Compiling with C

- cc [-g] [-o file] [-DNAME[=VAL] ...] [-IDIR ...] file.c ... [-lx]
  - [ ] - Brackets not part of command
  - [ ] - what is inside is optional
  - ... - last item repeats as many times as you want
  - -g - add debugging information for gdb
  - -o - name executable "file" rather than "a.out"
  - -DNAME - define cpp symbol NAME
  - -IDIR - look for <file.h> includes in DIR
  - -lx -- link with libx.a (x names like "m", "bsd", "curses",...)

C language topics

- "stdio" for I/O
  - #include <stdio.h>
  - printf, fgets, ...
  - printf ("%d", intval)
  - printf ("%s", pointer_to_a_char)
  - See Chapter 5 in text (will cover later in class)

C language topics (page 2)

- Uses C preprocessor
  - #include -- read .h (header) file
    - #include <system.h> -- system files
    - #include "user.h" -- local directory/user includes
  - #define NAME value   // define constants the old school way
  - #define MAX(a,b) ((a) < (b) ? (b) : (a))

  - #ifdef NAME   // conditional compilation
  - #else
  - #endif

- Modifiers
  - const int a = 3; // define constants the new way
  - void func (const char *param); // Won’t change contents
  - static -- limites visibility to file or function

Pointers in C

A Primary (The Primary) datatype in C

- Pointer -- Address of some place in memory
- Similar to "access all" in ada. Java has pointers, but not like C.
- C pointers can point to
  - global variables
  - local variables
  - dynamically allocated memory
  - functions

- Pointers and Arrays ... Same concept in C
  - Arrays use pointers
  - Not all pointers are arrays

- Dynamic Memory Allocation
  - malloc(3), free(3) (C standard library)

Simple pointers

Declaration:

```
int *iptr;  char *cptr;  double *dptr; ...
```

Dereferencing operator: *

```
*iptr = 3;  //  put a 3 in the location pointed to by iptr
printf ("%d", *iptr);  // print the value of the location pointed to ...
```

Address operator: &

```
int x;  int  *xptr = &x;
x = 3;
*xptr = 4;
printf ("%d", x); // Prints 3 or 4?
     prints 4 .... last value assigned to x via xptr
```

pointer to nowhere! NULL (defined in stdio.h and others)

```
#define NULL  (void *)0   // typical definition, don’t define it
```

Structures and simple pointers

```
struct rec { int a; float b};
struct rec* ptr;  // pointer to a struct (record)
(*ptr).a   //  follow pointer, select field named a
ptr->a     //  syntatic sugar for above
```

Pointers as parameters

- Declaration

  void setit (int *pm) { *pm = 3 }

- pointer is passed by value
- may still dereference pointer
- Call to setit:

  int myvar;

  ... setit (&myvar)

- Typical error:

  int *ptr;

  ... setit (ptr);

  - ptr is unitialized, may point to NULL or be garbage

- Very common method of data return from system calls:

  int status;

  ... wait(&status) .....

Dynamic Memory allocation

malloc(3), free(3), sizeof() built-in function

```
// Example program ... not much point of doing this for real.
#include <stdlib.h> ...
int *px;
px = (int *) malloc (sizeof(int));
*px = 3;
printf ("pointer %d points to %d\n", (int)px, *px);
free(px);
```

- See dynmem.c
- sizeof(typename) -- number of bytes used for that type
- Generic pattern of allocating with malloc(3)

```
typename *ptrvar;
// single element
ptrvar = (typename *) malloc (sizeof(typename));
// array of typename
ptrvar = (typename *) malloc (sizeof(typename)*number_of_elements);
// error checking
if (ptrvar == NULL) { // handle error case ..... }
```

More pointer work

Type declarations

```
typedef int * IntPtr;
```

type is "int *"

Typical misunderstanding:

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

=> a is type "int *", b is type int.

often written as: `int  *a, b;`

better way: `int *a;  int b;`

Arrays and Pointers!

Arrays and pointers are very similar concepts in C ..

- int ary[100];
- ary -- constant pointer to start of array.
- type is int *!

```
int ary[100];
ary[0] = 5; ary[1] = 3; ary[2] = 10;  // Normal indexing

int *pa = ary; //  pa points to first element of array!
pa can also be indexed!
pa[0] = 5; pa[1] = 3; pa[2] = 10;

example ptrarray.c
```

Pointer access vs indexing

```
int data[100];
int *dp;
int ix;

for ( dp=data, ix=0; ix < 100; dp++, ix++ )  *dp = ix;
```

- Comma operator: expr1 , expr2 , expr3 ;
- All three evaluated, value of list is expr3 (last one)
- dp++ -- change dp to point to next integer

```
for ( dp=data, ix=0; ix < 100; ix++ )  *(dp+ix) = ix;
```

- dp+ix -- point to the ix’th int in the array
- no change to dp

```
for ( dp=data, ix=0; ix < 100; ix++ )  dp[ix] = ix;
```

- dp[ix] IS identical to *(dp+ix)

Pointer access vs indexing (page 2)

```
int data[100];
// As pointers
for ( ix=0; ix < 100; ix++ )  *(data+ix) = ix;

// As an array
for ( ix=0; ix < 100; ix++ )  data[ix] = ix;
```

Parameters ...

The following two are identical in concept and implementation!

- int set2zero (int data[], int size);

- int set2zero (int *data, int size);

Array parameters ... just call by value pointers!

char arrays and pointers

```
char charary[1024];  // array of characters
char *cptr;          // a pointer to a character
cptr = charary;      // name of an array is the pointer to the first element
for (i = 0; i < 10; i++) charary[i] = 'A'+i; // set first 10 chars
for (i = 0; i < 10; i++) cptr[i] = 'A'+i; // set first 10 chars
for (i = 0; i < 10; i++) *(cptr+i) = 'A'+i; // set first 10 chars, pointer arithmetic
for (cptr = charary, i=0; i < 10; cptr++,i++) *cptr = 'A'+i; // again...
cptr = charary;
for ( i=0; i < 10; i++) *cptr++ = 'A'+i; // again...
```

- Precedence: ++ done first, then *, ++ is value before increment

Other loops:

```
i = 0;  while (charary[i] == ' ') i++;  // What does this do?
cptr = charary;  while (*cptr == ' ') cptr++; // ??
```

- Moves i or cptr past blank spaces

Use the easiest notation for you to understand!

String library has lots of useful functions. (man 3 string)

- Use strn versions if available: eg: strcpy vs strncpy
  - excption: strdup() may be the better one to use

Dynamic Arrays

- Use the array functionality of pointers

```
int *dynary;
dynary = (int *) malloc (sizeof(int) * 100);
```

  - use dynary like an array or pointer.

- Deallocation: (reclaim space for future use)

```
free(dynary);
```

  - Wasted computation

```
free(dynary);
exit(0);
```

  - Why wasted?
    - Like cleaning the kitchen before burning the entire house down.
- Dynamic local arrays

```
void func (int x) {
  char ary[x];
  .... use ary ...
} // No free() needed
```

Multidimensional dynamic arrays

Harder to do ....

- int *singdim;
- int **twodim; // ???
  - points to a memory location that points to an integer
  - Two step allocation ...
    - Allocate first array, an array of "int *"
    - For each element, allocate an array of "int"
- see arrays.c

Returning pointers, not arrays

```
int [] func ();  // illegal

int * func ();  // legal
```

Example:

```
int * zeroary (int size)
{
   int *p;

   p = (int *) malloc (size * sizeof(int));

   for (int ix = 0; ix < size; ix++) p[ix] = 0;
      // or   memset ((void *)p, 0, sizeof(int)*size);

   return p;
}
```

- Cast p to be a generic pointer ... void *.

Returning information via parameters (review, important!)

- Returning data as the value of a function:

```
int square (x) { return x*x; }
```

- Returning multiple data elements via parameters

```
void swap (int *a, int *b) { int t;  t=*a; *a=*b; *b=t; }
(Use) int x, y; ....   swap(&x, &y);
```

- Very common method of data return from system calls:

```
int status;
...  wait(&status) .....
```

- Arrays passed by pointer and thus the function can change the array

```
void toUpper (char *string) {
  char *ptr = string;  //  Don't change the parameter!
  while (*ptr) {
    *ptr = toupper(*ptr);
    ptr++;  // Don't put the ++ in the previous line
  }
}
```