Advanced C for the 1337 kernel h@x0rs
Albert Wong (awong at cs dot washington dot edu) CSE 451 - wi03

What is C
C is a procedural language. This means there is no language support for writing OO code. However, you can still write OO code. You just have to do it manually. This is similar to Java having language features for synchronizing threads and C/C++ not. You can still do multithreaded code in C/C++.

Major syntactic differences in C
- There are no classes. Structs are NOT the same classes as they are in C++.
- You can only declare variables at the top of a block (after an opening brace) before any other kind of statement (except perhaps typedefs).
- There is no new/delete operators. Only malloc and free functions.
- You use void*s a lot in C data structures.

Major syntactic differences in C

The C Preprocessor

- **Introduction**
The preprocessor deals strictly in text. Here is a list of the standard preprocessor directives and macros excluding #line.

- `#include <filename>`, `#include "filename"` – expands into contents of the given file into current position. The <> means to search the standard include path for the file while the "" means to search the current directory.
- `#error message`, `#warning message` – Causes the compiler to either halt or issue a warning if this line is reached. Useful for debugging.
- `#pragma` – Passes options to the compiler. Options change from compiler to compiler.
- `#if condition`, `#elif condition`, `#endif` – Includes or excludes a block of text dependent on the value of the condition. #if 0 is useful for removing a block of code from compilation.
- `#FILE`, `#LINE`, `#DATE`, `#func` – these macros expand into strings representing the current file, line, date, and in c99, the current function.

The C Preprocessor

- **#define basics**

  - **#define macros**

    - `#define SOME_LABEL Some list of literals`
    - `#define MIN(x,y) ((x) < (y) ? (x) : (y))`
    - `#define printf(x,…) fprintf (stdout, x, __VA_ARGS__)`

    - Macros can be used for quick and dirty constants. Though it is often preferable nowadays to do:
      ```
      const T name = value;
      ```
      where T is a type. This is because this creates a variable with type info.
    - Macros can be used to like functions. Think of them as a patterned search and replace. Some simple functions are often implemented as just a #define macro. Common examples are “min” and “printf.” Many libraries implement them in a fashion similar to the examples above.
    - You can even do variable argument macros by putting an ellipse (“…”) in the parameter list. The tag __VA_ARGS__ expands to all the extra arguments with the comma. (You may notice a problem with our definition of “printf” given our explanation of __VA_ARGS__. Most compilers extend the behavior of __VA_ARGS__ expansion to make up for this problem.)

- **#define fun!**

  - **#define macros** string manipulation operators

    - `#define concat (x,y)  x##y`
    - `#define mkstr(x)  #x`

      - ## performs a concatenation of the two preprocessor arguments. This may be useful for autogenerating mangled names or some other sort of textual manipulation. Thus:
        ```
        concat(wordA,wordB)
        ```
        is equivalent to:
        ```
        wordAwordB
        ```
      - # makes the following macro argument a string (with quotes). It also chomps whitespace so everything is only 1 space. Thus:
        ```
        mkstr(bu ha ha ha me lo lo weeeeeeee)
        ```
        becomes
        ```
        "bu ha ha ha me lo lo weeeeeeee"
        ```

- Basic C type mechanisms
  - **typedefs**

    - `typedef unsigned char byte;`
    - `typedef struct Name { int id; } Name;`
    - `typedef int (*Comparator)(void*, void*);`

      - Typedefs are a way of creating aliases for a type. The example above makes bytes have the same meaning to the compiler as an unsigned char.
      - You use typedef for 3 reasons.
        - Making a shorthand alias.
          - This is often done with structs and function pointers.
        - Adding an extra level of abstractions to the type.
          - Say you’re working between using wchar_t and a long int.
        - Designating a logical difference.
          - A byte is the same as an unsigned char, but when you see “byte” you think of 8 bits where when you see “char” you think of a or something similar.
Basic C type mechanisms – structs

Structs

- They are not classes.

- There is no language implemented support for inheritance or methods.

- However, with some discipline, one can simulate the functionality pretty well for the most part.

- There are no protection facilities (everything is public)

- You use structs for 2 reasons
  - Ordering memory
    Because structs guarantee a memory layout, they are useful for communicating with hardware.
  - Grouping related items
    This is more common usage, though it is kind of a side effect of the ordering behavior. You can use this to create really dumb “objects.”

Basic C type mechanisms – union

Union

- Unioned types

- Gives one location in memory, multiple type interpretations.

- This is probably one of the more useless types…unless you’re implementing some sort of polymorphism or talking to hardware.

- You use unions for 3 reasons
  - You want to save memory and you need to at any given time represent one of a number of types.
  - You can use a union to declare a variable that represents those types.
  - You need a location in memory to have more than 1 semantic meaning.
    - This may happen if you are talking to hardware (a memory mapped register may have more than 1 type it represents). Or it may happen in parameter passing or some other esoteric situations
  - You want to screw with someone’s head.
    ‘nuff said.

Basic C type mechanisms – enums

Enums

- enumerated types

- Creates a type with a limited set of label values.

- Creates a mapping from a label to a unique integer.

- You use enums for 2 reasons
  - Making an “option” type
    You can restrict the values assigned to the enum, so this is a natural usage.
  - Integral constants (kind of a misuse)
    Since enum values are in effect integers, they can be used as constants. This is kind of a hack, but it is common. You can assign specific numbers to each enum value if you want.

Basic C type mechanisms – pointers

Pointers

- memory locations

- T *name = NULL;

- Pointers are variables that hold a number representing a location in memory.

- Pointer arithmetic increments by units of type, not by address location. short n; is 2 bytes. So short *pn = (short*)10; pn++; will yield pn == (short*)12.

- Pointers are the size of the natural machine word. That means they are the same size as an int or a size_t.

- Pointers to functions do not need to explicit dereference syntax to use it. The compiler will do it implicitly for you.

- Use pointers to const types if you wish to pass without requiring a copy.

Advanced C types – arrays

Arrays

- homogeneous block of 1 type

- ARRAYS ARE NOT POINTERS

- Arrays are not values. You cannot say “name = & var;” or anything similar.

- The array name is the location in memory. It is not a variable holding an address of a location in memory. There is no space allocated for holding the address, it is resolved at compile time.

- Arrays have dimension. You can declare pointers to arrays of a specific dimension: char (*name)[3];. This is a pointer to an array of 3 chars.

Advanced C types – void*

Void Pointers

- void *ptr;

- Void pointers are refer to a generic untyped location in memory.

- That means they have no type.

- You must cast a void * to a typed pointer before using it in C.

- Any pointer will be implicitly promoted to a void* on assignment.

- You cannot perform pointer arithmetic on a void*.

- If you want to do arithmetic on the pointer in terms of memory addresses (rather than in terms of types), cast it to an unsigned char*.

- You use void* for 2 reasons
  - Generic programming
    They are the analogues to java’s Object type. Indeed they are even more general. For this reason, they are actually used more sparingly.
  - Generic memory reference
    Sometimes you really mean “this is just a chunk of memory.” Often times, this I represented as either a void* or a char*. One example of where void* is used is in malloc and free (the analogues to new and delete).
**Advanced C types - function pointers**

Function Pointers

int (*name)(int param, int param);

- Functions are just blocks of code at some location in memory.
- Thus they can be pointed to.
- The type of a function can be determined by its signature (return type and parameter list).
- The syntax for a declaring a function pointer is ugly, but you do not need to dereference the pointer to use it. Thus, after the above example, both name(1,2) and (*name)(1,2) are valid.
- Often, one typedefs function pointers before using it. 
  
Extern

- typedef int Comparator(void*);    
- Examples of function pointer

**Advanced C type modifiers - inline**

Inline

int inline func(void);

- The first rule of inline is, don’t use it.
- The second rule is if you really are going to, make sure you know why you are going to do it.
- Inline hints to the compiler that this function should be unrolled an inline into wherever it is called. This can avoid function call overhead.
- Problems with inline:
  - The compiler may happily ignore this. It is kind of like the register keyword in this manner. It is merely a hint; it is not a command.
  - The compiler often knows better than you. It’ll inline for you anyways if something is small enough and inlining seems smart.
  - You may (and probably will) make your program size larger.
  - You may make your program slower. Read the previous bullet as ‘cache miss’ or ‘page fault’.

Inline is a compiler hint. It really has no place in a high level language, but it is here because C isn’t completely high level.

**Advanced C type modifiers - static, extern**

Static

- static int i;
- static int func(void);

- static functions in header files that are not inline are only done by the braindead.
- static variables in header files are only done by the braindead.
- uninitialized static variables are by defaulted zeroed.
- static local variables have their initializer called only once during a program’s lifetime (somewhere before first usage, usually at program load).
- static variables are by defaulted zeroed.
- static variables in header files are only done by the braindead or the incredibly wise.

- Static has 2 meanings in C (3 in C++).
  - Restrict the visibility of the current identifier to the current translation unit. Essentially, make it private to a file. The equivalent in C++ is an anonymous namespace.
  - Allocate memory for the following variable in the static memory region (not on the stack or the heap).
  - static local variables have their initializer called only once during a program’s lifetime (somewhere before first usage, usually at program load).
  - uninitialzed static variables are by defaulted zeroed.
  - static variables in header files are only done by the braindead.
  - static functions in header files that are not inline are only done by the braindead or the incredibly wise.

**Advanced C type modifiers - extern**

- extern int i;
- extern int func(void);
- extern int func(void);

- extern tells the compiler to not generate storage for the following variable.
- Function prototypes are by default extern. Just like local variables are by default auto.
- All global variables in a header file shall be declared extern on pain of link error.
- All externed variables or functions must be allocated storage somewhere. It must only be allocated storage once!!!
- There is no protection for mismatching the definition of the variable with the declaration in 2 different translation units. That means if you define i to be char i in foo.c and then declare it as extern int i in main.c and then use it, your code will compile and link, but you will have problems.

**Random stuff**

- __attribute__. This is a gcc compiler extension that allows you to specify extra attributes to the compiler for a variable or function such as which memory segment it should go in.
- In C, you must explicitly say void in your function parameter list, otherwise, the compiler thinks that you have a variable argument list (like printf).
- To explicitly declare a variable argument list, you write put an ellipse for your last argument.
- If you do not declare a function before trying to use it, C automatically assumes it has a return type of int and that it takes variable arguments.
- prfunc “%x” etc.) is your friend.
- static variables cannot be affected by stack corruption. Useful for debugging. Don’t leave it in your production code though.