Everything Else

C Programming and Software Tools

N.C. State Department of Computer Science

Computer Science
NC STATE UNIVERSITY

BOOLEANS

Booleans

- In C99, bools are included! Sort of!

```
_Bool b = 1;
```

- Why so ugly? To not conflict with existing code.
- But if you want "nice" bools plus "true" and "false", ask for them by name:

```
#include <stdbool.h>
bool b = true;
bool c = false;
```

stdbool.h

```
#ifndef _STDBOOL_H
#define _STDBOOL_H

#ifndef __cplusplus

#define bool    _Bool
#define true    1
#define false   0

#else /* __cplusplus */

/* Supporting <stdbool.h> in C++ is a GCC extension.  */
#define _Bool   bool
#define bool    bool
#define false   false
#define true    true

#endif /* __cplusplus */

/* Signal that all the definitions are present.  */
#define __bool_true_false_are_defined   1

#endif  /* stdbool.h */
```

CONST & CONST POINTERS

Const pointer - summary

- Commonly used in argument/return types:

```
char *strcpy(char *dest, const char *src);
```

The `const` Keyword...

Indicates to the compiler that a value should not change during program execution

- should be initialized, but not changed

```
const int twopowfive = 32;
const float pi = 3.14159;

twopowfiv = 64; /* ERROR */
pi = 6.3; /* ERROR */
```

… (cont'd)

Is this better than macros?

```
#define TWOPOWFIV 32
#define PI 3.14159
```

Derived types can be **`const`** also

```
struct pet {
    char *name;
    unsigned short weight;
    unsigned char age;
    unsigned char type;
};
const struct pet mypet =
      { "Fluffy", 30, 5, DOG };
```

`const` and Pointers…

Is it the pointer that cannot be changed, or the thing it points at?

Changeable pointer to changeable character:

```
char * cp = &c;
*cp++ = 'A';  /* no problems */
```

Constant pointer to changeable character

```
char * const cp = &c;
*cp = 'Q';  /* No problems */
cp = &d ;   /* ERROR, changes pointer */
```

… (cont'd)

Changeable pointer to constant character

```
const char * cp = &c;
*cp = 'Z' ; /* ERROR, changes value
             * pointed to */
c = 'Z';    /* But this is OK! */
cp = &d;    /* No problems */
```

Constant pointer to constant character

```
const char * const cp = &c;
*cp++ = 'Z' ; /* ERROR, changes both */
```

Considered good practice; use whenever possible (particularly pointers passed to functions)

Commonly used in argument/return types:

```
char *strcpy(char *dest, const char *src);
```

ENUMS:
THE ENUMERATED DATA TYPE

Enumerated Data Type…

- Use for variables with small set of possible values, where actual encoding of value is unimportant

```
enum colors {red, blue, green, white, black};
enum colors mycolor;

mycolor = blue;
...
if ((mycolor == blue) || (mycolor == green))
   printf("cool color\n");
```

… (cont'd)

- Don't compare variables of different enumerated types - results not what you expect!

```
enum {blue, red, green, white, black}
  primarycolor;
enum {black, brown, orange, yellow}
  halloweencolor;

primarycolor = black;
halloweencolor = black;
if (primarycolor == halloweencolor)
   printf("Same color\n");
```

What will print?

Although you can interpret enumerated data types as integers, I don't recommend it

… (cont'd)

Compared to macros…?

```
#define BLUE 0
#define RED 1
#define GREEN 2
#define WHITE 3
#define BLACK 4

int primarycolor;
primarycolor = RED;
…
if (primarycolor == RED) …
```

GNOME: *"If you have a list of possible values for a variable, do not use macros for them; use an enum instead and give it a type name"*

TYPEDEF

Typedef

- Make an alias for a type:

```
typedef unsigned char byte;
typedef int* int_pointer;

byte x = 5;
int q = 12;
int_pointer pq = &q;
```

Typedef structs (1)

- Commonly used with structs:

```
typedef struct {
   char name[64];
   int age;
} Person;

Person bob = {"Bob", 65};
struct Person sue;
```

No such type!

Typedef structs (2)

- Sometimes you need it to be a named struct too, though…

```
typedef struct Node {
  int id;
  struct Node *next;
} Node;
```

☺

```
typedef struct Node {
  int id;
  Node *next;
} Node;
```

☹

Node is undefined at this time!

Typedef structs (3)

- It's common to typedef a pointer to a struct to make a "class-like thingy":

```
struct Person {
  char name[64];
  int age;
};
typedef struct Person* Person;

Person create_person(char* name, int age)
{
    …
}
```

```
Person bob = create_person("Bob",65);
```

Typedef arrays?

Even arrays can be **typedef**s

```
typedef int values[20];
values tbl1, tbl2;  /* two arrays, each with
                     * 20 ints */
```

- **typedef**s help make programs portable
  - to retarget a program for a different architecture, just redefine the typedefs and recompile
- Usually, **typedef**s are collected in a header file that is **#include**'d in all source code modules

UNIONS

The `union` Statement

- Defined like a `struct`, but only stores exactly one of the named members
  - motivation: use less memory
- Nothing in the `union` tells you which member is stored there!
  - usually, another variable indicates what is stored in the `union`

union Example

```
/* animal can have only one of the following */
union properties {
  unsigned short speed_of_flight;      // bird
  bool is_freshwater;                  // fish
  enum {VERY, SOME, NONE} hairiness;   // mammal
};

struct {
  unsigned char type;
  char * name;
  union properties info;
} animals[10];

animals[0].type = MAMMAL;
animals[0].name = "Polar Bear";
animals[0].info.hairiness = VERY;
```

Unions can decompose types

(Like a pointer cast without the pointer, or the cast)

```
union flippable_int {
        unsigned char bytes[4];
        int value;
};

int main() {
        union flippable_int x = { .value = 100000 };
        printf("value = %d (0x%08x)\n", x.value, x.value);
        printf("bytes = {%02x,%02x,%02x,%02x}\n",
                  x.bytes[0], x.bytes[1], x.bytes[2], x.bytes[3]);

        // convert to big endian
        unsigned char t;
        t = x.bytes[0]; x.bytes[0] = x.bytes[3]; x.bytes[3] = t;

        t = x.bytes[1]; x.bytes[1] = x.bytes[2]; x.bytes[2] = t;

        printf("value = %d (0x%08x)\n", x.value, x.value);
        printf("bytes = {%02x,%02x,%02x,%02x}\n",
                  x.bytes[0], x.bytes[1], x.bytes[2], x.bytes[3]);

}
```

```
value = 100000 (0x000186a0)
bytes = {a0,86,01,00}
value = -1601830656 (0xa0860100)
bytes = {00,01,86,a0}
```

VARIABLE NUMBER OF ARGUMENTS

Functions with a Variable Number of Arguments…

- Example: **`printf(char *fmt, …)`**
  - the first argument (**`char *fmt,`** the *named argument*) indicates how many, and what type, of unnamed arguments to expect
  - the **`...`** (the *unnamed arguments*) stands for an arbitrary list of arguments provided by the calling program

… (cont'd)

- Requires macros defined in **`<stdarg.h>`**
- In function f():
  1. Declare a variable of type **`va_list`**
  2. Call **`va_start`**; returns pointer to the first unnamed argument
  3. Call **`va_arg`** to return pointer to each successive unnamed argument
  4. Call **`va_end`** to end processing

… (cont'd)

- How many unnamed parameters?
  - this has to be indicated by the named parameter
- What are types of unnamed parameters?
  - either this is fixed (implicit), or the named parameter must explicitly indicate
  - example: the **`printf()`** format specifier

Example…

- A function **`sumup(num, …)`** which returns the sum of a list of **`num`** arguments, all of type **`int`**
- Calling **`sumup()`**:

```
#include <stdio.h>
#include <stdarg.h>
int sumup(int, …);

int main(void)
{
    int i = 295, j = 3, k = 450, res;
    res = sumup(3, i, j, k);
    …
}
```

Number of unnamed arguments

List of unnamed arguments

… (cont'd)

- Definition of **`sumup()`**:

```
int sumup(int num, …) {
    int sum;
    va_list ap;

    va_start(ap, num);
    sum = 0;
    for(int i = 0; i < num; i++)
        sum += va_arg(ap, int);

    va_end(ap);
    return sum;
}
```

Declare pointer to arguments

Makes ap point to first unnamed argument

Read unnamed arguments, all of type `int`

Clean up before exiting

Another Example…

- Function **`sumup(char *fmt, …)`**, where **`fmt`** specifies type and number of unnamed arguments
  - one character per unnamed argument
  - types = ‘i’ (**`int`**), ‘d’ (**`double`**), and ‘c’ (**`char`**)
  - Ex.: if **`fmt[]`** equals **`“iddic”`** ⇒
    there are 5 unnamed arguments,
    first and fourth are type **`int`**,
    second and third are type **`double`**,
    fifth is type **`char`**

```
float sumup(char *fmt, …);
…
    float res;
    res = sumup(“cid”, (char) ‘Q’, 2500, 3.141);
```

… (cont'd)

```
float sumup(char *fmt, …) {
    int i;
    float sum = 0, d;
    char c;
    va_list ap;
    va_start(ap, fmt);
    for(; *fmt != '\0'; fmt++)
        if (*fmt == 'c')
            sum += va_arg(ap, char));
        else if (*fmt == 'i')
            sum += va_arg(ap, int));
        else if (*fmt == 'd')
            sum += va_arg(ap, double));
    va_end(ap);
    return sum;
}
```

ENVIRONMENT VARIABLES

Environmental Variables

- A way for a user to customize the execution environment of programs
- Ex.:

```
cmd> echo $HOME
/home/jerry
cmd> HOME=/home/linda
cmd> echo $HOME
/home/linda
```

Common environment variables:

TERM
SHELL
USER
PATH
HOME
MAIL
GROUP
LANG
EDITOR
PRINTER

Reading / Writing E.V.'s in C

Read using **`getenv()`** (**`#include <stdlib.h>`**)

```
char *string = getenv("HOME");
printf("$HOME=%s\n", string);
```

And **`setenv()`** if you want to change them

```
setenv("HOME", "/home/new", 1);
```

BIT FIELDS

8. Bit Fields in C

- Way to pack bits into a single word; useful?
- Bit fields of a word are defined like members of a structure

Bit Fields Example... (http://www.cs.cf.ac.uk/Dave/C/)

- Frequently devices and OS communicate by means of a single word

```
struct Disk_register  {
      unsigned ready:1;
      unsigned error_occurred:1;
      unsigned disk_spinning:1;
      unsigned write_protect:1;
      unsigned head_loaded:1;
      unsigned error_code:8;
      unsigned track:9;
      unsigned sector:5;
      unsigned command:5;
};
```

…(cont'd)

```
struct Disk_register * dr =
       (struct Disk_register * ) MEMADDR;

/* Define sector and track to start read */
dr->sector = new_sector;
dr->track = new_track;
dr->command = READ;

/* ready will be true when done, else wait */
while ( ! dr->ready ) ;

if (dr->error_occurred) /* check for errors */
  {
    switch (dr->error_code)
    ......
  }
```

Warnings About Bit Fields

- Recommendation: always make bit fields unsigned
- # of bits determines maximum value
- Restrictions
  1. no arrays of bit fields
- Danger: files written using bit-fields are non-portable!
  - order in which bit-fields stored within a word is system dependent

Any Questions?