Control statements and operators

Lecture 2
CS 113 – Fall 2007

Announcements

- Assignment #1 will be posted online soon
  - Due next Wednesday at 11:59pm on CMS
- Add/drop deadline: Friday 8/31
- C compiler news
  - CodeWarrior is *not* available in the CIT labs
  - However, Eclipse is, and it’s better anyway
  - Eclipse is also available for free to download (see course website)

A simple C program

```
#include <stdio.h>

int main(void)
{
    int x = 1, y;
    int sum;
    y = 3;
    sum = x + y; /* compute sum */
    printf(“%d plus %d is %d\n”, x, y, sum);
}
```

A simple C program

```
#include <stdio.h>

int main(void)
{
    int x = 1, y;
    int sum;
    y = 3;
    sum = x + y; /* compute sum */
    printf(“%d plus %d is %d\n”, x, y, sum);
}
```

- Declare integer variables named `x` and `y`
- `y` is declared but not initialized.
  - Q: What is its initial value?

A tangent: C standards

- C has evolved over time
  - There are three common versions:
    - Original C (1978) “K&R C”
    - ANSI Standard C (1989) “Ansi C”
    - Revised ANSI Standard (1999) “C99”
  - Example: K&R and ANSI C required variable declarations to appear *before* any other statements in a function

Legal in K&R, Ansi C, C99:

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

Legal in C99 only:

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

- Most modern compilers support C99 extensions

A simple C program

```
#include <stdio.h>

int main(void)
{
    int x = 1, y;
    int sum;
    y = 3;
    sum = x + y; /* compute sum */
    printf(“%d plus %d is %d\n”, x, y, sum);
}
```

- Comments begin with `/*` and end with `*/`
  - Caution: you can’t nest comments

```
   /* /* /* this works ok */
   /* /* /* but this causes an error */ */ */
```

Another example

```
#include <stdio.h>

int main(void)
{
    int x = 1, y;
    int sum;
    y = 3;
    sum = x + y; /* compute sum */
    printf(“%d plus %d is %d\n”, x, y, sum);
}
```

- `printf` can format and print out values of variables
  - `%d` is a special *placeholder*
  - `printf` substitutes the values of the variables specified as arguments into the placeholders
  - `printf` has many other options and features

`if` statements

- Basic form: `if( condition ) statement;`
  - `statement` is executed iff condition is true
- Example:

```
if( 2 < 5 )
  printf(“Surprise! 2 is less than 5\n”);
```

- `statement` can also be multiple lines of code surrounded by braces, e.g.

```
if( 2 < 5 ) {
  printf(“Surprise! 2 is less than 5\n”);
  printf(“What a shock!”);
}
```

`if-else` statements

- `if(cond) statement1 else statement2;`
  - `statement1` is executed iff condition is true
  - `statement2` is executed iff condition is false
- Example:

```
if( a < 5 )
  printf(“a < 5\n”);
else
{
  if( a < 8 )
    printf(“a < 8\n”);
  else
    printf(“a >= 8\n”);
}
```

=

```
if( a < 5 )
  printf(“a < 5\n”);
else if( a < 8 )
  printf(“a < 8\n”);
else
  printf(“a >= 8\n”);
```

`if-else` statement pitfall

- What does this code do?

```
if( a > 0 )
  if( a < 5 )
    printf(“a < 5\n”);
else
  printf(“a <= 0, ”);
  printf(“a > 0\n”);

printf(“done.”);
```

=

```
if( a > 0 )
{
  if( a < 5 )
    printf(“a < 5\n”);
  else
    printf(“a <= 0, ”);
}
printf(“a > 0\n”);
printf(“done.”);
```

`if-else` statement pitfall

- What does this code do?

```
if( a > 0 )
  if( a < 5 )
    printf(“a < 5\n”);
else
  printf(“a <= 0, ”);
  printf(“a > 0\n”);

printf(“done.”);
```

≠

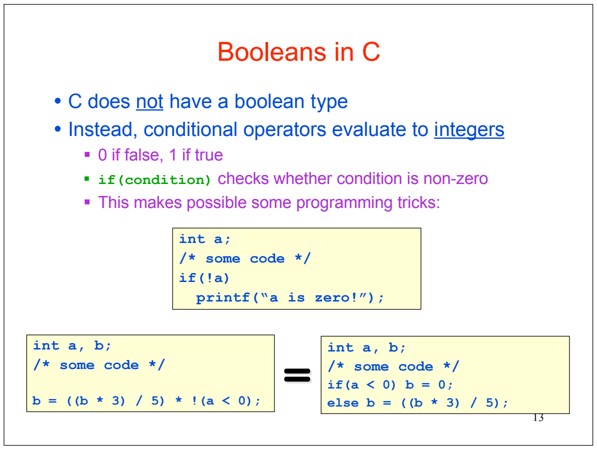
```
if( a > 0 )
{
  if( a < 5 )
    printf(“a < 5\n”);
}
else
{
  printf(“a <= 0, ”);
  printf(“a > 0\n”);
}

printf(“done.”);
```

Relational operators

- C has the following relational operators:

`a == b`	True iff a equals b
`a != b`	True iff a does not equal b
`a < b`	True iff a is less than b
`a > b`	True iff a is greater than b
`a <= b`	True iff a is less than or equal to b
`a >= b`	True iff a is greater than or equal to b
`a && b`	True iff a is true and b is true
`a || b`	True iff at least one of a or b is true
`!a`	True iff a is false



Booleans in C

- C does not have a boolean type
- Instead, conditional operators evaluate to integers
  - 0 if false, 1 if true
  - `if(condition)` checks whether condition is non-zero
  - This makes possible some programming tricks:

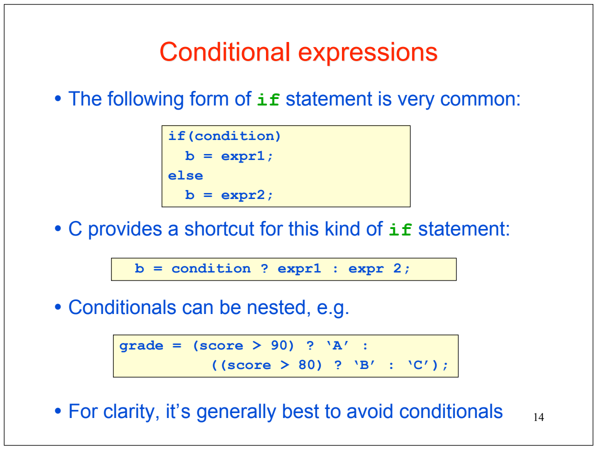
```
int a;
/* some code */
if(!a)
  printf(“a is zero!”);
```

```
int a, b;
/* some code */

b = ((b * 3) / 5) * !(a < 0);
```

=

```
int a, b;
/* some code */
if(a < 0) b = 0;
else b = ((b * 3) / 5);
```



Conditional expressions

- The following form of `if` statement is very common:

```
if(condition)
  b = expr1;
else
  b = expr2;
```

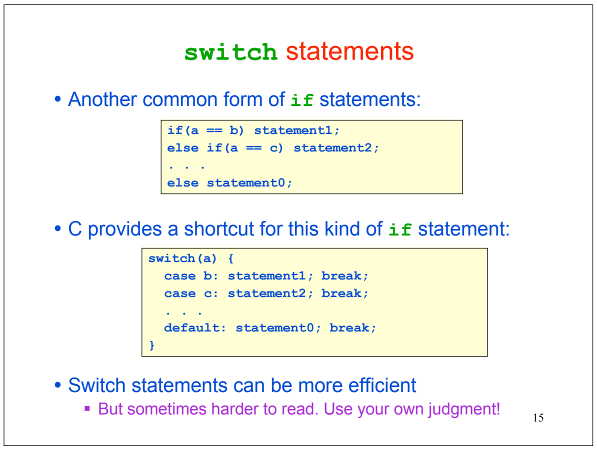
- C provides a shortcut for this kind of `if` statement:

```
  b = condition ? expr1 : expr 2;
```

- Conditionals can be nested, e.g.

```
grade = (score > 90) ? ‘A’ :
           ((score > 80) ? ‘B’ : ‘C’);
```

- For clarity, it’s generally best to avoid conditionals



`switch` statements

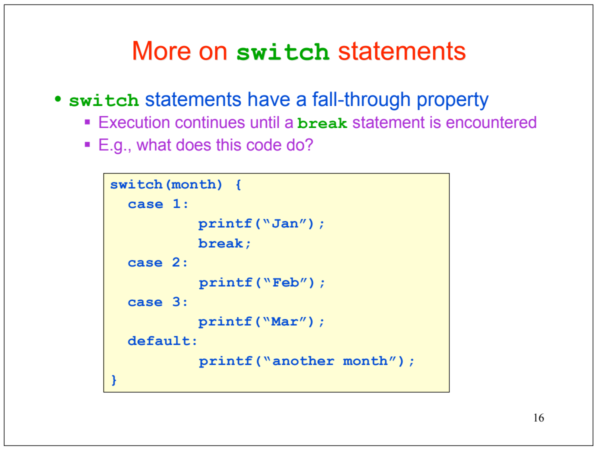
- Another common form of `if` statements:

```
if(a == b) statement1;
else if(a == c) statement2;
. . .
else statement0;
```

- C provides a shortcut for this kind of `if` statement:

```
switch(a) {
  case b: statement1; break;
  case c: statement2; break;
  . . .
  default: statement0; break;
}
```

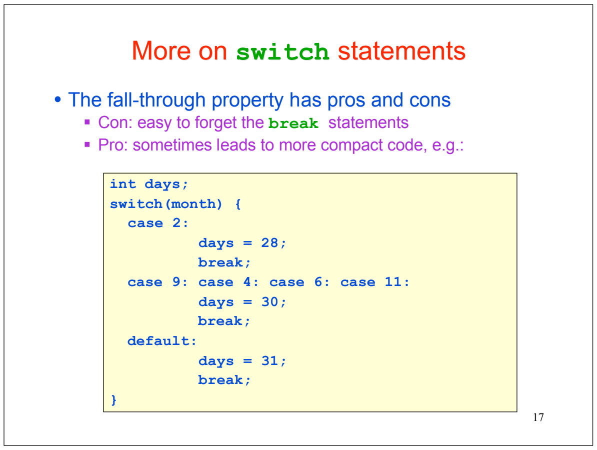
- Switch statements can be more efficient
  - But sometimes harder to read. Use your own judgment!



More on `switch` statements

- `switch` statements have a fall-through property
  - Execution continues until a `break` statement is encountered
  - E.g., what does this code do?

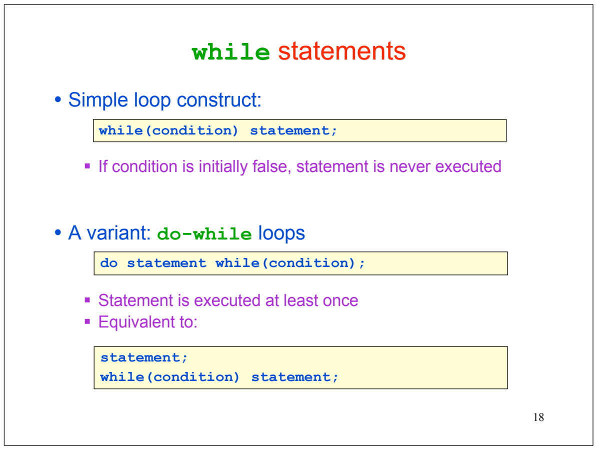
```
switch(month) {
  case 1:
          printf(“Jan”);
          break;
  case 2:
          printf(“Feb”);
  case 3:
          printf(“Mar”);
  default:
          printf(“another month”);
}
```



More on `switch` statements

- The fall-through property has pros and cons
  - Con: easy to forget the `break` statements
  - Pro: sometimes leads to more compact code, e.g.:

```
int days;
switch(month) {
  case 2:
          days = 28;
          break;
  case 9: case 4: case 6: case 11:
          days = 30;
          break;
  default:
          days = 31;
          break;
}
```



`while` statements

- Simple loop construct:

```
while(condition) statement;
```

  - If condition is initially false, statement is never executed

- A variant: `do-while` loops

```
do statement while(condition);
```

  - Statement is executed at least once
  - Equivalent to:

```
statement;
while(condition) statement;
```

`for` statements

```
for(statement1; condition; statement2)
   statement;
```

- `for` statements are more complicated loop constructs
  - `statement1` is executed first (and exactly once)
  - `condition` is evaluated. If true, `statement` is executed, then `statement2`. `condition` is evaluated. If true …
  - Equivalent to:

```
Statement1;
While(condition) {
  statement;
  statement2;
}
```

- Typically,
  - `statement1` is some initialization code (e.g. set a counter to 0)
  - `condition` is a stopping condition for the loop
  - `statement2` increments/decrements a counter

Loops

- Any loop can be written with a while, do-while, or for loop
  - Usually one type of loop is more natural
  - E.g. the following three are equivalent

```
i=0;
while(i<N) {
  printf(“%d\n”, i);
  i++;
}
```

```
i=0;
do {
  if(i < N) {
    printf(“%d\n”, i);
    i++;
  }
} while(i < N);
```

```
for(i=0; i<N; i++) {
  printf(“%d\n”, i);
}
```

Loops

- Any loop can be written with a while, do-while, or for loop
  - Usually one type of loop is more natural
  - E.g. the following three are equivalent

```
int n;
do {
  printf(“enter a number:”);
  n = read_int();
} while(n <= 0);
```

```
int n;
printf(“enter a number:”);
n = read_int();
while(n <= 0) {
  printf(“enter a number:”);
  n = read_int();
}
```

```
int n;
printf(“enter a number:”);
for(n = read_int(); n <= 0 ; n = read_int()) {
  printf(“enter a number:”);
}
```

break and continue

- A `break` inside a loop causes the loop to terminate immediately

```
int n = 10;
while( 1 ) {
  if(n == 0) break;
  n--;
}
```

- A `continue` statement causes the loop to immediately begin executing the next iteration

```
int n;
for(n=0; n < 10; n++) {
  if(n == 0) continue;
  printf(“%d\n”, n);
}
```

Common pitfalls

- What do these code snippets do?

```
int i;
for(i=0; i<10; i++);
  printf(“%d\n”, i);
```

```
int i=1;
while( i = 1 )
  i = 3;
```

Reserved words in C

- We’ve already covered half of the language!

`break`	`case`	`char`
`continue`	`default`	`do`
`double`	`else`	`enum`
`extern`	`float`	`for`
`goto`	`if`	`int`
`long`	`register`	`return`
`short`	`sizeof`	`static`
`struct`	`switch`	`typedef`
`union`	`unsigned`	`void`
`while`		

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);
```