cs3157 lecture #5 notes.

mon 17 feb 2003

http://www.cs.columbia.edu/~cs3157

- news
  - homework #1 is due today
  - quiz #1 is on Wed Feb 19 in 833 Mudd (no labs)
- today's topics
  - advanced C programming
    - advanced data types
    - structured data types
    - functions
    - programs with multiple files
    - extras

advanced data types (1) — typedef.

- defining your own types using `typedef`

```
typedef short int smallNumber;
typedef unsigned char byte;
typedef char String[100];

smallNumber x;
byte b;
String name;
```


advanced data types (2) — typedef.

- defining your own boolean:

```
typedef char boolean;
#define FALSE 0
#define TRUE 1
```

- generally works, but beware:

```
check = x > 0;
if ( check == TRUE ) {...}
```

- if `x` is positive, `check` will be non-zero, but may not be == 1

advanced data types (3) — enum.

- define new integer-like types as *enumerated* types:

```
enum weather { rain, snow=2, sun=4 };

typedef enum {
  Red, Orange, Yellow, Green, Blue, Violet
} Color;
```

- look like C identifiers (names)
- are listed (enumerated) in definition
- treated like integers
  - start with 0 (unless you set value)
  - can add, subtract — e.g., `color + weather`
  - cannot print as symbol automatically (you have to write code to do the translation)

advanced data types (4) — enum.

- just fancy syntax for an ordered collection of integer constants:

```
typedef enum {
   Red, Orange, Yellow
} Color;
```

is like

```
#define Red 0
#define Orange 1
#define Yellow 2
```

- here's another way to define your own boolean:

```
typedef enum {False, True} boolean;
```


advanced data types (5) — data objects.

- C does not have Objects in the OOP sense (like Java and C++ do)
- but C has *data objects* — i.e., variables

```
short int x;
char ch;
float pi = 3.1415;
float f, g;
```

- scope
  - variables defined in `{ }` block are active only in block — e.g., *local*
  - variables defined outside a block are *global* (persist during program execution)
  - *static* variables may be declared outside a block, but are not globally visible

advanced data types (6) — data objects.

- variables must be declared before they are used
- we have used variables within `main()` and within functions
- *global variables*
  - are declared *outside* main() and outside any function, usually at the top of the program file, after any #'s (preprocessor directives)
  - can be "seen" anywhere
- *local variables*
  - are declared within a program block or function
  - they can only be seen inside the block in which they are defined
  - function arguments are local to the function they are passed to

advanced data types (7) — usage.

- a variable is conceptually a container that can hold a value
- default value is (mostly) undefined — you should treat it as a random number
- the compiler may warn you about uninitialized variables, but not as reliably as Java
- variables are always passed *by value*, but you can pass the *address* of a variable to a function:

```
scanf( "%d%f", &x, &f );
```

advanced data types (8) — sizes.

- every data object in C has:
  - a name and data type (specified in definition)
  - an address (its relative location in memory)
  - a size (number of bytes of memory it occupies)
  - visibility (which parts of program can refer to it)
  - lifetime (period during which it exists)
- Unlike scripting languages and Java, all C data objects have a fixed size over their lifetime
  - *except dynamically created objects*
- size of object is determined when object is created:
  - global data objects at compile time (data)
  - local data objects at run-time (stack)
  - dynamic data objects by programmer (heap)

structured data types (1).

- structured data types are available as:

object	**property**
`array [ ]`	enumerated; indexed from 0
`struct`	names and types of fields
`union`	made up of multiple elements, but only one exists at a time; each element could be a native data type, a pointer or a struct

structured data types (2) — arrays.

- "arrays" are defined by specifying an element type and number of elements
  - statically:

```
int vec[100];
char str[30];
float m[10][10];
```

  - dynamically:

```
int *dynvec, num_elements;
printf( "how many elements do you want to enter? " );
scanf( "%d", &num_elements );
dynvec = (int *)malloc( sizeof(int) * num_elements );
```

- for an array containing N elements, indeces are 0..N-1
- stored as a linear arrangement of elements
- often similar to pointers

structured data types (3) — arrays.

- C does not remember how large arrays are (i.e., no length attribute, unlike Java)
- given:

```
int x[10];
x[10] = 5;  /* error! */
```

- ERROR! because you have only defined x[0]..x[9] and the memory location where x[10] is can become something else...
- `sizeof x` gives the number of bytes in the array
- `sizeof x[0]` gives the number of bytes in one array element
- thus you can compute the length of `x` via:

```
int length_x = sizeof x / sizeof x[0];
```

- note that this does not work if `x` is defined as:

```
int *x;
```

since in this case `sizeof x` refers to the pointer only

structured data types (4) — arrays.

- when an array is passed as a parameter to a function:
  - the size information is not available inside the function
  - array size is typically passed as an additional parameter

```
printArray( x, length_x );
```

  - or globally

```
#define VECSIZE 10
int x[VECSIZE];
```

  - or as part of a struct (best practice; object-like)

```
typedef struct {
  int x[10];
  int length_x;
} Array;
Array ax;
ax.length_x = 10;
printArray( ax );
```


structured data types (5) — arrays.

- array elements are accessed using the same syntax as in Java: `array[index]`
- C does not check whether array index values are sensible (i.e., no bounds checking)
- e.g., `x[-1]` or `vec[10000]` will not generate a compiler warning!
- if you're lucky, the program crashes with

```
Segmentation fault (core dumped)
```

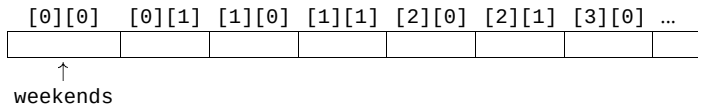
- C references arrays by the address of their first element:
  `array` is equivalent to `&array[0]`
- you can iterate through arrays using pointers as well as indexes:

```
int *v, *last;
int sum = 0;
last = &x[length_x-1];
for ( v = x; v <= last; v++ )
  sum += *v;
```


structured data types (7) — 2D arrays.

- 2-dimensional arrays

```
int weekends[52][2];
```



- you can use indices or pointer math to locate elements in the array
  - `weekends[0][1]`
  - `weekends+1`
- `weekends[2][1]` is same as `*(weekends+2*2+1)`, but NOT the same as `*weekends+2*2+1` (which is an integer)!

structured data types (8) — struct.

- `struct` is similar to a field in a Java object definition
- it's a way of grouping multiple data types together
- components can be any type (but not recursive)
- accessed using the same syntax struct.field

```
int main() {
  struct {
    int x;
    char y;
    float z;
  } rec;
  rec.x = 3;
  rec.y = 'a';
  rec.z = 3.1415;
  printf( "rec = %d %c %f\n",rec.x,rec.y,rec.z );
} /* end of main() */
```

structured data types (9) — struct.

- variables of `struct` types can be declared in two ways:
  - using a tag associated with the `struct` definition
  - wrapping the `struct` definition inside a `typedef`
- example:

```
int main() {
  struct record {
    int x;
    char y;
    float z;
  };
  struct record rec;
  rec.x = 3;
  rec.y = 'a';
  rec.z = 3.1415;
  printf( "rec = %d %c %f\n",rec.x,rec.y,rec.z );
} /* end of main()
```


structured data types (10) — struct.

- another example:

```
int main() {
  typedef struct {
    int x;
    char y;
    float z;
  } Record;
  Record rec;
  rec.x = 3;
  rec.y = 'a';
  rec.z = 3.1415;
  printf( "rec = %d %c %f\n",rec.x,rec.y,rec.z );
} /* end of main()
```


structured data types (11) — struct.

- overall size of `struct` is the sum of the elements, plus padding for alignment
- given previous 3 examples:
  `sizeof( rec )` $\rightarrow 12$
- but, it depends on the size and order of content (e.g., `int`s need to be aligned on word boundaries, since size of char is 1 and size of int is 4):

```
struct {
  char x;
  int y;
  char z;
} s1;
/*  x    y    z     */
/* |----|----|----| */
/* sizeof s1 -> 12  */
```

```
struct {
  char x, y;
  int z;
} s2;
/*  xy   z     */
/* |----|----| */
/* sizeof s2 -> 8  */
```


structured data types (12) — struct.

- pointers to structs are common — especially useful with functions (as arguments to functions or as function type)
- two notations for accessing elements: `(*sp).field` or `sp->field`
  (note: `*sp.field` doesn't work)

```
struct xyz {
 int x, y, z;
};
struct xyz s;
struct xyz *sp;
...
s.x = 1;
s.y = 2;
s.z = 3;
sp = &s;
(*sp).z = sp->x + sp->y;
```

structured data types (13) — extended example p1.

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

#define NAME_LEN 40

struct person {
  char  name[NAME_LEN+1];
  float height;
  struct { /* nested structure */
    int day;
    int month;
    int year;
  } birthday;
};

void printPerson( struct person * ); /* prototype */
```


structured data types (14) — extended example p2.

```
int main( void ) {
  struct person suzanne;    /* declare one */
  struct person class[120]; /* declare an array */
  /* store info in one */
  strcpy( suzanne.name,"suzanne" );
  suzanne.height = 60;
  suzanne.birthday.day = 16;
  suzanne.birthday.month = 5;
  suzanne.birthday.year = 1988;
  /* store info in the array */
  strcpy( class[0].name,"alex" );
  class[0].height = 48;
  class[0].birthday.day = 9;
  class[0].birthday.month = 5;
  class[0].birthday.year = 1995;
  strcpy( class[1].name,"jen" );
  class[1].height = 55;
```


structured data types (15) — extended example p3.

```
  class[1].birthday.day = 14;
  class[1].birthday.month = 4;
  class[1].birthday.year = 1992;
  /* print them... */
  printPerson( &suzanne );
  printPerson( &class[0] );
  printPerson( &class[1] );
} /* end of main() */

void printPerson( struct person *p ) {
  printf( "name   = [%s]\n",p->name );
  printf( "height = %5.2f inches\n",p->height );
  printf( "birthday = %02d/%02d/%4d\n",p->birthday.day,
           p->birthday.month,p->birthday.year );
}
```


structured data types (16) — union.

- `union`
- like struct:

```
union u_tag {
  int ival;
  float fval;
  char *sval;
} u;
```

- but only one of `ival`, `fval` and `sval` can be used in an instance of `u`
- overall size is largest of elements

functions (1).

- why?
  - useful if a program is too long
  - modularization — easier to code, debug, read, etc
  - promotes code reuse
- how?
  - passing arguments to functions
    - by value
    - by reference (pointer)
  - returning values from functions
    - by value
    - by reference (pointer)

functions (2).

- like methods in Java
- syntax:

```
<type> <name> ( <arguments> ) {
  <declarations>
  <statements>
} /* end of function */
```

- (replace all values in angle brackets with your definitions)
- the function name must be declared *before it is called*
- hence the use of function prototypes:
  - put this first, at the top of the program file:

    ```
    <type> <name> ( <arguments> );
    ```

  - then you can put the actual definition anywhere you want in the file

functions (3) — prototypes.

- prototypes are function header declarations and act similarly to Java interfaces
- here's a prototype to a function that is defined outside the file in which the prototype is (hence `extern`):

```
extern int putchar( int c );
```

- here's the function call:

```
putchar( 'A' );
```

- here's the function definition:

```
int putchar( int c ) {
  printf( "%c",c );
} /* end of putchar() */
```

- if defined *before* call in same file, then you don't need a prototype
- frequently, prototypes are defined in header (`.h`) file
- it's also a good idea to include the header file in the file where the actual definition resides — to ensure consistency

functions (4) — example.

```
#include <stdio.h>

/* function prototype */
void printN7( int n );

/* here's the main function */
int main( void ) {
  int n = 12345;
  printN7( n ); // function call
} // end of main()

/* function definition */
void printN7( int n ) {
  printf( "%d\n", n*7 );
} // end of printN7()
```

functions (5).

- `static` functions and variables hide themselves from those outside the file in which they are declared:

```
static int x;
static int times2( int c ) {
  return c*2;
}
```

- similar to `protected` class members in Java

functions (6).

- `const` keyword
- indicates that an argument won't be changed
- only meaningful for pointer arguments and declarations:

```
int myfunction( const char *s, const int x ) {
  const int VALUE = 10;
  printf( "x = %d\n", VALUE );
  return *s;
}
```

- if you attempt to change `*s` or `x` or `VALUE`, you'll get a compiler warning

functions (7) — variable number of arguments.

- "overloading" functions not allowed in C (like it is in Java)
- closest approximation is allowing variable number of arguments
- e.g., `printf()`
- prototype syntax:

```
int printf( const char *format, ... );
```

- for example, first call has 2 arguments, and second call has 4 arguments:

```
printf( "height = %5.2f inches\n",p->height );
printf( "birthday = %02d/%02d/%4d\n",p->birthday.day,
        p->birthday.month,p->birthday.year );
```


functions (8) — variable number of arguments.

```
#include <stdarg.h>
/* example: computes and returns product of its arguments */
double product( int number, ... ) {
  va_list list;
  double p;
  int i;
  va_start( list, number ); /* 2nd arg is the name of the last
                               parameter before the variable
                               argument list */
  p = 1.0;
  for ( i=0; i<number; i++ ) {
    p *= va_arg( list, double );
  }
  va_end( list );
  return p;
}
```

functions (9) — variable number of arguments.

- limitations:
  - need to copy to variables or local array
  - cannot access arguments in middle (unless you copy them first)
  - client and function need to know and adhere to type
- for more information: `unix$ man va_start`

programs with multiple files (1).

- file #1: hw.c

```
#include <stdio.h>   /* library header */
#include "mydefs.h"  /* my header */
int main( void ) {
  myfunction();      /* my function */
}
```

- file #2: mydefs.c

```
#include <stdio.h>
#include "mydefs.h"
void myfunction( void ) {
  mydata = 19;
}
```

- file #3: mydefs.h

```
void myfunction();  /* prototype */
int mydata;         /* global variable declaration */
```


programs with multiple files (2).

- the include file is automatically included at compile time
- but you need to link the files together:

```
unix$ gcc -c hw.c -o hw.o
unix$ gcc -c mydefs.c -o mydefs.o
unix$ gcc hw.o mydefs.o -o hw
```


extras (1) — copying strings.

- copying content vs. copying pointer to content

```
char s[1024];
char *t;
```

- saying `t = s;` copies the pointer, i.e., the address of `s` into `t`, so now they refer to the same address (memory location)
- use `strcpy( t,s );` to copy the *content* of `s` to `t`
- BUT make sure you have enough memory allocated for `t` to store all of `s`
- saying `s = "mydata";` is incorrect (though it may appear to work!)
- use `strcpy( s, "mydata");` instead

extras (2) — inside the string library.

- assumptions:

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

  – strings are NULL-terminated
  – all target arrays are large enough

- length function:

```
int strlen( const char *source );
```

  – returns number of characters in `source`, excluding NULL

- copying functions:

```
char *strcpy( char *dest, char *source );
```

  – copies characters from `source` array into `dest` array up to NULL

```
char *strncpy( char *dest, char *source, int num );
```

  – copies characters from `source` array into `dest` array; stops after `num` characters (if no NULL before that); appends NULL

extras (3) — inside the string library.

- search functions:

```
char *strchr( const char *source, const char ch );
```

  – returns pointer to first occurrence of `ch` in `source`; NULL if none

```
char *strstr( const char *source, const char *search );
```

  – return pointer to first occurrence of `search` in `source`

extras (4) — inside the string library.

- parsing function:

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

  – breaks string `s1` into a series of *tokens*, delimited by `s2`
  – called the first time with `s1` equal to the string you want to break up
  – called subsequent times with NULL as the first argument
  – each time is called, it returns the next token on the string
  – returns null when no more tokens remain

```
char inputline[1024];
char *name, *rank, *serial_num;
printf( "enter name+rank+serial number: " );
scanf( "%s", inputline );
name = strtok( inputline,"+" );
rank = strtok( null,"+" );
serial_num = strtok( null,"+" );
```


extras (5) — inside the string library.

- formatting functions — using internal buffers:

```
int sscanf(char *string, char *format, ...)
```

  – parse the contents of string according to format
  – placed the parsed items into 3rd, 4th, 5th, ... argument
  – return the number of successful conversions

```
int sprintf(char *buffer, char *format, ...)
```

  – produce a string formatted according to format
  – place this string into the buffer
  – the 3rd, 4th, 5th, ... arguments are formatted
  – return number of successful conversions

- format characters are like `printf` and `scanf` (see notes from earlier lectures)

extras (6) — character I/O.

- stdio functions:

```
int getchar()
```

  – read the next character from stdin; returns EOF if none

```
int fgetc(FILE *in)
```

  – read the next character from FILE in; returns EOF if none

```
int putchar(int c)
```

  – write the character `c` onto stdout; returns `c` or EOF

```
int fputc(int c, FILE *out)
```

  – write the character `c` onto out; returns `c` or EOF

extras (7) — line I/O.

- stdio functions:

```
char *fgets( char *buf, int size, FILE *in );
```

  – read the next line from in into buffer buf
  – halts at '\n' or after size-1 characters have been read
  – the '\n' is read, but not included in buf
  – returns pointer to strbuf if ok, NULL otherwise
  – do not use gets(char *) - buffer overflow

```
int fputs( const char *str, FILE *out );
```

  – writes the string str to out, stopping at '\0'
  – returns number of characters written or EOF

extras (8) — command-line arguments.

- how to get arguments from the unix command-line into the main program:

```
int main( int argc, char argv[] ) {
...
}
```

- `argc` is the argument count
- `argv` is the argument vector
  – array of strings with command-line arguments
  – `argv[0]` is the program executable name (unlike Java!)
  – `argv[1]` is the first argument

extras (9) — command-line arguments.

- example:
  – if you call:

```
unix$ myprogram a bc 123
```

  – then inside `main()` the values are:

```
argc ← 4
argv[0] ← "myprogram"
argv[1] ← "a"
argv[2] ← "bc"
argv[3] ← "123"
```

  – if you want to use them as numbers, you have to convert from string to numeric (just like in Java)!

```
int n;
sscanf( argv[3],"%d",&n );
```

extras (10) — void.

- `void` and `void *`
- function that doesn't return anything declared as `void`
- a function that takes no arguments has a `void` argument list, e.g.:
  `int main( void ) { ... }`
- the special pointer `void *` can point to anything:

```
#include <stdio.h>
extern void *f( void );

int main( void ) {
  f();
}

void *f( void ) {
  printf( "the big void\n" );
  return NULL;
}
```


extras (11) — function pointers.

- in this way you can define "function pointers"
- then you can "override" functions by leaving a prototype and changing the function based on the implementation
- syntax:

```
returnType (*ptrName)( arg1, arg2, ... );
```

- examples:

```
int (*fp)( double x );
```

  – is a pointer to a function that return an integer

```
double *(*gp)( int y );
```

  – is a pointer to a function that returns a pointer to a double

extras (12) — function pointers.

- a function that returns an integer:

```
int myfunction();
```

- a function that returns a pointer to an integer:

```
int *myfunction();
```

- a pointer to a function that returns an integer:

```
int (*myfunction)();
```

- a pointer to a function that returns a pointer to an integer:

```
int *(*myfunction)();
```


extras (13) — function pointers.

```
#include <stdio.h>

void myfunction( int d );
void mycaller( void (*f)(int), int arg );

int main( void ) {
  myfunction( 10 ); /* call myfunction with argument = 10 */
  mycaller( myfunction,10 ); /* do the same thing! */
}

void mycaller( void (*f)(int), int arg ) {
  (*f)( arg );
}

void myfunction( int d ) {
  printf( "d=%d\n",d );
}
```

extras (14) — printing escape sequences.

- start with a backslash (\)
- examples:

\n	new line
\t	tab
\a	alert (rings the bell)
\"	print a double quote (")
\\	print a backslash (\)

some last words on C...

- always initialize anything before using it (especially pointers)
- don't use pointers after freeing them
- don't return a function's local variables by reference
- there is no built-in exception handling! — so check for errors everywhere
- be CAREFUL about memory allocation
- Murphy's law, C version: anything that can't fail, will fail