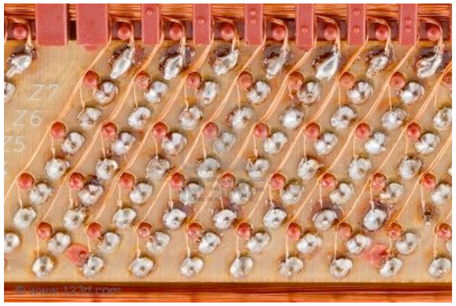
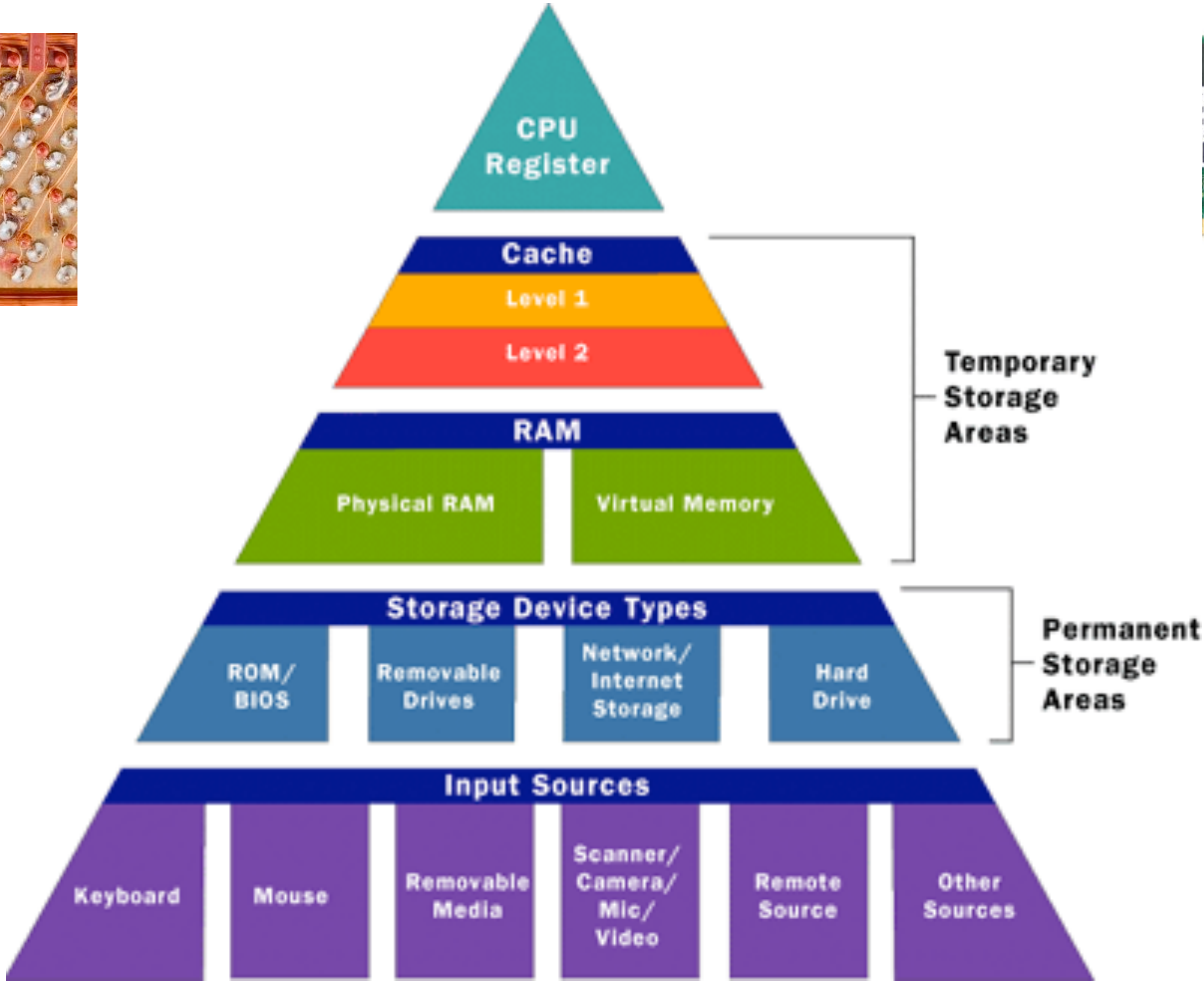
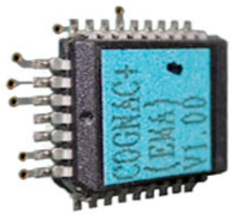
Memory: on the hardware side







@ http://computer.howstuffworks.com/computer-memory.htm/printable

Memory: on the software side

Each programming languages offers *a different abstraction*

The goal is to make programming easier and improve portability of code by hiding irrelevant hardware oddities

Each language offers a memory API – a set of operations for manipulating memory

Memory: the C++ Story

C++ offers a story both simpler and more complex than Java

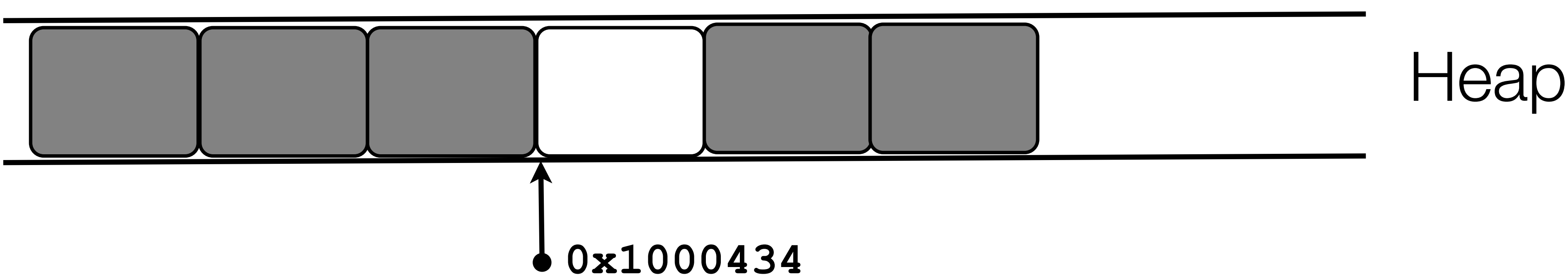
Memory is a sequence of bytes, read/written from an address

Addresses are values manipulated using arithmetic operations

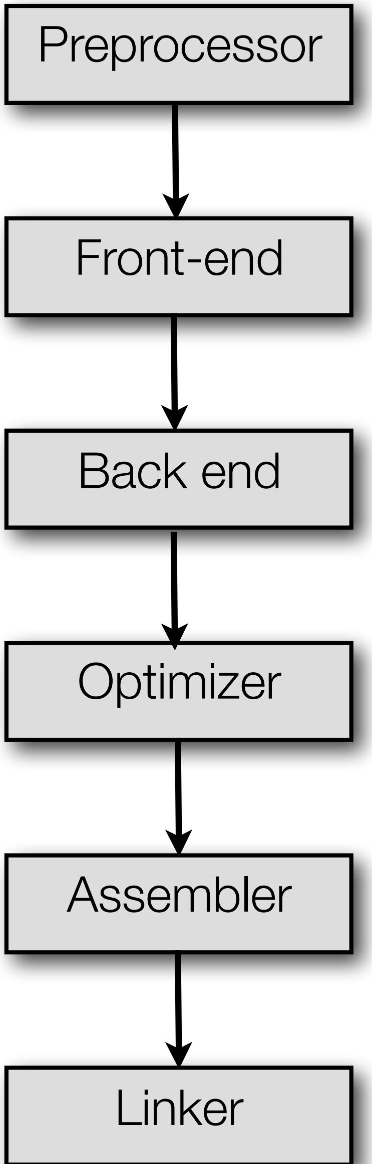
Memory can be allocated:

- Statically
- Dynamically on the stack
- Dynamically on the heap

Types give the compiler a hint how to interpret a memory addresses



Stages of Compilation



Debugging Options

-dletters -dumpspecs -dumpmachine -dumpversion …

Optimization Options

-falign-functions=n -finline-functions -fno-inline
-O -O0 -O1 -O2 …

Preprocessor Options

-Dmacro[=defn] -E -H ...

Assembler Option

-Wa,option -Xassembler option ...

Linker Options

object-file-name -llibrary -nostartfiles -nodefaultlibs
-nostdlib -pie -rdynamic -s -static -shared ..

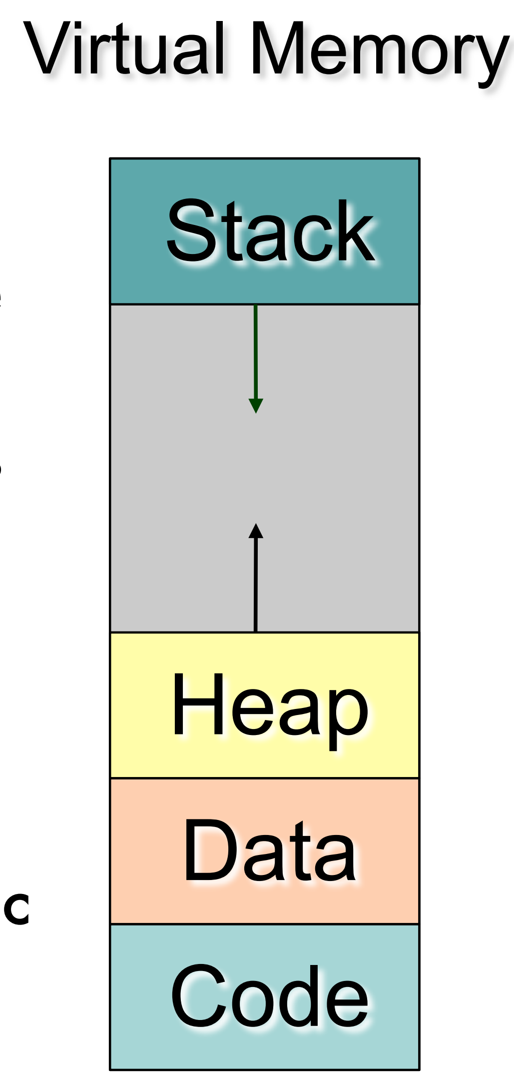
Code Generation Options

-fcall-saved-reg -fcall-used-reg -ffixed-reg -fexceptions
-fnon-call-exceptions -funwind-tables…

Trivia: Where does the name **`a.out`** comes from?
A: "assembler output"…

Memory areas

The operating system creates a process by assigning memory and other resources



- **Stack**: keeps track of the point to which each active subroutine should return control when it finishes executing; stores variables that are local to functions
- **Heap**: dynamic memory for variables that are created with new and disposed of with delete
- **Data**: initialized variables including global and static variables, un-initialized variables
- **Code**: the program instructions to be executed

Stack frame

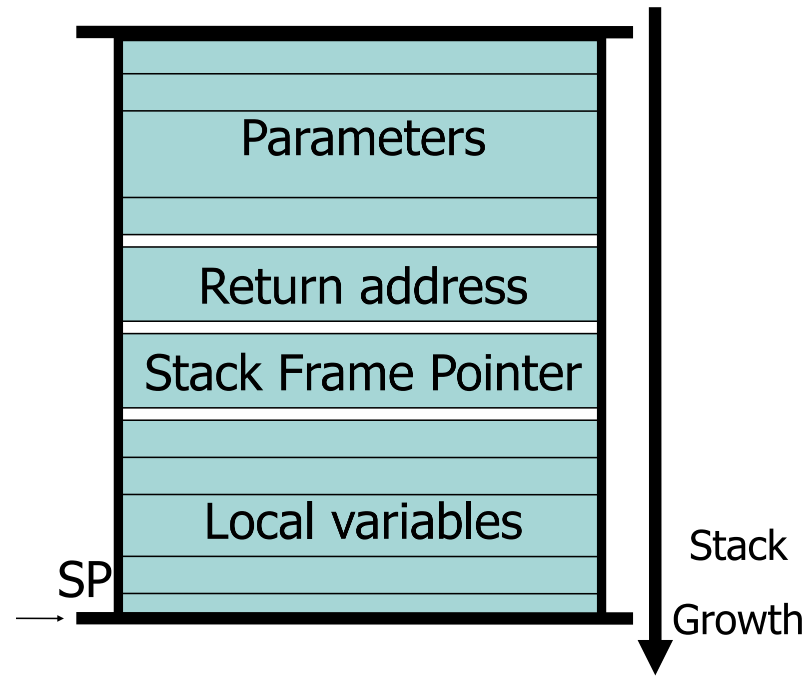
Parameters for the procedure

Save current PC onto stack (return address)

Save current SP value onto stack

Allocates stack space for local variables by decrementing SP by appropriate amount

Return value passed by register



Static and Stack allocation

- Static allocation with the keyword `static`
- Stack allocation automatic by the compiler for local variables
- `printf` can display the address of any identifier

```
#include <stdio.h>

static int sx;
static int sa[100];
static int sy;

int main() {
   int lx;
   static int sz;

   printf("%p\n", &sx);   0x100001084
   printf("%p\n", &sa);   0x1000010a0
   printf("%p\n", &sy);   0x100001230
   printf("%p\n", &lx);   0x7fff5fbff58c
   printf("%p\n", &sz);   0x100001080
   printf("%p\n", &main);0x100000dfc
```

Static and Stack allocation

Any value can be turned into a pointer (but **bad** style)

Arithmetics on pointers allowed

Nothing prevents a program from writing all over memory (again **bad**)

```
static int sx;
static int sa[100];
static int sy;

int main() {
   for(p= (int*)0x100001084;
        p<=(int*)0x100001230;
        p++)
  {
     *p = 42;
   }
   printf("%i\n",sx);
   printf("%i\n",sa[0]);
   printf("%i\n",sa[1]);
```

→

```
42
42
42
```

Byte

A byte = 8 bits

- Decimal 0 to 255
- Hexadecimal 00 to FF
- Binary 00000000 to 11111111

In C++:

- Decimal constant: `12`
- Octal constant: `014`
- Hexadecimal constant: `0xC`

0_{hex}	=	0_{dec}	=	0_{oct}	0	0	0	0
1_{hex}	=	1_{dec}	=	1_{oct}	0	0	0	1
2_{hex}	=	2_{dec}	=	2_{oct}	0	0	1	0
3_{hex}	=	3_{dec}	=	3_{oct}	0	0	1	1
4_{hex}	=	4_{dec}	=	4_{oct}	0	1	0	0
5_{hex}	=	5_{dec}	=	5_{oct}	0	1	0	1
6_{hex}	=	6_{dec}	=	6_{oct}	0	1	1	0
7_{hex}	=	7_{dec}	=	7_{oct}	0	1	1	1
8_{hex}	=	8_{dec}	=	10_{oct}	1	0	0	0
9_{hex}	=	9_{dec}	=	11_{oct}	1	0	0	1
A_{hex}	=	10_{dec}	=	12_{oct}	1	0	1	0
B_{hex}	=	11_{dec}	=	13_{oct}	1	0	1	1
C_{hex}	=	12_{dec}	=	14_{oct}	1	1	0	0
D_{hex}	=	13_{dec}	=	15_{oct}	1	1	0	1
E_{hex}	=	14_{dec}	=	16_{oct}	1	1	1	0
F_{hex}	=	15_{dec}	=	17_{oct}	1	1	1	1

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

Words

Hardware has a `Word size` used to hold integers and addresses

The size of address words defines the maximum amount of memory that can be manipulated by a program

Two common options:

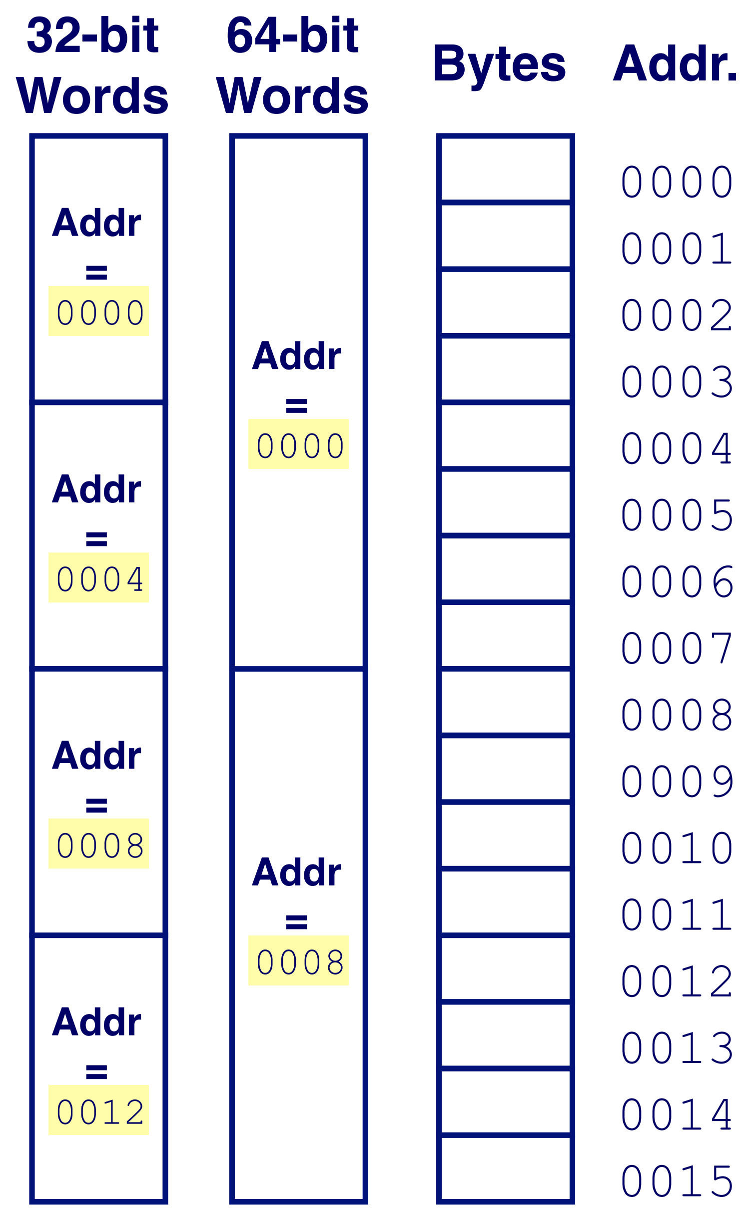
- 32-bit words => can address 4GB of data
- 64-bit words => could address up to 1.8×10^{19}

Different words sizes (integral number of bytes, multiples and fractions) are supported

Addresses

Addresses specify byte location in computer memory

- address of first byte in word
- address of following words differ by 4 (32-bit) and 8 (64-bit)



Data Types

The base data type

- **`int`** - used for integer numbers
- **`float`** - used for floating point numbers
- **`double`** - used for large floating point numbers
- **`char`** - used for characters
- **`void`** - used for functions without parameters or return value

Composite types are

- pointers to other types
- arrays of other types

Qualifiers, Modifiers & Storage

Type qualifiers

- **`short`** - decrease storage size
- **`long`** - increase storage size
- **`signed`** - request signed representation
- **`unsigned`** - request unsigned representation

Type modifier

- **`const`** - value not expected to change

Storage class

- **`static`** - variable that are global to the program
- **`extern`** - variables that are declared in another file

Sizes

Type	Range (32-bits)	Size in bytes
`signed char`	–128 to +127	1
`unsigned char`	0 to +255	1
`signed short int`	–32768 to +32767	2
`unsigned short int`	0 to +65535	2
`signed int`	–2147483648 to +2147483647	4
`unsigned int`	0 to +4294967295	4
`signed long int`	–2147483648 to +2147483647	4 or 8
`unsigned long int`	0 to +4294967295	4 or 8
`signed long long int`	–9223372036854775808 to +9223372036854775807	8
`unsigned long long int`	0 to +18446744073709551615	8
`float`	1×10^{-37} to 1×10^{37}	4
`double`	1×10^{-308} to 1×10^{308}	8
`long double`	1×10^{-308} to 1×10^{308}	8, 12, or 16

Character representation

ASCII code (American Standard Code for Information Interchange): defines 128 character codes (from 0 to 127),

Examples:

- The code for 'A' is 65
- The code for 'a' is 97
- The code for 'b' is 98
- The code for '0' is 48
- The code for '1' is 49

Strings

`"Hello"`

H	e	l	l	o	\0

- A string literal is a sequence of characters delimited by double quotes
- It has type `const char*` and is initialized with the given characters
- The compiler places a null byte (`\0`) at the end of each literal
- A double-quote (`"`) in a string literal must be preceded by a backslash (`\`)
- Creating an array of character:

```
const char* c = "Hello";
char c[6] = "Hello";
```

Declarations

The declaration of a variable allocates storage for that variable and can initialize it

```
int lower = 3, upper = 5;
char c = '\\', line[10], he[3] = "he";
float eps = 1.0e-5;
char arrdarr[10][10];
unsigned int x = 42U;
char* ardar[10];
char* a;
void* v;
```

Without an explicit initializer local variables may contain random values (static and extern are zero initialized)

Conversions

What is the meaning of an operation with operands of different types?

```
char c; int i;  … i + c …
```

The compiler will attempt to convert data types without losing information; if not possible emit a warning and convert anyway

Conversions happen for operands, function arguments, return values and right-hand side of assignments.

Conversions

T op T′: //symmetrically for T′

if `T=long double` then convert `long double`

elseif `T=double` then convert `double`

elseif `T=float` then convert `float`

elseif `T=unsigned long int` then convert `unsigned long int`

elseif `T=long int` then convert `long int`

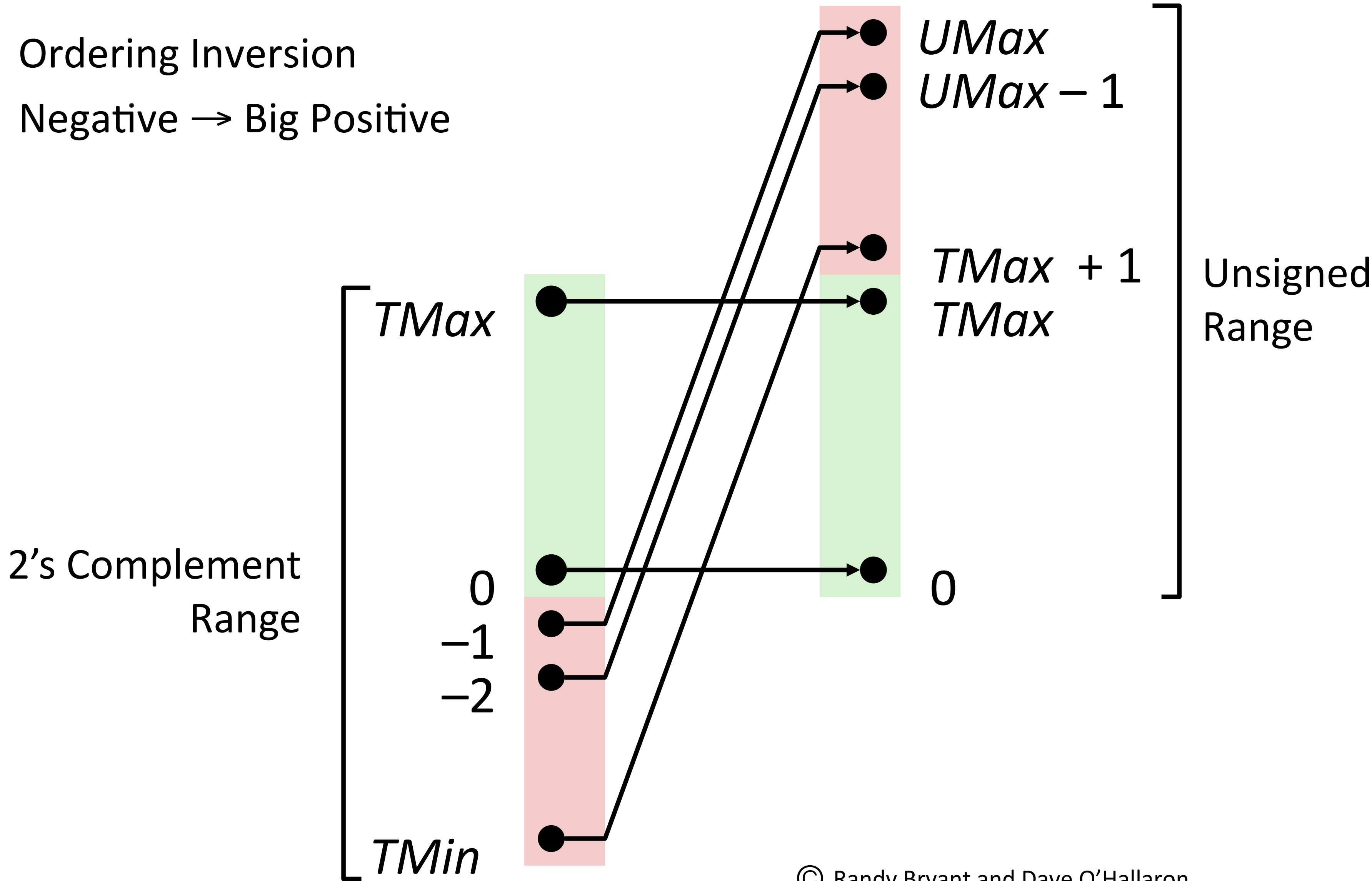
elseif `T=unsigned int` then convert `unsigned int`

- Conversions to between signed and unsigned integers slightly surprising due to *two's complement* representation (look it up)
- character can be converted to integral types

Conversion

signed to unsigned conversion

- Ordering Inversion
- Negative → Big Positive



Casts


```
static_cast<T>(x)
dynamic_cast<T*>(x)
reinterpret_cast<T>(x)
const_cast<T>(x)
```

A cast converts the value held in variable `x` to type `T`

With the exception of dynamic casts, all other casts leave the value unchanged, but return it at another type.

Dynamic casts are limited to pointers to objects, and return nullptr if the object is not of the required type

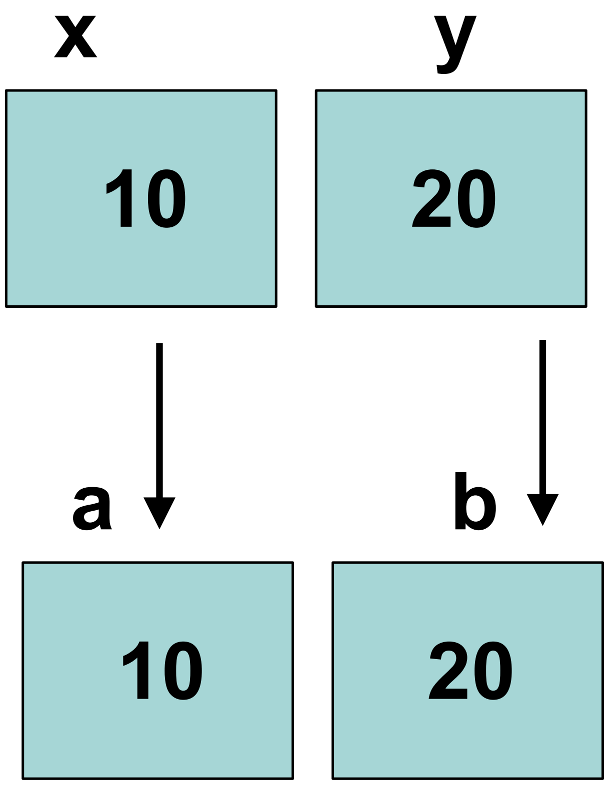
Parameter passing

By-value semantics:

- Copy of param on function entry, initialized to value passed by caller
- Updates of param inside callee made only to copy
- Caller's value is not changed (updates to param not visible after return)

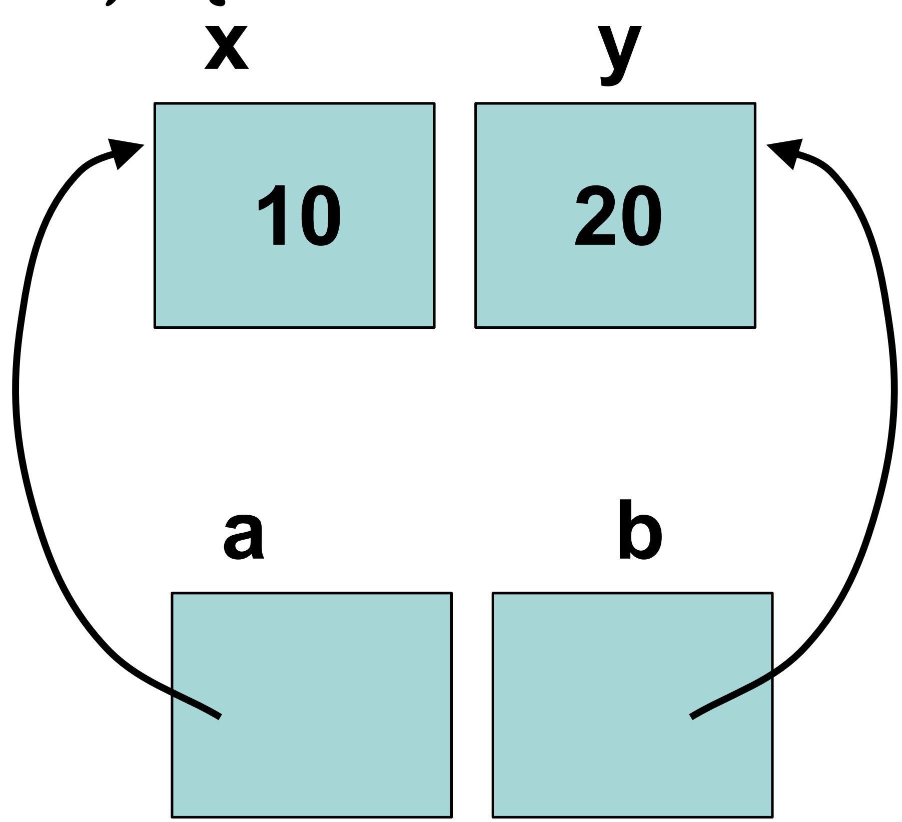
To swap or not to swap?


```
int y = 20, x = 10;
swap(x,y);

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



To swap!


```
int y = 20, x = 10;
swap(x,y);

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



Basics

`char c;` *declares a variable of type character*

`char* pc;` *declares a variable of type pointer to character*

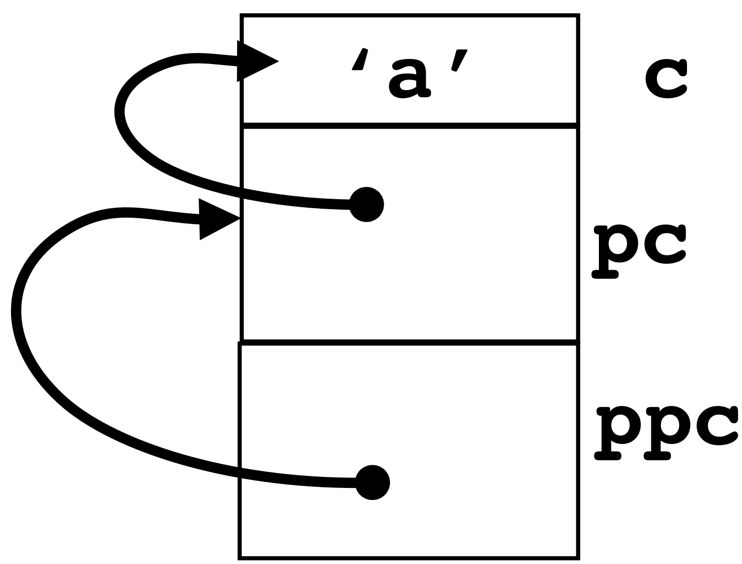
`char** ppc;` *declares a variable of type pointer to pointer to character*

`c = 'a';` *initialize a character variable*

`pc = &c;` *get the address of a variable*

`ppc = &pc;` *get the address of a variable*

`c == *pc == **ppc`



Experimenting...


```c
#include <stdio.h>

int main() {

    char c='a';
    char* pc=&c;
    char** ppc=&pc;

    printf("%p\n", pc);
    printf("%p\n", ++pc);
    printf("%p\n", ppc);
    printf("%p\n", ++ppc);
}
```

```
0x7fff6540097b
0x7fff6540097c
0x7fff65400970
0x7fff65400978
```

Basics

A variable declared as a pointer has the size of a memory address on the current architecture (e.g. 4 bytes or 8 bytes)

Incrementing a pointer adds a multiple of the pointer target size (e.g. 1 for characters, 2 for short, ...)

Pointers are initialized with addresses obtained by the **`&`** operator or the value **`nullptr`**

A pointer can be *dereferenced* by prefix a pointer value with the **`*`** operator

Attempting to dereference a **`nullptr`** pointer will result in an error caught by the hardware (bus error or segmentation fault)

Examples

`char c = 'a';`	value of c = 97, address of c=0xc00f4a20
`char* pc = &c;`	value of pc=0xc00f4a20, address of pc=0xc00eaa1c
`pc`	value 0xc00f4a20
`*pc`	value 97
`**pc`	compile warning, runtime error
`c`	value 97
`&c`	value 0xc00f4a20
`&&c`	compile error

Arrays


```
char a[2][3];
```

Creates a two dimensional array of characters

What is the value of a?

What is the address of a?

What is the relationship between arrays and pointers?

Can they be converted?

Experimenting...


```
char a[2][3];

printf("%p\n",  a      ); 0x7fff682ba976
printf("%p\n", &a      ); 0x7fff682ba976
printf("%p\n", &a[0]   ); 0x7fff682ba976
printf("%p\n", &a[0][0]); 0x7fff682ba976
printf("%p\n", &a[0][1]); 0x7fff682ba977
printf("%p\n", &a[0][2]); 0x7fff682ba978
printf("%p\n", &a[1][0]); 0x7fff682ba979
printf("%p\n", &a[1][1]); 0x7fff682ba97a
```

Arrays


```
char a[2][3];
```

An array variable's value is the address of the array's first element

A multi-dimensional array is stored in memory as a single array of the base type with all rows occurring consecutively

There is no padding or delimiters between rows

All rows are of the same size

Pointers and arrays

There is a strong relationship between pointers and arrays

```
int a[10];
int* p;
```

A pointer (e.g. **`p`**) holds an address while the name of an array (e.g. **`a`**) denotes an address

Thus it is possible to convert arrays to pointers

```
p = a;
```

Array operations have equivalent pointer operations

```
a[5]        ==        *( p + 5 )
```

Note that **`a=p`** or **`a++`** are compile-time errors.

Pointers to arrays


```
char a[2][3];
```

Multi-dimensional array that stores two strings of 3 characters. (Not necessarily zero-terminated)

```
char a[2][3]={“ah”,”oh”};
```

Array initialized with 2 zero-terminated strings.

```
char *p = &a[1];

while( *p != ‘\0’ ) p++;
```

Iterate over the second string

Pointer to pointer


```
int   i    = 5;
int  *p    = &i;
int  **pp = &p;
```

Think about it as `*pp` is an `int*`, that is, `p` is a pointer to pointer to int

```
char *s[3] = {"John", "Dan", "Christopher"};
```

// `s` is a `char **`

```
char **p = s;
```

Memory Allocation Problems

Memory leaks

- Alloc'd memory not freed appropriately
- If your program runs a long time, it will run out of memory or slow down the system
- Always add the free on all control flow paths afte a malloc

```
String *p = new String*[sz];
/*the buffer needs to double*/
String *newp = new String[sz*2];
for (int i=0;i<sz;i++) newp[i]=p[i];
p = newp;
```

Memory Allocation Problems

Use after free

- Using dealloc'd data
- Deallocating something twice
- Deallocating something that was not allocated

  Can cause unexpected behavior. For example, malloc can fail if "dead" memory is not freed.

  More insidiously, freeing a region that wasn't malloc'ed or freeing a region that is still being referenced

```
int *ptr = new int;
delete ptr;
*ptr = 7; /* Undefined behavior */
```

Memory Allocation Problems

Memory overrun

- Write in memory that was not allocated
- The program will exit with segmentation fault
- Overwrite memory: unexpected behavior


```
int* y= …
int* x= y++;
for(p=x; p>y; p++)
    *p=42;
```

Memory Allocation Problems

Fragmentation

- The system may have enough memory but not in contiguous region


```
int* vals[10000];

int i;
for (i = 0; i < 10000; i++)
  vals[i] = new int*;

for (i = 0; i < 10000; i = i + 2)
  delete vals[i];
```

A gentle recap of the story so far

strip.c


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

int main() {
 int c = 0, in = 0;
 char buf[2048]; char *p = buf;

 while((c = getchar()) != EOF) {
  if(c=='<' || c=='&') in=1;
  if(in) *p++=c;
  if(c=='>' || c==';') {
    in = 0;
    *p++ = '\0';
    if(strstr(buf,"nbsp")||strstr(buf,"NBSP"))
       printf(" ");
    p = buf;
  } else if(!in) printf("%c", c);
 }
}
```

Includes


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

- Tell the compiler about external functions that may be used by the program
- Pre-processor directives, expended early in the compilation
- `stdio` defines functions `getchar/printf`
- `string` defines `strstr`

Main


```
int main() {
    return 0;
}
```

- C programs must have a `main()` function
- `main()` called first when the program is started by the OS
- `main()` returns an integer
- without a `return` statement, undefined value is returned
- The correct signature for `main()` is:

```
int main(int argc, const char* argv[]) { }
```

Getchar/printf


```c
int c = 0;

while((c = getchar()) != EOF)
      printf("%c", c);
```

- `getchar()` returns 1 character from "standard input" converted to an int
- If the stream is at end-of-file or a read error occurs, `EOF` is returned
- `printf()` outputs a string to the standard output
- `printf()` takes a format string and a variable numbers of arguments that are converted to characters according to the requested format

Looping


```
int c = getchar();
while(c != EOF) {
    printf("%c", c);
    c = getchar();
}
```

- another way to express the same behavior
- assignments are expressions, the same program without nesting

Arrays & pointers


```
int c = 0, in = 0;
char buf[2048]; char *p = buf;

while((c = getchar()) != EOF) {
    if(c=='<' || c=='&')  in = 1;
    if(in)  *p++=c;
    if(c=='>' || c==';') {
        in = 0; *p++ = '\0';
    }
}
```

- `buf` is an array of 2048 characters;
- `p` is pointer in the buffer
- boolean value `false` is 0, any non-0 is `true`

Arrays


```
char buf[2048]; int pos=0;
while((c = getchar()) != EOF) {
  ...
  if(in) buf[pos++] = c;
  if(c=='>'||c==';') {
    buf[pos++]='\0';
    pos=0;
  }
```

- the same program without pointers
- an alternative to pointers is to use an index in the array of chars
- strings must be `\0` terminated (or risk a buffer overflow…)

Strstr


```
char buf[2048]; char *p = buf;
...
 if(state) *p++=c;
 ...
   *p++ = '\0';
   if(strstr(buf,"nbsp")||strstr(buf,"NBSP"))
      printf(" ");
   p = buf;
```

- `strstr(s1,s2)` locates the first occurrence of `s2` in `s1`.
- if `s2` occurs nowhere in `s1`, `nullptr` is returned; otherwise a pointer to the first character of the first occurrence of `s2` is returned
- `nullptr` is `false`, `||` is logical or

strip.c


```
int main() {
 int c = 0, in = 0;
 char buf[2048]; char *p = buf;
 while((c = getchar()) != EOF) {
  if(c=='<' || c=='&') in=1;
  if(in) *p++=c;
  if(c=='>' || c==';') {
    in = 0;
    *p++ = '\0';
    if(strstr(buf,"nbsp")||strstr(buf,"NBSP"))
       printf(" ");
    p = buf;
  } else if(!in) printf("%c", c);
 }
}
```

Arrays


```
char buf[2048];
buff[0] = ‘a’; buff[1] = buff[0];
```

- Array variables a declared with the `T[]` syntax
- Items that are not explicitly initialized will have an indeterminate value unless the array is of `static` storage duration
- Initialize x as a one-dimensional array with 3 members, because no size was specified and there are 3 initializers:

```
int x[] ={1,3,5};
```

- Bracketed initialization: `1`, `3`, and `5` initialize the first row of the array `y[0]`, namely `y[0][0],...` The initializer ends early:

```
float y[3][3] = {
        { 1, 3, 5 },
        { 2, 4, 6 },
        { 3, 5, 7 } };
```

Of chars and ints & conversions


```
int c;
char buf[1];
c = getchar();
buf[0] = c;
```

- Conversions from an integer value to a character do not lose information if the integer is in the valid range for characters
- The value EOF is not a valid character value


Files

Stdio.h

Provides general operations on files

A file is an abstraction of a non-volatile memory region:

- its contents remain even after the program exits
- it exposes the file abstraction using the **FILE** type:

  FILE *fp // *fp is a pointer to a file
- Can only access the file using the interfaces provided

File Systems

A file system specifies how information is organized on disk and accessed

- directories
- files

In UNIX the following are files

- Peripheral devices (keyboard, screen, etc
- Pipes (inter process communication)
- Sockets (communication via computer networks)

Files representation

- Text files (human readable format)
- Binaries (for example executables files)

File manipulation

Three basic actions:

- open the file: make the file available for manipulation
- read and write its contents

  No guarantee that these operations actually propagate effects to the underlying file system
- close the file: enforce that all the effects to the file are “committed”

File Descriptors

To operate on a file, the file must be opened

An open file has a non-negative integer called file descriptor

For each program the OS opens implicitly three files: standard input, standard output and standard error, that have associated the file descriptors 0, 1, 2 respectively

- Primitive, low-level interface to input and output operations
- Must be used for control operations that are specific to a particular kind of device.

Streams

Higher-level interface, layered on top of file descriptor facilities

More powerful set of functions

Implemented in terms of file descriptors

- the file descriptor can be extracted from a stream and used for low-level operations
- a file can be open as a file descriptor and then make a stream associated with that file descriptor.

Opening files


```
FILE* fopen(const char* filename, const char* mode)
```

- mode can be "r" (read), "w" (write), "a" (append)
- returns NULL on error (e.g., improper permissions)
- filename is a string that holds the name of the file on disk

```
int fileno(FILE *stream)
```

- returns the file descriptor associated with stream

```
char *mode = "r";
FILE* ifp = fopen("in.list", mode);
if(ifp==NULL){fprintf(stderr,"Failed");exit(1);}
FILE* ofp = fopen("out.list", "w");
if (ofp==NULL) {...}
```

Reading files

`fscanf` requires a **`FILE*`** for the file to be read

```
fscanf(ifp, “<format string>”, inputs)
```

Returns the number of values read or EOF on an end of file

Example: Suppose in.list contains

```
foo 70
bar 50
```

To read elements from this file, we might write

```
fscanf(ifp, “%s  %d”, name, count)
```

Can check against EOF:

```
while(fscanf(ifp,“%s %d”,name,count)!=EOF);
```

Testing EOF

Ill-formed input may confuse comparison with EOF

`fscanf` returns the number of successful matched items

```
while(fscanf(ifp,“%s  %d”,name,count)==2)
```

Can also use **`feof`**:

```
while (!feof(ifp))  {
 if (fscanf(ifp,“%s  %d”,name,count)!=2) break;
 fprintf(ofp, format, control)
}
```

Closing files


```
fclose(ifp);
```

Why do we need to close a file?

File systems typically buffer output

```
fprintf(ofp, “Some text”)
```

There is no guarantee that the string has been written to disk

Could be stored in a file buffer maintained in memory

The buffer is flushed when the file is closed, or when full

Raw I/O

Read at most **nobj** items of size **sz** from **stream** into **p**

feof and **ferror** used to test end of file

```
size_t fread(void* p,size_t sz,size_t nobj,FILE* stream)
```

Write at most **nobj** items of size **sz** from **p** onto **stream**

```
size_t fwrite(void*p,size_t sz,size_t nobj,FILE* stream)
```

File position

`int fseek(FILE* stream, long offset, int origin)`

Set file position in the stream. Subsequent reads and writes begin at this location

Origin can be **`SEEK_SET`**,**`SEEK_CUR`**,**`SEEK_END`** for binary files

For text streams, offset must be zero (or value returned by **`ftell`**)

`Return the current position within the stream`

`long ftell(FILE * stream)`

`Sets the file to the beginning of the file`

`void rewind(FILE * stream)`

Example


```
#include <stdio.h>
int main() {
 long fsize;
 FILE *f;

 f = fopen(“log”, “r”);

 fseek(f, 0, SEEK_END) ;
 fsize =  ftell(f) ;
 printf(“file size is: %d\n”, fsize);

 fclose(f);
}
```

Text Stream I/O Read

Read next char from stream and return it as an unsigned char cast to an int, or EOF

```
int fgetc(FILE * stream)
```

Reads in at most size-1 chars from the stream and stores them into null-terminated buffer pointed s. Stop on EOF or error

```
char* fgets(char *s, int size, FILE  *stream)
```

Writes c as an unsigned char to stream and returns the char

```
int fputc (int c, FILE * stream)
```

Writes string s without null termination; returns a non-negative number on success, or EOF on error

```
int fputs(const char *s, FILE *stream)
```