CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne
Switch statements
**switch Statement**

**Syntax**

```
switch (expression)
{
    case const-expr\(_1\): statements\(_1\)
    :
    case const-expr\(_n\): statements\(_n\)
    default: statements\(_{n+1}\)
}
```

- *expression* is evaluated and tested for a match with the *const-expr* in each *case* clause.
- The *statements* in the matching *case* clause is executed.
**switch Statement**

Flow Diagram (default)

```
START

Const-expr₁=expression?
  YES -> statement₁
  NO

Const-expr₂=expression?
  YES -> statement₂
  NO

... (repeated for each expression)

Const-exprₙ=expression?
  YES -> statementₙ
  NO

statementₙ₊₁

END
```

Notice that each statement falls through to the next

This is the default behavior of the **switch** statement
**switch Statement**

Flow Diagram (modified)

```
START

Const-expr₁ = expression?  
  YES  
  statement₁ break;
  NO

Const-expr₂ = expression?  
  YES  
  statement₂ break;
  NO

...  

Const-exprₙ = expression?  
  YES  
  statementₙ break;
  NO

statementₙ₊₁

END
```

Adding a **break** statement to each statement block will eliminate fall through, allowing only one case clause's statement block to be executed.
switch Statement

Simple example

```
switch (channel) {
    case 2:  puts("WBBM Chicago"); break;
    case 3:  puts("DVD Player"); break;
    case 4:  puts("WTMJ Milwaukee"); break;
    case 5:  puts("WMAQ Chicago"); break;
    case 6:  puts("WITI Milwaukee"); break;
    case 7:  puts("WLS Chicago"); break;
    case 9:  puts("WGN Chicago"); break;
    case 10: puts("WMVS Milwaukee"); break;
    case 11: puts("WTTW Chicago"); break;
    case 12: puts("WISN Milwaukee"); break;
    default: puts("No Signal Available");
}
```
switch Statement

Styling

switch Example 1

```c
switch (channel) {
    case 2:
        puts("WBBM Chicago");
        break;
    case 3:
        puts("DVD Player");
        break;
    case 4:
        puts("WTMJ Milwaukee");
        break;
...
}
```

Case 2:
break;
switch Statement

char letter = 'd';

switch Example 2

switch (letter) {
    case 'a':
        puts("Letter 'a' found.");
        break;
    case 'b':
        puts("Letter 'b' found.");
        break;
    case 'c':
        puts("Letter 'c' found.");
        break;
    default:
        puts("Letter not in list.");
}
switch Statement

Fall-through

switch Example 3

switch(channel) {
    case 4:
    case 5:
    case 6:
    case 7:
        puts("VHF Station");
        break;
    case 9:
    case 10:
    case 11:
    case 12:
        puts("VHF Station");
        break;
    default:
        puts("No Signal Available");
}
switch Statement

Range syntax

switch Example 3

```c
switch(channel) {
    case 4 ... 7:
        puts("VHF Station");
        break;
    case 9 ... 12:
        puts("VHF Station");
        break;
    default:
        puts("No Signal Available");
}
```
switch Statement

Real-world example

switch Example 2

```c
bool IsHex(char character)
{
    switch (character) {
    case 'a' ... 'f' :
    case 'A' ... 'F' :
    case '0' ... '9' :
        return true; break;
    default:
        return false;
    }
}
```
CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne
Software Engineering

Design

Build
Software Engineering
Design process
Software Engineering

Principles

• Use consistent styling

• Summary:
  – Utilize whitespace
  – Good variable/function names
  – Comments that describe non-obvious code behavior
  • "How?" and "why?" are good questions to answer in comments

    i++;
    /\; i = i + 1
Software Engineering

Formatting code

- Ugly code
- Beautiful code
Software Engineering

Formatting non-code

- Comments that describe non-obvious code behavior
  - "How?" and "why?" are good questions to answer in comments

With OS example

```c
// First, determine the length of both items' data, // given NULL data a -1 length so that it sorts to // the head of the list.
int len1 = -1;
if (item1->data) {
    len1 = strlen(item1->data);
}
...
```
Software Engineering

Principles

• Modularity is important

• Why?
  – Supports code reuse
  – Simplifies changes
  – Allows for testing

• How?
  – Keep functions small
  – Minimize side effects
  – Information hiding/encapsulation
Software Engineering

Principles

• Information hiding/encapsulation

• Summary:
  – Hide unimportant details from the user
  – Protects the user from breaking things
  – Separates backend from frontend

  **Static**

  *Setters/getters*

  ```
  setfoo(newfoo)
  foo = newfoo
  ```
Software Engineering

Mantras

- Keep it simple, stupid
  - KISS

- Summary:
  - Don't solve problems you don't need to
  - Don't introduce unnecessary complexity
  - Prioritize for readability and modularity
  - Don't be clever and/or cute
  - Applies to code architecture and specific code constructs

\[ \text{foo} ++ [\text{bar} \text{baz}] = 2 \]
Software Engineering

KISS example

Example

```c
ListItem *LinkedListGetFirst(ListItem *list)
{
    ListItem *tempPointer = NULL;
    if (list == NULL) {
        return NULL;
    }

    if (list->previousItem == NULL && list->nextItem != NULL) {
        return list;
    } else if (list->previousItem != NULL) {
        tempPointer = list;
        while (tempPointer->previousItem != NULL) {
            tempPointer = tempPointer->previousItem;
        }
    }

    return tempPointer;
}
```
Software Engineering

KIIS example

Example

```c
ListItem *LinkedListGetFirst(ListItem *list)
{
    while (list && list->previousItem) {
        list = list->previousItem;
    }
    return list;
}
```
Software Engineering
Mantras

• Don't repeat yourself
  – DRY

• Summary:
  – Write code only once
  – Simplifies refactoring/incremental development
  – Avoids copy/paste errors
Software Engineering
Mantras

• You aren't gonna need it
  – YAGNI

• Summary:
  – Don't introduce features that are unnecessary
  – Don't write more code than you have to
  – Start small and build from there
Software Engineering

Principles

• Principle of Least Astonishment

• Summary:
  – Be consistent with user's expectations
  – Build on user's intuition
  – Applies to users and developers
    • so both the code and library/program functionality
  – Lowers learning curve
Software Engineering

Principle of Least Astonishment

• Functions/variables should have clear names
  – That should match their functionality!
  – Same for comments
• Functions should not do more than you would think
  – Minimize side effects
• Code should be grouped logically
• Functionality should follow precedence if any exists
Software Engineering

Principles

• Garbage in, garbage out

• Summary:
  – "A system's output quality usually cannot be better than the input quality"
  – So bad input results in garbage output
    • Instead of an error condition
  – Can propagate through the system
  – Can be mitigated by checking the input data
Software Engineering

Principles

- Fault tolerant design
- Summary:
  - Plan for operating failures
    - Running out of memory
    - Data being corrupted
  - Provide fallback modes
  - Important for complex software where minor errors can be common
  - Part of defensive programming
Software Engineering

Principles

• Error tolerant design
• Summary:
  – Plan for user errors
    • "Fault tolerant design" applied to the human component
  – Primarily invalid user input
  – Important for complex software where minor errors can be common
  – Part of defensive programming
Software Engineering
Writing fault/error tolerant code

- Check return values for errors!
  - Many functions have special return values when there are errors, these should usually be checked
  - File accesses
  - scanf()
  - malloc()

- Your code should have special error values
  - LinkedList library

- Program should also return error if failure
Software Engineering
Writing fault/error tolerant code

- Errors should be exposed by libraries

**Good library**

```c
int LinkedListSwapData(ListItem *firstItem,
                        ListItem *secondItem);
int LinkedListSort(ListItem *list);
int LinkedListPrint(ListItem *list);
```

**Bad library**

```c
void LinkedListSwapData(ListItem *firstItem,
                        ListItem *secondItem);
void LinkedListSort(ListItem *list);
void LinkedListPrint(ListItem *list);
```
Software Engineering
Writing fault/error tolerant code

- Errors should be exposed by libraries
- And handled by the program
- Not all errors can be recovered from
  - Fatal errors

Embedded example
```c
int main(void) {
    if (!DataStoreInit()) {
        FATAL_ERROR();
    }
}
```

With OS example
```c
int main(void) {
    if (!DataStoreInit()) {
        return DATASTORE_ERROR;
    }
}
```
Software Engineering

Principles

• Eating your own dogfood
• Summary:
  – When engineers use their own creations, they're generally better
  – More likely that bugs are fixed, features are added because they directly impact the developers
  – In use by all of industry
  – I do it
Software Engineering

Pitfalls

- Premature Optimization
  - "root of all evil"

- Summary:
  - Optimizing code before performance is a critical factor
  - Optimizing reduces readability & modularity
  - Optimization not required for a lot of code
    - See Amdahl's Law
    - See KISS
Software Engineering

Teamwork

• Working as a group is the most challenging engineering practice
• Requires:
  – Good communication
• That's it!
Software Engineering

Teamwork

• Pair programming

• Summary:
  – Two developers work side by side: one driving, the other navigating
  – Just like driving:
    • Driver writes code
    • Navigator plans ahead, thinks of edge cases, double-checks driver
  – Requires frequent role switching to be effective!
Software Engineering

Teamwork

- Division of labor
- Summary:
  - Divide work into tasks that can be split between team members
  - Requires coordination to not step on each other's toes
  - **Documentation is very important!**
  - Can be useful to split testing and development between different people
CMPE-013/L

Bounce (or Hardware)

Maxwell James Dunne
Bounce

- Digital I/O
- A/D
- Timers
- Debouncing
Static Events;

main() {
    
    Timer1 Init {
        Events = BCE();
    }

    3
    3
    3
Debouncing
Last event = Down
1) start with one button
   BTN 4

2) copy & paste

3) User input
Lighting all

00000