C Lecture 2 Outline

- Casting between Types
- Variables
- Printing to the screen – printf()
- Flow Control

Conditional Branching
- The `if` statement.
- The `switch,case` construct.

Loops
- The `do... while` construct.
- The `for` loop.
- Breaking out of loops - `continue` and `break`

Casts: Converting types

In Lecture-1

* As mentioned above, some operators automatically change the type of their operands so that they all match in type.
* We can also explicitly change the type of a value using a `cast`.
* Specifying the name of a type surrounded by `( )` tells the compiler to convert the following value to the indicated type.
* For example, we can cast a `float` to a `double` like so:

```
myDouble = (double) myfloat;
```

* Obviously, casting a value to a type which does not support it (casting to int a value bigger than the largest value an int can store, for example) will cause a loss of information or unexpected effects.

Variables

* So far, we can perform calculations, but we have nowhere to store their results.
* *Variables* provide a way to store values of a given type, with an associated identifier ("variable name") to refer to the storage location.
* Variable names may be of any length and can be made up of any combination of letter, digit and _ characters which do not look like a literal value or a C *keyword*. Cannot start with a number
* Because the identifier refers to the *storage location*, we can change the value stored in a given variable with no inconsistency.
* Variables must be declared with a type, which defines how values stored in them will be interpreted.

Declaring Variables

* Before you can use a variable identifier, you must *declare* it, letting the compiler know that you are going to want to use a name to refer to a location for storing data of a given type.
* Variable declarations are of the form type of data we want to store names we want to use for the storage locations, where we separate names by commas and end with a ;.

```
int a ;
double b, c;
```

Declaring Variables

* If you use the simple declaration form, then C does not guarantee what value a variable will contain to start with.

```
int a ;
double b, c;
```

Operators on Variables.

* All of the previously mentioned operators will work on the values stored in variables (with appropriate types).
* In addition, there are some operators specifically designed to modify the values stored in variables.

Operation	Symbol	Precedence	Example
Assignment	=	Low	a = 2; (a contains value 2)
Add to stored value	+=	Low	a += 2 (a == 4)
↑ Equivalent versions exist for the other arithmetic operators. ↑			
Increase / Decrease by 1	++/--	High	a++ (a == 5)

a += 2; is equivalent to a = a + 2; += -= *= /=

Gotchas…

E.g. if (x == 4) "do something"
This uses the equality operator == to test if variable x is equal to the value 4, or not.

On the other hand, writing:
if (x = 4) "do something"
This (incorrect statement) uses the assignment operator = to set the variable x to 4. The if-test will always evaluate "true" (x is "non zero") because the value of x has been set to four!

a++ increments a by one; ++a does the same; but we will see that they behave differently in some circumstances:

Eg. if(a++ == 3) is not necessarily the same as if (++a == 3). The first performs the equality test first and then increments 'a'; the latter does the increment first. So if a=2 then the first test returns false but the second test returns true!

The printf() Function

A standard C function included in the library <stdio.h>-used to print to the screen.

Standard form is: printf(format, expression-1, expression-2, …)

Example: printf("Twice %d is %d\n",i, j);

"Format" "Expressions"

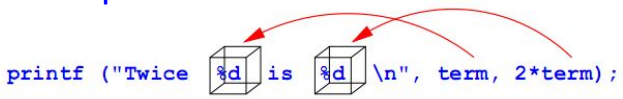
```
#include <stdio.h>

    int main()
    {
        int i, j;
        i = 3;
        j = i * 2;
        printf("Twice %d is %d\n",i, j);
        return 0;
    }
```

The %d is called the "conversion" and the "d" means this is an integer-conversion which tells the compiler to interpret the arguments (i and j) as integers. For floating point numbers, %f is used.

Note: You have to use the conversion that correctly matches your expression.

Definition contains "expression-n" (not "variable-n") because could replaced j with i * 2

printf Format Statements



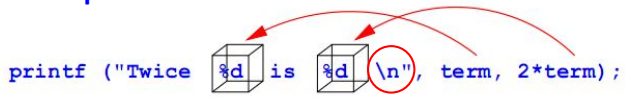
Besides %d there are other conversion specifications, here is an overview of the most important ones:

Conversion	Argument Type	Printed as
%d	integer	decimal number
%f	float	[-]m.dddddd (details below)
%X	integer	hex. number using A..F for 10..15
%c	char	single character
%s	char *	print characters from string until '\0'
%e	float	float in exp. form [-]m.ddddde±xx

In addition, the precision and additional spaces can be specified:

%6d	decimal integer, at least 6 characters wide
%8.2f	float, at least 8 characters wide, two decimal digits
%.10s	first 10 characters of a string

printf Format Statements



Special characters or escape characters starting with '\' move the cursor or represent otherwise reserved characters.

Character	Name	Meaning
\b	backspace	move cursor one character to the left
\f	form feed	go to top of new page
\n	newline	go to the next line
\r	return	go to beginning of current line
\a	audible alert	'beep'
\t	tab	advance to next tab stop
\'	apostrophe	character '
\"	double quote	character "
\\	backslash	character \
\nnn		character number nnn (octal)

Conditional Branching

* So far, there has been no opportunity for decision making in the code written.
* It would be useful to be able to choose to do one thing, or another thing, on the basis of the result of some test.
* This concept is called a "conditional branch", from the metaphor of a tree which splits into multiple branches as it grows.

The if statement.

* The most basic conditional structure in C is the if.... else if... else construct.
* This block is executed only if the test evaluates to true.

```
if (test1) {
  some statements;
}
```

Example

```
/*********************************************
* if.c - Program to illustrate "if"          *
*                                            *
* David Britton, Jan 2014                    *
*                                            *
*********************************************/

#include <stdio.h>

        int main()
        {

        int x = 4;
        if (x == 3)  {
          printf("x = %d \n", x );
        }
        else if (x > 3) {
          puts("x is greater than 3");
        }
        else
          puts("x is less than 3");

      return 0;
      }
```

The switch,case construct.

* We can use long chains of `else ifs` to test a variable for many different values, and do different things dependant on them.
* However, this is ugly and inefficient, and also doesn't let us reuse common code between some of the blocks.
* The `switch,case` construct is a more natural fit for this kind of problem.

switch, case

* `switch` is given an expression which will evaluate to an integer value.
* Each case has an integer value which will be compared to the switch expression, followed by a `:`.
* Code in the `switch` block is executed starting *immediately after* the case label which matches value.
* If no case matches the value, execution starts after the (optional) `default:` label.

```
switch (expression) {
  case value1:
    some statements;
    break;
  case value2:
    more statements;
  default:
    further statements;
}
```

switch, case

* If the expression were equal to *value1*, these statements would be executed.
* The `break` statement stops execution when encountered, and moves to the end of the block (outside the switch itself).
* If the expression instead were equal to *value2*, these statements would be executed.
* The lack of a `break` allows all the statements after the case label to be executed, including those after subsequent labels.

```
switch (expression) {
  case value1:
    some statements;
    break;
  case value2:
    more statements;
  default:
    further statements;
}
```

Example

```
/*********************************************
* switch.c – Program to illustrate "switch"    *
*                                              *
* David Britton, Jan 2014                      *
*                                              *
*********************************************/

#include <stdio.h>

        int main()
        {

        int x = 4;
        switch (x) {
            case 1:
                puts("This is case-1");
                break;
            case 2:
                puts("this is case-2 with drop-through");
            case 3:
                puts("this is case-3 or drop-through from case-2");
                break;
            default:
                puts("this is the default case");
        }
      return 0;
      }
```

Loops

* Now we can do different things based on a test, but we can only do each thing *once*.
* There are lots of processes that essentially reduce to either:
  Do (operation) on (every member) of (some list of items).
  Do (operation) until (some condition is met).
* Both are examples of loop structures, which repeat a core operation many times.
* C provides several loop constructs which are designed to handle the first or second cases particularly naturally.

The do... while construct.

* The `do ... while` construct repeats until a condition is met.
* Every iteration:
  1. The statements in this block are executed.
  2. The test is performed.
  3. If `true`, we jump back up to the start of the block.
  4. If `false`, we continue on to the next statements after the block.

```
do {
  some statements;
} while (test) ;
```

While Construct

```
while (test) {
    ...
    some statements;
    ...
}
```

The for loop.

* Whilst a do-loop repeats until a condition is met, a for-loop is intended for situations where you are explicitly counting loops (or doing the same thing for multiple items).
* For example, multiplying all the elements of a list of values by a number, or printing something out a certain number of times.

e.g.

```
#include <stdio.h>

    int main() {

        for (int i = 0; i < 4; i++) {
            printf("The counter i = %d \n",i);
        }
    return 0;
    }
```

```
for(init ; test ; iter  )
{
  some statements;
}
```

The for loop.

* The for statement contains three semicolon separated expressions.
* The first (and only the first) time we encounter the statement, the init expression is executed. Usually this is used to set a starting value for a "counter" variable.
* In C99 and later, we can declare a variable and initialise it in the init expression, limiting the scope of our counter to the loop itself. (i.e. `int i = 0`)

e.g.

```
#include <stdio.h>

    int main() {

        for (int i = 0; i < 4; i++) {
            printf("The counter i = %d \n",i);
        }
    return 0;
    }
```

```
for(init ; test ; iter  )
{
  some statements;
}
```

`init` expression is typically something like: `i=0;` or `int i = 0`

The for loop.

* Every time (including just after the init), the test is performed.
  - If `true`, we execute this block.
  - If `false`, we skip to the end of block and continue on.

e.g.

```
#include <stdio.h>

    int main() {

        for (int i = 0; i < 4; i++) {
            printf("The counter i = %d \n",i);
        }
    return 0;
    }
```

```
for(init ; test ; iter  )
{
  some statements;
}
```

`test` expression is typically something like: `i < 4`

The for loop.

* If we did execute the block, we then execute the iter expression.
* Usually this is used to update a counter variable (i.e `i++`), or get a new bit of data to work on.
* We then jump up to the top of the block again, and do the test...

e.g.

```
#include <stdio.h>

    int main() {

        for (int i = 0; i < 4; i++) {
            printf("The counter i = %d \n",i);
        }
    return 0;
    }
```

```
for(init ; test ; iter  )
{
  some statements;
}
```

Example

```c
#include <stdio.h>

    int main() {

        for (int i = 0; i < 4; i++) {
            printf("The counter i = %d \n",i);
        }
    return 0;
    }
```

Breaking out of loops - `continue` and `break`

* While the loop structures above are usually sufficient as they are, sometimes we might need to modify their flow a little bit for special cases.
* The `continue` and `break` statements let alter the flow of execution inside the loop block.
* We've already met `break` when we covered `switch`.
* It works similarly in a loop block, jumping out of the loop block immediately, and continuing with whatever statements happen after it.

Break means "break out of the loop"

* The `continue` statement is like a `break`, but just for that iteration of the loop.
* We jump to the end of the loop block, but still perform the *test* (and, in a `for` loop, the *iter* expression) to see if we should keep on looping.

Continue means "quit this iteration of loop but continue immediately with next iteration"