Advanced C Features

ITSC 2181: Introduction to Computer Systems
UNC Charlotte
College of Computing and Informatics

COLLEGE OF COMPUTING
AND INFORMATICS

Flow of Control

- Flow-of-control statements in C:
  - `if-then-else`
  - `conditional operator (? : )`
  - **`switch-case`**
  - `for`
  - `continue and break`
  - `while and do-while`

The `switch` Statement

- A cascaded **`if`** statement can be used to compare an expression against a series of values:

```
if (grade == 4)
  printf("Excellent");
else if (grade == 3)
  printf("Good");
else if (grade == 2)
  printf("Average");
else if (grade == 1)
  printf("Poor");
else if (grade == 0)
  printf("Failing");
else
  printf(”Invalid grade");
```

COLLEGE OF COMPUTING AND INFORMATICS

The **switch** Statement (cont’d)

- The **switch** statement is an alternative:

```
switch (grade) {
  case 4:  printf("Excellent");
           break;
  case 3:  printf("Good");
           break;
  case 2:  printf("Average");
           break;
  case 1:  printf("Poor");
           break;
  case 0:  printf("Failing");
           break;
  default: printf(”Invalid grade");
           break;
}
```

The **switch** Statement (cont’d)

- A **switch** statement may be easier to read than a cascaded **if** statement.
- **switch** statements are often faster than **if** statements.
- Most common form of the **switch** statement:

```
switch ( expression ) {
  case constant-expression : statements
  …
  case constant-expression : statements
  default : statements
}
```

The `switch` Statement (cont’d)

- The word `switch` must be followed by an integer expression—the ***controlling expression***—in parentheses.
- Characters are treated as integers in C and thus can be tested in `switch` statements.
- Floating-point numbers and strings don’t qualify, however.

The `switch` Statement (cont'd)

- Each case begins with a label of the form

  `case` *constant-expression* `:`

- A ***constant expression*** is much like an ordinary expression except that it cannot contain variables or function calls.

  `5` is a constant expression, and `5 + 10` is a constant expression, but `n + 10` isn't a constant expression (unless `n` is a macro that represents a constant).

- The constant expression in a case label must evaluate to an integer (characters are valid).

The `switch` Statement (cont’d)

- After each case label comes any number of statements.
- No braces are required around the statements.
- The last statement in each group is normally `break`.

The **switch** Statement (cont’d)

- Duplicate case labels are not allowed.
- The order of the cases doesn’t matter, and the **default** case doesn’t need to come last.
- Several case labels may precede a group of statements:

```
switch (grade) {
  case 4:
  case 3:
  case 2:
  case 1:   printf("Passing");
            break;
  case 0:   printf("Failing");
            break;
  default: printf("Invalid grade");
            break;
}
```

The **switch** Statement (cont’d)

- To save space, several case labels can be put on the same line:

```
switch (grade) {
  case 4: case 3: case 2: case 1:
           printf("Passing");
           break;
  case 0:  printf("Failing");
           break;
  default: printf("Invalid grade");
           break;
}
```

- If the **default** case is missing and the controlling expression’s value doesn’t match any case label, control passes to the next statement after the **switch**.

(see **date.c** in *Code samples and Demonstrations* in *Canvas*)

The Role of the **`break`** Statement

- Executing a **`break`** statement causes the program to “break” out of the **`switch`** statement; execution continues at the next statement after the **`switch`**.
- The **`switch`** statement is really a form of “computed jump.”
- When the controlling expression is evaluated, control jumps to the case label matching the value of the **`switch`** expression.
- A case label is nothing more than a marker indicating a position within the **`switch`**.

(see **`date.c`** in *Code samples and Demonstrations* in *Canvas*)

COLLEGE OF COMPUTING AND INFORMATICS

The Role of the **break** Statement (cont’d)

- Without **break** (or some other jump statement) at the end of a case, control will flow into the next case.
- Example:

```
switch (grade) {
  case 4:  printf("Excellent");
  case 3:  printf("Good");
  case 2:  printf("Average");
  case 1:  printf("Poor");
  case 0:  printf("Failing");
  default: printf("Invalid grade");
}
```

- If the value of `grade` is 3, the message printed is

  `GoodAveragePoorFailingInvalid grade`

COLLEGE OF COMPUTING
AND INFORMATICS

The Role of the **break** Statement (cont’d)

- Omitting **break** is sometimes done intentionally, but it’s usually just an oversight.
- It’s a good idea to point out deliberate omissions of **break**:

```
switch (grade) {
  case 4: case 3: case 2: case 1:
          num_passing++;
          /* FALL THROUGH */
  case 0: total_grades++;
          break;
}
```

(see **date.c** in *Code samples and Demonstrations* in *Canvas*)

- Although the last case never needs a **break** statement, including one makes it easy to add cases in the future.

Enums

Enumerated Data Type

- Used for variables with small set of possible values, where actual encoding of value is unimportant

```
enum colors {red, blue, green, white, black};
enum colors mycolor;

mycolor = blue;
...
if ((mycolor == blue) || (mycolor == green))
   printf("cool color\n");
```

(see `colors.c` in *Code samples and Demonstrations* in *Canvas*)

Enumerated Data Type (cont’d)

Don’t compare variables of different enumerated types - results **not** what you expect!

```
enum {blue, red, green, white, black}
  primarycolor;
enum {black, brown, orange, yellow}
  halloweencolor;

primarycolor = black;
halloweencolor = black;
if (primarycolor == halloweencolor)
   printf("Same color\n");
```

What will print?

(see `color_comparison.c` in *Code samples and Demonstrations* in *Canvas*)

Although you can interpret enumerated data types as integers, it is **not recommended**

Enumerated Data Type (cont’d)

Compared to **macros…**?

```
#define BLUE 0
#define RED 1
#define GREEN 2
#define WHITE 3
#define BLACK 4

int primarycolor;
primarycolor = RED;
…
if (primarycolor == RED)  …
```

GNOME: *“If you have a list of possible values for a variable, do* ***not*** *use macros for them; use an enum instead and give it a type name”*

`typedef`

The typedef Statement

Assigns an alternate name (synonym) to a C data type

– more concise, more readable

typedef name, not a declaration of a variable

```
typedef char * cptr;
cptr cp;
char * dp;    /* same type as cp */
```

```
typedef struct {
   int val;
   cptr name;
   struct mystruct *next;
} llnode;
llnode entries[100];
```

The typedef Statement (cont’d)

Arrays can be typedefs

```
typedef int values[20];
values tbl1, tbl2;  /* two arrays, each with
                     * 20 ints */
```

- typedefs help make programs portable
  - to retarget a program for a different architecture, just redefine the typedefs and recompile
- Usually, typedefs are collected in a header file that is #include’d in all source code modules

Command Line Arguments

Command Line Arguments

To use command line arguments, define main as:

```
int main(int argc, char *argv[]) {}
```

- `argc`: argument count
  - Includes the program itself
- `argv`: argument vector
  - Array of pointers to command line arguments stored as strings
  - `argv[0]`: name of program
  - `argv[1]` to `argv[argc-1]`: other arguments
  - `argv[argc]`: null pointer

Processing Command Line Args

- Using arrays

```
for (int i = 1; i < argc; i++)
    printf("%s\n", argv[i]);
```

- Using pointers

```
for (char **p = &argv[1]; *p != NULL; p++)
    printf("%s\n", *p);
```

(see `cmd_line_args.c` in *Code samples and Demonstrations* in *Canvas*)

COLLEGE OF COMPUTING AND INFORMATICS

Generic Pointers

The `void *` Type and Type Recasting

- The C type `void *` represents a generic pointer:
  - A pointer to any type (`int`, `float`, `char`, `struct`, etc.)
  - Or a pointer to an unspecified type.
- Typical use is in dynamic memory allocation and systems code (e.g., when creating threads).
- Must be converted to specific type before use. For example:

```
int *array;
array = (int *)malloc(sizeof(int) * 10); // recast void *
*array = 10;
```

COLLEGE OF COMPUTING AND INFORMATICS

References

- S. J. Matthews, T. Newhall and K. C. Webb, *Dive into Systems*, Version 1.2. Free online textbook, available at: https://diveintosystems.org/book/

- 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.