Operating Systems and C

Fall 2022

5. C Primer

Reading Code

btest.c
(datalab)

what does it all mean?
today, goal: reference for reading C.

IT UNIVERSITY OF COPENHAGEN

btest.c

```c
/* Not declared in some stdlib.h files, so define here */
float strtof(const char *nptr, char **endptr);

/*************************
 * Configuration Constants
 *************************/

/* Handle infinite loops by setting upper limit on execution time, in
   seconds */
#define TIMEOUT_LIMIT 10

/* For functions with a single argument, generate TEST_RANGE values
   above and below the min and max test values, and above and below
   zero. Functions with two or three args will use square and cube
   roots of this value, respectively, to avoid combinatorial
   explosion */
#define TEST_RANGE 500000

/* This defines the maximum size of any test value array. The
   gen_vals() routine creates k test values for each value of
   TEST_RANGE, thus MAX_TEST_VALS must be at least k*TEST_RANGE */
#define MAX_TEST_VALS 13*TEST_RANGE

/**********************************
 * Globals defined in other modules
 **********************************/
/* This characterizes the set of puzzles to test.
   Defined in decl.c and generated from templates in ./puzzles dir */
extern test_rec test_set[];

/************************************************
 * Write-once globals defined by command line args
 ************************************************/

/* Emit results in a format for autograding, without showing
   and counter-examples */
static int grade = 0;

/* Time out after this number of seconds */
static int timeout_limit = TIMEOUT_LIMIT; /* -T */

/* If non-NULL, test only one function (-f) */
static char* test_fname = NULL;

/* Special case when only use fixed argument(s) (-1, -2, or -3) */
static int has_arg[3] = {0,0,0};
static unsigned argval[3] = {0,0,0};

/* Use fixed weight for rating, and if so, what should it  be? (-r) */
static int global_rating = 0;
```

The Dark Side

```
int * (* (*fp1) (int) ) [10];

char *const *(*next) ();
```

to e.g. read something like this.
(there's a method to the madness)

Outline

1. **Pointers**
2. Declarations and definitions
3. Type specifiers and qualifiers
4. Type conversions
5. Symbol overloading
6. Operator precedence
7. Unscrambling declarations

C pointers

”A pointer is a variable that contains the address of a variable.”

K & R

that’s really all. (cf. data representation).
small level of abstraction on top of `mov`:

Pointers let C programmers directly control CPU addressing.

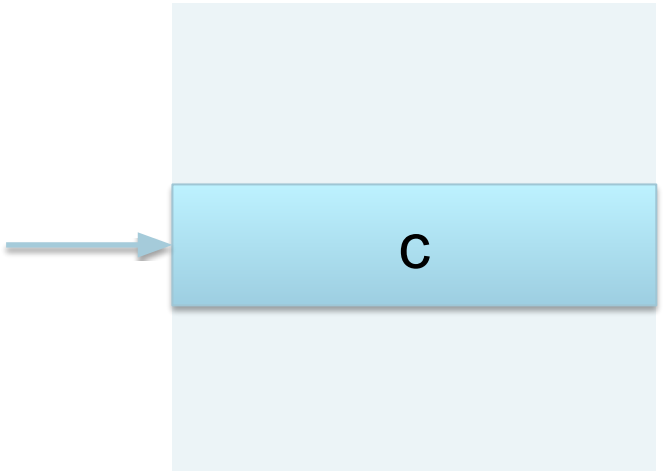
Notation

p is a char pointer. It is a variable that contains the address of a char variable.

1 byte (memory is byte-addressable)

```
char *p;
char c;
p = &c;
```

(char *) &c



address-of

Q: how many bytes (or bits) to represent a pointer?

can also write `char* p;`
Linux: next to var name

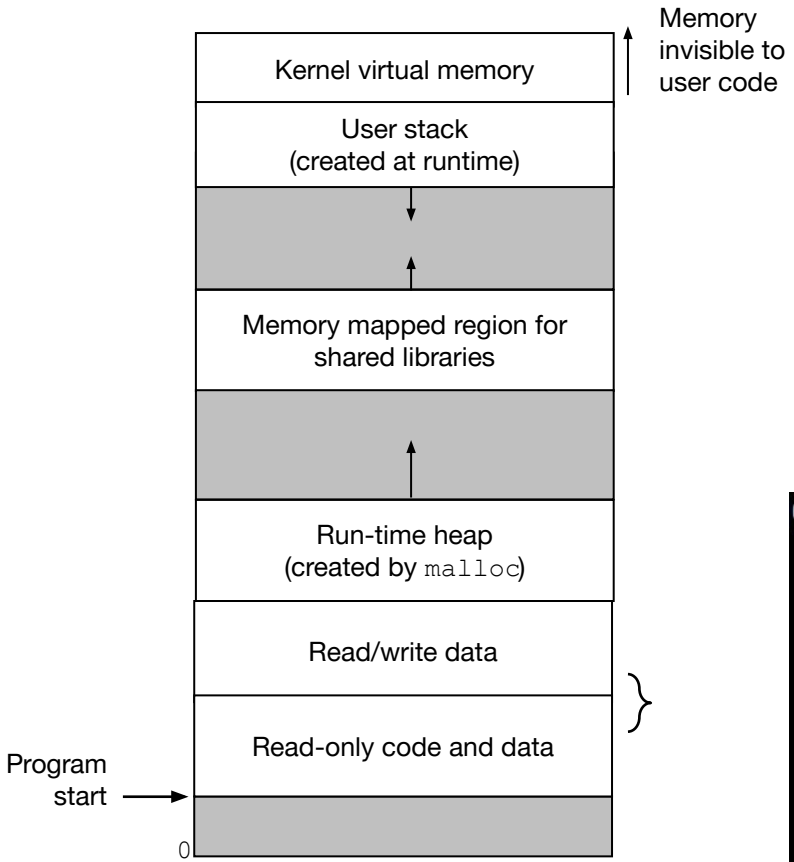
Why pointers?

To manage data placement / locality and memory allocation (**explicit memory management**).

To share data without copies

To manage indirections
(more on this later with function pointers)

Virtual Memory



a pointer can point to anything in virtual memory. (usually stack, heap) (program = data)

```
(gdb) x /10i main
   0x400596 <main>:     push   %rbp
   0x400597 <main+1>:   mov    %rsp,%rbp
   0x40059a <main+4>:   sub    $0x60,%rsp
   0x40059e <main+8>:   mov    %edi,-0x54(%rbp)
   0x4005a1 <main+11>:  mov    %rsi,-0x60(%rbp)
   0x4005a5 <main+15>:  mov    %fs:0x28,%rax
   0x4005ae <main+24>:  mov    %rax,-0x8(%rbp)
   0x4005b2 <main+28>:  xor    %eax,%eax
   0x4005b4 <main+30>:  movb   $0x50,-0x20(%rbp)
   0x4005b8 <main+34>:  movb   $0x68,-0x1f(%rbp)
```

Exercise

Write type declarations for the following variables:

- q: a pointer to an integer pointer
- t: a pointer to a byte
- u: a pointer to a byte array

Q: spend 2 mins on this

Exercise

Consider the following code:

le3ex1.c

```c
#include <stdio.h>

int main() {
    int *ptr;
    *ptr = 20;
    printf("%d\n", *ptr);
    return 0;
}
```

What is wrong?

Exercise

Consider the following code:

le3ex1.c

```c
#include <stdio.h>

int main() {
    int *ptr;
    *ptr = 20;
    printf("%d\n", *ptr);
    return 0;
}
```

not initialized (indeterminate pointer); where to store the 20?

What is wrong?

(compiler *should* complain, but compiler *could* do anything)

Pointers

Pointers are valid, null or indeterminate.

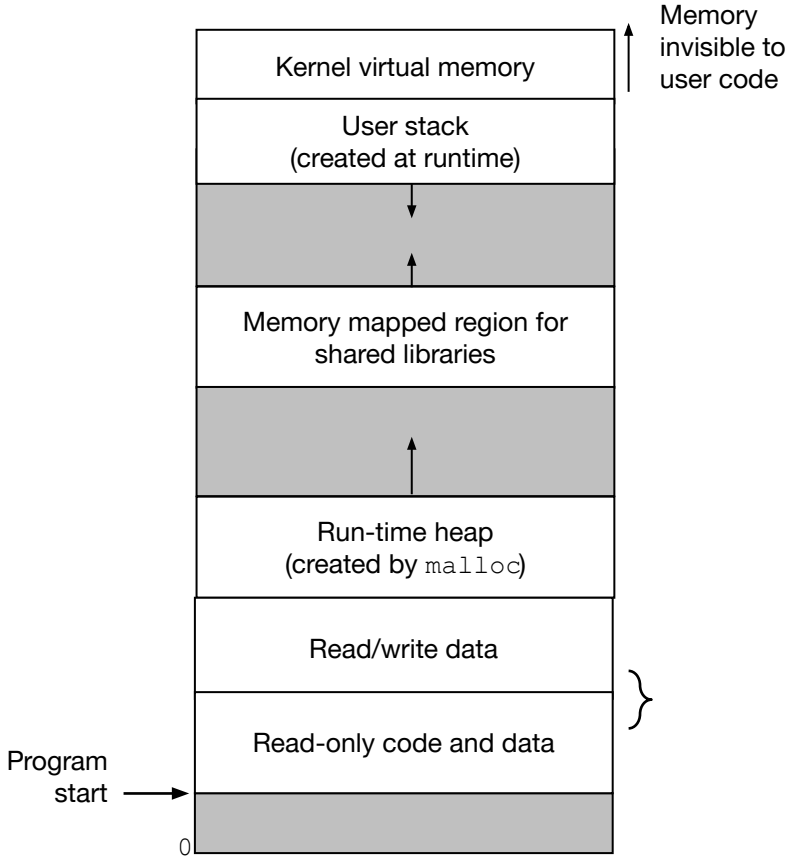
A pointer is null when assigned 0

Null pointers evaluate to false in logical expressions

Dereferencing indeterminate pointers leads to **undefined behaviour**

Always initialize pointers!

Virtual Memory



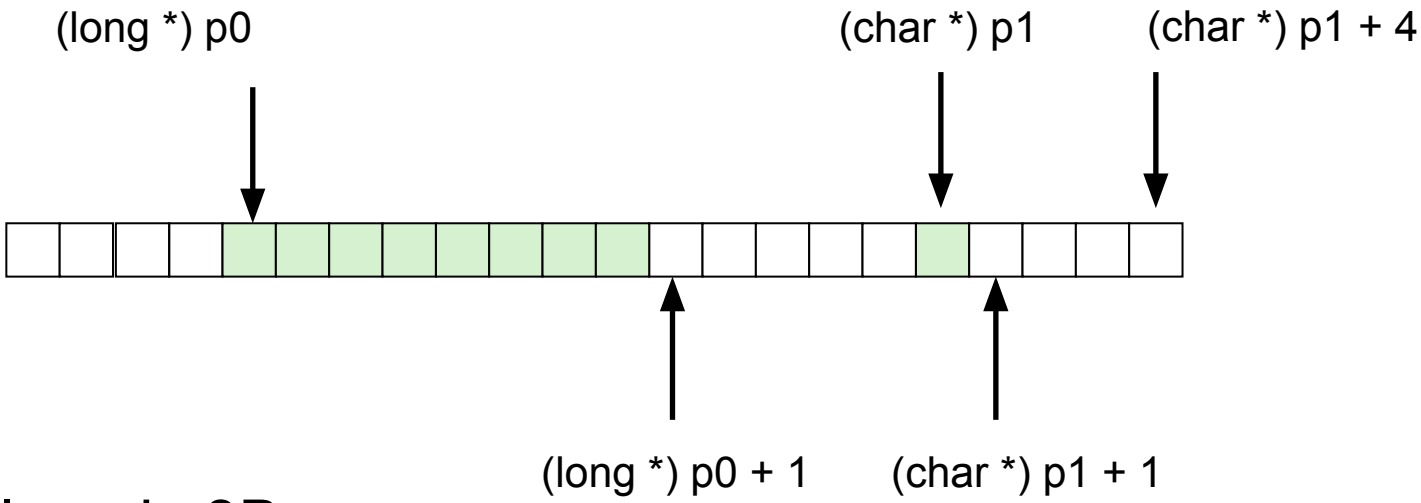

```
phbo@mac610891 ~/D/C/BOSC-E19> cc -Wall l3ex1.c -o l3ex1
l3ex1.c:5:3: warning: variable 'ptr' is uninitialized when used here [-Wuninitialized]
        *ptr = 20;
         ^~~
l3ex1.c:4:10: note: initialize the variable 'ptr' to silence this warning
        int *ptr;
                ^
                 = NULL
1 warning generated.
```

compiler decided to initialize to null.

```
(gdb) p ptr
$2 = (int *) 0x0
(gdb) p &ptr
$3 = (int **) 0x7fffffffe3a8
```

Pointer Arithmetic

recall: 1 cell is 1 byte.
memory is byte-addressable.



A long is 8B
A char is 1B

you can do arith. on addr.
how far you skip, depends on
type of pointer.

Array and pointers
NOTATIONS

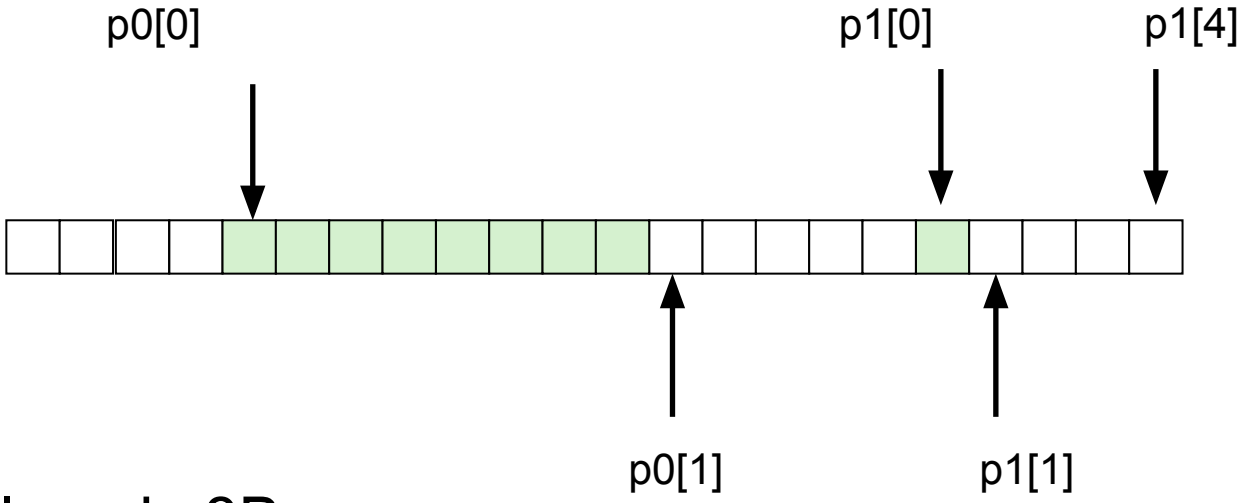
<u>Notations</u> can be used **interchangeably**:

a[i] *is equivalent to* *(a+i)

Regardless of whether a is declared as an array or a pointer

array **represented as a pointer to** the first element of array in memory. (but array ≠ pointer; see next+1 slide)

Pointer Arithmetic



A long is 8B
A char is 1B

Pointers are NOT arrays #1

(1) Arrays have a size, pointers do not !

```
int* a;
int b[10];

a = b;
```

a now points to &b[0], the **size is lost**

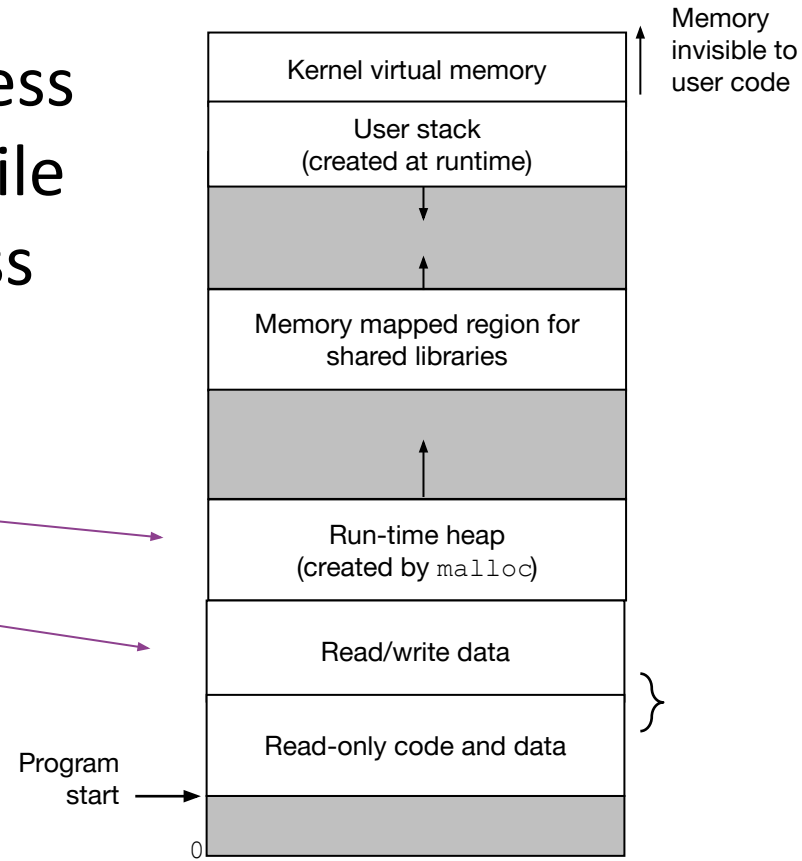
you have extra information in array, that you do not have in pointer.

Pointers are NOT arrays #2

(2) Arrays are assigned an address in memory at compile time, while pointers are assigned an address in memory at run time.

```
int* a;
int b[10];

a = b;
```

must specify array size, so compiler can allocate space at compile time



Binky Video

old instructional video from Stanford

Outline

1. Pointers
2. **Declarations and definitions**
3. Type specifiers and qualifiers
4. Type conversions
5. Symbol overloading
6. Operator precedence
7. Unscrambling declarations

Declaration and definition

Definition: specifies
what a function does or where a variable is stored.

Declaration: describes type/name of variable/function.
No space is allocated.

Variables and functions are defined exactly once,
but may be declared several times.

W: think of dec as "intent",
and def as "enact"

def fun: what it does
def var: how it's stored
dec: just a signature.

dec: x exists,
def: dec + allocate mem for x

Scope of variables

A variable defined in a function is local to that function. It is an **automatic** variable. It does not retain its value across function calls (lives in stack frame).

A variable defined outside any function is an *external* variable. It is a **global** variable.

Before a global variable can be accessed in other files, it must be declared with the `extern` prefix.

A global variable does not need to be declared in the file where it is defined.

typically, these are collected in a header file.

Scope of variables

The scope of a global variable can be <u>restricted</u> to the file where it is defined with the `static` prefix.

`static` and `extern` are mutually exclusive.

An automatic variable can retain its value across calls to a function when it is defined with `static`.

Recap so far

What is an automatic variable?

A: local to function

How is a global variable <u>def</u>ined when it can be accessed by all C files contributing to an executable?

A: in files where not <u>def</u>, it must be <u>dec</u> w/ `extern`

How is a global variable <u>def</u>ined when it can only be accessed from the C file where it is <u>def</u>ined?

A: `static`

How is a global variable <u>dec</u>lared outside the file where it is <u>def</u>ined?

A: `extern`

Example

function definition; we are specifying what function does

local to function, but retains values across calls.

scope.c

```c
#include<stdio.h>
int fun()
{
        static int count = 0;
        count++;
        return count;
}

int main()
{
        printf("%d ", fun());
        printf("%d ", fun());
        return 0;
}
```

```
% make scope
cc      scope.c    -o scope
% ./scope
1 2
```

Example

scope2.c

```c
extern int var;
int main(void)
{
        var = 10;
        return 0;
}
```

```
% make scope2
cc     scope2.c   -o scope2
Undefined symbols for architecture x86_64:
  "_var", referenced from:
      _main in scope2-4b8294.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
make: *** [scope2] Error 1
```

solution:
define it in a header file. (next)

Example

int var implicitly defined (to 0)

scope2.c

```c
#include "scope2.h"

extern int var;
int main(void)
{
        var = 10;
        return 0;
}
```

scope2.c scope2.h

```c
int var;
```

```
% make scope2
cc      scope2.c    -o scope2
```

Restrictions

You can't have:

- A function that returns a function

  Never foo()()
- A function that returns an array

  Never foo()[]
- An array of function

  Never foo[]()

(type system doesn't allow it, despite conceptually making sense)

But you can have

A function returning a pointer to a function

*fun() ()

A function returning a pointer to an array

*fun()[]

An array of function pointers

*foo[]()

(due to restrictions in C type system, we use pointers as an indirection-level)

Outline

data science students: if you use Cython https://cython.org/ (optimizing compiler for Python), then you must understand C type system.

1. Pointers
2. Declarations and definitions
3. **Type specifiers and qualifiers**
4. Type conversions
5. Symbol overloading
6. Operator precedence
7. Unscrambling declarations

Type Specifiers

specifies how many bytes, and how interpreted by ops.

- **char, int, short, long, float, double**
- **signed / unsigned**

- Pointer: *****
- Array: **[]**

Reading Code

btest.c

defined in another file
(which: comment is helpful)

local to this file

```
btest.c
25
26 /* Not declared in some stdlib.h files, so define here */
27 float strtof(const char *nptr, char **endptr);
28
29 /*************************
30  * Configuration Constants
31  *************************/
32
33 /* Handle infinite loops by setting upper limit on execution time, in
34    seconds */
35 #define TIMEOUT_LIMIT 10
36
37 /* For functions with a single argument, generate TEST_RANGE values
38    above and below the min and max test values, and above and below
39    zero. Functions with two or three args will use square and cube
40    roots of this value, respectively, to avoid combinatorial
41    explosion */
42 #define TEST_RANGE 500000
43
44 /* This defines the maximum size of any test value array. The
45    gen_vals() routine creates k test values for each value of
46    TEST_RANGE, thus MAX_TEST_VALS must be at least k*TEST_RANGE */
47 #define MAX_TEST_VALS 13*TEST_RANGE
48
49 /**********************************
50  * Globals defined in other modules
51  **********************************/
52 /* This characterizes the set of puzzles to test.
53    Defined in decl.c and generated from templates in ./puzzles dir */
54 extern test_rec test_set[];
55
56 /************************************************
57  * Write-once globals defined by command line args
58  ************************************************/
59
60 /* Emit results in a format for autograding, without showing
61    and counter-examples */
62 static int grade = 0;
63
64 /* Time out after this number of seconds */
65 static int timeout_limit = TIMEOUT_LIMIT; /* -T */
66
67 /* If non-NULL, test only one function (-f) */
68 static char* test_fname = NULL;
69
70 /* Special case when only use fixed argument(s) (-1, -2, or -3) */
71 static int has_arg[3] = {0,0,0};
72 static unsigned argval[3] = {0,0,0};
73
74 /* Use fixed weight for rating, and if so, what should it  be? (-r) */
75 static int global_rating = 0;
```

Type specifiers

Struct

Union

Enum

Type specifiers: struct

a **record**

Struct: a bunch of data items grouped together (in memory)

```
struct tag {
        type_1 identifier_1;
        type_2 identifier_2;
        ...
        type_N identifier_N;
} ;
struct tag variable_name;
```

Type specifier: struct

The data items in struct are accessed through dot operator. When using a *pointer to struct*, the data items dereferenced through the pointer are accessed through arrow operator.

```
/* struct that points to the next struct */
struct node_tag {
    int datum;
    struct node_tag *next;
};
struct node_tag a,b;
a.next = &b;
a.next->next=NULL;
```

def

dec

def

def

foo->bar shorthand for (*foo).bar

shorthand for (*(a.next)).next

Type specifiers: struct

giving names to bits inside a struct.

Structs can have bit fields, unnamed fields, and word-aligned fields.

```
/* process ID info */
struct pid_tag {
    unsigned short int inactive :1;
    unsigned short int :1;          /* 1 bit of padding */
    unsigned short int refcount :6;
    unsigned short int :8;          /* pad to short length */
    short pid_id;
    struct pid_tag *link;
};
```

1 unsigned short; naming its bits

Type specifier: struct

```c
#include <stdio.h>
#include <string.h>

struct {
   unsigned int age : 3;
} Age;

int main( ) {

   Age.age = 4;
   printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
   printf( "Age.age : %d\n", Age.age );

   Age.age = 7;
   printf( "Age.age : %d\n", Age.age );

   Age.age = 8;
   printf( "Age.age : %d\n", Age.age );

   return 0;
}
```

```
phbo@parallels-vm ~/C/C/Lectures> ./l3ex2
Sizeof(Age): 4
Age.age: 4
Age.age: 7
Age.age: 0
```

Q: if I assign 10, and print, what gets printed?

Type specifier: struct, the beauty of

Definition
Space is reserved

```
struct s_tag { int a[100]; };
struct s_tag orange, lime, lemon;
struct s_tag twofold (struct s_tag s) {
    int j;
    for (j=0;j<100;j++) s.a[j] *= 2;
    return s;
}

main() {
    int i;
    for (i=0;i<100;i++) lime.a[i] = 1;
    lemon = twofold(lime);
    orange = lemon; /* assigns entire struct */
}
```

array as input

function cannot return an array,
but
function can return a struct.
(cf. returning a pointer)

this will **copy** the entire structure!
(if that's not what you want, then use pointers)

Type specifier: union

Unions have a similar appearance to structs, but the memory layout has one crucial difference. **Instead of each member being stored after the end of the previous one, all the members have an offset of zero**. The storage for the individual members is thus <u>overlaid</u>: only one member at a time can be stored there.

```
union bits32_tag {
    int whole; /* a 4B value */
    struct {char c0,c1,c2,c3;} byte; /* 4 * 1B values */
}
```

example use: TCP frames

Type specifier: enum

Enums (enumerated types) are simply a way of associating a series of names with a series of integer values.

```
enum sizes { small=7, medium, large=10, humongous };
```

8 (7+1)

11

Type qualifier #1: const

const qualifies a read-only variable; one that cannot be a left value in an assignment following the variable declaration.

const.c

```c
int main()
{
        const int i;
        i = 12;

        return 0;
}
```

```
% make const
cc      const.c   -o const
const.c:4:4: error: cannot assign to variable 'i' with const-qualified type 'const int'
        i = 12;
        ~ ^
const.c:3:12: note: variable 'i' declared const here
        const int i;
        ~~~~~~~~~~^
1 error generated.
make: *** [const] Error 1
```

Type qualifier #1: const

The combination of const and * is usually only used to simulate call-by-value for array parameters. It says, "I am giving you a pointer to this thing, but you may not change it."
Expert C programming

```
int * const p;
// p cannot be left value in an assignment
```

Type qualifier #1: const

(note: cannot have)
```
const int limit;
limit = 10;
```

```
const int limit = 10;
const int * limitp =
&limit;
int i=27;
limitp = &i;
```

pointer to constants. can point to other constants (i.e. something that cannot be on lhs of assignment.

```
int limit = 10;
int * const limitp =
&limit;
int i=27;
limitp = &i;
```

constant pointer. cannot point to other things. (**W:** but can overwrite pointee; can e.g. update limit)

Type qualifier #2: volatile

`volatile` qualifies a variable that might be modified outside the program.

For example, a register that can be modified by a device can be tested/read repeatedly by a program that never modifies it directly.

Assigning a `volatile` object to a pointer results in undefined behaviour.

Type qualifier #2: volatile

can be initialized outside your program.

```
struct devregs{
   unsigned short volatile csr;
   unsigned short const volatile data;
};
```

void

Void is the type of a function that does not return a result.

void *

void * defines a pointer to data of unspecified type.

Outline

1. Declarations and definitions
2. Type specifiers and qualifiers
3. **Type conversions**
4. Symbol overloading
5. Operator precedence
6. Unscrambling declarations

Type conversions

Explicit:

A value of one type is explicitly cast to another type

Implicit:

1. A value of one type is assigned to a variable of a different type
2. An operator converts the type of its operands
3. **A value is passed as argument to a function or when a value is returned from a function**

Unsigned and signed

Same bit level representation, different interpretations

If there is a mix of unsigned and signed in single expression, *signed values are implicitly cast to unsigned*

be very careful
(recall kernel mem copy)

Pointer conversions

- A pointer to one type of value can be converted to a pointer to a different type. However, the result may be undefined because of the alignment requirements and sizes of different types in storage.
- A pointer to an object can be converted to a pointer to an object whose type requires less or equally strict storage alignment, and back again without change.
- A pointer to void can be converted to or from a pointer to any type, without restriction or loss of information. If the result is converted back to the original type, the original pointer is recovered.
- If a pointer is converted to another pointer with the same type but having different or additional qualifiers, the new pointer is the same as the old except for restrictions imposed by the new qualifier.

Pointer conversions

A pointer value can also be converted to an integral value. The conversion path depends on the size of the pointer and the size of the integral type:

- If the size of the pointer is greater than or equal to the size of the integral type, the pointer behaves like an unsigned value. It cannot be converted to a floating value.
- If the pointer is smaller than the integral type, the pointer is first converted to a pointer with the same size as the integral type, then converted to the integral type.

Conversely, an integral type can be converted to a pointer type according to the following rules:

- If the integral type is the same size as the pointer type, the conversion simply causes the integral value to be treated as a pointer (an unsigned integer).
- If the size of the integral type is different from the size of the pointer type, the integral type is first extended or truncated to fit the size of the pointer. It is then treated as a pointer value.

Pointer conversions

skip

Name

malloc, free, calloc, realloc - allocate and free dynamic memory

Synopsis

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

```
/* j is a pointer to an array of 20 char */
char (*j)[20];
j = (char (*)[20]) malloc( 20 );
```

Type compatibility: Subtleties

skip

```
char * cp;
const char *ccp;
ccp = cp;
```

```
foo(const char **p) { }

main(int argc, char **argv)
{
foo(argv);
}
```

OK if both operands are pointers to qualified or unqualified versions of compatible types, and type pointed to by the left has all the qualifiers of the type pointed to by the right.

Outline

1. Declarations and definitions
2. Type specifiers and qualifiers
3. Type conversions
4. **Symbol/keyword overloading**
5. Operator precedence
6. Unscrambling declarations

Keyword Overloading

From expert C programming

Symbol	Meaning
`static`	Inside a function, *retains its value between calls* At the function level, *visible only in this file* [1]
`extern`	Applied to a function definition, *has global scope* (and is redundant) Applied to a variable, *defined elsewhere*
`void`	As the return type of a function, *doesn't return a value* In a pointer declaration, the type of a generic pointer In a parameter list, *takes no parameters*

can be used for different things.
(just need to be aware of the overloading)

Symbol Overloading

From expert C programming

lol

lol

lol

lol

*	The multiplication operator Applied to a pointer, indirection In a declaration, a pointer
&	Bitwise AND operator Address-of operator
= ==	Assignment operator Comparison operator
<= <<=	Less-than-or-equal-to operator Compound shift-left assignment operator
< <	Less-than operator Left delimiter in #include directive
()	Enclose formal parameters in a function definition Make a function call Provide expression precedence Convert (cast) a value to a different type Define a macro with arguments Make a macro call with arguments Enclose the operand of the sizeof operator when it is a typename

Outline

1. Declarations and definitions
2. Type specifiers and qualifiers
3. Type conversions
4. Symbol/keyword overloading
5. **Operator precedence**
6. Unscrambling declarations

Operator Precedence

From http://en.cppreference.com/w/c/language/operator_precedence

a mess.

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	( )	Function call	
	[ ]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(*type*){*list*}	Compound literal(C99)	
2	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	Type cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
	_Alignof	Alignment requirement(C11)	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13[note 1]	?:	Ternary conditional[note 2]	Right-to-Left
14	=	Simple assignment	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= |=	Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

Operator Precedence

"Some operators have the wrong precedence"

Kernighan and Ritchie.

Operator Precedence

From expert C programming

skip

Precedence problem	Expression	What People Expect	What They Actually Get
. is higher than * the p->f op was made to smooth over this	`*p.f`	the f field of what p points to `(*p).f`	take the f offset from p, use it as a pointer `*(p.f)`
[] is higher than *	`int *ap[]`	ap is a ptr to array of ints `int(*ap)[]`	ap is an array of ptrs-to-int `int *(ap[])`
function () higher than *	`int *fp()`	fp is a ptr to function returning int `int (*fp) ()`	fp is a function returning ptr-to-int `int *(fp())`
== and != higher precedence than bitwise operators	`(val&mask != 0)`	`(val&mask) !=0`	`val & (mask !=0)`
== and != higher precedence than assignment	`c=getchar() !=EOF`	`(c=getchar()) != EOF`	`c=(getchar() !=EOF)`
arithmetic higher precedence than shift	`msb<<4 + lsb`	`(msb<<4)+lsb`	`msb<<(4+lsb)`
, has lowest precedence of all operators	`i = 1,2;`	`i= (1,2);`	`(i=1), 2;`

Operator Precedence

Always put **parentheses** around an expression that mixes booleans, arithmetic, or bit manipulation with anything else.

when in doubt: use parentheses!
has higher precedence over rest.

Outline

1. Declarations and definitions
2. Type specifiers and qualifiers
3. Type conversions
4. Symbol/keyword overloading
5. Operator precedence
6. **Unscrambling declarations**

The rules for understanding C declarations

1. Declarations are read by <u>starting with the name</u> (of the variable, function or type)
2. The following precedence rules apply:
   A. Parentheses grouping together part of the declaration
   B. The postfix operators
      - Parenthesis indicating a function
      - Square brackets indicating an array
   C. The prefix operator
      - * denoting a pointer to
3. If a const or volatile is next to a type specifier it qualifies it, otherwise const or volatile applies to the * on its immediate left

The Dark Side

```
char* const *(*next)();
```

Q: so, what is this? (brief pause; next)

Understanding C declarations

Next is	(1)
a pointer to	(2A)
a function returning	(2B)
a pointer to	(2C)
a constant pointer to	(3)
char	

The Dark Side

```
int * (* (*fp1) (int) ) [10];
```

Q: so, what is this? (brief pause; next)

The Dark Side

```
% cdecl
Type `help' or `?' for help
cdecl> explain int * (* (*fp1) (int) ) [10]
declare fp1 as pointer to function (int) returning pointer to array 10 of pointer to int
cdecl>
```

How about?

```
char *(*c[10])(int **p)
```

Q: so, what is this? (brief pause; next)

Solution

c is a array 10 of pointer to function (pointer to pointer to int) that returns pointer to char

Reading Code

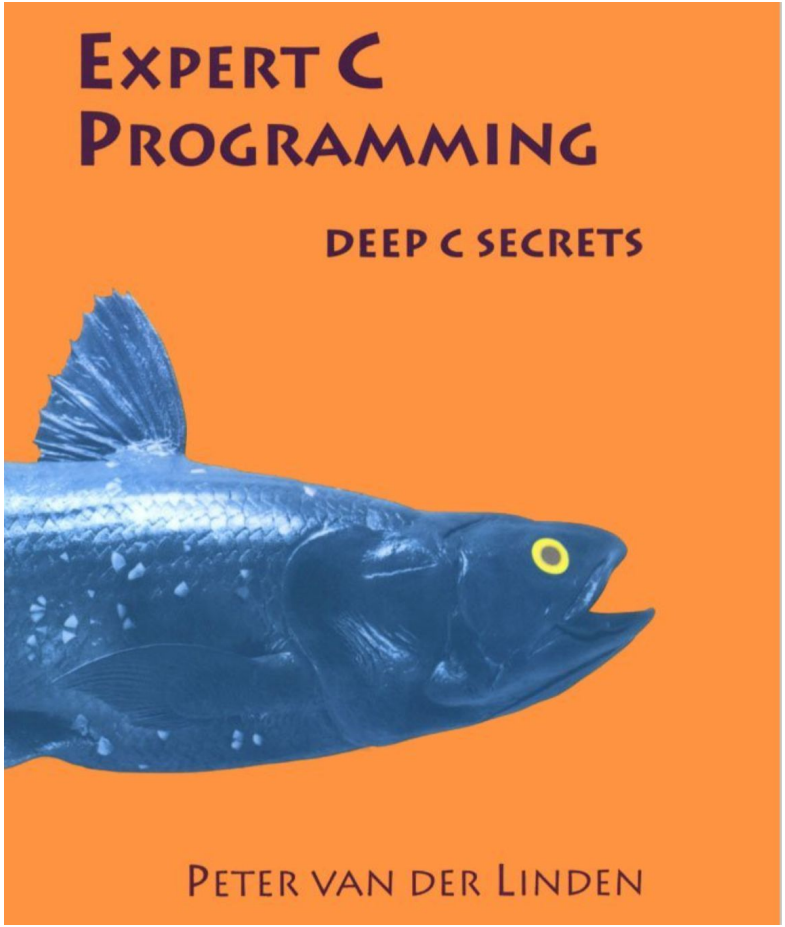
now you can read code.

btest.c

```c
/* Not declared in some stdlib.h files, so define here */
float strtof(const char *nptr, char **endptr);

/*************************
 * Configuration Constants
 *************************/

/* Handle infinite loops by setting upper limit on execution time, in
   seconds */
#define TIMEOUT_LIMIT 10

/* For functions with a single argument, generate TEST_RANGE values
   above and below the min and max test values, and above and below
   zero. Functions with two or three args will use square and cube
   roots of this value, respectively, to avoid combinatorial
   explosion */
#define TEST_RANGE 500000

/* This defines the maximum size of any test value array. The
   gen_vals() routine creates k test values for each value of
   TEST_RANGE, thus MAX_TEST_VALS must be at least k*TEST_RANGE */
#define MAX_TEST_VALS 13*TEST_RANGE

/**********************************
 * Globals defined in other modules
 **********************************/
/* This characterizes the set of puzzles to test.
   Defined in decl.c and generated from templates in ./puzzles dir */
extern test_rec test_set[];

/***********************************************
 * Write-once globals defined by command line args
 ***********************************************/

/* Emit results in a format for autograding, without showing
   and counter-examples */
static int grade = 0;

/* Time out after this number of seconds */
static int timeout_limit = TIMEOUT_LIMIT; /* -T */

/* If non-NULL, test only one function (-f) */
static char* test_fname = NULL;

/* Special case when only use fixed argument(s) (-1, -2, or -3) */
static int has_arg[3] = {0,0,0};
static unsigned argval[3] = {0,0,0};

/* Use fixed weight for rating, and if so, what should it  be? (-r) */
static int global_rating = 0;
```

Expert C Programming

the bible.
it's really old.
best ref for C programming



Take-Aways

You should remember:

1. A pointer is a variable that contains the address of a variable
2. The difference between declaration and definition
3. The scope of variables (automatic / global)
4. The difference between type qualifier and specifier
5. The meaning of const and volatile
6. The nature of structs
7. When type conversions takes place
8. What happens when signed and unsigned are mixed
9. Beware operator precedence
10. Use cdecl when in doubt about a declaration

packed together

qualifier: const, volatile, …
specifier: char, int, *, …