

COMP1511 Week 2

Variables and if-Statements

Joanna Lin

Reminders

- Please log onto the course forum if you haven't already
- Respond to my welcome email titled 'Welcome to COMP1511'. If you can't find that email and other course-related emails, please send me an email at z5311239@unsw.edu.au.
- Week 2 labs are due **Monday 8pm** in **week 3**.
- Assignment 0 is out. Due **Monday 8pm** in **week 4**.

What we'll cover today

variables and if-statements

- **Variables**
 - **Declaring** and **initialising** variables.
 - **Printing** and **scanning** in variables.
 - **Arithmetic operators.**
 - Difference between variables and **#defines**
- **if-statements**
 - Why do we need if-statements?
 - How do we structure if-statements?
 - **Logical operators.**

As we go along, we'll be building a program that scans dimensions of a rectangle from a user and prints facts about it.

Variables

Declaring and Initialising

- Programs consist of two fundamental components: storing information and doing stuff with that information. Variables allow us to store information.
- **Declaring** a variable means informing the program that the variable exists.
 - In C, we must provide the type and variable name: `int width;`
 - The computer knows to assign a block of memory of size appropriate for that type.
 - Types: `int` (whole number e.g. 5), `double` (decimal point e.g. 5.1), `char` (single character e.g. 'a')
 - A character is stored as an integer representing the ASCII value of character. Use command `ascii -d` to see the numbers corresponding to each letter.
 - This means we can do arithmetic like `'a' + 1`, which sums together 1 and the ASCII value of 'a' (which is conveniently 'b').
 - **Style tip:** the variable name should be descriptive where possible.
- **Initialising** a variable means giving it a value.
 - We can either initialise the variable on a line after declaration or on the same line as declaration:

```
int width;  
width = 5;
```

OR

```
int height = 10;
```

Printing and Scanning Variables

- Printing and scanning variables requires format identifiers
 - `%d` to print integers (or ASCII value of characters) in base 10.
 - `%lf` to print doubles.
 - `%c` to print single characters

- When printing, the format identifiers within the double quotes are replaced in order by the list of values given after.

```
printf("Width: %d, Height: %d\n", width, height);
```

- When scanning, we accept input in a certain format, which are then assigned in order to the variables on the right.
- `&` indicates the address of the variable, so that `scanf` knows where to put the value

```
int width;  
int height;  
scanf("%d %d", &width, &height);
```

Variables

Changing their values

- The point of variables is that we can use and change their value.
- Think of `=` as an assignment operator, rather than 'equals to'.
 - We're evaluating the right and assigning that value to the variable on the left.

Step 1

```
// Swap x and y
```

```
int temp = x;
```

Step 2

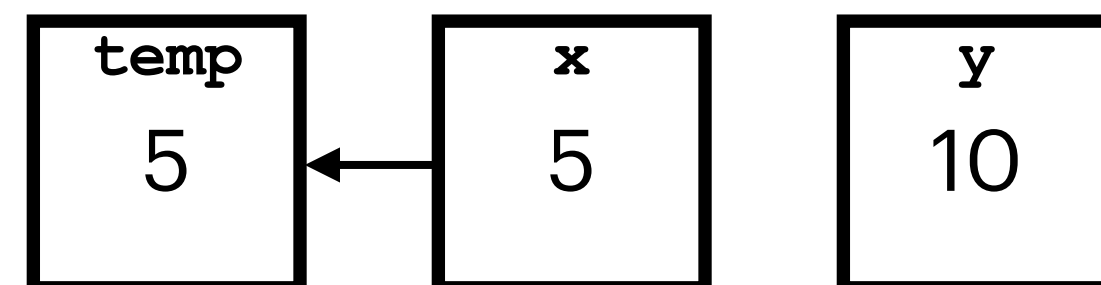
```
x = y;
```

Step 3

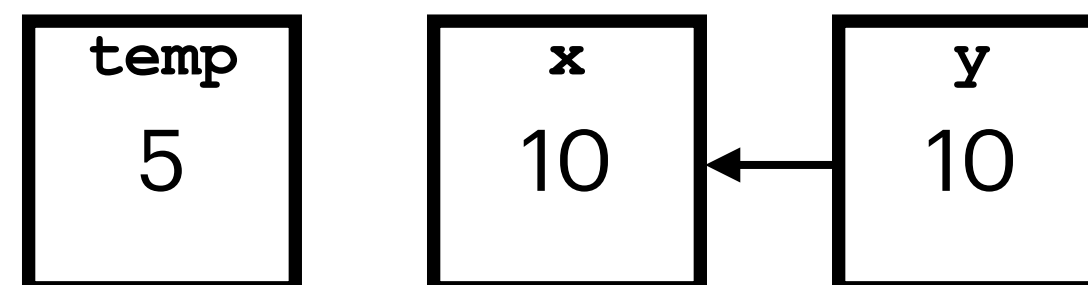
```
y = temp;
```

Memory Model

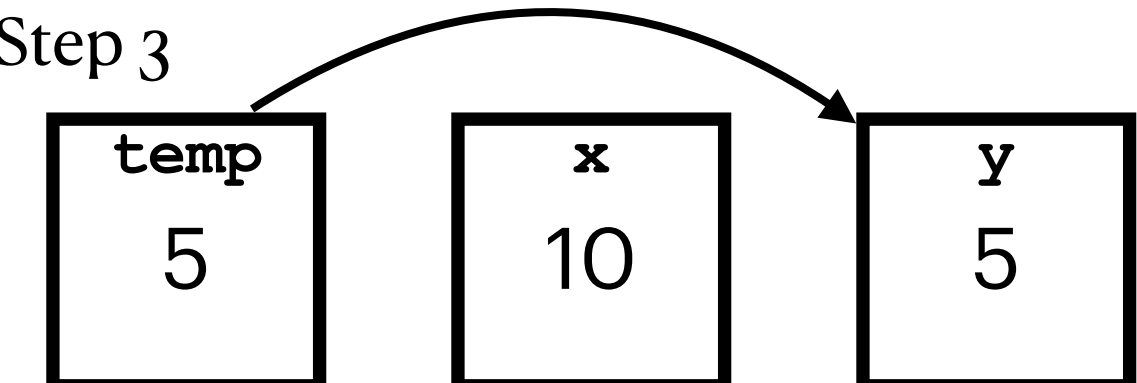
Step 1



Step 2



Step 3



Each box corresponds to the block of memory given to the variable upon declaration.
We're assuming that variables `x` and `y` have been initialised to 5 and 10 respectively.

Arithmetic Operators

Doing quick maff

- Basic operations:

- + addition
- - subtraction
- * multiplication
- / division
- % remainder

- Note that:

- decimal part is ignored when dividing two integers (e.g. $2 / 3$ evaluates to 0)
- the remainder operator exhibits weird behaviour when dealing with negatives

Basic examples

```
int area = height * width;
int perimeter = 2 * (height + width);
```

Printing

```
printf("Area: %d, Perimeter: %d\n", area, perimeter);
```

```
printf("Area: %d, Perimeter: %d\n", height * width, 2 * (height + width));
```

Adding a variable to itself

```
int perimeter = 0;
perimeter += 2 * height;
perimeter += 2 * width;
```

is shorthand for

```
int perimeter = 0;
perimeter = perimeter + 2 * height;
perimeter = perimeter + 2 * width;
```


#define'd constants

What are they and how do they differ from variables?

- **#define**'d constants are important for making our code more readable.

```
#define MAX_LENGTH 45000
```

- They cannot change from the provided value, unlike a variable (hence 'constant')
- **#define** is a preprocessor directive.
 - A preprocessor directive is dealt with at the compilation stage. The compiler acts upon these directives, altering the C source code before machine code is produced.
 - In the case of **#define**, the compiler replaces every instance of the constant with its actual value.
 - This means that, when you actually run your program, a **#define**'d constant doesn't actually take up memory, unlike a variable

if-statements

Why do we need them and how do we write them?

- We don't want to always run the same piece of code. What if the user enters a negative side length, or if we want to safeguard against integer overflows?
 - We can use if-statements to run code only if a given condition is true.
- All if-statements must start with the **if** keyword, followed by a condition enclosed in brackets and a set of curly braces within which we write code. Optionally, we can add **else if**(s) and/or an **else** to account for other cases when needed.
- **Style tips**
 - code within curly braces are indented 4 spaces further into the line than the line containing the **if**
 - closing curly brackets are aligned with the line containing the **if**.
 - **else if** and **else** should be on the same line as the closing curly bracket corresponding to the previous **if** or **else if**

```
if (condition1) {  
    // run if condition1 is true  
} else if (condition2) {  
    // run if condition2 is true  
    // AND condition1 is false  
} else {  
    // run if condition1 and condition2  
    // are false  
}
```

ifs vs ifs and else ifs

Important difference

```
if (condition1) {  
    // run if condition1 is true  
}  
if (condition2) {  
    // run if condition2 is true  
}
```

When we have if-statements one after another, every condition is checked, regardless of whether the previous condition was true or false.

```
if (condition1) {  
    // run if condition1 is true  
} else if (condition2) {  
    // run if condition2 is true  
    // AND condition1 is false  
}
```

When we chain together **if** and **else ifs**, only a maximum of one branch of the chain is run — the first whose condition is true. Once we find a true condition, the rest of the chain is skipped.

Logical operators

Writing conditions

- `==`, `!=` equal to, not equal to
- `>`, `>=` greater than, greater than or equal to
- `<`, `<=` less than, less than or equal to
- `&&` logical and (true if left and right is true)
- `||` logical or (true if left or right is true)
- Conditions written with logical operators evaluate to either 0 or 1, corresponding to false and true respectively.
- This means we can store a long condition inside an `int` variable, improving readability

Example use

```
if (0 <= width && width <= MAX_LENGTH && 0 <= height && height <= MAX_LENGTH) {  
    // User entered valid coordinates  
    ...  
} else {  
    printf("Please enter positive lengths less than %d\n", MAX_LENGTH);  
}
```

```
int are_lengths_valid = 0 <= width && width <= MAX_LENGTH &&  
                        0 <= height && height <= MAX_LENGTH;  
if (are_lengths_valid) {  
    // User entered valid coordinates  
    ...  
} else {  
    printf("Please enter positive lengths less than %d\n", MAX_LENGTH);  
}
```

Refactoring Logic

Error-checking first approach

- We often prefer to exit the program if we find any errors first.
- This makes our code a lot less nested (less indented), so it's much neater.
- We can get all the invalid (usually simpler) cases out of the way first, giving us a peace of mind when writing the rest of the program.

Original approach (okay)

```
if (are_lengths_valid) {  
    // User entered valid coordinates  
    ...  
} else {  
    printf("Please enter positive lengths less than %d\n", MAX_LENGTH);  
}
```



Check for errors first (better!)

```
if (!are_lengths_valid) {  
    printf("Please enter positive lengths less than %d\n", MAX_LENGTH);  
    return 1;  
}  
  
// User entered valid coordinates  
...
```