C Introduction
Comparison w/ Java, Memory Model, and Pointers

https://xkcd.com/138/
The C Language

- Systems language originally developed for Unix
- Imperative, compiled language with static typing
- “High level” at the time; now considered low-level
- Allows “direct” access to memory (subject to architecture)
- Many compilers and standards: we’ll use GNU and C99

Ken Thompson  
(inventor of B language and Unix)

Dennis Ritchie  
(inventor of C language and coauthor of C book)

Brian Kernighan  
(coauthor of C book and contributor to Unix/C)
Review: Compilation

![Compilation diagram](image)

**Figure 1.3** The compilation system.

```
linux> gcc -o hello hello.c
```
The compilation process is usually streamlined using a build system (we'll use Make)

Provide a “Makefile” that contains targets, dependencies, and build commands

Example Makefile:

```
hello: hello.c
    gcc -g -O0 -o hello hello.c
```
Hello, World

• How is this different from Java?

```c
#include <stdio.h>

int main()
{
    printf("Hello, world!\n");
    return 0;
}
```
Similarities to Java

• Semicolons!
• Comments (both // and /* */ styles)
• Basic types: int, char, float, double
  – Char is just a number
• Blocks w/ curly braces
• Loops: do, while, for
• Switch statements
  – Parameter must be integer
• Function definitions
Differences from Java

- Preprocessor macros (`#include`, `#define`)
- Functions must be declared before use
  - New distinction: `declaration` vs. `definition`
  - Interface (.h) vs implementation (.c)
- Fewer built-in types
  - Booleans are "bool" (not built-in; must include `stdbool.h`)
    - Actually integers: 0 is “false”, anything else is “true”
  - No built-in string type (C strings are just arrays of chars)
- No classes, packages, or built-in exceptions
- Different I/O functions: `printf`, `fgets`, `scanf` (in `stdio.h`)
  - For `printf`, embed variables in output using formatting codes
  - E.g., use "%d" to embed an integer (see documentation for more codes)
Variables in C

• Declared by **type** and **name** like in Java
  - Can be initialized when declared (this is recommended!)
  - E.g., `int file_counter = 0;`
  - If not initialized, contents are **undefined** until assigned
  - Can be declared `const`
    • Read-only, similar to `final` in Java—must be initialized!

• Multiple declarations per line are allowed
  - E.g., `int x, y;`
  - E.g., `int x = 0, y = 1;`
  - Mixed-initialization and multiple declarations is not recommended
    • E.g., `int x, y = 1; // only initializes y!`
C data types

• Integer types: `char` and `int`
  – Can be signed (default) or `unsigned`
  – `short`, `long`, and `long long` modifiers for `int`

• Real types: `float` and `double`
  – Floating-point representation

<table>
<thead>
<tr>
<th>Data type</th>
<th>Size on stu (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>char / bool</code></td>
<td>1</td>
</tr>
<tr>
<td><code>short int</code></td>
<td>2</td>
</tr>
<tr>
<td><code>int</code></td>
<td>4</td>
</tr>
<tr>
<td><code>long int / long long int</code></td>
<td>8</td>
</tr>
<tr>
<td><code>float</code></td>
<td>4</td>
</tr>
<tr>
<td><code>double</code></td>
<td>8</td>
</tr>
</tbody>
</table>

1 byte = 8 bits
Explicit-width integer types

• C standard doesn't mandate integer widths
  – It only specifies a minimum
  – This causes problems when different architectures or compilers provide different actual sizes

• More portable alternative: `stdint.h` types
  – Basic format: `XintY_t`
  – `X` can be empty (signed) or 'u' (unsigned)
  – `Y` can be 8, 16, 32, or 64 (bits)
  – Examples: `int8_t`, `uint8_t`, `int32_t`, `uint64_t`
Variable attributes (CS 430 preview)

- Name
- Value
- Type
- Address
- Scope
- Lifetime
Variable attributes  (CS 430 preview)

- **Name**: identifier used to refer to the variable in code
- **Value**: current **contents** of a variable
- **Type**: range of values a variable can store
- **Address**: location of variable’s value
  - Most common locations: register, stack, heap, or static data
  - We’ll focus on the non-register locations for now
- **Scope**: code range where a variable is visible
  - Global: visible anywhere in file (code module)
  - Local: visible only inside a function or block
- **Lifetime**: time period when variable access is valid
  - Static: allocated when program starts; de-allocated on exit
  - Dynamic: allocated and de-allocated while program runs
Memory management

- The fundamental difference between C and Java is how they handle memory
  - Java is a managed language, where the compiler and runtime handle memory management for the programmer and direct access to memory is difficult or impossible
  - C is not a managed language, meaning we can directly access and manipulate memory using arbitrary addresses
  - This makes it possible to do the kind of low-level experimentation we want to do in CS 261, and it also enables optimizations that are not possible using Java
  - However, it is also far more dangerous!

“With great power comes great responsibility.”
Pointers

- A **pointer** is a variable that contains a memory address
- Type modifier: “*” indicates one level of pointer
  - int *p;
  - int **p;    // yes, this works
- Often initialized using the “&” operator (“address of”)
  - int x;
  - p = &x;
- Dereferenced with “*” operator (“follow the pointer”)
  - *p = 7;
- Set a pointer to **NULL** to mark them as invalid
- C does NOT check pointers before dereferencing them!
  - int *p = NULL; *p = 123;   // this will segfault!
• Pointers are variables, so they have a type
  – The type describes what kind of data it points to
  – An int has type \texttt{int}
  – A \textit{pointer to an int} has type \texttt{int*}
  – A pointer to a pointer to an int has type \texttt{int**}

• Expressions also have a type
  – If \texttt{x} has type \texttt{int}, then \texttt{x+4} also has type \texttt{int}
  – If \texttt{x} has type \texttt{int}, then \texttt{&x} has type \texttt{int*}
  – If \texttt{p} has type \texttt{int*}, then \texttt{*p} has type \texttt{int}
  – If \texttt{p} has type \texttt{int*}, then \texttt{&p} has type \texttt{int**}
int a = 42;
int b = 7;
int c = 999;
int *t = &a;
int *u = NULL;
printf("%d %d\n", a, *t);

c = b;
u = t;
printf("%d %d\n", c, *u);

a = 8;
b = 9;
printf("%d %d %d %d\n", b, c, *t, *u);

*t = 123;
printf("%d %d %d %d %d\n", a, b, c, *t, *u);
What does the following C code do?

```c
int* c, d;
```

- A) Declares two integers “c” and “d”
- B) Declares two integer pointers “c” and “d”
- C) Declares one integer pointer “c” and one integer “d”
- D) Declares one integer “c” and one integer pointer “d”
- E) The behavior is undefined
The following code doesn't do what you think it does:

- `int* c, d;`

Recommendation: put asterisk next to variable names in declarations

- `int *c, *d;`
- Or declare only one variable per line!

Exception: function declarations (since there can only be one return value)

- `int* myfunc();`
Every process has its own virtual private memory called an *address space*.

The address space is divided into *regions*. Some regions are *static* and do not change size while the process runs, while others are *dynamic*, changing size if necessary.

The *stack* region expands when a function is called and shrinks when a function returns. The *heap* region expands when `malloc()` is called.

Some regions begin at a *randomized* location (different on every run) for security reasons.
Dynamic memory allocation

- If you do not know how much memory you need until after the program is running, you must allocate memory on the heap
- Allocate with `malloc()` function (or `calloc`)
  - Pass it the number of bytes you need
  - Often calculated using the `sizeof` operator
  - Returns a pointer to the beginning of the allocated region
- De-allocate with `free()` when you are done
  - Pass it a pointer to the beginning of the region you want to free
  - Good code practice: set pointer to `NULL` afterwards
  - Neglecting to free memory will result in a memory leak
Variable summary

- **Global variables**
  - Static data address, global scope, static lifetime
- **Local variables (regular)**
  - Stack address, local scope, dynamic lifetime
  - Valid while the function executes
- **Local variables declared ‘static’**
  - Static data address, local scope, static lifetime
  - Similar to global variable but with local scope
- **Dynamically-allocated memory**
  - Heap address, local scope (via pointer), dynamic lifetime
  - Valid from malloc until free
  - Pointer(s) themselves are usually local variables (see above)
int global_var;

void foo()
{
    static int foo_st_var;
    int foo_var;
}

int main()
{
    int main_var;
    int *malloc_var = (int*)malloc(sizeof(int));
    foo();
    return 0;
}

For each of the following variables, classify their address as *static*, *stack*, or *heap*:

- global_var
- foo_st_var
- foo_var
- main_var
- malloc_var
- *malloc_var

Does this program leak memory? If so, where, and how would you fix it?