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 gototos (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
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);
#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.
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))
C is not type-safe

Note that

```c
printf("%f",2.1);
```

prints what you would expect, while

```c
printf("%d",2.1);
```

prints (predictable but useless) garbage.

Why? The second tries to print (the first half of) a double-precision number as an integer, which doesn't work the way one might expect!
C is not polymorphic
   foo(int) and foo(float) are the same function.
   You must distinguish between argument types!
Note that an expert C programmer is always conscious of speed of execution.

Note that

```c
while(x!=0) { printf("x is %d\n",x); x=x-1; }
```

means exactly the same thing as

```c
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.

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

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

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

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:

```c
struct foo2 h;
struct foo *j = (struct foo *) &h;
return j;
```

Unfortunately, I can get around this:
If `j` is as above, then
```c
*((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.