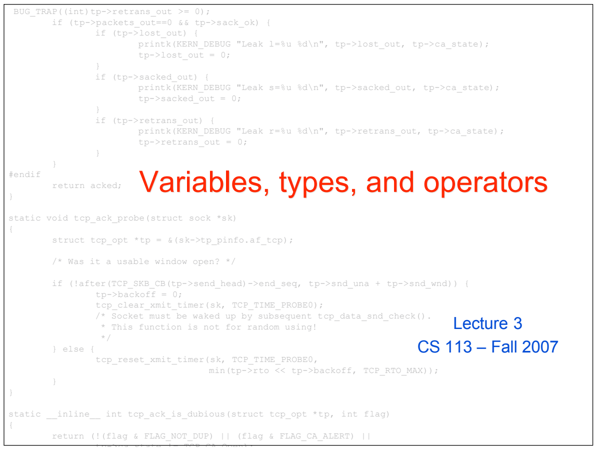
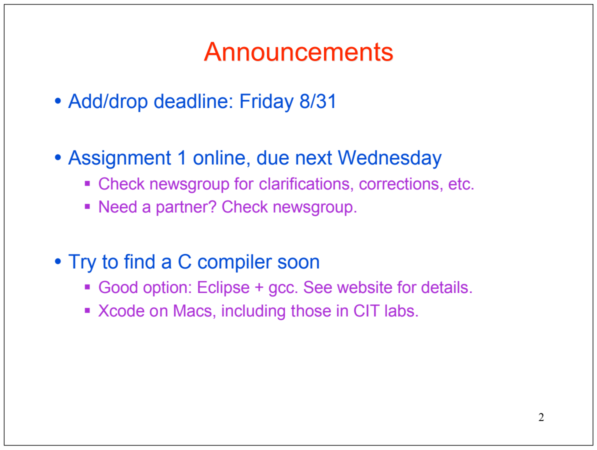


```
BUG_TRAP((int)tp->retrans_out >= 0);
        if (tp->packets_out==0 && tp->sack_ok) {
                if (tp->lost_out) {
                        printk(KERN_DEBUG "Leak l=%u %d\n", tp->lost_out, tp->ca_state);
                        tp->lost_out = 0;
                }
                if (tp->sacked_out) {
                        printk(KERN_DEBUG "Leak s=%u %d\n", tp->sacked_out, tp->ca_state);
                        tp->sacked_out = 0;
                }
                if (tp->retrans_out) {
                        printk(KERN_DEBUG "Leak r=%u %d\n", tp->retrans_out, tp->ca_state);
                        tp->retrans_out = 0;
                }
        }
#endif
        return acked;
}

static void tcp_ack_probe(struct sock *sk)
{
        struct tcp_opt *tp = &(sk->tp_pinfo.af_tcp);

        /* Was it a usable window open? */

        if (!after(TCP_SKB_CB(tp->send_head)->end_seq, tp->snd_una + tp->snd_wnd)) {
                tp->backoff = 0;
                tcp_clear_xmit_timer(sk, TCP_TIME_PROBE0);
                /* Socket must be waked up by subsequent tcp_data_snd_check().
                 * This function is not for random using!
                 */
        } else {
                tcp_reset_xmit_timer(sk, TCP_TIME_PROBE0,
                                     min(tp->rto << tp->backoff, TCP_RTO_MAX));
        }
}

static __inline__ int tcp_ack_is_dubious(struct tcp_opt *tp, int flag)
{
        return (!(flag & FLAG_NOT_DUP) || (flag & FLAG_CA_ALERT) ||
```

Variables, types, and operators

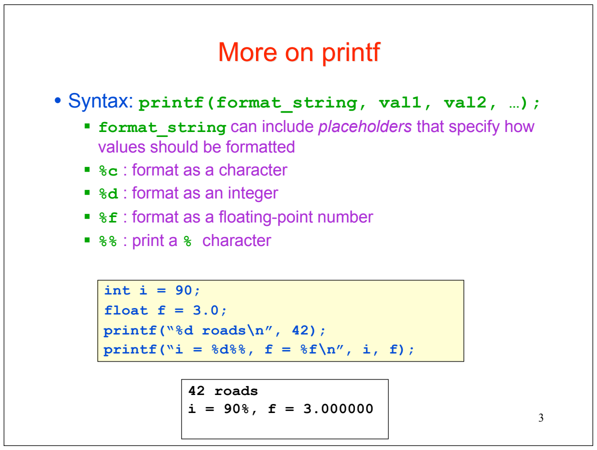
Lecture 3

CS 113 – Fall 2007



Announcements

- Add/drop deadline: Friday 8/31
- Assignment 1 online, due next Wednesday
  - Check newsgroup for clarifications, corrections, etc.
  - Need a partner? Check newsgroup.
- Try to find a C compiler soon
  - Good option: Eclipse + gcc. See website for details.
  - Xcode on Macs, including those in CIT labs.

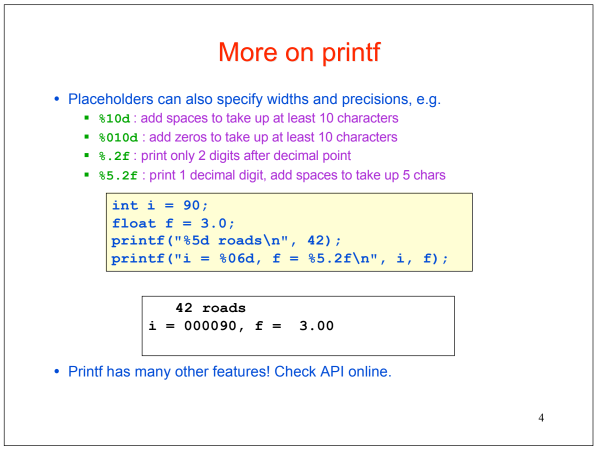


More on printf

- Syntax: `printf(format_string, val1, val2, …);`
  - `format_string` can include *placeholders* that specify how values should be formatted
  - `%c` : format as a character
  - `%d` : format as an integer
  - `%f` : format as a floating-point number
  - `%%` : print a `%` character

```
int i = 90;
float f = 3.0;
printf(“%d roads\n”, 42);
printf(“i = %d%%, f = %f\n”, i, f);
```

```
42 roads
i = 90%, f = 3.000000
```



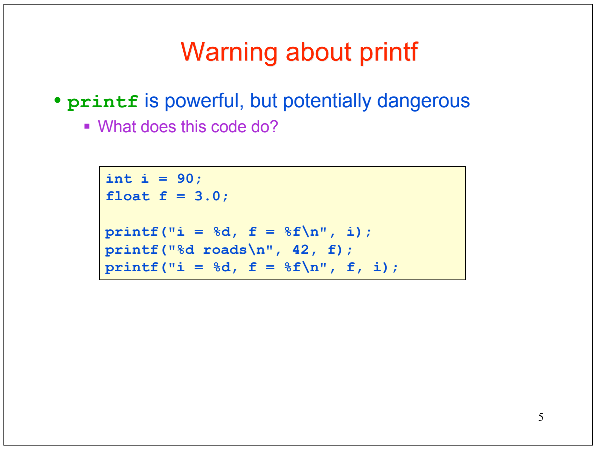
More on printf

- Placeholders can also specify widths and precisions, e.g.
  - `%10d` : add spaces to take up at least 10 characters
  - `%010d` : add zeros to take up at least 10 characters
  - `%.2f` : print only 2 digits after decimal point
  - `%5.2f` : print 1 decimal digit, add spaces to take up 5 chars

```
int i = 90;
float f = 3.0;
printf("%5d roads\n", 42);
printf("i = %06d, f = %5.2f\n", i, f);
```

```
   42 roads
i = 000090, f =  3.00
```

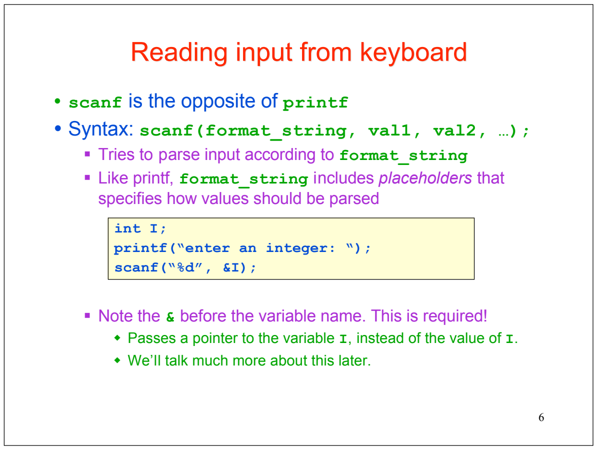
- Printf has many other features! Check API online.



Warning about printf

- `printf` is powerful, but potentially dangerous
  - What does this code do?

```
int i = 90;
float f = 3.0;

printf("i = %d, f = %f\n", i);
printf("%d roads\n", 42, f);
printf("i = %d, f = %f\n", f, i);
```



Reading input from keyboard

- `scanf` is the opposite of `printf`
- Syntax: `scanf(format_string, val1, val2, …);`
  - Tries to parse input according to `format_string`
  - Like printf, `format_string` includes *placeholders* that specifies how values should be parsed

```
int I;
printf(“enter an integer: “);
scanf(“%d”, &I);
```

  - Note the `&` before the variable name. This is required!
    - Passes a pointer to the variable `I`, instead of the value of `I`.
    - We'll talk much more about this later.

More scanf examples

- Read a float from the keyboard

```
float F;
printf(“enter a float: “);
scanf(“%f”, &F);
```

- Parse a date into month, day, year

```
int month, day, year;
printf(“enter a date: “);
scanf(“%d/%d/%d”, &month, &day, &year);
```

scanf editorial

- `scanf` is powerful, but awkward and dangerous.
  - Error handling is difficult
  - What does this code do?

```
int I;
printf(“enter an integer: “);
scanf(“%d”, I);
```

  - Use it for now. We’ll see better ways of handling input later.

Variables

- Variables have a *name* and a *type*
- Restrictions on variable names
  - Must begin with a letter
  - Can contain letters, digits, and underscores (_)
  - Can’t be a reserved word (if, else, void, etc.)
  - Only the first 31 characters matter
- C has 4 basic built-in types
  - char, int, float, double

More on types

- C also defines *type qualifiers* that modify basic types
  - Short, long, unsigned, signed
  - Warning: meaning differs between compilers and machines!

Type	Typical size	Typical range
char	1 byte	[0, 256]
signed char	1 byte	[-128, 127]
short int	2 bytes	[-32768, 32767]
int	4 bytes	[-2,147,483,648, 2,147,483,647]
unsigned int	4 bytes	[0, 4,294,967,295]
long long int	8 bytes	[-9,223,372,036,854,775,808, 9,223,372,036,854,775,807]
float	4 bytes	Approx. ±[1.40e-45, 3.40e+38]
double	8 bytes	Approx. ±[4.94e-324 to 1.80e+308]

Variable declaration and initialization

- C requires all variables to be declared *before* any other statements
  - Although this was relaxed in C99 standard

```
int main() {
    int x = 1, y;
    int sum;
    y = 3;
    return 0;
}
```

```
int main() {
    int x = 1, y;
    y = 3;
    int sum; /*compiler error!*/
    return 0;
}
```

- The initial value of a variable is *undefined*

```
int i;
printf(“%d\n”, i); /* undefined behavior */
```

Other variable qualifiers

- `extern` : used to share variables across C source files
- `static` : used to prevent variables from being accessed in other source files
  - We’ll see other uses of static later
- Qualifiers that are used infrequently:
  - `register` : requests that the compiler store the variable in a processor register instead of in memory
  - `volatile` : tells the compiler that the variable’s value might be changed by some external force (another thread, etc.)

Numeric Constants

- Examples of numeric constants
  - 1234 : integer constant
  - 1234L : long integer constant
  - 1234u : unsigned integer constant
  - 3.1415 : double constant
  - 3.1415f : float constant
  - 0x1f : integer constant, expressed in hexadecimal
  - 0134 : integer constant, expressed in octal

Characters

- Character constants are surrounded by single quotes
  - E.g. `'a'` , `'0'`, `'\n'`
- Escape sequences used to write special constants, e.g.:
  - `'\n'` : newline
  - `'\"'` : double quote
  - `'\t'` : tab
  - `'\\'` : backslash
- Character constants are converted to integers using ASCII value
  - `‘a’ == 97, ‘b’ == 98, … , ‘z’ == 122`
  - `‘A’ == 65, ‘B’ == 66, …, ‘Z’ == 90`
  - `‘0’ == 48, ‘1’ == 49, …, ‘9’ == 57`
  - `‘\n’ == 10, ‘\\’ == 92, …`

Example: character constants

```
char one = ‘1’, two = ‘2’;

printf(“one = %c, two = %c\n”, one, two);
printf(“one = %d, two = %d\n”, one, two);
printf(“%c %d %c %d\n”, 97, 97, ‘a’, ‘a’);
```

```
one = 1, two = 2
one = 49, two = 50
a 97 a 97
```

Another example

- Print an ASCII table in decimal and hexadecimal

```
#include <stdio.h>

int main(void) {
  char j;
  for(j=‘a’; j<=‘m’; j++)
    printf(“%c %3d %3x\n”, j, j, j);
  return 0;
}
```

```
a  97  61
b  98  62
c  99  63
d 100  64
e 101  65
f 102  66
g 103  67
h 104  68
i 105  69
j 106  6a
k 107  6b
l 108  6c
m 109  6d
```

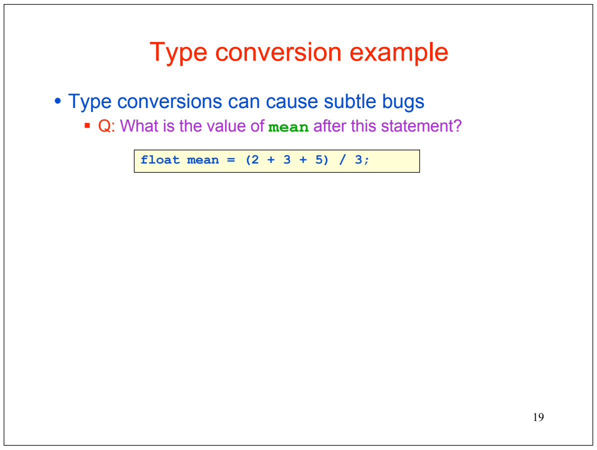
Type conversions

- C is very flexible with type conversions
  - C is *weakly typed* compared to other languages like Java
- If an operator has operands of different types, they are all *implicitly converted* to the wider type
- Conversions also occur when assigning a value of one type to a variable of another type
  - Careful: Information may be lost by this conversion!
  - Example: if `f` is a `float` and `i` is an `int`, `i=f` will truncate the fractional part of `f`

Explicit casts

- *Casting* lets you change the type of a value explicitly
  - Syntax: `(newtype) value`
  - Example:

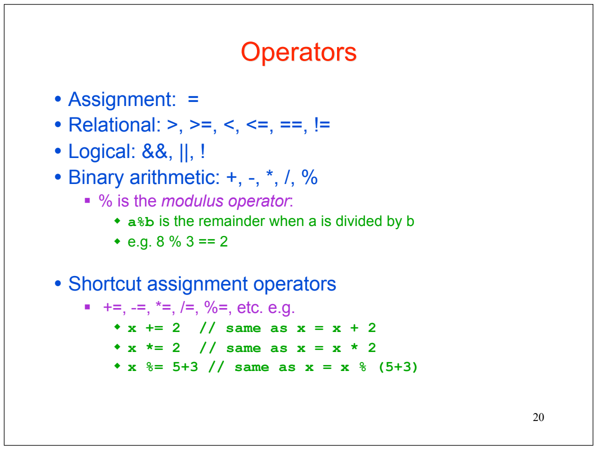
```
float PI = 3.1415;

float int_part = (int) PI;
float frac_part = PI - int_part;
```



Type conversion example

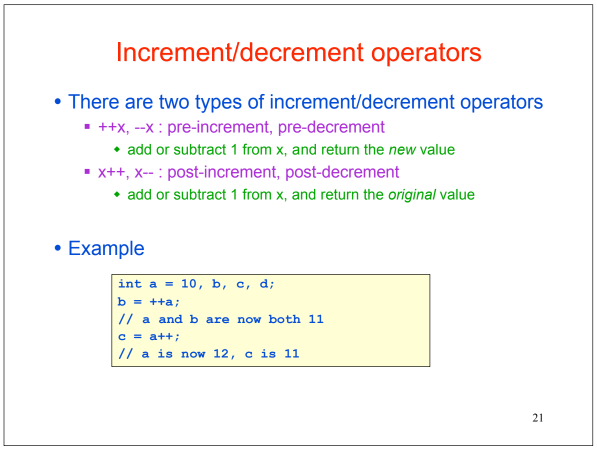
- Type conversions can cause subtle bugs
  - Q: What is the value of **mean** after this statement?

```
float mean = (2 + 3 + 5) / 3;
```



Operators

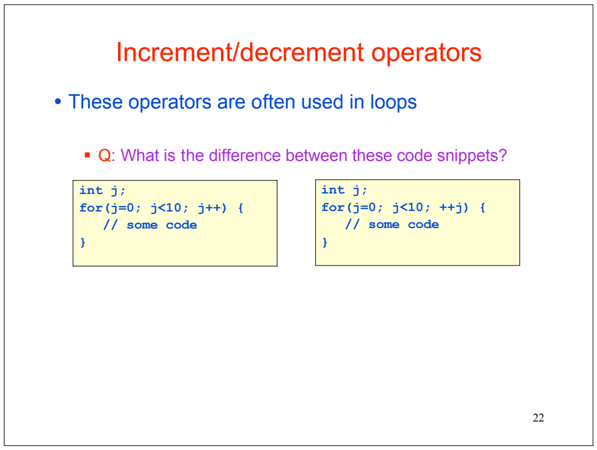
- Assignment: =
- Relational: >, >=, <, <=, ==, !=
- Logical: &&, ||, !
- Binary arithmetic: +, -, *, /, %
  - % is the *modulus operator*:
    - **a%b** is the remainder when a is divided by b
    - e.g. 8 % 3 == 2
- Shortcut assignment operators
  - +=, -=, *=, /=, %=, etc. e.g.
    - `x += 2  // same as x = x + 2`
    - `x *= 2  // same as x = x * 2`
    - `x %= 5+3 // same as x = x % (5+3)`



Increment/decrement operators

- There are two types of increment/decrement operators
  - ++x, --x : pre-increment, pre-decrement
    - add or subtract 1 from x, and return the *new* value
  - x++, x-- : post-increment, post-decrement
    - add or subtract 1 from x, and return the *original* value
- Example

```
int a = 10, b, c, d;
b = ++a;
// a and b are now both 11
c = a++;
// a is now 12, c is 11
```

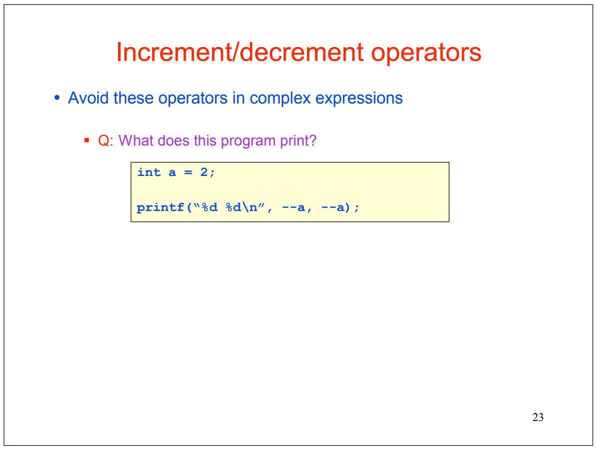


Increment/decrement operators

- These operators are often used in loops
  - Q: What is the difference between these code snippets?

```
int j;
for(j=0; j<10; j++) {
   // some code
}
```

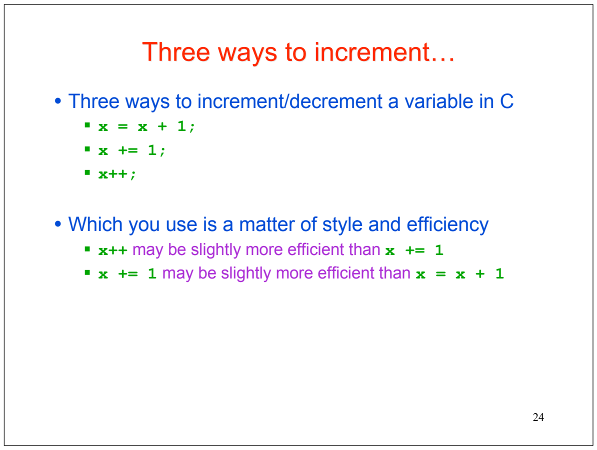
```
int j;
for(j=0; j<10; ++j) {
   // some code
}
```



Increment/decrement operators

- Avoid these operators in complex expressions
  - Q: What does this program print?

```
int a = 2;

printf(“%d %d\n”, --a, --a);
```

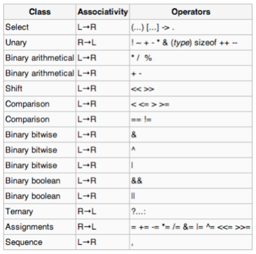


Three ways to increment…

- Three ways to increment/decrement a variable in C
  - `x = x + 1;`
  - `x += 1;`
  - `x++;`
- Which you use is a matter of style and efficiency
  - **x++** may be slightly more efficient than **x += 1**
  - **x += 1** may be slightly more efficient than **x = x + 1**

Order of evaluation

- *Operator precedence* and *associativity* rules define the order in which operators are evaluated
  - Some examples:
    - 5 + 3 / 2 ≡ 5 + (3/2)
    - 1 - 1 - 1 ≡ (1 - 1) - 1
    - 3 < 5 + 2 ≡ 3 < (5 + 2)



Class	Associativity	Operators
Select	L→R	(...) [...] -> .
Unary	R→L	! ~ + - * & (*type*) sizeof ++ --
Binary arithmetical	L→R	* / %
Binary arithmetical	L→R	+ -
Shift	L→R	<< >>
Comparison	L→R	< <= > >=
Comparison	L→R	== !=
Binary bitwise	L→R	&
Binary bitwise	L→R	^
Binary bitwise	L→R	|
Binary boolean	L→R	&&
Binary boolean	L→R	||
Ternary	R→L	?...:
Assignments	R→L	= += -= *= /= &= |= ^= <<= >>=
Sequence	L→R	,

Avoid confusing expressions

- Use parentheses to make precedence clear
  - Q: What does this code do?

```
void main()
{
  int a = -2, b = -1, c = 0;
  if( a < b < c )
    printf( "True.\n" );
  else
    printf( "False.\n" );

  if (a >= b >= c)
    printf( "True.\n");
  else
    printf( "False.\n");
}
```