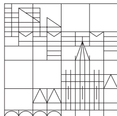
The C Programming Language - Part II

Accompanying Tutorial to Operating Systems Course

Alexander Holupirek, Stefan Klinger

Database and Information Systems Group
Department of Computer & Information Science
University of Konstanz

Summer Term 2009



Schedule For Today

So far: Learned about the memory layout of a C program

- Observed correlation between storage class specifiers, sections in ELF file and location in virtual memory
- Discussed basic C language constructs

Today: Some of the (important) di?erences to Java

- Dynamic allocation of memory on the heap
- A closer look at **pointers and arrays**
- Introduce structures, unions, typedefs, enumerations

Dynamic Memory Allocation

Pointers and Addresses

Pointers and Function Arguments

Pointers, Arrays and Address Arithmetic

Character Pointers and C Strings

Basics of Structures

Self-referential Structures

Unions

Enumerations

Typedef

Pointers to Functions

Dynamic Memory Allocation - malloc(3)

malloc(3) and calloc(3) obtain blocks of memory

```
#include <stdlib.h>

void *
malloc(size_t size);

void *
calloc(size_t nmemb, size_t size);
```

malloc(3)

- Returns a pointer to size bytes of uninitialized storage
- NULL if the request can not be satisfied

Dynamic Memory Allocation - calloc(3)

malloc(3) and calloc(3) obtain blocks of memory

```
#include <stdlib.h>

void *
malloc(size_t size);

void *
calloc(size_t nmemb, size_t size);
```

calloc(3)

- calloc(3) returns a pointer to enough space for an array of nmemb objects of the specified size
- NULL if the request can not be satisfied
- The storage is initialized to zero

Extend Or Reduce Allocated Memory

realloc(3) modifies size of (previously) allocated memory

```
#include <stdlib.h>

void *
realloc(void *ptr, size_t size);
```

realloc(3)

- realloc(3) changes the size of the object pointed to by ptr to size bytes
- It returns a pointer to the (possibly moved) object
- Be sure to not have any more references to the old location
- The content of the old memory area is automatically moved to the new location

Alignment Of Obtained Storage

A pointer to dynamically allocated storage

- `void *` is the proper type for a *generic* pointer
- It is a pointer to any data type, but has to be converted (*coerced*) into some other type before usage
- The pointer returned by `malloc(3)` or `calloc(3)` has the proper alignment for the object in question
- It can be cast explicitly into the appropriate type

```
int *ip;

/* explicit cast to coerce returned void pointer */
ip = (int *) calloc(n, sizeof(int));
```

Freeing Allocated Memory

free(p) frees the space pointed to by p

- Only storage obtained by malloc or calloc can be freed
- If p is a NULL pointer, no action occurs
- It is an error to use something after it has been freed

Incorrect code that frees items from a list:

```
for (p = head; p != NULL; p = p->next) /* WRONG */
  free(p);
```

The right way is to save whatever is needed before freeing:

```
for (p = head; p != NULL; p = q) {
  q = p->next;
  free(p);
}
```

Dynamic Memory Allocation

Pointers and Addresses

Pointers and Function Arguments

Pointers, Arrays and Address Arithmetic

Character Pointers and C Strings

Basics of Structures

Self-referential Structures

Unions

Enumerations

Typedef

Pointers to Functions

Pointers And Addresses

- A **pointer** is a variable that contains the address of a variable
- Pointer also carries the notion of what *type* of data it points to

Let's try to illustrate this observing running programs:

```
int
main(void)
{
    char c;
    char *p;

    c = '@';  /* @ = 0x40 */
    p = &c;

    return (0);
}
```

Simplified Picture Of Memory Organization

Memory as array of consecutively numbered memory cells

- Can be manipulated individually or in contiguous groups
- Typical situation: Any byte (often the shortest accessible unit) can be a `char`; `sizeof(char)` is one by definition
- Adjacent bytes may form a short, integer, long, "string" ...
- A pointer is a group of cells that can hold an address

```
p: 0xcfbe5174:     0x7b -.  4 cells interpreted as address
           75:     0x51  |-------------------------.
           76:     0xbe  |      (little endian)    |
           77:     0xcf -'                         |
             :     ----                            |
             :     ----                            |
             :     ----                            |
c: 0xcfbe517b:     0x40 <-  1 memory cell ('@')  <-'
```

The Unary Address Operator: &

```
p: 0xcfbe5174:     0x7b -.  4 cells interpreted as address
           75:     0x51  |-------------------------.
           76:     0xbe  |      (little endian)     |
           77:     0xcf -'                          |
            :      ----              p = &c;        |
            :      ----                             |
            :      ----                             |
c: 0xcfbe517b:     0x40 <-  1 memory cell ('@')  <-'
```

- Unary operator & gives the address of an object
- `p = &c;` the address of c is assigned to p
- p is said to *point to* c
- & operator only applies to objects *in memory*
  - variables, array elements, and functions

Pointing To Integers

- As said, the pointer is associated with its *type*
- Let us now consider a pointer to an int

```
int
main ( void )
{
     unsigned int i ;
     unsigned int *pi ; /* pointer to int */

     i = 0xdeadbeaf ;
     pi = &i ; /* address of i in pointer variable */

     return (0);
}
```

Simplified Picture Of Memory Organization II

```
(gdb) p /x i                      /* examine content of i in hex */
$2 = 0xdeadbeaf
(gdb) p pi                                /* print content of pi */
$3 = (unsigned int *) 0xcfbe2958 /* address uint is stored */
(gdb) x /4b 0xcfbe2958  /* examine 4 bytes at this address */
0xcfbe2958: 0xaf  0xbe  0xad  0xde          /* little endian */
(gdb) p &pi     /* print the address of the pointer to int */
$4 = (unsigned int **) 0xcfbe2954
(gdb) x /4b 0xcfbe2954         /* print what is stored there */
0xcfbe2954: 0x58  0x29  0xbe  0xcf       /* the address of i */
```

```
pi:  0xcfbe2954:    0x58 -.
             55:    0x29  |
             56:    0xbe  | ----.
             57:    0xcf -'     |  pointer to int
                                |
 i:  0xcfbe2958:    0xaf   <----'
             59:    0xbe
             5a:    0xad
             5b:    0xde
```

The Unary Dereferencing Operator: *

```
(gdb) p /x &pi
$10 = 0xcfbe2954
(gdb) p pi
$3 = (unsigned int *) 0xcfbe2958
(gdb) p /x i
$13 = 0xdeadbeaf

pi:  0xcfbe2954:     0xcfbe2958  /* address of an int */
 i:  0xcfbe2958:     0xdeadbeaf

(gdb) p /x *pi  /* print (in hex) the object pointed at */
$9 = 0xdeadbeaf
(gdb) p /x *&i
$12 = 0xdeadbeaf
(gdb) p /x *i
Cannot access memory at address 0xdeadbeaf
```

- Unary operator * is the *indirection* or *dereferencing* operator
- Applied to a pointer the object the pointer points to is accessed

Artificial Pointer Operations

```
int x = 1, y = 2, z[10];

int *ip; /* ip is a pointer to int */

ip = &x; /* ip now points to x (contains address of x) */

y = *ip; /* y is now 1 */

*ip = 0; /* x is now 0 */

ip = &z[0]; /* ip now points to z[0] */
```

Declaration Of A Pointer

The declaration of the pointer `ip` is intended as a mnemonic

```
int *ip;
```

- It says that the expression `*ip` is an `int`
- The syntax of the declaration for a variable mimics the syntax of expressions in which the variable might appear
- This reasoning applies to function declarations as well:

```
double *dp, atof(char *);
```

- It says that in an expression `*dp` and `atof(s)` have values of type `double`, and the argument of `atof` is a pointer to `char`

Pointers Reference A Specific Data Type

A pointer is constrained to point to a particular kind of object

- Exception: Pointer to `void` is used to hold any type of pointer
  $\longrightarrow$ Can not be dereferenced
- Assume `ip` points to the integer `x`
- Then `*ip` can occur in any context where `x` could:

```
1 int *ip;
2 int x = 0;
3
4  ip = &x;
5 *ip = *ip + 10;
```

- Line 5 increments `*ip` (and therefore `x`) by 10

Binding Of Unary Operators

Unary operator * and & bind more tightly than arithmetic ones

```
y = *ip + 1;
```

- takes whatever ip points at
- adds 1, and assigns the result to y

```
*ip += 1;
++*ip;
(*ip)++;
```

- Increment what ip points to
- The parentheses are necessary in this last example.

*Without, the expression would increment ip instead of what it points to. Unary operators like * and ++ associate right to left.*

Precedence And Associativity Of Operators

OPERATORS	ASSOCIATIVITY
`( ) [ ] -> .`	left to right
`! ~ ++ -- + - * &` *(type)* `sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	left to right
`= += -= *= /= %= &= ^= |= <<= >>=`	right to left
`,`	left to right

Table: Unary +, −, and ∗ have higher precedence than the binary forms

Pointers Are Variables

Pointers are variables and can be used without dereferencing

```
iq = ip;
```

- Copies the contents of `ip` into `iq`
- Makes `iq` point to whatever `ip` pointed to

Dynamic Memory Allocation

Pointers and Addresses

Pointers and Function Arguments

Pointers, Arrays and Address Arithmetic

Character Pointers and C Strings

Basics of Structures

Self-referential Structures

Unions

Enumerations

Typedef

Pointers to Functions

Passing Function Arguments By Value

- In C arguments to functions are passed by value

⇒ Called function can not alter a variable in the calling function

```
void swap(int, int);

void
swap(int x, int y)   /* WRONG */
{
  int tmp;

  tmp = x;
  x = y;
  y = tmp;
}
```

- The function only swaps *copies* of a and b

Passing Pointers To Functions

- Pointer arguments enable a function to access and change objects in the function called it

```c
void swap(int *, int *);

void
swap(int *px, int *py)
{
  int tmp;

  tmp = *px;
  *px = *py;
  *py = tmp;
}
```

```
swap: px:  ---.
              |
      py:  ---+--.
              |  |
              |  |
              |  |
              |  |
              |  |
caller: a: <--'  |
                 |
        b: <-----'
```

Dynamic Memory Allocation

Pointers and Addresses

Pointers and Function Arguments

Pointers, Arrays and Address Arithmetic

Character Pointers and C Strings

Basics of Structures

Self-referential Structures

Unions

Enumerations

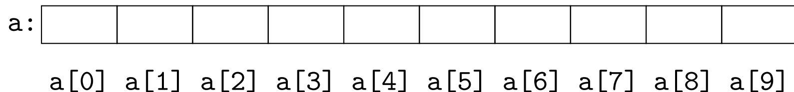
Typedef

Pointers to Functions

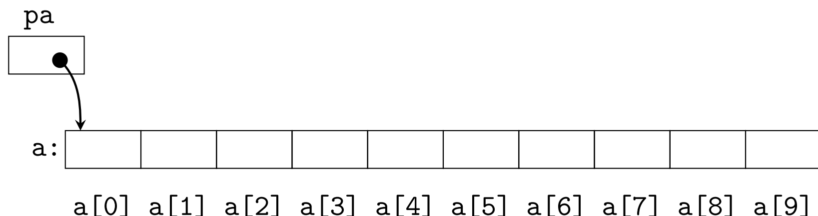
Pointers And Arrays

- There is a strong relationship between pointers and arrays
- Any operation achieved by array subscripting ...
- ... can also be done with pointers

```
int a[10];  /* Define an array a of size 10 */
int *pa;    /* Pointer to an integer */
int x;
```

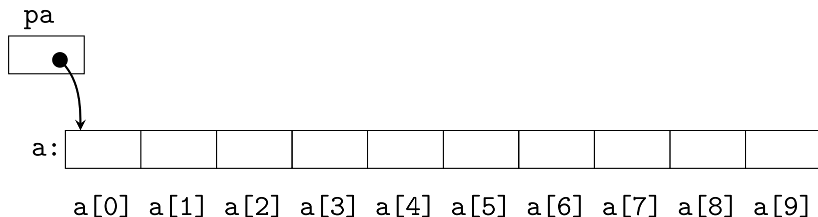


Pointer Into Array



- Assignment `pa = &a[0];`
- Sets pa to point to element zero of a
- pa contains the address of a[0]

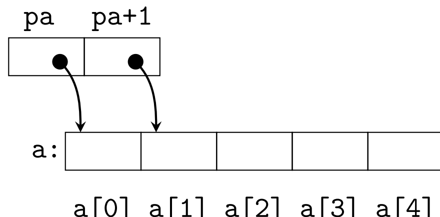
Pointer Into Array (cont.)



- Assignment `x = *pa;`
- Copies the content of `a[0]` into `x`

Adding 1 To A Pointer

- If `pa` points to a particular element of an array
- Then, by definition, `pa+1` points to the next element

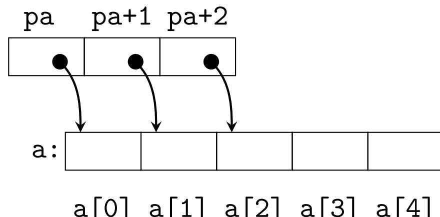


In general:

- `pa+i` points `i` elements after `pa`
- `pa-i` points `i` elements before

Adding i To A Pointer

- Thus, if pa points to a[0]
- Then *(pa+2) refers to the contents of a[2]



In general:

- pa+i is the address of a[i]
- *(pa+i) is the contents of a[i]

Adding A Pointer And An Integer

A pointer and an integer may be added (or substracted)

- The construction `p + n` means the address of the `n`-th object beyond the one `p` currently points to
- `n` is scaled according to the size of the object `pa` points to (which is determined by the declaration of `p`)
- Holds regardless of the type or size of the variables in the array
- If an `int` is four bytes, for example, `n` is scaled by four

Array-and-Index And Pointer-and-Offset

Very close correspondence between indexing and pointer arithmetic

Equivalence of array-and-index and pointer-and-offset expr.:

```
 a[i]  <==>  *(a+i)
&a[i]  <==>  a+i
pa[i]  <==>  *(pa+i)
```

- `a[i]` is converted to `*(a+i)` immediately (in evaluation)
- Applying the & address operator to both sides of equivalence
- If `pa` is a pointer, expressions may use it with a subscript

Indexing Backwards

- With pointers into arrays we can use pointer arithmetic to access nearby cells of the array
- If we are sure that an element exist, it is also possible to index backwards in an array `p[-1]`, `p[-2]` and so on
- This refers to objects before what `p` points to
- Illegal to refer to objects that are not within the array bounds

```
p[-1]  ⟺  *(p + -1)  ⟺  *(p - 1)
```

Figure: Expressions `p[-1]` converted to the pointer form

Example: Scaling According To Type

```
int a[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
char a[] = "abcdefghij";
```

a+i	a[i]	(a+i+1) - (a+i)	(a+i+1) - (a+i)
0xcfbf2a44	0	0xcfbf2a48 - 0xcfbf2a44	1
0xcfbf2a48	1	0xcfbf2a4c - 0xcfbf2a48	1
0xcfbf2a4c	2	0xcfbf2a50 - 0xcfbf2a4c	1
...	...	...	...
0xcfbf2a74	a	0xcfbf2a75 - 0xcfbf2a74	1
0xcfbf2a75	b	0xcfbf2a76 - 0xcfbf2a75	1
0xcfbf2a76	c	0xcfbf2a77 - 0xcfbf2a76	1
...	...	...	...

Difference Between Pointer And Array Name

One difference between an array name and a pointer:

- A pointer is a variable
- An array name is not a variable

As such:

```
int *pa;
int a[3];

pa = a; /* legal */
pa++;   /* legal */

/* a = pa;  illegal */
/* a++;     illegal */
```

```
*a = 1;
*(a + 1) = 2;
*(a + 1) = 3;
*(a + 2) = 4;

printf("%d, %d, %d\n",
    a[0], a[1], a[2]);
```

Arrays In C

The value of a variable or expression of type array is the address of element zero of the array

- In most languages, the value of an array is the entire array
- If an array appears on the right-hand sign of an assignment, the entire array is assigned, and the left-hand side had better be an array, too
- C *does not* work this way
- C never lets you manipulate entire arrays

Array Name As Address Of First Element

Definition:

The value of a variable (or expression) of type array is the address of element zero of the array

- The value of an array, when it appears in an expression, is a pointer to its first element
- Shorter: The value of the array a simply is `&a[0]`

```
/* equivalent assignments/rhs expressions*/
  pa = &a[0];
  pa = a;
```

- One can often read that, when an array appears in an expression, it *decays* into a pointer to its first element

Passing An Array To A Function

When an array name is passed to a function

- what is passed is the location of the initial element
- what is passed is a pointer
- what is passed is a variable containing an address

As a consequence, within the called function, this argument is a local variable

An Array As Formal Argument

Function definition:

As *formal parameters* `char s[]` and `char *s` are equivalent

- The latter may be prefered, because it says more explicitly that the parameter is a pointer
- When an array name is passed to a function, the function can at its convenience believe that it has been handed either an array or a pointer, and manipulate it accordingly
- It can even use both notations if it seems appropriate and clear

Accordingly, within `f`, the parameter declaration can read:

```
f(int arr[]) { ... }
  or
f(int *arr) { ... }
```

Passing Parts Of An Array To A Function

- It is possible to pass part of an array to a function, by passing a pointer to the beginning of the subarray:

`f(&a[2]);` $\iff$ `f(a+2);`

Figure: Pass address of subarray that starts at `a[2]` to the function `f`

- So as far as `f` is concerned, the fact that the parameter refers to part of a larger array is of no consequence

Computing String Length Using A Pointer

```c
/* strlen: return the length of string s */
int
strlen(char *s)
{
        int n;

        for (n = 0; *s != '\0'; s++)
                n++;

        return n;
}
```

- Since `s` is a pointer, incrementing it is perfectly legal
- `s++` has no effect on the character string in the caller function
- It merely increments `strlen`'s private copy of the pointer

Legal Calls To strlen?

Given the last two slides, which calls will work?

```
strlen("hello, world"); /* string constant */
strlen(array);          /* char array[100]; */
strlen(ptr);            /* char *ptr; */
```

The NULL pointer

- Pointers and integers are not interchangeable
- Zero is the sole exception
- The constant zero may be assigned to a pointer, and a pointer may be compared with the constant zero
- The symbolic constant `NULL` is often used in place of zero
- It is a mnemonic remainder to indicate more clearly that this is a special value for a pointer
- `NULL` is defined in `<stdio.h>`

```
#define NULL    0L
```

Comparison Of Pointers

Pointers may be compared under certain circumstances

- If `p` and `q` point to members of the same array
- Then relations like `==`, `!=`, `<`, `>=`, etc. work properly
- `p < q`, for instance, is true if `p` points to an earlier member of the array than `q` does
- The behaviour is undefined for arithmetic or comparisons with pointers that do not point to members of the same array
- There is one exception: The address of the first element past the end of an array can be used in pointer arithmetic

Pointer Substraction

Pointer substraction to determine string length

- If p and q point to elements of the same array and p < q
- Then q-p+1 is the number of elements from p to q inclusive
- This fact can be used to write yet another version of strlen

```
/* strlen: return length of string s */
int
strlen(char *s)
{
        char *p = s;

        while (*p != '\0')
                p++;
        return p - s;
}
```

Computing strlen Using A Pointer Substraction

```
/* strlen: return length of string s */
int
strlen(char *s)
{
        char *p = s;

        while (*p != '\0')
                p++;
        return p - s;
}
```

- In its declaration, p is initialized to s
- That is, to point to the first character of the string
- while loop: examine each char until '\0' is seen
- p points to characters $\rightarrow$ p++ advances p to the next char
- Finally, p - s gives the number of characters advanced over, i.e., the string length

Valid Pointer Operations

Legal pointer operations summarized

- Assignment of pointers of the same type
- Adding or subtracting a pointer and an integer
- Subtracting or comparing two pointers to members of the same array
- Assigning or comparing to zero
- All other pointer arithmetic is illegal

Illegal pointer operations

- Not legal to add, multiply, divide, shift, or mask two pointers
- Add `float` or `double` to pointers
- Assign a pointer of one type to a pointer of another type without cast (Exception is `void *`)

Dynamic Memory Allocation

Pointers and Addresses

Pointers and Function Arguments

Pointers, Arrays and Address Arithmetic

Character Pointers and C Strings

Basics of Structures

Self-referential Structures

Unions

Enumerations

Typedef

Pointers to Functions

String Constants, String Literals

A string constant or string literal

- written as `"I am a string"` is an array of characters
- is (automatically) terminated with the null character `'\0'`
- occupies one more length in storage than the number of characters between the double quotes

Accessing string constants

- Most often string constants appear as arguments to functions
- For example: `printf("Hello World!\n");`
- Access to the constant is provided through a character pointer
- `printf` receives a pointer to the beginning of the char array

→ A string constant is accessed by a pointer to its first element

Character Pointers And C Strings

Text strings are represented by arrays of characters

- Since arrays are very often manipulated via pointers, character pointers are probably the most common pointers in C

Good to remember:

C does not provide any operators for processing an entire string of characters as a unit

- What are we doing here?

```
char *pmessage;
pmessage = "now is the time";
pmessage = "hello, world";
```

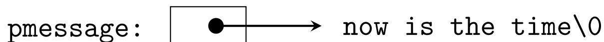
- Assigning two pointers, not copying two entire strings

Character Pointers & Character Arrays Differ

What is the difference between these two?

```
char *pmessage = "now is the time";
char amessage[] = "now is the time";
```

Basic illustration of how pointer and array differ:



amessage: now is the time\0

Different Ways String Literals Are Used

```c
char amessage[] = "now is the time";
```

- String literal used as initializer for the array amessage
- amessage is an array of 16 characters
- We may overwrite them at a later point → writeable

```c
char *pmessage = "now is the time";
```

- String literal is used to create a little block of characters somewhere in memory
- Pointer pmessage is initialized to point to it
- We may reassign pmessage later to point somewhere else
- As long as it points to the string literal, we can not modify the characters it points to

Exemplify Different Usage Of String Literals

```
char amessage[] = "now is the time";
char *pmessage = "now is the time";
```

```
amessage[0] = 'N';
```

- Perfectly right → `"Now is the time"`

```
pmessage[0] = 'N';
```

- Equivalent to `*pmessage = 'N'`
- Would not necessarily work, in fact, it is not allowed
- Why?

Exemplify Different Usage Of String Literals

```
char *pmessage = "now is the time";
pmessage[0] = 'N';
```

- Compiler might have placed the "little block of characters" in read-only memory
- Consider the usage of

```
char *pmessage = "now is the time";
char *qmessage = "now is the time";
```

- Compiler might have used the same little block of memory to initialize both pointers
- We wouldn't want a change to one to alter the other

String Copy

Direct consequence of not treating a string as a unit

We need functions to copy string `t` to string `s` or to compare them

- Would be nice to say `s = t`
- However, this will only copy pointers, not the characters
- To copy the characters we need a loop

```
void
strcpy(char s[], char t[])
{
        int i;

        for(i = 0; t[i] != '\0'; i++)
                s[i] = t[i];
        s[i] = '\0';
}
```

String Copy Using Pointers

Pointer Version of strcpy()

```
void
strcpy(char *s, char *t)
{
        while(*t != '\0')
                *s++ = *t++;
        *s = '\0';
}
```

- The value `*t++` is the character that `t` pointed to before `t` was incremented
- The postfix `++` doesn't change `t` until after this character has been fetched

Incidental Remark On Standard Idioms

- Expressions like `*p++` and `*--p` may seem cryptic at first sight
- `*--p` decrements `p` before fetching the character `p` points to

→ Analogous to array subscript exp. like `a[i++]` and `a[--i]`

Standard idioms for pushing and popping a stack

```
*p++ = val;  /* push val onto stack */
 val = *--p; /* pop top of stack into val */
```

Comparing Strings

Comparison of strings is analogous to copying

- We cannot assign one string to another using "="
- Same holds for the comparison of two strings using "=="
- What we would compare with "==" is the two pointers
- If the pointers are equal, they point to the same place, so they certainly point to the same string
- But if we have two strings in two different parts of memory, pointers to them will always compare different even if the strings pointed to contain identical sequences of characters

Source Code: String Comparison

`strcmp(char *s, char *t)`

- It compares the character strings `s` and `t`, and returns negative, zero or positive if `s` is lexicographically less than, equal to, or greater than `t`
- "Greater than" and "less than" are interpreted based on the relative values of the characters in the machine's character set
- This means that `'a' < 'b'`
- Also (at least with ASCII character set): `'B' < 'a'`
- In other words, capital letters will sort before lower-case letters
- The positive or negative number returned is the difference between the values of the first two characters that differ

Source Code: String Comparison

```
/* strcmp:  return <0 if s<t, 0 if s==t, >0 if s>t */
int
strcmp(char *s, char *t)
{
        int i;

        for (i = 0; s[i] == t[i]; i++)
                if (s[i] == '\0')
                        return (0);
        return s[i] - t[i];
}
```

```
/* strcmp as pointer version */
int
strcmp(char *s, char *t)
{
        for (; *s == *t; s++, t++)
                if (*s == '\0')
                        return (0);
        return *s - *t;
}
```

Intermediate Summary

Character pointer derived from string literal

```c
char *pmessage = "now is the time";
```

- is usable (*i.e.*, readable)
- but not writable
- that is, the characters pointed to are not writable

Destination string has to be a writable array with enough space

- for the number of characters in the string we are copying
- plus one for the terminating `'\0'`

What About Those Code Snippets?

```
char *p1 = "Hello, world!";
char *p2;
strcpy(p2, p1);
```

Wrong. p2 doesn't point anywhere

```
char *p = "Hello, world!";
char a[13];
strcpy(a, p);
```

Wrong. Array a is writable, but there is not enough space for `'\0'`

```
char *p3 = "Hello, world!";
char *p4 = "A string to overwrite";
strcpy(p4, p3);
```

Wrong. p4 points to memory not allowed to be overwritten

Valid Code Snippet & string.h

A correct example would be:

```
char *p = "Hello, world!";
char a[14];
strcpy(a, p);
```

- Another option is to obtain some memory for the string copy

→ Dynamic memory allocation for destination string

String manipulation with the standard library

The header `<string.h>` contains declarations for the functions mentioned in this section, plus a variety of other string-handling functions from the standard library.

Some More Versions Of strcpy(s, t)

Coding Style

- In the following we revisit the `strcpy` function
- Three rather compressed version are shown

```
/* strcpy: copy t to s; array subscript version */
void
strcpy(char *s, char *t)
{
        int i;

        i = 0;
        while ((s[i] = t[i]) != '\0')
                i++;
}
```

strcpy(s, t): Pointer Version 1/3

```
/* strcpy: copy t to s; pointer version 1 */
void
strcpy(char *s, char *t)
{
        while ((*s = *t) != '\0') {
                s++;
                t++;
        }
}
```

- Because arguments are passed by value, `strcpy` can use the parameters `s` and `t` in any way it pleases
- Here they are conveniently initialized pointers, which are matched along the arrays one character at a time . . .
- . . . until the `'\0'` that terminates `t` has been copied to `s`

strcpy(s, t): Pointer Version 2/3

```
/* strcpy: copy t to s; pointer version 2 */
void
strcpy(char *s, char *t)
{
        while ((*s++ = *t++) != '\0')
                ;
}
```

- Increment of s and t moved into the test part of the loop
- *t++ is the char that t pointed to before t was incremented
- postfix ++ doesn't change t until the char has been fetched
- It is stored into the old s position before s is incremented
- It is also the value that is compared against '\0'
- Effect: chars are copied from t to s, up to and including '\0'

strcpy(s, t): Pointer Version 3/3

```
/* strcpy: copy t to s; pointer version 3 */
void
strcpy(char *s, char *t)
{
        while (*s++ = *t++)
                ;
}
```

- We can observe that a comparison against `'\0'` is redundant
- The question is merely whether the expression is zero
- Idioms like this are frequently used in C programs
- What do you think about these abbreviations?

Dynamic Memory Allocation

Pointers and Addresses

Pointers and Function Arguments

Pointers, Arrays and Address Arithmetic

Character Pointers and C Strings

Basics of Structures

Self-referential Structures

Unions

Enumerations

Typedef

Pointers to Functions

Structures

A structure is a collection of one or more variables

- possibly of different types
- grouped together under a single name for convenient handling

A structure

- organizes complicated data, particulary in large programs
- permits a group of related variables to be treated as a unit

Structure Declaration (Example)

Declare a structure:

```
struct point {
  int x;
  int y;
};
```

Some variables of that type:

```
struct point here, there;
```

Combination of the upper two:

```
struct point {
  int x;
  int y;
} here, there;
```

Structure Declaration

Keyword `struct` introduces a structure declaration:

```
struct structure tag {
    /* list of member declarations */
    type name;
    type name;
} list of variable declarations;
```

Structure declaration has four parts:

- keyword `struct`
- *structure tag* (optional)
- brace-enclosed list of declarations for the *members* (optional)
- list of variables of the new structure type (optional)

Struct Declaration Defines A Type

- A `struct` declaration defines a type
- Terminating right brace may be followed by a list of variables:

```
struct { ... } x, y, z;
```

- This is syntactically analogous to:

```
int x, y, z;
```

- Each statement declares `x`, `y`, and `z`
  - to be variables of the named *type*
  - and causes space to be set aside for them
- A structure declaration not followed by a list of variables
  - reserves no storage
  - merely describes a template or the shape of a structure

Structure Tag

Tagged structure

- A previous established structure tag can be used subsequently as a shorthand for the part of the declaration in braces:

```
struct point pt; /* Structure tag as a shorthand */
```

Anonymous structure

```
struct {
  int i;
  int j;
} a;
```

Operations On And Initialization Of Structures

Legal operations

- copying it
- assigning to it as a unit
- taking its address with &
- accessing its members

Illegal operation

- Structures may not be compared

Initialization

- A list of constant member values initializes a structure

```
struct point maxpt = { 320, 200 };
```

- An automatic structure may also be initialized
  - by assignment
  - by calling a function returning a struct of apt type

Structures And Functions Example

```
/* makepoint: make a point from x and y components */
struct point
makepoint(int x, int y)
{
  struct point tmp;

  tmp.x = x;
  tmp.y = y;
  return tmp;
}
```

```
struct point p1;
struct point p2;

p1 = makepoint(0,0);
p2 = makepoint(XMAX, YMAX);
```

Structures And Functions

There is nothing special about structures and functions

- Pass/return components separately
- Pass/return entire structure
- Pass/return a pointer to structure

If a large structure is to be passed to a function a pointer may be the better choice (pass by value copies the whole structure).

Structure pointers are just like pointers to ordinary variables:

```
struct point *pp;

*pp = makepoint(1,3); /* *pp is the structure */
(*pp).x += 2;         /* (*pp).x is a member */
```

The Structure Member Operators “.” And “->”

Structure operator “.” connects structure- and member name

A member of a particular structure is referred to in an expression by *structure-name.member*

```
printf("%d,%d", pt.x, pt.y);
```

Structure operator “->” as *shorthand*

If `ps` is a pointer to a structure with member `m`, than

$$\texttt{(*ps).m} \iff \texttt{ps->m}$$

are equivalent by definition.

Nested Structures

Structures can be nested

```
struct rect {
        struct point pt1;
        struct point pt2;
}
```

```
struct rect screen;

screen.pt1.x;
```

- The `rect` structure contains two `point` structures
- `screen.pt1.x` is the `x` coord. of the `pt1` member of `screen`

Dynamic Memory Allocation

Pointers and Addresses

Pointers and Function Arguments

Pointers, Arrays and Address Arithmetic

Character Pointers and C Strings

Basics of Structures

Self-referential Structures

Unions

Enumerations

Typedef

Pointers to Functions

Self-referential Structures: Binary Tree

- Data structure: Binary Tree (to store words lexicographically)
- One *node* per distinct word
- Each node contains:
  - a pointer to the text of the word
  - a count of the number of occurences
  - a pointer to the left child node
  - a pointer to the right child node

This reads in C:

```
struct tnode {
        char *word;
        int count;
        struct tnode *left;
        struct tnode *right;
};
```

Lexicographic Order In Binary Tree

- Each node has either zero, one or two children
- Given a node and its *word*
  - Left subtree: All words are lexicographically less than *word*
  - Right subtree: All words are lexicographically greater than *word*

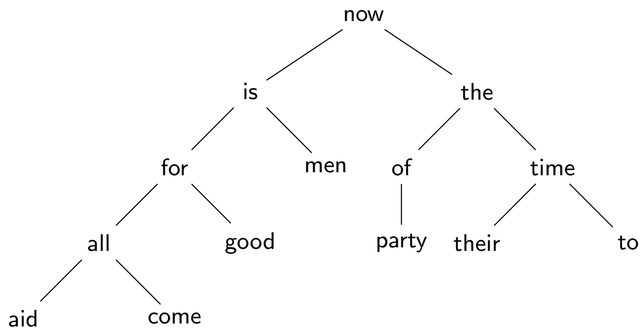
Consider the following input sentence:

now is the time for all good men to come to the aid of their party

Output And Tree View Of bintree.c

now is the time for all good men to come to the aid of their party

```
1 aid
1 all
1 come
1 for
1 good
1 is
1 men
1 now
1 of
1 party
2 the
1 their
1 time
2 to
```



Source Code Binary Tree

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100

struct tnode {
  char *word;
  int count;
  struct tnode *left;
  struct tnode *right;
};

struct tnode *
addtree(struct tnode *, char *);
void treeprint(struct tnode *);
int getword(char *, int);

int
main(void)
{
  struct tnode *root;
  char word[MAXWORD];

  root = NULL;
  while (getword(word, MAXWORD) != EOF)
    if (isalpha(word[0]))
      root = addtree(root, word);
  treeprint(root);
  return (0);
}

struct tnode *talloc(void);
```

```
struct tnode *talloc(void);
char *strdupl(char *);

/* add a node with w, at or below p */
struct tnode *
addtree(struct tnode *p, char *w)
{
  int cond;

  if (p == NULL) { /* new word arrived */
    p = talloc();  /* make a new node */
    p->word = strdupl(w);
    p->count = 1;
    p->left = p->right = NULL;
  } else if ((cond = strcmp(w, p->word)) == 0)
    p->count++; /* repeated word */
  else if (cond < 0)  /* less than -> left */
    p->left = addtree(p->left, w);
  else
    p->right = addtree(p->right, w);
  return (p);
}
```

Source Code Binary Tree

```c
/* treeprint: in-order print of tree p */
void
treeprint(struct tnode *p)
{
  if (p != NULL) {
    treeprint(p->left);
    printf("%4d␣%s\n", p->count, p->word);
    treeprint(p->right);
  }
}

#include <stdlib.h>

/* talloc: make a tree node */
struct tnode*
talloc(void)
{
  struct tnode *tn;
  tn = (struct tnode *)
    malloc(sizeof(struct tnode));
  return tn;
}
```

```c
/* make a duplicate of s */
char *
strdupl(char *s)
{
  char *p;

  p = (char *) malloc(strlen(s) + 1);
  if (p != NULL)
    strcpy(p, s);
  return p;
}

/* get single word from input */
int
getword(char *word, int max)
{
  int c, i;

  i = 0;
  while ((c = getchar()) != EOF && i < max - 1
    && c != '␣' && c != '\n' && c != '\t')
    word[i++] = c;
  word[i] = '\0';

  return (c == EOF) ? EOF : i;
}
```

Dynamic Memory Allocation

Pointers and Addresses

Pointers and Function Arguments

Pointers, Arrays and Address Arithmetic

Character Pointers and C Strings

Basics of Structures

Self-referential Structures

Unions

Enumerations

Typedef

Pointers to Functions

Unions

- A *union* is a variable that may hold (at different times) objects of different types and sizes.
- Unions provide a way to manipulate different kinds of data in a single area of storage.

The syntax is based on structures:

```
union u_tag {
  int ival;
  float fval;
  char *sval;
} u;
```

- The variable `u` will be large enough to hold the largest of the three types (the specific size is implementation-dependent)
- It is the programmer's responsibility to keep track of which type is currently stored in a union

Unions (cont.)

- Unions may occur within structures and arrays, and vice versa
- Notation for accessing a member of a union in a structure (or vice versa) is identical to that for nested structures

```
struct {
  char *name;
  int flags;
  int utype;
  union {
    int ival;
    float fval;
    char *sval;
  } u;
} symtab[NSYM];
```

```
symtab[i].u.ival
*symtab[i].u.sval   /* first char of string sval */
symtab[i].u.sval[0] /* dito */
```

Dynamic Memory Allocation

Pointers and Addresses

Pointers and Function Arguments

Pointers, Arrays and Address Arithmetic

Character Pointers and C Strings

Basics of Structures

Self-referential Structures

Unions

Enumerations

Typedef

Pointers to Functions

Enumerations

- Enumerations provide a conventient way to associate constant values with names
- An alternative to `#define` with the advantage that the values can be generated automatically
- A debugger may also be able to print values of enumeration variables in symbolic form

```
enum boolean { NO, YES };
```

Enumerations (cont.)

- An enumeration is a list of constant integer values
- First value in an `enum` has value 0, the next 1 ...
- ... unless explicit values are specified

```
enum escapes { BELL = '\a',
    BACKSPACE = '\b', TAB = '\t' };
```

- If not all values are specified, unspecified values continue the progression from the last specified value

```
/* FEB is 2, MAR is 3 ... */
enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
              JUL, AUG, SEP, OCT, NOV, DEC };
```

Dynamic Memory Allocation

Pointers and Addresses

Pointers and Function Arguments

Pointers, Arrays and Address Arithmetic

Character Pointers and C Strings

Basics of Structures

Self-referential Structures

Unions

Enumerations

Typedef

Pointers to Functions

Typedef: New Data Type Names

`typedef` is a facility for creating new data type names:

```
typedef int Length;
```

- makes the name `Length` a synonym for `int`
- Type `Length` can be used exactly in the same way as type `int`

Reasons for using `typedefs`

- Portability issues
  - Types like `size_t`, `ptrdiff_t` are examples
- (Better) Documentation for a program
  - A type `Treeptr` may be easier to understand than one declared as a pointer to a complicated structure

Further typedefs

- In effect, `typedef` is like `#define`
- Except that it is interpreted by the compiler
- Therefore its capabilities are beyond textual substitutions

```
typedef int (*PFI)(char *, char *);
```

- Creates the type PFI (*pointer to function (of two* `char *` *argument) returning* `int`

```
typedef enum {ON, OFF, BROKEN} state;
```

- Creates the type state

Dynamic Memory Allocation

Pointers and Addresses

Pointers and Function Arguments

Pointers, Arrays and Address Arithmetic

Character Pointers and C Strings

Basics of Structures

Self-referential Structures

Unions

Enumerations

Typedef

Pointers to Functions

Pointers To Functions

- A function itself is not a variable
- But it is possible to define pointers to functions
  - which can be assigned, placed in arrays
  - passed to functions, returned by functions

```
int (*func)(char *, char *);
```

- Declares a pointer to a function that has two `char *` arguments and returns an `int`
- `func` is a pointer to a function
- `(*func)` is the function, as such the function call reads:

```
(*comp)("abc", "def");
```

Pointer To Functions Example

```
#include<stdio.h>

void
print_one(void)
{
        printf("1\n");
}

void
print_two(void)
{
        printf("2\n");
}

int
main(void)
{
        void (*func[])(void) = { print_one, print_two };

        (*func[0])();
        (*func[1])();
        return (0);
}
```