# The C Programming Language

CIS-4020
Vermont State University
Peter Chapin

# Design Goals

- C was designed as a systems language.
  - Low level control of machine resources.
    - Direct access to memory.
    - Manipulation of raw bytes in a type-unsafe manner.
    - Bit manipulation.
    - Low overhead.
      - "If you don't write it, it doesn't happen."
  - Designed to replace assembly language when writing operating systems.
    - ... or device drivers, or other "systems" application (virtual machines, memory managers, etc).

# Applications?

- C is not really an applications language.
  - Too low level.
    - Does not provide many (any) convenience services.
  - Too unsafe.
    - Easy to modify out of bounds memory unintentionally.
      - Creates security problems (buffer overflow bugs).
      - A large number of security vulnerabilities in software are a direct result of C's lack of safety.
    - Easy to treat objects of one type as another type.
      - Useful for certain specialized situations.
      - Generally an error in normal applications.

# Review?

- These slides are intended to illustrate the features of C we need for operating systems.
  - I do not bore you with details of `if` and `while`!
  - Topics:
    - Pointer arithmetic and arrays. Pointers to void.
    - Pointers to functions.
    - Typedef.
    - Bit manipulation.
    - Structure layout.
    - Unions.
    - Macros and conditional compilation.

# Pointer Arithmetic

- Consider:
  - ```
    int  array[1024];
    int *p = array;

    ++p;          // Points at next element.
    p[ 0] = 1; // Really *(p + 0) = 1;
    p[-1] = 1; // Really *(p - 1) = 1;
    if (p - array > 1) { ... }
    ```
    ```

# Exotic Pointer Arthimetic

- Consider:
  - ```
    int array[1024];
    char *p = (char *)array;

    ++p;          // Points at next byte.
    *p = 1;       // Modifies one byte.
    *(int *)p = 1;  // Modifies an int.
    ```
  - The last assignment causes a value to be placed into the array that overlaps two array elements.
    - Might fail on some systems due to alignment problems.
    ```

# Pointers to void

- General pointer that can point at anything.
  - Used to hold pointers of any type.
  - Requires a cast before it can be used.
  - ```
    struct example object;
    void *thing = &object;
    ...
    struct example *p =
        (struct example *)thing
    p->member = 1;
    ```
    ```

# Uses of `void *`

- Kernel uses void * to give third parties a way of storing custom data in kernel data structures.
  - ```
    struct kernel_internal {
      ...
      void *private;
    };
    ```
  - Kernel passes a pointer to `kernel_internal` to a module.
    - Module can allocate custom data structure of any type and store its address in `private` member.
    - Module can later access that member to get back the custom data.
    ```

# Pointers to Functions

- Functions have addresses as well.
  - ```
    int (*pf)(int, char *);
    int function(int x, char *p);
    pf = function;
    pf(1, "Hello");
    ```
  - The variable `pf` can be made to point at any function with the right type signature.
  - The name of a function without an argument list is a pointer to that function.
  - Dereferencing a pointer to function is implicit.
    ```

# Function Pointers in the Kernel

- The kernel uses pointers to functions widely.
  - ```
    struct operations {
      int (*read)(void *buffer, int n);
      int (*write)(void *buffer, int n);
    };
    ...
    struct kernel_internal {
      struct operations *ops;
    };
    ...
    struct kernel_internal *p;
    ...
    p->ops->read(buffer, 1024);
    ```
    ```

# Typedef

- Introduce an *alias* for an existing type.
  - `typedef int counter_t;`
    `counter_t n = 0;`
  - The `counter_t` type is just a new name for `int`.
    - Can be mixed with `int` freely.
    - The `_t` part of the name is just a convention.
- Used for two purposes.
  - Give a simple name to a complex type.
  - Centralize a type definition to a single place (in a header file).

# Typedef in the Kernel

- The kernel uses many typedef names.
  - Some kernel specific
  - Some shared with applications.
- Examples
  - `pid_t`
    - Type for representing process ID numbers
  - `uid_t`
    - Type for representing user ID numbers
  - `loff_t`
    - Type for representing offsets in potentially large files ("long offset type")

# Bit Manipulation

- C has many bit manipulation operators.
  - `x & y` (bitwise AND)
  - `x | y` (bitwise OR)
  - `x ^ y` (bitwise XOR)
  - `~x` (bitwise complement)
  - `x << y` (bitwise left shift)
  - `x >> y` (bitwise right shift)
- Very fast
  - Typically compile to single machine instructions.

# Flags and Masking

- Common use of bitwise operators:
  - Store independent flag values in a single int.
  - ```
    #define RED   0x00000001
    #define GREEN 0x00000002
    #define BLUE  0x00000004

    int flags = RED|BLUE;
    ...
    if (flags & GREEN) { ... }
    ...
    flags ^= RED;
    ```
    ```

# Structure Layout

- Consider:
  - ```
    struct example {
      char  x;
      int   y;
      char *z;
    };
    ```
  - C standard requires:
    - First member be at offset zero (`&example_object` can be cast to a pointer to `char` and used to access `x`).
    - Members layed out in order of declaration (offset of `y` is greater than offset of `x`, etc).
    - However, the compiler is allowed to include padding.
    ```

# Unions

- Similar to a structure.
  - ```
    union example {
      float value;
      char raw[4];
    };
    ```
  - Members *overlap* in memory. Only one value can be stored at a time.
    - ```
      example_object.value = 3.14F;
      example_object.raw[1] ^= 0x08;
      ```
    - Toggles one bit of the floating point representation.
    - Also used to save memory.

# Preprocessor

- Lines begining with # are preprocessor directives.
  - Technically they are handled *before* the compiler processes the source file.
    - Many compilers process the preprocessed source right behind the preprocessor (so only a single pass is needed).
  - `#include`, `#define`, `#if`, etc.
    - Treat your program as a text file.
      - Technically the preprocessor knows (next to) nothing about C.
      - C preprocessor sometimes used for other purposes.

# Object-Like Macros

- Preprocessor symbols that are simple names.
  - `#define MAX_BUFFER_SIZE 1024`
    - Give a name to a raw number.
      - Better documentation; easier to read and understand.
      - Easier to change.
  - `#define LOOP while (1)`
    - Hide arbitrary text inside the macro.
    - 
```
LOOP {
  x = f(y); // Or whatever...
}
```

# Function-Like Macros

- Preprocessor symbols that look like functions.
  - `#define max(x, y) \`
    `((x > y) ? (x) : (y))`
    - Inline expanded (low overhead).
    - Can expand to code fragments (that by themselves would not compile).
    - Tricky...
      - `biggest = max(a++, b);`
      - `biggest = ((a++ > b) ? (a++) : (b))`
        - Oops! Might increment `a` twice. Probably not intended.

# Conditional Compilation

- Compiler selectively skips material depending on other preprocessor symbols.
  - ```
    #define DEBUG
    ...
    #ifdef DEBUG
      printk("Debugging output...\n");
    #endif
    ```
  - ```
    #define CONFIG_SMP
    ...
    #ifdef CONFIG_SMP
      // Do SMP special stuff here.
    #endif
    ```

# Kernel Configuration

- Configuration Tool...
  - Creates header with many `#define` values like `CONFIG_SMP`, etc.
  - Kernel code uses `#if` / `#endif` directives to selectively compile different code depend on configuration.
    - C source really many programs in one
    - A different program for each combination of configuration settings.
      - Suppose there are 50 CONFIG macros... $2^{50}$ different kernel configurations!
      - Do you think they are all tested?