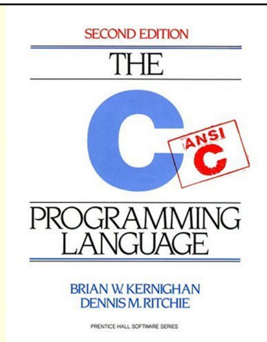

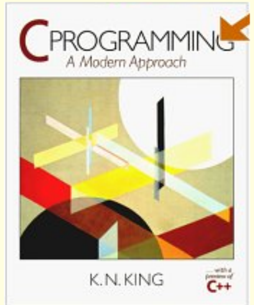

C-Programming

Part I: basics
prep. for Lab 8

Why C?

- See http://www.tiobe.com/tpci.htm

02/07 rank	02/06 rank	movement	Language	share	Δ in last year
1	1	=	Java	18.978%	-3.45%
2	2	=	C	16.104%	-2.23%
3	3	=	C++	10.768%	-0.53%
4	5	↑	PHP	8.847%	-0.07%
5	4	↓	(Visual) Basic	8.369%	-1.03%
6	6	=	Perl	6.073%	-0.63%
7	8	↑	Python	3.566%	+0.90%
8	7	↓	C#	3.189%	-0.78%
9	10	↑	JavaScript	2.982%	+1.47%
10	20	↑ 10	Ruby	2.528%	+2.12%

C How it Stacks Up

- As U can C, the C language (and its extensions/ derivatives) dominates the software community
  - Java also a strong showing
  - Python worth a peek
- Advantages of C:
  - compiled code runs FAST
  - allows low-level device control
  - a foundation of the programming world
- Disadvantages of C:
  - strings are a pain in the @$$
  - awkward conventions (pointers can be difficult to learn)
  - requires a compiler

What we will and won't do

- We will learn:
  - to write simple programs
  - basic interface
  - control flow, math, printing
  - data types
  - enough to be dangerous
- We won't learn:
  - advanced pointer operations
  - large projects (linking separate programs)
  - distinctions between public, private, external variables
  - enough to be really dangerous

C File Types

- Source Code
  - the stuff you type in: has `.c` extension
- Compiled "Executable"
  - the ready-to-run product: usually no extension in Unix, `.exe` in DOS
- Header Files
  - contain definitions of useful functions, constants: `.h` extension
- Object Files
  - a pre-linked compiled tidbit: `.o` in Unix, `.obj` in DOS
  - only if you're building in pieces and linking later

A typical (short) program

```
#include <stdio.h>

int main(void)
{
  int i=53;

  printf("The illustrious variable, i, is %d\n",i);

  return 0;
}
```

- Notes:
  - first include is so we have access to `printf` (standard I/O)
  - define the main program (must be called `main`) to take no arguments (thus `void`) and return an integer
  - braces surround the program
  - print value of integer, `i`, in formatted line
  - return zero (common return value for successful program)

More on program

```
#include <stdio.h>

int main(void)
{
  int i=53;

  printf("The illustrious variable, i, is %d\n",i);

  return 0;
}
```

- semicolons end each line within program
- spacing is not required, but makes for easier reading
- all variables must be declared before they are used
- could have simply said: `int i;` then declared later that `i=53;`
- the `\n` is a newline; the `%d` formats as decimal integer

Alternate form

```
#include <stdio.h>

int main(void)
{ int i; i=53; printf("i = %04d\n",i); return 0; }
```

- semicolons delimit separate statements, but this program, while compact, is harder on the eyes
- this time, we defined and assigned the variable in separate steps (more commonly done)
- we shortened the print statement fluff
- the format is now 4 characters wide, forcing leading zeros
  - output will be: `i = 0053`
- could compactify even more, if sadistic

Variable types

```
#include <stdio.h>

int main(void)
{
  char c;                    // single byte
  int i;                     // typical integer
  long j;                    // long integer
  float x;                   // floating point (single precision)
  double y;                  // double precision

  c = 'A';
  i = 356;
  j = 230948935;
  x = 3.14159265358979;
  y = 3.14159265358979;

  printf("c = %d = 0x%02x, i = %d, j = %ld, x = %f,
         y = %lf\n",c,c,i,j,x,y);
  c = i;
  i = 9259852835;
  printf("c = %d, i = %d, x = %.14f, y = %.14lf\n",c,i,x,y);

  return 0;
}
```


Output of previous program

- Output looks like:

```
c = 65 = 0x41, i = 356, j = 230948935, x = 3.141593, y = 3.141593
c = 100, i = 669918243, x = 3.14159274101257, y = 3.14159265358979
```

- Notes:
  - `c` “wrapped” around 256 when assigned to be 356
  - `i` couldn’t handle the large value, and also wrapped
    - `int` is actually the same as `long` on this machine
  - The `float` can’t handle the full precision set out
  - broke `printf` line: spacing irrelevant: semicolons do the work
  - The d, x, ld, f, and lf format codes correspond to decimal, hex, long decimal, float, and long float, respectively

Feeding data to the program

- Command line arguments allow the same program to be run repeatedly with different inputs (very handy)
- How to do it:
  - `main()` now takes arguments: traditionally `argc` and `argv[]`
  - `argc` is the number of command line arguments
    - minimum is one: the command itself
  - `argv[]` is an array of strings (words)
    - one for each of the space-separated blocks of text following the command on the command line
  - C arrays are numbered starting at zero
  - The command line entry: `one_ray -10.0 1.0 0.0` has:
    - `argc = 4`
    - `argv[0] = one_ray`; `argv[1] = -10.0`; etc.


```
#include <stdio.h>                    // for printf(),sscanf()
#include <stdlib.h>                   // for exit()

int main(int argc, char* argv[])
{
  int int_val;
  double dbl_val;

  if (argc > 2)
  {
    sscanf(argv[1],"%lf",&dbl_val);
    sscanf(argv[2],"%d",&int_val);
  }
  else
  {
    printf("usage: %s double_val int_val\n",argv[0]);
    exit(-1);
  }

  printf("Got double_val = %f; int_val = %d\n",dbl_val,int_val);

  return 0;
}
```

Result

- If I run simply `prog_name`, without arguments, I get:
  - `usage: prog_name double_val int_val`
  - normally, these would be given more descriptive names, like `initial_x_position` and `number_of_trials`
- If I run `prog_name 3.14 8`, I get:
  - `Got double_val = 3.140000; int_val = 8`
- Note that:
  - we needed a new header file for `exit()`
  - we are using `sscanf()` to scan a value into a variable
  - the `&` symbol before the variable name points to that variable's memory address so `sscanf` knows where to put the value
  - `printf` (and `sprintf`, `fprintf`, etc.) is forgiving about `%f` vs `%lf`, etc., but **not so** with scan functions (`scanf`, `sscanf`, `fscanf`, etc.)

For Loops

```
int k,count;

count = 0;
for (k=0; k < 10; k++)
{
  count += 1;
  count %= 4;
  printf ("count = %d\n",count);
}
```

- Notes:
  - declared more than one integer on same line (common practice)
  - `k` starts at zero, remains less than 10 (will stop at 9), increments by one each time through loop
    - `k++` adds one to variable: same as `k += 1;` same as `k = k + 1;`
  - adds one to `count` each time (see rule above)
  - "mods" `count` by 4 (remainder of `count/4`)
  - output is: 1, 2, 3, 0, 1, 2, 3, 0, 1, 2
  - could (and often do) use `k` as `int` value within loop
  - `for (;;)` is a way to get an indefinite loop (Ctrl-C to quit)

Math

```
#include <math.h>
…
double x,y,z,pi,ampl=3.0,sigma=1.2;

pi = 3.14159265358979;
x = sin(60.0*pi/180.0);
y = sqrt(fabs(2*x + pi));
z = ampl*exp(-0.5*pow(x/sigma,2.0))
```

- Notes:
  - Must include `math.h`
    - if compiling on linux/unix, use `-lm` flag to link math
  - note mixed assignment in variable declarations
  - `fabs` is "floating absolute value", and here keeps `sqrt` from getting a negative argument
    - otherwise result *could* generate `NaN` (Not a Number)
  - `pow(x,y)` raises x to the y power (x^y)

Math Warnings

- Number one mistake by C newbies: disrespecting variable type

```
int i,j=2,k=3;
double x,y,z;

i = 2/3;
x = 2/3;
y = 2/3.0;
z = 2.0/3;

printf("i = %d; x = %f; y = %f; z = %f; other = %f\n",i,x,y,z,j/k);

i = 0; x = 0.000000; y = 0.666667; z = 0.666667; other = 0.000000
```

  - `i` is an integer, so 2/3 truncates to zero
  - even though `x` is a double, 2/3 performs integer math, then converts to double
    - "`other`" value in `printf` shows same is true if `j/k` used
  - as long as one value is a float, it does floating-point math

Casting

- when necessary, one may "cast" a value into a type of your choice:
  - `(double) j` → 2.0
  - `((double) j)/k` → 0.666667
  - `j/((double) k)` → 0.666667
  - `(double) (j/k)` → 0.000000 (integer math already done)
  - `(int) 6.824786` → 6
- lesson is to take care when mixing variable types
- also, get into habit of putting .0 on floating point math numbers, even if strictly unnecessary

Talking to the Parallel Port in Windows

- We will use the inpout32.dll package
  - parallel port access in linux/unix is very straightforward
  - Windows 98 and before was also easy
  - new Hardware Abstraction Layer (HAL) gets in the way
  - this inpout32 package bridges the HAL
  - see www.logix4u.net to get the package (already installed on MHA-3574 machines)
  - http://www.hytherion.com/beattidp/comput/pport.htm for test programs
- Can also access via LPT file handle
  - discussed at end of lecture
  - runs 25 times slower than the inpout32 version
    - because you have to open/close the port all the time

Sample code (stripped down to fit on slide)

```
#include <stdio.h>
#include <conio.h>
#include <windows.h>
#define PPORT_BASE 0xD010     // usu. 0x378 if built-in
typedef void (_stdcall *oupfuncPtr)(short portaddr, short datum);
oupfuncPtr oup32fp;

void Out32(short portaddr, short datum){
  (oup32fp)(portaddr,datum);             }

int main(void)
{
  HINSTANCE hLib;
  short x=0xAA;               // value to write (expr. in hex)

  hLib = LoadLibrary("inpout32.dll");
  oup32fp = (oupfuncPtr) GetProcAddress(hLib, "Out32");

  Out32(PPORT_BASE,x);        // the actual output command

  FreeLibrary(hLib);
 return 0;
}
```


Looping to make a waveform

```
short outval = 0;
for (;;)              // way to make infinite loop: ^C kills
 {
   outval += 1;
   outval %= 256;
   Out32(PPORT_BASE,outval);
 }
```

- The code above makes a ramp of output values, then cuts down to zero and starts again
  - repeat until Ctrl-C kills it
- Each time:
  - the outval is increased by 1
    - statement equivalent to `outval = outval + 1`
  - then mod by 256 (256→0, and start over)
    - statement is equivalent to `outval = outval % 256`

How does it look In Linux/Unix?

```
#include <stdio.h>
#include <unistd.h>     // needed for ioperm()
#include <asm/io.h>     // for outb() and inb()

#define DATA 0x378      // parallel port memory address

int main()
{
   int x = 0xAA;

   if (ioperm(DATA,3,1))
   {
     printf("You must be root to run this program\n");
     exit(1);
   }

   outb(x,DATA);     // sends 1010 1010 to the Data Port

   return 0;
}
```

`outb()` performs direct write to hardware/memory address

LPT Method on Windows

```
#include <stdio.h>                // fprintf()...
#include <stdlib.h>               // exit()
#include <io.h>                   // open()...
#include <fcntl.h>                // O_xxx

int main()
{
  int out;
  char ch;

  out = open("LPT1:", O_WRONLY|O_BINARY);       // open parport
  if(out < 0)
  {
    printf("Can't open LPT1\n");
    exit(2);                      // exit with error status
  }
  ch = 0x55;
  write(out, &ch, 1);
  close(out);                     //flushes windows OS buffering

  return 0;                               // return success
}
```


Description

- Notes on previous program:
  - lots of includes to get all the features we want
  - open port as write only, binary mode
  - parallel port is assigned to an integer (`out`)
    - in essence, a temporary address for the program's use
  - checks that `out` is not negative (would indicate error)
    - aborts if so, with an error message
  - assigns a test value (hex 55) to the one-byte `ch`
    - 0x55 is a nice pattern: 01010101
    - 0xAA also nice: 10101010
  - writes this to the port
    - the `&` points to the location of the value to be sent
    - the 1 indicates one byte to be sent
  - closes port and exits
  - any looping must open and close port with each loop!!

References

- Lots of books on C out there
  - Kernighan and Ritchie is the classic standard
    - they wrote the language
  - the book by K. N. King is exceptionally good
- Also check the web for tutorials
  - C-geeks are exactly the kind of people who write web-pages, so there is a profusion of programming advice out there!