Introduction to Pointers

A pointer is a *variable* whose value is *the address of another variable*.

The size of the pointer variable must be *n* bits where 2^n bytes is the size of the address space.
For the Intel x86 and SPARC systems, address space size is 4GB and we have *n=32*.

The amount of space required to hold a pointer variable is *always 4* bytes on these hardware platforms and is *not related* to the size of the entity to which it points.

To declare a pointer in a program just use *the type it points to followed by* *.

```
int            *a;
short          *b;
unsigned char  *c;
```

To refer to the value of the pointer itself just *use the variable name*:

```
a = &x;      // assign the address of x to pointer a
```

To refer to the entity to which the pointer points *prepend* *

```
*a = 132;  // assign the value 132 to x (since a now points to x)
```

Initialization of pointers

Like all variables pointers must be initialized before they are used.

```
/* p17.c */

/* Example of a common error: failure to intialize */
/* a pointer before using it.. This program is     */
/* is FATALLY flawed....                           */

main()
{
   int* ptr;

   *ptr = 99;
```


```
    printf("val of *ptr = %d and ptr is %p \n", *ptr, ptr);
}
```

But unfortunately, on Linux this program appears to work!

```
class/215/examples ==> p17
val of *ptr = 99 and ptr is 0x40015360
```

The program *appears* to work because the address 0x40015360 just happens to be a legal address in the address space of this program. Unfortunately, it *may* be the address of *some other variable* whose value is now 99!!!

This situation is commonly referred to as a *loose pointer* and bugs such as these may be *very* hard to find.

We can convert the bug from latent by active by changing the location of the variable *ptr*.

Here we move it down the stack by declaring an array of size 200.

```
class/215/examples ==> cat p18.c
/* p17.c */

/* Example of a common error: failure to intialize */
/* a pointer before using it                       */

main()
{
   int  a[200];  // force ptr to live down in uninit turf
   int* ptr;

   printf("val of ptr is %p \n", ptr);

   *ptr = 99;

   printf("val of *ptr = %d and ptr is %p \n", *ptr, ptr);
}

class/215/examples ==> p18
val of ptr is (nil)
class/215/examples ==>
```

Note that in this case the *second printf() is not reached* because the program *segfaulted* and died when it illegally attempted to assign the value 99 to memory location 0 (*nil).*

Minimizing latent loose pointer problems

Never *declare* a pointer without *intializing* it in the declaration.

```
     int *ptr = NULL;
```

Never us NULL as synonym for integer or float 0.

Using *gdb* to find the point of failure:

The *gdb* debugger is a handy tool for identifying the location at which a program failed.

To use the debugger it is necessary to compile with the -g option.

```
class/215/examples ==> gcc -g -o p18 p18.c
```

To start the debugger use the *gdb* command and specify the program name

```
class/215/examples ==> gdb p18
```

At the *(gdb)* prompt you will usually want to tell the debugger to halt the program when it reaches the start of the *main()* function. The *b* command is short for *breakpoint* and tells the debugger where to stop. After a function is entered source code line numbers can be used to specify breakpoints.

```
(gdb) b main
Breakpoint 1 at 0x804833b: file p18.c, line 11.
```

To start the program use the *run* command:

```
(gdb) run
Starting program: /local/westall/class/215/examples/p18
```

When the program reaches a breakpoint *gdb* will tell you and display the *next* line of code to be executed.

```
Breakpoint 1, main () at p18.c:11
11          printf("val of ptr is %p \n", ptr);
```

Use the *next* command to execute a single source statement. The *next* command will treat a function call as a single statement and not single step into the function being called. If you want to single step through the function use the *step* command to step into it. The output of the *printf()* is intermixed with *gdb's* output and is shown in blue below.

```
(gdb) next
val of ptr is (nil)
13          *ptr = 99;
```

Saying *next* again will cause the program to execute the flawed assignment. *gdb* will show you the line that caused the error (line # 13).

```
(gdb) next

Program received signal SIGSEGV, Segmentation fault.
0x08048357 in main () at p18.c:13
13          *ptr = 99;
```

The where command can show you where the failure occured (along with a complete function activation trace.

```
(gdb) where
#0  0x08048357 in main () at p18.c:13
#1  0x40049917 in __libc_start_main () from /lib/libc.so.6
(gdb)
```

To print the value of a variable use the *print* command.

```
(gdb) print ptr
$1 = (int *) 0x0
```

Attempting to print what *ptr* points to reaffirms what the problem is:

```
(gdb) print *ptr
Cannot access memory at address 0x0
```

Use the *q (quit)* command to terminate *gdb*

```
(gdb) quit
The program is running.  Exit anyway? (y or n) y
class/215/examples ==>
```

Correct use of the pointer

In the C language, variables that are declared within any basic block are allocated on the run-time stack at the time the basic block is activated.

```
/* p19.c */

main()                Addresses of y and ptr
{
   int  y;  <-------------  0xbffff894
   int* ptr;<-------------  0xbffff890
   static int a;

   ptr = &y;   // assign the address of y to the pointer


   *ptr = 99;  // assign 99 to what the pointer points to (y)

   printf("y = %d ptr = %p addr of ptr = %p \n", y, ptr, &ptr);

   ptr = &a;

   printf("The address of a is %p \n", ptr);
}
```

```
class/215/examples ==> p17
y = 99 ptr = 0xbffff894 addr of ptr = 0xbffff890
The address of a is 0x804958c
```

Note that *a* is heap resident and has an address far removed from the stack resident *y* and *ptr*.

Use of pointers in processing C-strings.

Recall that a "C-string" is an array of characters whose logical end is denoted by a *zero valued byte.*
The C standard library has a number of functions designed to work with C strings.
The *strtol()* function is one of them. You can see most of them on a Solaris system via the command:

```
man -s 3C string
```

```
     string, strcasecmp, strncasecmp, strcat,  strncat,  strlcat,
     strchr,  strrchr, strcmp, strncmp, strcpy, strncpy, strlcpy,
     strcspn, strspn, strdup, strlen,  strpbrk,  strstr,  strtok,
     strtok_r - string operations
```

In this example we will see how *strcat* might be implemented. Its prototype is shown below.

```
char *strcat(char *s1, const char *s2);
```

An implementation of *strcat*

The name *zstrcat* is used to avoid potential nastygrams and name conflicts with the "real" *strcat*.

```
/* p12b.c */

#include <stdio.h>
#include <error.h>

/* The mission of this function is to catenate */
/* the string pointed to by p2 to the end of   */
/* the string pointed to by p1                 */

char *zstrcat(
char *p1,          /* string to be extended */
char *p2)          /* string to be appended */
{
   char *src;     /* source pointer      */
   char *dst;     /* destination pointer */

   dst = p1;
   src = p2;

/* Start by advancing dest to the end */
/* of the string to be extended       */

   while (*dst != 0)
      dst = dst + 1;

/* Now tack on the string to be appended */

   while (*src != 0)
   {
      *dst = *src;
      dst = dst + 1;
      src = src + 1;
   }

/* Complete the job by NULL terminating the new string */

   *dst = 0;
   return(p1);
}
```

An alternative implementation

It is possible to shrink and obfuscate the code in an attempt to demonstrate ones C language *machismo*. This approach produces code that is difficult to read, painful to maintain, but may *(or may not)* produce a trivial improvement in performance . When tempted, *just say no!*

```
char *zstrcat(
char *p1,
char *p2)
{
   char *r = p1;
   while (*p1++);
   p1--;
   while (*p1++ = *p2++);
   return(r);
}

main()
{
    char *s1;
    char *s2;
    char *result;

    s1 = (char *)malloc(40);
    s2 = (char *)malloc(40);
    result = (char *)malloc(81);
    *result = 0;

    fgets(s1, 40, stdin);
    fgets(s2, 40, stdin);

    zstrcat(result, s1);
    zstrcat(result, " ");
    zstrcat(result, s2);

    fputs(result, stdout);
}
class/215/examples ==> p12b
Hello
World
Hello
World
```

Using a pointer to process an array of numbers

This program demonstrates

(1) how to process command line parameters and
(2) how to process an array of *ints* using a pointer.

```
/* p24.c */

/* This program illustrates the use of pointers in */
/* reading and processing an array of integers     */
/* It also shows how to access command line args   */

/* An upper bound on the number of ints to be read */
/* must be specified on the command line..         */
/*    p2 1000                                      */

#include <stdio.h>

int main(
int argc,       /* number of command line args */
char* argv[])   /* array of pointers to  args   */
{
   int *base;     // points to start of the array
   int *loc;      // current array location
    int  max;     // maximum number of values to read in
    int  count;   // actual number of values read in
    int  largest; // largest number in the array
    int  i;
```

Processing the command line argument with *atoi()*

The *argc* parameter contains then number of command line parameters *including* the program name itself. Thus a command such as

```
a.out 300
```

will cause *argc* to be set to 2. It's very important to ensure that the user has provided the number of parameters you need *before* you attempt to process them!

```
if (argc < 2)   /* Make sure at least one arg was given */
{
   printf("Usage is p20 upper-bound \n");
   exit(1);
}
```

This program expects that a numeric value representing the maximum number of values that are present in the standard input will be present on the command line. We have seen that we can capture this parameter with *sscanf(argv[1], "%d", &max);*

We can also do it with the *atoi()* function. This function expects to be passed a pointer to the ASCII string encoding of an integer value. It will convert the string to a binary integer and return the value. The downside of *atoi()* is that its *behavior in the presence of an error is undefined.*

Therfore its important to intialize the variable to some *illegal* value before calling *atoi()* and then checking to see if the value becomes legal.

```
/* The pointer argv[0] points to the name of the program (p20) */
/* argv[1] points to the first command line argument           */
/* The atoi() function converts the ascii character            */
/* representation an integer to a binary int value.            */

   max = 0;
   max = atoi(argv[1]);

   if (max <= 0)
   {
      printf("upper-bound must be a positive integer \n");
      exit(2);
   }
```

Allocating storage for the array of *ints*.

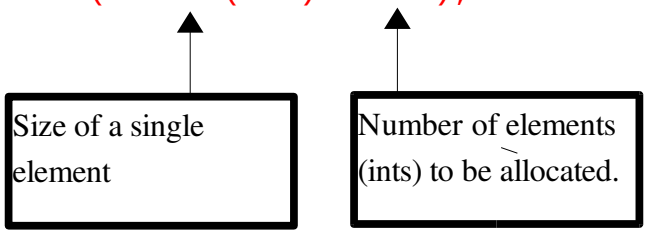
- The *malloc()* (memory allocate) function is can be used to dynamically allocate an area of memory to be used as an informal array.

- The parameter passed to *malloc()* is the size of the area to be allocated *in bytes.*

- Therefore, if we want to allocate space for 100 *ints,* we must pass 400 to *malloc()* because each *int* occupies 4 bytes.

- The *sizeof()* facility should *always* be used instead of coding the constant 4.

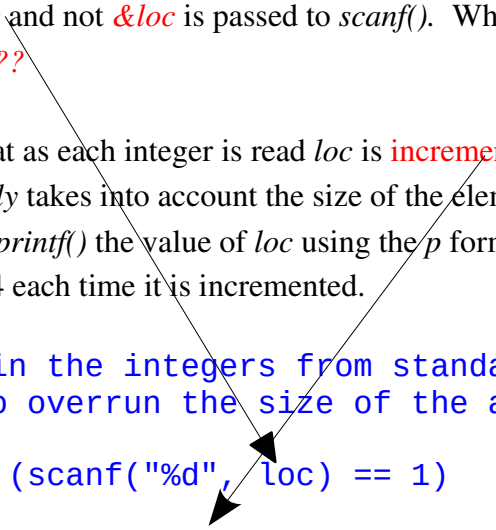
```
count = 0;
base = (int *)malloc(sizeof(int) * max);
loc = base;
```



Reading the input values

Note that *loc* and not *&loc* is passed to *scanf()*. What would happen if a programmer "accidentally" passed *&loc??*

Also note that as each integer is read *loc* is incremented by 1 and not by 4. The C language *automagically* takes into account the size of the element pointed to when doing pointer arithmetic! If you were to *printf()* the value of *loc* using the *p* format code, you would see that the actual value *does* increase by 4 each time it is incremented.



```
/* Read in the integers from standard input making sure */
/* not to overrun the size of the array                 */

   while (scanf("%d", loc) == 1)
   {
      loc = loc + 1;
      count = count + 1;
      if (count == max)
         break;
   }
```

Identifying the largest value in the array

Before starting the search for the largest number the value of *loc* is reset to point to the *base* of the array.

```
    loc = base;     // point loc back to the start of the array
    largest = *loc; // init largest to the first value in the array
    loc = loc + 1;

    for (i = i; i < count; i++)
    {
       if (*loc > largest)
          *loc = largest;

       loc = loc + 1;
    }
    printf("Largest was %d \n", largest);
}
```

Exercise: This program actually has a couple of nasty bugs in it. Use *gdb* to find and fix them.

Other ways to consume command line parameters

The standard runtime library of functions that is normally distributed with C compilers provides a variety of ways to consume command line parameters. In someways they are better than *atoi()* because they better indicate that the user entered incorrect data. Nevertheless, *atoi()* is probably the most widely used.

The *sscanf()* function

As we saw earlier, the *sscanf()* function may be used to attempt to convert ASCII strings in a memory resident buffer to a numeric value. Since *argv[1]* is a pointer to a memory resident buffer containing the string entered as parameter 1 we could replace the *atoi()* call in the previous example by:

```
code = sccanf(argv[1], "%d", &max);
if (code != 1)
{
     fprintf(stderr, "Yeow! bad string in parm 1 \n");
}
```

Since *sscanf()* returns the number of values it converted, the variable *code* will be 1 if it was successful.

The *strttol()* function

The *strotol()* function is more powerful still. It will fill in a pointer to the first illegal character it encounters in the string. If it was successful in producing a valid value, *badchar* will point to the NULL character that terminates the string.

```
char *badchar = NULL;
long max = 0;

max = strtol(argv[1], &badchar, 10)
if (*badchar != 0)
{
     fprintf(stderr, "Yeow! bad character %c in value\n",
          *badchar);
}
```

Using a single dimension array to represent 2 - D data

Suppose the integer variables *numrows* and *numcols* represent the number of rows and columns in an image and that they have been correctly set using information in the *.ppm* header.

Grayscale images

Then space for a *grayscale image* encoded in binary can be allocated by:

```
unsigned char* imageloc;
imageloc = (unsigned char *)malloc(numrows * numcols);
```

To read in the grayscale image from the standard input:

```
pixcount = fread(imageloc, 1, numrows * numcols, stdin);
if (pixcount != numrows * numcols)
{
   fprintf(stderr, "pix count err - wanted %d got %d \n",
                numrows * numcols, pixcount);
   exit(1);
}
```

Accessing a specific element in malloc'd two dimensional data

The value of any grayscale pixel at location *(row, col)* within the image is accessed in the following way:

```
pix = *(imageloc + row * numcols + col)
```

or equivalently

```
pix = imageloc[row * numcols + col]
```

For example, if the value of *numcols* is 10, then there are 10 pixels per image row. To reach the pixel whose (row, column) address is *(3, 5)* it is necessary to pass over three complete rows (row 0, row 1, and row 2) and 5 pixels in row three (pixels 0, 1, 2, 3, and 4).

Thus, the offset of the pixel at *(3, 5)* is *3 * 10 + 5* as shown above.

Processing an image one row at a time

In this example we print pixel values of an entire image with one line of output per row of pixel data:

```
for (row = 0; row < numrows; row = row + 1)
{
   unsigned char* loc;
   loc = imageloc + row * numcols; // first pix in row
   printf("\n");
   for (col = 0; col < numcols; col = col + 1)
   {
      printf("%03x", *loc);
      loc = loc + 1;
   }
}
```

An equivalent formulation that doesn't use pointer notation at all is:

```
for (row = 0; row < numrows; row = row + 1)
{
   printf("\n");
   for (col = 0; col < numcols; col = col + 1)
   {
      printf("%03x", imageloc[row * numcols + col]);
   }
}
```

Color images

A *color image* is often called an *rgb image* because the *red, green,* and *blue* intensities of each pixel are stored together. Space for a color image in binary *rgb* format can be allocated by:

```
unsigned char* imageloc;
imageloc = (unsigned char *)malloc(3 * numrows * numcols);
```

To read in the *rgb* image:

```
pixcount = fread(imageloc, 3, numrows * numcols, stdin);
if (pixcount != numrows * numcols)
{
   fprintf(stderr, "pix count err - wanted %d got %d \n",
               numrows * numcols, pixcount);
   exit(1);
}
```

Accessing the individual pixels of the binary *rgb* image

Here the process is slightly grubbier because each pixel is represented by three constituent components *(r, g, and b)* where each of *(r, g, and b)* are represented by a single unsigned char.

Nevertheless a bit of reflection yields:

```
red   = *(imageloc + 3 * row * numcols + 3 * col);
green = *(imageloc + 3 * row * numcols + 3 * col + 1);
blue  = *(imageloc + 3 * row * numcols + 3 * col + 2);
```

or equivalently

```
red   = imageloc[3 * row * numcols + 3 * col];
green = imageloc[3 * row * numcols + 3 * col + 1];
blue  = imageloc[3 * row * numcols + 3 * col + 2];
```

It is necessary to muliply by 3 because each pixel occupies three bytes of memory. Since our pointer is a pointer to type *unsigned char* which is a single byte, there is no magic available from the compiler to help us out here.

Functions that modify variables of their callers

We saw earlier that when a function tries to modify a parameter that is passed to it, it has no effect on the *callers* copy of that variable.

A caller *can* allow a function to modify its variables by passing a pointer to the value:

```
int getdims(
int argc,
char *argv[],
int *numcols,
int *numrows)
{
   *numcols = atoi(argv[1]);
   *numrows = atoi(argv[2]);
   return(0);
}

int main(
int argc,
char *argv[])
{
   int rows;
   int cols;

   getdims(argc, argv, &cols, &rows);
   fprintf(stdout, "rows = %d cols = %d \n",
                    rows, cols);
}
```

An alternative formulation:

```
int getdims(
int argc,
char *argv[],
int *numcols,
int *numrows)
{
   sscanf(argv[1], "%d", numcols);
   sscanf(argv[2], "%d", numrows);
   return(0);
}
```

A function that swaps two integer values

```
int swap(
int *x,
int *y)
{
   int temp;
   temp = *x;
   *x = *y;
   *y = temp;
}
```

A caller of this function would operate as follows.

```
ndx = 0;
while (ndx < (count – 1))
{
   if (a[ndx] < a[ndx + 1])
     swap(&a[ndx], &a[ndx + 1]);
   ndx = ndx + 1;
}
```

An alternative notation

If *a* is an array or a pointer the names *&a[ndx]* and *a + ndx* are equivalent. Therefore the above code may be written in pointer notation as:

```
ndx = 0;
while (ndx < (count – 1))
{
   if (*(a + ndx) < *(a + ndx + 1))
     swap(a + ndx, a + ndx + 1]);
   ndx = ndx + 1;
}
```

A fully pointerized version

The most efficient way to construct a loop of this type is something like:

```
int *loc = a;
while (ndx < (count – 1))
{
   if (*(loc) < *(loc + 1))
     swap(loc, loc + 1);
   loc = loc + 1;
}
```