Advanced C Topics

- Overview
- C Standards
- C Standards Continued
- Primitive C Types
- Pointer Types
- Array Types
- Structure Types
- Structure Types Continued
- Union Types
- Union Types Continued
- Double Decode Example Setup
- Double Decode Example
- Double Decode Example Continued
- Double Decode Example Continued
- Double Decode Example Log
- Double Decode Example Log Continued
- Double Decode Example Log Continued
- Function Types
- Function Types Continued
- No String Type
- Arrays
- Pointers and Arrays

- When Arrays are Pointers
- Ragged Arrays
- Matrix Multiplication: Before C99
- C99 Matrix Multiplication
- Using `typedef`'s
- Complex Declarations
- Complex Declarations Continued
- Decoding Complex Declarations
- Example of Decoding a Complex Declaration
- A Practical Example of a Complex Declaration
- Abstract Declarators
- Type Declarations Semantic Restrictions
- Memory Layout Example
- Memory Layout Example Continued
- Function Parameters and Return Values
- Returning Values via Pointer Parameters
- C Preprocessor Overview
- Expression Macros
- Expression Macros Continued
- Statement Macros
- Statement Macros Continued
- Statement Macros Continued
- Token Stringification and Concatenation Example Motivation

- Token Stringification and Concatenation Example
- Token Stringification and Concatenation Example Continued
- Token Stringification and Concatenation Example Continued
- Token Stringification and Concatenation Example Continued
- Token Stringification and Concatenation Example Continued
- Header File Inclusion Idiom
- Idiom for Number of Array Elements
- Idiom for Number of Array Elements Continued
- Asymmetric Bounds
- Asymmetric Bounds Continued
- Asymmetric Bounds Continued
- Asymmetric Bounds Continued
- External Variable Declarations
- External Variable Declarations Continued
- External Variable Declarations Continued
- Static
- Static Continued
- Using Data Instead of Code
- Using Data Instead of Code Continued
- Using Data Instead of Code Continued

- Name Spaces
- Name Spaces Example
- Dynamic Memory Allocation
- Dynamic Memory Allocation Continued
- Dynamic Memory Allocation Continued
- Abstract Data Types in C
- ADT Example
- ADT Stack Implementation
- ADT Implementation Continued
- References
- References Continued

Overview

- C Standards
- C types.
- Pointers and arrays.
- Using and understanding the limitations of `typedef`'s.
- Complex declarations.
- The C preprocessor.
- C idioms.
- Using asymmetric bounds.
- External variable declarations.
- Using data instead of code.
- Name spaces

- Dynamic memory allocation.
- ADTs.

C Standards

1. K&R C. No standard. Described in first edition of Kernighan and Ritchie's *The C Programming Language*, 1978. Several incompatible implementations.

2. ANSI-C, 1989. Major change included addition of function prototypes; minor change included # and ## preprocessor operators. The language described in the 2nd Edition of *The C Programming Language*, aka K&R2.

3. ISO-C. Basically ANSI-C.

4. Amendment 1 to ISO-C, 1994. Additions for internationalization (I18N) and localization (L10N) with additions for multibyte and wide `chars`.

C Standards Continued

1. C99. Approved end of 90's. `//` comments, anywhere declarations, inline functions, macro varargs, `_Bool` constants `true` and `false`, named `struct` initialization, variable length arrays, `struct` hack where last member of `struct` is an incomplete array type standardized, etc. Implementations close to complete with major exception Microsoft, though that may be changing.

2. C11. Approved recently. Implementations in progress; features include removal of `gets()`, multithreading support, alignment specification.

Primitive C Types

Primitive types include:

- Integral types: in order of non-decreasing size `char`, `short`, `int`, `long`, `long long`. Signed by default, can be made unsigned by using modifier `unsigned` (but the signed-ness of plain unmodified `char` is undefined). Examples (initializers can be omitted, but are usually a good idea whenever possible to avoid uninitialized variable errors):

```
unsigned char c1 = 'a', c2 = 'b';
char c = c1 + 1;            //signed-ness not known.
long long length = 1024LL;
unsigned int index = 0;
```

- Floating point types: in order of non-decreasing size `float`, `double`, `long double`.

```
float result = 0.0;
const long double pi2 = 2 * PI;
```

Pointer Types

If *T* is a type, then *T* * is of type pointer to *T*.
Examples:

```
int i;
int *iP = &i;
const int *icP = &i; //value pointed to cannot be changed via pointer
int *const ciP = &i; //pointer cannot be changed.
enum { N = 10 };
double *dP = malloc(N * sizeof(double));
void *genericP = dP; //must be cast to pointer to specific type
                     //before being used.
...
qsort(genericP, N, sizeof(double), doublesCompare);
```

Array Types

If *T* is a type, then *T*[ ] is of type array of *T*. Note that the number of elements in the array is not part of the type. No multi-dimensional arrays but can have array of arrays. Examples:

```
int a[10];    //uninitialized (unless static (top-level) when 0).
double piMults[] = { PI, 2 * PI, 3 * PI };
float multi[5][10]; //array of 5 array's of 10 floats.
char message[] = { 'h', 'e', 'l', 'l', 'o' }; //probably a error
```

Structure Types

If *T1*, *T2*, `...` *Tn* are types, then `struct {` *T1* `t1; ...` *Tn* `tn }` is a structure type (corresponds to **cartesian product** of types *T1* x *T2* x `...` x *Tn*). Examples:

```
typedef struct { //typedef alias for type
  double x;
  double y
} Point2;

//A type which can be used for singly-linked list node
typedef struct NodeStruct {  //NodeStruct is struct tag
  void *value;
  struct NodeStruct *succ;
} Node;

//Positional initialization
Point2 line[] = { { 0, 0 }, { { 1, 1 } };

//can also use c99 designated initializer
Point2 triangle[] = {
  { .x = 0, .y = 0 },
  { .y = 1, .x = 0 },
  { .x = 1, .y = 0 }
};
```

Structure Types Continued

```
//Last element can be an array with size fixed at runtime using
//dynamic memory allocation
typedef struct {
  size_t size;
  void *elements[];  //flexi array
} SizedArray;
enum { SIZE = 22 };
SizedArray *a = malloc(sizeof(SizedArray) + SIZE*sizeof(void *));
a->size = SIZE;

for (int i = 0; i < SIZE; i++) a->elements[i] = i;
a->elements[4] = ...; //ok
a->elements[22] = ...; //unpredictable runtime error
```

Union Types

If *T1*, *T2*, `...` *Tn* are types, then `union {` *T1* `t1; ...` *Tn* `tn }` is a union type (corresponds to **undiscriminated union** of types *T1* x *T2* x `...` x *Tn*). At any time only one of `T1`, `...`, `Tn` can exist in the union; consequently, the sizeof the union is the size of the largest `Ti`.

Often used where inheritance would be used in OO-languages and paired with an `enum` as a **discriminated union**. Example:

```
typedef struct {
  enum { CIRCLE, RECTANGLE, ... } type;
  union {
    struct {
      Point2 origin;
      double radius;
    } circle;
    struct {
      Point2 topLeft;
      double width;
      double height;
    } rectangle;
    ...
 };                              //anonymous union from c11
} Shape;
```

Union Types Continued

```
double sum_areas(Shape shapes[], int numShapes) {
  double sum = 0.0;
  for (int i = 0; i < numShapes; i++) {
    switch (shapes[i].type) {
      case CIRCLE:
        sum += PI * shapes[i].circle.radius *shapes[i].circle.radius;
        break;
      case RECTANGLE:
        sum += shapes[i].rectangle.width * shapes[i].rectangle.height;
        break;
      ...
    }
  }
  return sum;
}
```

Double Decode Example Setup

Another typical situation requiring the use of `union`'s is accessing some value as several different types.

- Recall that a IEEE float is typically laid out (starting from the MSB): 1 sign bit, bits for biased exponent, bits for **normalized** mantissa. Value is sign x 2^(exp - bias) x (1 + (mantissa value)).
- Can alias a `double` to both a `byte` array as well as bit-fields with widths corresponding to the `sign`, `exponent` and `mantissa` field in a `double`.

```
union {
  double value;
  unsigned char bytes[sizeof(double)];
  struct {
    unsigned long mantissa : MANTISSA_BITS;
    unsigned long exp : EXP_BITS;
    int sign : SIGN_BITS;
  };
};
```

Double Decode Example

Following program ./programs/double-decode.c:

```
int
main(int argc, const char *argv[])
{
  enum {
    SIGN_BITS = 1,
    //DBL_MANT_DIG from <float.h> gives # of unnormalized mantissa bits
    MANTISSA_BITS = DBL_MANT_DIG - 1,
    EXP_BITS = sizeof(double)*CHAR_BIT - MANTISSA_BITS - SIGN_BITS
  };
  struct {
    const char *desc;
    union {
      double value;
      unsigned char bytes[sizeof(double)];
      struct {
        unsigned long mantissa : MANTISSA_BITS;
        unsigned long exp : EXP_BITS;
        int sign : SIGN_BITS;
      };
    } u;
  } values[] = {
```

Double Decode Example Continued

```
  { "1.0", {1.0} }, { "-1.0", {-1.0} }, { "0.5", {0.5} },
  {  "2.5", {2.5} }, { "-128.0", {-128.0} },
  //limits
  { "DBL_MIN", {DBL_MIN} }, { "DBL_MAX", {DBL_MAX} },
  { "DBL_EPSILON", {DBL_EPSILON} },
  //unnormalized numbers
  { "Unnormalized DBL_MIN/2", {DBL_MIN/2} },
  { "Unnormalized DBL_MIN/8", {DBL_MIN/8} },
  { "Unnormalized DBL_MIN/64", {DBL_MIN/64} },
  //special numbers
  { "INFINITY", {INFINITY} }, { "-INFINITY", {-INFINITY} },
  { "NAN", {NAN} }, { "-1/INFINITY", {-1/INFINITY} },
};
assert(sizeof(values[0].u) == sizeof(double));
```

Double Decode Example Continued

```
  FILE *out = stdout;
  fprintf(out, "double layout: %d %d %d\n",
          SIGN_BITS, EXP_BITS, MANTISSA_BITS);
  for (int i = 0; i < sizeof(values)/sizeof(values[0]); i++) {
    fprintf(out, "desc = %s; value = %g\n",
            values[i].desc, values[i].u.value);
    fprintf(out, "bytes = ");
    for (int j = sizeof(double); j > 0; j--) {
      fprintf(out, "%02x ", values[i].u.bytes[j-1]);
    }
    fprintf(out, "\n");
    fprintf(out, "sign = %d; ", values[i].u.sign);
    fprintf(out, "exp = %lu; ", (unsigned long)values[i].u.exp);
    fprintf(out, "mantissa = %lx\n\n",
            (unsigned long)values[i].u.mantissa);
  }
  return 0;
}
```

Double Decode Example Log

```
$ ./double-decode
double layout: 1 11 52
desc = 1.0; value = 1
bytes = 3f f0 00 00 00 00 00 00
sign = 0; exp = 1023; mantissa = 0

desc = -1.0; value = -1
bytes = bf f0 00 00 00 00 00 00
sign = -1; exp = 1023; mantissa = 0

desc = 0.5; value = 0.5
bytes = 3f e0 00 00 00 00 00 00
sign = 0; exp = 1022; mantissa = 0

desc = 2.5; value = 2.5
bytes = 40 04 00 00 00 00 00 00
sign = 0; exp = 1024; mantissa = 4000000000000

desc = -128.0; value = -128
bytes = c0 60 00 00 00 00 00 00
sign = -1; exp = 1030; mantissa = 0
```

Double Decode Example Log Continued

```
desc = DBL_MIN; value = 2.22507e-308
bytes = 00 10 00 00 00 00 00 00
sign = 0; exp = 1; mantissa = 0

desc = DBL_MAX; value = 1.79769e+308
bytes = 7f ef ff ff ff ff ff ff
sign = 0; exp = 2046; mantissa = fffffffffffff

desc = DBL_EPSILON; value = 2.22045e-16
bytes = 3c b0 00 00 00 00 00 00
sign = 0; exp = 971; mantissa = 0

desc = Unnormalized DBL_MIN/2; value = 1.11254e-308
bytes = 00 08 00 00 00 00 00 00
sign = 0; exp = 0; mantissa = 8000000000000

desc = Unnormalized DBL_MIN/8; value = 2.78134e-309
bytes = 00 02 00 00 00 00 00 00
sign = 0; exp = 0; mantissa = 2000000000000

desc = Unnormalized DBL_MIN/64; value = 3.47668e-310
bytes = 00 00 40 00 00 00 00 00
sign = 0; exp = 0; mantissa = 400000000000
```

Double Decode Example Log Continued

```
desc = INFINITY; value = inf
bytes = 7f f0 00 00 00 00 00 00
sign = 0; exp = 2047; mantissa = 0

desc = -INFINITY; value = -inf
bytes = ff f0 00 00 00 00 00 00
sign = -1; exp = 2047; mantissa = 0

desc = NAN; value = nan
bytes = 7f f8 00 00 00 00 00 00
sign = 0; exp = 2047; mantissa = 8000000000000

desc = -1/INFINITY; value = -0
bytes = 80 00 00 00 00 00 00 00
sign = -1; exp = 0; mantissa = 0

$
```

Function Types

If *T*, *T1*, *T2*, `...` *Tn* are types, then the declaration *T* `f(`*T1, ... , Tn*`)` declares `f` to be a function taking arguments of types *T1*, *T2*, `...` *Tn* and returning a value of type *T*. Note the following:

- Above types of declarations were not present in K&R C but added in ANSI-C as *prototypes*.
- When declaring functions, parameters may or may not be named: hence both `void f(int i, int j);` and `void f(int, int);` declare `f()` to be a function taking two `int` arguments and not returning any value.
- If the argument list is specified as `()`, then it means taking some unspecified number of arguments of unspecified type. Example: `double f();` means `f()` is a function taking some unspecified number of arguments of unspecified type returning a `double`.

Function Types Continued

- To indicate that a function takes no arguments use `(void)`. So `void procedure(void);` declares `procedure()` to be a function which takes no arguments and does not return a result.

- Functions can take a varying number of arguments, indicated using ellipsis; example `int printf(const char *fmt, ...);`. Can be defined portably using library `stdlib.h`. Usually number of actual arguments specified by explicit arguments or by context.

- Function types are not first-class: they cannot be stored in data-structures, passed as arguments to functions, or returned as function return types. However, **pointers** to functions can achieve those roles (consequently, it is impossible to define a function at runtime, but possible to pass around pointers to existing functions).

No String Type

C does not have any string type. However, it does have **string literals** which are represented as a sequence of characters or escape sequences within double quotes. Those literals stand for the array of character codes for the characters specified by the sequence followed by the `NUL`-character `'\0'`.

```
char hello[] = "hello"; //sizeof(hello) == 6
```

For this course we are ignoring wide characters, wide character strings, and C support for Unicode.

Arrays

- A C array does not know the number of elements it has. When passing arrays it is necessary to explicitly pass the number of elements or use some kind of sentinel value in the array to delimit its end.

- C has only 1-dimensional arrays. Elements of arrays can be arrays, which permits array notation for multiple dimensions for `static` and `auto` (but not dynamic) arrays.

- When declaring a multi-dimensional array, first (leftmost) dimension can be unspecified as in `a[][10][3]`.

- Impossible to have multi-dimensional array as function parameters without all but the leftmost dimension specified. So `int a[][]` is not allowed as a function parameter.

Pointers and Arrays

- Only operations for an array are getting its size or obtaining a pointer to element 0 of the array. All other array ops are really pointer ops.

- `a[i]` is equivalent to `*(a + i)` which is equivalent to `*(i + a)` which is equivalent to `i[a]`.

- `char *msg = "hello";` is not the same as `char msg[] = "hello";`. In the first case, `msg` is a pointer to an anonymous 6-character block of memory initialized to 'h', 'e', 'l', 'l', 'o', '\0' with `sizeof(msg)` equal to the size of a pointer (often 4). In the 2nd case, `msg` indicates a block of memory 'h', 'e', 'l', 'l', 'o', '\0' with `sizeof(msg)` equal to 6.

- It is wrong to have one file define `char msg[] = "hello";` and another file declare `extern char *msg;` as the compiler will generate different access code.

When Arrays are Pointers

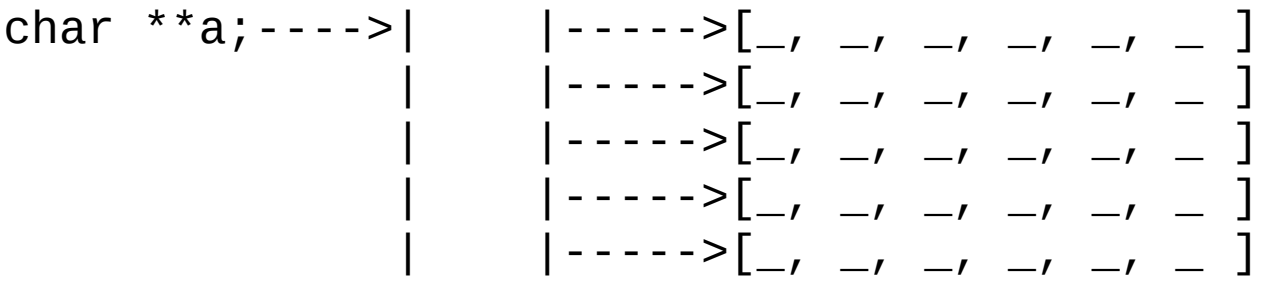
Peter Van Der Linden:

- The use of an array name in an **expression** (not declaration) is treated by the compiler as a pointer to the first element of the array.
- A subscript is always equivalent to an offset from a pointer.
- An array name in the declaration of a function parameter is treated by the compiler as a pointer to the first element of the array. This is because arrays are passed by reference for efficiency reasons (all non-array data are passed by value).

Ragged Arrays

As an alternative to multi-dimensional array of arrays, implement a *n*-dimensional array as a vector of pointers to *(n-1)*-dimensional arrays. Known as **ragged arrays** or **iliffe vectors**. In the 2-dimensional case, this means that each row can have a different number of elements. Saves memory due to ragged rows but uses extra memory for pointer storage. Higher time cost due to extra indirections involved in access.

For example, a 5 x 6 `char` array a can be implemented as follows:



Allows array notation: `a[i]` refers to `i`'th pointer. Hence `a[i][j]` refers to `j`'th char in vector pointed to `i`'th pointer.

Matrix Multiplication: Before C99

Consider declararing a function for matrix multiplication. Before C99, the only possibilities were (assume `c[n1][n3]` is result of multiplying matrix `a[n1][n2]` by `b[n2][n3]`):

```
//Use ragged arrays
void matrix_multiply(double **a, double **b, double **c,
                     int n1, int n2, int n3)
{
  for (int i = 0; i < n1; i++) {
    for (int j = 0; j < n3; j++) {
      c[i][j] = 0;
      for (int k = 0; k < n2; k++) c[i][j] += a[i][k]*b[k][j];
    }
  }
}

//however, something like this is illegal:
void matrix_multiply(double a[][], double b[][], double c[][],
                     int n1, int n2, int n3);

//Could use specific sizes, but then can only multiple matrices of
//specified sizes.
void matrix_multiply(double a[][10], double b[10][5], double c[][5]);
```

Similar problems with dynamically-allocated multi-dimensional arrays.

C99 Matrix Multiplication

C99 allows array dimensions to be specified at run-time. Hence following would work:

```
void matrix_multiply(int n1, int n2, int n3, double a[n1][n2],
                     double b[n2][n3], double c[n1][n3]);
```

but usually:

```
void matrix_multiply(double b[n2][n3], double c[n1][n3], double a[n1][n2],
                   int n1, int n2, int n3);
```

would cause a compilation error or not work as intended.

Using `typedef`'s

`typedef`'s may not always work as intended:

- Usually not a good idea to bury pointers within a `typedef`.

```
typedef struct Person *PersonP;

const PersonP personP;  // const pointer to struct Person;
                        // not pointer to const struct Person.
```

- `typedef`'s do not define a new type; merely an *alias* for an existing type. For example, early Unix versions used `int`'s for process ID and user ID's, but current Unix versions typically use *arithmetic types* `pid_t` and `uid_t` specified as `typedefs`. Typically, this will not catch errors like having a function `f()` declared `void f(pid_t, uid_t)` but calling it as:

```
uid_t uid;
pid_t pid;
...
f(uid, pid)
```

Complex Declarations

- C declarations are hard to read as language designers tried to make the *declaration* of a variable looks just like the *use* of that variable.
- A simple declaration looks like a **basic type** followed by a variable; e.g. `int i;`, `double d;`.
- A complex declaration surrounds the variable with constructors for derived types or grouping parentheses (just as that variable would be used to result in the *basic type*).

Complex Declarations Continued

- Derived types are of 3 types:

  Pointer to
  Denoted using a prefix `*`.

  Array of
  Denoted using a postfix `[]`.

  Function returning
  Denoted using a postfix `()`.

- Note that parentheses are used for two distinct purposes:

  1. The traditional use of grouping to override the default precedence of the operators.

  2. To indicate function arguments.

Decoding Complex Declarations

Following rules result from the fact that postfix type constructors `[]` and `()` have higher precedence than prefix `*`:

1. Start with variable.
2. Traverse postfix type constructors to the right as far as possible, appending the constructors to the type of the variable.
3. Traverse prefix type constructors to the left as far as possible.
4. Repeat steps 2 and 3 as indicated by grouping parentheses.
5. Finally add basic type.

Example of Decoding a Complex Declaration

```
int (*fs[])()
```

1. `fs` is ...
2. `fs` is array of ... (stop because of unbalanced right paren).
3. `fs` is array of pointer to ... (stop when encountering balancing left paren).
4. `fs` is array of pointer to function returning ... (because of `( )` indicating function arguments).
5. `fs` is array of pointer to function returning `int`.

A Practical Example of a Complex Declaration

```
void ( *signal(int sig, void (*handler)(int sig)) )(int sig);
```

`signal` is a function taking 2 arguments `sig` and `handler` returning a pointer to a function taking an `int` and returning `void`. The argument `sig` is an `int`, while `handler` is a pointer to a function taking an `int` and returning `void`.

Can be simplified using a `typedef`:

```
typedef void (*sighandler_t)(int);

sighandler_t signal(int sig, sighandler_t handler);
```

Abstract Declarators

When writing decaration of function arguments, a cast expression or the operand for `sizeof()`, we need to write a type expression without declaring an object of that type. Simply omit the object name. Examples:

```
void f(int* (*)());
```

`f()` is a function with no return value. It takes a single argument which is a pointer to a function taking unspecified arguments and returning a pointer to an `int`.

```
sizeof(int *[3])
```

The size of an array of 3 pointers to `int`.

```
(int (*)[3])p
```

Cast p to pointer to array of 3 `int`.

```
double (*const [10])(unsigned long)
```

An abstract declarator for an array of 10 `const` pointers to functions taking a single `unsigned long` argument and returning a `double`.

Type Declarations Semantic Restrictions

- Cannot have arrays of functions. `int a[]()` illegal. `int (*a[])()` legal.
- Functions cannot return functions. `int (f())()` illegal. `int (*f())()` legal.
- Functions cannot return arrays. `int f()[]` illegal. `int (*f())[]` legal.
- For array declarations only the leftmost [] can be undimensioned.
- `void` can only be used as a pointer base type, function return type or in function prototype to indicate that a function takes no arguments.

Memory Layout Example

```
//inside a function: hence all vars are auto-allocated on stack
typedef struct { short x, y; } Point;
typedef struct {
  const char *desc;
  const char[4] label;
  Point points[2];
} Line;
Line lines[] = {
  { "first", "L1", { { .x = 512, .y = 16 }, { 128, 10 } } },
  { "second", "L2", { { .x = 12, .y = 160 }, { 18, 100 } } },
};
char *lP = &lines[0];
const short *vP = &lines[1].points[1].y;
const char *cP1 = &lines[0].desc[3];
Point *pointP = malloc(sizeof(Point));
char *cP2 = ((char *)&pointP->y) + 1;
```

Memory Layout Example Continued

Assume 32-bit little-endian machine with 2-byte `short`'s. Stack area for above starts at `0x8000`, string constants at `0x1000` and heap at 0x4000.

```
                Stack Area                        String Constants
lines: 0x8000 | 00  10  00  00  | --->  0x1000  |'f' 'i' 'r' 's' |
       0x8004 | 'L' '1' 00  xx  |       0x1004  |'t'  0  xx  xx  |
       0x8008 | 00  02  10  00  |  +->  0x1008  |'s' 'e' 'c' 'o' |
       0x800C | 80  00  0A  00  |  |    0x100C  |'n' 'd'  0  xx  |
       0x8010 | 08  10  00  00  | -+
       0x8014 | 'L' '2' 00  xx  |
       0x8018 | 0C  00  A0  00  |
       0x8024 | 12  00  64  00  |
lP:    0x8030 | 00  80  00  00  |
vP:    0x8034 | 26  80  00  00  |
cP1:   0x8038 | 03  10  00  00  |                    Heap
pointP:0x803C | 00  40  00  00  | ----> 0x4000  | xx  xx  xx  xx |
cP2:   0x8040 | 03  40  00  00  |
```

All numbers in hex (`xx` means undefined); some pointers shown with both lines and values; others shown only with values.

Storage for `"second"` could start immediately after storage for `"first"`.

Function Parameters and Return Values

- C uses *call-by-value*, hence callee cannot change actual argument in caller.

```
static void f(int a) { a = 27; }
...
int n = 2;
f(n);
printf("%d\n", n); //still 2
```

- It follows that the only way a function can return a value to its caller is:
  - Using the function return value.
  - Using global variables (strongly discouraged).
  - Using a pointer parameter to change what the pointer points to in the caller (used by `scanf()` and friends).

Returning Values via Pointer Parameters

- If function `f( )` wants to return an `int` value using a pointer parameter:

```
void f(int *iP) { ...; *iP = fValue; ... }

...
int x = someInitValue;
f(&x); //x is now fValue.
```

- If a function needs to return a pointer value via a function parameter then the caller must pass in the address of a pointer variable. For example, if `f( )` wants to return a pointer to some type `T` via a function argument:

```
void f(T **t) { T *tP = malloc(sizeof(T);  *t = tP; ... }

...
T *t;
f(&t);  //t is value set by f().
```

C Preprocessor Overview

- File inclusion using `#include`.
- Conditional compilation (using `#if`, `#ifdef`).
- Very weak macro processings (textual substitution, no recursion).
- Modern practice discourages complex macros (except for constants).
- Sequel discusses cpp fine-points.

Expression Macros

Macros are not functions:

Consider

```
#define min(a, b)  (a < b) ? a : b
```

But

```
   min(a, b) + 1
== (a < b) ? a : b + 1
== (a < b) ? a : (b + 1)
```

and

```
   min(x&y, z)
== (x&y < z) ? x&y : z
== (x & (y < z)) ? (x & y) : z
```

Expression Macros Continued

The usual fix is to put parentheses around the entire macro body (fixing the first problem above) as well as the use of each macro parameter (fixing the second problem above):

```
#define min(a, b)  (((a) < (b)) ? (a) : (b))
```

But even with the new definition, in `min(a++, b)`, `a` may be incremented twice!!.

Consider expansion of `min(a, min(b, min(c, d)))`:

```
(((a) < ((((b) < ((((c) < (d)) ? (c) : (d))))
 ? (b) : ((((c) < (d)) ? (c) : (d)))))) ? (a)
 : ((((b) < ((((c) < (d)) ? (c) : (d)))) ? (b)
 : ((((c) < (d)) ? (c) : (d))))))
```

Moral: use functions rather than function-like macros whenever possible. Especially, with availability of inline functions in C99.

Statement Macros

Consider following definition:

```
#define dump(x)        fprintf(stderr, "%d\n", x); exit(1);
```

We'd like to use `dump( )` like a procedure. But consider:

```
if (a<0) dump(a); else a = 1;
```

which expands to:

```
if (a < 0) fprintf(stderr, "%d\n", a); exit(1);; else a = 1;
```

which is a syntax error (unmatched `else`).

Statement Macros Continued

Consider putting braces around the macro body:

```
#define dump(x) { fprintf(stderr, "%d\n", x); exit(1); }
```

That would still cause problems in:

```
if (a<0) dump(a); else a = 1;
```

which would expand to:

```
if (a < 0) { fprintf(stderr, "%d\n", a); exit(1); };
else a = 1;
```

which would still have a syntax error.

Statement Macros Continued

A solution (which is a standard idiom) is to wrap the macro body in `do-while(0)`:

```
#define dump(x) \
  do { fprintf(stderr, "%d\n", x); exit(1); } while (0)
```

with

```
if (a<0) dump(a); else a = 1;
```

expanding to:

```
if (a<0)
  do { fprintf(stderr, "%d\n", x); exit(1); } while (0);
else a = 1;
```

with the `;` being absorbed by the `do-while`.

Token Stringification and Concatenation Example Motivation

ANSI-C introduced # for stringification and ## for token concatenation. Check exact semantics which are quite complex.

Following example illustrates two points:

- Important: **Don't Repeat Yourself** (DRY) principle. Something to keep in mind in all programming, irrespective of programming language.

- Trivial: C preprocessor token stringification and concatenation features which are relatively obscure C preprocessor features.

Similar idea used in Fraser and Hanson's `lcc` compiler.

Token Stringification and Concatenation Example

Consider defining a set of enums:

```
typedef enum { RED_C, BLUE_C, GREEN_C, ... } Color;
```

To print out a color, we could print out the enum value:

```
Color c = BLUE_C;
printf("c = %d\n", c);
```

which would print out a 1, which is difficult to interpret.

Token Stringification and Concatenation Example Continued

One solution is to define an array of color-names in parallel with the `enum`:

```
typedef enum { RED_C, BLUE_C, GREEN_C, ... } Color.
const char *colors[] = { "RED", "BLUE", "GREEN", ... };
```

To print out a color we could print out the corresponding array value:

```
Color c = BLUE_C;
printf("c = %s\n", colors[c]);
```

which would print out `BLUE` which is meaningful.

Token Stringification and Concatenation Example Continued

But the problem is that the `colors[]` array must be consistent with the `enum`. If we changed the `enum` but forgot to change the `colors[]` array:

```
typedef enum { WHITE_C, RED_C, BLUE_C, GREEN_C, ... } Color.
const char *colors[] = { "RED", "BLUE", "GREEN", ... };
```

we would get incorrect results:

```
Color c = BLUE_C;
printf("c = %s\n", colors[c]);
```

would print `GREEN`!

Token Stringification and Concatenation Example Continued

A solution is to list the color names in a single place, and have the C preprocessor generate the `enum` and array automatically:

```
#define COLORS \
  T(RED), \
  T(BLUE), \
  T(GREEN), \
  ...
```

To define the `enum`:

```
#undef T
#define T(c) c ## _C
typedef enum { COLORS N_COLORS } Color;
```

where `N_COLORS` serves to absorb the last `,` as well as provide a symbol giving the # of colors.

Token Stringification and Concatenation Example Continued

To define the `colors[]` array:

```
#undef T
#define T(c) #c
const char *colors[] = { COLORS };
```

Header File Inclusion Idiom

To prevent a header file `hdr.h` from being included multiple times because of nested includes, the idiom of enclosing the contents of the header file in a preprocessor conditional is often used:

```
#ifndef _HDR_H
#define _HDR_H

/* Contents of header file */

#endif /* ifndef _HDR_H */
```

Rob Pike objects strongly to this idiom, since this can lead to the header file being read multiple times during a compilation. However, compilers like `gcc` recognize this idiom and avoid reading the file multiple times.

Idiom for Number of Array Elements

Consider array `a[]` declared as:

```
T a[] = { ... };
```

The number of elements in the array is given by the expression:

```
sizeof(a)/sizeof(T)
```

or equivalently:

```
sizeof(a)/sizeof(a[0])
```

Idiom for Number of Array Elements Continued

Could use idiom as:

```
T a[] = { ... };
int i;
for (i = 0; i < sizeof(a)/sizeof(a[0]); i++) {
  ...
}
```

or could package up the idiom in a macro:

```
#define N_ELEMENTS(a) (sizeof(a)/sizeof(a[0]))
```

Note that this idiom works only for `static` or `auto` allocated arrays. It will not work for dynamically allocated arrays.

Asymmetric Bounds

Koenig: Express a range by the first element of the range and the first element beyond it: i.e., inclusive lower bounds and exclusive upper bounds. Example: `10 <= i && i < 20`. Advantages:

- The number of elements is the difference between the bounds. 20 - 10 is 10 elements.
- The bounds are equal when the range is empty.
- The upper bound is never less than the lower bound, even when the range is empty.

The inclusive lower bound gives the first *occupied* element of some sequence and the exclusive upper bound gives the first *free* element of some sequence.

Asymmetric Bounds Continued

Hence use:

```
enum { N = 10 };
int a[N];
int i;
for (i = 0; i < N; i++) { a[i] = 0; }
```

instead of:

```
enum { N = 10 };
int a[N];
int i;
for (i = 0; i <= N - 1; i++) { a[i] = 0; }
```

Asymmetric Bounds Continued

For looping backwards thru an array:

```
enum { N = 10 };
int a[N];
int i;
for (i = N - 1; i > -1; i--) { ... }
```

Asymmetric Bounds Continued

But since an array index can be `unsigned`, if we declare `i` `unsigned`, we can get into trouble:

```
enum { N = 10 };
int a[N];
unsigned i;
for (i = N - 1; i > -1; i--) { ... }
```

One solution:

```
enum { N = 10 };
int a[N];
unsigned i;
for (i = N; i > 0; i--) {
  unsigned j = i - 1;
  /* use j to index a[] */
  ...
}
```

Moral: don't use `unsigned` except for low-level stuff like bit-fields.

External Variable Declarations

Which external variable declarations are *defining occurrences* and which are *referencing occurrences*? There are 4 models:

Initializer Model

The presence of an initializer indicates a defining occurrence (irrespective of absence or presence of the `extern` qualifier). For each variable, there should only be a single defining occurrence in a program but there can be multiple referencing occurrences. This model is adopted by ISO C.

Omitted Storage Class Model

The presence of `extern` indicates a referencing occurrence whereas the absence of `extern` indicates a defining occurrence.

External Variable Declarations Continued

Common Model

All declarations for same variable are merged together at link time with the initializer (if any). If there are multiple initializers, then the results are undefined.

Mixed Common Model

Hybrid between *omitted storage class* and *common* models. Used in many old Unix versions.

External Variable Declarations Continued

Recommended practice:

1. For each external variable have a single definition in only 1 source file. Omit the `extern` qualifier and include an explicit initializer.

```
int flags = 0x0;
```

2. In each source or header file referencing a variable, use the `extern` storage class and do not include a initializer:

```
extern int flags;
```

Static

Distinguish between the general word *static* and the keyword `static`. *Static* data or code is data which has lifetime equal to the lifetime of the program. The keyword `static` implies the *static* property, but the *static* property can also be implied using an implicit or explicit `extern`.

- Static data/code can be declared using the storage specifier `extern` or `static`.
- If `static` or `extern` declaration is within a function, then normal scope rules apply and the declaration is restricted to the scope in which it is declared.
- The scope of a top-level `extern` declaration/definition is the entire program. If a top-level declaration/definition does not have a storage-class specifier then it defaults to `extern`.

Static Continued

- The scope of a top-level `static` declaration/definition is only the current *compilation unit* (file). Hence two top-level `static` definitions of the same identifier in different files are totally independent of each other.

- It is normally good practice to declare top-level functions or data which are only used within a single file `static`.

Using Data Instead of Code

Given complex code, extract complexity into regular data and write simple code which is an interpreter for the data.

Typical example is a *finite-state machine*. Can be implemented in complex code where state corresponds to position in the code and transitions encoded using tests and `goto`'s. Alternately, represented using a state transition table with a simple interpreter.

Using Data Instead of Code Continued

Another example: consider a sequence of tests:

```
if (complex_cond1) {
  action1;
}
else if (complex_cond2) {
  action2;
}
...
else if (complex_condn) {
  actionN;
}
```

Using Data Instead of Code Continued

Replace with functions `complex_cond1()`, `action1`, ..., `complexCondN()`, `actionN` with a table (`Cond` and `Action` are assumed to be `typedef`'d to suitable function pointers):

```
struct { Cond cond; Action action } condActions[] = {
  { complexCond1, action1 },
    ...
  { condN, actionN }
};

for (int i = 0;
     i < sizeof(condActions)/sizeof(condActions[0]); i++) {
  if (condActions[i].cond()) {
    condActions[i].action();
    break;
  }
}
```

Dynamic dispatch in OOP is when basically the conditions are type tests.

Name Spaces

The same identifier can be used for different purposes, even in the same scope, if the uses are in different name spaces. The different name spaces are:

- Objects (variables), functions, typedef and `enum`-constants.
- Labels.
- `struct`, `union` and `enum` tags.
- A separate name space for the fields of each `struct` or `union`.

Name Spaces Example

```
int f(int a) {
  typedef struct f {
    int f;
    struct f *a;
  } F;
  F s;
  f: /* label */
  ...
}
```

Dynamic Memory Allocation

Minimally, return values from `malloc()` and friends must be checked. Can be implemented by using simple wrapper functions:

```
#include <stdio.h>

void *mallocChk(size_t s) {
  void *p = malloc(s);
  if (!p) {
    perror("memory allocation error");
    exit(1);
  }
  return p;
}
```

Dynamic Memory Allocation Continued

Raise abstraction-level for memory allocation by using sub-allocators:

Chunk Allocation

Allocate type `T` in chunks of `N` elements at a time; Link all free `T` elements into a free-list and manage free-list with allocation/deallocation requests for individual elements. If the free-list exhausted at an allocation request, allocate a new chunk.

Vector Allocation

Implement dynamically resizable vectors which grow as needed. Can be done using `realloc()` or ragged array of arrays.

Dynamic Memory Allocation Continued

Pool Allocation

Use the fact that data structures are often allocated and freed together. Hence write a suballocator which controls a pool of memory; the suballocator merely allocates memory without any deallocation. When the data structures in the pool are no longer needed, the entire pool is destroyed and returned to the `malloc()` allocator.

Generalized to resource pools in Apache's httpd server.

Garbage Collection

It is possible to use garbage collection in C/C++ programs using *conservative garbage collection* techniques.

Abstract Data Types in C

An ADT hides the implementation of a data type. The data type is available to clients only via certain published operations.

- Implemented in C using a header file to publish the ADT syntactic specification.
- Header file declares the ADT type as a opaque incomplete `struct`.
- Operation declarations only use pointers to ADT type.
- Implementation file completes definition of ADT type.

ADT Example

Normal ADT example is a stack. Sample header file stack.h (errors ignored):

```
#ifndef _STACK_H
#define _STACK_H

typedef struct StackImpl Stack; /* incomplete struct */
typedef void *StackEntry;     /* generic entry */

Stack *new_stack(void);
void free_stack(Stack *s);
void push_stack(Stack *s, StackEntry e); /*  */
StackEntry pop_stack(Stack *s);
int size_stack(const Stack *s);

#endif /*ifndef _STACK_H */
```

ADT Stack Implementation

Expand incomplete `struct StackImpl` declaration for a stack implemented as a fixed size array in implementation file stack.c:

```
enum { MAX_STACK_SIZE = 10 };

struct StackImpl {
  /** index of next free entry. */
  int index;
  /** storage for stack entries */
  StackEntry entries[MAX_STACK_SIZE];
};

Stack *
new_stack(void)
{
  return calloc(1, sizeof(Stack));
}

void
free_stack(Stack *s) {
  return free(s);
}
```

ADT Implementation Continued

```
void
push_stack(Stack *s, StackEntry e) {
  s->entries[s->index++] = e;
}

StackEntry
pop_stack(Stack *s) {
  return (s->index > 0) ? s->entries[--s->index] : NULL;
}

int
size_stack(const Stack *s) {
  return s->index;
}
```

References

Christopher Fraser and David Hanson *A Retargetable C Compiler: Design and Implementation*, Addison-Wesley, 1995.

Steve Friedl, *Reading C type declarations*, at http://www.unixwiz.net/techtips/reading-cdecl.html

Samuel P. Harbison and Guy L. Steele Jr., *C: A Reference Manual*, 5th Edition, Prentice-Hall, 2002.

Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, 2nd Edition, Prentice-Hall, 1988.

Andrew Koenig, *C Traps and Pitfalls*, Addison-Wesley, 1989.

Rob Pike, *Notes on Programming in C*, at http://www.lysator.liu.se/c/pikestyle.html. Written in 1989, but still worth a read; I disagree on not using camel-cased identifiers and the section on include files.

References Continued

Steve Summit, *C Programming FAQs*, Addison-Wesley, Nov. 1995. Partially online at http://www.eskimo.com/~scs/C-faq/top.html.

Peter Van Der Linden, *Expert C Programming: Deep C Secrets*, Prentice-Hall, 1994.

C99 Draft Standard at http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf.

C11 Draft Standard at http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1570.pdf.

Memory Management Reference at http://www.memorymanagement.org.

http://www.slideshare.net/olvemaudal/deep-c/ Strongly recommended.

Lockheed Martin Corporation, *Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program*, Dec, 2005 at http://www.stroustrup.com/JSF-AV-rules.pdf.

2004 MISRA C Standard, guidelines for automotive programming at http://caxapa.ru/thumbs/468328/misra-c-2004.pdf. Newer version of the standard is also available at the MISRA web site. Summary at http://home.sogang.ac.kr/sites/gsinfotech/study/study021/Lists/b7/Attachments/91/Chap%207.%20MISRA-C%20rules.pdf.