C

What's different about C? (vs. Java)

- It's older
  - Procedural, not object-oriented
  - Explicit, low-level memory model
    - Requires manual memory allocation and de-allocation
  - Unsafe basic data structures
    - E.g., no array bounds checking
  - Requires explicit interface (header) files
  - Fewer standardized libraries

What's good about C?

- C is appropriate when the extra control over data & performance trade-offs is required
  - Embedded software
  - Low-level systems programs
  - Run-time systems of higher-level languages
- Inappropriate when a higher-level language would be fine

Why learn C?

- Complement knowledge of higher-level languages e.g. Java & csh
  - Understand trade-offs between different styles of languages
- Lots of existing software written in C or C++, some of it appropriately
  - And lots of future software
- Impact on society from security problems caused by poor C code J

What about C++?

- C++ is (almost) a superset of C
  - Adds object-oriented features, like classes
    - Similar to but more powerful & complicated than Java's classes
  - Adds templates
    - Similar to but more powerful & complicated than Java 1.5's generic types
  - Adds some nicer syntax for some things
- We'll focus on the C subset of C++

A trivial C(++) program

```
#include <stdio.h>

int main(int argc, char** argv) {
  if (argc > 0) {
     fprintf(stderr, "bad args\n");
     return –1;
  }
  printf("hello, class!\n");
  return 0;
}
```

Some comparisons to Java

- Similar statements & expressions as Java (e.g. `if`, function calls, return)
- Similar data types to primitive ones in Java (e.g. `int`, `char`)
  - But has pointer data types too (e.g. `char**`)
- C is procedural, not OO
  - Functions are declared at top-level
  - Variables can be declared at top-level too
    - "Global variables"; they're bad style
- Libraries "imported" using `#include`

Program entry point

- A C program starts with the *unique* procedure named `main`
- Optionally takes a length and an "array of strings" of that length which are the command line arguments
  - "Array of strings" = `char**`; ugh
- Returns the program's exit code
  - 0 = success, non-zero = failure

Simple text output

- Java:

```
System.out.print("hi ");
System.out.println("there");
```

- C:

```
#include <stdio.h>
…
printf("hi ");
printf("there\n");
```


Tools

- `gcc –c` *file*`.c`
  - Compile C source *file*`.c` into object *file*`.o`
  - C++ source uses `.C`, `.cpp`, `.cc`, or `.c++`
- `gcc -o` *program file*`.o` …
  - Link one or more object *file*`.o` into executable *program*
- `gdb` *program*
  - Debug *program*
    - Compile with `-g` option for source-level debugging
  - Run `gdb` under `emacs`!

C memory model

- C exposes the memory resources of the underlying machine
  - **Static**, **stack**, and **heap** memory, composed of bits, bytes, and words
  - Allows programmers to control where their data values are stored and how much space they consume
- Different memory regions have different costs for use, different requirements for correct use
  - Programmers can make explicit cost trade-offs
  - C puts correctness burden on programmers

Static (a.k.a. global) memory

- Fixed size
- Allocated when program starts
- Deallocated when program ends

- Top-level (global) variables stored here
  - Akin to Java's static variables

Stack memory

- Variable (total) size
- A fixed-size chunk is allocated whenever a procedure is called
- Deallocated automatically when the procedure returns
- Procedure arguments and local variables stored here, just as in Java

Heap memory

- Variable (total) size
- Allocated on demand, by a `new` expression (or a `malloc(…)` call)
  - Like Java's `new` expression
- Deallocated on demand, by a `delete` statement (or a `free(…)` call)
  - Java does this automatically via garbage collection

What's in memory?

- Each region of memory made up of a sequence of *bits*
  - A bit is a single binary digit, a 0 or a 1
- 8 bits are grouped into a *byte*
  - Standard unit of memory, e.g. megabytes
- Some number of bytes are grouped into a *word*
  - Typically 4 bytes = 1 word (32-bit machines)
  - Sometimes 8 bytes = 1 word (64-bit machines)

C numeric data types

- `char`: 1 byte
- `short`: 2 bytes
- `int`, `long`, `long long`: 4 bytes – 2 words
- `float`: 4 bytes
- `double`: 8 bytes
- No `boolean`; just use `int`

Variable declarations

- Each variable declaration allocates space to hold the variable's value
  - Size of memory allocated determined by type of variable
  - Memory region determined by whether the declaration is of a global or a local variable
- Variable names the allocated memory block
- Allocated memory isn't initialized automatically!
  - Unlike Java
  - Can be unsafe, bug-prone!
- In C (not C++): all var decls at start of scope

Addresses and pointers

- Each byte of memory has an *address*
  - Like an integer index into an array of bytes
- Can store an address in memory
  - A *pointer*
- Can dereference the pointer to read or update the contents of the pointed-to memory
  - Java's object references are pointers

Pointers in C

- C has a new kind of type: a pointer
  - Pointer itself consumes 1 word of memory
  - Also specifies the type of the pointed-to memory
- Can declare variables to be of pointer type
  - [Crappy syntax; don't declare multiple pointer variables with the same declaration!]
- Examples:

```
int* pi;     // a pointer to an int
char* pc;    // a pointer to a char
int** ppi;   // a pointer to a pointer to an int
```


Creating pointer values

- Simple way to make pointers: take the address of a named variable
  - `&`*var*
  - Pointer target type is type of *var*
- Ex:

```
int i = 5;
int* pi = &i;
int** ppi = &pi;
```


Dereferencing pointers

- Given a value of pointer type, can:
  - Read the memory it points to
  - Update (assign to) the memory it points to
  - Collectively called *dereferencing* the pointer
- Use `*` prefix operator to dereference a pointer, on either side of assignment
- Ex.

```
int i = 5;
int* pi = &i;
*pi = *pi + 1;
// now, what's the value of i? of pi?
```


More on dereferencing

- Can use a null pointer in place of a valid pointer
  - Ex: `int* pi = NULL;`
    - (use `NULL` if `#include <stdio.h>`, `0` otherwise)
  - Dereferencing a null pointer is illegal and can do bizarre things (often "segmentation violation")
    - Not as fail-stop as in Java
- What if dereference an uninitialized pointer?

```
int* pi;
…
*pi = *pi + 1;
```


Pointers to heap memory (nicer but C++-specific version)

- Can also create pointers by allocating new heap memory, and getting its address
  - "`new` *type*" (an expression):
    - allocates (but does not initialize!) memory in the heap to hold a value of *type*
    - returns its address (which has type *type*`*`)
- Ex:

```
int* pi2 = new int;
int** ppi2 = new int*;
```


Uglier C version

- Use `malloc` fn call instead of `new`
  - `malloc` takes the number of bytes to allocate (not the type; ugh)
  - `malloc` returns a `char*` (not a `type*`; ugh)
- C++:

```
int* pi = new int;
```

- C:

```
#include <stdlib.h>
…
int* pi = (int*)malloc(sizeof(int));
```

Deallocating heap memory (C++-specific version)

- When done with heap-allocated memory, must explicitly *deallocate* it
  - "`delete` *expr*" (a statement):
    - evaluates *expr*, which should yield a pointer to heap memory
    - deallocates the memory pointed to (not the pointer!), making it available for reuse for future heap allocations
- Ex:

```
int** ppi2 = new int*;
…
delete ppi2;
```


C version

- Use `free` function call instead of `delete` statement
- C++:

```
int** ppi2 = new int*;
…
delete ppi2;
```

- C:

```
#include <stdlib.h>
…
int** ppi = (int**)malloc(sizeof(int*));
…
free(ppi);
```


Some possible deallocation errors

- Static type checking ensures `delete` only applied to a pointer
- What if try to deallocate non-heap memory?
- What if forget to deallocate heap memory?
  - A *storage leak*

Lifetime of pointers

- Pointers may not be valid indefinitely
  - A pointer becomes invalid when the memory it points to is deallocated
    - A *dangling pointer*
  - Dereferencing an invalid pointer can cause undefined bad behavior (crash, data loss, security hole, …)
- When does a pointer to a global variable become invalid? To a local variable? To heap-allocated memory?

Java & pointer lifetime errors

- Java's references to objects are all pointers
- But Java doesn't allow the program to ever reference an invalid pointer
  - Cannot create pointers to locals
  - Cannot explicitly deallocate memory
- Java also ensures no storage leaks