People who have taken CS2617 should instead read the full set of notes (cnotes).

Note: 2nd ed of textbook was for C-89 (classic "ANSI-C". Third is based on the 2018 revision. There were many changes in 30 years.

C-11 and C-99 are also in widespread use.

A new version (currently called C-2X) is expected in 2021 or 2022. Change look fairly small (new "decimal floats", if you care.)
The Preprocessor

Unlike Java, the C compiler has a "preprocessor" that does textual substitutions and "conditional compilation" before the "real compiler" runs.

This was borrowed from assembly languages.

Fancy uses of preprocessor are often deprecated, especially since C99.

Still commonly used use for defining constants and in doing the C equivalent to Java’s "import".

Preprocessor "directives" start with #. No ‘;’ at end
Macros in the C Preprocessor
--------------------------------

#define FOO 12*2

thereafter, wherever the symbol FOO occurs in your code, it is textually replaced by 12*2. Think find-and-replace.

Macro can have parameters

#define BAR(x) 12*x

If we have BAR(3*z) in your code, it is textually replaced by 12*3*z

#undef BAR

will "undefine" BAR. After this, BAR(3*z) remains as is.
Perils of Macros

#define SQUARE(x)  ((x)*(x))

used much later with

int x = 0;
int y = SQUARE(x++);

What value for x is expected by a programmer looking only at these lines?

What value does x actually have?
Making Strings and Concatenating Tokens

(optional fancy topic)

In a macro definition, the # operator puts quote marks around a macro argument.

#define STRINGIFYME(x) # x

STRINGIFYME(owen) expands to "owen"

The ## operator joins two tokens together.

#define DECL_FUNC(name) int name ## _xy (int z)\
    return(3 + name);

then DECL_FUNC(owen) ends up defining a function called owen_xy that returns 3 + owen
Predefined Macros

__________

__FILE__ and __LINE__ expand to the current src
code file name and line number. Good for
auto-generated warning messages etc.

__func__ name of current function (C99)
Conditional Compilation in C
----------------------------------------

#if ZZ == 1       #ifdef FOO
...somecode1...  .....  
#else            #else
...somecode2...  .....  
#endif          #endif

also #ifndef

Check a Boolean condition that can be evaluated at compile time, or chk definedness

** Determines whether somecode1 or somecode2 is seen by the real compiler

Only one of the 2 sequences is used **
Conditional Compilation part II
---------------------------------

Conditional compilation is often used to tailor the code for a particular models of computer.

And to ensure that a given "include file" isn't seen too many times (more later)
#include statement
----------------

#include "foo.h" will cause the contents of your file foo.h to be inserted in place of the #include statement.

"copy paste the file here, please"

#include is usually used like "import" in Java. It makes available library things.

Most C programs have a line #include <stdio.h>

since you can’t print otherwise.

(note angle brackets for system libraries)
Simple C program
--------------

#include <stdio.h>
#define N 5
int g;

// C99-style comment: program needs a main()
int main(void) {
    int i;
    
    g = N;
    for (i=0; i < g; ++i)
        printf("Hello world");
}
Stack Frames

Compilers generate stack frames according to the approved calling conventions for that system.

AMD has a long document giving the recommended calling conventions. Apparently GCC/Linux follow it, but Windows has its own ideas.

https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64

Interesting Red Zone (above the TOS) where you can put temporaries or even a whole frame!

Note the "Frame Pointer". Some calling conventions use frame pointers (aka BP); others don’t.
Important Difference with Java

C was originally envisioned as a platform dependent language.

So an int means different things on different platforms. (Is it 16 bits? 32 bits? 64 bits?)

C can be used on systems that don’t use 2’s complement

    short must be able to represent -32767 to 32767.
    int   must be able to represent  -32767 to 32767.
    long: -2147483647 to 2147483647

But ranges can be larger. However, range of short cannot exceed range of int, which cannot exceed range of long.
Fixed-Width Integer Types

It’s hard to do some low-level programming without knowing bit width. So C99 adds header file `<stdint.h>`

Defines data types like `int32_t`, `uint64_t` etc.

`int32_t` is guaranteed to give 32 bits range

There are macros to declare constants of these types and for printf format strings

Eg, `uint64_t x = UINT64_C(12345);`
    `printf("Value is %" PRIu64 "\n", x);"
Signed vs Unsigned Overflow
-------------------------------------

Since C was defined support some weird architectures that don’t use 2’s complement, what happens if your signed arithmetic overflows is undefined. Compilers would be free to generate code that behaves strangely. And they can! :

```c
int i=0;
while (true) {
    if (i < 0) printf("neg"); // ****
    i = i+1;
}
```

Compiler within its rights to decide to remove ****.

Contrast to Java, Go, etc where 2’s complement behaviour is required.

Unsigned numbers are defined to wrap to 0 sanely.
String Tidbits

In C, strings are just null-terminated char arrays. Just like in assembly language.

Libraries have many convenient string functions.

Join strings with `strncat`. No `+` operator, sadly.

Hint: Linux command `man 3 strcat` tells you all about the `strcat` function, which is in section 3 (C library) of manuals.

In source code, consecutive string literals are concatenated by compiler (at least C99 and on).

Handy; makes legal

```c
printf("%" PRIu64 " is value of x\n",x);
```

or splitting long string constants over several source code lines.
Volatile
--------

Besides "const" qualifiers, types can also be qualified with volatile. (Same keyword with similar meaning exists in Java but is not widely used)

Volatile means that the compiler cannot assume that the value won’t change, mysteriously.

(eg, value coming from a hardware device or value that another thread could be messing with)

volatile int v;
....
while (v) {...something...}

is NOT equivalent to

if (v)
  while (true) {...something...}
The Harder Parts of C
---------------------

* Pointers. Lower level access to memory than Java
  Chapter from K&R in D2L. Read carefully.
* Memory Management. Lower level.
  Read wikibook section.
* How nontrivial programs are segmented into modules
  Read my notes on the website
Const and Pointer Declarations

const int *p;  <-- p cannot be used to change the pointee
               p can be reassigned to point to other constant integers

int * const q; <-- q points to an integer
               q can be used to alter the pointee
               q cannot be reassigned.

const int * const r <-- r points to an integer
                       r cannot be reassigned, nor can r be used to change pointee
Const pointers and typecasting
-----------------------------------

```c
void foo( int *p) {...do stuff with p...}

int bar( const int *q){...
  foo(q);...
}
```

You get an error or warning about implicitly stripping the const-ness from q when handing it to foo.

You _should_ make
  ```c
  int foo (const int *p) if foo can guarantee this
  or
  int bar (int *q) if foo cannot.
  ```

or do explicit cast `foo( (int *) q)` to shut up compiler but get officially undefined behaviour.
More on the Heap

The heap is a region of memory, where chunks of this memory get handed out on demand.

Demand for C: malloc(). For Java: "new". Later, some of these chunks may become reusable, because a C programmer called free().

The heap starts out at some size. But eventually a malloc() may find there is no free block that is big enough.

Maybe all memory is in use. Maybe not, but only small blocks are available.

When this happens, the heap needs to grow. It expands dynamically as needed (OS controls).
Memory Layout

So we have the stack, which occupies no set amount of space. It grows and shrinks dynamically. And we now have the heap, which also grows (and maybe shrinks) dynamically.

The processor has a limited number of memory addresses. We’d like to be able to support almost all addresses for the stack (highly recursive pgm) or almost all addresses for the heap (malloc-intensive).

Classic approach: Stack starts at a very high address and grows down; heap starts at a small address and grows up. As long as the top of the stack doesn’t move into the heap area, we are okay.
Other Memory Areas

Other possible regions of memory are for
* your machine code
* your global initialized variables
* your global uninitialized variables
* constants

[Draw picture on board]

Machine-code and constants can be in read-only or flash memory, whereas global variables, stack and heap need to be in RAM. Constants can be a part of the global initialized region, or (as with LC3/ARM examples) in with the machine code.

Many calling conventions dedicate registers to point at the bases of the global variable or constants regions. Facilitate use of the CPU’s base+offset addressing modes
Putting Code into RAM

Although a single-purpose controller can have its program in ROM or flash memory, if you want to have an OS load various apps or programs on demand, you will need for all memory areas to be in RAM.

Putting code into RAM enables a historical horror called self-modifying code. The program can overwrite some of its own machine code while it is running. Commonly write NOP (no-op) instructions on top of things you want to disable. Or, write BL instructions into strategically placed NOPs in the code.
Evils of Self-Modifying Code

+ Every software engineering principle known to man is probably violated. You cannot easily tell what the code is going to do by inspecting the source code – what happens at runtime is determined by earlier stuff that happened at runtime.

+ Hardware designers can boost performance if they can assume that the machine codes are unchanging at the time the program starts running. (In 3813, you might see that this kind of assumption can simplify the "instruction cache" design.)
Arrays in C

---

int A[20], B[20][30];  <-- declare a 1D and 2D array

1D case is similar to Java
   int [] A = new int[20];

However, with C if the declaration is outside of any function (or is in a function, but prefixed with "static"), the bytes are allocated in the same memory region as other uninitialized global variables (which will be set to 0, same as Java).

A non-static array as a local variable takes up space in the activation record. Different from Java. If uninitialized, its value is weird junk, unlike Java.
Bounds checking
-------------
In Java, all array accesses are checked to see whether there is a bounds error.

Not the case in C, where you will read junk and write corruption...

C11 may help with this.
Array Initializers

Similar to Java, C arrays can be initialized

```c
int a[] = {3,1,4,1,5};

char *names[] = {"bob","penny","yulin"};

int b[2][2]={ {1,2},{3,4} };```

If a global, it will be stored in a memory region with other initialized globals (different region than the uninitialized globals).
Pointers and Arrays (review?)

Pointers and arrays - very closely related in C.

```c
int A[10];  // similar to Java
            int [] A = new int[10];
int *pa = A;  // allowed
A = pa;   // not allowed, array name is constant
*pa = 5;   // same as A[0] = 5
*(pa+i)    is same as A[i] or pa[i]
```
Pointer Arithmetic

To make it so that *(pa+i) means same as pa[i], we need that pa+i points to the address of the ith element of the array. If each array element occupies k bytes, we need the memory address of (pa+i) to be i*k bytes after the address of pa.

So: What does while ( *p++){} do?

If pa and pb are pointers to different parts of the same array, you are allowed to subtract them. Result is the number of array elements between them.
Pointer Comparisons

If p and q point to elements of the same array (or fields of the same object), then it is allowed to compare them for ==, !=, >, >= etc.

The integer 0 is used as NULL in C.

In earlier versions C, pointers are integers were somewhat interchangeable. Not any more. However, 0 is special.

It is okay to do a pointer comparison between a pointer within an array and a pointer that is ONE past the end of an array.

0 (as a pointer or integer) is equivalent to false in C.

Hence

\[
\text{if } ( \text{p } \&\& \text{ *p } == 'a' ) \quad \ldots.. \quad \text{or } \quad \text{if } ( \text{p } != 0 \quad \&\& \text{ *p } == 'a' ) \quad \ldots..
\]
Malloc, Calloc, Realloc and Free
---------------------------------

Malloc allocates (uninitialized) memory on the heap.

So does calloc(n,s) where n is the number of items and s is the size of each. All bytes zeroed.

realloc(p, s) attempts to enlarge the malloc’ed area at p to size s. If this can’t be done, it mallocs a new area of size s and copies the data at p over to it. Returns a pointer, regardless

free(p) releases the previously allocated memory. It can then be reused.

Forget to release: "memory leak"
Release twice: heap corruption.
Valgrind and GCC options

Manual memory management is the source of much efficiency and many tears.

Software tools exist that check for memory leaks, heap corruption and other common errors. One is "valgrind". Maybe in CS2263 (?)

Within the last few years, the GNU C compiler suite has added options that let you check for similar corruption and also use of uninitialized values.

-Wuninitialized -Wmaybe-uninitialized
-fsanitize variations such as -fsanitize=address
or -fsanitize=undefined
Valgrind
--------
Valgrind basically runs a program using a virtual machine that provides a number of tools.

You can use it for several things, but our interest is in its ability to check for illegal memory accesses, memory leaks, etc.


Pro: can run on any unmodified executable

Con: useful messages require having compiled with -g and -00 or -O1
    pgm runs dozens of times slower under valgrind

Example use: valgrind --leak-check=yes a.out arg1 . instead of    
a.out arg1
Sanitizers
---------
Found in recent GCC and Clang. Extra instrumentation code is inserted by the compiler to check for various illegal or undesirable things. Requires that you compile the program specially, but the sanitizing code is often fast enough that you could consider leaving it in place in production.

There are different sanitizers: address, thread etc. Newer version of the compiler will have more.
Address Sanitizer

On GCC, add on command line: -fsanitize=address

There are a bunch of further settings that you can control. Set Linux environment variable ASAN_OPTIONS to a colon-separated list.

```
export ASAN_OPTIONS="help=1:replace_str=true"
gcc -fsanitize=address ........
```

help=1 means the full list of settings will be shown when you run the program.

option replace_str=true means that string arguments must be null terminated.
(no need to specify: "true" is the default for it)
See https://github.com/google/sanitizers/wiki/AddressSanitizerFlags
https://github.com/google/sanitizers/wiki/SanitizerCommonFlags
for a list of some options/defaults.
Leak Sanitizer

The leak sanitizer is part of the address sanitizer but can be run separately (lower overhead) if you really want.

Uses LSAN_OPTIONS
Undefined Behaviour Sanitizer
--------------------------------

Several behaviours are undefined in C. Portable code should avoid them.

In GCC, -fsanitize=undefined can catch many of them

(You can control which specific undefined things are checked for, if you really want to.)
Alloca
---------

The alloca(s) function is like malloc() except that it generates space in the stack frame.

Can be more efficient, but its use is discouraged.

Space created with alloca is automatically recovered when the function returns.

DO NOT CALL FREE on space gotten with alloca
In Java, you have a reference to objects. To get an actual object, you need "new".

So assignment     jRef1 = jRef2 means that there is 1 object, and 2 references to it.
                        jRef1.field = 5   changes jRef2.field.

In C, struct assignment is more like basic type assignment. It is "by value". Declaring the struct variable makes "the object".

point_t point1, point2;  \(\text{--- 2 different structs} \)
point1 = point2;  \(\text{--- still 2 different structs} \)
point1.xcoord = 4.2;  \(\text{--- no change to point2.xcoord} \)
Structs and Functions

point_t foo( point_t pt) { .... }
point2 = foo(point1);

Passing point1 as parameter means that a bitwise assignment (copy) of point1 is put into the activation record of foo. Subsequent changes made to parameter pt have no effect on point1.

When foo returns, a full assignment is done into point2.

C structs as parameters and returns: handled more the way that Java handles basic types (not how Java handles object types).
Incomplete/Undefined Struct types
----------------------------------

You are allowed to declare the existence of a struct BEFORE the compiler has seen the actual definition of the struct. Useful with parameters if they are pointers to structs.

```c
struct s z;

int foo(struct s *zz) {} // legal!

int bar(struct s zzz) {} // illegal

int zs() { z.a = 1; } // move down to make legal

struct s {
    int a, b;
};
```
Unions
-------

In a struct, the members are laid out in memory, one after another (with possible padding).

A union is basically a struct where all the members overlap in their storage space.

It’s a low-level horror missing from Java.

Approved use: you want to store an int or a double, but not both at the same time.

```c
struct { /* union inside a struct */
    int storing_an_int; /* boolean */
    union { int ival; double dval;} u;
} s;

s.storing_an_int = 0;
s.u.dval = 3.1415;
```
Type Punning with Unions

A union provides a way to refer to a glob of bits as if several different types.
Wanna peek at the bits of a float?

union {float f; int i;} u;
u.f = 3.141592;
printf("The exponent field is %d\n",
    (u.i & 0x7f800000)>>23);

We are relying on the compiler laying out f and i both starting at the same address (not padding one and not the other). This may not be technically safe with C89. But it is apparently safe in C99. (cf Wikipedia page on "type punning")
Type Punning with Pointers
----------------------------------

Can do something similar without a union:

Take a pointer to one type. Typecast it to another pointer type. Dereference the result.

Strictly speaking, the result might be "undefined", but most compilers have historically done what you want.

float f = 3.141592;
float *pf = &f;
int *pi = (int *) pf;
int i = *pi;

(If ints have more stringent alignment rules on a platform than floats, you could get in trouble here.)
Bit fields

struct {
    unsigned int opcode : 6,
    unsigned int register1: 4,
    unsigned int register2: 4,
    unsigned int constant1: 12
} my_packed_structure;

This declares opcode to be a 6-bit field that is packed with two 4-bit fields and a 12-bit field;

my_packed_structure.opcode = 0b110100;
my_packed_structure.register1 = 9;

Whether opcode is the most- or least- significant 6 bits is system dependent, as are most aspects of bit fields. Good for very low-level un-portable code.
Additional Features for Low-Level Programming

While maybe not part of the C language definition, major C compilers (GCC, Intel's, CLANG, Visual C) tend to have similar support for additional features for low-level programming.

* Intrinsic functions
* Inline assembly

Intrinsics are slightly higher level and I’d consider them more "modern".

Intrinsics and inline assembly can be contrasted with just writing some subroutines in assembly, and then linking the resulting object code with object code resulting from compiling C source.
Intrinsics
---------
(cf Wikipedia: Intrinsic_function)

An intrinsic function is syntactically a library function. However, the compiler will have some baked-in magic when it sees this "function" used, and it will usually compile the intrinsic to a straight-line code sequence that makes use of some particular special CPU instruction.

Intrinsics make available special CPU capabilities while still appearing to be library function calls.
Portable and Non-Portable Intrinsics

You can’t use an Intel x86 intrinsic when compiling for ARM, and vice versa, unless both platforms have the same capability. But some builtins are portable.

Here’s one that might be pretty portable:

```c
int __builtin_parity (unsigned int x);
```

Here’s one that generates a particular fancy AVX2 instruction for AVX2. So only recent AMD/Intel platforms:

```c
v32qi __builtin_ia32_pmaddubsw256 (v32qi,v32qi)
```
Embedded/inline assembly

For details: https://locklessinc.com/articles/gcc_asm/

In case you feel the need to break into assembly language in the midst of some C, you can use "inline assembly". That way, you can use some super-duper new machine instruction that does not yet have an intrinsic function.

```c
for (i = 0; i < 10; ++i) {
    asm(
        "mov %rdx, %rdx\n\t"
        "mov %rdx, %rdx\n\t"
        "mov %rdx, %rdx"
    );
}
```

Note: This old-fashioned form of asm has an argument that is a single string. Register names are what you expect.
Compilers and inline assembly

Early compilers supporting inline assembly would throw up their hands when encountering a glob of inline assembly. They would assume it could do anything. Any value in a register or memory location could possibly be trashed.

So they did very conservative and inefficient things to compensate (eg, save all registers before, restore all after).

Nowadays, you need to declare information about what are inputs to, output from the glob, and what gets clobbered by the glob.

And then maybe the compiler’s optimizer will feel free to rearrange/discard code in your glob, just like it feels free to rearrange code that its earlier phases generated.

Full details are too hairy for this course.
int var14;
int func14(int p1)
{
    asm ("add %1, %0"
         :"+r" (p1) : "rm" (var14) : "cc");
    return foo(p1);
}

declared to clobber cc (condition flags). Compiler is free to choose which register will be %0, which will be initialized to p1. And p1 will be updated by the instruction. (The "+" means it is both an input and an output.) %1 describes the input: it can be a compiler-chosen register or a direct memory access, at GCC's discretion.

Note: argument to asm is 4 sections separated by : This is new-style asm.
What GCC does
-------------------

int var14;
int func14(int p1){
    asm ("add %1, %0"
        ":"+r" (p1) : "rm" (var14) : "cc");
    return foo(p1);}

GCC chooses which registers will be %0 and %1, say %0 = RAX and %1 = RBX, then generates the following func14:
    ...code for preamble of fun....
    ...move whatever stores p1 into RAX
    ...move from var14 in memory into RBX
    add %rax, %rbx
    ... move from RAX into whatever stores p1
    ...code for foo(p1)..
    ret

If %0 is chosen to be the register used for parameters and return values, can reduce moves.
Another x64 example: read the timestamp counter

RDTSC instruction: Read 64-bit timestamp ctr.
Dates to 32-bit ISA, where 64 bit quantities were handled by a pair of 32-bit registers.

Code fragment forms a real 64-bit value, using the RAX register (the "a" in ":=ra") to compute. (Compiler is constrained: %0 is RAX, EAX or AX.)

This code clobbers RDX (and probably CC too??) Example from GCC docs.

```c
uint64_t starttime;
assembler volatile ( 
    "rdtsc\n\t" // Returns time in EDX:EAX.
    "shl $32, %rdx\n\t" // Shift upper bits left.
    "or %rdx, %0" // 'Or' in the lower bits.
    ":=a" (starttime) : : "rdx");

printf("starttime: %" PRIx64 "\n", starttime);

Note "volatile" - compiler must not cache results.
```
C Compiler’s Symbol Table
------------------------

Just like an assembler maintains a symbol table, so does a C compiler.
Other C99 Features
_________________

* restrict qualifier: "while this pointer exists, it is the _only_ way to reach the things that it points at". Helps compiler. (next slide)

* inline keyword. Functions declared inline as a hint to the compiler that it can insert their bodies in place of their calls. (Compilers can inline other functions anyway and can ignore inlining requests...)

* long long int (at least 64 bits long) %lli

* complex numbers (in various precisions)

* bool

* variable sized arrays
Restrict qualifier

---

```c
void f1(int n, float * restrict a1,
       const float * restrict a2){
    for ( int i = 0; i < n; i++ )
        a1[i] += a2[i];
}
```

Compiler would like to unroll this loop, (assuming n is even)

```c
for (int i = 0; i < n; i += 2) {
    a1[i+1] += a2[i+1];
    a1[i] += a2[i];
}
```

No restrict: suppose a1 = a2+1.
Unrolled answer different for a2[2].

See https://www.oracle.com/technetwork/server-storage/solaris10/cc-restrict-139391.html
Other C11 Features
-------------------

* alignment control (aligned_alloc() etc)
  
  int *p = aligned_alloc(512, 20*sizeof *p);

  the address will be a multiple of 512, if the
  platform supports 512-byte alignment. 0 otherwise.

  POSIX-based platforms support sizeof(void *) << k
  for positive integer k.

* support for multithreaded programming (wait for OS course)

* options for bounds checking on arrays
  (not widely supported)