

UNIX System Programming

C programming

What's in a C program?

- The C language
- Compiler directives
 - INCLUDE statements - #include
 - Constant declarations - #define
- function prototypes
- main()
- functions()

The C language

- Similar to all other programming languages as it has flow control structures (if{}, while{}, for{})
- There is no data type for boolean values. You can however make your own.
 - a value of zero evaluates as FALSE
 - non-zero values are considered TRUE.

- Use `#define` to make your own:

```
#define FALSE 0
```

```
#define TRUE 1 (not zero!)
```

The C language

- There are predefined variable types: int, double, float, char
- You can define arrays of all of the different types:

`typename variablename[size]`

- typename is any type
- variablename is any legal variable name
- size is a number the compiler can figure out

– For example

`int a[10];`

- Defines an array of `int(s)` with subscripts ranging from 0 to 9
- There are `10*sizeof(int)` bytes of memory reserved for this array.
- You can use `a[0]=10; x=a[2]; a[3]=a[2];` etc.
- You can use `scanf("%d",&a[3]);`

Array-Bounds Checking

- C, **unlike many languages, does NOT check** array bounds subscripts during:
 - Compilation (some C compilers will check literals)
 - Runtime (bounds are never checked)
- If you access off the ends of any array, it will calculate the address it expects the data to be at, and then attempts to use it anyways
 - may get “something...”
 - may get a memory exception (segmentation fault, bus error, core dump error)
- It is the programmer’s responsibility to ensure that their programs are correctly written and debugged!
 - This does have some advantages but it does give you all the rope you need to hang yourself!

Multidimensional Arrays

- Arrays in C can have virtually as many dimensions as you want.
- Definition is accomplished by adding additional subscripts when it is defined.
- For example:
 - `int a [4] [3] ;`
 - defines a two dimensional array
 - `a` is an array of `int[3];`
- In memory:



Initializing Multidimensional Arrays

- The following initializes `a[4][3]`:

```
int a[4][3] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10, 11, 12} };
```

- Also can be done by:

```
int a[4][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

— is equivalent to

```
a[0][0] = 1;
```

```
a[0][1] = 2;
```

```
a[0][2] = 3;
```

```
a[1][0] = 4;
```

```
...
```

```
a[3][2] = 12;
```

--- Which one is easier to read?

Arrays as Function Parameters

- In this program, the array addresses (i.e., the values of the array names), are passed to the function `inc_array()`.
- This does not conflict with the rule that “parameters are passed by values”.

```
void inc_array(int a[ ], int size)
{
    int i;
    for(i=0;i<size;i++)
    {
        a[i]++;
    }
}
```

```
void inc_array(int a[ ],int size);

main()
{
    int test[3]={1,2,3};
    int ary[4]={1,2,3,4};
    int i;

    inc_array(test,3);

    for(i=0;i<3;i++)
        printf("%d\n",test[i]);

    inc_array(ary,4);

    for(i=0;i<4;i++)
        printf("%d\n",ary[i]);
    return 0;
}
```


~~Strings~~ are Character Arrays

- ~~Strings~~ in C are simply arrays of characters.
 - Example: `char s[10];`
- This is a ten (10) element array that can hold a character string consisting of ≤ 9 characters.
- This is because C does not know where the end of an array is at run time.
 - By convention, C uses a NULL character `'\0'` to terminate all strings in its library functions
- For example:
`char str[10] = {'u', 'n', 'i', 'x', '\0'};`
- It's the string terminator (not the size of the array) that determines the length of the string.

Accessing Individual Characters

- The first element of any array in C is at index 0. The second is at index 1, and so on ...

```
char s[10];
```

```
s[0] = 'h';
```

```
s[1] = 'i';
```

```
s[2] = '!';
```

```
s[3] = '\0';
```

h	i	!	\0	?	?	?	?	?	?
---	---	---	----	---	---	---	---	---	---

s [0] [1] [2] [3] [4] [5] [6] [7] [8] [9]

- This notation can be used in all kinds of statements and expressions in C:
- For example:

```
c = s[1];
```

```
if (s[0] == '-') ...
```

```
switch (s[1]) ...
```

Notice the single quote around a single character. That denotes a type *char* in a comparison. A single character in double quotes is of a different data type! Automatically assumes an array of *char*.

The C String Library

- String functions are provided in an ANSI standard string library.
 - Access this through the include file:
`#include <string.h>`
 - Includes functions such as:
 - Computing length of string
 - Copying strings
 - Concatenating strings
 - This library is guaranteed to be there in any ANSI standard implementation of C.

NOTE: Many of the have variants which control the number of characters to use.

The C language - Functions

Functions return values: some standards.

Function calls to system/kernel functions typically perform a specific task.

They return a “value” upon completion.

- Normally a non-zero value indicates a non-standard termination of the call. i.e.
- For system functions, a return value of -1 indicates a failure.
- It is IMPERATIVE to check the return value for system calls.
- It is also **GOOD PRACTICE**, to mimic this convention when writing your own code.

The C language

For this course, the GOTO reference for the C language is:

`https://www.gnu.org/software/libc/manual/`

`https://www.gnu.org/software/gnu-c-manual/`

A Sample Program

```
int main( int argc, char *argv[] )  
{  
    exit(0);  
}
```

The *int* in front of the function name (main) indicates that when the code exits, it will return a “status code” back to the calling routine.

- When programs terminate normally, the status code is usually zero.
- You can set this value using the *exit* system call.
- When it is non-zero, you can indicate conditions to the calling program so it can take countermeasure for the error.

In this program, main receives two items from the O/S.

- The number of parameters on the command line: **argc**
- A char array containing the values present after the verb. **argv[]**

#INCLUDE

The #INCLUDE compiler directive tells the compiler to load the function definitions for the specified library.

If you are using specific functions, you need to include the appropriate header file so the compiler/linker can find all of the “parts” they need.

If you check the *man* page for the function you are using, you can find out which header file contains the appropriate function prototypes.

#INCLUDE

Function	Header to #include
cos(), sin(), log()	math.h
str <u>n</u> cpy(), str <u>n</u> cmp()	string.h
asctime(), localtime()	time.h
atoi(), free(), malloc()	stdlib.h

#define

You can use the **#define** compiler directive to set the values of constants for your code.

This allows you to define the length of strings, the dimensions of array or any other constant and simply use a set name in the code to make it more legible.

It also simplifies global changes to the code.

The compiler replaces the defined name with its constant value AT COMPILE TIME. This is NOT a variable.

In our example: **#define MAX_PARAMS 5**

#define

e.g.

```
#define MAXBUFFER 1024
```

```
#define MAXROWS 20
```

```
#define MAXCOLS 20
```

Then later in your code...

```
char    FileBuffer[MAXBUFFER] ;
```

```
double HeatFlow[MAXROWS][MAXCOLS] ;
```

Function prototypes

A function prototype gives the compiler a description of:

- The name of a function
- A detailed description of the parameters that will be passed.
- What type of value the function is returning to the calling program

The compiler can then use this information to ensure that all calls to the function in your code as syntactically correct.

The `#include` statements typically load the function definitions (headers) for the different pre-defined routines.

Function prototypes

```
// prototype for the function defined later.
```

```
int my_fopen(int *fd, char *UserFile);
```

Later in the code...

```
//-----  
  
int my_fopen(int *fd, char *UserFile)  
{  
    *fd = open( UserFile , O_RDONLY ); // open file for reading  
    .....  
} // end of my_open
```

Example #2 – Hello World!

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

```
int  main(int argc, char *argv[])  
{  
    printf( "Hello World !!!\n" );  
} /* end main */
```

Example #3 – A bit more...

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

```
#define TRUE          1
#define FALSE         0
#define MAXLINE      4096
#define MAXFILENAME  64
```

```
int  my_fopen(int *fd, char *UserFile);    // prototype for the
      function defined later.
```

```

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

    char buffer[MAXLINE];           // max read buffer length defined at compile time
    char UserFile[MAXFILENAME];     // name of the control file from the command line

    ...

    strncpy(UserFile,argv[1],MAXFILENAME); // only take the first 64 characters
                                           // (check for "/", ".", ".." and NULL)

    if( my_fopen( &fd, UserFile) )
    {
        printf("Processing file: %s\n", UserFile);
        // this is where the code to read the file should be...
    }
    else
        { exit(2); }

    ...
} /* end main */

//-----

int my_fopen(int *fd, char *UserFile)
{
    *fd = open( UserFile , O_RDONLY ) ;    // open file "data" for reading
    .....
} // end of my_open

```



The C language

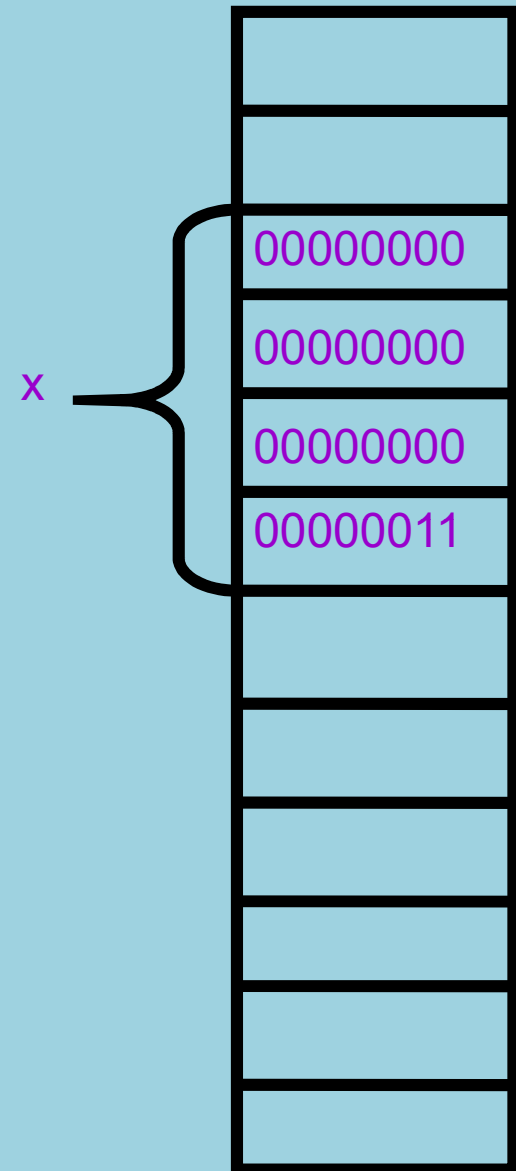
- You can pass information to functions either:
 - By value
 - by reference.

“by reference” means using the address of the memory location where a specific value is stored.

If you have never dealt with pointers before, a good reference is: <https://beej.us/guide/bgc/>

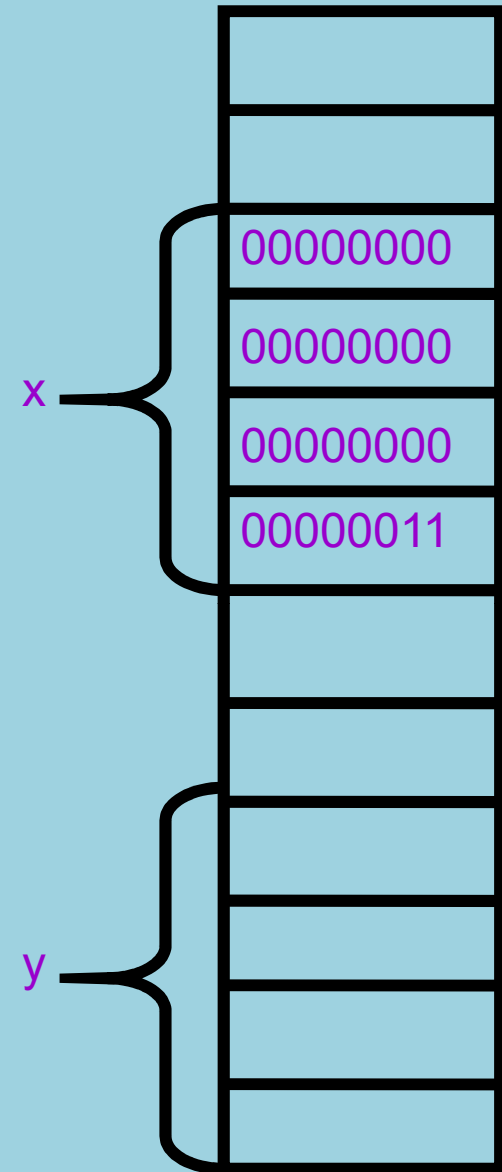
Pointer Fundamentals

- When a variable is defined the compiler (linker/loader actually) allocates a real memory address for the variable.
 - `int x;` will allocate 4 bytes in the main memory, which will be used to store an integer value.
- When a value is assigned to a variable, the value is actually placed to the memory location that was allocated.
 - `x=3;` will store integer 3 in the 4 bytes of memory.



Pointers

- When the value of a variable is used, the contents in the memory are used.
 - `y=x;` will read the contents in the 4 bytes of memory, and then assign it to variable `y`.
- `&x` can get the address of `x`. (referencing operator `&`)
- The address can be passed to a function:
 - `printf("%d", &x);`
- The address can also be stored in a variable



Pointers

- To declare a pointer variable

```
type * pointername;
```

- Pointers are addresses.
- All addresses on an operating system are of the same size (bytes)
- The compiler checks the TYPE of the pointer you are attempting to use.
- It will issue a warning if you are mixing the TYPE of the pointers in an expression
- That's a head's up that you are doing something really WRONG.

Pointers

- For example:

- `int * p1;` // *p1 is a variable that holds an address which points to a memory location which is just large enough to hold an integer.* (or p1 is a int pointer) (4 bytes)
- `char *p2;` // pointer to a memory location (1 byte)
- `unsigned int * p3;` // pointer to a unsigned integer (4bytes)

```
p1 = &x;          /* Store the address in p1 */  
printf("%d", p1); /* i.e. printf("%d",&x); */  
p2 = &x;          /* Will get warning message */
```

Initializing Pointers

- Like other variables, always initialize pointers before using them!!!
- For example:

```
int main(){
```

```
    int x;
```

```
    int *p;
```

```
    printf("%d",p); /*
```

```
    p = &x;
```

```
    printf("%d",p); /* Correct */
```

```
}
```



Don't

Using Pointers

- You can use pointers to access the values of other variables, *i.e.* the contents of the memory for other variables.
- To do this, use the `*` operator (dereferencing operator).
 - Depending on different context, `*` has different meanings.
- For example:

```
int n, m=3, *p;
```

```
p=&m;           // p = address of m
```

```
n=*p;          // store what is pointer to by p into m
```

```
printf("%d\n", n);
```

```
printf("%d\n", *p);
```

Pointers as Function Parameters

- Sometimes, you want a function to assign a value to a variable.
 - e.g. `scanf()`
- E.g. you want a function that computes the minimum AND maximum numbers in 2 integers.
- Method 1, use two global variables.
 - In the function, assign the minimum and maximum numbers to the two global variables.
 - When the function returns, the calling function can read the minimum and maximum numbers from the two global variables.
- This is bad because the function is not reusable.

Pointers as Function Parameters

- Instead, we use the following function

```
void min_max(int a, int b, int *min, int *max)
{
    if(a>b)
    {
        *max=a;
        *min=b;
    }
    else
    {
        *max=b;
        *min=a;
    }
}
```

Storing values into the memory locations pointed to by max & min

Passing the address of small & big

```
int main()
{
    int x,y;
    int small, big;

    printf("Two integers: ");
    scanf("%d %d", &x, &y);

    min_max(x,y,&small,&big);

    printf("%d <= %d", small, big);
    return 0;
}
```

Pointer Arithmetic (1)

When a pointer variable points to an array element, there is a notion of adding or subtracting an integer to/from the pointer.

```
int a[ 10 ], *p;
```

```
p = &a[2];
```

```
*p = 10;
```

```
*(p+1) = 10;
```

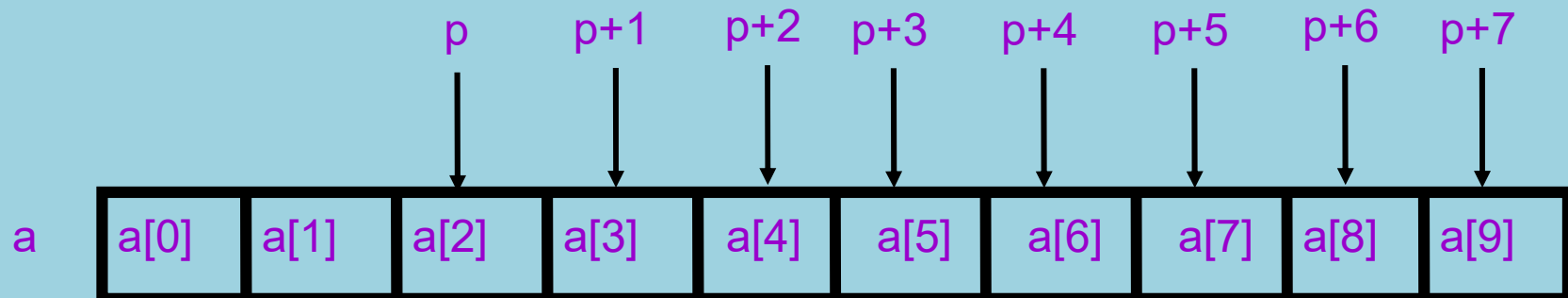
```
printf("%d", *(p+3));
```

```
int a[ 10 ], *p;
```

```
a[2] = 10;
```

```
a[3] = 10;
```

```
printf("%d", a[5]);
```



Note: +1, +2 etc are increasing the memory location by the **SIZE** of the **data type** and not simply by one or two ! the compiler “KNOWS” how many bytes each **data type** uses!

Pointers and Arrays

- Recall that the value of an array name is also an address.
- In fact, pointers and array names can be used interchangeably in many (but not all) cases.
 - E.g. `int n, *p; p=&n;`
 - `n=1; *p = 1; p[0] = 1;`
- The major differences are:
 - Array names come with valid spaces where they “point” to. And you cannot “point” the names to other places.
 - Pointers do not point to valid space when they are created. You have to point them to some valid space (initialization).

Using Pointers to Access Array Elements

❖ A pointer can be used just like an array

```
int a[ 10 ], *p;
```

```
p = &a[2];
```

```
p[0] = 10;
```

```
p[1] = 10;
```

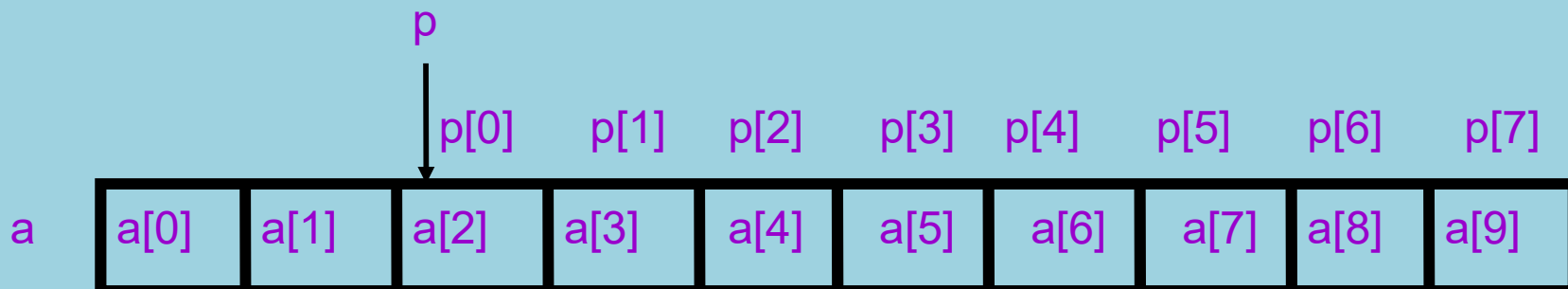
```
printf("%d", p[3]);
```

```
int a[ 10 ], *p;
```

```
a[2] = 10;
```

```
a[3] = 10;
```

```
printf("%d", a[5]);
```



An Array Name is Like a Constant Pointer

- Array name is like a constant pointer which points to the first element of the array.

```
int a[10], *p, *q;  
p = a;           /* p = &a[0] */  
q = a + 3;       /* q = &a[0] + 3 */  
a++;             /* illegal !!! */
```

- Therefore, you can “pass an array” to a function. Actually, the address of the first element is passed.

```
int a[ ] = { 5, 7, 8 , 2, 3 };  
sum( a, 5 ); /* Equal to sum(&a[0],5) */  
.....
```

An Example

```
/* Sum - sum up the ints
    in the given array
*/

int sum(int *ary, int size)
{
    int i, s;
    for(i = 0, s=0; i<size;i++){
        s+=ary[i];
    }
    return s;
}
```

```
/* In another function */
int a[1000],x;
.....
x= sum(&a[100],50);

/* This sums up a[100],
a[101], ..., a[149] */
```

Now the address of the first parameter to the function sum() is the address of the 100th element of the array a. sum() therefore adds up the next 50 elements from that point on!

Allocating Memory for a Pointer (1)

- The following program is wrong!

```
#include <stdio.h>
int main()
{
    int *p;
    scanf("%d",p) ;
    return 0;
}
```

- This one is correct:

```
#include <stdio.h>
int main()
{
    int *p;
    int a;
    p = &a;
    scanf("%d",p) ;
    return 0;
}
```

Allocating Memory for a Pointer (2)

- There is another way to allocate memory so the pointer can point to something:

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int *p;
    p = (int *) malloc( sizeof(int) ); /* Allocate 4 bytes */
    scanf("%d", p);
    printf("%d", *p);

    free(p);      /* This returns the memory to the system.*/
                  /* Important !!! */
}
```


Allocating Memory for a Pointer (3)

- Prototypes of `malloc()` and `free()` are defined in `stdlib.h`
`void * malloc(size_t number_of_bytes);`
`void free(void * p);`
- You can use `malloc` and `free` to dynamically allocate and release the memory;

```
int *p;  
p = (int *) malloc( 1000 * sizeof(int) );
```

```
for(i=0; i<1000; i++)  p[i] = i;
```

```
p[1000]=3; /* Wrong! */
```

```
free(p);
```

```
p[0]=5; /* Wrong! */
```



Structures (1)

- Structures are C's way of grouping collections of data into a single manageable unit.
 - This is also the fundamental element of C upon which most of C++ is built (i.e., classes).
 - Similar to Java's classes.
- An example:
 - Defining a structure type:

```
struct coord {  
    int x ;  
    int y ;  
};
```
 - This defines a new type **struct coord**. No variable is actually declared or generated.

Structures (2)

- Define **struct** variables:

```
struct coord {  
    int x,y ;  
} first, second;
```

- Another Approach:

```
struct coord {  
    int x,y ;  
};  
  
.....  
struct coord first, second; /* declare variables */  
struct coord third;
```

The C language

You can build your own “types” using typedef
syntax: typedef “original C variable type” name_name

e.g. `typedef unsigned char byte_type;`
`typedef double real_number_type;`

To make a type definition of an array, you first provide the type of the element, and then establish the number of elements at the end of the type definition:

```
typedef char array_of_bytes[5];  
array_of_bytes five_bytes = {0, 1, 2, 3, 4};
```

Structures (3)

- You can even use a **typedef** if your don't like having to use the word “**struct**”

```
typedef struct coord coordinate;
```

```
coordinate first, second;
```

(syntax: **typedef** **struc** **def'n** **synonym**)

- In some compilers, and all C++ compilers, you can usually simply say just:

```
coord first, second;
```

Structures (4)

- Access structure variables by the dot (.) operator

- Generic form:

`structure_var.member_name`

- For example:

`first.x = 50 ;`
`second.y = 100 ;`

- These member names are like the public data members of a class in Java (or C++).

– No equivalent to function members/methods.

- `struct_var.member_name` can be used anywhere a variable can be used:

– `printf ("%d , %d", second.x , second.y) ;`
– `scanf ("%d, %d", &first.x, &first.y) ;`



Structures (5)

- You can assign structures as a unit with =
first = second;
instead of writing:
first.x = second.x ;
first.y = second.y ;
- Although the saving here is not great
 - It will reduce the likelihood of errors and
 - Is more convenient with large structures
- This is different from Java where variables are simply references to objects.
first = second;
makes first and second refer to the same object.

Structures Containing Structures

- Any “type” of thing can be a member of a structure.
- We can use the coord struct to define a rectangle

```
struct rectangle {  
    struct coord topleft;  
    struct coord bottomrt;  
} ;
```

- This describes a rectangle by using the two points necessary:

```
struct rectangle mybox ;
```

- Initializing the points:

```
mybox.topleft.x = 0 ;  
mybox.topleft.y = 10 ;  
mybox.bottomrt.x = 100 ;  
mybox.bottomrt.y = 200 ;
```

An Example

```
#include <stdio.h>

struct coord
{
    int x;
    int y;
};

struct rectangle
{
    struct coord topleft;
    struct coord bottomrt;
};

int main () {
    int length, width;
    long area;
    struct rectangle mybox;

    mybox.topleft.x = 0;
    mybox.topleft.y = 0;
    mybox.bottomrt.x = 100;
    mybox.bottomrt.y = 50;

    width= mybox.bottomrt.x - mybox.topleft.x;
    length=mybox.bottomrt.y - mybox.topleft.y;

    area    = width * length;

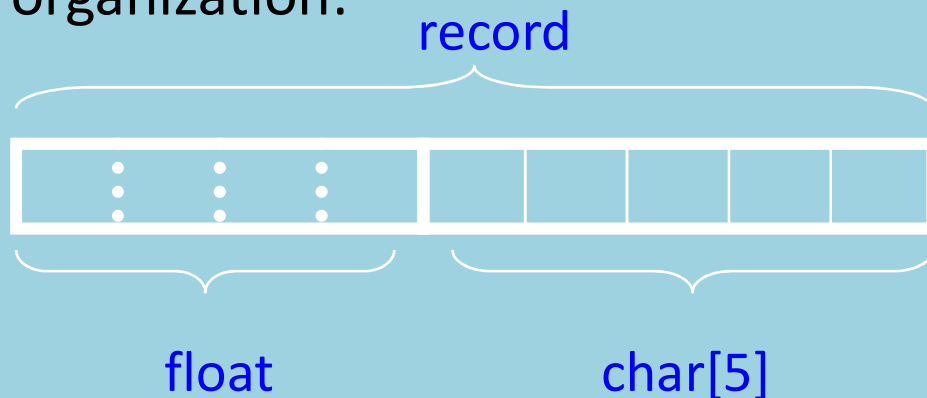
    printf ("The area is %ld units.\n", area);
}
```

Structures Containing Arrays

- Arrays within structures are the same as any other member element.
- For example:

```
struct record {  
    float  x;  
    char   y[5];  
} ;
```

- Logical organization:



An Example

```
#include <stdio.h>
struct data
{
    float amount;
    char  fname[30];
    char  lname[30];
}  rec;

int main ()
{
    struct data rec;
    printf ("Enter the donor's first and last names, \n");
    printf ("separated by a space: ");
    scanf ("%s %s", rec.fname, rec.lname);
    printf ("\nEnter the donation amount: ");
    scanf ("%f", &rec.amount);
    printf ("\nDonor %s %s gave $%.2f.\n",
            rec.fname, rec.lname, rec.amount);
}
```

Arrays of Structures

- The converse of a structure with arrays:
- Example:

```
struct  entry {  
        char  fname[10] ;  
        char  lname[12] ;  
        char  phone[8]  ;  
    } ;  
struct entry list[1000];
```

- This creates a list of 1000 identical entry(s).
- Assignments:

```
list[1] = list[6];  
strcpy(list[1].phone, list[6].phone);  
list[6].phone[1] = list[3].phone[4] ;
```

An Example

```
#include <stdio.h>
struct entry {
    char fname[20];
    char lname[20];
    char phone[10];
};
```

```
int main()
{
    struct entry list[4];
    int i;
    for (i=0; i < 4; i++) {
        printf ("\nEnter first name: ");
        scanf ("%s", list[i].fname);
        printf ("Enter last name: ");
        scanf ("%s", list[i].lname);
        printf ("Enter phone in 123-4567 format: ");
        scanf ("%s", list[i].phone);
    }
    printf ("\n\n");
    for (i=0; i < 4; i++)
    {
        printf ("Name: %s %s", list[i].fname, list[i].lname);
        printf ("\t\tPhone: %s\n", list[i].phone);
    }
}
```

Initializing Structures

- Simple example:

```
struct sale {  
    char  customer[20] ;  
    char  item[20] ;  
    int amount ;  
};
```

```
struct sale mysale = { "Acme Industries",  
                      "Zorgle blaster",  
                      1000 } ;
```

Initializing Structures

- Structures within structures:

```
struct customer
```

```
{  
    char firm[20] ;  
    char contact[25] ;  
};
```

```
struct sale
```

```
{  
    struct customer buyer ;  
    char item[20] ;  
    int amount ;  
} mysale =  
    {{"Acme Industries","George Adams"},"Zorgle Blaster",1000};
```


Initializing Structures

- Arrays of structures

```
struct customer
```

```
{  
    char firm[20] ;  
    char contact[25] ;  
} ;
```

```
struct sale
```

```
{  
    struct customer buyer;    } ;  
    char item[20] ;  
    int amount ;  
} ;
```

```
struct sale y1990[100] = {  
    { { "Acme Industries",  
        "George Adams"} ,  
        "Left-handed Idiots",  
        1000},  
    { { "Wilson & Co.",  
        "Ed Wilson"} ,  
        "Thingamabob",  
        290}
```

Pointers to Structures

```
struct part {  
    float price ;  
    char name[10] ;  
} ;
```

```
struct part *p , thing;
```

```
p = &thing;
```

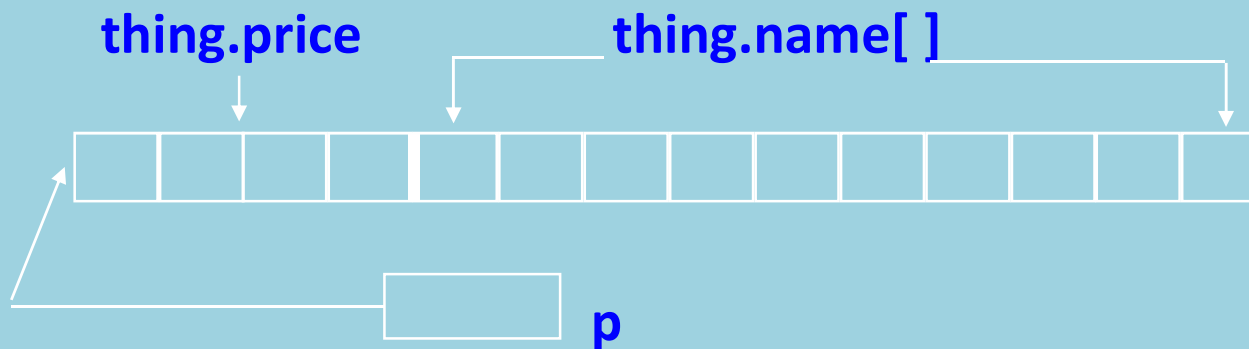
```
/* The following three statements are equivalent */
```

```
thing.price = 50;
```

```
(*p).price = 50;    /* () around *p is  
    needed */
```

```
p -> price = 50;
```

Pointers to Structures



- `p` is set to point to the first byte of the `struct` variable

Pointers to Structures

```
struct part * p, *q;
```

```
p = (struct part *) malloc( sizeof(struct part) );  
q = (struct part *) malloc( sizeof(struct part) );  
p -> price = 199.99 ;
```

```
strcpy( p -> name, "hard disk" );
```

```
(*q) = (*p) ;
```

```
q = p;
```

```
free(p) ;
```

```
free(q) ; /* This statement causes a problem !!! Why? */
```

Pointers to Structures

- You can allocate a structure array as well:

```
{
    struct part *ptr;
    ptr =(struct part *) malloc( 10 * sizeof(struct part));

    for( i=0; i< 10; i++)
    {
        ptr[i].price = 10.0 * i;
        sprintf( ptr[i].name, "part %d", i );
    }
    .....
    free(ptr) ;
}
```

Pointers to Structures

- You can use pointer arithmetic to access the elements of the array:

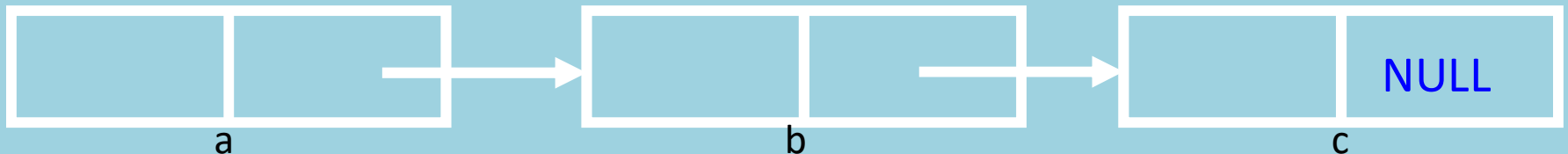
```
{
    struct part *ptr,  *p;
    ptr=(struct part *) malloc( 10 * sizeof(struct part));

    for( i=0, p=ptr; i< 10; i++, p++)
    {
        p -> price = 10.0 * i;
        sprintf( p -> name, "part %d", i );
    }
    .....
    free(ptr) ;
}
```

Pointer as Structure Member

```
struct node{  
    int data;  
    struct node  
    *next;  
};  
struct node a,b,c;  
  
a.next = &b;  
b.next = &c;  
c.next = NULL;
```

```
a.data = 1;  
  
a.next->data = 2; /* b.data =2 */  
a.next->next->data = 3;  
/* c.data =3 */  
  
c.next =  
    (struct node *)  
    malloc(sizeof(struct node));  
.....
```



Assignment Operator vs. memcpy

- This assign a struct to another
- Equivalently, you can use memcpy

```
{  
    struct part a,b;  
    b.price = 39.99;  
    b.name = "floppy";  
    a = b;  
}
```

```
#include <string.h>  
.....  
{  
    struct part a,b;  
    b.price = 39.99;  
    b.name = "floppy";  
    memcpy (&a, &b, sizeof (part) )  
    ;  
}
```


Array Member vs. Pointer Member

```
int main()
{
    struct book a,b;
    struct book {
        float price;
        char name[50];
    };
    b.price = 19.99;
    strcpy(b.name, "C handbook");
    a = b;
    strcpy(b.name, "Unix handbook");
    puts(a.name);
    puts(b.name);
}
```

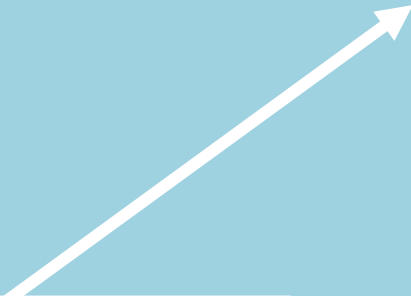
Array Member vs. Pointer Member

```
int main()
{
    struct book {
        float price;
        char *name;
    };

    struct book a,b;

    b.price = 19.99;
    b.name = (char *) malloc(50);
    strcpy(b.name, "C handbook");

    a = b;
    strcpy(b.name, "Unix handbook");
    puts(a.name);
    puts(b.name);
    free(b.name);
}
```



A function called
strdup() will do the
malloc() and strcpy()
in one step for you!

Passing Structures to Functions (1)

- Structures are passed *by value* to functions
 - The parameter variable is a local variable, which will be assigned by the value of the argument passed.
 - Unlike Java.
- This means that the structure is copied if it is passed as a parameter.
 - This can be inefficient if the structure is big.
 - In this case it may be more efficient to pass a pointer to the `struct`.
- A `struct` can also be returned from a function.

Passing Structures to Functions (2)

```
struct book {  
    float price;  
    char abstract[5000];  
};  
  
void print_abstract(  
    struct book *p_book)  
{  
    puts( p_book->abstract );  
};  
  
struct pairInt {  
    int min, max;  
};  
  
struct pairInt min_max(int x,int y)  
{  
    struct pairInt pair;  
    pair.min = (x > y) ? y : x;  
    pair.max = (x > y) ? x : y;  
    return pair;  
}  
  
int main(){  
    struct pairInt result;  
    result = min_max( 3, 5 );  
    printf("%d<=%d", result.min,  
        result.max);  
}
```

Overlaying multiple structures - union

```
/*-----  
|   Define the structure of Rock_Paper_Scissor packets.  
|  
*/  
typedef struct PacketInfo_def {  
    union {  
        struct {  
            char    AppId[4];          /* 3 characters RPS and the magic NULL */  
            char    PktType[2];        /* type of message being passed      */  
            char    PktData[94];       /* The REAL data                      */  
        } PS ;                        /* defines the Packet Structure      */  
        struct {  
            char    AppId[4];          /* 3 characters RPS and the magic NULL */  
            char    PktType[2];        /* type of message being passed      */  
            short   UGames;            /* Number of games played            */  
            short   Uwins;             /* Number of wins                    */  
            short   Ulosses;           /* Number of losses                  */  
            char    Filler[88];  
        } GS ;                        /* Games stats                       */  
        struct {  
            char    RawDatum[100];     /* defines the RAW data for socket xfer*/  
        } RP;                         /* Raw Packet structure              */  
    } PT;                             /* PT  Packet type - UNION          */  
  
} PacketInfo;
```

Additional things with C

Interacting with the O/S

Accessing command line parameters

At run time, it is often desirable to pass information to our program.

The easiest way to do this is at the command line.

Information on the running of a program can be retrieved from within your code if you define your `main()` function properly.

Command line parameters

Defining that you intend to access data provided at process creation time:

```
int  main(int argc, char *argv[])  
{  
    ... You code goes here  
}
```

Where `argc` = # of parameters on the command line

`*argv[]` is an array of strings which contain the values passed.

Command line parameters

The `argv[]` array, like all other C arrays starts at the index 0.

If I issue the following command:

```
jacques@ubuvvm64> ./my_prog Hello value2 6.3
```

In code, `argv[]` will look like:

```
argv[0]="my_prog"  
argv[1]="Hello"  
argv[2]="value2"  
argv[3]="6.3"           // notice the double quotes!!!
```

Note: `argv[]` is an array of NULL terminated arrays of ***char***. You **MUST** convert them to whatever variable type you need before you use them

Command line parameters

Conversions to other variable types: (stdlib.h)

atof() – ASCII string to a float

atoi() – ASCII string to an int

atol() – ASCII string to a long

You can test the values of the parameters and change the logic flow of your program:

```
if( argv[1] == 'a' ) {  
    printf( 'user selected A\n'; }
```

Command line parameters

A more “industry standard” way of parsing command line arguments is with the `getopt()` function.

For beginners, It appears a little more daunting than just a bunch of if statements.

(see: https://www.gnu.org/software/libc/manual/html_node/Example-of-Getopt.html)

```

#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main (int argc, char **argv)
{
    int aflag = 0;
    int bflag = 0;
    char *cvalue = NULL;
    int index;
    int c;

    opterr = 0;

```

```

while ((c = getopt (argc, argv, "abc:")) != -1)
    switch (c)
    {
        case 'a':
            aflag = 1;
            break;
        case 'b':
            bflag = 1;
            break;
        case 'c':
            cvalue = optarg;
            break;
        case '?':
            if (optopt == 'c')
                fprintf (stderr, "Option -%c requires an argument.\n", optopt);
            else if (isprint (optopt))
                fprintf (stderr, "Unknown option `-%c'.\n", optopt);
            else
                fprintf (stderr, "Unknown option character `\\x%x'.\n", optopt);
            return 1;
        default:
            abort ();
    }
printf ("aflag = %d, bflag = %d, cvalue = %s\n", aflag, bflag, cvalue);

for (index = optind; index < argc; index++)
    printf ("Non-option argument %s\n", argv[index]);
return 0;
}

```

Printing output on STDOUT

C provides the `printf()` function as an easy method for creating formatted output to your console window.

```
printf("format string", list of variables to be printed);
```

where the format string can be:

- some text to be printed as-is
- different format field specifiers

printf() format specifiers

Format string	Usage
%c	A single character: 'a'
%d	Signed Decimal integer: -137
%f	Floating point number: 1523.56
%p	The value of a pointer (memory address) 0x7ffceba83660
%s	A string: "Student Name"
There are many more !	

You can also specify the width of the field as well as justification:

e.g.:

%4d := a 4 digit integer: 1523

%04d := a 4 digit integer with leading zeros: 0076

%6.3f := a 6 character floating point number with 3 decimal places: 15.234

%30s := a 30 character long string field (right justified)

%-30s := a 30 character long string field (**left justified**).

Printing with printf ()

- Example:

```
char str[ ] = "Mickey Mouse";  
printf ("Student Name: %s\n", str);
```

- **printf** expects to receive a string as an additional parameter when it sees **%s** in the format string
 - Can be from a character array.
 - Can be another literal string.
 - Can be from a character pointer (more on this later).
- **printf** knows how much to print out because of the NULL character at the end of all strings.
 - When it finds a **\0**, it knows to stop.

Printing with puts()

- The `puts` function is a much simpler output function than `printf` for string printing.
- Prototype of `puts` is defined in `stdio.h`
`int puts(const char * str)`
 - This is more efficient than `printf`
 - Because your program doesn't need to analyze the format string at run-time.
- For example:
`char sentence[] = "The quick brown fox\n";`
`puts(sentence);`
- Prints out:
The quick brown fox

Inputting Strings with gets()

- `gets()` gets a line from the standard input.
- The prototype is defined in `stdio.h`
`char *gets(char *str)`
 - `str` is a pointer to the space where `gets` will store the line to, or a character array.
 - Returns `NULL` upon failure. Otherwise, it returns `str`.

```
char your_line[100];  
printf("Enter a line:\n");  
gets(your_line);  
puts("Your input follows:\n");  
puts(your_line);
```

- You can overflow your string buffer, so be careful!

Inputting Strings with scanf ()

- To read a string include:
 - `%s` scans up to but not including the “next” white space character
 - `%ns` scans the next `n` characters or up to the next white space character, whichever comes first
- Example:
 - `scanf ("%s%s%s", s1, s2, s3);`
 - `scanf ("%2s%2s%2s", s1, s2, s3);`
 - Note: No ampersand(&) when inputting strings into character arrays! (We'll explain why later ...)
- Difference between gets
 - `gets()` read a line
 - `scanf("%s",...)` read up to the next space

Compiling your code

Unlike scripts, C programs must be converted into a binary format before they can be run.

This process is called compiling.

For this course we are going to use the GNU project's C compiler: gcc

You can use the ***man*** pages to find out more about it.

Compiling your code

By default, the compiler creates a binary image of your code called `a.out`.

This is not very descriptive of your code and we should be creating better names to make the purpose of the executable more obvious.

Also, if all we use is `a.out`, how do you keep more than one binary in any one directory?

The `-o` option of `gcc` allows you to specify the name of the binary that is to be created.

Compiling your code

NOTE: be careful naming your binary.

If you accidentally call it by the same name as your source file, the compiler will **OBLITERATE** your source and overwrite it with the binary.

You will then be back at step 1.

Compiling your code

The second compiler option you may want to invoke is the WARNINGS flag.

You can add `-Wall` (Warnings all) to the compile line to get a more thorough listing of what might wrong with your code.

Compiling your code

On the system used for this course, you will be using the gcc compiler.

Usage: `gcc -Wall -o executable_output_name sourcecode_filename.c`

You may need to add some additional parameters depending on what your code call. For many math heavy applications, you may need to add `-lm` to the compile line.

Running your code

*NIX uses the **PATH** environment variable to search the system for the different commands you enter.

Your work directory is almost certainly NOT in your PATH. The system will not find your executable.

You must therefore prefix your executable name with a directory path

e.g.:

Using a full path:

```
jacques@UBU64vm> /home/jacques/lab3/read_directory
```

or, using a path relative to the current directory:

```
jacques@UBU64vm> ./read_directory
```


Documentation

Probably one of the MOST important parts of your code.

It is a requirement to make your application code clear and concise to someone auditing it.

(like the person grading your assignment!)

If you write as you go, it is fresh in your mind and it removes the drudgery of going back later!

Documentation

- Be short and sweet, don't ramble along.
- The documentation need not explain all of the theory behind the algorithm. It just needs to explain which step is being performed.
- If you are doing something "clever", don't assume it is obvious.
- Once you are done writing your code, go through it again and make sure your comments are appropriate.

Documentation

- Pick a naming convention for your variables.
- Stick to it.
- Proper variable names make your lines of code legible and self-documenting.
- Variable names such as p, j, a are NOT acceptable (see the grading scheme).

Allocating memory at run time.

Normally, you know the size of a structure you want to use in a program:

```
#define CLASS_SIZE 100  
int student_number[CLASS_SIZE]
```

Sometimes, the compiler will do this for you at compile time:

```
char my_address[] = "1600 West Bank Dr, Peterborough, ON K9J 0G2"
```

Allocating memory at run time.

in C, if you don't know, you can allocate memory to a variable at run time using `malloc()`.

```
int * student_number;  
student_number = (int *) malloc( how_many * sizeof(double));  
if( student_number == NULL )  
{  
    printf("error allocating memory(student numbers).\n");  
    exit(1);  
}
```

You could then access each value as if it were a simple array element:

```
printf("the student number is %d\n", student_number[16]);
```

make sure you `free()` the memory when done !!!

A Sample Program

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

#define MAX_PARAMS 5

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

    parameter_count = 0;
    printf("\nWe have found %d parameters on the command line. They are:\n\n",argc);
    if( argc > MAX_PARAMS )
    {
        printf("There are too many parameters on the command line. (line wrap)
               You must use fewer than %d\n",MAX_PARAMS);
        exit(1);
    }
    while( parameter_count < argc )
    {
        printf("\tparameter[%d]: %s\n",parameter_count,argv[parameter_count]);
        parameter_count++;
    }
    printf("\n");
    exit(0);
}
```