Unix and C Review

Fall 2012

These are adapted from Prof Davison's 2009 Slides

Accessing The Suns

- Use your ssh client to connect to a CSE/ECE Sun
  - Lehigh students can get a number of ssh-compatible implementations here for various operating systems.
- If you have forgotten your CSE/ECE Sun account password, or don't yet have an account, send mail to **help (at) cse.lehigh.edu**

Accessing The Suns

- On our Suns (and Linux) your default shell is **bash** (Bourne-Again Shell). Most shells, including bash, offer:
  - tab completion -- just type the first few letters and press tab to complete the command name
  - command history -- use the up and down arrows to select from your history of past commands
  - emacs-like keybindings -- C-a, C-e, C-t, C-b, C-f, C-p, C-n all same
- When finished, type exit to close the shell.

Hello World

- What does a C version of hello world look like?
- Use an editor to create hello.c
- Compile using gcc hello.c
- Assuming no errors, run using ./a.out
- If no errros, the program will output its greeting and exit
- How about a version that also says, “This is day X of the semester.” where X is a variable set earlier?

GNU Emacs

- One of, if not the, most popular and full-featured editors
- Available for most every platform/OS
- More than just an editor; it can also read email, browse web, etc
- Start emacs by typing emacs (and return) at a shell prompt
- If you are running within X-windows, a new window will open.
- Otherwise, emacs will run full-screen (within the starting terminal)
- Menus are available (use F10 to operate if not running within X-window)

Operating Emacs

- Emacs provides many, many keyboard shortcuts (as well as longer commands)
- Some combinations of keystrokes have special meanings
  - C-x (spoken as “control x”) means to press and hold the control key while you press the x key once
- Example command sequence: C-x C-f will prompt you for the name of a file to open

Essential Commands

- Quitting Emacs
  - C-x C-c -- Exit emacs permanently
  - C-x -- Suspend emacs (iconify under X)
- File Commands
  - C-x C-f -- Open a file for editing
  - C-x C-s -- Save current file to disk
  - C-x i -- Insert contents of another file
  - C-x C-w -- Write/Save-As
  - C-x k -- Close current file
- Handling errors
  - C-g -- Aborts the current command
  - C-_ or C-x u -- undoes most recent change

Essential Commands

- Searching
  - C-s -- search forward
  - RET --Abort current search at current location
  - Repeated C-s or C-r will search for next location
- Cutting and Pasting
  - Backspace(and sometimes C-Backspace) will usually delete letter to left
  - C-d (and often Delete key) will delete letter under cursor
  - C-k -- kill to end of line (i.e. cut)
  - C-y -- Yank from kill buffer (i.e. paste)
  - A sequence of repeated C-k will put all such lines in the same buffer
- Miscellaneous
  - TAB -- indent current line (depending on mode)
  - C-x 1-- delete all other windows with emacs

Files Created by Emacs

- Emacs will let you open new files
- If you close a file that was modified (without saving), it leaves a temporary file called #filename#
- If you edit and save an existing file, emacs renames the old file as filename~

Using Unix

- Reminder: How to find information about UNIX commands/utilities?
  - Type man <program> for any system command, most utilities and system calls
  - Type info <program> for any GNU utility or program (and man pages!)
  - Check your reference books
  - The rest: Google, instructors, etc
- What if you don't know the name of the program?
  - man –k <keyword>

Using Unix Shells

- When you run a program, e.g. ls or emacs, it typically takes the place of the shell, and returns you to the shell when it is finished.
- How can you stop the program if it is not running properly?
- How can you make it run in the background (so that you can continue to use the shell)?
- How can you capture the output of a program to a file?
- How do you feed the contents of a file as input to a program?
- How do you take the output of a program and use it as the input to a different program?

Finished Using Unix

- Any questions?
- Let’s move on to C

The C Programming Language

- is low-level
  - C can directly access structures that are tied to hardware
- is structured
  - Structures are used to control the flow of execution (e.g. while, for, if/then)
- is procedural
  - Can re-use code in subroutines (but not object oriented)
- is weakly typed
  - You can read data from memory as any type you like
- is staticly typed
  - Types are only checked at compile-time

C Versus C++

- C++ is mostly a superset of C
- So what are the major changes?

Major C differences from C++

- No classes or objects – all code is in functions
- C structures cannot have methods
- I/O in C is based on library functions
- No function overloading
- No new or delete (use library functions instead)
- No reference variables (aliases)

Other Differences

- Variables must be declared at the beginning of a block, typically at the beginning of the function
  - This is no longer true for C99
- No bool datatype
- No << and >> operators for I/O
- Cannot substitute and, or, and not for boolean operators &&, ||, and !
- Different approach to strings

Basic C Data Types

- Pretty much the same as in C++
  - int (%d, -40), short int, long it, long long int, unsigned int
  - float (%f, 3.14), double, long double
  - char(%c, ‘a’), unsigned char, signed char
- Variable names are also similar to C++
  - made of letters, digits, underscore
  - cannot start with a number

Operators

- Arithmetic operators
  - Typical binary operators: + - * / %
  - Unary + and -
- Logical (Boolean) operators
  - &&, ||, !
- Bitwise operators
  - &, |, ^, ~, <<,>>
- Relational operators
  - <, <=,>,=>,==,!=
- Conditional operator
  - condition> expression1:expression2;

Assignment operators

- Assignment = (e.g. a=4;)
- Modify and assign
  - Increment++ and decrement –
  - Combinations *=, /=, %=, +=, -=, &=, ^=, |=
    - a+=4;

Type conversions

- C permits (and performs) many kinds of type conversions
  - Compiling with –Wall should warn you about them if they are not explicitly cast
  - When a float is converted to an int, the value is truncated
  - Integer arithmetic generates integer results (even when the result has a fraction value)
  - If any value in an expression is of type float (or double, etc), then the result will be of that type
- Use of the type cast operator explicityly tells the compiler to do the conversion (and has higher precedence than anything but unary +/-)
  - (int) 29.55+(int) 21.99 is the same as 29+21 in C

Control Flow

- Loops

```
while (expression)
        statement
for (expr1; expr2; expr3)
        statement
do
        statement
while (expression);
```

- The break statement drops control out of the innermost loop while moves on to the next iteration.

Control Flow

- Conditionals

```
if (expression)
    statement1
else
    statement 2
 switch (expression) {
    case const-expression: statements
     case const-expression: statements
     default: statements
    }
```

C Arrays

- Assuming we know the number of elements, we can easily create and manipulate arrays of items

```
int i[5];
i[0]=i[1]=1;
i[2]=2;
i[i[2]+1]=i[2]+2;
```

- Unfortunately, we cannot set all values of an array at once, nor assign one array to another. Instead, we must iterate through each item.

```
int i, a[10];
// clear of copy all values
 for (i=0;i++; i<1-_ {
    a[i]=0; // or a[i]=x[i];
}
```

Character Arrays

- An array of characters works the same. We can create and initialize them with the same syntax

  char hello[]={'h','e','l','l','o',' !'};
- There are many characters that are not letters, numbers or keyboard symbols. Such characters are represented by an escape sequence using the backslash.
- Common characters include:

  \n a "newline" character (e.g. a line feed)

  \b a backspace

  \r a carriage return (without a line feed)

  \' a single quote (e.g. in a character constant)

  \" a double quote (e.g. in a string constant)

  \\ a single backslash
- Character constants always use single quotes.

C Strings

- A string constant is zero or more characters enclosed in double quotes.

```
print (“hello world.\n”);
```

- Strings in C are always terminated internally by a null character

```
char word[]={“hello!”};
char word[7]=“hello!”;
char word[]={‘H’, ‘e’,’l’,’l’,’o’,’!’,’\0’};
printf(“She said %s\n”,word);
```

- Let’s write code to concatenate two strings. Here is a skeleton that we can fill out.

Using Strings in C

- There are many useful functions to help us use strings in C.
- For example, since C does not let us assign entire arrays, we use the strcpy function to copy one string to another;

```
#include <string.h>
char string1[]="Hello, world!";
char string2[20];
strcpy(string2, string1);
```

use strncpy

Use Strings in C

- Let us compare arrays

```
char string3[]="this is";
char string4[]="a test";
 if (strcmp(string3, string4)==0)
    printf("strings are equal\n");
 else
    printf("strings are different\n");
```

- This code fragment will print "strings are different". Notice strcmp does not return a boolean result.
- And of course the string concatenate function (which we implemented previously) is available as strcat.

Using Strings in C

- Often you'll need to know how long a string is
  - .e.g to see if a copy of it will fit into a destination buffer
  - You can call strlen, which returns the string length(i.e. the number of characters in it), not including the terminating null:

```
char string7[]="abc";
int len=strlen(string7);
printf("%d\n",len);
```

which might be implemented as

```
int mystrlen(char str[]){
  int i;
  for (i=0; str[i];i++);
  return i;
}
```

  - We can print strings using printf() with the %s format (e.g. printf("%s\n",string5);).

Structures

- Sometimes it is useful to collect a set of variables together, and reference them as a unit.

```
struct date {
    int month;
    int day;
    int year;
};
  struct date today;
```

- In C, this is accomplished with a structure
- Access members of a struct using the period operator.

```
today.day=2
today.month=4;
```

Structures

- Can be passed as a parameter (by value) to a function.
- Can have arrays of structs, and structs containing arrays or other structs.

```
struct date {
  int month;
  char monthname[10];
  int day;
  int year;
  struct tm time; // struct inside
};
struct date year[365];
```

C Libraries Use Structs

- Consider acquiring the current time.
- The time call returns the number of seconds since the Unix epoch (Jan 1, 1970)
- The localtime function converts that to a struct:

```
struct tm {
    int tm_sec; /* seconds after the minute  */
    int tm_min; /* minutes after the hour */
    int tm_hour; /* hour since midnight - [0,23] */
    int tm_mday; /* day of the month- [1,31] */
    int tm_mon; /* months since January –[0,11] */
    int tm_year; /* years since 1900 */
    int tm_wday; /* days since Sunday-[0,6] */
    int tm_yday; /* days since Jan 1st*/
    int tm_isdst; /* flag for daylight savings time */
 };
```

Pointers

- With pointers, we can manipulate addresses or contents referenced by the addresses
- We can first declare a pointer variable

  int *ip;

- Which tells the compiler that variable ip is of type (int *) or equivalently that *ip is of type int.

Pointers

```
int ip;
```

- Pointers contain addresses (Note that ip above is currently undefined). We can set ip to the address of another variable easily, as in

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

- At this point, the contents of ip and the address of i are the same – they both refer to the same memory location which contains the number 5.

Pointers

- We can read the contents of the location a pointer points to by prefixing the pointer name with an asterix, as in

  print(“%d\n”, *ip);
- We can also write to the contents of the location that a pointer points to in the same way

  *ip=4;
- So now the value of i will also be 4.
  We can also examine the variable ip itself, by casting it to a type that is usable in printf(), as in”

  printf(“%uld\n”,(unsigned long) ip);
- Note that here we will see the 32-bit value of ip (that is the memory location), and not the value contained in the location to which ip points.

Parameters in C Functions

- Parameters to C functions are always call-by-value.
  - So formal parameters are copies of actual arguments.
- Arrays are passed by using the name of the array, which effectively passes the address of the array.
  - Changes to array elements will affect original array
- Passing the address (using &) of a variable is how we permit functions to modify the contents of variables
- Must use the *var convention to modify contents of memory address.

```
 void multbytwo(int *x) }
   *x=*x*2;
}
void main(void) {
    int a=4;
    multbytwo(&a);
    multbytwo(&a);
    printf("a is %d\n", a);
}
```

Pointer Declarations

- As with any other variable types, you can initialize the value of a pointer variable when you declare it, as in

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

- But you cannot initialize the value of the memory location to which it points, as something like

  ```
  int *ip=5;
  ```

- Will only tell the compiler to use address 5 as the initial value for ip (and the contents of address 5 are undefined, and probably off-limits to your program anyway).

Pointer Declarations

- While the compiler thinks these are equivalent:

```
int *j;
int* j;
```

- The later leads to possible problems later, such as writing

```
int*  i,j;
```

when you wanted two pointers;

Pointer Arithmetic

- In addition to single variables, pointers can be used to access parts of an array

```
int *ip;
int a[20];
ip=&a[3];
```

- Given that ip points to element 3 of a, we can use pointer arithmetic to access elements before or after 3, as in

```
ip++;
*ip=7;
*(ip+1)=8;
*(ip-2)=3;
```

- Which sets element 4 to 7, element 5 to 8 and element 2 to 3.

String Operations Using Pointers

- mystrcmp() using pointers

```
char *p1=&str1[0], *p2=&str2[0];
while(1) {
  if (*p1!=*p2)
   return *p1-*p2;
  if (*p1==\0’|| *p2==‘\0’)
    return 0;
  p1++;
  p2++;
 }
```

- mystrcpy() using pointers

```
  char *dp= &dest[0], *sp=&src[0];
   while (*sp!=‘\0’)
     *dp++=*sp++;
   *dp=\0’;
```

Null Pointers

- A null pointer is a special value that is known to not point anywhere. Such a pointer is never valid.
- One way to get a null pointer is to use the constant NULL:

```
#include <stdio.h>
 int *ip = NULL;
```

- And then you can test the value of ip to see if it is a valid pointer, as in

```
if (ip!=NULL)
   printf("%d\n", *ip);
```

- NULL is implemented as a macro for the number 0, much like the null character '\0' is also the number 0, but is of type

```
#define NULL (void*) 0
```

Null Pointers

- Null pointers are useful as markers to say that the pointer is not ready for use, or for failure when you would otherwise return a valid pointer.
- For example, the strstr function returns a pointer to the first occurrence of one string within another string , but returns a null if not found.
- Also helps prevent the use of uninitialized pointers (e..g those with undefined values) which can cause unrepeatable problems.

Pointers and Arrays

- It turns out that pointes and arrays have much in common.

```
int a[10];
int *ip;
ip=a;
```

- It is as if we had written ip=&a[0];
- We can also use array subscription with pointers e.g.

```
ip[3]==*(ip+3)==a[3];
```

  is also valid and evaluates to true
- This is how the compiler lets us pass arrays as parameters!
  - A function call: myfunc(a,10) is actually myfunc(&a[0],10)
  - Similarly, the definition void myfunc(int array[], int size) is treated as if it had been void myfunc(int *array, int size) since later uses of array[x] are still permitted when array is a pointer.

Strings As Pointers

- Since arrays and pointers can be used interchangeably, it is common to and manipulate character pointers as strings
- This means
  - Any function declared to take a string (char array), will also accept a character pointer, since even if an array is passed, the function actually receives as pointers the first element of the array.
    printf %s actually expects a character pointer
  - Many programs extensively manipulate strings as character pointes and never explicitly declare any actual arrays

Strings As Pointers

- One caveat in initialization, however

  char string1[]="Hello 1";

  char *string2="Hello 2";

  string1[0]='J';

  string2[0]='J';

- The first assignment is fine; the second may crash! The first declaration created an array with the initial contents of "Hello 1". The second created a pointer to a string constant, which might be placed in an area that is read-only.

Pointers and Storage

- Pointers always point to something, even when not initialized. They are easy to use in this state:

```
int *x;
[..]
*x = 45;
```

- Therefore,
- It is essential to keep careful track of pointers, e.g. by using null pointers

```
int *x=NULL;
[…]
if (x)
   *x=45;
```

- This is particularly true when using dynamically allocated storage.

Static vs Dynamic Allocation

- "Static" allocation of space is straightforward, with variables or arrays declared locally or globally.

```
char myarray[1000];
char *ptr=0;
for (ptr=myarray; (ptr-myarray)< 1000;ptr++)
 // do something with each entry of myarray
```

- Dynamic allocation asks the OS for space for the pointer to access

```
#include <stdlib.h>
char *myarray=0, *ptr=0;
myarray=(char *)malloc(1000);  // ask for 1000 byte block
for (ptr=myarray; (ptr-myarray)<1000; ptr++)
 // do something with each entry of myarray
```

Why Dynamic Allocation?

- Dynamic allocation (e.g. with malloc) is
  - necessary when a function returns a pointer to a structure created by the function (not just passed in)

```
struct myobj *create_object(void);
```

  - useful when you don't know the size of the allocation at compile time

```
#include <stdio.h>
int numobjs;
int ret;
ret= scanf("%d", &numobjs);
// check return value for error
```

Why Dynamic Allocation?

- Dynamic Allocation (e.g. with malloc) is
  - Useful when you want to make a copy of a variable/array (esp without wasting extra space)

```
char *somestring=0, *copy=0;
copy=(char *) malloc(strleng(somestring)+1); /* +1 for '\0' */
strcpy(copy, somestring);
```

How much to allocate?

- One byte more than the string length if you are copying a string
- In general, you want the number of objects times the size of an object
- So if you were allocating an array of ints, rather than chars

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

- Sizeof() is a compile-time operator that determines the size of the data type passed as the parameter
- Note that just like an array, it is easy to have a pointer (or array index) exceed the allocated space (and cause problem)

Potential Problem

- Consider this function:

```
int myfunc(void) {
    int *ptr = (int *) malloc(10000*sizeof(int));
   if (!ptr)
     exit(-1);
   [some computation using space in ptr]
  return ptr[0];
}
```

- Important to realize that allocated space persists, even if no pointers point to it (C does not have automatic garbage collection)
  - Forgetting this leads to “memory leaks” causing your program to use more and more memory

Returning memory

- Eventually, the space allocated via malloc may no longer be useful to your program. Perhaps
  - The program is shutting down
  - The pointer to the data is going away
  - You want to be able to continue to allocate memory in the future

Returning memory

- Memory that has been malloc()ed is returned to the OS via free

```
#include <stdlib.h>
int *ip=(int *)malloc(1000*sizeof(int));
 …
free(ip);
ip=0;
```

- Note the null pointer assignment
- Note also that free() will succeed even if you
  - Free memory that was already free()d
  - Free random memory locations that were never malloc()ed

  But either one will eventually cause your program to crash.

Functions that return pointers

- When using routines that return pointers, you must determine who is responsible for the memory
  - Possibilities include
    - Pointer to a global value – memory never needs to be freed, but might get overwritten by later function calls (e.g, some networking code)
    - Pointer is to a dynamically-allocated local structure that must be destroyed or freed with another library call
    - Pointer to a dynamically-allocated block of memory that could later move or be freed independently of this pointer (e.g. strstr())
- In general, you need to know whether
    - You are now responsible to free() this memory
    - This memory might get overwritten in the future

User-Defined Structures

- We have previously mentioned structs as in

```
struct complex{
   double real;
   double imag;
};
```

- Which defined a new type struct complex
- We could also have declared some new variables of that type in two ways

```
struct complex {
   double real;
   double imag;
} var1, var2;
struct complex var3, var4;
```

Access to Structs

- We also saw that we could access contents using the period operator:

```
c1.real=c2.real+c3.real;
```

- Pointers to structs work as expected

```
struct complex *p1, *p2;
*p1 = *p2;
printf("%lf\n", (*p1).real);
```

- Parentheses are needed. Could instead use -> operator

```
printf("%lf\n",p1->real);
```

which is an equivalent (and common) shortcut

User-Defined Types

- A struct is a user-defined type
- The typedef operator also permits us to use an alternate name for a defined type.
  Thus,

```
typedef char *StringPtr;
typedef struct complex Complex;
StringPtr string;
Complex  c1, c2;
```

- Would be equivalent to

```
char *string;
struct complex c1, c2;
```

- We often define new type names
  - For convenience
  - To make code more self-documenting
  - To make it possible to change the actual base type used for a lot of variables without declarations of all those variables