CS395T: Introduction to Scientific and Technical Computing

Instructors:

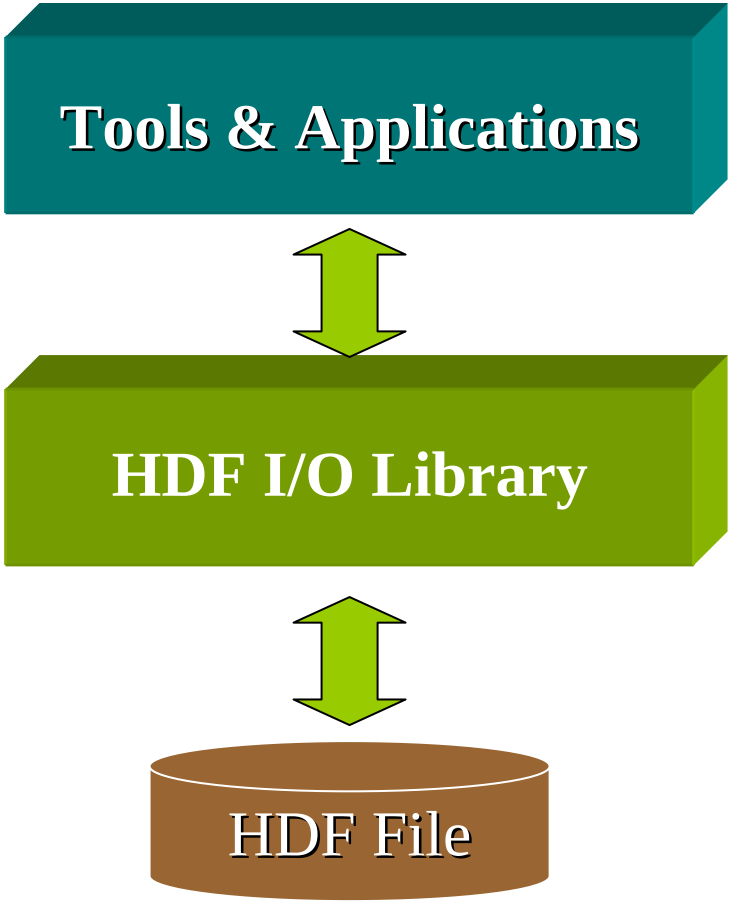
Dr. Karl W. Schulz, TACC
Dr. Victor Eijkhout, TACC

Outline

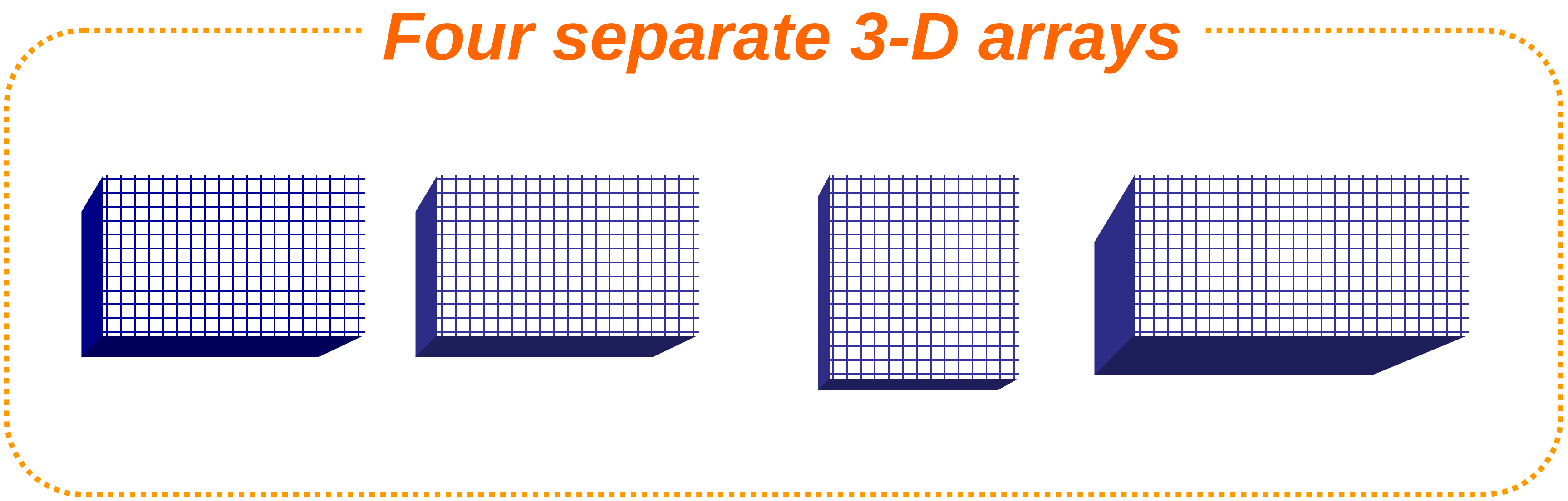
- Administrative Details
  - Grades for #2 to be posted by Thursday
  - Midterm Exam next Thursday
    - Architectures
    - UNIX Fundamentals
    - Compilers/Make/CVS/Batch Systems
- Scientific Data Representations
  - HDF
- Debugging
  - Motivation
  - Common signals
  - Basic tracing/debugging

What is HDF?

- HDF = Hierarchical Data Format
  - format and software for scientific data (developed to aid scientists and programmers in the storing, transfer and distribution of data sets)
  - stores images, multidimensional arrays, tables, etc. (*you can mix and match all of these structures in HDF5)*
  - emphasis on storage and I/O efficiency
  - free and commercial software support
  - emphasis on standards
  - users from many engineering and scientific fields

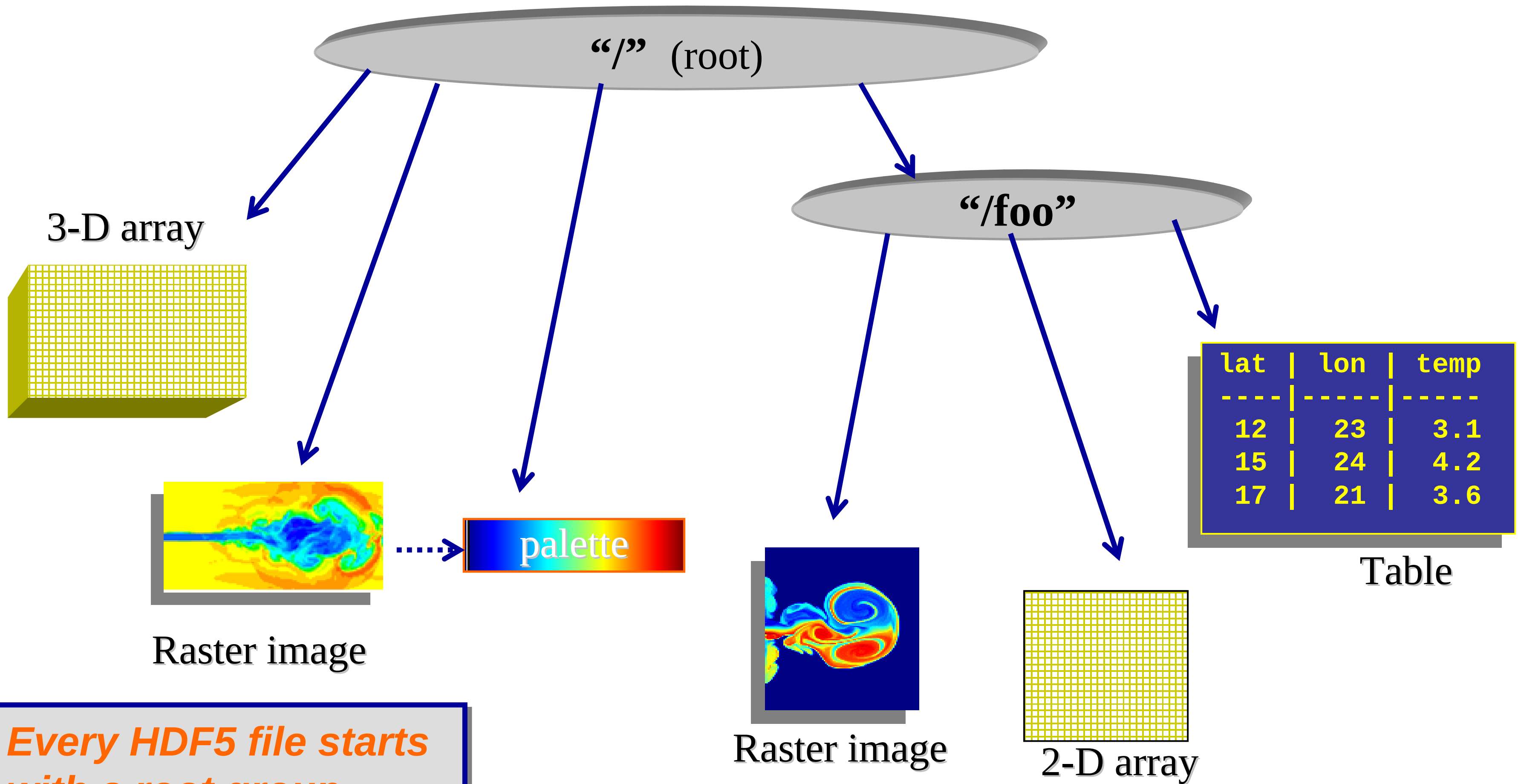


An HDF File:
A Collection of Scientific Data Objects



Typically include metadata to describe the objects

Example HDF5 File Contents



Every HDF5 file starts with a root group

HDF History

- HDF4 - Based on original 1988 version of HDF
  - backwards compatible with all earlier versions
  - 6 basic objects: *raster image*, *multidimensional array (SDS), palette, group (Vgroup), table (Vdata), annotation*
  - limits on file size (< 2GB) and number of objects (<20K)

- HDF5 - First released in 1998
  - new format(s) and library are ***not compatible*** with HDF4
  - includes only 2 basic primitive objects (groups and datasets)
    - No limit on the HDF5 file size and number of objects in the file
    - HDF5 file is portable across all computing platforms

HDF5 Supported Platforms

- **Solaris (32 and 64-bit)**
- **IRIX6.5 IRIX64-6.5**
- **HPUX 11.00**
- **AIX (32 and 64-bit modes)**
- **OSF1**
- **FreeBSD**
- **Linux (including 64-bit)**
- **Altix (SGI)**
- **IA-32 and IA-64**
- **Windows 2000, XP**
- **MAC OS X**
- **Crays (T3E, SV1, T90IEEE)**
- **DOE National Labs machines**
- **Linux Clusters**

HDF Supported Languages

- **C**
- **Wrappers:**
  - **C++**
  - **Fortran90**
  - **Java**
- **Vendors' compilers (SUN, IBM, HP, etc.)**
- **PGI, Intel, and Absoft (Fortran)**
- **GNU C (e.g. gcc 3.3.2)**

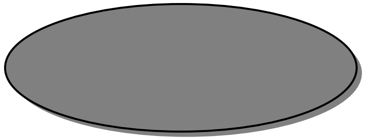
HDF5 File Objects (conceptual view)

- Containers for storing scientific data
  - Primary Objects:
    - Groups
    - Datasets
  - Secondary Objects:
    - Datatypes
    - Dataspaces
- Additional means to organize data
    - Attributes
    - Sharable objects
    - Storage and access properties

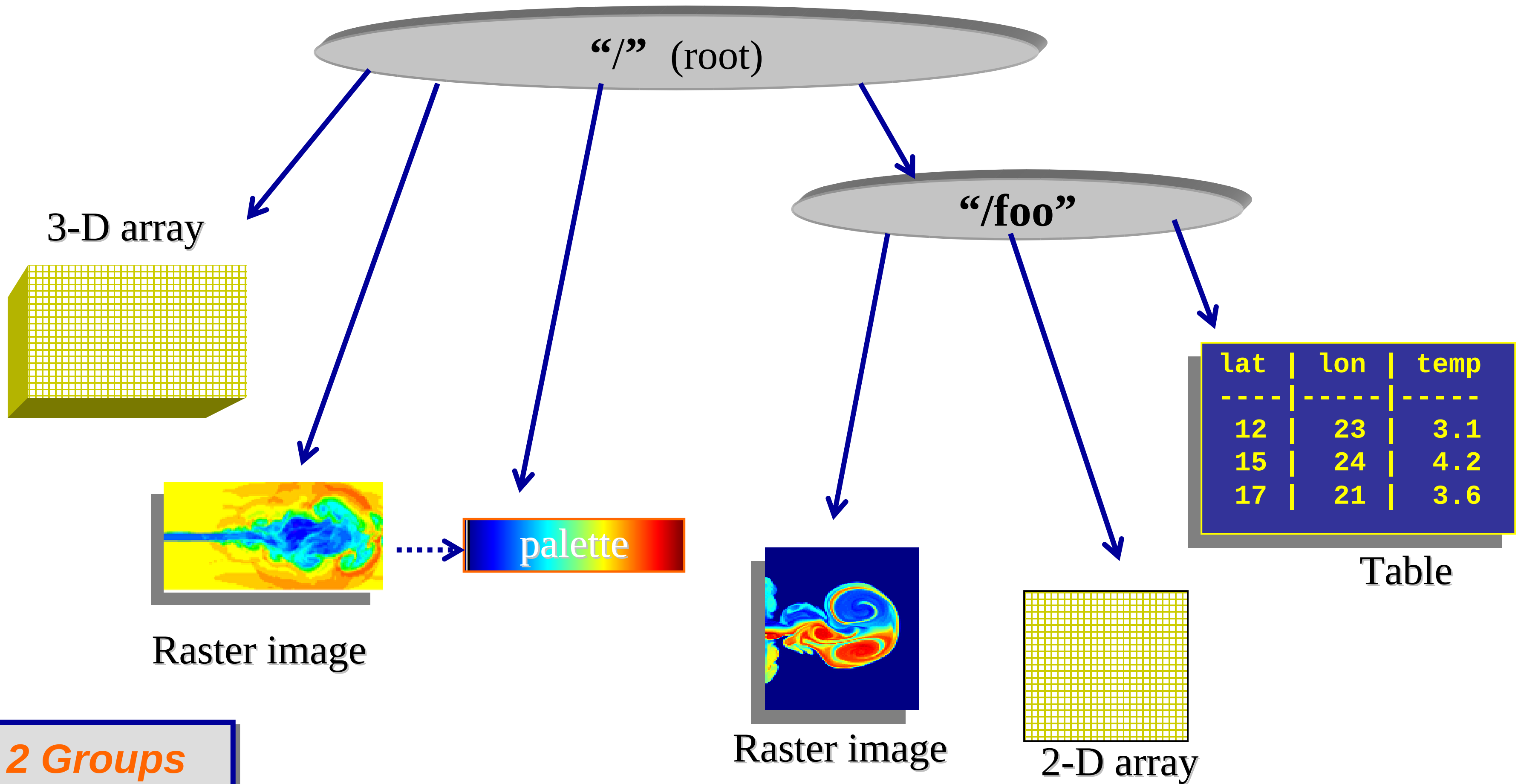
HDF5 Data Model

- Dataset
  - multidimensional array of elements, together with supporting metadata

- Group
  - directory-like structure containing datasets, groups, other objects



Example HDF5 File Contents



2 Groups

Components of an HDF Dataset

- Array
  - an ordered collection of identically typed data items distinguished by their indices (subscripts)
- Dataspace
  - information about the size and shape of a dataset array and selected parts of the array
- User-defined attribute list
- Special storage options
  - extendable, chunked, compressed, external

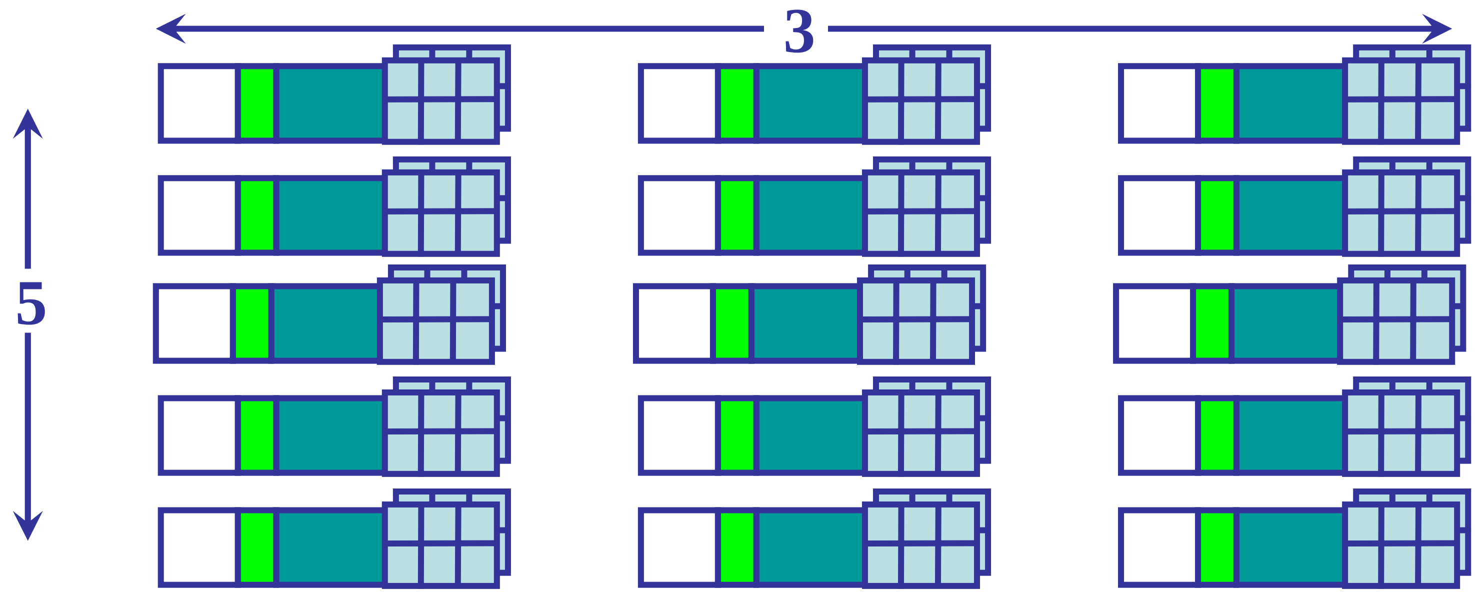
HDF5 Datatypes

- A datatype is
  - A classification specifying the interpretation of a data element
  - Specifies for a given data element:
    - the set of possible values it can have
    - the operations that can be performed
    - how the values of that type are stored

HDF5 Datatypes

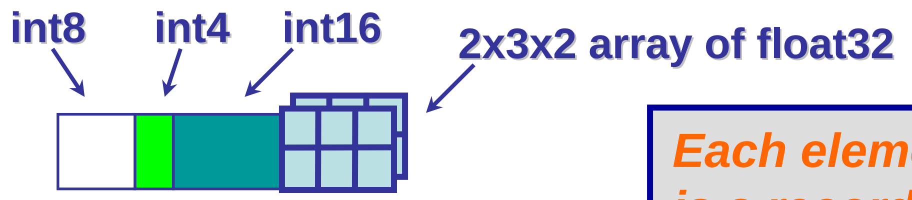
- Atomic types
  - standard integer & float
  - user-definable scalars (e.g. 13-bit integer)
  - variable length types (e.g. strings)
  - pointers - references to objects/dataset regions
  - enumeration - names mapped to integers
- Compound types
  - Comparable to C structs
  - Members can be atomic or compound types
  - Members can be multidimensional

HDF5 dataset: Example Array of Records



Dimensionality: 5 x 3

Datatype:



Record

Each element in 3D array is a record with 4 values

HDF Data Spaces

- A dataset *dataspace* describes the dimensionality of the dataset. Dimensions of a dataset can be:
  - *fixed* (unchanging), or
  - *unlimited*, which means that they are extendible (i.e. they can grow larger
- Properties of a *dataspace* consist of:
  - the *rank* (number of dimensions) of the data array,
  - the *actual sizes of the dimensions* of the array
  - the *maximum sizes of the dimensions* of the array
    - For a fixed-dimension dataset, the actual size is the same as the maximum size of a dimension.
    - When a dimension is unlimited, the maximum size is set to the value H5P_UNLIMITED

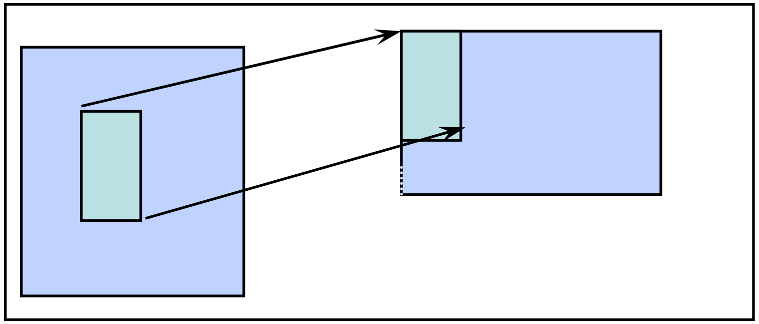
HDF Data Spaces

- A *dataspace* can also describe portions of a dataset, making it possible to do partial I/O operations on *selections*

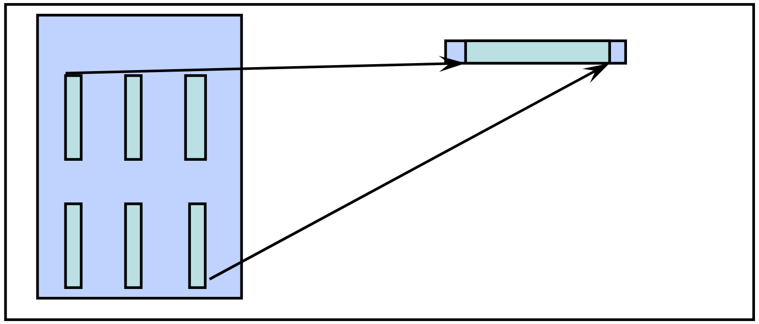
- *Selection* is supported by the dataspace interface: given an n-dimensional dataset, there are currently four ways to do partial selection:
  - Select a logically contiguous n-dimensional hyperslab
  - Select a non-contiguous hyperslab consisting of elements or blocks of elements (hyperslabs) that are equally spaced
  - Select a union of hyperslabs
  - Select a list of independent points

- Note: to perform partial read/write operations on the data, you must provided: file dataspace, file dataspace selection, memory dataspace and memory dataspace selection ***(ie. a mapping between the file layout and desired memory layout)***

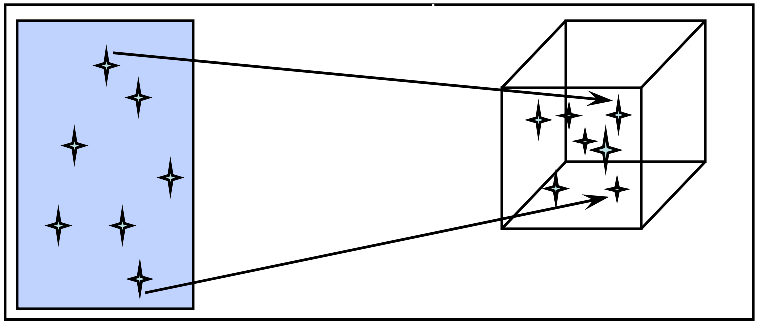
Example Mappings



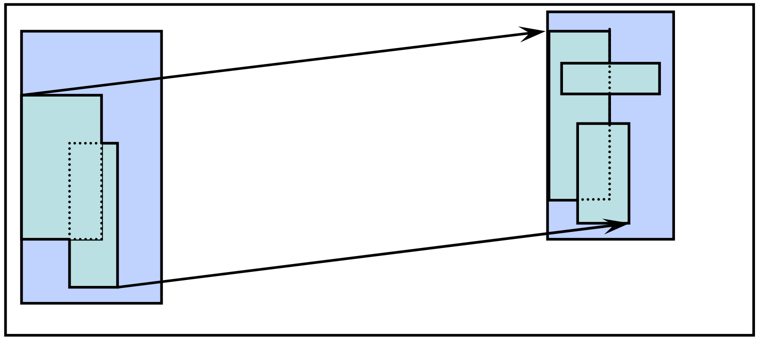
(a) A hyperslab from a 2D array to the corner of a smaller 2D array



(b) A regular series of blocks from a 2D array to a contiguous sequence at a certain offset in a 1D array



(c) A sequence of points from a 2D array to a sequence of points in a 3D array.



(d) Union of hyperslabs in file to union of hyperslabs in memory. Number of elements must be equal.

HDF Attributes

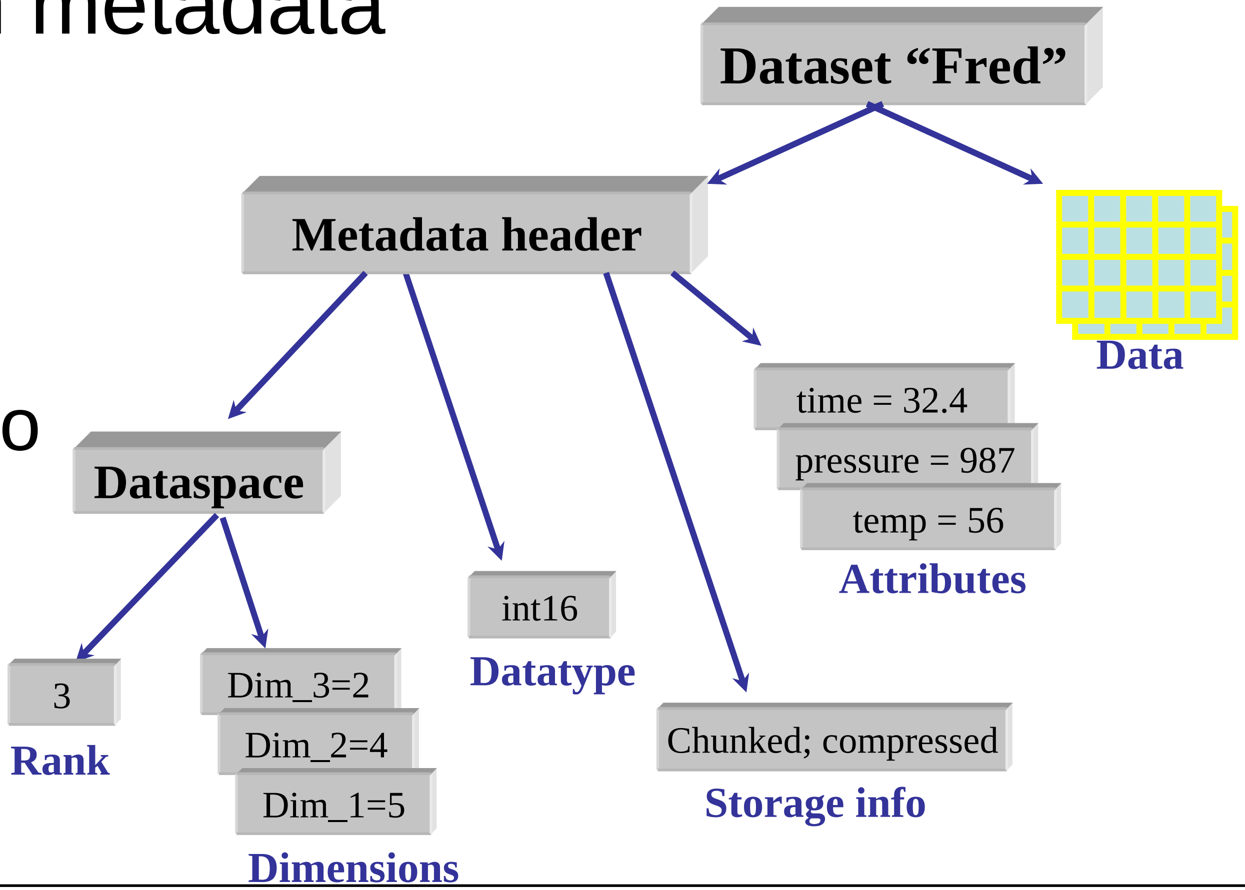
- Are small pieces of data
- Attached to datasets or groups
- Operations are scaled-down versions of the dataset operations
  - Not extendible
  - No compression
  - No partial I/O

Storage Options

- The HDF5 format makes it possible to store data in a variety of ways
- The default storage layout format is *contiguous*, meaning that data is stored in the same linear way that it is organized in memory
- Two other storage layout formats are currently defined for HDF5:
  - ***compact*** - used when the amount of data is small and can be stored directly in the object header
  - ***chunked*** - involves dividing the dataset into equal-sized "chunks" that are stored separatelyL
    - achieves good performance when accessing subsets of the datasets, even when the subset to be chosen is orthogonal to the normal storage order of the dataset.
    - makes it possible to compress large datasets and still achieve good performance when accessing subsets of the dataset
    - makes it possible efficiently to extend the dimensions of a dataset in any direction.

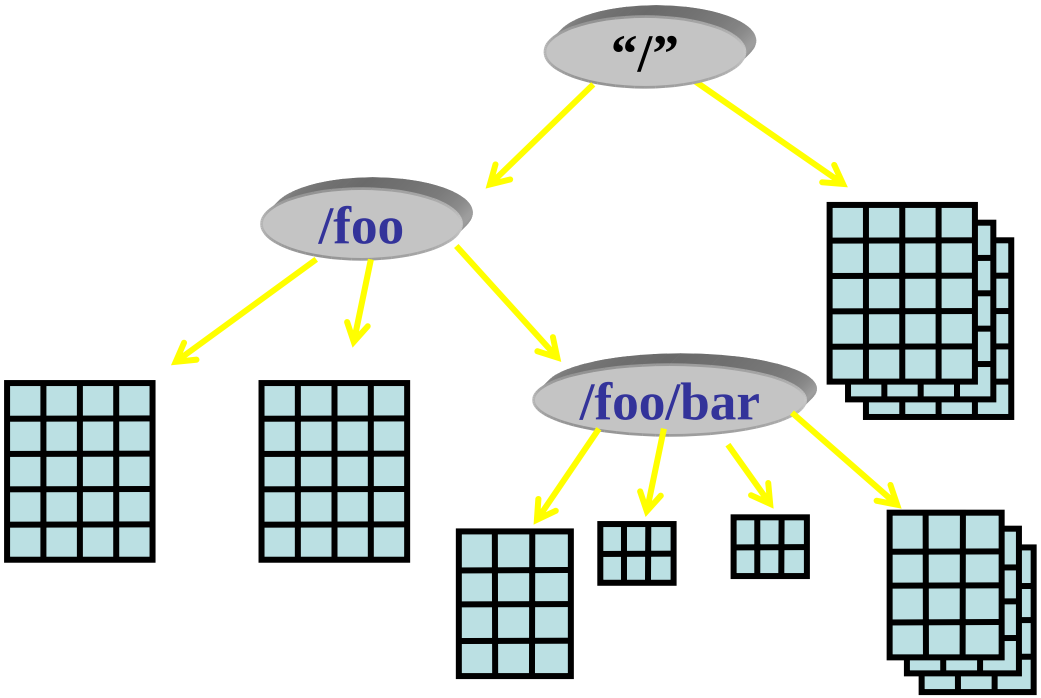
Dataset Components

- a multidimensional array of data elements
- header with metadata
  - datatype
  - dataspace
  - attributes
  - storage info



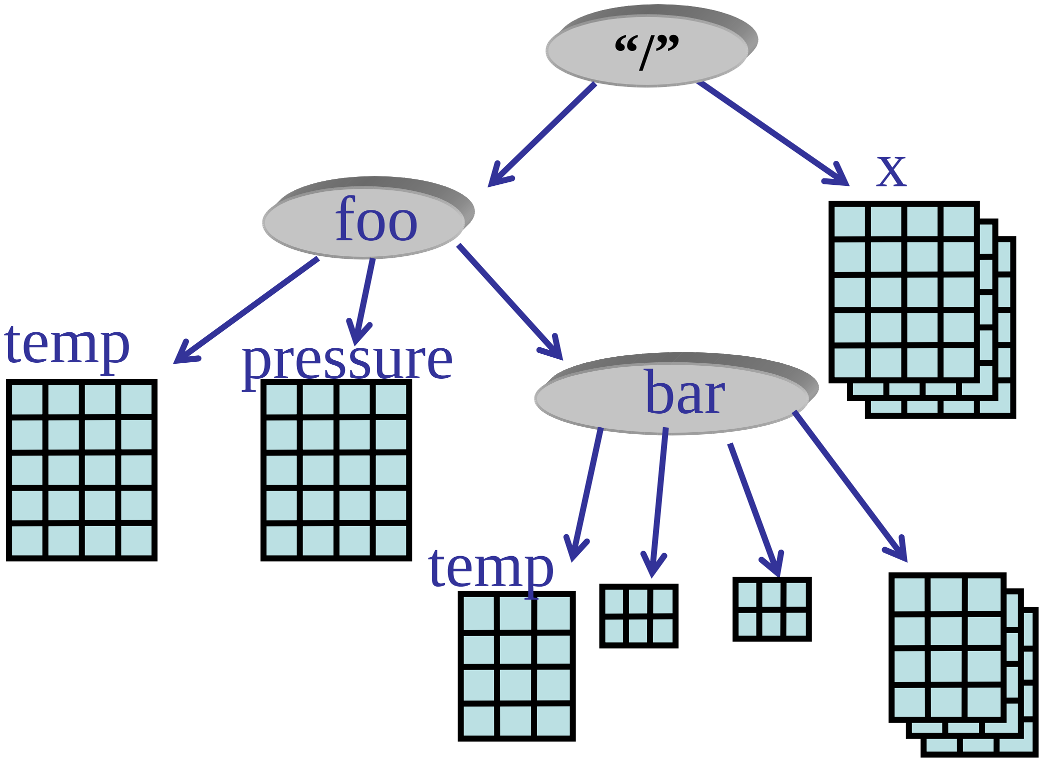
Groups

- A mechanism for collections of related objects
- Every file starts with a root group
- Similar to UNIX directories
- Can have attributes

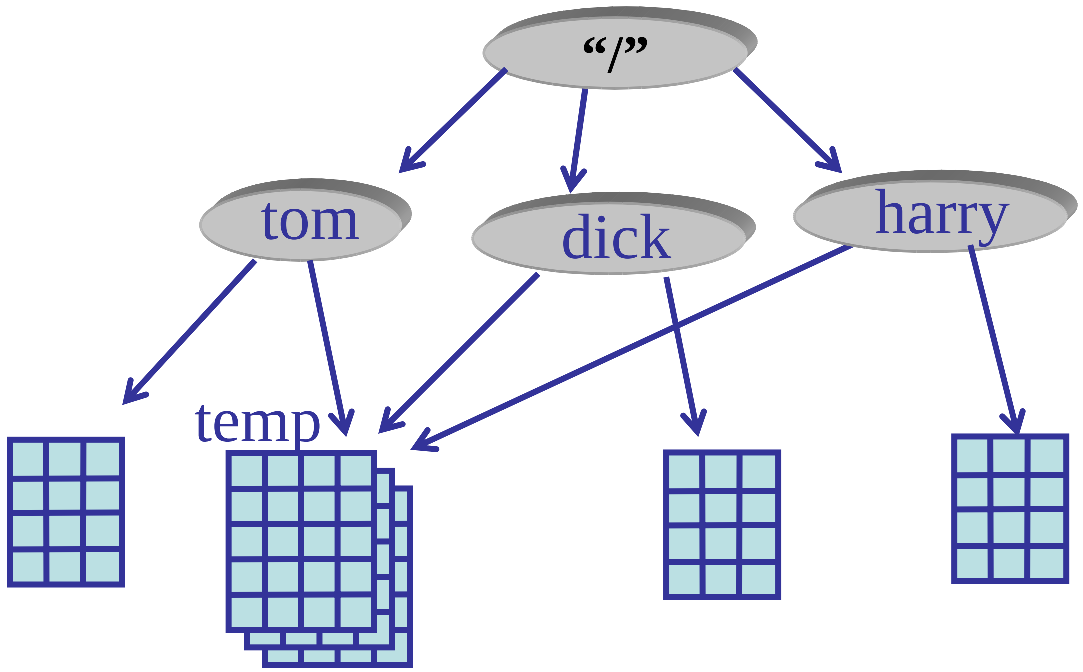


HDF5 objects are identified and located by their *pathnames*

/ (root)
/x
/foo
/foo/temp
/foo/pressure
/foo/bar/temp



HDF Members Can be Shared



Note: Three pathnames identify the same object

/tom/temp

/dick/temp

/harry/temp

The General Programming Paradigm

1. Objects are first opened or created
2. objects are accessed (read/write)
3. Objects are closed

```
CALL h5fopen_f ("myfile", H5F_ACC_RDWR_F, file_id, err)
CALL h5dopen_f (file_id, "velocity", dset_id, err)
CALL h5dread_f (dset_id, H5T_NATIVE_INTEGER, data, err)
CALL h5dclose_f (dset_id, error)
CALL h5fclose_f (file_id, error)
```

Anything that can be set from the API can also be queried

Creating an HDF5 Dataset

- Create an identifier for the dataset
- Independently define dataset characteristics
  - datatype, dataspace, property list
- Create the dataset
  - specify path, datatype, dataspace, etc.
- Close datatype, dataspace, dataset, etc.

Atomic Data Types

- The library has predefined native atomic types:

H5T_NATIVE_CHAR	H5T_NATIVE_INT8
H5T_NATIVE_USHORT	H5T_NATIVE_UINT16
H5T_NATIVE_INT	H5T_NATIVE_INT32
H5T_NATIVE_LONG	H5T_NATIVE_LLONG
H5T_NATIVE_FLOAT	H5T_NATIVE_FLOAT32
H5T_NATIVE_DOUBLE	H5T_NATIVE_FLOAT64
H5T_NATIVE_STRING	H5T_NATIVE_TIME
H5T_NATIVE_DATE	H5T_NATIVE_BITFIELD
H5T_NATIVE_OPAQUE	H5T_NATIVE_INT64

- Some non-native types will be added for common architectures.
- New types can be derived from existing types

Dataset I/O

- Dataset I/O involves
  - reading or writing
  - all or part of a dataset
- During I/O operations data is translated between the source & destination
  - data types (e.g. 16-bit integer => 32-bit integer)
  - dataspace (e.g. 10x20 2d array => 200 1d array)
  - also compressed/uncompressed, etc.

Order of operations

- The library imposes an order on the operations by argument dependencies.
  - Example: a file must be opened before a dataset because H5Dopen() takes a file handle as an argument.
  - Example: a data space must be created before a dataset because H5Dcreate() takes a data space handle as an argument.
- Objects can be closed in any order and reusing a closed object will result in an error.
- All objects are closed by normal program exit or H5close().

Useful HDF5 Binaries

- These binaries are generally built during the normal HDF installation
  - **h5ls** - lists contents of HDF5 file
  - **h5dump** - higher level view of the file
  - **h5diff** - show difference between two HDF files

```
lonestar2--> h5dump SDS.h5
HDF5 "SDS.h5" {
GROUP "/" {
 DATASET "IntArray" {
    DATATYPE  H5T_STD_I32LE
    DATASPACE  SIMPLE { ( 5, 6 ) / ( 5, 6 ) }
    DATA {
    (0,0): 0, 1, 2, 3, 4, 5,
    (1,0): 1, 2, 3, 4, 5, 6,
    (2,0): 2, 3, 4, 5, 6, 7,
    (3,0): 3, 4, 5, 6, 7, 8,
    (4,0): 4, 5, 6, 7, 8, 9
    }
 }
}
}
```

HDF Compilations

- Once you start using the HDF API, you will need to start linking your application against the library

- Also need to include the appropriate header file in your application:

  #include "hdf5.h"

- Example compile:

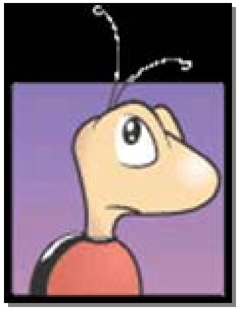
  > **icc -I /opt/apps/hdf5/hdf51.6.5/include/**
  -L /opt/apps/hdf5/hdf5-1.6.5/lib/
  h5_write.c -lhdf5

TACC

References/Acknowledgements

- HDF Group: http://www.hdfgroup.org/HDF5/

Debugging Scientific Applications



- Motivation for developing good debugging skills:
  - Unless you are from a new planet, you will introduce bugs at some point in your code
  - Even if you use community applications written by others, they will introduce bugs
  - And yes, commercial applications have bugs too
- Extra problems:
  - As scientific researchers, we cannot simply concern ourselves with bugs that prevent the application from running
  - We actually care deeply about the accuracy and repeatability of the result (*eg. negative density values in a flow code are probably bad)*
  - Stability is a concern (*eg. an iterative based solver that never converges*)
- The addition of strong debugging skills to your toolbox will greatly enhance your efficiency and add confidence to the numerical results

Defensive Programming Tips

- One of the best defenses against runtime bugs is to use basic defensive programming techniques:
  - Check *all* function return codes for errors
  - Check *all* input values controlling program execution to ensure they are within acceptable ranges (even those from flat text files in which you know there could not possibly be an error)

  - Echo all physical control parameters to a location that you will look at routinely (eg. *stdout*). Better yet, save all the parameters necessary to repeat an analysis in your solution files (*remember the metadata options in netCDF and HDF?)*
  - In addition to monitoring for obvious floating-point problems (eg. *divide-by-zero*), check for non-physical results in your simulations (*eg.supersonic velocities predicted in a low-speed aerodynamic simulation)*

Defensive Programming Tips

- Additional suggestions:
  - Maintain test cases for regression testing - is there an analytic test case you can benchmark against?
  - Use version control systems (CVS, Subversion, etc)
  - Maintain a clean, modular structure with documented interfaces: goto's, long-jumps, clever/obscure macros, etc. are dangerous over the long haul
  - Why not include some comments? Your colleagues will thank you and it just might save your dissertation when you are revisiting a tricky piece of code after a year or two
  - Strong error checking is the mark of a sage programmer and will give you more confidence in your numerical results

Defensive Programming

- Q: Isn't checking all the error codes a waste of time?
- A: It is substantially less wasteful than long debugging sessions which could have been avoided by simple error checks
- Useful error checks indicate you know what you are doing - at an **absolute minimum**, please check your memory allocations:

```
p = (float *)calloc(nnodes+1,sizeof(float));
if(p == NULL)
  {
    printf("Allocation error for p!\n");
    exit(1);
   }
```

C

```
allocate(buf(na), stat = ierror)
 if(ierror > 0) then
    print*,'ERROR: Unable to allocate array buf'
    stop
 endif
```

Fortran

TACC

Defensive Programming

- An easy defensive strategy for Fortran programmers is to use **`IMPLICIT NONE`** in all your routines (and specifically typecast all used variables). Avoids any undesired conversions
- Be sure to initialize all variables and arrays that require it (don't count on the architecture/OS to do this for you)
- During the testing and validation phase, make use of available compiler options to debugging options to trap:
  - Intel Fortran Examples:

    -check all - enable runtime checks for out-of-bounds array subscripts, unitialized variables, etc
    -warn all - display all relevant warning messages
    -warn errors - tells the compiler to change all warning-level messages into error-level messages
    -fpe0 - tells the compiler to abort when any floating point exceptions occur

  - GCC flags to display all warnings and catch errors:

    -Wall, -Wextra, -Wshadow, -Wunreachable-code

Consult your compiler documentation for available runtime checks

Defensive Programming Example

- Consider the following example code (problem.c):

```
int main()
{
  int a, b;
  int x1, x2;

  if (a = b)
    printf("%d\n", x1);
  return 0;
}
```

- **`> gcc -o problem problem.c`**
  `> ./problem`
  `1074729080`
  Use the -Wall function to help find errors at compile time

Defensive Programming Example

Now, use the “-Wall” option to have the compiler point out possible trouble spots

```
> gcc -Wall -o problem problem.c
problem.c: In function main:
problem.c:6: warning: suggest parentheses around assignment
used as truth value
problem.c:7: warning: implicit declaration of function printf
problem.c:7: warning: incompatible implicit declaration of
built-in function printf
problem.c:4: warning: unused variable x2
```

Warnings along with line numbers are provided

```
1         int main()
2         {
3           int a, b;
4           int x1, x2;
5
6         if (a = b)
7             printf("%d\n",
x1);
8           return 0;
9         }
```

Defensive Programming

- Provide one or more levels of instrumentation in your code for debugging (*eg. a debug mode and a verbose debug mode)*
- C programmers should take advantage of the **`assert`** macro to ensure values fall within appropriate ranges

C

```c
#include <stdio.h>
#include <assert.h>

int main()
{
  int n;
  float x[100];
  n = 1000;

  /* Assert that n <= 100 */

  assert ( n <= 100);

  return 0;
}
```

```
> gcc -o macro macro.c
> ./macro
> macro: macro.c:12: main: Assertion `n <= 100' failed.
Abort
> gcc -DNDEBUG -o macro macro.c
> ./macro
>
```

Defensive Programming

- Q: What is the equivalent of assert in Fortran?
- You are on your own, but it is easy for programmers to improvise (some pre-processing is handy)
- This looks like mixed code, how do we compile this?

Fortran

```
 program main
      implicit none
      integer n
      real x(100)
      n = 1000

#ifdef DEBUG
      if ( n > 100) then
         print*,' Assertion (n <= 100) is false'
         print*,' File: ',__FILE__,' Line: ',__LINE__
         stop
      endif
#endif

      stop
      end
```

```
> ifort -DDEBUG -cpp macro.f
> ./a.out
  Assertion (n <= 100) is false
  File: macro.f Line:           10
```

Bug Identification

- Common instances in which bugs identify themselves:
  - Build errors (Makefile, preprocessor, compiler, linker)
  - Improper memory reads/writes
    - pointer errors, array bounds overruns, unitialized memory references
    - alignment problems, exhausting memory, memory leaks
  - Misinterpretation of memory
    - Type errors, e.g. when passing parameters
    - Scope/naming errors (e.g., shadowing a global name with a local name)
  - Illegal numerical operations (divide by zero, overflow, underflow)
  - Infinite loops
  - Stack overflow
  - I/O errors
  - Logic / algorithmic errors
  - Poor performance

Bug Identification

- What are the symptoms if your application has a memory bug?
  - wrong answers derived when using values from incorrect memory space
  - application behaves differently when different levels of optimization are applied; a classic memory bug symptom is as follows:
    - you compile with full optimization (-O3 for example) and your code crashes unexpectedly
    - you disable all optimization (-O0 for example) and the code runs fine
  - adding additional print statements in the program to try and isolate the bug seems to make it disappear
  - application receives an unexpected symbol and terminates
- We need to understand what it means for our application to receive an extern signal

Signals

- A signal is an *asynchronous* event which is delivered to a process.

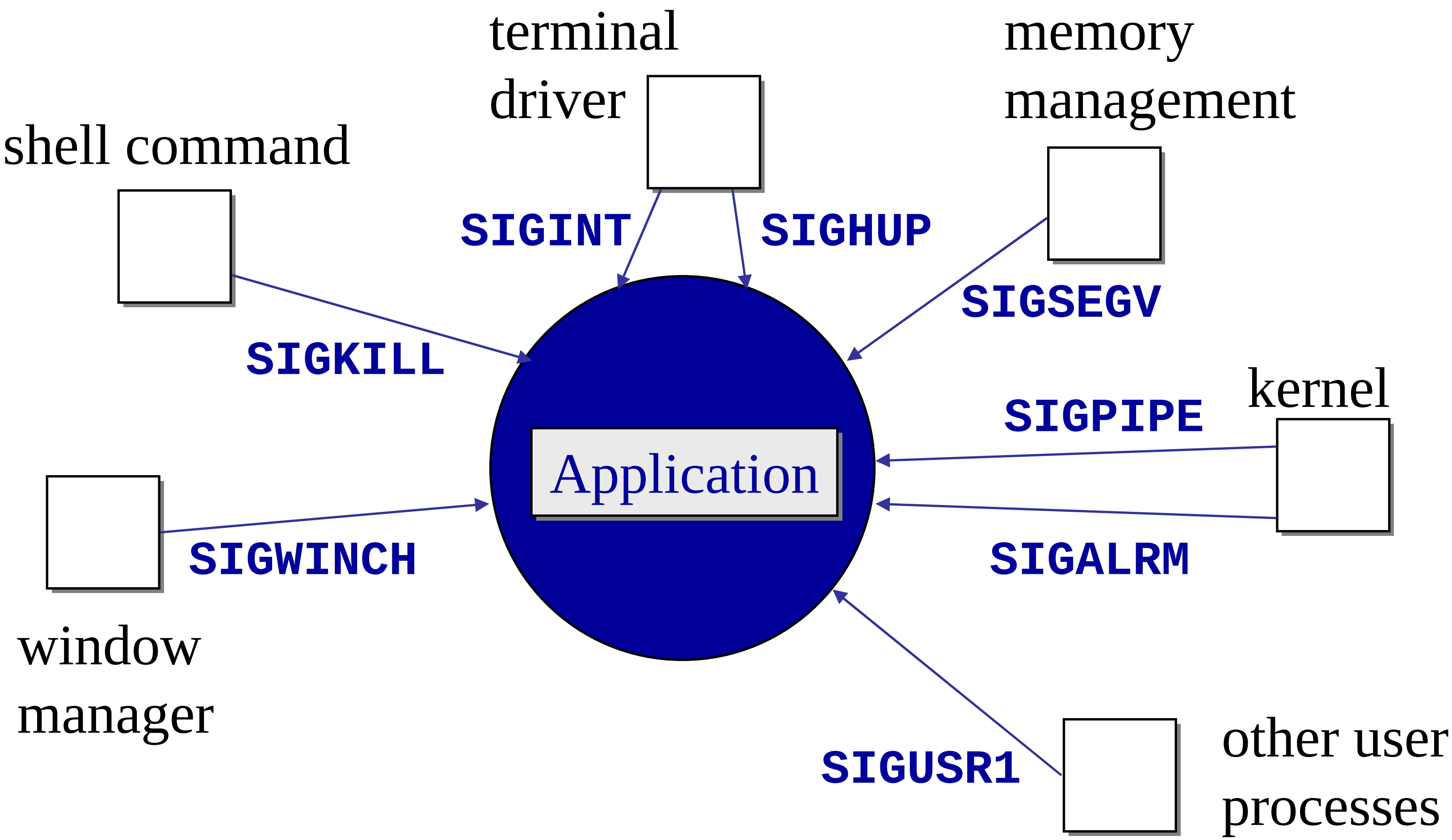
- Asynchronous means that the event can occur at any time
  - may be unrelated to the execution of the process
  - e.g. user types `ctrl-C`, or the operating system detects an error and sends a signal to your application

Common Signal Types

Name	*Description*	*Default Action*
`SIGINT`	Interrupt character typed	terminate process
`SIGQUIT`	Quit character typed (`^\`)	create core image
`SIGKILL`	Kill signal	terminate process
`SIGFPE`	Floating exception	create core image
`SIGSEGV`	Invalid memory reference	create core image
`SIGPIPE`	Write on pipe but no reader	terminate process
`SIGALRM`	`alarm()` clock ‘rings’	terminate process
`SIGUSR1`	user-defined signal type	terminate process
`SIGUSR2`	user-defined signal type	terminate process
	:	

- See `man 7` signal for more information

Common Signal Sources for Applications



Core Dumps

- Recall that the default action for a SIGSEGV was to create a core image and abort
- What in the world is a core image?
  - a **core dump** is a record of the raw contents of one or more regions of working memory for an application at a given time
  - commonly used to debug a program that has terminated abnormally
  - Note: if your application is "segfaulting" and you are not getting a core file, you may need to alter your shell limits. For example, in TCSH:

    ```
    > unlimit coredumpsize
    ```

  - Main benefit of a core file is that a post-mortem analysis can be performed on an application that failed. If the symbol table was included, you can backtrace to exactly where the application when the exception occurred.
  - Note: the location of an exception may not be the original location of a bug (particularly for memory bugs)

Debugging Process

- We recognize that defensive programming can greatly reduce debugging needs, but at some point we all have to roll up our sleeves and track down a bug

- The basic steps in debugging are straightforward in principle:
  - Recognize that a bug exists
  - Isolate the source of the bug
  - Identify the cause of the bug
  - Determine a fix for the bug
  - Apply the fix and test it

- In practice, these can be difficult for particularly pesky bugs; hence, we need some more tools at our disposal (a *debugger*)

Standard Debuggers

- Command line debuggers are powerful tools to aid in diagnosing problematic applications and are available on all Unix architectures for C/C++ and Fortran

- Example debuggers:
  - Linux: gdb
  - AIX: dbx
  - SUN: dbx

- The basic use of these debuggers is as a front-end for stepping through your application and examining variables, arrays, function returns, etc at different times during the execution

- Gives you an opportunity to investigate the dynamic runtime behavior of the application

  Note: we will focus primarily on gdb, but concepts and syntax are similar in dbx

Debugging Basics

- For effective debugging a couple of commands need to be mastered:
  - show program backtraces (the calling history up to the current point)
  - set breakpoints
  - display the value of individual variables
  - set new values
  - step through a program

Debugging Basics

- A breakpoint is a pseudo instruction that the user can insert at any place into the program during a debugging session
- Conceptually, the execution is controlled by the debugger and the debugger will interpret the breakpoints
- When execution crosses a breakpoint, the debugger will pause program execution so that you can:
  - inspect variables,
  - set or clear breakpoints, and
  - continue execution

Debugging Basics

- The notion of a conditional breakpoint also exists in which additional logic can be associated with the breakpoint

- When a conditional breakpoint is crossed during execution, the program will pause only if the breakpoint's break condition holds

- Example break conditions:
  - A given expression is true
  - The breakpoint has been crossed N times ("hit count") - this is very handy when you know something bad is happening on a particular iteration
  - A given expression has changed its value

Debugging Basics

- For debugging sessions, you should compile your application with extra debugging information included (eg. the symbol table)
- The symbol table maps the binary execution calls back to the original source code definitions
- To include this information, add “-g” to your compilation directives:

```
> gcc -g -o hello hello.c
```

Running GDB

- gdb is started directly from the shell
- You can include the name of the program to be debugged, and an optional core file:
  - **`gdb`**
  - **`gdb a.out`**
  - **`gdb a.out corefile`**

spawns a new instance of ./a.out

examines trapped state in corefile

- gdb can also attach to a program that is already running; you just need to know the PID associated with the desired process
  - **`gdb a.out 1134`**

useful if an application seems to be slow or stuck and you want to see what it is doing currently

gdb Basics

- Common commands for gdb:
  - run - starts the program; if you do not set up any breakpoints the program will run until it terminates or core dumps - program command line arguments can be specified here
  - print - prints a variable located in the current scope
  - next - executes the current command, and moves to the next command in the program
  - step - steps through the next command. Note: if you are at a function call, and you issue next, then the function will execute and return. However, if you issue step, then you will go to the first line of that function
  - break - sets a break point.
  - continue - used to continue till next breakpoint or termination

Note: shorthand notations exist for most of these commands: eg. 'c' = continue

TACC

START HERE

gdb Basics

- More commands for gdb:
  - list - show code listing near the current execution location
  - delete - delete a breakpoint
  - condition - make a breakpoint conditional
  - display - continuously display value
  - undisplay - remove displayed value
  - where - show current function stack trace
  - help - display help text
  - quit - exit gdb

gdb Basics

- Consider the following C code for subsequent examples (hello.c):

```
#include <stdio.h>
void foo();

int main()
{
  printf("inside main\n");
  foo();
  return;
}

void foo()
{
  int i, total=0;
  printf("inside foo\n");
  for(i=0;i<1000;i++)
    total += i;
}
```

Example GDB Session

```
> gcc -g -o hello hello.c
> gdb ./hello
GNU gdb Red Hat Linux (6.3.0.0-1.134.fc5rh)
Copyright 2004 Free Software Foundation, Inc.

(gdb) run
Starting program: /home/karl/cs395t/hello
inside main
inside foo

Program exited with code 0347.

(gdb) break main
Breakpoint 1 at 0x8048384: file hello.c, line 5.

(gdb) run
Starting program: /home/karl/cs395t/hello

Breakpoint 1, main () at hello.c:5
5       {
(gdb) where
#0  main () at hello.c:5
```

Example GDB Session (continued)

```
(gdb) break foo
Breakpoint 2 at 0x80483b5: file hello.c, line 13.
(gdb) cont
Continuing.
inside main

Breakpoint 2, foo () at hello.c:13
13          int i, total=0;

(gdb) list
8           return;
9         }
10
11        void foo()
12        {
13          int i, total=0;
14          printf("inside foo\n");
15          for(i=0;i<1000;i++)
16            total += i;
17        }
```

TACC

Example GDB Session (continued)

```
(gdb) cont
Continuing.
inside foo

Program exited with code 0347.
(gdb) delete breakpoints 1 2
(gdb) break hello.c:16
Breakpoint 3 at 0x80483d1: file hello.c, line 16.

(gdb) condition 3 i==401
(gdb) run
Starting program: /home/karl/cs395t/hello
inside main
inside foo
Breakpoint 3, foo () at hello.c:16
16          total += i;
(gdb) print i
$1 = 401
(gdb) where
#0  foo () at hello.c:16
#1  0x080483a6 in main () at hello.c:7
```

TACC

References/Acknowledgements

- Arctic Region Supercomputing Center, Core Skills for Computational Science: http://people.arsc.edu/~cskills/

- Debugging with GDB: http://sources.redhat.com/gdb/current/onlinedocs/gdb.html#SEC_Top