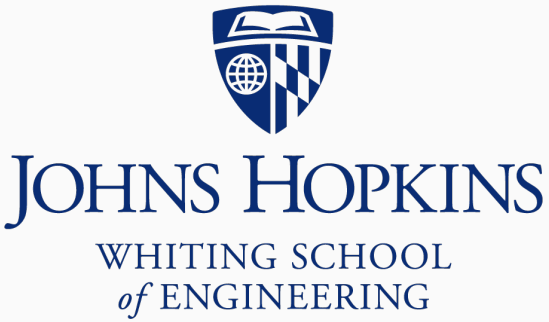
Decisions

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Decisions

```c
#include <stdio.h>

int main() {
    int number = 12;
    if(number % 2 == 0) {
        printf("yes\n");
    }
    return 0;
}
```

Does it say “yes”?

if example

```c
#include <stdio.h>

int main() {
    int number = 12;
    if(number % 2 == 0) {
        printf("yes\n");
    }
    return 0;
}
```

```
$ gcc -o if1 if1.c -std=c99 -pedantic -Wall -Wextra
$ ./if1
yes
```

if example

```c
#include <stdio.h>

int main() {
    int number = 13;
    if(number % 2 == 0) {
        printf("yes\n");
    }
    else {
        printf("no\n");
    }
    return 0;
}
```

```
$ gcc -o if2 if2.c -std=c99 -pedantic -Wall -Wextra
$ ./if2
no
```

if example

```c
#include <stdio.h>

int main() {
    int number = 13;
    if(number = 1) {
        printf("yes\n");
    }
    else {
        printf("no\n");
    }
    return 0;
}
```

if / else

```
$ gcc -o if3 if3.c -std=c99 -pedantic -Wall -Wextra
if3.c: In function 'main':
if3.c:5:8: warning: suggest parentheses around assignment used as truth value
[-Wparentheses]
     if(number = 1) {
        ^~~~~~
$ ./if3
yes
```

if / else

Careful: Expression `number = 1` evaluates *to the assigned value*, which is 1

- Makes it easier to confuse `number = 1` and `number == 1`

Fortunately, modern compilers can warn us if we mistakenly use assignment in a condition

if / else if / else

```c
#include <stdio.h>
int main() {
    int x = 79;
    if(x >= 90) {
        printf("A\n");
    }
    else if(x >= 80) {
        printf("B\n");
    }
    else if(x >= 70) {
        printf("C\n");
    }
    else if(x >= 60) {
        printf("D\n");
    }
    else {
        printf("F\n");
    }
    return 0;
}
```

if / else if / else

```
$ gcc -o grading grading.c -std=c99 -pedantic -Wall -Wextra
$ ./grading
C
```

switch / case

```c
#include <stdio.h>
int main() {
    char grade = 'C';
    int points = 0;
    switch(grade) {
        case 'A':
            points = 4;
            break;
        case 'B':
            points = 3;
            break;
        case 'C':
            points = 2;
            break;
        case 'D':
            points = 1;
            break;
        default:
            points = 0;
    }
    printf("Grade %c contributes %d GPA points\n", grade, points);
    return 0;
}
```

switch / case

```
$ gcc grading_switch.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
Grade C contributes 2 GPA points
```

Compound assignments

Compound assignment operators perform an operation with a variable operand and assign the result back to teh variable

Example	Equivalent
`a += 5`	`a = a + 5`
`a -= 5`	`a = a - 5`
`a *= 5`	`a = a * 5`
`a /= 5`	`a = a / 5`
`a %= 5`	`a = a % 5`

Increment and decrement

Increment and decrement operators act like `+= 1` and `-= 1`

Example	Equivalent	Evaluates to
`a++`	`a += 1`	original (smaller) a
`++a`	`a += 1`	new (larger) a
`a--`	`a -= 1`	original (larger) a
`--a`	`a -= 1`	new (smaller) a

for

```c
#include <stdio.h>
int main() {
    for(int i = 0; i < 10; i++) {
        printf("%d ", i);
    }
    return 0;
}
```

```
$ gcc -o for_example for_example.c -std=c99 -pedantic -Wall -Wextra
$ ./for_example
0 1 2 3 4 5 6 7 8 9
```

while

```c
#include <stdio.h>
int main() {
    int i = 1;
    while((i % 7) != 0) {
        printf("%d ", i);
        i++;
    }
    return 0;
}
```

```
$ gcc while_example.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
1 2 3 4 5 6
```

do / while

```c
#include <stdio.h>
int main() {
    int i = 0;
    do {
        printf("%d ", i);
        i++;
    } while((i % 7) != 0);
    return 0;
}
```

```
$ gcc do_while_example.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
0 1 2 3 4 5 6
```

scanf loop

```c
#include <stdio.h>
int main() {
    int sum = 0;
    while(1) {
        int addend = 0;
        if(scanf("%d", &addend) != 1) {
            break; // immediately exit loop
        }
        sum += addend;
    }
    printf("%d\n", sum);
    return 0;
}
```

This continues to scan even when you press enter. To signal end-of-input, press `Ctrl-D` (possibly twice).

Nesting

Loops can be nested inside other loops

Loops & decision statements can be arbitrarily nested

- We use terms *outer* & *inner* to distinguish levels of nesting

```
while(...) {            // OUTER LOOP
    if(...) {
        do {            // INNER LOOP
            if(...) {
                ...
            } else {
                ...
            }
        } while(...);
    }
}
```

break & continue

break exits a loop, continue advances to the next iteration

They affect the *most immediate loop* containing the break/continue.

```
while(...) {            // OUTER LOOP
    if(...) {
        break;          // exit outer loop
    }
    do {                // INNER LOOP
        if(...) {
            continue; // advance inner loop
        }
    } while(...);
}
```

break/continue work *only* with loops, not with if/else, functions, etc

Code blocks

In examples so far, we used `{...}` to delimit a block of code:

```
while(...) {
    if(...) {
        break;
    }
    do {
        if(...) {
            continue;
        }
    } while(...);
}
```

Code blocks

If block consists of single statement, can omit { and }

```
while(...) {
    if(...) break;          // { } omitted
    do {
        if(...) continue; // { } omitted
    } while(...);
}
```

Loop summary

- `while(boolean expression) { statements }`
  - Iterates ≥ 0 times, as long as expression is true
- `do { statements } while(boolean expression)`
  - Iterates ≥ 1 times; always once, then more times as long as expression is true
- `for(initialize; boolean exp; update) { stmts }`
  - `initialize` happens first; usually declares & assigns "index variable"
  - Iterates ≥ 0 times, as long as boolean expression is true
  - Right after `stmts`, `update` is run; often it increments the index variable (`i++`)
- `break` immediately exits loop
- `continue` immediately proceeds to next iteration of loop