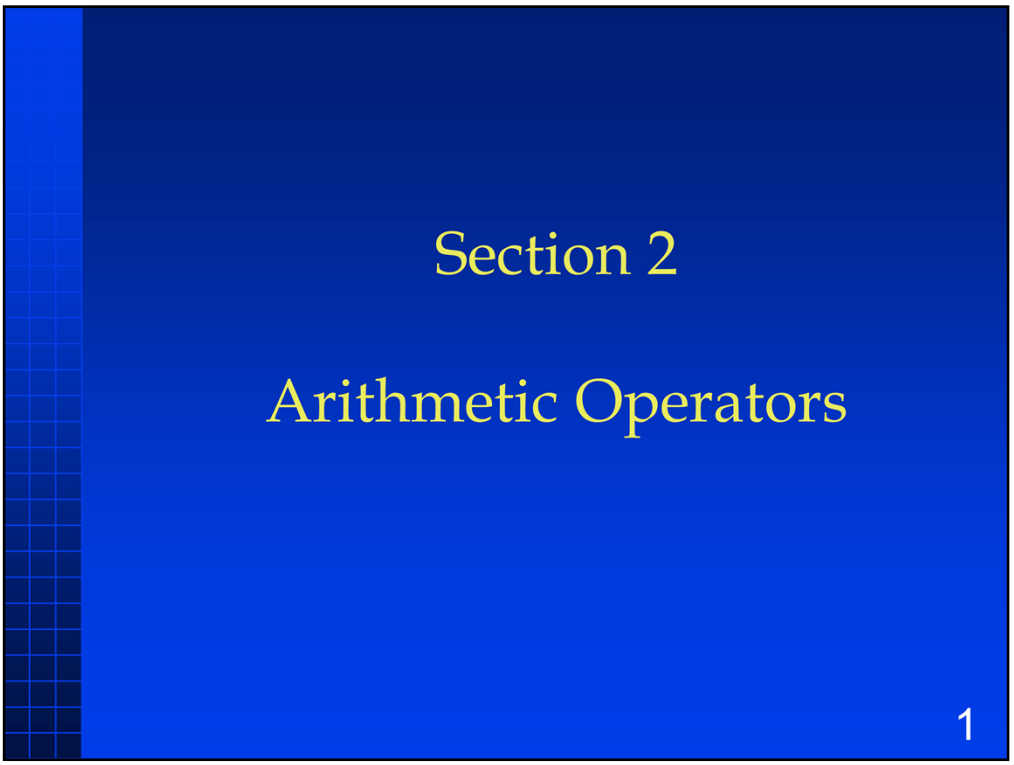


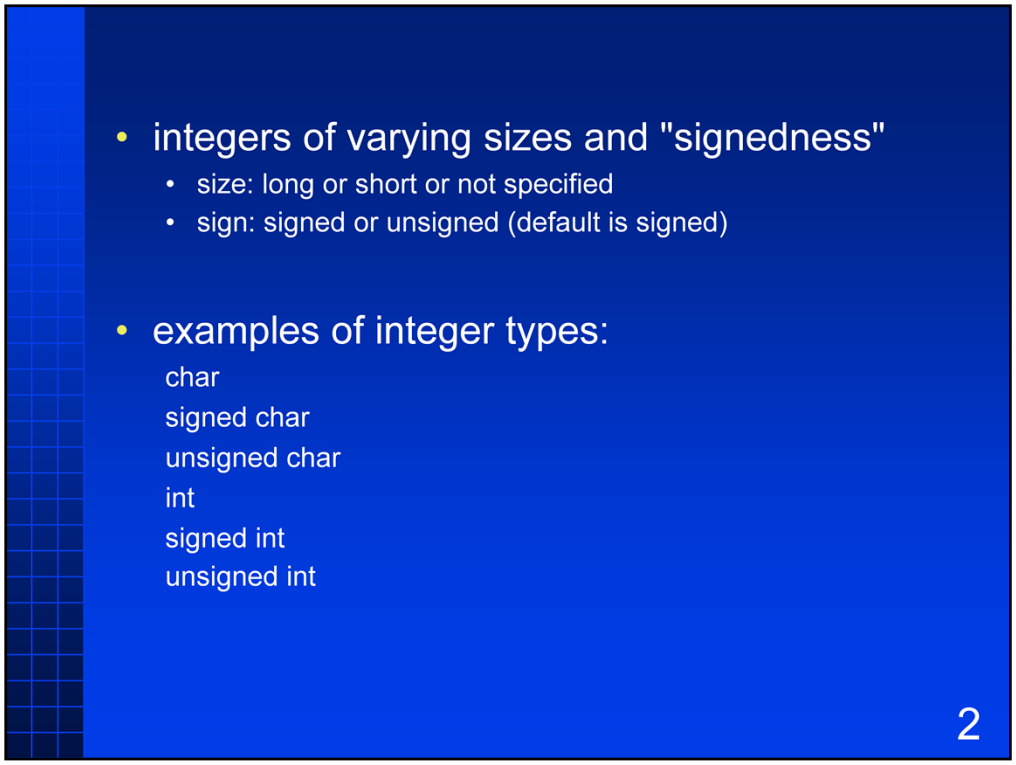
C has lots of operators -- originally, an attempt to model instructions sets of hardware (esp PDP-11)

Lots of them (powerful, terse, overwhelming at times)

Before discussing operators, need to have a quite look at arithmetic types and declarations (specify ranges of values and storage):

Basically, most things are integers (ints), considered to be equivalent to mahine word. Historically, a word was the smallest addressible unit of storage (PDP-11, might even be true for x86, who knows?).

Character (char) is also considered to be an “arithmetic” type; its just a very small integer (can only how values from 0 to 255 or -128 to +127). Generally assume that a char is the smallest unit of data (corresponds to a byte of storage).



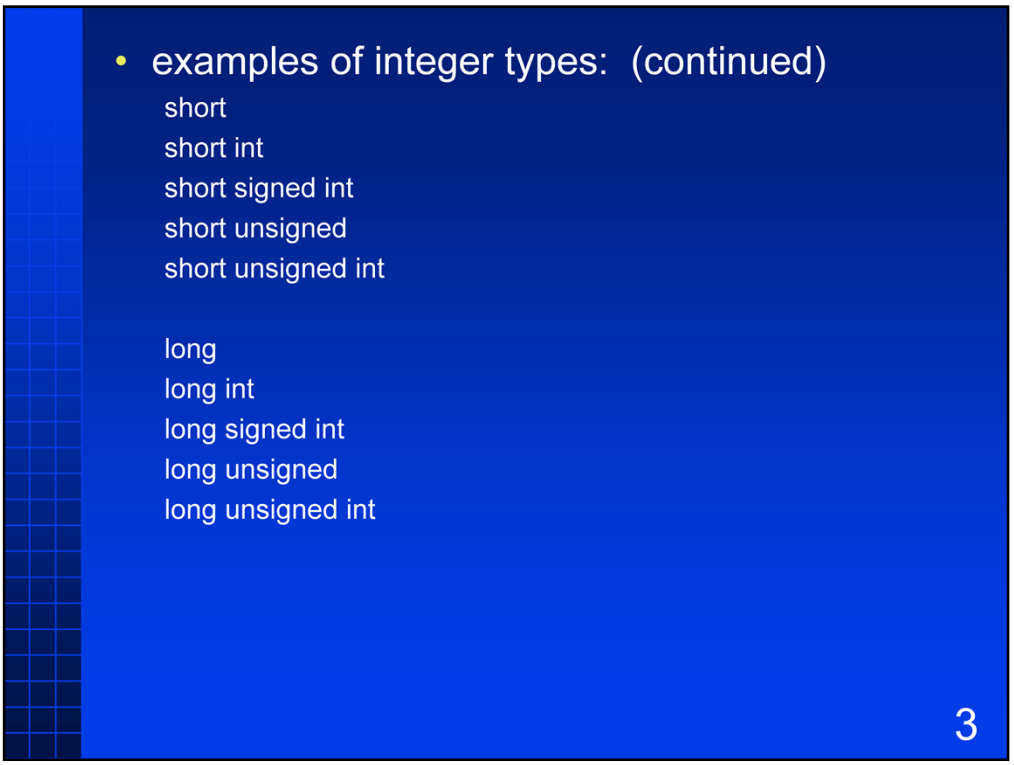
declarations: what the type is and what the name of the variable is.

aside: rules for identifiers as usual, remember case sensitive.

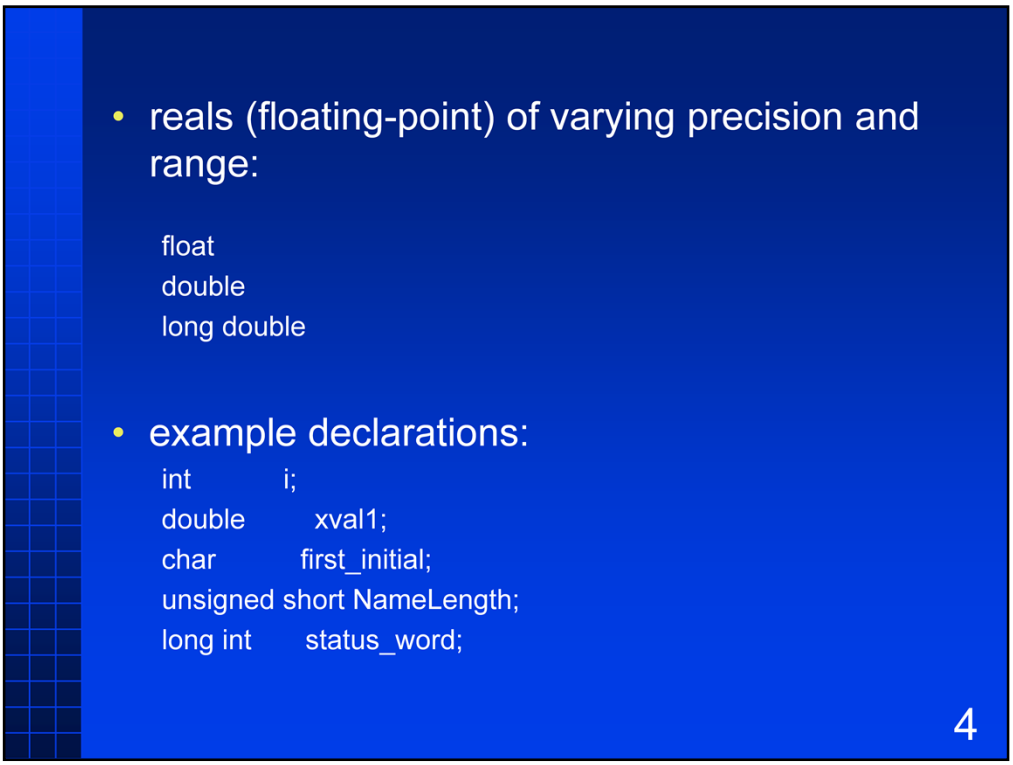
integers: lots of different modifiers that can be applied

size: how much storage, system-dependent unspecified means system default (for PC, 16 or 32 bits depending on OS), long and short a relative to system default.

sign: should the number be consiferd signed or unsigned. may or may not be of concern (affects things like relative comparisons, overflow conditions)

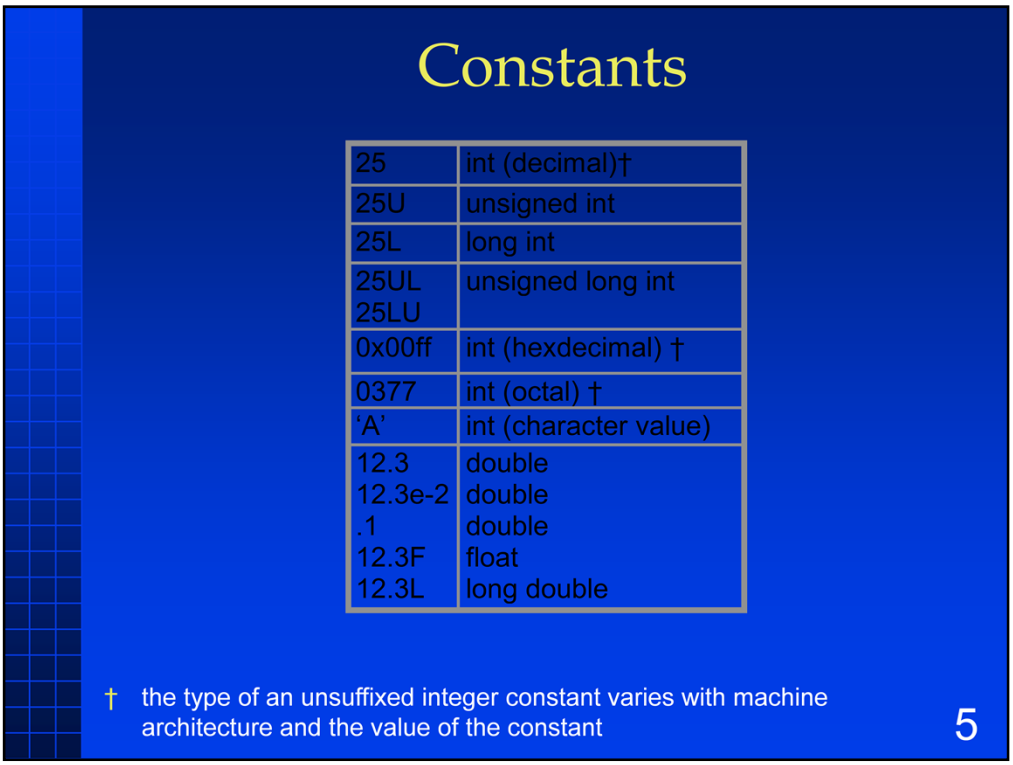

examples of integer types: (continued)
short
short int
short signed int
short unsigned
short unsigned int
long
long int
long signed int
long unsigned
long unsigned int
3

lots of ints



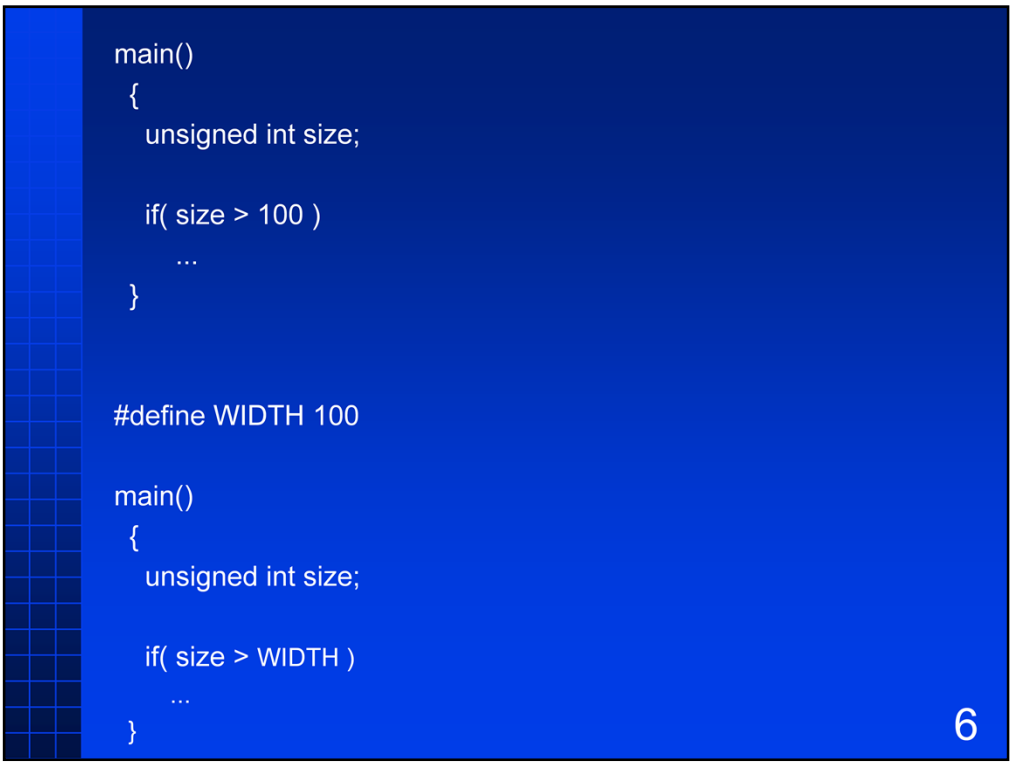
floats are all system-dependent: PC uses IEEE 2, 4, 8 bytes, etc

declarations: pick the apprropriate type to model the data being represented.



can force constants to acquire specific types (controls amount of storage).

note single character constant (not a string)



dealing with constants:

constants are a fact of life, good programming practice to use symbolic constants

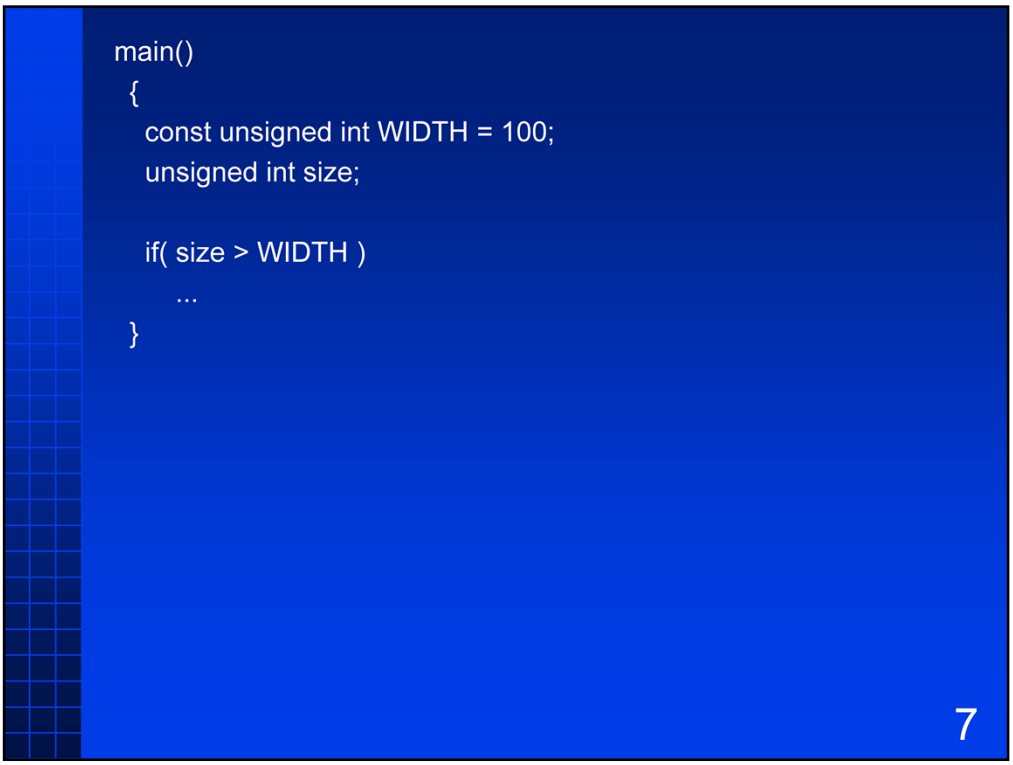
eg compare against 100 as literal const. better engineering to use constant.

use #define: another one of the preprocessor directives:

two parts, item and replacement. compiler will replace item with replacement wherever it occurs (very simple macro). substitution occurs at compile-time (referred to as a lexical replacement)

replacement can be arbitrary; can take time to compile

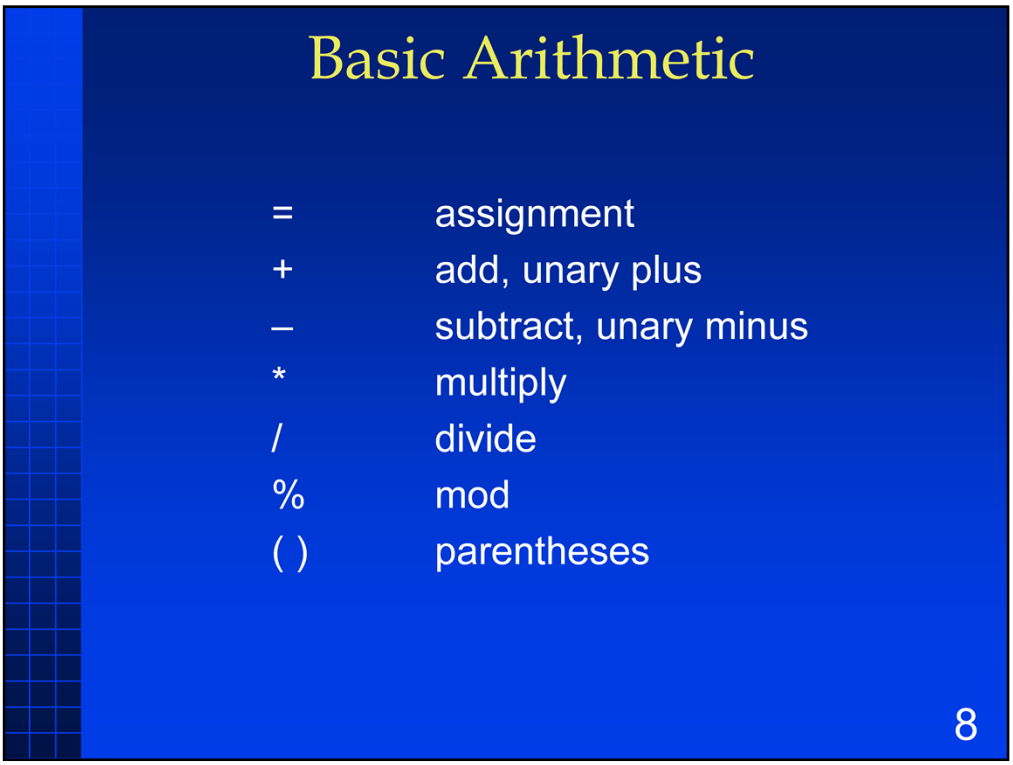
type of constant can change depending on context (eg in this example would be unsigned, since conpared to unsigned; if changes to signed variable, constant would be considered signed).



different kind of constant, use a “storage class” in a variable declaration.

says that value of variable does not change (and provides its initial value)

type cannot change, but compiler could optimize storage (eg assember literals instead of actual storage).



traditional priority of ops.

assignment is a binary operator, its “value” is the lhs.

the act of assigning is almost like a side-effect, so in a simple assignment statement the value is discarded and we use the side effect.

so, statements like a=b=c work. (equiv to a = (b=c))

but note:

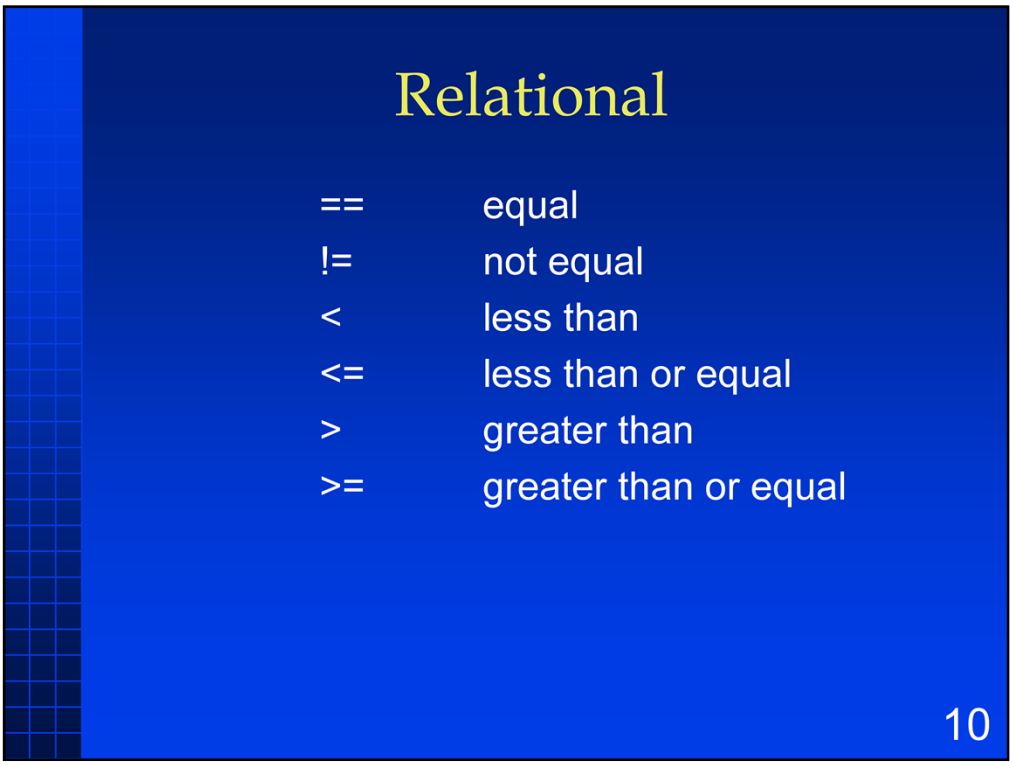
```
if (a = b)
```

is probably not what you want.

For example:

```
x = y + 3 - 6/(3 + z);

x = y = 0;
```



remember == for equality test

highlight ! for not

For example:

```
if( x > y )
  {
    if( x < z )
      {
        ....

x = (y == z);

TRUE  == 1  /* any non-zero */
FALSE == 0
```

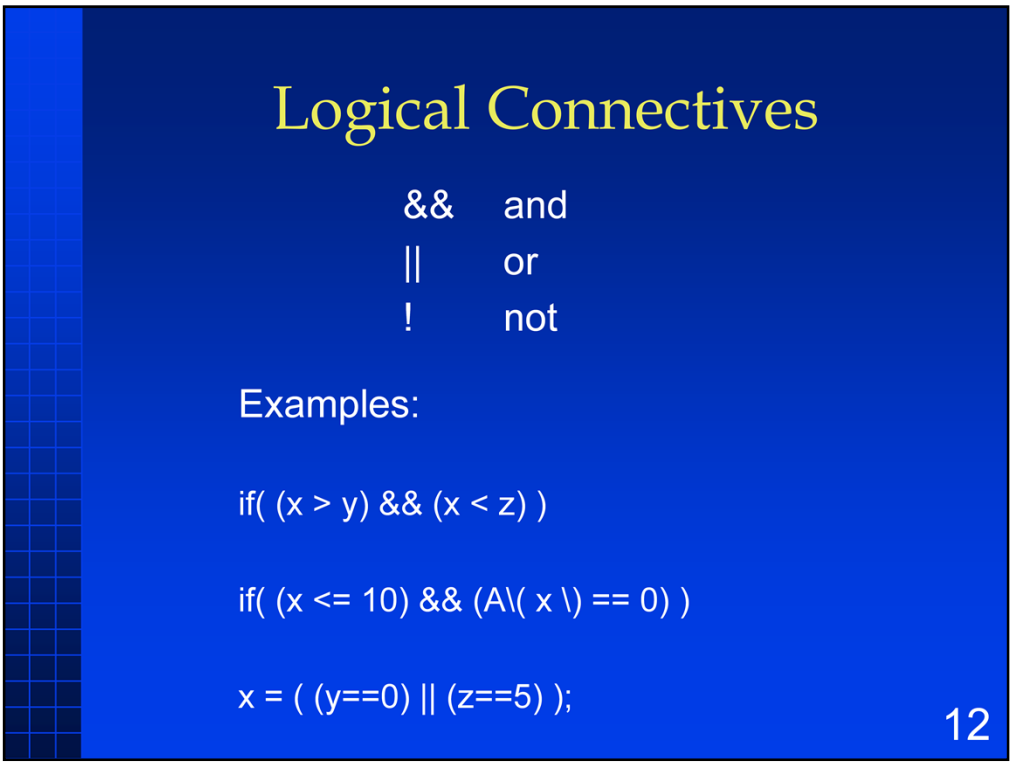
11

result of a relational op is an integer value that is 0 for false and not 0 for true (typically 1 or -1)

this is why

```
if( a=b)
```

is so much trouble



logical vs bitwise

yield integers with same meanings as relational

Bitwise

&	and
|	or
^	exclusive or
~	not
>>	shift right
<<	shift left

For example:

```
#define MASK 0x00000040   /* bit 25 */
unsigned int status;

status = status | MASK;   /* set mask */

if( status & MASK )       /* test mask */

status = status & ~MASK;  /* clear mask */

status = status ^ MASK;   /* toggle mask */
```

Auto Increment and Decrement

++ increment

– – decrement

For example:

```
z = --a;    /* Pre decrement */

x = ++a;    /* Pre increment */

z = a--;    /* Post decrement */

x = a++;    /* Post increment */
```

Special Assignment

+=	plus assign
– =	subtract
*=	multiply
%=	mod
/=	divide
&=	and
|=	or
^=	exclusive or
>>=	right shift
<<=	left shift

For example:

```
y += 5;     /* equivalent y = y + 5  */

y += z--;   /* Side effect only done once */

status ^= MASK;
```

Conditional Expression

<expr1> ? <expr2> : <expr3>

For example:

```
min = ( a < b ) ? a : b;
```

```
if( a < b )
  {
    min = a;
  }
else
  {
    min = b;
  }
```

Comma Operator

<expr1> , <expr2>

For example:

```
c = ( a , b );
```

```
a;
c = b;
```

Operators and Associativity in decreasing precedence

( )	left to right	parentheses
! ~ ++ – – + – size of * &	right to left	logical not bitwise not (1's complement) auto ince., decr. (pre, post) unary plus, minus get storage size dereference, address of
(type)	right to left	force type (typecast)
* / %	left to right	multiply, divide, modulus
+ –	left to right	plus, minus
>> <<	left to right	shift bits right, left
< <= > >=	left to right	less than, ..or equal greater than, ..or equal

Operators and Associativity in decreasing precedence (continued)

== !=	left to right	equal, not equal
&	left to right	bit-wise “and”
^	left to right	bit-wise “exclusive or”
|	left to right	bit-wise “or”
&&	left to right	connective “and”
||	left to right	connective “or”
?:	right to left	conditiona; expr
= op=	right to left	assignment
,	left to right	comma

Basic Arithmetic Types

Type	Size	Notes
char signed char unsigned char	>= 8-bits >= 8-bits >= 8-bits	signed or unsigned -127 .. 127 0 .. 255
short int unsigned short	>= 16-bits >= 16-bits	-32,767 .. 32,767 0 .. 65,535
int unsigned int	>= 16-bits >= 16-bits	machine dependent unsigned version
long int unsigned long	>= 32-bits >= 32-bits	-2,147483,647 .. 2,147,483,647
float	32-bits	real numbers
double	64-bits	

Conversion Rules

- Hierachy of conversions:

  signed-char < unsigned-char < short < unsigned-short < int < unsigned-int < long-int < unsigned-long-int < float < double < long-double

- Promote smaller types to int

Conversion Rules

Op1	Op2	Result
-any-	long double	long double
-any-	double	double
-any-	float	float
-any-	unsigned long	unsigned long
-any-	long int	long int †
-any-	unsigned int	unsigned int
-any-	int	int

† only if long-int is really bigger than int

```
/* oper-1.c : Monthly Payment Schedule */

#include <stdio.h>

main()
 {
  float balance, principal, interest;
  int month;

  balance = 10000.00;
  month = 1;
  printf( " month  balance" );
  printf( " interest principal\n\n" );
  interest = balance * 0.01;
  principal = 750.00 - interest;
```

```
while( balance > principal )
  {
   printf( "%6d%9.2f%9.2f%10.2f\n",
           month, balance,
           interest, principal );
   balance = balance - principal;
   interest = balance * 0.01;
   principal = 750.00 - interest;
   month++;
  }
 printf( "%6d%9.2f%9.2f%10.2f\n\n",
         month, balance,
         interest, principal );
 printf( "number of months to repay is %d\n",
         month );
}
```

month	balance	interest	principal
1	10000.00	100.00	650.00
2	9350.00	93.50	656.50
3	8693.50	86.93	663.07
4	8030.44	80.30	669.70
5	7360.74	73.61	676.39
6	6684.35	66.84	683.16
7	6001.19	60.01	689.99
8	5311.20	53.11	696.89
9	4614.31	46.14	703.86
10	3910.46	39.10	710.90
11	3199.56	32.00	718.00
12	2481.56	24.82	725.18
13	1756.37	17.56	732.44
14	1023.94	10.24	739.76
15	284.18	2.84	747.16

number of months to repay is 15

```
/* oper-2.c : Monthly Payment Schedule */

#include <stdio.h>

#define RATE    0.01
#define INITIAL 10000.00
#define PAYMENT 750.00

main()
 {
  float balance, principal, interest;
  int month;

  balance = INITIAL;
  month = 1;
  printf( " month  balance" );
  printf( " interest principal\n\n" );
  interest = balance * RATE;
```

```
principal = PAYMENT - interest;
while( balance > principal )
 {
  printf( "%6d%9.2f%9.2f%10.2f\n",
          month, balance,
          interest, principal);
  balance = balance - principal;
  interest = balance * RATE;
  principal = PAYMENT - interest;
  month++;
 }
printf( "%6d%9.2f%9.2f%10.2f\n\n",
        month, balance,
        interest, principal );
printf( "number of months to repay is %d\n",
        month );
}
```