More Control Flow, More Types

COM S 113

January 29, 1999

Announcements

Textbook status

Quizzes

Assignment 1

What's wrong with this?

```
#define MAXLEN 10;
int main() {
  char a[MAXLEN]="University";
  int i;
  for (i=1; i<=MAXLEN; i++);
     printf("%c", a[i]);
  putchar("\n");
  return 0;
}
```

Labels and Goto

Any statement can be prefixed by a label, which is an identifier followed by a colon

Scope of a label is enclosing function body

Can unconditionally jump to label with "`goto` *label*`;`" statement

Normally very bad style

`break` Statements

Used to exit from immediately enclosing loop

```
while (...) {
    ...
    break;
    ...
}
next:;
```

```
do {
    ...
    break;
    ...
} while (...);
next:;
```

```
for (...) {
    ...
    break;
    ...
}
next:;
```

`goto` sometimes useful as multilevel `break`

`continue` Statements

Used to skip remaining portion of current iteration

```
while (...) {
    ...
    continue;
    ...
    next:;
}
```

```
do {
    ...
    continue;
    ...
    next:;
} while (...);
```

```
for (...) {
    ...
    continue;
    ...
    next:;
}
```

Example of `break` and `continue`

```
#include <ctype.h>
main() {
  while (1) {
    int i = getchar();
    if (i == '\n') break;
    else if (ispunct(i)) continue;
    putchar(i);
}}
```

Standard Library

Described in Appendix B of K&R

Contains functions for input and output (`<stdio.h>`), character class tests (`<ctype.h>`), string fcns (`<string.h>`), mathematical functions (`<math.h>`), misc utility fcns (`<stdlib.h>`), and diagnostics (`<assert.h>`)

Warning for users of Unix cc or gcc: use "`-lm`" option

Example of `assert`

```
/* #define NDEBUG /* must occur before #inclusion */
#include <assert.h>
#define NUMFIB 100
main() {
  int i, fib[NUMFIB] = {1,1};
  for (i=2; i<NUMFIB; i++) {
     fib[i] = fib[i-1] + fib[i-2];
     assert(fib[i] > fib[i-1]);   } }
```

The `switch` Statement

```
switch (expression) {
   case const-expr: statements
   case const-expr: statements
   default: statements
}
```

Warning: Case labels act as labels! Almost always need `break` (when not?), almost never use braces around statements (when would you?)

Example of `switch` Statement (K&R p. 59)

```
int c, nwhite = 0, nother = 0, ndigit[10];
while ((c = getchar()) != EOF) {
  switch (c) {
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
      ndigit[c-'0']++; break;
    case ' ': case '\n': case '\t': nwhite++; break;
    default: nother++; break; } }
```

Checkpoint

Now completed Chapter 3 of K&R

Enumeration Types

C is low-level, enumeration types are just `int`!

```
enum day { SUN, MON, TUE, WED, THU, FRI, SAT };
enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
              JUL, AUG, SEP, OCT, NOV, DEC };
enum boolean { NO, YES, FALSE = 0, TRUE };
enum escapes { BELL='\a', TAB='\t', NEWLINE='\n' };
```

Enumeration Types (continued)

No errors for misassigning, performing arithmetic, etc.

Enumerated constant can't appear in two `enum`s

Can iterate if enumerated constants sequential:

```
enum day d;
for (d = SUN; d <= SAT; d++)
  ...
```

Enumeration Types (continued)

Can define variables of this type by naming them at the end:

```
enum day { SUN, MON, TUE, WED, THU, FRI, SAT }
  yesterday, today, tomorrow;
```

Enumeration Tags

Enumeration tags form their own namespace!

```
enum boolean { FALSE, TRUE } boolean;
```

is legal (though poor style)

Enumeration Tags Optional

Program behavior unchanged if removed tag and created variables of type `int`:

```
enum { SUN, MON, TUE, WED, THU, FRI, SAT };
main() {
  int d;
  for (d = SUN; d <= SAT; d++) ....
```

Enumeration Types with `switch`

```
int daysinmonth(enum months m) {
  switch (m) {
    case APR: case JUN: case SEP: case NOV: return 30;
    case FEB: return 28;
    default: return 31;
  }
}
```

Why no `break` needed? What is implicitly assumed?

Alternative `daysinmonth()`

```
#include <assert.h>


int daysinmonth(enum months m) {
  int days[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
  assert(m >= JAN);
  assert(m <= DEC);
  return days[months - JAN];
}
```

Structures

Used to create composite objects containing elements of different types

```
struct {
   component-declarations
}
```

Sample Structures

```
struct {
  double x, y;
} a, b, c[9];
```

```
struct {
  int year;
  short int month, day;
} date1, date2;
```

Access structure elements with dot operator:

`a.x` `c[8].y` `date1.year`

Structure Tags

If a structure type doesn't have a name, you can only create a constant number of objects of that type (unlike for `enum`s)

```
struct employee {
  char name[30];
  int id;
};
```

Structure Tags (continued)

Structure tags form their own namespace, as do structure fields

Can pass structures to and from functions—with call-by-value semantics!

```
struct point {
  double x, y;
};
```

Example Use of Structures

```
#include <math.h>


double distance(struct point a, struct point b) {
  int xoff = a.x - b.x;
  int yoff = a.y - b.y;
  return sqrt(xoff * xoff + yoff * yoff);
}
```

Another Example of Structure Usage

```
struct point midpoint(struct point a,
                      struct point b) {
  struct point m = { (a.x + b.x)/2, (a.y + b.y)/2 };
  return m;
}
```

Defining Types with `typedef`

Used to give names to enumeration and derived types, or to give new names to previously defined types

```
typedef float miles, speed;
typedef float a[5];
typedef struct { float x, y; } point;
```

Defining Variables with New Types

```
typedef float a[5];
typedef struct { float x, y; } point;

a distances = { 2.0, 3.1, 6.9, 4.8, 7.2 };
point origin = { 0.0, 0.0 };
```

Typical Use of `typedef`

```
typedef struct {
  char name[LN];
  char room[LR];
  char ext[LE];    /* extension */
  char desig[LD];  /* designation */
  char compid[LC]; /* company id */
  char logid[LL];  /* login id */
} emp;
```

Scope of Type Definitions

File *or block* scope, depending on placement

This applies to all kinds of type definitions—`enum`, `struct`, `typedef`, etc.