University of Arizona

CSc 340

Foundations of Computer Systems

Christian Collberg

April 3, 2001

C Misc.

Bit Fields

- C allows a structure to have a field that is smaller than a byte. This is called a bit field, and looks like a normal field declaration except the number of bits is specified:

```
struct {
   unsigned int x:3;
   unsigned int y:2;
   unsigned int z:1;
} foo;
```

- The structure `foo` has three fields, `x`, `y`, and `z`, of size 3,2, and 1 bits, respectively. The entire structure is smaller than a byte.

Short-circuited Expressions

- C expressions are short-circuited — as soon as it is determined that the expression cannot be true, the rest of it isn't evaluated.
- The '0' ensures that the expression cannot be true, regardless of the value of x. The value of x is never tested.

```
if (0 && (x > 10)) {
      ...
```


Short-circuited Expressions...

- If ptr is `NULL`, the expression cannot be true so the value of `ptr->value` is never tested. This useful because this test will cause the program to crash if `ptr` is indeed `NULL`.

```
if ((ptr != NULL) && (ptr->value > 10))\
       ...
```

Slide 19–4

Conditional Expressions

- C provides a shortcut for expressions whose value depends on the result of a conditional. This is called a conditional expression, and an example is:

```
if (a > b) {
    c = b;
} else {
    c = a;
}
```

This code fragment sets `c` to the minimum of `a` and `b`. This can also be written:

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

Slide 19–5

Conditional Expressions...

- Conditional expressions have the form

```
test ? true : false.
```

If the expression test is non-zero, the expression true is evaluated and its result used as the result of the conditional expression, otherwise the expression false is evaluated and its result used.

Slide 19–6

Function Pointers

- In addition to pointer variables that contain the addresses of other variables, C also allows pointers to contain the addresses of functions. For example,

```
void   (*func)(int foo);
```

declares a variable named `func` that is a pointer to a function that has one integer parameter, and returns no value.

Slide 19–7

Function Pointers...

- The function to which a function pointer points can be invoked using the '`*`' operator:

```
/* 1 */   void foo (void) {printf("hello\n");}
/* 2 */   void main (void) {
/* 3 */       void (*ptr)(void);
/* 4 */       ptr = foo;
/* 5 */       (*ptr)();
/* 6 */   }
```

At `3`, the pointer variable `ptr` is declared. At `4`, `ptr` points to `foo`. At `5`, we invoke the function to which `ptr` points (`foo`). This program prints `"hello"`.

Slide 19–8

Function Pointers. . .

- Function pointers are a very powerful and useful mechanism.
- The C library routine qsort uses the quicksort algorithm to sort the elements of an array. The elements of the array can be of any type, and since C doesn't handle this sort of polymorphism very well, the function prototype for qsort is:

```
void qsort(void *base, size_t nel, size_t width,
int (*compar) (void *, void *));
```

Slide 19–9

Function Pointers. . .

- `base` is the address of the array, `nel` is the number of elements in the array, `width` is the size of each element, in bytes, and `compar` is a pointer to a function that compares two elements.
- The comparison function returns a negative number if the element pointed to by the first parameter is less than the second, 0 if they are equal, and a positive number if the first is greater than the second.
- You write the comparison routine to compare elements of whatever type you like, (e.g. `element_compare`) and pass it to `qsort`. `qsort` can then sort the array.

Slide 19–10

Disassembler with Function Pointers

```
void class1(char* name, int x) {
   printf("\t%s\t%s\n",name, reg(x,11));
}

typedef void (*decodeFun)(char*,int);

decodeFun decode[] = {
   &class0,&class1,&class2,&class3,···};

void disassemble(int x) {
   ··· decode[class[i]](instr[i],x); ···
}
```

Slide 19–11

Function Pointers. . .

- Another example use of function pointers is to implement keyboard shortcuts.
- Each shortcut is represented by a structure that has a field containing the characters in the shortcut, and a pointer to a function to be invoked when the user types the shortcut. This arrangement makes it easy to add shortcuts and change what a shortcut does by changing the function pointer.

Function Pointers. . .

- Another example use of function pointers is to implement objects, as in Java.
- An object in C can be represented as a structure with fields for variables, and function pointers to the functions that implement the object's methods. The function pointers can be changed to overload the methods.

string.h

```
void *memcpy(void *, const void *, size_t);
void *memmove(void *, const void *, size_t);
char *strcpy(char *, const char *);
char *strncpy(char *, const char *, size_t);

char *strcat(char *, const char *);
char *strncat(char *, const char *, size_t);

int memcmp(const void *, const void *, size_t);
int strcmp(const char *, const char *);
int strcoll(const char *, const char *);
int strncmp(const char *, const char *, size_t);
size_t strxfrm(char *, const char *, size_t);
```


string.h. . .

```
void *memchr(const void *, int, size_t);
char *strchr(const char *, int);
size_t strcspn(const char *, const char *);
char *strpbrk(const char *, const char *);
char *strrchr(const char *, int);
size_t strspn(const char *, const char *);
char *strstr(const char *, const char *);
char *strtok(char *, const char *);
char *strtok_r(char *, const char *, char **);
void *memset(void *, int, size_t);
```


stdlib.h

```
extern double atof(const char *);
extern int atoi(const char *);
extern long int atol(const char *);

extern void abort(void);
extern int atexit(void (*)(void));
extern void exit(int);
extern char *getenv(const char *);
extern int system(const char *);

extern void *bsearch(const void *, const void *, size_t, size_t,
int (*)(const void *, const void *));
extern void qsort(void *, size_t, size_t,
int (*)(const void *, const void *));
```

Slide 19–16

stdlib.h. . .

```
extern int abs(int);
extern div_t div(int, int);
extern long int labs(long);
extern ldiv_t ldiv(long, long);

extern double drand48(void);
```

Slide 19–17

Sort and Search Example

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

struct critter {char *name; char *species;};

struct critter muppets[] = {
  {"Kermit", "frog"}, {"Piggy", "pig"},
  {"Gonzo", "whatever"}, {"Fozzie", "bear"},
  {"Sam", "eagle"},  {"Robin", "frog"},
  {"Animal", "animal"},  {"Camilla", "chicken"},
  {"Sweetums", "monster"},  {"Dr. Strangepork", "pig"},
  {"Link Hogthrob", "pig"},  {"Zoot", "human"},
  {"Dr. Bunsen Honeydew", "human"},  {"Beaker", "human"},
  {"Swedish Chef", "human"}};
```

Slide 19–18

```
int count = sizeof (muppets) / sizeof (struct critter);

int critter_cmp (const struct critter *c1,
                 const struct critter *c2) {
  return strcmp (c1->name, c2->name);
}

void print_critter (const struct critter *c) {
  printf ("%s, the %s\n", c->name, c->species);}

void find_critter (char *name) {
  struct critter target, *result; target.name = name;
  result = bsearch (&target, muppets, count,
                    sizeof (struct critter), critter_cmp);
  if (result)  print_critter (result);
  else         printf ("Couldn't find %s.\n", name);
}
```

Slide 19–19

```
int main (void) {
  int i;

  for (i = 0; i < count; i++) print_critter (&muppets[i]);
  printf ("\n");

  qsort (muppets, count, sizeof (struct critter), critter_cmp);

  for (i = 0; i < count; i++) print_critter (&muppets[i]);
  printf ("\n");

  find_critter ("Kermit");
  find_critter ("Gonzo");
  find_critter ("Janice");
}
```