The Rest of C

ITCS 2116: C Programming

College of Computing and Informatics

Department of Computer Science

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)

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 freshwater_or_saltwater;       // 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;
```

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;
}
```

Environmental Variables

- A way for users to customize execution environment of programs
- Example:

```
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 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
  2. **no pointers** to a bit field
- Danger: files written using bit-fields are **non-portable!**
  - order in which bit-fields stored within
  - a word is **system dependent**

“Bit Twiddling”

- C has operators that treat operands simply as sequences of bits

- Question: Why do bit level operations in C (or any language)?
- Answer #1: lets you **pack** information as efficiently as possible
- Answer #2: some processing is faster to implement with bit-level operations than with arithmetic operators

“Bit Twiddling”... (cont’d)

- Ex: **image processing**
  - pack 64 B&W pixel values into a single **`long long`** operand, and process 64 pixels with one instruction
  - mask one image with another to create overlays
- Other applications:
  - **data compression,**
  - **encryption**
  - **error correction**
  - **I/O device control**
  - **...**

Working in Binary With C?

- There is **no standard way** to…
  - …write a constant in binary

```
i = 01011011;
```

☠ *common source of bugs* ☠
thinking sequence of 1's and 0's means base 2

  - …input an ASCII-encoded binary string and convert to an integer

```
scanf("%b", &i);
```

  - …output an integer as an ASCII-encoded binary string

```
printf("%b", i);
```

- Alternatives?
  - **Use octal or hexadecimal representation**

BitOps: One Operand

- Bit-wise complement **(~)**
  - operand must be integer type
  - result is ones-complement of operand (flip **every** bit)
  - Example:

```
  ~0x0d    // (binary 00001101)
== 0xf2    // (binary 11110010)
```

Not the same as Logical NOT **(!)** or sign change **(-)**

```
char i, j1, j2, j3;
i = 0x0d;     // binary 00001101
j1 = ~i;      // binary 11110010
j2 = -i;      // binary 11110011
j3 = !i;      // binary 00000000
```

BitOps: Two Operands

- Operate **bit-by-bit** on operands to produce a result operand of the same length
- And **(&):** result 1 if both inputs 1, 0 otherwise
- Or **(|):** result 1 if either input 1, 0 otherwise
- Xor **(^):** result 1 if one input 1, but not both, 0 otherwise
- Operands **must** be of type integer

Two Operands... (cont'd)

- Examples

```
00 111 000
&
11 011 110
-----------
00 011 000
```

```
00 111 000
|
11 011 110
-----------
11 111 110
```

```
00 111 000
^
11 011 110
-----------
11 100 110
```

Differences: Logical and Bit Ops

Results?

```
int a, b, c,
    d, e, f;

int i = 30;
int j = 0;
a = i && j;
b = !j;
c = !i;

float x = 30.0;
float y = 0.0;
d = x || y;
e = !y;
f = !x;
```

☠ *common source of bugs* ☠

difference between logical and bit-level operators

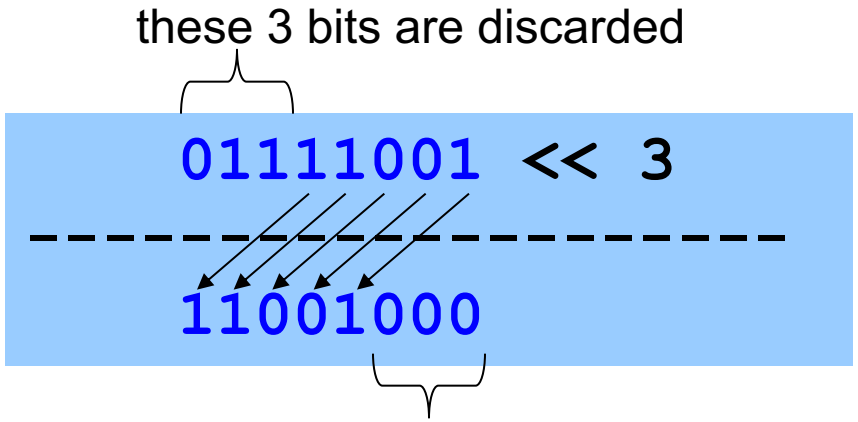
Difference? Problems?

```
int a, b, c,
    d, e, f;

int i = 30;
int j = 0;
a = i & j;
b = ~j;
c = ~i;

float x = 30.0;
float y = 0.0;
d = x | y;
e = ~y;
f = ~x;
```

Shift Operations

- **`x << y`** is left (**logical**) shift of **`x`** by **`y`** positions
  - **`x`** and **`y`** must both be integers
  - **`x`** **should** be unsigned or positive
  - 0 <= **`y`** <= number of bits in **`x`**
  - **`y`** leftmost bits of **`x`** are discarded
  - zero fill **`y`** bits on the right

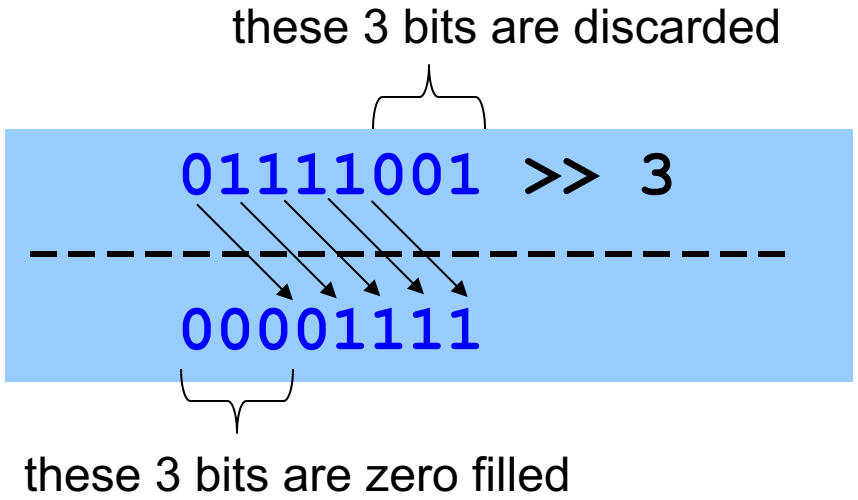
☠ *common source of bugs* ☠
logical shifts on negative numbers



ShiftOps... (cont'd)

- `x >> y` is right (**logical**) shift of `x` by `y` positions
  - `y` rightmost bits of `x` are discarded
  - zero fill `y` bits on the left

☠ *common source of bugs* ☠
logical shifts on negative numbers



ShiftOps... (cont'd)

- It is occasionally useful to know that...
  - right logical shift of an unsigned number x by y positions is equivalent to dividing x by 2^y
  - left logical shift of an unsigned number x by y positions is equivalent to multiplying x by 2^y

```
unsigned char j, k, m;
j = 121;
k = j << 3
m = j >> 3;
printf("%d %d %d\n", j, k, m);
```

Other Useful Bit Operations

- Complementing, Anding, Oring, and Xoring bits are all provided directly by C operators
- What about the following?
  - clearing all or selected bits to 0's, or setting all or selected bits to 1's
  - testing if all or selected bits are 0's, or 1's
  - counting the number of bits that are 0's, or that are 1's
  - copying all or selected bits from x to y
  - copying a bit or bits from position i of x to position j of y

Clearing Bits to 0's

- Using C operators:
  - & with 0 will clear, & with 1 means "no change"
- So, create a mask with 0's where you want to clear, and 1's everywhere else

If input is…	And mask is…	Then input & mask =
0	0	0 (no change)
0	1	0 (no change)
1	0	0 (clear)
1	1	1 (no change)

Clearing… (cont'd)

- How would you clear (to **0**) all the bits in a **`char`**?

```
unsigned char m = 0x00;
a = a & m;
```

```
a: 00 111 011
&
m: 00 000 000
-------------

a: 00 000 000
```

- How would you clear the **right** two bits (without changing the **other** bits)?

```
unsigned char m = 0374;
a = a & m;
```

```
a: 00 111 011
&
m: 11 111 100
-------------

a: 00 111 000
```

Setting Bits to 1's

- Using C operators:
  - | with 1 will set, | with 0 means "no change"
- So, create a mask with 1's where you want to set, and 0's everywhere else

If input is...	And mask is...	Then input | mask =
0	0	0 (no change)
0	1	1 (set)
1	0	1 (no change)
1	1	1 (no change)

Setting… (cont'd)

- How would you set (to **1**) **all** the bits in a **`char`** ?

```
unsigned char m = 0377;
a = a | m;
```

```
a: 00 111 110
|
m: 11 111 111
--------------

a: 11 111 111
```

- How would you set the **right two bits** without changing the other bits?

```
unsigned char m = 0003;
a = a | m;
```

```
a: 00 111 110
|
m: 00 000 011
--------------

a: 00 111 111
```

Complementing (Inverting) Bits

- Using C operators:
  - ^ with 1 will complement, ^ with 0 means “no change”
- So, create a mask with 1’s where you want to complement, and 0’s everywhere else

If input is...	And mask is...	Then input ^ mask =
0	0	0 (no change)
0	1	1 (complement)
1	0	1 (no change)
1	1	0 (complement)

Complementing... (cont'd)

- How would you complement (invert) **all** the bits in a **`char`** ?

```
unsigned char m = 0377;
a = a ^ m;

a = ~a; //also works
```

```
a: 00 111 110
^
m: 11 111 111
--------------

a: 11 000 001
```

- How would you complement the **right two bits** without changing the other bits?

```
unsigned char m = 0003;
a = a ^ m;
```

```
a: 00 111 110
^
m: 00 000 011
--------------

a: 00 111 101
```

Testing Bits for 1's

- Using C operators:
  1. & with 1 where you want to test, & with 0 elsewhere
  2. then check if result == mask
- So, create a mask with 1's where you want to test, and 0's everywhere else

If input is...	And mask is...	Then input & mask =
0	0	0 (matches mask)
0	1	0 (won't match mask)
1	0	0 (matches mask)
1	1	1 (matches mask)

Test... (cont'd)

- How would you test (if == 1) **all** the bits in a **`char`** ?

```
unsigned char m = 0377;
if ((a & m) == m)
    ...
```

```
a: 00 111 110
&
m: 11 111 111
-------------

   00 111 110
```

- How would you test if the **right two bits** == 1?

```
unsigned char m = 0003;
if ((a & m) == m)
    ...
```

Not equal to m →

```
a: 00 111 110
&
m: 00 000 011
-------------

   00 000 010
```

Counting the Bits That Are 1's

- Using C operators:
  1. you already know how to test if a specific bit == 1
  2. do this for each bit, one at a time
  3. each time the bit == 1, add 1 to a counter
- A movable mask
  - (0001 << i) creates a mask with a 1 in the ith position from the right, and 0 everywhere else

Test… (cont'd)

```
unsigned char m;
unsigned int cnt = 0;
for (i = 0; i < 8; i++) {
    m = 0001 << i;
    if ((a & m) == m)
        cnt += 1;
}
```

Testing Bits for 0's

- Using C operators:
  - (you try it)
- How would you test (if == 0) all the bits in a char?

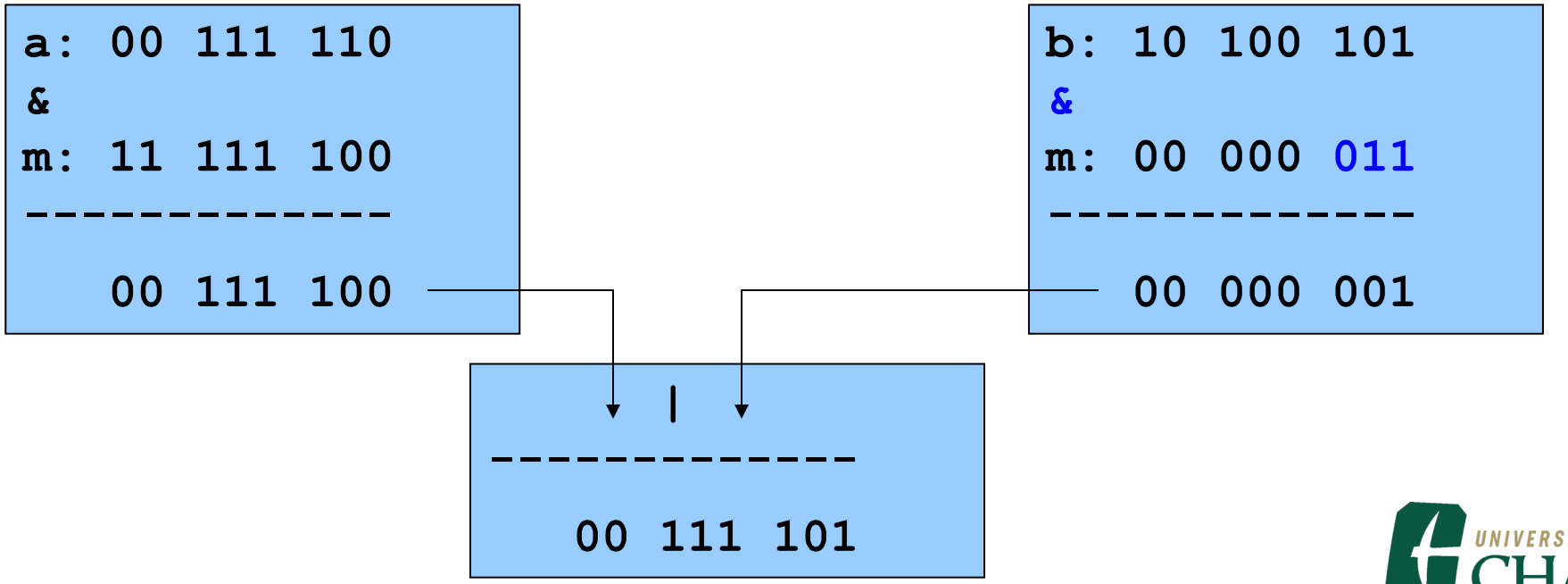
```
???
```

- How would you test if the two right bits == 0?

```
???
```

Copying Selected Bits (from b to a)

- Using C operators:
  - clear all the bits in a you do want to replace
  - clear all the bits in b you don't want to copy
  - | a with b to get result



References

- K. N. King, *C Programming: A Modern Approach*, 2nd Edition. W. W. Norton & Company. 2008.

- D.S. Malik, *C++ Programming: From Problem Analysis to Program Design*, Seventh Edition. Cengage Learning. 2014.

- Slides source: CSC 230 - C and Software Tools
  © NC State University Computer Science Faculty. Modified for class use.