

# COMP2521 20T1 ♦ Programming Style

---

- COMP1511 Style
- COMP2521 Style
- Poor Style
- Comments
- Use of Brackets
- Assignment in Expressions
- Conditional Expressions
- Control Structures
- Switch-statements
- For-loops
- **break** and **continue**
- Functions and **return**
- Relaxed Style



## ❖ COMP1511 Style

---

Required use of a restricted subset of C:

- layout, use of brackets (always)
- use only **if**, **while** and **for**
- no side-effects in expressions
- no conditional expressions
- all functions have one return statement

But ... this style is not used in texts or real code.

## ❖ COMP2521 Style

---

Extends the range of allowed constructs:

- to better reflect how C is used in books and online

Some things will not change:

- consistent use of indentation
- indentation reflecting the nested control structures
- meaningful names for functions and variables\*
- use *one* style throughout one software system

\* unless the variable is an array index and/or used in a very limited scope

## ❖ Poor Style

---

Examples of poor style:

```
int fff(int n)
{
    int flab = 1;
    if (n < 1) return -1;
    for (int z = 1; z <= n; z++)
        flab = flab * z;
    return flab;
}
```

```
int ff(int n) {
int f = 1;  if (n < 1) return -1;
for (int xy = 1; xy <= n; xy++) f *= xy;
return f; }
```

## ❖ Comments

---

COMP1511 used (exclusively?) `/*...*/`  
comments

Many books, code-bases use `//... comments`

Either is ok, but prefer

- `//` for short comments at end of line

```
int nc; // count of characters
```

- `/*...*/` for extended comments, e.g. at start of function

(and C doesn't support `#... style`, since `#` used for e.g. `#include`)

## ❖ Use of Brackets

Put control-group start bracket after conditional expression

Can omit brackets if control structure owns a single statement

Examples:

```
if (x > 0) {  
    y = y * x;  
}
```

or

```
if (x > 0)  
    y = y * x;
```

or even (slightly naughty)

```
if (x > 0) y *= x;
```

```
while (*c != '\0') {  
    c++;  
}
```

```
while (*c != '\0')  
    c++;
```

```
while (*c != '\0') c++;
```

## ❖ ... Use of Brackets

---

If condition followed by **return**, **continue**, **break**, use one line, e.g.

```
// handle incorrect parameter
if (x < 0) return -1;

// early exit from loop
for (c = str; *c != '\0'; c++) {
    if (*c == 'z') break;
    ... process next char in string ...
}

// ignore spaces in string
for (c = str; *c != '\0'; c++) {
    if (isspace(*c)) continue;
    ... process non-space char ...
}
```

## ❖ ... Use of Brackets

---

Can put function start bracket on line after function header, e.g.

```
int myFun(parameters) {  
    ... function body ...  
}
```

or

```
int myFun(parameters)  
{  
    ... function body ...  
}
```

or

```
int  
myFun(parameters) { // name at start of line  
    ... function body ...  
}
```



## ❖ Assignment in Expressions

Can use assignment statements in expressions, e.g.

```
// assign same value to multiple variables
```

```
i = j = k = 0;
```

or

```
i = (j = (k = 0));
```

or

```
k = 0; j = 0; i = 0;
```

```
// scan stdin, char-by-char
```

```
while ((ch = getchar()) != EOF) {
```

```
    ...process next char...
```

```
}
```

but you should try to minimise their use in this way

## ❖ Conditional Expressions

---

Conditional expressions return a value, based on a test

Handle a moderately common practical case:

```
if (x > 0)
    y = x + 1;
else
    y = 0;
```

can be expressed as

```
y = (x > 0) ? x+1 : 0;
```

Requires: same variable in both **if** branches; one statement in each branch.

## ❖ Control Structures

Can use more C control structures

- **if, switch, while, do, for, break, continue**
- but NOT **goto, setjmp(), longjmp()**

Examples:

```
ch = getchar();
while (ch != EOF) {
    if (isalpha(ch)) nalpha++;
    ch = getchar();
}
or
do {
    ch = getchar();
    if (isalpha(ch)) nalpha++;
} while (ch != EOF);
or
while ((ch = getchar()) != EOF) {
    if (isalpha(ch)) nalpha++;
}
```

## ❖ Switch-statements

---

**switch** encapsulates a common selection:

```
if (v == C1) {  
    S1;  
} else if (v == C2) {  
    S2;  
}  
...  
else if (v == Cn) {  
    Sn;  
}  
else {  
    Sn+1;  
}
```

## ❖ ... Switch-statements

---

Multi-way **if** becomes:

```
switch (v) {  
  case  $C_1$ :  
     $S_1$ ; break;  
  case  $C_2$ :  
     $S_2$ ; break;  
  ...  
  case  $C_n$ :  
     $S_n$ ; break;  
  default:  
     $S_{n+1}$ ;  
}
```

Note: **break** is critical; if not present, falls through to next case.

## ❖ ... Switch-statements

Example of "fall-through" (when **break** absent):

```
switch (ch) {  
  case 'a': printf("a\n");  
  case 'b': printf("b\n"); break;  
  case 'c': printf("c\n"); break;  
  case 'd': printf("d\n");  
  default: printf("?"); // break optional here  
}
```

- if **ch** == **'a'**, then prints **'a'** and **'b'**
- if **ch** == **'b'**, then prints only **'b'**
- if **ch** == **'c'**, then prints only **'c'**
- if **ch** == **'d'**, then prints **'d'** and **'?'**

## ❖ For-loops

---

**for** encapsulates a common loop pattern:

```
initialise;
while (Continuation) {
    do stuff;
    increment;
}
```

as

```
for (initialise; Continuation; increment) {
    do stuff;
}
```

## ❖ break and continue

These constructs affect how a loop operates, e.g.

```
while (Continuation) {  
    ... do stuff1 ...  
    if (Test1) continue;  
    ... do stuff2 ...  
    if (Test2) break;  
    ... do stuff3 ...  
}
```

- ***stuff*<sub>1</sub>** is always executed
- if ***Test*<sub>1</sub>** succeeds, go straight to ***Continuation*** test
- if ***Test*<sub>1</sub>** fails, then execute ***stuff*<sub>2</sub>**
- if ***Test*<sub>2</sub>** succeeds, terminate the loop
- if ***Test*<sub>2</sub>** fails, then execute ***stuff*<sub>3</sub>** and do next iteration



## ❖ Functions and return

---

COMP1511 and "proper" style suggest that ...

- all functions should have one **return**, at the end

Pragmatically, multiple **returns** can be useful to ...

- handle errors (escape with error return value)
- simplify logic in later parts of function

## ❖ ... Functions and return

Example: compute  $n!$ ; return **-1** if error; no overflow check

```
int factorial(int n)
{
    int fac = 1;
    if (n < 1) return -1; // error return
    for (int i = 1; i <= n; i++) {
        fac = fac * i;
    }
    return fac; // return result
}
```

```
int factorial(int n)
{
    if (n < 1) return -1;
    else if (n == 1) return 1;
    else return n * factorial(n-1);
}
```

## ❖ ... Functions and return

Example: search for **key** in array **a[]** of length **n**

```
int search(int key, int a[], int n)
{
    int where = -1; // not found value
    for (int i = 0; i < n; i++) {
        if (a[i] == key) where = i;
    }
    return where; // return result or not found
}
```

or

```
int search(int key, int a[], int n)
{
    for (int i = 0; i < n; i++) {
        if (a[i] == key) return i; // return result
    }
    return -1; // not found value
}
```

## ❖ Relaxed Style

---

**Good:** gives you more freedom and power

- more choice in how you express programs
- can write code which is more concise (simpler)

**Bad:** gives you more freedom and power

- can write code which is more cryptic
- can lead to **incomprehensible, unmaintainable** code

So, you **must** still use some discipline.

Produced: 1 Jun 2020