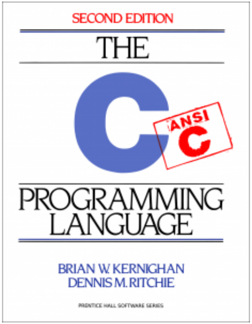
CS 241
Data Organization
Pointers and Arrays

Brooke Chenoweth

University of New Mexico

Fall 2014

Read Kernighan & Richie



6 Structures

Pointers

- A pointer is a variable that contains the address of another variable. A pointer variable is created using the '`*`' character when declaring it:

```
   int *ptr;
```

- In this example `ptr` is a pointer variable that contains the address of a variable of type `int`.
- Declaring a pointer does not allocate storage for the pointer to point to. The above ptr variable initially contains an undefined value (address).

Pointers. . .

- The ‘*’ character is also used to refer to the value pointed to by the pointer (dereference the pointer). Suppose we want to store the value 10 into memory at the address contained in ptr. Do the following:

```
  *ptr = 10;
```

- Similarly, to get the value stored at ptr do the following:

```
  y = *ptr + 5;
```

- The value of `y` is now 15.

Pointers. . .

- How do you set the value of ptr in the first place? Use the '&' character to get the address of a variable, e.g.:

```c
int y;
int *ptr;
ptr = &y;
*ptr = 21;
```

- The value of `y` is now 21.
- What does the following do?

```c
*(&y) = 42;
```

Pointer Example

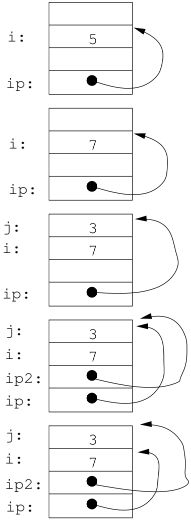
```
int *ip;
int i = 5;
ip = &i;

*ip = 7;

int j = 3;
ip = &j

int *ip2=ip;

ip = &i;
```



Pointer Types

- C cares (sort of) about pointer types; the type of a pointer must match the type of what it points to. The following will cause the C compiler to print a warning:

```
short x;
int   *ptr;
ptr = &x;
```

- I say it “sort of” cares because while it issues a warning, it goes ahead and compiles the (probably incorrect) code.

Pointer Types. . .

- If you know you want to do the above, you can stop the compiler from complaining by casting the pointer, e.g.:

```
short x;
int   *ptr;
ptr = (int *) &x;
```

- Your program may still not do the right thing, however.

Pointer Types. . .

- Casting pointers is a popular thing to do in C, as it provides you with a certain amount of flexibility. The type `void *` is a pointer to nothing at all — it contains a memory address, but there is no type associated with it. You can't do:

```
int  x;
void *ptr = (void *) &x;
*ptr = 10;
```

  for example.

Pointer Types…

- Instead, you have to cast the pointer to the proper type:

```
int      x;
void     *ptr = (void *) &x;
*((int *) ptr) = 10;
```

Pointers

$$\left.\begin{array}{r}\text{Pointer}\\ \text{Address}\\ \text{Reference}\end{array}\right\}\text{A location in memory}$$

```c
void main(void)
{
  int x=6;
  int *y; /* y will be a pointer to an int. */
  y = &x; /* y is assigned the address of x. */
  printf("x=%d, y=%p, *y=%d\n", x, y, *y);
}
```

Output:

```
x=6, y=0x7fff1405a74c, *y=6
```

Overloaded Operators

In the C programming Language, what does '*' mean?

```c
void main(void)
{
  int a = 6;        /* binary: 0110 */
  int b = 3;        /* binary: 0011 */
  int *c = &a;      /* The '&' means address of */
  int x = a*b;      /* The '*' means multiply */
  int y = a + *c;   /* The '*' means dereference */
  int z = a & b;    /* The '&' means bitwise AND */
  printf("%d, %d, %d\n", x, y, z);
}
```

Output:

```
18, 12, 2
```

Swap Error: Pass by Value

```c
void swapNot(int x, int y)
{
  printf("swapNot (1) x=%d, y=%d\n", x,y);
  int tmp = x;
  x = y;
  y = tmp;
  printf("swapNot (2) x=%d, y=%d\n", x,y);
}


void main(void)
{
  int v[] = {33, 44, 55, 66, 77};
  printf("main (1) v[0]=%d, v[1]=%d\n", v[0],v[1]);

  swapNot(v[0], v[1]); /* Passed by Value */
  printf("main (2) v[0]=%d, v[1]=%d\n", v[0],v[1]);
}
```

```
main (1) v[0]=33, v[1]=44
swapNot (1) x=33, y=44
swapNot (2) x=44, y=33
main (2) v[0]=33 v[1]=44
```

Working Swap: By Array Elements

```c
void swapElements(int v[], int i, int k)
{
  int tmp = v[i];
  v[i] = v[k];
  v[k] = tmp;
}
void main(void)
{
  int v[] = {33, 44, 55, 66, 77};
  printf("main (1) v[0]=%d, v[1]=%d\n", v[0], v[1]);

  swapElements(v, 0, 1); /* passes address of v[0] */
  printf("main (4) v[0]=%d, v[1]=%d\n", v[0], v[1]);
}
```

```
main (1) v[0]=33, v[1]=44
main (4) v[0]=44, v[1]=33
```

Working Swap: Using pointers

```c
void swap (int *x, int *y)
{
  int tmp = *x; /* tmp assigned value at address x. */
  *x = *y; /* val at addr x assigned val at addr y. */
  *y = tmp;
}

void main(void)
{
  int v[] = {33, 44, 55, 66, 77};
  printf("main (1) v[0]=%d, v[1]=%d\n", v[0],v[1]);

  swap(&v[0], &v[1]); //Passed by Reference
  printf("main (3) v[0]=%d, v[1]=%d\n", v[0],v[1]);
}
```

```
main (1) v[0]=33, v[1]=44
main (3) v[0]=44, v[1]=33
```

Array argument is a pointer

```
void swapElements (int v[], int i, int k)
/*       same as: (int* v,  int i, int k) */
/*       same as: (int *v,  int i, int k) */
/*       same as: (int*v,   int i, int k) */
```

- Before, we have said that array arguments are passed by reference.
- It would be more accurate to say that the address of the array (a pointer) is passed by value.

Pointer Declaration Style

```
1  void main(void)
2  {
3    int* a, b;
4    *a = 5;
5    b = 7;
6    printf("%d, %d\n", *a, b);
7  }
```

Output:

```
5, 7
```

Line 3 is bad style: a is a pointer; b is an int. Should use one of:

```
int *a, b;
```

```
int* a;
int b;
```

```
int *a;
int b;
```

Pointer Arithmetic

- You can perform arithmetic on pointers:

```
ptr2 = ptr1 + 3;
```

- Pointer arithmetic is type-specific: the value of `ptr+x` is equal to `ptr` plus `x` multiplied by the size of the type to which `ptr` points.
- Said another way, the result of `ptr+x` if `ptr` points to type `type` is

```
((int) ptr) + x * sizeof(type).
```

Pointer Arithmetic. . .

The resulting address depends on the type:

```
ptr = 100;
ptr = ptr + 3;
```

Type of ptr	Result
`char *`	103
`short *`	106
`int *`	112
`int **`	112
`struct foo *`	100 + 3*sizeof(struct foo)

Arrays

- C offers a convenient short-hand for pointer arithmetic using square- braces `[]`. The notation

  ```
  ptr[x]
  ```

  is equivalent to

  ```
  *(ptr+x)
  ```

Arrays. . .

- Addresses can be taken of individual array elements:

  ```
  &array[1]
  ```

  is the address of the 2nd element in the array.

  ```
  &array[0]
  ```

  is the same address as `array`.
- Arrays of pointers are also possible:

  ```
  int *array[3];
  ```

- This allocates an array of three pointers to integers, not three integers. On a 32 bit system, they are often the same, but on a 64 bit system they might be different.

Initializing Arrays

- You already know how to initialize an array of ints

```c
int vals[] = {10, 17, 42};
```

- Similarly, you can initialize an array of pointers.

```c
char *colors[] = {"red", "green", "blue"};
```

Multi-dimensional Arrays

- Multi-dimensional arrays are arrays of arrays:

```
int matrix[10][5];
```

  `matrix` is an array of 10 arrays, each containing 5 elements. The array is organized in memory so that `matrix[0][1]` is adjacent to `matrix[0][0]`.

- A multi-dimensional array can be initialized:

```
int x[2][3] = {
  {0, 1, 2},
  {3, 4, 5}
};
```

Multi-dimensional Arrays. . .

- When passing a multi-dimensional array as a parameter all but the first dimension must be specified so the correct address calculation code is generated:

```
void foo ( int x[][3] );
```

- Arrays of pointers to arrays are often used instead of multi-dimensional arrays:

```
int *foo[2];
```

is an array of two pointers to integers.

Arrays of pointers

- We can then create two sub-arrays, possibly of different size, and index them like a multi-dimensional array:

```
int *foo[2]; /* Array of two integer pointers *
int a[3];    /* Array of three ints */
int b[4];    /* Array of four ints */
foo[0] = a;
foo[1] = b;
foo[0][0] = 0;   /* a[0] = 0; */
foo[0][1] = 1;   /* a[1] = 1; */
foo[1][3] = 3;   /* b[3] = 3; */
foo[0][3] = 3;   /* ERROR! */
```

Multi-dimensional Arrays. . .

- These arrays of pointers to arrays are especially useful for arrays of strings:

```
char *colors[3] = {"red", "green", "blue"};
```

`colors` is an array of pointers to arrays of characters, each a different size:

```
colors[0][0] == ’r’
colors[1][0] == ’g’
colors[2][0] == ’b’
```

Pointer to String Constant

```c
 1 #include <stdio.h>
 2 void main(void)
 3 {
 4   char str1[] = "Hello World";
 5   char *str2 = "Hello World";
 6   str1[6] = 'X';
 7   printf("str1=%s\n", str1);
 8   printf("str2=%s\n", str2);
 9   str2[6] = 'X';
10   printf("str2=%s\n", str2);
11 }
```

```
str1=Hello Xorld
str2=Hello World
Segmentation fault
```

Line 9 fails because `str2` is in read-only memory.

Address Arithmetic

```c
 1 int n=17;
 2 int* a = &n;
 3 short* b = (short*)&n;
 4 char* c = (char*)&n;
 5
 6 printf("%p %p %p\n", a, b, c);
 7 a++; b++; c++;
 8
 9 printf("%p %p %p\n", a, b, c);
10 printf("%d\n", n);
```

Line 7: The values at `*a`, `*b` and `*c` are undefined and may segfault.

```
0x7ffffba2610c 0x7ffffba2610c 0x7ffffba2610c
0x7ffffba26110 0x7ffffba2610e 0x7ffffba2610d
17
```

String Length by Index & Address Arithmetic

```
int strLen(char s[])
{
  int i=0;
  while (s[i]) i++;
  return i;
}
```

s[i]: Machine Code

```
get s
get i
add
get *topofstack
```

```
int strLen2(char *s)
{
  char *p = s;
  while (*p) p++;
  return p - s;
}
```

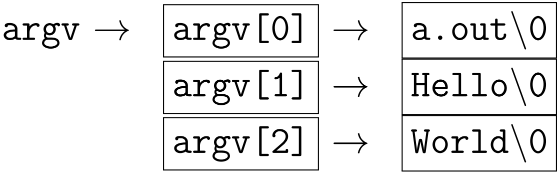
*p: Machine Code

```
get p
get *topofstack
```

Command Line Arguments

```
int main(int argc, char *argv[])
{
```

Call program: `a.out Hello World`



argv → argv[0] → a.out\0

argv[1] → Hello\0

argv[2] → World\0

`argv` is a pointer to an array of pointers.
Each pointer in the array is the address of the first `char` in a null terminated string.

Echo Arguments: Array Style

```
void main(int argc, char *argv[])
{
  int i;
  printf("Number of arguments = %d\n", argc);
  for (i=0; i<argc; i++)
  {
    printf("   argv[%d]=%s\n", i, argv[i]);
  }
}
```

```
a.out pi is 3.1415
```

```
Number of arguments = 4
   argv[0]=a.out
   argv[1]=pi
   argv[2]=is
   argv[3]=3.1415
```

`argv[i]` is address of a null terminated string.

Echo Arguments: Pointer Style

```
void main(int argc, char *argv[])
{ printf("main(): argc=%d\n", argc);
  while (argc-- > 0) /* test first, then decrement */
  {
    printf("argc=%d: %s\n", argc, *argv++);
  }
}
```

What is going on in `*argv++`?

1. Dereference `argv`. This is `argv[0]`: a pointer to the first argument.
2. Send that pointer to `%s`.
3. Increment `argv` (not `*argv`). Now `argv` points to what was originally `argv[1]`.

```
a.out Hello World
```

```
main(): argc=3
argc=2: a.out
argc=1: Hello
argc=0: World
```

What, in the name of Dennis Ritchie, is *argv++

```c
1 void main(int argc, char *argv[])
2 {
3   printf("%p: %p->%s\n", argv, *argv, *argv);
4   argv++;
5   printf("%p: %p->%s\n", argv, *argv, *argv);
6 }
```

Changing line 4 to `*argv++` has no effect! Why?

```
a.out Hello World
```

```
0x7fff34de98e0: 0x7fff34dead40->a.out
0x7fff34de98e8: 0x7fff34dead46->Hello
```

Why is address of 'a' 6 less than address of 'H'?

Double Echo Arguments: Array Style

```
./a.out Hello World
```

```
#include <stdio.h>
void main(int argc, char *argv[])
{ int i, k;
  char* str;
  printf("Number of arguments = %d\n", argc);
  for (i=0; i<argc; i++)
  { printf("argv[%d]=%s\n", i, argv[i]);
    k=0;
    str = argv[i];
    while (str[k])
    { printf(" %c ",str[k]);
      k++;
    }
    printf("\n");
  }
}
```

```
Number of arguments = 3
argv[0]=./a.out
 .  /  a  .  o  u  t 
argv[1]=Hello
 H  e  l  l  o 
argv[2]=World
 W  o  r  l  d 
```

charCmpCaseInsensitive()

```c
int charCmpCaseInsensitive(char c1, char c2)
{
  int lowerCaseOffset = ’A’ - ’a’;
  if (c1 >= ’a’ && c1 <= ’z’)
  {
    c1 += lowerCaseOffset;
  }
  if (c2 >= ’a’ && c2 <= ’z’)
  {
    c2 += lowerCaseOffset;
  }
  return c1==c2;
}
```

findSubstringCaseInsensitive()

```
char *findSubstringCaseInsensitive(char *haystack,
                                   char *needle)
{ int len = strlen(needle);
  int matchCount = 0;
  while (*haystack)
  { if ( charCmpCaseInsensitive(
           *(needle+matchCount), *haystack))
    { matchCount++;
      if (matchCount == len)
      {
        return (haystack - len)+1;
      }
    }
    else {haystack -= matchCount; matchCount = 0;}
    haystack++;
  }
  return NULL;
}
```

Redone with Single Exit Code Style

```
char *findSubstring(char *haystack, char *needle)
{ int len = strlen(needle);
  int matchCount = 0, done = 0;
  char *startPt = NULL;
  while (*haystack && !done)
  { if ( charCmpCaseInsensitive(
           *(needle+matchCount), *haystack))
    { matchCount++;
      if (matchCount == len)
      { startPt = (haystack - len)+1;
        done = 1;
      }
    }
    else {haystack -= matchCount; matchCount = 0;}
    haystack++;
  }
  return startPt;
}
```

scanf(. . .): read from stdin

```c
#include <stdio.h>

void main(void)
{ int n, m, a;
  float x;
  scanf("%d %d %f %d", &n, &m, &x, &a);

  printf("%d %d %f %d\n", n, m, x, a);
}
```

Input:

```
2 49 3.1415
128
```

Output:

```
2 49 3.141500 128
```

See Kernighan & Ritchie, 7.4 Formatted Input

sscanf(. . .): read from a string

```c
void main(void)
{
  char sentence[] = "Rudolph is 12 years";
  char s1[20], s2[20];
  int i;

  sscanf (sentence,"%s %s %d",s1,s2,&i);
  printf ("[%s] [%s] [%d]\n",s1,s2,i);
}
```

Output:

```
[Rudolph] [is] [12]
```

DANGER! `scanf("%s",str);`

```c
char str[256];
scanf("%s", str);
printf("%s\n", str);
```

- There is only one thing that really need to be said about using `scanf(...)` or `gets(char *str)` to read a character string: Do not do it.
- Both have the exact same problem with memory overrun: You can easily read in more characters than your `char*` can hold.

fgets: Get a String From a Stream

SYNOPSIS

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
```

DESCRIPTION

The fgets() function shall read bytes from stream into the array pointed to by s, until n-1 bytes are read, or a <newline> is read and transferred to s, or an end-of-file condition is encountered. The string is then terminated with a null byte.

strtol: Convert String to Long

SYNOPSIS

```
#include <stdlib.h>
long strtol(const char *nptr, char **endptr, int base
```

DESCRIPTION

- The strtol() function converts the string pointed to by nptr to a long int representation.
- The first unrecognized character ends the string. A pointer to this unrecognized character is stored in the object addressed by endptr
- If base ([0, 36]) is non-zero, its value determines the set of recognized digits.

strtol: Example

```c
#include <stdio.h>
#include <stdlib.h>
void main(void)
{ char *endPtr;
  long n = strtol("1001", &endPtr, 2);
  printf("n=%ld, char at endPtr=[%c]\n", n, *endPtr);
  n = strtol("1011a", &endPtr, 2);
  printf("n=%ld, char at endPtr=[%c]\n", n, *endPtr);
}
```

Output:

```
n=9, char at endPtr=[]
n=11, char at endPtr=[a]
```

Pointers have Tremendous Power, But. . .

- Pointers, if used incorrectly, lead to very difficult to find bugs: bugs that only sometimes manifest:
  - When you write to an ill-defined memory location it may often be that the location is unused.
  - On such occasions your program will run just fine, without complaint ☺
  - Perhaps one day one of your arrays has more data than usual. . . Perhaps on that day the overwritten memory contains critical data ☹
- Code that uses pointers is often harder for humans to read.
- Code that uses pointers is much harder for compilers to optimize (especially vector and parallel optimizations).