STYLE

Style Guidelines

Learning Outcomes

After reading this section, you will be able to:

- Self-document programs using comments and descriptive identifiers

Introduction

A well-written program is a pleasure to read. The coding style is consistent and clear throughout. The programmer looking for a bug sees a well-defined structure and finds it easy to focus on the portion of the code that is suspect. The programmer looking to upgrade the code sees how and where to incorporate changes. Although several programmers may have contributed to the code throughout its lifetime, the code itself appears to have been written by one programmer.

This chapter describes the coding style used throughout these notes and recommended for an introductory course in programming. The style is referred to as the Allman coding style.

Identifiers

All identifiers in a program should be self-descriptive. The reader should not have to search through the code for their meaning. It is better to embed the meaning in the name, rather than to explain it in a comment elsewhere in the code. By all means, avoid referring the reader to a document external to the code itself.

A program with short names is easier to read than one with long names. The human eye infers the meaning of a word from just a few letters that make up that word and the context within which the word is used. Reading long identifiers tires the eyes when searching through code. We follow the sophisticated conventions of our own languages and complying with them makes our programs all the more readable. Nouns describe objects, verbs describe actions.

Notations, such as Hungarian notation, that incorporate the type into the identifier will clutter source code unnecessarily. C compilers know the type of each identifier and readers do not need reminders in every place the identifier appears.

When selecting identifiers:

- adopt self-descriptive names, adding comments only if clarification is necessary
- prefer nouns for variable identifiers
- keep variable identifiers short - `temp`, rather than `temporary`, `id`, rather than `identification`,
- avoid cryptic identifiers - use just enough letters for the eye to infer the meaning from the context but no less (if you want to represent 'amount owed', `ao` is cryptic, while `amtOwed` is clear but concise)
- keep the identifiers of counters very short - use `i` rather than `loop_counter`, and `n` rather than `numberOfIterations`. This is context dependant and should only be applied to iterators and counters otherwise, the name becomes meaningless or cryptic.
- avoid decorating the identifier with Hungarian or similar notations (data type)
- use "camelNotation" (first letter of each word is capitalized with the exception of the first word)
- avoid underscore characters which are commonly used in system libraries to avoid conflicts

Layout

Professionals in the field of human-computer interaction confirm that layout and arrangement affects comfort and accessibility. Poorly laid out code frustrates and promotes misreading's.

Typographers, artists, and photographers know that negative space surrounding an image is as important as the image itself. Space itself can visually separate, making it easier to find something and draw attention to a certain part of a page.

Layout tools at our disposal include:

- indentation

- line length
- braces
- spaces
- comments

Indentation

Indentation helps define where a code block starts and ends, clearly showing the structure of our logic. The recommended indent in C programs is a tab of 4 or 8 characters.

Example:

```
for (i = 0; i < n; i++)
{
    for (j = 0; j < n; j++)
    {
        for (k = 0; k < n; k++)
        {
            int ijk = i * j * k;
            if (ijk != 0)
            {
                printf(" %4d", ijk);
            }
            else
            {
                printf("    ");
            }
        }
        printf("\n");
    }
    printf("\n");
}
printf("That's all folks!!!\n");
```

Using tabs for indentation rather than spaces enables other programmers to adjust the indentation without difficulty in their own text editors. Using 8 characters per tab position heavily indents code to the far right and identifies code that may be a hog of compute cycles and a likely candidate for refactoring.

To minimize the effects of indentation with switch constructs, we align the subordinate case labels with the switch keyword:

```
switch(c)
{
case 'A' :
case 'a' :
        cost = 1.50;
        break;
case 'B' :
case 'b' :
        cost = 1.10;
        break;
case 'C' :
case 'c' :
        cost = 0.75;
        break;
default:
        c = '?';
        cost = 0.0;
}
```

Line Length

The practical limit on line length is 80 columns, including indentation. Many windows default to an 80-column width and break longer lines into chunks that are then difficult to read. Lines longer than 80 columns either truncate or wrap in hard copy printouts, which confuses readers.

String literals pose a special problem. We break them into a set of sub-string literals separated only by whitespace. C compilers discard the whitespace and concatenate the sub-string literals into a single string literal.

```
printf("%d substrings"
       " display as a"
       " single string"
       "\n", 3);
```

Produces the following output:

```
3 substrings display as a single string
```

Braces

The style of bracing used in these notes is that proposed by Eric Allman. We put the opening brace on its own line indented to the preceding statement and the closing brace on its own line in alignment with the opening brace.

```
if (i == 7)
{
    cost = 1.75;
    printf("Congrats!\n");
}
```

```
if (i == 7)
{
    cost = 1.75;
    printf("Congrats!\n");
}
else
{
    cost = 2.75;
    printf("Good luck next time!\n");
}
```

The opening and closing braces for a `do/while` construct is an exception:

```
do {
    printf("Guess i : ");
    scanf("%d", &i);
    if (i == 7)
    {
        cost = 1.75;
        printf("Congrats!\n");
    }
    else
    {
        cost = 2.75;
        printf("Good luck next time!\n");
    }
} while (i != 7);
```

Although braces are unnecessary with single statements, it is more clear to read and maintain when they are provided:

```
if (i == 7)
{
    printf("Congrats!\n");
}
else
{
    printf("Good luck next time!\n");
}
```

```
if (i == 7)
{
    printf("Congrats!\n");
    done = 1;
}
else
{
    printf("Good luck next time!\n");
}
```

Spaces

We add a single space after commas, semi-colons, most keywords and around most operators, except between parentheses and identifiers or constants, after unary operators and call identifiers.

For example:

```
int i;// space after keyword

scanf("%d", &i);// no space after unary operator

i = i * i;// single spaces around binary operators

if (i == 7)// no spaces between identifiers or constants and parentheses
{
    printf("Congrats!\n");
}

for (i = 0; i < 10; i++)// single space after ;
{
    printf("%d ", i);// no space after call identifier
}
```

We avoid trailing spaces at the end of a line.

We add blank lines to distinguish the end of one construct from the start of another whenever either construct contains some complexity. However, we avoid superfluous blank lines.

Comments

We use comments to describe what is done, rather than how it is done. Comments introduce or summarize what follows. We keep them brief and avoid decoration or cuteness.

If it is important to comment data, we do so at the variable's declaration. Where units matter, we identify them. Where we comment variable declarations, we declare each variable on a separate line and use inline comments.

We preface every source file with a header comment that includes:

- the title of the program
- the source file name
- the name of the author(s)

For example:

```
/* Payroll Deductions
  * payroll.c
  * Jane Doe
  */
```

Such header comments are helpful in locating the e-copy corresponding to a hard copy that we have in hand.

We align comments with the code they describe, indenting both identically, showing no preference for either comment or code.

```
// display even integers below 11
//
for (j = 0; j < 11; j += 2)
{
    printf(" %d", j);
}
```

Such comments summarize the code that follows and help the reader avoid studying that code in detail if it is not the target code for which they are searching.

Magic Numbers

We refer to values that appear out of nowhere in program code as magic numbers. These may be mathematical constants, standard rates or default values. We avoid magic numbers by identifying them with symbolic names and using those names throughout the code. We set their value in either of two ways:

- using an unmodifiable variable
- using a macro directive

Unmodifiable Variables

An **unmodifiable** variable takes the form

```
 const type SYMBOL = value;
```

For example:

```
const double PI = 3.14159;

int main(void)
{
    double radius, area;

    printf("Enter radius : ");
    scanf("%lf", &radius);

    area = PI * radius * radius;
    printf("The area of your circle is : %lf\n", area);

    return 0;
}
```

Macro Directive

A macro is **NOT** a variable but is used for substitution at compile-time. A macro directive takes the form:

```
#define SYMBOL value
```

We terminate this directive with an end of line character immediately following value.

The `define` directive instructs the C compiler to **substitute** every occurrence of SYMBOL with value throughout the code.

NOTE

Notice the absence of a semi-colon at the end of the directive. The substitution is a straightforward **search** and **replace**. The value itself may include whitespace.

For example,

```
#define PI 3.14159

int main(void)
{
        double radius, area;

        printf("Enter radius : ");
        scanf("%lf", &radius);

        area = PI * radius * radius;
// At compile-time, the above statement becomes:
// area = 3.14159 * radius * radius

        printf("The area of your circle is : %lf\n", area);

        return 0;
}
```

Miscellaneous

Other guidelines for enhancing and maintainability readability include:

- we avoid global variables (see Global Scope sub-section in the chapter on Information Hiding)
- we avoid variable identifiers that end in numbers
- we avoid using the character encodings for a particular environment (for example, ASCII or EBCDIC). Instead, we use escape sequences, which are universal. For

example, to initialize c to the linefeed character (10 in ASCII and 37 in EBCDIC), use:

```
// prefer
//
char c = '\n';

// avoid
//
char c = 10;// ASCII
```

- we initialize iteration variables in the context of the iteration:

```
// prefer
//
for (i = 0; i < 10; i++)

// avoid
//
i = 0;
for (; i < 10; i++)
```

- we add special comments where code has been fine-tuned for efficient execution
- we avoid iterations with empty bodies
- we limit the initialization and iteration clauses in for statements to the iteration variables
- we avoid assignment expressions nested inside logical expressions
- we add an extra pair of parentheses where an assignment expression is also used as a condition
- we remove unreferenced variable declarations from our source code
- we remove all commented code and debugging statements from our release and production code

```
```