Physics 2660

Lecture 11

Today
- More on bitwise operations
- Reading/writing binary files
- Dynamic memory allocation
- A few more C++ techniques

Today we'll start out by continuing our discussion about C's bitwise operators, and then we'll move on to talking about reading and writing binary files.

The Exclusive Or (XOR) Operator:

The “^” operator performs a “exclusive or” on its two arguments. This is like a reglar OR, with one exception. If one bit is “1”, then the resulting bit is “1”. If both bits are zero, or both bits are <u>one</u>, the result is “0”.

a =	1	1	0	1	0	0	1	0
	^	^	^	^	^	^	^	^
b =	0	1	1	0	0	1	1	0
c = a^b =	1	0	1	1	0	1	0	0

Constructing an Exclusive OR:

Not all compilers have an XOR operator. You can always do an Exclusive OR using other operators, though, since:

$$\boxed{a \wedge b = (a|b) \& \sim(a\&b)}$$

Note that the right-hand side can just be read as "*a* or *b*, and not *a* and *b*", which is just another way of stating the definition of XOR.

In other words, an XOR is just like an OR, except for bits that are equal to 1 in both *a* and *b*.

Cryptography with XOR:

The XOR operator is often used as part of cryptographic systems.

If you have some plain text, and you XOR it with a secret key, the result is encrypted data that can be decrypted by anyone else who knows the key. This makes use of the following property of XOR:

If `crypt = plain ^ password`

then `plain = crypt ^ password`

The weakness of this scheme is that the following is also true:

`password = crypt ^ plain`

5

If the password is randomly-generated and the same length as the plain text, this form of encryption is very strong. The only way to crack is is by brute force: just trying different passwords until you find the one that works.

Because of the weakness noted at the bottom, the same password shouldn't be used more than once, though. In the days of the cold war, Soviet spies came to the US armed with a pad full of passwords. Their associates back in the USSR had identical pads. Whenever a spy needed to send back some information, he'd use one of the passwords to encrypt it, then throw away that password. When his compatriot received the message, he'd decrypt it using the first passsword on his pad, and then discard that password. This type of encryption is called a “one-time pad”.

Review of Bitwise Operators:

`&`	`a&b`	Bitwise *and*
`|`	`a|b`	Bitwise *or*
`^`	`a^b`	Exclusive *or*
`&=`	`a &= b`	Short for *a* = *a&b*
`|=`	`a |= b`	Short for *a* = *a|b*
`^=`	`a ^= b`	Short for *a* = *a^b*
`<<`	`a<<b`	Left shift
`>>`	`a>>b`	Right shift
`>>=`	`a >>= b`	Short for *a* = *a>>b*
`<<=`	`a <<= b`	Short for *a* = *a<<b*
`~`	`~a`	Bitwise inverse

Review of Testing and Setting Bits:

Test:	`a & 1<<n`	Test bit *n* of *a*
Set:	`a |= 1<<n`	Set bit *n* of *a*
Clear:	`a &= ~(1<<n)`	Clear bit *n* of *a*

The fputc Function:

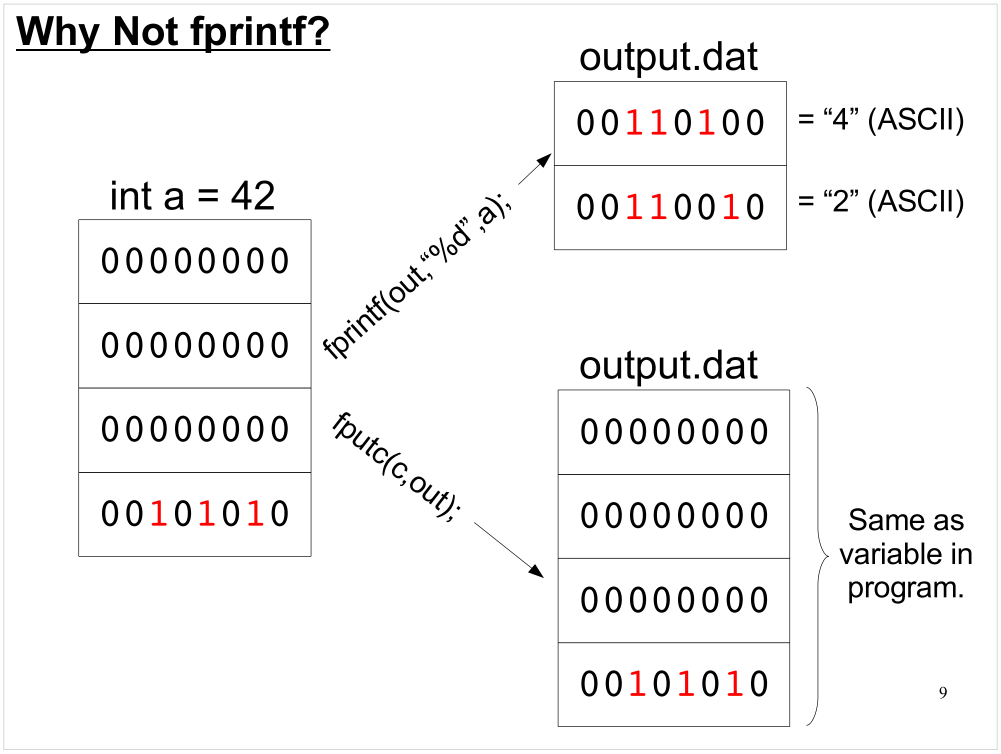
We can use the “fputc” function to write raw, unformatted binary data into a file, one byte at a time:

```
int fputc (int c, FILE *fp);
```

```
int main () {
  FILE *file = fopen ("8bits.txt","wb");

  for (int i= 0 ; i <= 255 ; i++) {
    unsigned char c = i;
    fputc (c, file);
  }
  fclose (file);

  return 0;
}
```

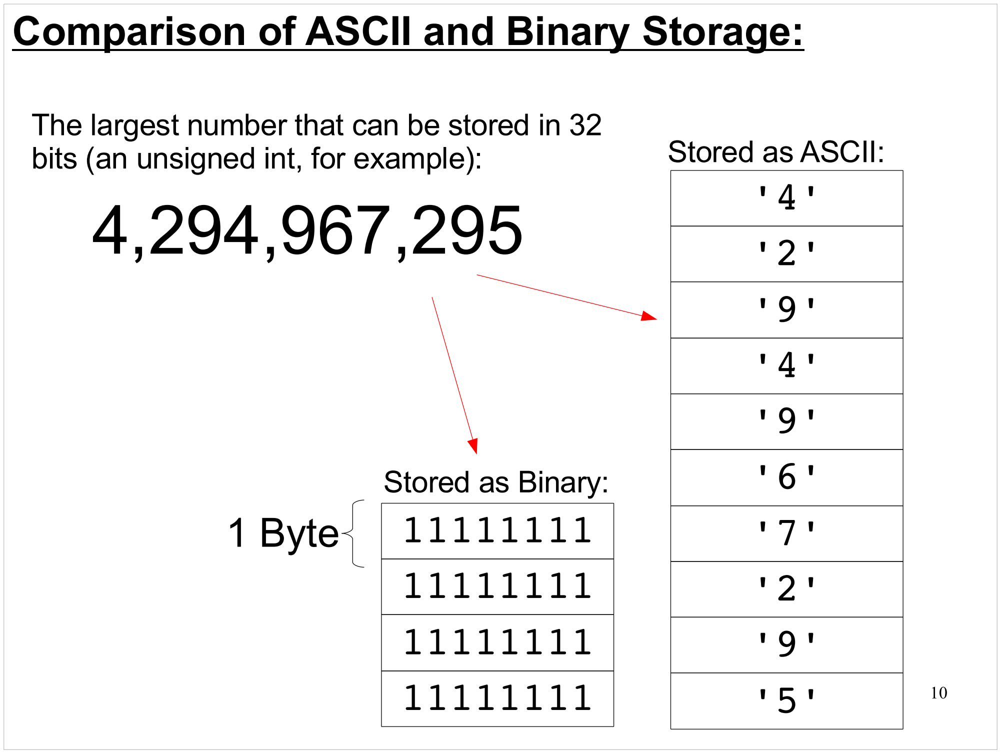
It's good practice to use “rb” or “wb” when opening a file for bitwise reading or writing. Some computers differentiate between binary files and other files.

The example above just writes all of the possible combinations of 8 bits into the file “8bits.txt”.



It may be confusing that we use the “char” variable type within our program to store non-character data that we write out with fputc, but we use fprintf to write out variables of types like “int” as characters.

It may help if you don't think of “char” as storing a character. We don't use it for that purpose in the things we're talking about today. we just use it as 8 bits of storage that we can put any kind of data into.



As we'll see later, it can often take a lot more space to store information as ASCII text than as binary data.

The char Variable Type:

`char a =` | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | = 'A' = 65 = 0x41

$\underbrace{\underbrace{0100}_{4}\ \underbrace{0001}_{1}}_{A}$

ASCII Dec Hex

Dec	Hex	Char
0	00	Null
...		
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
...		

“char” type variables are stored in 8 bits (one byte) of memory. These bits can either be interpreted as an ASCII character from the table at left, or as a number.

Because each char is exactly one byte, it's a convenient variable type to use when we're manipulating data one byte at a time.

Unsigned Versus Signed Variables:

1	1	0	1	0	0	1	0

- Signed variable types can store either positive or negative numbers.
- Unsigned types can store only positive numbers.

In principle, signed variables could just reserve one bit (say, the left-most one) to indicate whether the number is positive or negative. This "sign bit" method would work fine, and early computers did it this way.

A problem arises when we start doing arithmetic with sign bit notation, though:

Consider the case of adding two numbers with sign bits together. We must first check to see if either of the numbers is negative, and if it is, we need to subtract its value rather than adding it.

Two's Complement Notation:

To simplify arithmetic operations, modern computers use a different representation for negative numbers, called “Two's Complement”. Here's how the numbers 1 and -1 look on Galileo:

`char a =`	0	0	0	0	0	0	0	1	= 1
`char b =`	1	1	1	1	1	1	1	1	= -1

If we use two's complement notation, the computer doesn't need to do anything special when it adds a negative number. Just adding the numbers normally, without worrying about sign, produces the right result:

`a + b =`	0	0	0	0	0	0	0	0	= 0

The number -1 has all of the bits set. If you think of these eight bits as the digits of a car's odometer, you can see what happens when we add 1 to the number. All of the digits roll over, and the number becomes all zeros.

If we used a “sign bit”, and represented -1 as “10000001”, then when we added 1 to -1 we'd get “10000010”, or -2 in this notation! With sign-bit notation, we'd always need to check whether a number was positive or negative before adding it.

To form the two's complement of an 8-bit negative number, you can do this:

* Subtract the number from 2^8.
* Add one to the number.

For integers of other sizes (32 bits, for example), start out by subtracting the number from 2 to some other power (32, for example).

Writing Other Data Types as Binary Data:

The “write_out” function below uses fputc to write any kind of data into a file. The data doesn't need to be in 8-bit chunks, and it doesn't need to be integers. Consider the following example:

```
void write_out(unsigned char *c, int nbytes, FILE* file){
  for (int i=0; i<nbytes; i++){
    fputc (*c, file);
    c++;
  }
}

int main () {
  double a[]={12,13,15,123e23};
  FILE *out=fopen("out.dat","wb");

  write_out((unsigned char *)a, sizeof(a),out);

  fclose(out);
  return 0;
}
```

An array of 4 “doubles”. (A double occupies 8 bytes.)

Cast the array's starting address as a char * pointer.

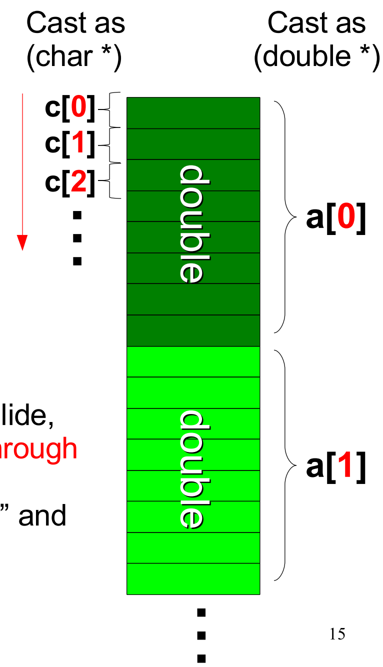
The total size of the array, in bytes.

We tell write_out to just treat this chunk of memory as a bunch of bytes, without worrying about what they represent.

Breaking Data into Byte-Sized Chunks:

```
void write_out(
   unsigned char *c,
   int nbytes,
   FILE* file){

  for (int i=0; i<nbytes; i++){
    fputc (*c, file);
    c++;
  }
}
```

Increment by size of one “char” (i.e., 1 byte).



Continuing the example from the previous slide, the “write_out” function just works its way through the array of doubles, one byte at a time, interpreting each byte as an “unsigned char” and writing it out to a file.

The space where the array lives is just a hunk of bits. We can divide it any way we want. The data will only make sense if we interpret it as a bunch of “double”s, but if we're just interested in copying bits from one place to another, it doesn't matter whether each chunk of bits is a sensible number or not.

The “od” Command:

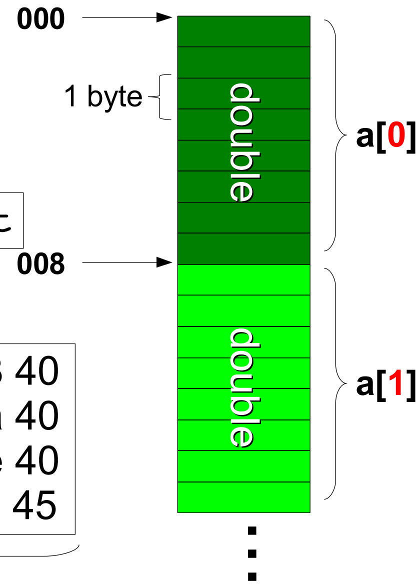
From the command line, you can look at the contents of a file byte by byte using the “od” command:

```
od -Ad -w8 -tx1 out.dat
```

```
0000000 00 00 00 00 00 00 28 40
0000008 00 00 00 00 00 00 2a 40
0000016 00 00 00 00 00 00 2e 40
0000024 09 7e 12 ac 40 59 24 45
```

Starting Addresses

Contents of each byte
(Each pair of hex characters is one byte.)



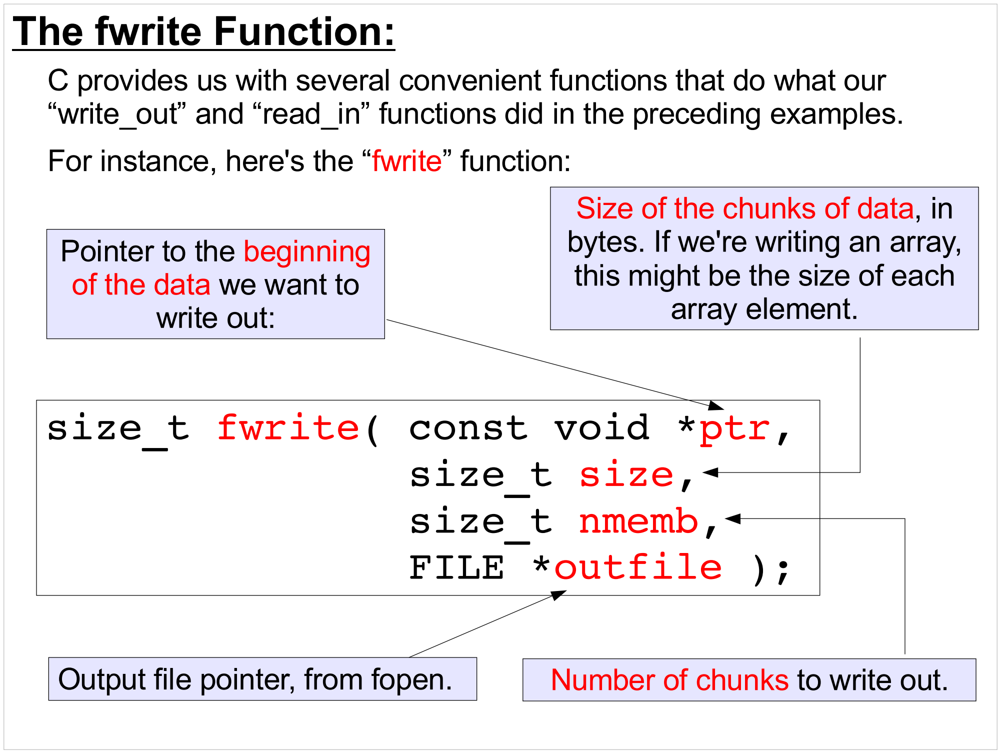
The fgetc Function:

You can use the fgetc function to read data from a file, one byte at a time:

```
void read_in(unsigned char *c, int nbytes, FILE* file){
  for (int i=0; i<nbytes; i++){
    *c = fgetc (file);
    c++;
  }
}

int main () {
  double a[4];
  FILE *in=fopen("out.dat","rb");

  read_in((unsigned char *)a, sizeof(a),in);

  for (int i=0;i<4;i++) {
    printf("%g\n",a[i]);
  }

  fclose(in);
  return 0;
}
```

Notice that we're reading the data in one byte at a time, but then interpreting it as an array of "double"s.

Compare fgetc to fscanf, which reads formatted data from a file.



In the preceding examples we created functions called “write_out” and “read_in” to loop through the bytes of our data one at a time. These functions used fputc and fgetc to write or read each byte. C provides us with standard functions that can do all of this work for us, without having to write our own functions.

“size_t” is just an integer here. Different operating systems use different types of integers (int, long, long long) to represent the size of a chunk of data. “size_t” is just an alias meaning, “a variable appropriate for holding the size of a chunk of storage on this computer”.

fwrite returns the number of bytes successfully written.

An fwrite Example:

We can replace our “write_out” function with a call to “fwrite”.

Notice that fwrite doesn't care how you break your data into chunks. In this example, we could either write out one chunk that's the size of the whole array, or four chunks each the size of one array element.

```
int main () {
  double a[]={12,13,15,123e23};
  FILE *out=fopen("out.dat","wb");

  fwrite((void *)a, sizeof(a),1,out);

// Alternatively:
//fwrite((void *)a, sizeof(double),4,out);

  fclose(out);
  return 0;
}
```

Size of chunks.

Number of chunks.

The fread Function:

Similarly, the fread function can be used to read in a bunch of data from a file:

Pointer to the beginning of the data we want to read in:

Size of the chunks of data, in bytes. If we're writing an array, this might be the size of each array element.

```
size_t fread(void *ptr,
             size_t size,
             size_t nmemb,
             FILE *infile);
```

Input file pointer, from fopen.

Number of chunks to read in.

An fread Example:

```
int main () {
  double a[4];
  FILE *in=fopen("out.dat","rb");

  fread((void *)a, sizeof(a),1,in);

// Alternatively:
// fread((void *)a, sizeof(double),4,in);

  for (int i=0;i<4;i++) {
    printf("%g\n",a[i]);
  }

  fclose(in);
  return 0;
}
```

Array big enough to hold the data we're going to read.

Size of chunks.

Number of chunks.

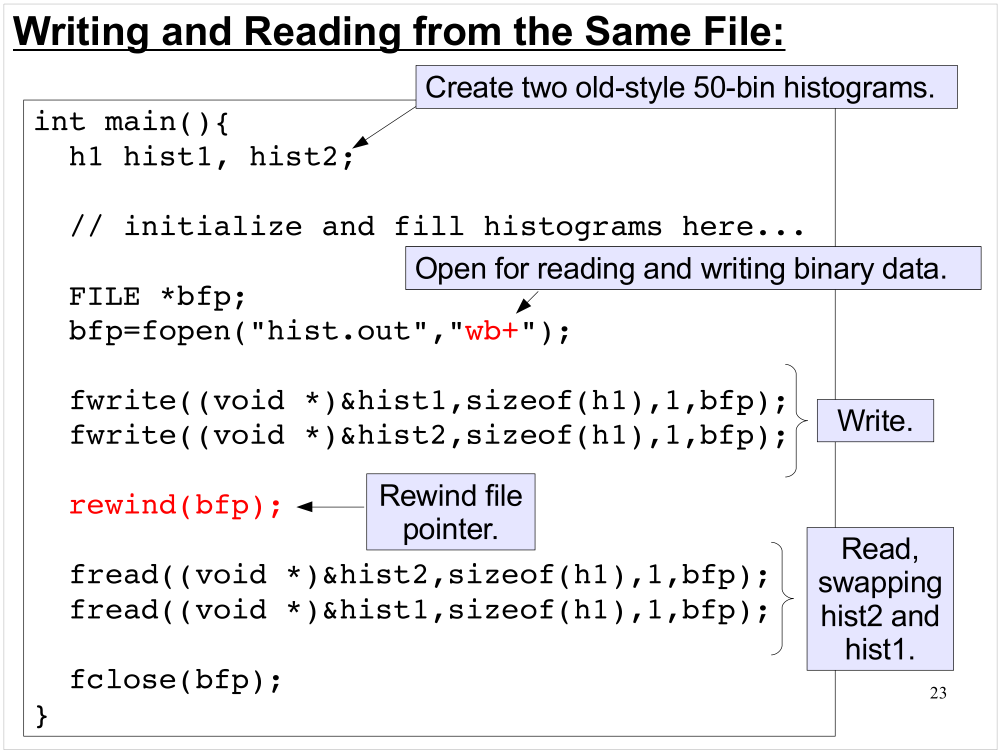
Need to make sure this matches the number of elements in the array.

File I/O Modes in fopen:

When we open a file with fopen, we can specify any of the following "read/write modes". We can add a "b" to any of them to explicitly say we're going to be doing bitwise I/O.

r	Open file for reading. File must exist.
r+	Open file for reading and writing. File must exist.
w	Open file for writing. File is created if necessary.
w+	Open file for writing and reading. File is created if necessary.
a	Open file for appending. File is created if necessary.
a+	Open file for appending and reading. File is created if necessary.

You can use any of these as the second argument to "fopen".

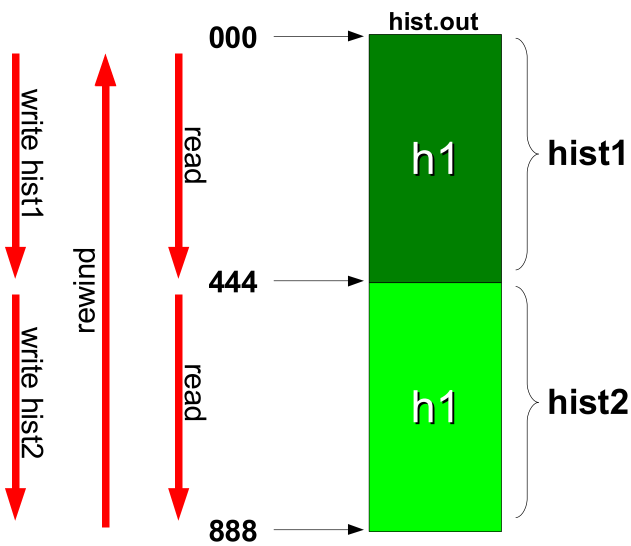


Why old-style histograms? Because they use a fixed-size array to hold their data. The new-style histograms just have a pointer to a variable-sized array of bins stored elsewhere. We'll look at this kind of thing in the second part of today's lecture.

Each of the old-style histograms takes up 444 bytes of memory or disk space.

The rewind Function:

As we read and write data, C remembers our current position in the file. If we want to go back to the beginning of the file and start again, we can use the “rewind” function.



Every time we do an fread or fwrite, we start at whatever the current position is in the file.

The fseek Function:

We can move to an arbitrary position within the file by using the "fseek" function:

```
int fseek(FILE *file, long offset, int whence);
```

How far to move... (in bytes)

...from where.

Valid values for "whence" (defined in stdio.h):

SEEK_SET	Offset relative to beginning of file.
SEEK_CUR	Offset relative to the current position.
SEEK_END	Offset relative to the end of the file.

fseek Examples:

A few examples of fseek usage:

Place file pointer at the end of the file:

```
fseek(file_p, 0, SEEK_END);
```

Back up sizeof(float) bytes from the end of the file:

```
fseek(file_p, -1*sizeof(float), SEEK_END);
```

Go to the beginning of the file:

```
fseek(file_p, 0, SEEK_SET);
```

Go forward 10*sizeof(double) bytes from the current location:

```
fseek(file_p, 10*sizeof(double), SEEK_CUR);
```

The ftell Function:

You can use the “ftell” function to find out where you are within the file:

```
FILE *fp = fopen(“file.dat”, “rb”);

fseek ( fp, 0, SEEK_END );
size = ftell(fp);

printf (“Size of file.dat (in bytes) is: %ld\n”,
        size);
```

Set position to end of file.

Where are we?

ftell reports the current position as a number of bytes from the beginning of the file.

It can be used, as above, to find out how big a file is.

The feof Function:

When reading in data from a file, you can use the feof function to tell you when you get to the end of the file:

```
int feof(FILE *file);
```

Here's a usage example:

```
while (1) {
    char c = (char) fgetc(infile);
    if (feof(infile)) break;
    fputc(c,outfile);
  }
```

If we've hit the end of the file, quit reading.

Reading and Writing Structs:

Just as with simple variables, you can read and write arbitrarily complicated structs:

```
typedef struct {
   double re;
   double im;
 } Complex;

 Complex c[10], z;

 c[2].re = 3.14;
 c[2].im = 1.41;

 FILE *file=fopen("out.dat","wb+");
 fwrite((void *)c, sizeof(Complex), 10, file);

 rewind(file);

 fseek(file, 2*sizeof(Complex), SEEK_CUR);
 fread((void* )&z, sizeof(Complex), 1, file);

 printf ("Third number: re=%lf, im=%lf\n",
         z.re, z.im);
```

This example does the following:

- writes out an array of 10 structs,
- goes back to the beginning of the file,
- skips over the first two structs,
- reads the third one back in.

Comparison of Binary and ASCII I/O:

It's convenient to be able to read or write a whole array of data with a single C statement:

Binary File:

```
FILE *binfile=fopen("binary.dat","wb");
fwrite((void *)c, sizeof(Complex), NMEMB, binfile);
fclose(binfile);
```

File size = 1.6 kB

To preserve the full precision of our numbers, we need to write out many characters if we write a text file:

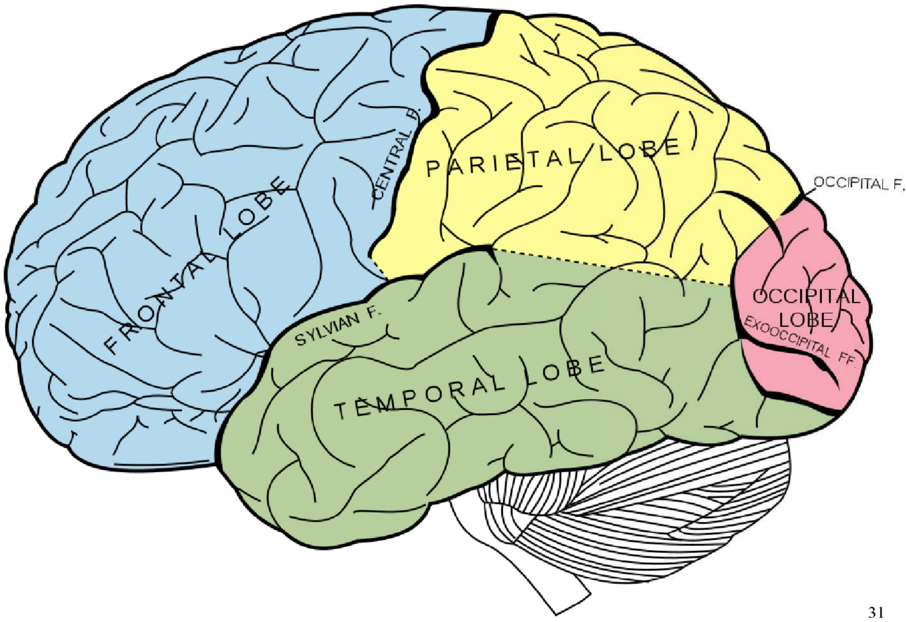
ASCII File:

```
FILE *file=fopen("ascii.dat","w");
for (int i=0;i<NMEMB;i++) {
 fprintf(file,"%56.53lf %56.53lf\n",
         c[i].re,c[i].im);
}
fclose(file);
```

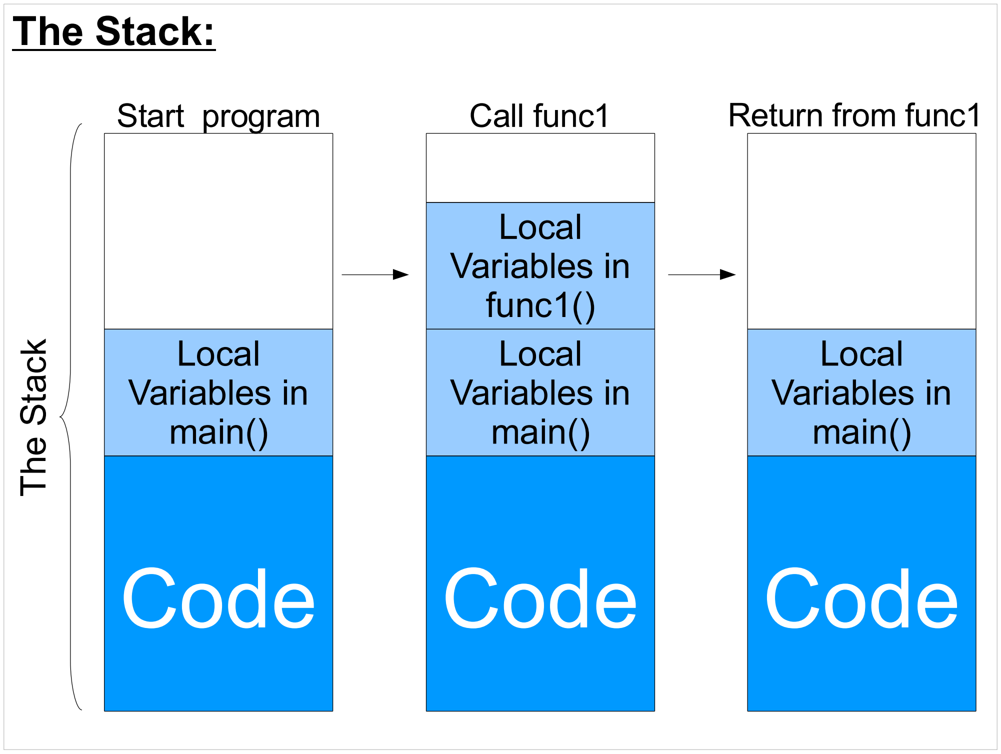
File size = 11.4 kB

The file sizes are for files containing 100 complex numbers, using the Complex struct we used earlier.

Part 2: Dynamic Memory Allocation



Until now, all of the memory used by our variables has been defined at the time we compiled our program. What if we want to change the size of an array while our program is running, or create a new array on the fly?



When a program starts running, the code describing all of your functions gets loaded into a dedicated section of memory called a stack. As functions are called, their local variables get pushed on top of the stack. When a function completes, this memory is freed up for other use.

Each running program has its own stack. The stack has a limited size (typically a few megabytes). To see the stack size of your current terminal process on galileo, type:

```
limit stacksize
```

It's possible to use up all of the availabe memory in the stack. Imagine a recursive function that invokes itself over and over again. Each time it's invoked, a new set of local variables is pushed onto the top of the stack. If the recursion goes too far before reaching its termination condition, the program may die with a “Stack Overflow” error.

Note that global and static variables are stored elsewhere, in their own memory space.

The Heap:

The program has another section of memory available to it, called the "heap". Programs can dynamically allocate memory in the heap. The memory there won't be reclaimed until the program explicitly "frees" the memory.

The heap is usually much larger than the stack. It includes much of the otherwise-unused memory available on the computer.

The malloc Function:

A program can request a chunk of memory in the heap by using the "malloc" ("memory allocate") function:

```
void *malloc(size_t size);
```

The size, in bytes, of the requested chunk of memory.

If enough memory is available, malloc returns a void * pointer pointing to the beginning of the newly-allocated chunk of memory.

If malloc fails, it returns the special value NULL.

The calloc Function:

The calloc function is similar to malloc, but it's more convenient when requesting space to store an array:

```
void *calloc(size_t nmemb, size_t size);
```

The number of array elements.

The size, in bytes, of each array element.

Unlike malloc, the calloc function automatically initializes the space by setting it all to zero.

The free Function:

Once you're done with the allocated memory, you should use the “free” function to free it up again. This will make it available for other uses.

```
void free(void *ptr);
```

Pointer to a previously-allocated chunk of memory.

One of the most common C programming problems is a “memory leak”. This happens when your program keeps allocating more and more memory, without ever freeing it. Over time, the program's memory usage grows until no more memory is available (possibly causing problems for other programs), and your program crashes.

When your program exits, all of its allocated memory will automatically be freed.

The realloc Function:

After you've allocated a chunk of memory, you may decide that it needs to be bigger. You can grow or shrink the size of an allocated chunk by using the “realloc” (“re-allocate”) function.

```
void *realloc(void *ptr, size_t size);
```

Returns location of new chunk.

Location of old chunk.

New size.

Note that realloc doesn't actually resize the chunk you're currently using. Instead, it copies your data into a new location with a different size, and returns the address of the new location.

You can also use realloc to reduce the size of an allocated chunk of memory. If you give realloc a size of 0, it's equivalent to calling “free”.

Allocation Failure:

The computer won't always have enough memory available to satisfy your allocation request. Here's an example showing how you can deal with that possibility:

```
int num = 100;
long *lptr = (long *) malloc(num * sizeof(long));

if (lptr == NULL) {
   printf("Can't allocate memory\n");
   return(1);
}
```

A Dynamic Memory Example:

```
int main (int argc, char *argv[]) {
  int howmany = atoi(argv[1]);
  int *ptr;

  ptr = (int *)calloc( howmany, sizeof(int) );
  if ( ptr == NULL ) {
    printf ("Could not allocate memory.\n");
    return(1);
  }

  for (int i=0; i<howmany; i++)
    ptr[i] = rand();

  for (int i=0; i<howmany; i++)
    printf ("%d\n", ptr[i]);

  free(ptr);
}
```

Request space for an array of the given size.

Use the array as usual.

Free the memory when we're done.

Should always be paired.

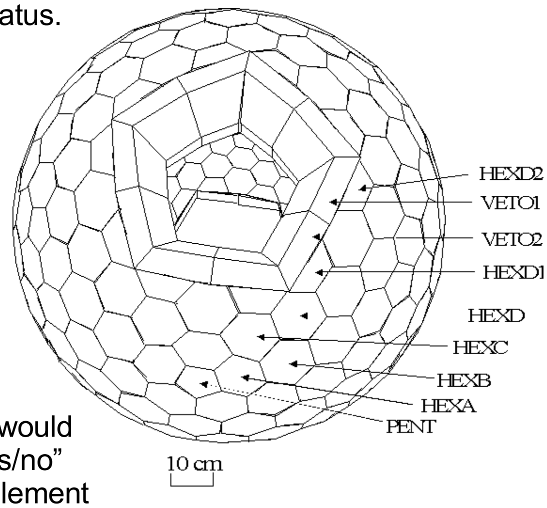
Another Example:

Consider the case of an experiment with many particle detectors that react to a shower of particles given off by events that sometimes occur in the center of the apparatus.

In this experiment, we only want to know which detectors fired during each event.

Each event will cause a different number of detectors to fire. Generally, only a small fraction of the detectors will fire for each event.

If we have millions of events, it would be very inefficient to store a “yes/no” array for each event, with one element for each detector.



This is a pretty general problem. How do you store and retreive chunks of data that have different sizes? Until now, we've always read chunks of data that have a fixed size. For example, we might read 100 pairs of x,y values from a file, where each line of the file just holds two floating-point numbers.

What if we wanted to read the names of each student in each Physics class? Each class has a different number of students.

There are many ways to solve this problem, but the following shows a common way of doing it when we're reading and writing binary data.

Reading Data of Variable Size:

```
typedef struct{
  int ndet;
  int *detector;
} Event;

FILE *file=fopen("data.dat","rb");

Event e;

while (1) {
 fread( (void *)&e.ndet, sizeof(int), 1, file );
 if (feof(file)) break;

 e.detector = (int *) malloc(e.ndet*sizeof(int));
 fread( (void *)e.detector, sizeof(int), e.ndet, file);

 // Do stuff with the data...

 free(e.detector);
}
```

Number of detectors that fired.

List of detectors that fired.

Struct to hold an event.

Start by reading the number of detectors that fired.

Get space for array of detectors.

Read in array.

Free array.

So, we start out by reading in the number of detectors, then we allocate enough memory to store this list, then read in the list of detector numbers. We keep repeating this until we get to the end of the file.

This is something we mentioned earlier, when talking about C++ classes. Let's take another look at it.

Duplicate Function Names:

```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

void swap(float *a, float *b) {
  float tmp = *a;
  *a = *b;
  *b = tmp;
}
```

```
int main(){
   int i=1,j=2;
   float x=3.14,y=2.71;

   swap(&i,&j);
   swap(&x,&y);
}
```

In C++, you can define multiple functions with the same name, as long as each function has a unique calling syntax.

In this example, one version of “swap” takes two integers as its argments, and the other “swap” function takes two floats.

When your program says “swap”, the compiler determines which one you mean by looking at the types of the variables you give it.

Note that the compiler doesn't look at the type of variable returned, when deciding between the two functions. It only looks at the argument types.

Return Types of Overloaded Functions:

```
int max(int *a, int *b) {
  if (*a > *b)
   return *a;
  else
   return *b;
}

float max(float *a, float *b) {
  if (*a > *b)
    return *a;
  else
   return *b;
}
```

Not that the functions don't need to return the same type of data.

In this case, one version returns “int” and the other returns “float”.

```
int main(){
  int i=1,j=2;
  float x=3.14,y=2.71;

  printf(“%i\n”,max(&i,&j));
  printf(“%f\n”,max(&x,&y));
}
```


Overloading is often convenient and can increase readability, but only if overloaded versions perform the same tasks!

Generic Functions:

Sometimes, rather than function overloading, it's better to write a single generic function that can deal with data of many different types. This can be done using void * pointers to memory locations, as we've seen before. One advantage of this approach is that it can be used in either C or C++.

Inefficient function to clear memory byte-by-byte.

```
void clearmem(void *start, int nbytes) {
     for (int i=0; i<nbytes; i++)
          *(start+i)=0;
}

int main(){
     double d_ary[100];
     short s_ary[100];

     clearmem((void *)d_ary, 100 * sizeof(double));
     clearmem((void *)s_ary, 100 * sizeof(short));
}
```

The same function can clear an array of doubles, ints or any other type.

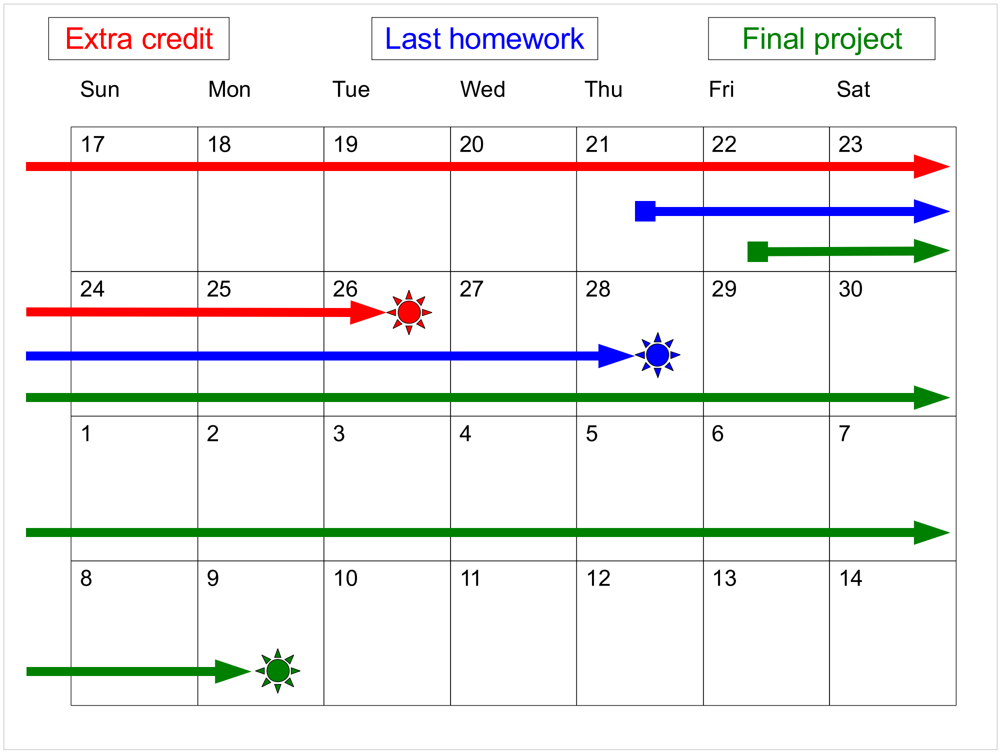
Next Time:

- Data structures

This week's Lab:

- Pipes and digital sound

Brooks Ch. 1, Ch. 2 sections 1-2.2



Here'e our schedule for the next few weeks. Note that there will be two more lab sessions, on the 21st and 28th, and we'll have two more lectures, on the 26th and the 3rd .

I'll be posting the final project assignment this Friday.

If you're doing the extra credit assignment, please e-mail it to me by 5pm on the 26th .

The End

Thanks!