today

wed 25 sep 2002

- homework #2 — was posted monday
- quiz #1 — today
  - 30-45 minutes long
  - one page of notes
  - topics: C
- structured data types (union)
- functions
- programs with multiple files
- extras (copying strings, string library, internal buffers, character I/O, line I/O, command-line arguments, void, function pointers, escape sequences)

structured data types — 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
```

programs with multiple files (3) — approximating data hiding.

- implementation defines real data structure (with “private” fields):

```
#define QUEUE_C
#include "queue.h"
typedef struct queue_t {
  struct queue_t *next;
  int data;
} *queue_t, queuestruct_t;
queue_t NewQueue(void) {
  return q;
}
```

- header file defines only “public” data:

```
#ifndef QUEUE_C
typedef struct queue_t *queue_t;
#endif
queue_t NewQueue(void);
```

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