C was developed specifically for writing operating systems
   Low level of abstraction.
   "Just above machine language."
   Direct access to the machine when desired.
   Highly efficient.

Has two predecessors and one successor:
   A: assembler.
   B: block program structure (but no block data).
   C: block structure and block data.
   (D: adds memory descriptors)

C++ is not "more powerful" than C; it is just a
((somewhat) safer) dialect.
C programmers know the cost of everything but the value of nothing. Lisp programmers know the value of everything but the cost of nothing.
Speed and danger
○ C is an "unsafe" language.
○ No bounds checking on arrays
○ No type checking during subroutine calls.
○ Pointers (at whatever cost)
○ Computed gotos (using function pointers).
○ Why? **Speed is more important than safety.**
C design principles
  ○ Trust the programmer.
  ○ Speed over correctness.
  ○ No run-time checking.
  ○ Information hiding by not exposing structure.
#include <stdio.h>
int main(int argc, char **argv, char **envp) {
    printf("hello\n");
    exit(0);
}

#include <stdio.h>

/* include a header file "stdio.h"
containing reusable declarations */

int main(int argc, char **argv, char **envp)
/* program is always called "main".
argc is number of arguments on the
command line,
argv is an array of arguments (as
strings).
envp is an array of environment
variables. */
{
    printf("hello\n");
    /* call the library function "printf" */
    exit(0);
    /* stop running with "success" code:
    system call! */
}


To compile a C program "hello.c", on comp111
   gcc -g -c hello.c
   # creates hello.o, a relocatable object file
   gcc -o hello hello.o
   # creates hello from hello.o
   ./hello
   # run the program

Or, alternatively
   gcc -g hello.c
   # means
   gcc -o a.out -g hello.c
There are profound differences between C and higher-level languages:

<table>
<thead>
<tr>
<th>C</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>No storage descriptors at runtime (gdb and gcc -g option construct storage descriptors for debugging only)</td>
<td>Storage descriptors store type of each object</td>
</tr>
<tr>
<td>Pointers</td>
<td>References</td>
</tr>
<tr>
<td>Programmer manages memory recovery (explicit free)</td>
<td>Garbage collection is automatic, based upon storage descriptors (implicit free)</td>
</tr>
<tr>
<td>No type reflection</td>
<td>Can query the type of any object at runtime</td>
</tr>
<tr>
<td>No factories</td>
<td>Can build objects based upon dynamic information using factories</td>
</tr>
</tbody>
</table>

C for Java/Python/C# programmers
Wednesday, September 19, 2018 9:12 AM
### Some really basic differences between C and Java

<table>
<thead>
<tr>
<th>C</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>No boolean type; integers are interpreted as boolean</td>
<td>Boolean type for logic</td>
</tr>
<tr>
<td>Nonzero is true, Zero is false</td>
<td>True and False are boolean values.</td>
</tr>
<tr>
<td>Logical operations output 1 or 0 as integers</td>
<td>Logical operations return True or False</td>
</tr>
</tbody>
</table>
The code:

```c
if (fork()) {
    // in the parent
} else {
    // in the child
}
```

is equivalent with

```c
pid = fork()
if (pid != 0) { // nonzero is True
    // in the parent
} else {
    // in the child
}
```
Assignments have value

Every expression has a value, including assignment statements. The statement

    a = b = c;

is right associative, and equivalent with

    a = (b = c);

The value of \( a = b \) is the value of \( b \) cast to the type of \( a \).
a += b  means  a = a + b
a -= b  means  a = a - b
a *= b  means  a = a * b
a /= b  means  a = a / b
++a  means  a = a + 1
--a  means  a = a - 1
a++  means  (tmp = a; a = a + 1, tmp)
a--  means  (tmp = a; a = a - 1; tmp)
(a, b) -- outside function calls, is just b.
The syntax
  for (a; b; c) d;
means
  a;
  while (!(b)) {
    d;
    c;
  }

Specifically, the syntax:
  for (i=0; i<n; ++i) printf("i=%d\n",i);
means
  i=0;
  while (!(i<n)) {
    printf("i=%d\n",i)
    ++i;
  }

In turn,
  while(a) b;
is equivalent with:
  back:
    if a goto out;
  b;
  goto back;
out:
which is a fairly accurate representation of the machine language that implements it.
An advanced C programmer will save computation cycles via "expression folding". For example,

```c
if ((pid = fork()) {
    // in parent
} else {
    // in child
}
```

- `pid = fork()` sets `pid` to the return value of `fork()`, but also returns that value to the if statement!
- Much more complex expression folding is common:
  ```c
  if ((status = system("cat /etc/motd")) != 0) {
      // command failed
  }
  ```
  Means
  ```c
  status = system("cat /etc/motd")
  if (status != 0) {
      // command failed
  }
  ```
In C, an integer can have several meanings, depending upon operations used on it:

1. A regular integer, representing a positive or negative count. Operations include +, -, *, /
2. A bit field of binary bits, e.g., an array of bits. Operations include &, |, ^, ~, <<, >>
3. A logical value, where nonzero is true and zero is false. Operations include &&, ||, !.
1<<a: one left shift a: a bitfield with bit a equal to 1, others zero.
a&b: bit i of a&b is 1 if both bits i of a and b are 1, zero otherwise.
a|b: bit i of a|b is 1 if either bit i of a or bit i of b is 1.
a^b: bit i of a^b is 1 if exactly one of bit i of a or bit i of b is 1. (exclusive or)
~a: bit i of ~a is 1 - bit i of a (complement of a as a set)
Nonzero means True.
Zero means False.
a && b: logical and of a and b
a || b: logical or of a and b
! a: negation of a

Logical **duality:**
• Logical values: nonzero is True
• Values of connectives: 1 or 0
if (c = 1) {
    // always happens and c is set to 1!
}
Correct:
if (c == 1) {
    // when c is 1
}

if (a & b) {
    // doesn't work properly if a or b is not 1 or 0
}
Correct:
if (a && b) {
    // only if a and b are non-zero
}
It is possible to create two variables with the exact same name in the same program:
{
    int i = 2;
    {
        int i = 1;
        printf("i=%d\n", i);
    }
    printf("i=%d\n", i);
}
int foo(int i) { int tmp = i; i=42; return tmp+1; }

- All arguments are copied.
- Changing an argument doesn't change the outside world.
- A copy of a pointer points to the same place.

int bar(int *i) { *i=7; return i; }

- Sets the location pointed to by I to 7.
- Returns that value as an integer.
- To call this
  
  ```c
  int j;
  bar(&j);  // address of j  
  printf("j=%d\n", j);
  ```
C is not polymorphic: foo(x) and foo(y, z) call the very same function.

C is type-unsafe: the function foo is responsible for figuring out how to use its arguments; types of arguments are often not enforced!
Some very common function problems

foo(1);
    // returns 2, which is discarded.

foo;
    // returns the location of foo (a pointer to function), which
    // is discarded. **Does not call the function.**

j = foo();
    // calls foo on a random integer that just happens
    // to be on the stack, and returns that value + 1.

k = foo(2.1);
    // calls foo on a double-precision 2.1e0, which is not
    // sensible. Garbage results.
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Wall</td>
<td>Warn about every potential problem in the code.</td>
</tr>
<tr>
<td>-Wshadow</td>
<td>Warn about shadowed variables. Shadowing is not a programming error!</td>
</tr>
<tr>
<td>-g</td>
<td>Store type descriptors in extra storage at the end of the a.out file, for use in gdb</td>
</tr>
</tbody>
</table>
Using the debugger: gdb or xxgdb
to diagnose a core dump.
to single-step a program.

How to debug memory allocation problems
(malloc and free)
what can happen?
how do you find it?

How caches work, and how they affect performance
Using gdb

Monday, October 2, 2017  12:11 PM

gcc -g file.c
gdb a.out
  run
  run arg-list
  break location
  print variable
  exit

gdb a.out core.something
  exhibits the situation during a core dump.
  Allows you to print values
The big example from 40 that we will constantly use: the effects upon caches upon runtime

A cache is a copy of memory used because it can be accessed more quickly.

Facts about caches
- limited size
- updated at page level.
- pages can be "clean" or "dirty"

A "dirty page" must be written back to memory.
A "clean page" can be deleted from cache without writing to memory.

The performance of a program is based upon its data locality: the extent to which it manipulates data in a local region so that it stays in cache.
valgrind:
simulates malloc and free
keeps detailed notes about what is legal.
stops a program if it tries to do something illegal with memory.
printf("format", arguments...) is general printing statement. "format" represents a string, which is an array of characters.
   "string" a string
   's' a character

"format" can contain:
   %d: print an integer
   %f: print a floating point number
   %s: print a string
   %%%: print a percent character(!)
   \n: print a carriage return

printf("one is %d \n", 1);
   => "one is 1\n"
printf("two point zero is %f \n", 2.0);
   => "two point zero is 2.0"

printf("these are %d, %f\n", 1, 2.0);

man printf tells all.
#include <stdio.h>
int main(int argc, char **argv, char **envp) {
    int i;
    printf("ARGUMENTS\n");
    for (i=0; i<argc; i++)
        printf("argument %d is '%s'\n", i, argv[i]);
    printf("ENVIRONMENT\n");
    for (i=0; envp[i]!=NULL; i++)
        printf("environment variable %d is '%s'\n", i, envp[i]);
}

argv[0]-argv[argc-1] are the command-line arguments. 
envp[0]-... are the environment variables. 
envp[i]==0 means stop listing!
printf is not part of C. Instead, it is a library function that is documented in the linux manual pages. From the linux command line, type

    man printf

or

    man 3 printf # check in section 3: the C library to learn about it.
```c
#include <stdio.h>
int main(int argc, char **argv, char **envp) {
    int pid = fork(); /* a system call */
    if (pid!=0) {
        printf("I'm the parent\n");
        sleep(5); /* library */
        printf("killing pid %d\n", pid);
        kill(pid,9); /* kill my child! system call */
        printf("killed pid %d\n", pid);
    } else { /* I'm the child: wait to die! */
        int count = 0;
        printf("I'm the child!\n");
        while(1) {
            sleep(1);
            ++count;
            printf("I've been around for %d seconds\n",count);
        }
    }
    exit(0);
}
```
Explanation of the program

fork(): split into two processes.
sleep(5): sleep for five seconds.
kill(pid,9): send a kill signal (9) to the process numbered pid.

For more details

man 2 kill
man 2 fork
man 3 sleep

kill is a system call: these are documented in section 2 of the manual.

Note that

man 1 kill
or
man kill

describes the kill command, which is different.

Really useful:

man -k thing
returns a list of all manual pages whose title mentions thing.
There are three things to know in programming in C:
  the language C itself
  how to use the available system calls (man 2 x)
  how to use the available library functions (man 3 x)

Using a function
  read the man page.
  #include the appropriate headers (e.g., <stdio.h> for printf)
  Call the function with the appropriate arguments (e.g.,
  printf("%d\n",i))
Note that an expert C programmer is always conscious of speed of execution.

Note that
while(x!=0) { printf("x is %d\n",x); x=x-1; }
means exactly the same thing as
while(x) { printf("x is %d\n", x); --x; }
but the second is a bit faster. An expert would not write the first version!
You will commonly see very terse C in OS code.

One can:
  test for whether a pointer is NULL (0) by treating it as boolean!
  test for whether an integer is 0 by treating it as boolean!
  etc.

const char *p; p = "hello\n";
const char *q;
q=p;
while (*q) { printf("%c",*q); q++; }

The optimizer debate:
"New school" programmers contend that this is handled by the optimizer.
"Old school" programmers are more explicit and do not rely upon that.
Each statement has a **value**.

- \(x=1\) has value 1
- \(x=y+z\) has the value of \(x\) after the assignment.
- \(++x\) has the value of \(x\) **after** increment
- \(x++\) has the value of \(x\) **before** increment

E.g.

\[
x=1 \\
y=(x++)
\]

sets \(x\) to 2 and \(y\) to 1!

Reason for this: **loop optimization**. Consider:

\[
x=0; \text{while}(x<20) \ a[x++]=\text{0};
\]

This sets elements \(a[0]\) to \(a[19]\) to 0, but stops before element 20.

Equivalent to:

\[
x=0; \text{while} \ (x<20) \ {\{ \ a[x]=\text{0}; \ x=x+1; \}}
\]

but the former avoids a register reload.
C structures
  ○ Concatenation of basic types
    ○ Example:
    
    ```c
    struct foo {
      int i;
      double d;
    } s;
    ```

    ANSI standard layout:
    "Lay out structure by ascending address, but align each type on a **multiple of its length.**"
struct foo x;
x.i=4;
x.d=5.6;
struct foo *p = &x;
p->i=7;
C has no classes

Structs: the data part of a class.
The function part requires polymorphism, which C doesn't have.
You will see OS programmers "growing their own classes".

Ex:

```c
struct foo {
    ...
};
struct foo *foo_create();
void foo_destroy(struct foo *);
void foo_set(struct foo *, int i);
...
```
Unions

- Allow one to create aliases for the same kind of memory.
- Example
  ```c
  union g {
    int i;
    double d;
  } t;
  ```
  t.i contains the first eight bytes of t.d

Very often, in programming an OS, we need to decide on the fly what kind of data a block contains. Unions allow us to define things that can represent multiple kinds of things, very efficiently.

```c
struct discriminated_union {
  enum {WROTE_I, WROTE_D} discriminant;

  union g {
    int i;
    double d;
  } t;
};
```
There are several mechanisms for information hiding in C that are different than mechanisms in C++
1. void *: hide the whole type.
2. The "prefix rule": hide part of a type
Remember that a pointer type determines lots of things about the pointer:

- What *p means
- What p+1 means

These are relative to the size of the type

Some times, we don't want to tell the function anything at all about the data.

For this reason, we point to that data with a "pointer to void"

```
void *p
```

This gets complicated really fast, as we will see in the next lecture:

- Type is void * from the point of view of the operating system.
- Type is something tangible from the point of view of the program
- Type is again void * whenever the operating system has to do something with it.

There is a really good reason that malloc returns void * and free takes an argument that is void *.
They do not want to know about the actual type of the data.
The Prefix Rule

- If one structure is a prefix of another, the two structures are organized in memory identically for the common prefix.

- Example:
  ```c
  struct foo {
    int i;
    double d;
  } s;
  struct foo2 {
    int i;
    double d;
    float f;
  } u;
  ```
C Information Hiding
  ○ If you want to hide something about a struct, give the user a pointer to a prefix of the struct; leave other data unavailable.
  ○ Example:
    struct foo2 h;
    struct foo *j = (struct foo *) &h;
    return j;

    "--- unused ---"

    j -> i  j -> d  no j -> f

    Unfortunately, I can get around this:
    If j is as above, then
    *((float*)((double*)((int*)j)+2)+1))
    refers to the element we tried to hide.
The prefix rule is exactly what enables C++ subclassing.

- The prefix is the superclass.
- The whole structure is the subclass.