

> uname -a (type of linux system and name)

> which icc

> locate libblas.a

## What does a Compiler?

Translate

a **high level** language program

into

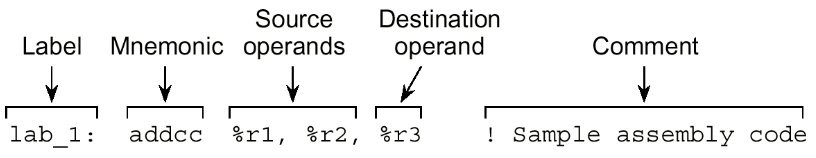
an equivalent **assembly** language program.

## High Level Language

C:

```
for (ix=ioTx; ix<nx+1; ix++) {
   for (iz=ioTz; iz<nz+1; iz++) {
      txz[ix*n1+iz] += mul[ix*n1+iz]*(
      c1*(vx[ix*n1+iz]     - vx[ix*n1+iz-1] +
          vz[ix*n1+iz]     - vz[(ix-1)*n1+iz]) +
      c2*(vx[ix*n1+iz+1]   - vx[ix*n1+iz-2] +
          vz[(ix+1)*n1+iz] - vz[(ix-2)*n1+iz]) );
        }
    }
```

## Assembly Language Format

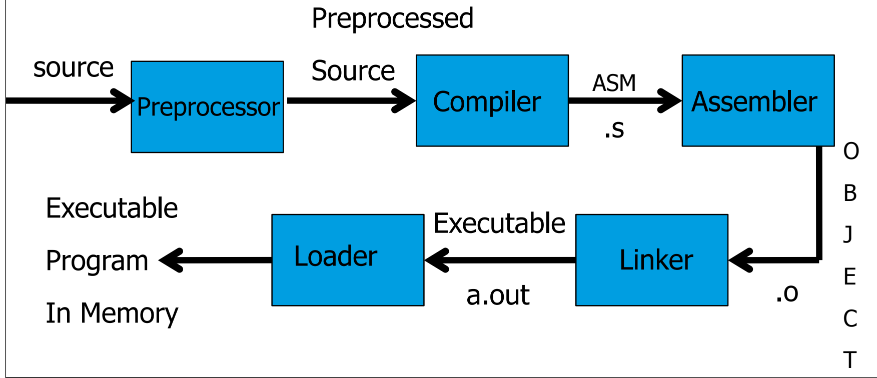


```
lab_1:  addcc  %r1, %r2,  %r3    ! Sample assembly code
```

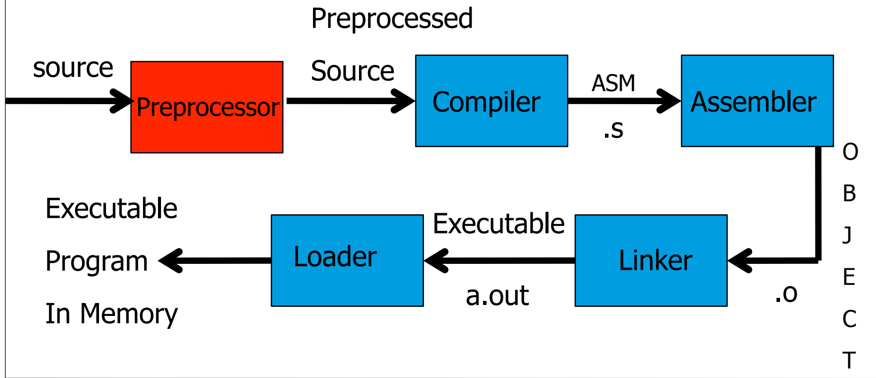
```
L121:
    # basic block 120
    movl    -80(%ebp), %eax
    imull   -44(%ebp), %eax
    addl    -48(%ebp), %eax
    addl    268(%ebp), %eax
    subl    $4, %eax
    movss   (%eax), %xmm1
    movl    -80(%ebp), %eax
    imull   -44(%ebp), %eax
    addl    -48(%ebp), %eax
    sall    $2, %eax
    movss   (%eax), %xmm0
    movaps  %xmm1, %xmm2
    subss   %xmm0, %xmm2
    movaps  %xmm2, %xmm0
    movaps  %xmm0, %xmm2
```

## Program Compilation

## Breaking Down CC



## Breaking Down CC



## C Preprocessor (cpp)

- Pass over source
  - Insert included files
  - Perform macro substitutions
- Define macros
  - #define NUM 100
  - #define xx(v,name,op,metrics) \
    v=xxinit(op,name,IR->metrics)
- gcc –E example.c sends preprocessor output to stdout

## Preprocessor

- Performs textual substitution.
- The preprocessor produces modified source code.
  → What the compiler sees is not what you gave it.
- In C/C++ the preprocessor is a standard part of the compilation system and has its own specific commands.
- E.g.
  #define
  #if

## Before and After Preprocessing

```
/* foo.h */

#define foo 1
```

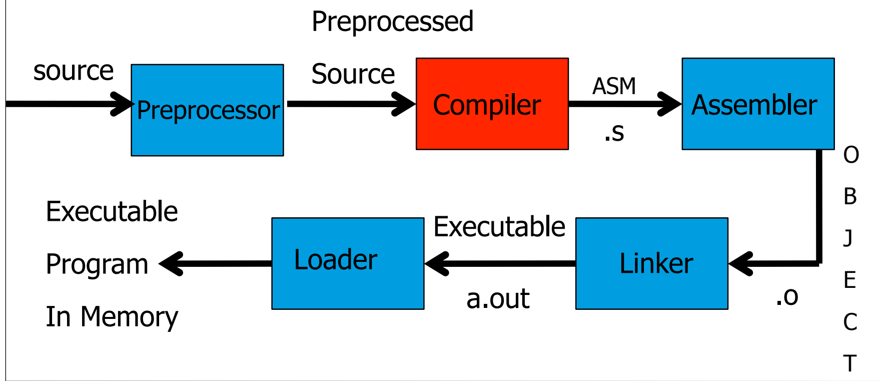
```
int foo;
#include "foo.h"
int usefoo( ) {
    foo = 3;
    return foo;
}
```

```
int usefoo( ) {
   1 = 3;
   return 1;
}
```

## Bad Macros

- Suppose we have the following macro:
  #define mult(x, y) (x * y)
- It was written with this common case in mind:
  c = mult (a, b);
- But if we have: c = mult (a+1, b);
- We get: c = (a + 1 * b);
- Which means: c = (a  + (1 * b));
- The macro should be (accounting for future uses):
  #define mult (x,y) ((x) * (y))

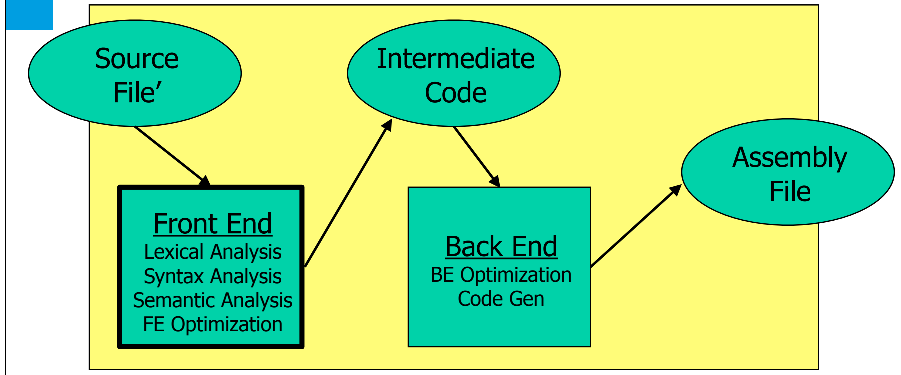
## Breaking Down CC



## Compiler

- gcc actually name of a script
- Compiler translates one language to another
- gcc compiler translates C to assembler
- gcc –S example.c "saves" example.s
- Compiler consists of
  - Parser
  - Code generation
  - Mysticism

## Within Compiler different Phases



## Front End

- The **lexical** and **syntax** analysis phases are built using declarative languages – by identifying what IS valid. Everything else is not valid.
  - Harder for those phases to produce insightful error messages.
- Most Front End errors are reported as syntax errors.
- The semantic analysis phase is created by hand (though there are some tools to help) and explicitly identifies what IS NOT valid.

## Not All Compilers Are the Same

- With 6 different compilers, the answers they give to errors vary widely.
- e.g. By mistake use a 1 for an i on the left hand side.

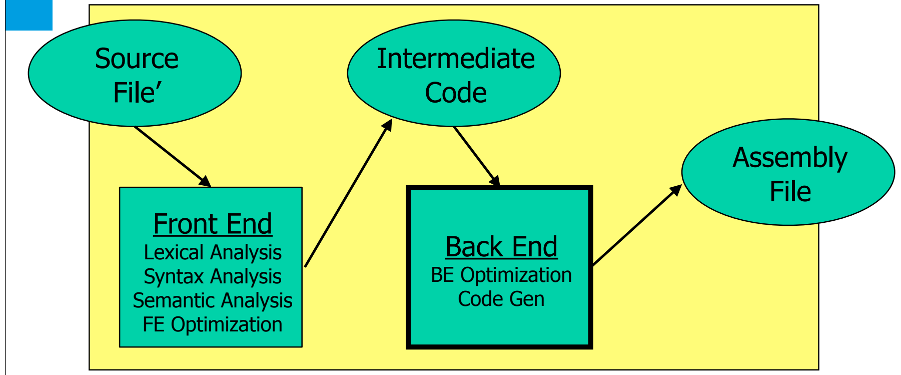
## Typo – 1 for i

```
main()
{
    int i, j;

    1 = 10;
    j = 2;
}
```

## Typo – 1 for i

1. In this statement, "1" is not an lvalue, but occurs in a context that requires one.
2. The left operand cannot be assigned to.
3. left operand must be modifiable lvalue: op "="
4. invalid lvalue in assignment
5. '=' : left operand must be l-value
6. not an lvalue

## Compiler Phases



## Optimization

- Take the program structure and rewrite it to make it more efficient.
- For instance one techniques that would help would be constant propagation.

```
A = 5;
B = A + 1;
```

↓

```
B = 6;
```

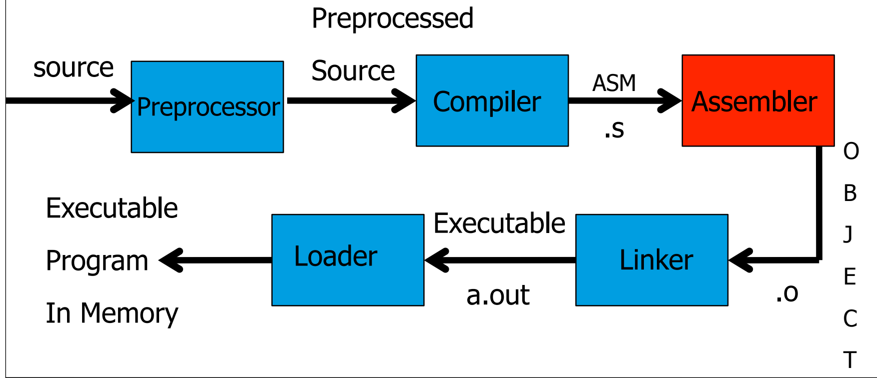
## Partial Redundancy Elimination Optimization

temp = a + b
x = temp + c;

```
x = a + b + c;
If (z == true) {
    y = a + b;
}
else {
      y = 5;
}
```

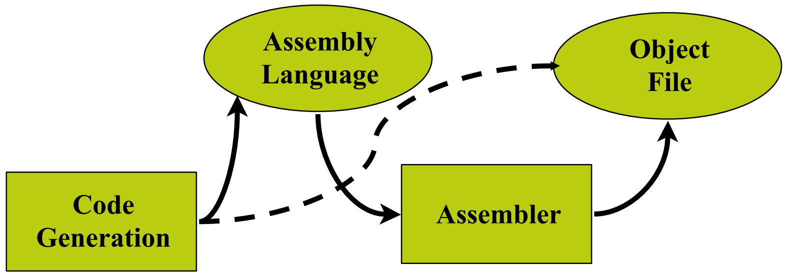
```
temp = a + b;
x = temp + c;
If (z==true) {
   y = temp;
}
else {
   y = 5;
}
```

## Breaking Down CC



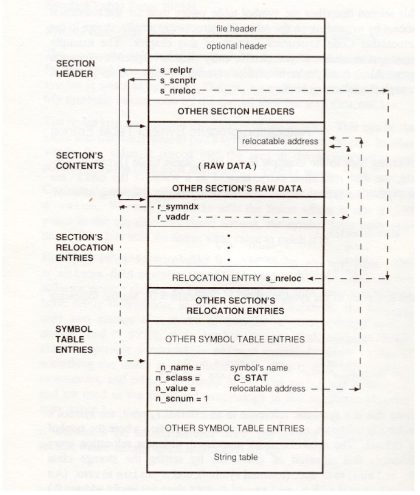
## Code Generation

- Take the intermediate form (an internal representation of the program) and generate instructions.
- The output of code generation can be an assembly file (for more modularity of phases) or an actual object file.

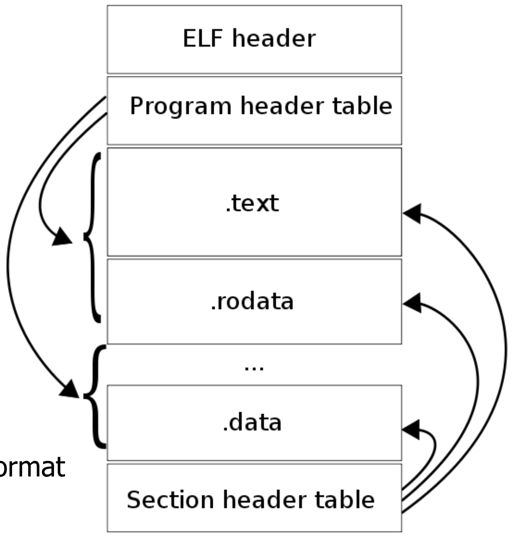


## Object File

- An object file is a collection of records that outline how a program in memory would look.
- An executable is a reconciled (no more undefined references) object file. An object file has references yet to be defined.
- Most of what you have is either: executable code, data (to be filled in later), or constant values.
- Debugging information is included.



## ELF

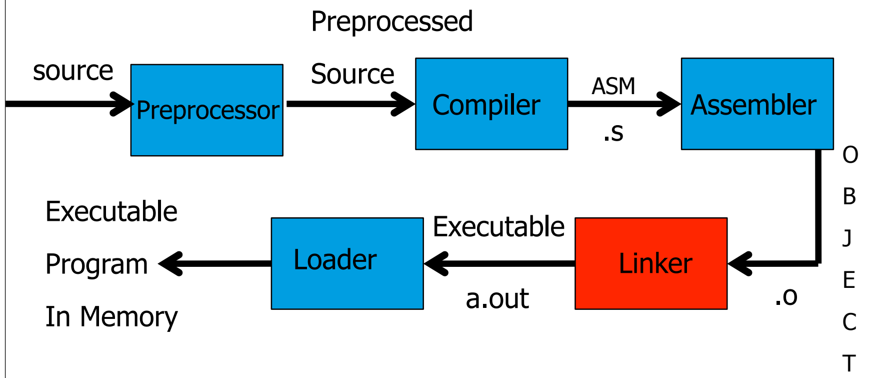


Executable and Linkable Format

## DWARF

- DWARF is a widely used standardized debugging data format. DWARF was designed along with ELF.

## Breaking Down CC



## Linker

- Combine objects, both user .o files and libraries, make an executable
- gcc *.o –lm yields a.out
- gcc –o myExec *.o –lm
- Use nm to look at objects and executables

## Archiver

- Put multiple object files into a library
- Linker takes all or nothing from an object file in a library – put 1 function per file or get a really big executable.

## Static linking

- Linker looks for undefined functions and adds library to executable if it has found one.
- Order of linked libraries on command line is of importance !
  - example sgemm_ is defined in libf77atlas.a
  - but libf77atlas uses functions not used in the program but used in the more general libatlas.a
  - link line 1: -latlas -lf77atlas   correct
  - link line 2: -lf77atlas -latlas   fails

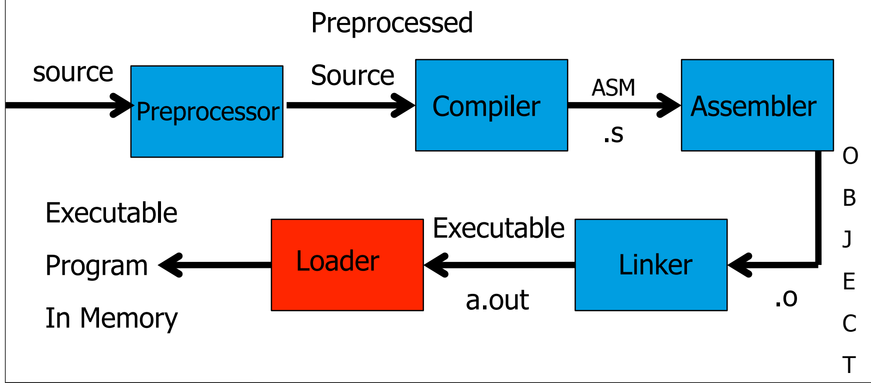
## Static and Dynamic Linking

- A program whose necessary library functions are embedded directly in the program's executable binary file is *statically* linked to its libraries
- The main disadvantage of static linkage is that every program generated must contain copies of exactly the same common system library functions
- *Dynamic* linking is more efficient in terms of both physical memory and disk-space usage because it loads the system libraries into memory only once

## Dynamic Linking

- Linking postponed until execution time
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**

## Breaking Down CC



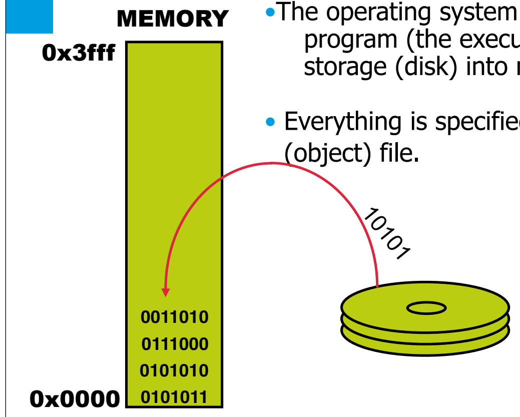
## Loader

- Gets an address to place program
- Changes necessary addresses (if any)
- Places code into memory

## Operating System

- Oversees whole process
- “Recognises” gcc example.c command
- Parses flags and arguments
- Invokes gcc executable
- Performs memory management (malloc)
- Chooses “address” to place program

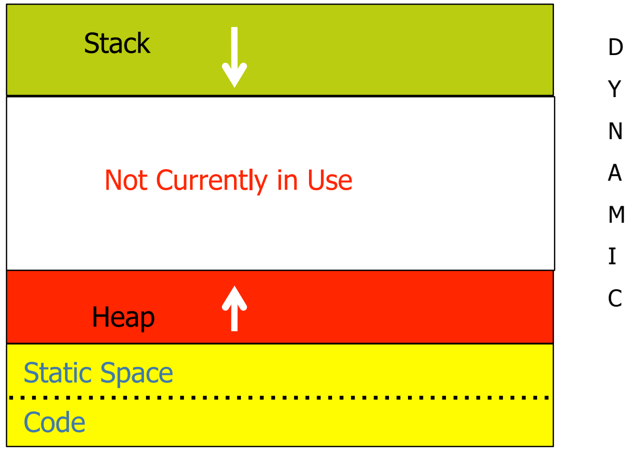
## How Does a Program Start Running



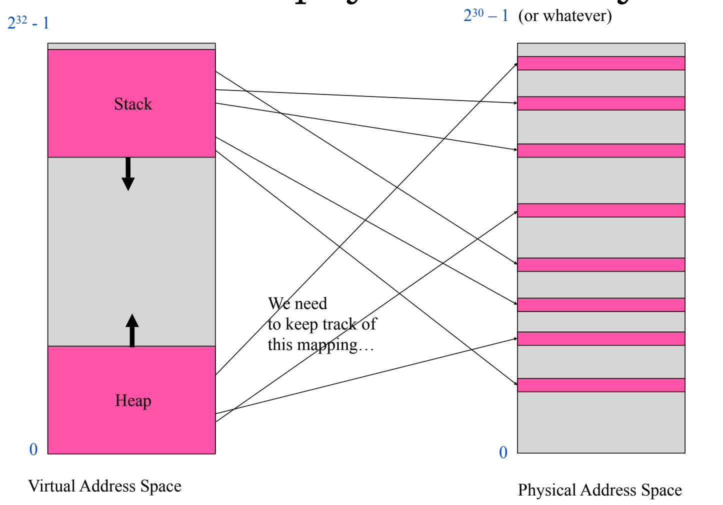
- The operating system (the loader part) copies a program (the executable) from permanent storage (disk) into memory.
- Everything is specified by the executable (object) file.

http://dbp-consulting.com/tutorials/debugging/linuxProgramStartup.html

## Logical Memory Layout

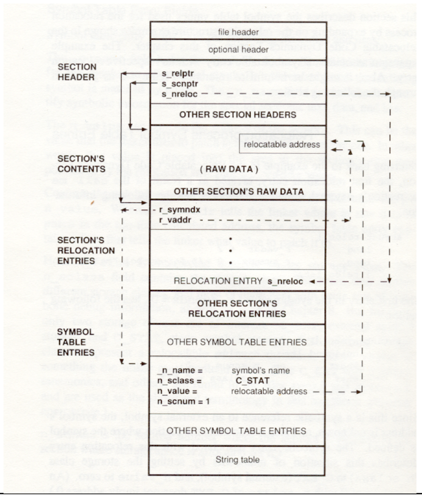


## From virtual to physical memory



## What Bits Go Where

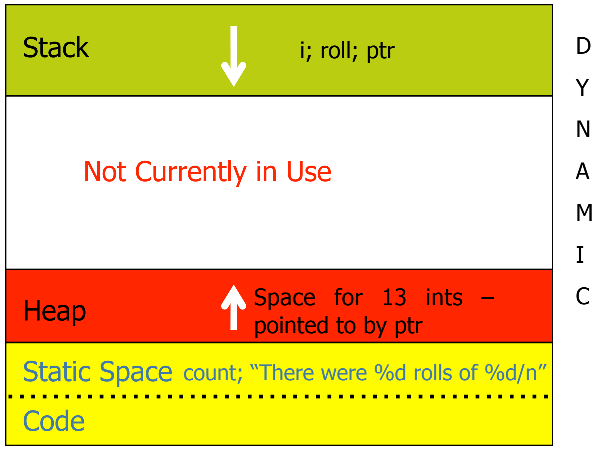
- The loader is the part of the OS that creates the memory representation of the program:
  - Executable code
  - Constants
  - Data
- The symbol and string table along with the debugging information is not put into memory.
- Everything is specified by the executable (object) file.



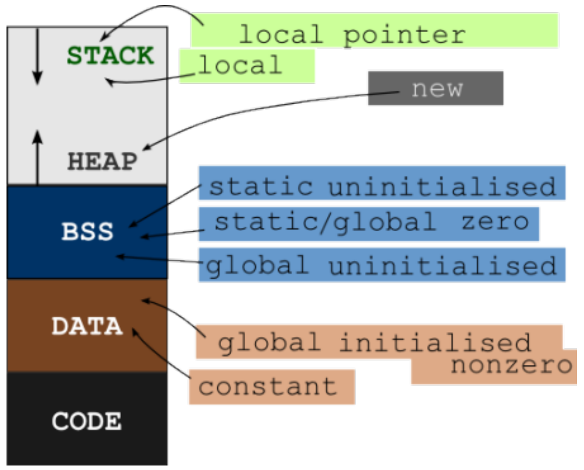
## Example: Dice.c

```
int count = 100000;
main()
{
     int i, roll, *ptr;
     ptr = (int) malloc (13 * sizeof(int));
     for (i = 0; i < 13; i++)  ptr[i] = 0;
     for ( i = 0; i < count; i++)
        roll = rand() % 6 + rand() %6 + 2
        ptr[roll]++;
     for (i = 2; i < 13; i++)
         printf("There were %d rolls of %d/
n", ptr[i],i);
}
```

## Where Are the Variables?



Stack
i; roll; ptr
Not Currently in Use
Heap
Space for 13 ints – pointed to by ptr
Static Space count; "There were %d rolls of %d/n"
Code
D Y N A M I C

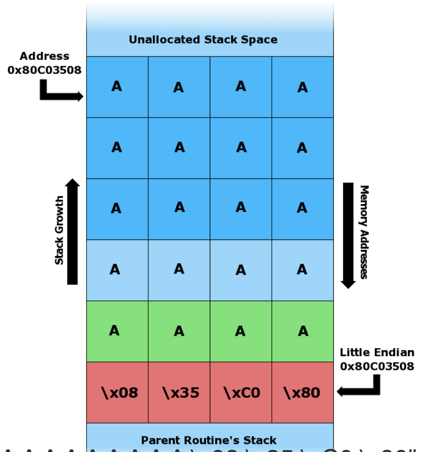


## Stack Buffer Overflow

```
#include <string.h>

void foo (char *bar)
{
   char  c[12];

   strcpy(c, bar);  // no bounds checking
}

int main (int argc, char **argv)
{
   foo(argv[1]);
}
```

http://en.wikipedia.org/wiki/Stack_buffer_overflow

## Stack Buffer Overflow



"A A A A A A A A A A A A A A A A A A A A \x08 \x35 \xC0 \x80" is the first command line argument.

"hello" is the first command line argument. Before data is copied.

## Compilers

## Compilers

- Intel: icc, ifort
  - free license for one-api compilers
- PortlandGroup: PGI, pgf90, pg (now part of Nvidia)
  - free from Nvidia
- GNU: gfortran, gcc, g++
  - free open-source
- AMD: aocc
  - free from AMD based on clang
- Apple OSX
  - clang: C-compiler bundled with X-code

## Compiler Option Groups

- Language options: -ansi
- PreProcessor: -cpp
- Output, debug, reports: -g -vec-report3
- Diagnostics: -W1
- Optimisation: -O3
  - floating point: -ffast-math
  - machine specific: -xW -march=x86-64
  - inlining, loop unrolling:
- Parallelisation OpenMP: -fopenmp (gcc) -qopenmp (intel)

## Compilation

```
> icc -O3 -w1 -DLINUX -I/home/thorbcke/include
-xAVX2 -c fdelmodc.c
```

- produces fdelmodc.o object file

```
> icc -O3 -w1 -DLINUX -I/home/thorbcke/include
-xAVX2 -S fdelmodc.c
```

- Produces fdelmodc.s assembler file
- Tip: objdump -S -l objectfile.o

```
-bash-3.00$ nm -a fdelmodc.o
0000000000000000 n .comment
0000000000000000 d .data
0000000000000000 n .note.GNU-stack
0000000000000000 a .rel.data
0000000000000000 a .rel.text
0000000000000000 r .rodata
0000000000000000 r .rodata.str1.32
0000000000000000 r .rodata.str1.4
0000000000000000 a .strtab
0000000000000000 a .symtab
0000000000000000 t .text
0000000000000048 r _2__STRING.0.0
0000000000000000 r _2__STRING.1.0
00000000000019c8 r _2__STRING.99.0
         U __assert_fail
         U __intel_cpu_indicator
         U __intel_cpu_indicator_init
         U __intel_new_proc_init
         U _intel_fast_memset
         U acoustic4
         U calloc
         U defineSource
         U elastic4
0000000000000000 a fdelmodc.c
         U free
         U getParameters
         U getRecTimes
         U getparint
         U initargs
0000000000000000 T main
0000000000002560 t main.A
0000000000000028 t main.J
         U malloc
         U readModel
         U requestdoc
0000000000000000 D sdoc
         U taperEdges
         U viscoacoustic4
         U viscoelastic4
         U vmess
         U wallclock_time
         U writeRec
         U writeSnapTimes
```

## Linking

```
> icc -O3 -w1 -DLINUX -I/home/thorbcke/include
-axW -static -o fdelmodc fdelmodc.o acoustic4.o
viscoacoustic4.o elastic4.o viscoelastic4.o
defineSource.o getParameters.o getWaveletInfo.o
getModelInfo.o applySource.o getRecTimes.o
writeSnapTimes.o writeRec.o fileOpen.o recvPar.o
readModel.o taperEdges.o verbosepkg.o SU2su.o
gaussGen.o spline3.o wallclock_time.o name_ext.o
atopkge.o docpkge.o getpars.o -L/home/thorbcke/lib
-lgenfft -lm
```

## Intel Compiler Options

- Vectorisation for fast code
  - -O3 -axP (version 10.x)
  - -O3 -mavx (version 11.x) -xAVX (version > 13)
  - -march=core-avx2
    -qopt-report -qopt-report-annotate -qopt-report-phase=all
     for (non)-vectorisation loop optimisation info

- Auto Parallelisation
  - -parallel
  - #pragma ivdep or #pragma simd

- OpenMP Parallelisation
  - -qopenmp (also during linking)

## GNU Compiler Options

- Vectorisation for fast code
  - -O3 -ffast-math -ftree-vectorize
  - -mavx2 -mfpmath=sse -march=broadwell
  - -fopt-info-vec (for vectorisation loops info)
  - -fopt-info-vec-missed (for non-vectorisation loops info)

- Aliasing of pointer (in C)
  - -fstrict-aliasing
  - #pragma GCC ivdep

- OpenMP Parallelisation
  - -fopenmp (also during linking)

## Compiler listing

```
76.  1-----<  for (ix=ioXx; ix<nx+1; ix++) {
77.  1 2---<    for (iz=ioXz; iz<nz+1; iz++) {
78.  1 2            vx[ix*n1+iz] += rox[ix*n1+iz]*(
79.  1 2            c1*(p[ix*n1+iz]      - p[(ix-1)*n1+iz]) +
80.  1 2            c2*(p[(ix+1)*n1+iz] - p[(ix-2)*n1+iz]));
81.  1 2--->    }
82.  1-----> }
```

```
CC-6290 CC: VECTOR File = acoustic4.c, Line = 76
  A loop was not vectorized because a recurrence was found between "p" and
"vx" at line 78.
CC-6308 CC: VECTOR File = acoustic4.c, Line = 77
  A loop was not vectorized because the loop initialization would be too
costly.
```

## helping compiler with `ivdep`

```
76.              #pragma ivdep
77.   1------< for (ix=ioXx; ix<nx+1; ix++) {
78.   1          #pragma ivdep
79.   1 Vr4--<    for (iz=ioXz; iz<nz+1; iz++) {
80.   1 Vr4          vx[ix*n1+iz] += rox[ix*n1+iz]*(
81.   1 Vr4          c1*(p[ix*n1+iz]      - p[(ix-1)*n1+iz]) +
82.   1 Vr4          c2*(p[(ix+1)*n1+iz] - p[(ix-2)*n1+iz]));
83.   1 Vr4-->    }
84.   1------> }
CC-6294 CC: VECTOR File = acoustic4.c, Line = 77
  A loop was not vectorized because a better candidate was found at line 79.
CC-6005 CC: SCALAR File = acoustic4.c, Line = 79
  A loop was unrolled 4 times.
CC-6204 CC: VECTOR File = acoustic4.c, Line = 79
  A loop was vectorized.
```

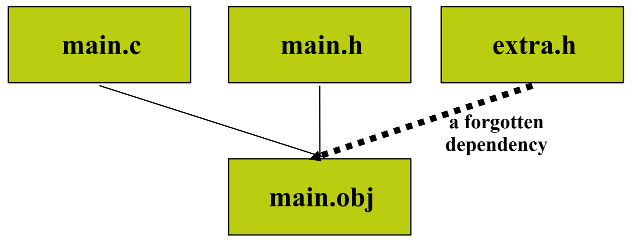
## same with global flag `-h restrict=a`

```
76.  1-----<  for (ix=ioXx; ix<nx+1; ix++) {
77.  1 Vr4-<    for (iz=ioXz; iz<nz+1; iz++) {
78.  1 Vr4          vx[ix*n1+iz] += rox[ix*n1+iz]*(
79.  1 Vr4          c1*(p[ix*n1+iz]      - p[(ix-1)*n1+iz]) +
80.  1 Vr4          c2*(p[(ix+1)*n1+iz] - p[(ix-2)*n1+iz]));
81.  1 Vr4->    }
82.  1-----> }
```

```
CC-6254 CC: VECTOR File = acoustic4.c, Line = 76
  A loop was not vectorized because a recurrence was found on "vx" at line 78.
CC-6005 CC: SCALAR File = acoustic4.c, Line = 77
  A loop was unrolled 4 times.
CC-6204 CC: VECTOR File = acoustic4.c, Line = 77
  A loop was vectorized.
```

## Librarian

- Under unix it is a **Makefile**
- Under windows controlled by Visual Studio (or the IDE)
- Java JDK does it for you
- Most problems arise when the specification is incorrect.



## Make

## Makefile

- Make is a utility to automatically building executables and libraries from source code.
- Makefile specifies how (options) to compile and link the targets (library executable) mentioned in the Makefile.
- Only rebuilds things that have changed

```
helloworld: helloworld.o
        cc -o $@ $<

helloworld.o: helloworld.c
        cc -c -o $@ $<

clean:
        rm -f helloworld helloworld.o
```

## Makefile variables

- CC=icc (C-compiler)
- CPP=icc (C++ compiler)
- FC=ifort (Fortran 90 compiler)

- CFLAGS=
- FFLAGS=
- LDFLAGS=

## Makefile Example

```
CC      = icc
CFLAGS  = -O1
PRG = floatCycl

SRC     = $(PRG).c \
        wallclock_time.c \

OBJ     = $(SRC:%.c=%.o)

.c.o: .c
        $(CC) $(CFLAGS) -c $<

$(PRG): $(OBJ)
        $(CC) $(CFLAGS) -o $(PRG) $(OBJ) $(LIBS) $(DEFINES) -

clean:
                rm -f core $(OBJ) $(PRG)
```

## Number Representations

## Representations

- What can be represented in N bits?
  - integers range 0 : 2^N
  - negative numbers?
- Floating point numbers
  - what about pi, sqrt(3) and 0.1?
  - very large numbers?
  - very small numbers?

## Signed Fixed Point Numbers

- For an 8-bit number, there are $2^8 = 256$ possible bit patterns. These bit patterns can represent negative numbers if we choose to assign bit patterns to numbers in this way. We can assign half of the bit patterns to negative numbers and half of the bit patterns to positive numbers.
- Four signed representations are discussed briefly:

  Signed Magnitude

  One's Complement

  Two's Complement

  Excess (Biased)

## Signed Magnitude

- Also know as "sign and magnitude," the leftmost bit is the sign (0 = positive, 1 = negative) and the remaining bits are the magnitude.
- Example:

$\mathbf{+25_{10} = 00011001_2}$

$\mathbf{-25_{10} = 10011001_2}$

$$\begin{array}{rcl} 2 & = & 0010 \\ -4 & = & 1100 \quad + \\ \hline -2 & \neq & 1110 \end{array}$$

Two representations for zero: $\mathbf{+0 = 00000000_2}$, $\mathbf{-0 = 10000000_2}$.

Largest number is $\mathbf{+127_{10}}$, smallest number is $\mathbf{-127_{10}}$, using an 8-bit representation.

## One's Complement

- The leftmost bit is the sign (0 = positive, 1 = negative). Negative of a number is obtained by subtracting each bit from 2 (essentially, complementing each bit from 0 to 1 or from 1 to 0). This goes both ways: converting positive numbers to negative numbers, and converting negative numbers to positive numbers.

- Example:

**$+25_{10}$ = $00011001_2$**
**$-25_{10}$ = $11100110_2$**

```
 2 = 0010
-4 = 1011  +
-2 = 1101
```

- Two representations for zero: **+0 = $00000000_2$, -0 = $11111111_2$.**
- Largest number is **$+127_{10}$**, smallest number is **$-127_{10}$**, using an 8-bit representation.

## Two's Complement

- The leftmost bit is the sign (0 = positive, 1 = negative). Negative of a number is obtained by adding 1 to the one's complement negative. This goes both ways, converting between positive and negative numbers. Addition of positive and negative numbers works the same way.

- Example (recall that **$-25_{10}$** in one's complement is **$11100110_2$**):

**$+25_{10}$ = $00011001_2$**
**$-25_{10}$ = $11100111_2$**

```
 2 = 0010
-4 = 1100  +
-2 = 1110
```

- One representation for zero: **+0 = $00000000_2$, -0 = $00000000_2$.**
- Largest number is **$+127_{10}$**, smallest number is **$-128_{10}$**, using an 8-bit representation.

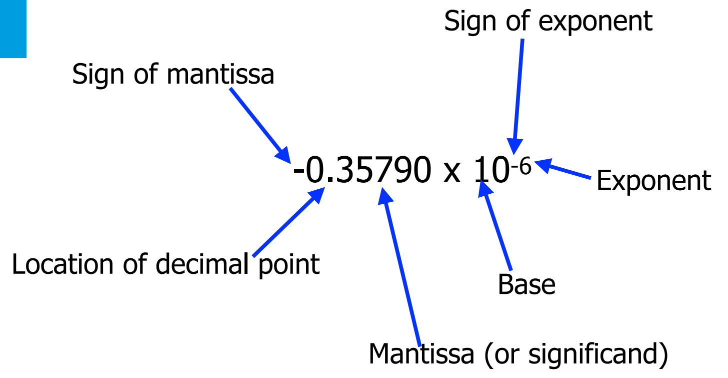
## Excess (Biased)

- The leftmost bit is the sign (usually 1 = positive, 0 = negative). Positive and negative representations of a number are obtained by adding a bias to the two's complement representation. This goes both ways, converting between positive and negative numbers. The effect is that numerically smaller numbers have smaller bit patterns, simplifying comparisons for floating point exponents.
- Example (excess 128 "adds" 128 ($2^8$) to the two's complement version, ignoring any carry out of the most significant bit) :

**$+12_{10}$ = $10001100_2$**
**$-12_{10}$ = $01110100_2$**

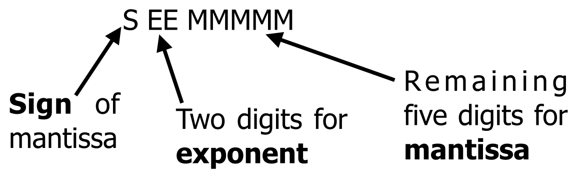
- One representation for zero: **+0 = $10000000_2$, -0 = $10000000_2$.**
- Largest number is **$+127_{10}$**, smallest number is **$-128_{10}$**, using an 8-bit representation.

## Floating-point Numbers



## Floating-point Format in bytes

- **Sign** the digit representing the sign of the mantissa
- **Mantissa** representing the mantissa
  - more bits for mantissa increases accuracy
- **Exponents** are digits representing the exponents
  - more bits in exponents increases the range
  - usually stored in the excess-N notation to avoid sign representation.
- Example 8 digit decimal



## Normalisation

- The base 10 number **254** can be represented in floating point form as **254** $\times$ **10$^0$**, or equivalently as:

  **25.4** $\times$ **10$^1$**, or

  **2.54** $\times$ **10$^2$**, or

  **.254** $\times$ **10$^3$**, or

  **.0254** $\times$ **10$^4$**, or

  infinitely many other ways, which creates problems when making comparisons, with so many representations of the same number.
- Floating point numbers are usually **normalised**, in which the radix point is located in only one possible position for a given number.
- Usually, but not always, the normalised representation places the radix point immediately to the left of the leftmost, nonzero digit in the fraction, as in: **.254** $\times$ **10$^3$.**

## Floating Point Arithmetic decimal

- Example: Add 9.999 × 10$^1$ and 1.610 × 10$^{-1}$ assuming 4 decimal digits

  align decimal point of number with smaller exponent
  1.610 × 10-1 = 0.0161 × 10$^1$

  add significands:
  ```
    9.999
    0.0161 +
   10.015 => one digit lost due to shifting,
  ```

  shift sum to put in normalized form 1.0015 × 10$^2$

  significand has only 4 digits round the sum => 1.002 × 10$^2$

## Floating-point Format

- There are two ways to convert p to the right length.
- **Chopping**, simply discard superfluous bits of p
- **Rounding**, superfluous bits of p are discarded, but if the first bit discarded is a 1 then 1 is added to the value of p

## Floating-point Format

- Floating point numbers uses a fixed number of bits
- This representation may result in error
- There are two measures of error
  - Absolute error
  - Relative error
- In numerical code errors can accumulate/propagate and at the end produce wrong answers.

## Exercise: Accuracy

- Simple programs to show the accuracy of floating point numbers.
- On your git clone: cd HPCourse/FloatPrecision
- Check the README for instructions.
- What have you learned from these examples?

## example precision

```
int main(int argc, char *argv[])
{
  float a;
  double c, d;
  int i;

  a=1.0;
  d=1.0;
  c=1.0;
  fprintf(stderr,"a=%f c=%g\n",a, c);

  for (i=0; i<10; i++) {
    c = 0.1*c;
    a = a + c;
    d = d + c;
    fprintf(stderr,"i=%d a=%10.9f c=%10.9g d=%16.14g \n",i, a, c, d);
  }

  return 0;
}
```

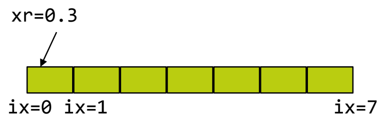
## running example

```
c86:FloatPrecision jan$ gcc float.c
c86:FloatPrecision jan$ ./a.out
a=1.000000 c=1
i=0 a=1.10000002384186 c=0.10000000000000 d=1.10000000000000
i=1 a=1.11000001430511 c=0.01000000000000 d=1.11000000000000
i=2 a=1.11100006103516 c=0.00100000000000 d=1.11100000000000
i=3 a=1.11110007762909 c=0.00010000000000 d=1.11110000000000
i=4 a=1.11111009120941 c=0.00001000000000 d=1.11111000000000
i=5 a=1.11111104488373 c=0.00000100000000 d=1.11111100000000
i=6 a=1.11111116409302 c=0.00000010000000 d=1.11111110000000
i=7 a=1.11111116409302 c=0.00000001000000 d=1.11111111000000
i=8 a=1.11111116409302 c=0.00000000100000 d=1.11111111100000
i=9 a=1.11111116409302 c=0.00000000010000 d=1.11111111110000
```

## mistake due to rounding

- would like to place receivers with a fixed distance:

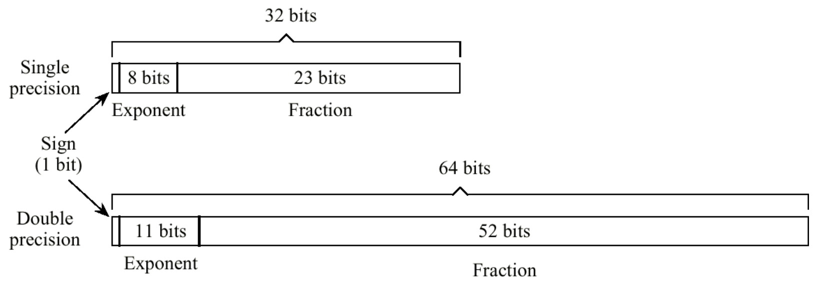
```
for (ir=0; ir<nrcv; ir++) {
    xr[nrec]=xrcv1[iarray]-sub_x0+ir*dxrcv;
    zr[nrec]=zrcv1[iarray]-sub_z0+ir*dzrcv;

    x[nrec]=NINT((rec->xr[nrec])/dx);
    z[nrec]=NINT((rec->zr[nrec])/dz);
    nrec++;
}
```



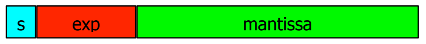
## mistake due to rounding

```
(base) JansMac:FloatPrecision jan$ ./a.out dx=3.2 dxrcv=12.8 x0=-1000
dx=3.200000 dz=3.200000 dxrcv=12.800000 dzrcv=0.000000
xr=526.399963 zr=1.600000 at grid coordinates ix=164 iz=1
xr=539.200012 zr=1.600000 at grid coordinates ix=169 iz=1
xr=590.399963 zr=1.600000 at grid coordinates ix=184 iz=1
xr=603.200012 zr=1.600000 at grid coordinates ix=189 iz=1
xr=654.399963 zr=1.600000 at grid coordinates ix=204 iz=1
xr=667.200012 zr=1.600000 at grid coordinates ix=209 iz=1
xr=718.399963 zr=1.600000 at grid coordinates ix=224 iz=1
xr=731.200012 zr=1.600000 at grid coordinates ix=229 iz=1
xr=782.399963 zr=1.600000 at grid coordinates ix=244 iz=1
xr=795.200012 zr=1.600000 at grid coordinates ix=249 iz=1
```

## IEEE-754 Floating Point Formats

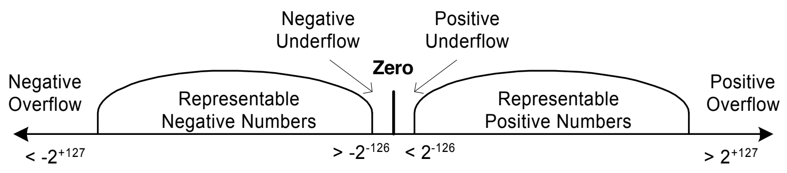


## IEEE-754



- **Exponent**: bias 127 integer
  - actual exponent = exp-127 with 0<exp<255
- **Mantissa**:
  - sign + magnitude, normalized binary significand with hidden integer bit: 1.mantissa
- N = $(-1)^s\ 2^{(E-127)}$ 1.M
- 1.0 = 0 0111 1111 1.0...0 = $(-1)^0 * 2^{(127-127)} *$ 1.M
- 0.5 = 0 0111 1110 1.0...0 = $(-1)^0 * 2^{(126-127)} *$ 1.M

## Range



## IEEE-754 Examples

| | Value | Sign | Exponent | Fraction |
|---|---|---|---|---|
| | | Bit Pattern | | |
| (a) | $+1.101 \times 2^5$ | 0 | 1000 0100 | 101 0000 0000 0000 0000 0000 |
| (b) | $-1.01011 \times 2^{-126}$ | 1 | 0000 0001 | 010 1100 0000 0000 0000 0000 |
| (c) | $+1.0 \times 2^{127}$ | 0 | 1111 1110 | 000 0000 0000 0000 0000 0000 |
| (d) | $+0$ | 0 | 0000 0000 | 000 0000 0000 0000 0000 0000 |
| (e) | $-0$ | 1 | 0000 0000 | 000 0000 0000 0000 0000 0000 |
| (f) | $+\infty$ | 0 | 1111 1111 | 000 0000 0000 0000 0000 0000 |
| (g) | $+2^{-128}$ | 0 | 0000 0000 | 010 0000 0000 0000 0000 0000 |
| (h) | $+\text{NaN}$ | 0 | 1111 1111 | 011 0111 0000 0000 0000 0000 |
| (i) | $+2^{-128}$ | 0 | 011 0111 1111 | 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 |

## IEEE 754 Standard

| Parameter | Single Precision | Single Extended | Double Precision | Double Extended |
|---|---|---|---|---|
| Word width (bits) | 32 | >= 43 | 64 | >= 79 |
| Exponent width (bits) | 8 | >= 11 | 11 | >= 15 |
| Exponent bias | 127 | Unspecified | 1023 | Unspecified |
| Max exponent | 127 | >= 1023 | 1023 | >= 16383 |
| Min exponent | -126 | <= -1022 | -1022 | <= -16382 |
| Number range (base 10) | $10^{-38}$, $10^{+38}$ | Unspecified | $10^{-308}$, $10^{+308}$ | Unspecified |
| Mantissa width (bits) | 23 | >= 31 | 52 | >= 63 |
| No. of exponents | 254 | Unspecified | 2046 | Unspecified |
| No. of fractions | $2^{23}$ | Unspecified | $2^{52}$ | Unspecified |
| No. of values | $1.98 \times 2^{31}$ | Unspecified | $1.98 \times 2^{63}$ | Unspecified |

## ASCII Character Code

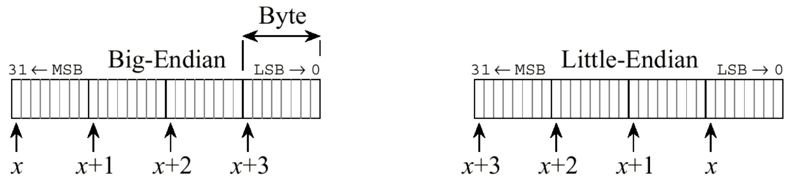
- ASCII is a 7-bit code, commonly stored in 8-bit bytes.
- "A" is at $41_{16}$. To convert upper case letters to lower case letters, add $20_{16}$. Thus "a" is at $41_{16} + 20_{16} = 61_{16}$.
- The character "5" at position $35_{16}$ is different than the number 5. To convert character-numbers into number-numbers, subtract $30_{16}$: $35_{16} - 30_{16} = 5$.

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | NUL | 10 | DLE | 20 | SP | 30 | 0 | 40 | @ | 50 | P | 60 | ` | 70 | p |
| 01 | SOH | 11 | DC1 | 21 | ! | 31 | 1 | 41 | A | 51 | Q | 61 | a | 71 | q |
| 02 | STX | 12 | DC2 | 22 | " | 32 | 2 | 42 | B | 52 | R | 62 | b | 72 | r |
| 03 | ETX | 13 | DC3 | 23 | # | 33 | 3 | 43 | C | 53 | S | 63 | c | 73 | s |
| 04 | EOT | 14 | DC4 | 24 | $ | 34 | 4 | 44 | D | 54 | T | 64 | d | 74 | t |
| 05 | ENQ | 15 | NAK | 25 | % | 35 | 5 | 45 | E | 55 | U | 65 | e | 75 | u |
| 06 | ACK | 16 | SYN | 26 | & | 36 | 6 | 46 | F | 56 | V | 66 | f | 76 | v |
| 07 | BEL | 17 | ETB | 27 | ' | 37 | 7 | 47 | G | 57 | W | 67 | g | 77 | w |
| 08 | BS | 18 | CAN | 28 | ( | 38 | 8 | 48 | H | 58 | X | 68 | h | 78 | x |
| 09 | HT | 19 | EM | 29 | ) | 39 | 9 | 49 | I | 59 | Y | 69 | i | 79 | y |
| 0A | LF | 1A | SUB | 2A | * | 3A | : | 4A | J | 5A | Z | 6A | j | 7A | z |
| 0B | VT | 1B | ESC | 2B | + | 3B | ; | 4B | K | 5B | [ | 6B | k | 7B | { |
| 0C | FF | 1C | FS | 2C | ´ | 3C | < | 4C | L | 5C | \ | 6C | l | 7C | \| |
| 0D | CR | 1D | GS | 2D | - | 3D | = | 4D | M | 5D | ] | 6D | m | 7D | } |
| 0E | SO | 1E | RS | 2E | . | 3E | > | 4E | N | 5E | ^ | 6E | n | 7E | ~ |
| 0F | SI | 1F | US | 2F | / | 3F | ? | 4F | O | 5F | _ | 6F | o | 7F | DEL |

| | | | | | |
|---|---|---|---|---|---|
| NUL | Null | FF | Form feed | CAN | Cancel |
| SOH | Start of heading | CR | Carriage return | EM | End of medium |
| STX | Start of text | SO | Shift out | SUB | Substitute |
| ETX | End of text | SI | Shift in | ESC | Escape |
| EOT | End of transmission | DLE | Data link escape | FS | File separator |
| ENQ | Enquiry | DC1 | Device control 1 | GS | Group separator |
| ACK | Acknowledge | DC2 | Device control 2 | RS | Record separator |
| BEL | Bell | DC3 | Device control 3 | US | Unit separator |
| BS | Backspace | DC4 | Device control 4 | SP | Space |
| HT | Horizontal tab | NAK | Negative acknowledge | DEL | Delete |
| LF | Line feed | SYN | Synchronous idle | | |
| VT | Vertical tab | ETB | End of transmission block | | |

## Big-Endian and Little-Endian Formats

- In a byte-addressable machine, the smallest datum that can be referenced in memory is the byte. Multi-byte words are stored as a sequence of bytes, in which the address of the multi-byte word is the same as the byte of the word that has the lowest address.
- When multi-byte words are used, two choices for the order in which the bytes are stored in memory are: most significant byte at lowest address, referred to as ***big-endian***, or least significant byte stored at lowest address, referred to as ***little-endian***.



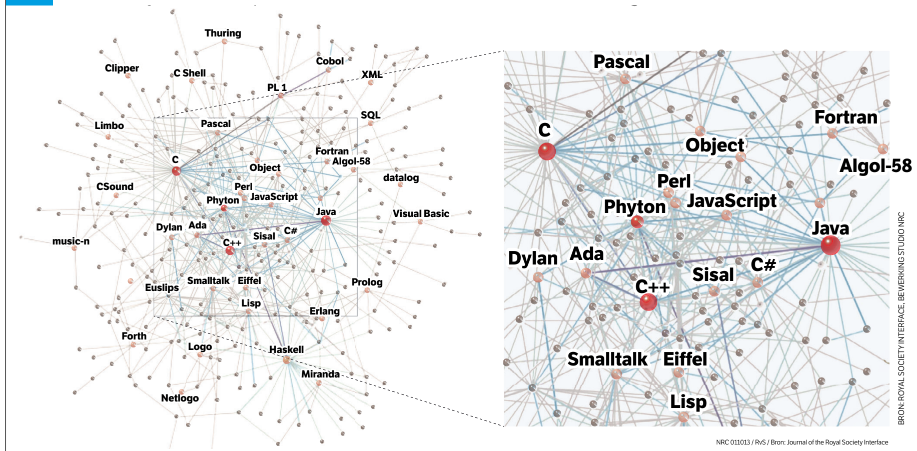
Word address is $x$ for both big-endian and little-endian formats.

## Programming Languages

## Imperative language types

- **Procedural** languages
  - Traditional languages where code is a sequence of instructions executed (approximately) in sequence
  - BASIC, FORTRAN, PASCAL, C
  - Code can become 'spaghetti' for large tasks unless procedural methods are added
- **Object-oriented** languages
  - Objects contain data and the methods to work on that data
  - Extensive use of message passing
  - C++, Java, FORTRAN90
  - Good for large projects due to concepts such as encapsulation and inheritance

## Evolution of programming languages



## Which languages are there, … and suited for/use in HPC

| | |
|---|---|
| • FORTRAN | • ~~COBOL~~ |
| • FORTRAN90 | • ~~PASCAL~~ |
| • ~~BASIC~~ | • ~~Delphi~~ |
| • C | • Perl |
| • ~~ADA~~ | • ~~TCL~~ |
| • C++ | • Python |
| • ~~C#~~ | • Linda |
| • ~~JAVA~~ | • ~~Smalltalk~~ |
| • ~~IDL~~ | • ~~LISP~~ |
| • ~~Prolog~~ | • ~~MATLAB~~ |

## C

- Flexible – a general-purpose language
- Efficient for everything
- Huge amount of code exists
- Complex data structures are simple to create
- Very portable
- Huge range of libraries/add-ons available

- Lowish-level - a single command doesn't do much
- Can be hard to read
- C lets you do bad things, C++ is pickier
- 'pointers' are a source of confusion for beginners

## Fortran 77/90/95/2008

- Designed for scientific programming
- Efficient for numerical work (complex type)
- Huge amount of code exists
- Good for conversion to parallel environments

- COMMON blocks, computed GOTOs (77) are crashes waiting to happen
- No dynamic memory handling (77)
- No low-level hardware access (playing with bytes tricky)
- Often code relies on 'standard' extensions to the language (which are not standard at all!)
- FORTRAN90 adds object-oriented programming to a language never intended to have it.

## C++

- C with added object-oriented capabilities
- Usual benefits of OOP
- Good for large collaborative programming projects
- Standard Template Library provides complex objects
  - vectors (slow, array bounds checking)
  - maps
  - stacks
  - algorithms

- Easy to write 'C with objects'
- Can be hard to read for class details
- C lets you do bad things, C++ is pickier
- C++ compilers not that good in optimization
- not that well suited for development of numerical code

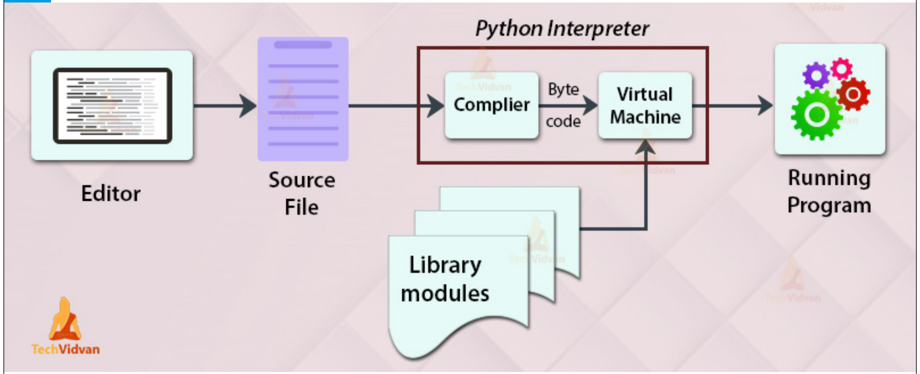
## C++ and performance

- If you want to use C++ and write good performing code. Here are some guidelines:
  - avoid the creation of unexpected copies, which cause constructor calls
  - When passing pointers, use the restrict keyword whenever appropriate; this allows the compiler to do optimizations
  - test your code with another compiler than g++
  - operator overloading be used very carefully (it can break fused multiply-add operations, and SSE/AVX optimization)

## Python

- High level programming language
  - intepreted : no compilation needed
- Portable, large standard and third party libraries, widely used.
- Can write kernels in C/Fortran:
  - Numba is a just-in-time (JIT) compiler for Python that works best on code that uses NumPy arrays and functions
- numpy, scipy, matplotlib are great tools to use.

## Python interpreter



## Java

- Completely object-oriented
- C++ with the nasty/interesting bits taken out
- No operator overloading
- No templates
- Very portable due to 'virtual machine'
- Can create graphical applications (or web applets) relatively easily

- Slow due to virtual machine
- Not much scientific programming done in it
- Keeps changing!

## Scripting

Example of why scripting helps:

- I have a program which reads a file containing 100 integers, sums them, and prints the answer
- I have 200 files like that, and I want to process them all

Do I …

- (a) run my program 100 times, typing in a different file name each time?
- (b) change the program so it reads a list of files and processes them each in turn
- (c) write a script which feeds each file in turn to the existing program

## Scripting languages: Shell

- C-shell (bash)
  - the standard scripting language for UNIX
  - uses the same commands you'll use on the command line
  - only integer arithmetic, but floats with a trick
  - Text handling is tricky

## Programming in C and Fortran

## Guidelines from David Parnas

The essence of good software construction are clean interfaces between the components, so that one could rewrite one without having to modify the other.

One bad programmer can easily create two new jobs a year. Hiring more bad programmers will just increase our perceived need for them. If we had more good programmers, and could easily identify them, we would need fewer, not more.

Doing it right is hard work. Shortcuts lead you in the wrong direction and they often lead to disaster.

## General structure

- Main program to call functions/subroutines use objects.
- Don't try to write one big program.
- Use functions/subroutines/objects to handle specific parts:
    - parameters handling
    - File IO
    - computational part
- Maintainability, also by other people.
- Don't hesitate to **rewrite** (big) parts of your code for better structure or performance (even after 10 years of successful usage).

## Function / Subroutine / Object

- Has a well defined scope within your program.
- Could be reused by other programs. Typically IO, parameters definition, compute kernels
- If it becomes too big to handle, try to split it in multiple parts.
- In the design (find out) phase try to avoid to do object oriented programming right from the start.

## General names

- Use self-explanatory names for variables and functions:
    - ix, iy for loops over x and y directions
    - GeophonePosition: might be a bit too long for a loop variable
    - ComputeGreen(): too general for a function
        - GreenLayeredMedia()
- Do not overdo it.
- CaMel notation
- use_of_underscores

## C program instructions

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

Loop

```
if (a<eps){
}
```

Branch

```
A=(float *)malloc(N*sizeof(float));
free(A);
```

Memory

```
fp=fopen(filename, 'w+');
fwrite (data, size, nelem, fp);
fread(data, size, nelem, fp);
flose(fp);
```

IO

## C structures

- A structure is a (logical) collection of variables and arrays

```
typedef struct _waveletPar { /* Wavelet */
      char *file_src;
      int nt;
      int nx;
      float dt;
      float fmax;
      int random;
} wavPar;
```

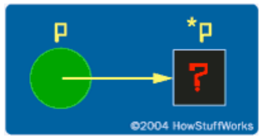
- Another example is the segy header (defined in segy.h)

## Lets look at an Example

- ....

## C-pointers

```
int *p;
p = (int *)malloc(sizeof(int));
```



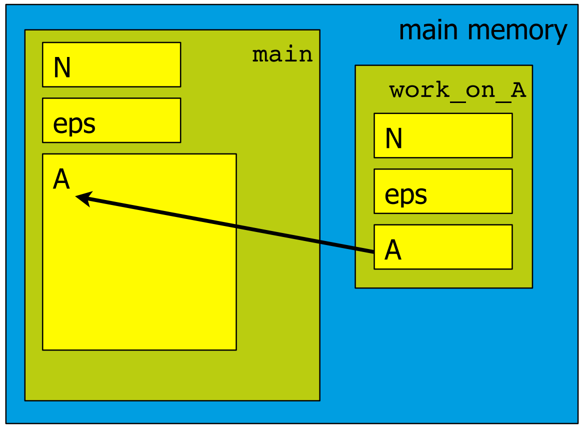
## C-pointers

- Why do we need and use pointers?
  - Function cannot return more than one value. But the same function can modify many pointer variables and function as if it is returning more than one variable.
  - Pointers allow you to dynamically request memory to store off information for use later. They allow you to create linked lists and other algorithmically oriented data structures.
  - The raw ability to work with particular memory locations is a useful option to have.

## C-pointers

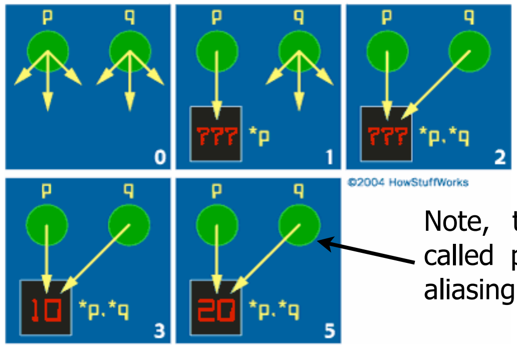
```
float *A;
int N=10000;

A = (float *)malloc(N*sizeof(float));

work_on_A(A, N, eps);

int work_on_A(float *array, int N, float eps)
{
    for (i=0; i<N; i++){
        array[i] = ....;
     }
     eps = 1e-6;
}
```

## C-pointers



## C-pointers

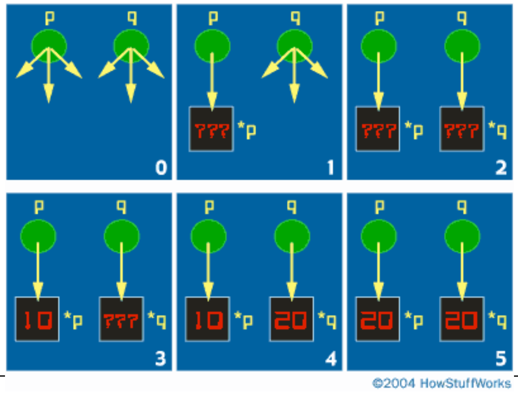
```
int *p, *q;
  p = (int *)malloc(sizeof(int));
  q = p;
  *p = 10;
  *q = 20;
```



Note, this is called pointer aliasing

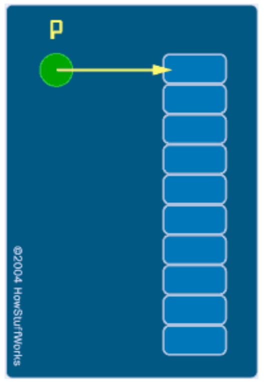
## C-pointers

```
int *p, *q;
  p = (int *)malloc(sizeof(int));
  q = (int *)malloc(sizeof(int));
  *p = 10;
  *q = 20;
  *p = *q;
```



## Pointers to arrays

```
int *p;
int i;
p = (int *)malloc(10*sizeof(int));
for (i=0; i<10; i++)
        p[i] = 0; (or *(p++) = 0;)
free(p);
```



## C pointers

| | Address | Value |
|---|---|---|
| ptr | 0x10110F34 | 7 |
| | 0x10110F38 | undefined |
| | 0x10110F3C | 10 |

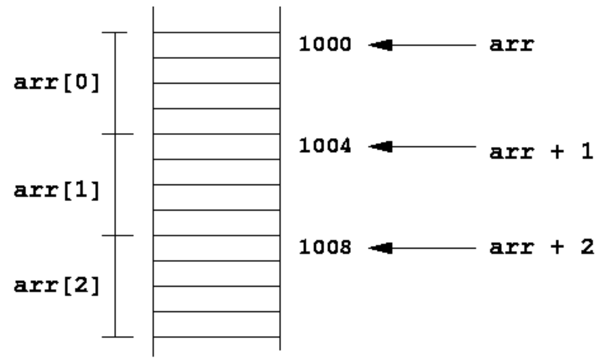
```
arr[0] = 7;
ptr  = &arr[0];
*(ptr+2) = 10
```

Pointer shift with the element size

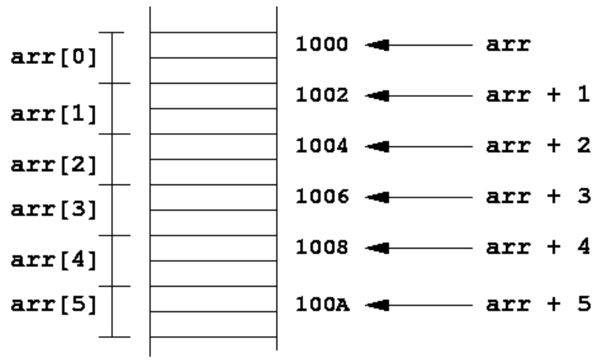
in git HPCourse/ptrs

## Pointer shift with the element size

```
int arr[10];
```



```
short arr[10];
```



## Arrays in C

- array index starts numbering at position **zero**
- arrays are contiguous in memory
- 2D array for example A[3][6]

| Address | Value |
|---|---|
| 0x00000000 | A[0][0] |
| .... | ... |
| 0x00000014 | A[0][5] |
| 0x00000018 | A[1][0] |

## Arrays in Fortran

- array index starts numbering at position **one**
- arrays are contiguous in memory
- 2D array for example A(3,6) (row,column)

| Address | Value |
|---|---|
| 0x00000000 | A(1,1) |
| .... | ... |
| 0x00000014 | A(3,2) |
| 0x00000018 | A(1,3) |

## Note on Matrix Storage

- A matrix is a 2-D array of elements, but memory addresses are "1-D"
- Conventions for matrix layout
  - by column, or "column major" (Fortran default); A(i,j) at A+i+j*n
  - by row, or "row major" (C default) A[i][j] at A+i*n+j

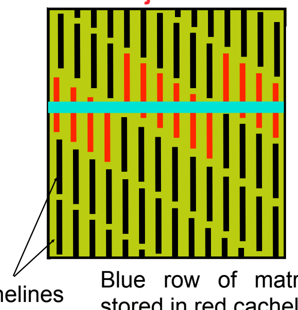
Column major

| | | | |
|---|---|---|---|
| 0 | 5 | 10 | 15 |
| 1 | 6 | 11 | 16 |
| 2 | 7 | 12 | 17 |
| 3 | 8 | 13 | 18 |
| 4 | 9 | 14 | 19 |

Row major

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 |

Column major matrix in memory



Blue row of matrix is stored in red cachelines

Figure source: Larry Carter, UCSD

## Example 2D array allocation in C

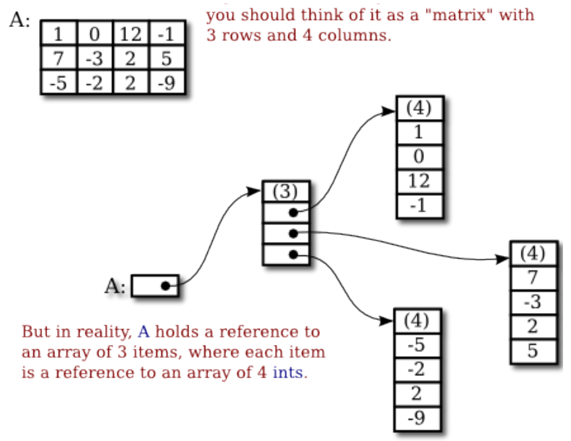
allocate a C-array with size 3x4

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |

```
int **p;
int i;
p = (int **)malloc(3*sizeof(*int));
for (i=0; i<3; i++)
     p[i] = (int *)malloc(4*sizeof(*int));

for (i=0; i<3; i++) {
   for (j=0; j<4; j++) {
        p[i][j] = …
   }
}
```

$$\begin{array}{c|cccc} & 1 & 2 & \cdots & n \\ \hline 1 & a_{11} & a_{12} & \cdots & a_{1n} \\ 2 & a_{21} & a_{22} & \cdots & a_{2n} \\ 3 & a_{31} & a_{32} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ m & a_{m1} & a_{m2} & \cdots & a_{mn} \end{array}$$

## Example 2D array allocation in C


A:
1 0 12 -1
7 -3 2 5
-5 -2 2 -9
you should think of it as a "matrix" with 3 rows and 4 columns.
A:
(3)
(4)
1
0
12
-1
(4)
7
-3
2
5
(4)
-5
-2
2
-9
But in reality, A holds a reference to an array of 3 items, where each item is a reference to an array of 4 ints.

## Question: is this safe coding, why?

```
// TAB -- The Ampersand BUG function
   // Returns a pointer to an int
   int* TAB() {
      int temp;
      return(&temp); // returns a pointer to the local
   int
   }

   void Victim() {
      int* ptr;
      ptr = TAB();
      *ptr = 42;
   }
```

see HPCourse/Pointers/ampersand.c

## Calling C from Fortran

- Fortran subroutines always pass pointers:
- C
```
getdata_(float *a, int *n1, int *n2);
```
- called in Fortran
```
integer n1, n2
real a(n1,n2)
call getData(a, n1, n2)
```

## Calling Fortran from C

- Fortran subroutines always pass pointers:
- Fortran
```
subroutine getData(a, n1, n2)
integer n1, n2
real a(n1,n2)
```
- C
```
int n1, n2;
float *a;
getdata_(a, &n1, &n2);
```

## Comparison C and Fortran 77/90

- dynamic memory allocation (C and F90)
- structures (C)
- Complex number type (Fortran and C99)
- pointers (C and F90)
- arrays:
  - C last index is fastest A[slowest][slower][fast]
  - Fortran first index is fastest A(fast, slower, slowest)
- start numbering, C at 0, Fortran at 1
- parameter passing to functions

## Profiling and Debugging

## Profiling applications

- Software level
  - Usually a recompile (-p -g flag) is needed to insert instrumentation instructions
  - after running the program a file ‘gmon.out’ is produced
  - ‘gprof a.out gprof.out’ for analysis (does not work on apple)
- Hardware level
  - uses the hardware counters which are build into the CPU
  - detailed information about flops, cache misses, TLB, ...
    - AMD: CodeAnalyst (free, requires patch in Linux kernel)
    - Intel: Vtune (not free, requires patch in Linux kernel)

## GNU profiling example

```
-bash-3.00$ gprof ../fdelmodc gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 97.02    136.90   136.90     3001    0.05     0.05  viscoelastic4
  2.82    140.88     3.98     3001    0.00     0.00  taperEdges
  0.06    140.97     0.09        1    0.09     0.09  readModel
  0.04    141.03     0.06      751    0.00     0.00  getRecTimes
  0.01    141.05     0.02                             byte32sse2
  0.01    141.06     0.01        3    0.00     0.00  getModelInfo
  0.01    141.07     0.01        1    0.01   141.05  main
```

## PAT (profiler on Cray hardware)

```
  100.0% | 3111 |Total
 |------------------------
 |  98.6% | 3066 |USER
 ||------------------------
 ||  88.5% | 2753 |elastic4
3||        |      | jan/FD/fdelmodc/elastic4.c
 ||||----------------------
4|||   21.0% |   654 |line.88
4|||   18.7% |   583 |line.100
4|||    6.6% |   205 |line.150
4|||   22.3% |   693 |line.154
4|||   19.8% |   617 |line.181
 ||||======================
 ||   9.5% |  294 |taperEdges
3||        |      | jan/FD/fdelmodc/taperEdges.c
 ||||----------------------
4|||    2.7% |    85 |line.23
4|||    1.6% |    50 |line.35
4|||    3.2% |    99 |line.47
4|||    1.9% |    60 |line.59
 ||========================
 |   1.4% |   45 |ETC
 |========================
```

## PAT (Cray hardware profile)

```
USER / elastic4
------------------------------------------------------------------------
  Samp%                                            88.5%
  Samp                                              2753
  DATA_CACHE_MISSES        38.009M/sec    1669361262 misses
  PAPI_TOT_INS           1353.653M/sec   59452452138 instr
  PAPI_L1_DCA            1132.779M/sec   49751666083 refs
  PAPI_FP_OPS             838.940M/sec   36846242855 ops
  User time (approx)       43.920 secs  105408000000 cycles
  HW FP Ops / User time   838.940M/sec   36846242855 ops  17.5%peak(DP)
  HW FP Ops / Inst                                 62.0%
  Computational intensity    0.35 ops/cycle     0.74 ops/ref
  Instr per cycle                                  0.56 inst/cycle
  MIPS                    1353.65M/sec
  MFLOPS (aggregate)       838.94M/sec
  Instructions per LD & ST  83.7% refs           1.19 inst/ref
  D1 cache hit,miss ratios  96.6% hits           3.4% misses
  D1 cache utilization (M)  29.80 refs/miss     3.725 avg uses
```

## Profiling tools

- Valgrind: http://valgrind.org
  - also good in detecting memory leakage problems
- OpenSpeedshop: http://www.openspeedshop.org/wp/
  - difficult to get installed
- GNU-gprof
  - no source line level
- Vtune (from Intel):

## Debugging

- compile with -g
  - usually optimisation level is lowered, so don’t forget to remove the -g flag after debugging.
- run program with special debug program
  - idb (intel) gdb (gnu) lldb (OSX) : command line
  - totalview, ddd : graphical user interface
- compile options
  - -traceback

## LAPACK BLAS

- The **BLAS** (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing basic vector and matrix operations.
- **LAPACK** (Linear Algebra PACKage) is written in Fortran90 and provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems.
- http://www.netlib.org/lapack/

## Using LAPACK BLAS

- All CPU manufacturers have optimised (assembler) implementations ons BLAS/LAPACK and most of them also provide optimised FFT's
- Intel: MKL (Math Kernel Library)
- AMD: AOCL (AMD Optimizing CPU Libraries)
  - BLIS (BLAS) , Libflame (~LAPACK)
- Nvidia: CUBLAS (BLAS for Nvidia GPU's)
- ATLAS (Automatically Tuned Linear Algebra Software)

## Other Libraries

- Numerical Recipes: http://www.nr.com/

– Equation solving
 – Fitting/minimisation
 – Random number generation (built-in functions are all poor!)
– written for functionality not for performance

- Netlib.org: http://www.netlib.org/
- Trilinos: http://trilinos.sandia.gov/
- NAG (Numerical Algorithm Group): http://www.nag.co.uk/

– as numerical recipes, but commercial
 – a good solution, IF you can understand the manual

- FFTW: http://www.fftw.org/

## Other Libraries

- PETSc http://www.mcs.anl.gov/petsc/petsc-as/

– scientific applications modeled by partial differential equations
– Portable to any parallel system supporting MPI
– mostly FEM

- Metis http://glaros.dtc.umn.edu/gkhome/views/metis

– partitioning unstructured graphs and hypergraphs and computing fill-reducing orderings of sparse matrices

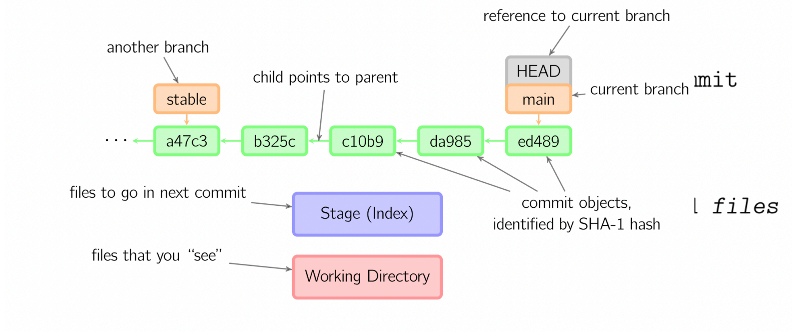
- Boost: http://www.boost.org

– C++ , general many algorithms and templates
– not specific for HPC

## Version Control

- Software development, management systems
- Makes is easy to develop files and keep track of the changes made in the source code.
- Check-in (working) versions of your code at regular times
- Needs some time to set set up
- https://git-scm.com : you are already working with it !
- http://subversion.apache.org/
- http://code.google.com/hosting

## GIT



http://onlywei.github.io/explain-git-with-d3/#

https://marklodato.github.io/visual-git-guide/index-en.html#basic-usage

## Exercise: Loop interchange

- To be discussed in the optimization lecture next week.
- On your git clone: cd HPCourse/LoopChange
- Check the README for instructions.
- Make sure you use a low optimisation level (-O0) for the compiler
- The Python code also has a loops related problem.

## Exercise Vectorization

- To be discussed in the optimization lecture next week.
- On your git clone: cd HPCourse/Vect
- Check the README for instructions.
- Look into the c-file and see which loop is (should be) vectorised by the compiler.
- This exercise will also give you some compile and linking experience.

## Exercise: Matrix Multiply

- To be discussed in the optimization lecture next week.
- On your git clone: cd HPCourse/MatMul
- Check the README for instructions.
- In the Makefile change LIBS= to a path where the BLAS library is.
- This exercise also requires some compile and linking experience.

## Exercise: Matrix Multiply + Unroll

- To be discussed in the optimization lecture next week.
- On your git clone: cd HPCourse/MatMulUnroll
- Check the README for instructions.
- In the Makefile change LIBS= to a path where the BLAS library is.
  if you can not find BLAS just set LIBS= (empty)
-

## Exercise: Debug

- fdelmodc with build in programming bugs and runtime errors.
- On your git clone: cd HPCourse/FD
- Check the README for instructions.
- If you try to run the code it will give an error, see if you can solve it using a debugger.
- Note this is not an easy exercise. It has two parts
  1. Lexical format problems in the source code
  2. runtime problems: an exercise to use a debugger

## Exercise: Profile

- Profile a program which you are using (in case you don't have one use fdelmodc of the previous exercise).
  - compile with -p -g (or -pg lookup the option of gcc)
  - run program 'a.out'
  - run 'gprof a.out gprof.out'
- If you did not succeed do no hesitate to ask me (by e-mail or during the course).