CS113: Lecture 10

Topics:

- More data types
- Command line arguments
- Odds and ends

Struct variants

Structs are useful for grouping related data into records, but they may use more space for storage than is really necessary:

```
struct student_struct {
    char idnum; /* Only 50 students in the class */
    char year; /* Freshman, sophomore, etc. */
    char gender;
    char credit; /* Credit or audit? */
} student;
```

- We use 4 chars (32 bits) when we only need 7 bits for idnum, 2 bits for year, 1 bit for gender, 1 bit for credit (11 bits total).

- Bit fields (or packed structures) specify the width of the fields in a structure, forcing the program to use the minimum amount of space to store the struct, as constrained by memory alignment issues.

Bit fields

A more compact structure:

```
struct student_struct {
    unsigned idnum : 7;
    unsigned year : 2;
    unsigned gender : 1;
    unsigned credit : 1;
} student;
```

- Only `int` data types (`unsigned` or `signed`, but not `long`) may be used in the bit field.

- This structure will probably take up 16 bits, not 11, due to the need to align data types in appropriate memory addresses (usually multiples of 8 bits, e.g. 16, 32, 64).

- WARNING: Bit fields will save space, but access will probably be very slow. If you need compactness and speed, you will probably want to use bit shift operators on built-in data types instead.

Unions

Unions are special structs that *overlay* their contents in memory:

```
union example_union {
    double d; /* 8 bytes */
    char c[2]; /*2 bytes */
    int i; /* 4 bytes */
};
```

- A total of 8 bytes is used — the size of the largest component. `d` will take up 8 bytes, `c` will overlay the first two bytes of `d`, and `i` will overlay the first 4 bytes of `d` and therefore all of `c`.

- The union can act like any one of its component data types, but only one at a time, and the data stored are mutually exclusive.

- Often used when talking to device drivers and control over data alignment is important.

Union example

```
#include <stdio.h>

union example_union {
    double d; /* 8 bytes */
    char c[2]; /*2 bytes */
    int i; /* 4 bytes */
};

void main( void ) {
    union example_union U;

    U.i = 15; /* Now U acts like an int */
    printf("%d\n", U.i); /* prints 15 */

    U.c[0] = 'H'; U.c[1] = 'i'; /* U acts like char array */
    printf("%c%c\n", U.c[0], U.c[1]); /* prints Hi */

    U.d = 7.58930; /* U acts like a double;
                                    overwrites H,i */
    printf("%f\n", U.d); / prints 7.58930 */

    printf("%c%c\n", U.c[0], U.c[1]); /* prints ??? */
}
```

Command line arguments

Alternative version of main():

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

- Allows access to command line arguments (if invoked from UNIX command line, say)
- `argc` holds the number of command line arguments
- `argv` is an array of strings; the strings are the arguments passed to the program on the command line, starting with the program name, e.g.

  prompt% myprogram xyz bbc -5

  Then `argc` is 4, `argv[0]` is the string `"myprogram"`, `argv[1]` is `"xyz"`, `argv[2]` is `"bbc"`, and `argv[3]` is`"-5"`. Each of the strings is null terminated.
- Use the facilities in `getopt.h` to process standard format command line arguments if you're a UNIX programmer (this is not part of standard C).

More about the ternary ? : operator

- Recall: A type of conditional expression. Form:

  `test ? expr1 : expr2`

  `test` is evaluated first. If it is non-zero ("true"), then `expr1` is evaluated, and the entire expression has value `expr1`. Otherwise, `expr2` is evaluated, and the entire expression has value `expr2`.

- Example. Instead of

```
if( a > b )
   z = a;
else
   z = b;
```

  We can write

```
z = ( a > b ) ? a : b;
```

- A little trick...

  Can something like the following be done (without duplicating `complex_expression`)?

```
((condition) ? a : b ) = complex_expression;
```

  Yes!

```
*((condition) ? &a : &b ) = complex_expression;
```

The comma operator

- Form: `expr1, expr2`.
- Most common use: in `for` loop.

```
void reverse( char *s )
{
   int temp, i, j, len;
   len = strlen( s );
   for( i = 0, j = len - 1; i < j; i++, j-- )
   {
      temp = s[i];
      s[i] = s[j];
      s[j] = temp;
   }
}
```

- Evaluated left-to-right. All side-effects resulting from evaluation of left expression are completed before right expression evaluated.
- Type and value of the result are the type and value of the right operand.
- Example:

```
int a = 3, b = 6, c;
c = (a++, (b++) + a);
printf( "a is %d, b is %d, c is %d.\n", a, b, c );
```

(Prints 4, 7, 10.)

Note on array notation

- Does the seemingly insane expression `5["0abcdefgh"]` make sense?

- Yes, it does! Array subscripting in C is "commutative", i.e., `a[e]` is identical to `*((a) + (e))` for any two expressions `a` and `e`.

- Thus, the following are all equal.

```
a[e]
*((a) + (e))
*((e) + (a))
e[a]
```

- ...and `5["0abcdefgh"]` is equal to `"0abcdefgh"[5]`, which is `'e'`.

Loop Unrolling

Which is faster?

```
for( i = 0; i < 8 * n; i++ )
{
   a[i] = i;
}

for( i = 0; i < n; i += 8 )
{
   a[i] = i;
   a[i+1] = i+1;
   a[i+2] = i+2;
   a[i+3] = i+3;
   a[i+4] = i+4;
   a[i+5] = i+5;
   a[i+6] = i+6;
   a[i+7] = i+7;
}
```

Tom Duff (while at Lucasfilm) wanted to copy chunks memory, *quickly*. Original code:

```
send(to, from, count)
register short *to, *from;
register count;
{
   do
      *to = *from++;
   while(--count>0);
}
```

Duff's Device

"Many people (even bwk?) have said that the worst feature of C is that switches don't break automatically before each case label. This code forms some sort of argument in that debate, but I'm not sure whether it's for or against."

– Tom Duff

```
send(to, from, count)
register short *to, *from;
register count;
{
   register n=(count+7)/8;
   switch(count%8){
      case 0: do{     *to = *from++;
      case 7:         *to = *from++;
      case 6:         *to = *from++;
      case 5:         *to = *from++;
      case 4:         *to = *from++;
      case 3:         *to = *from++;
      case 2:         *to = *from++;
      case 1:         *to = *from++;
              }while(--n>0);
   }
}
```

Curiosity: A self-reproducing program

Note that 34 is the ASCII value of the double-quote character.

```
char*s="char*s=%c%s%c;main(){printf(s,34,s,34);}";
main(){printf(s,34,s,34);}
```

(There should be no carriage return in the middle of the program; one was inserted for the sake of formatting.)

Known as a "quine" after logician and philosopher of language Willard Van Ormand Quine, who studied (among other things) indirect self-reference.

Think about the phrase "yields falsehood when appended to its own quotation". True or false?