Week 4 Lecture 2: Best Practice & Functions

Review and Preview

- Best Practice
  - Comments
  - Variable Names
- The Midterm!

- Best Practice
  - Program formatting
  - Functions

K&R: Chapters 1-3
Sections 1.1 to 1.6
Sections 2.1 to 2.6
Sections 3.1 to 3.6

UNIX Command of the Day: find

- `find` allows you to find files that exist in a directory hierarchy that match some pattern.

  ```
  $ find path expression
  ```

- `find` recursively descends the directory tree for the path listed, evaluating an expression in terms of each file in the tree.

- `$ find /home/debs  -name "*.c" -print`

  - Print out a list of all the files whose names end in `.c`

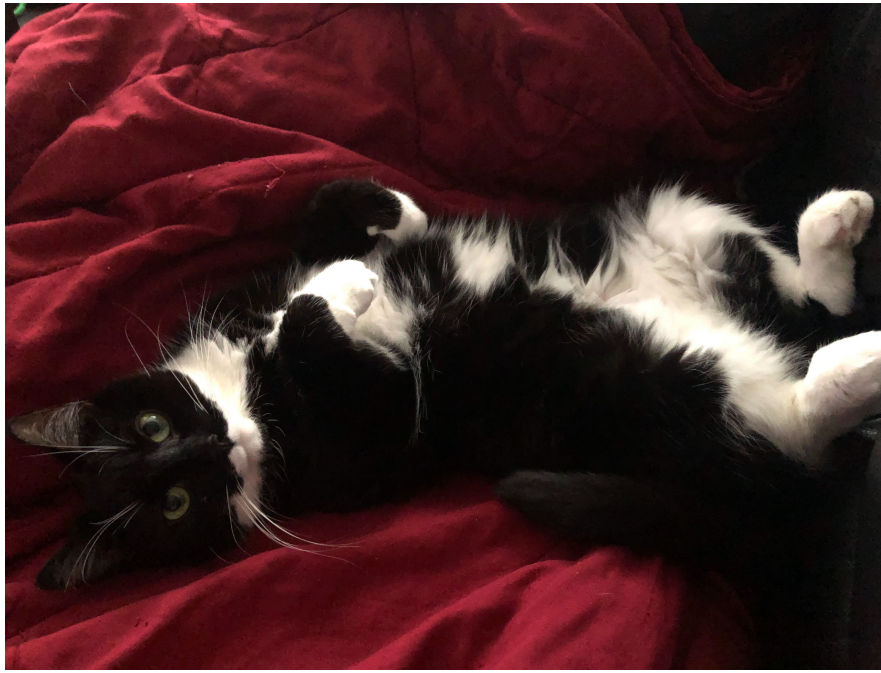
Program Formatting

- Why?
- How?

Pretty Code - Does it Matter?

- Once again this is best practice - the computer does not care how you format the code.
- Readability and understandability is enhanced by formatting your code in a consistent and meaningful way.
- Things to consider:

1. Where to put the curly braces {}
2. How to indent the code

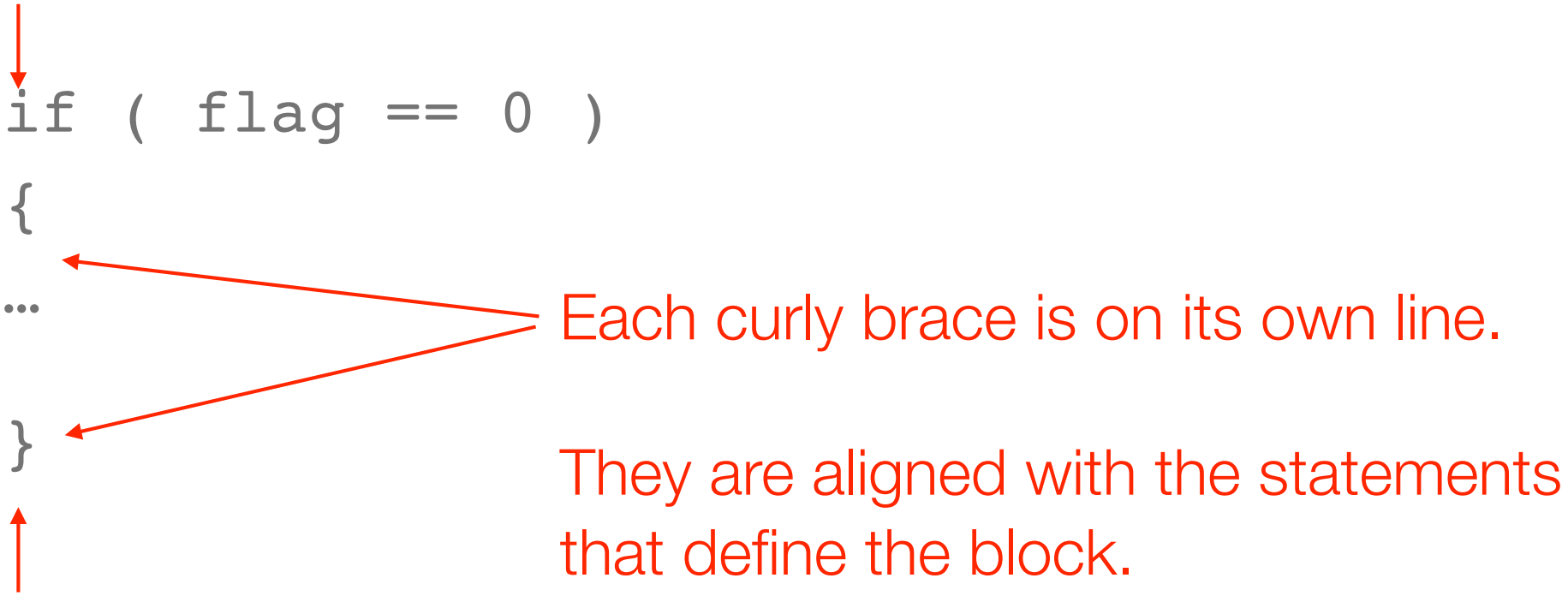


Curly Braces

- The curly braces surround blocks of code:
  - Functions including the main function
  - If else blocks
  - Loops - for and while
- There are two styles:
  - New line
  - End of line

Curly Braces - New Line

- This is the “original” style and is often called the K&R style.



Curly Braces - End of Line

- This is the “modern” style and is generally preferred now.

```
if ( flag == 0 ){
…

}
```

First curly brace is on the same line as the statement that starts the block.

The end brace is aligned with the statements that define the block.

Indentation

- Indentation is used to illustrate which statements are to be “grouped” together.
- For example, they can show if one loop is “inside” another one.
- They make it easier for your eye to see the flow and organization of the program.
- Your only decision is “how many spaces should I use for an indent”? (And should I use tabs?)

```
/* Check to see if the start year is a leap year */
if (y1 % 4 == 0 || (y1 % 100 == 0 && y1 % 400 == 0)) {
   leap1 = 1;
   monthLength[1] = 29;
}

/* Check if start date is a proper date */
if (m1 < 1 || m1 > 12 || d1 < 1 || d1 > monthLength[m1-1]) {
   printf ("Start date is not a proper date\n");
   return ( 1 );
}

/* Check if end date is a proper date */
if (m2 < 1 || m2 > 12 || d2 < 1 || d2 > monthLength[m2-1]) {
   printf ("End date is not a proper date\n");
   return ( 1 );
}
```

```
if ( flag == 0) {
    for ( i=0; i<10; i++ ) {
        for ( j=0; j<2; j++ ) {
            printf ( "%d %d\n", i, j );
        }
    }
}
```

CAT LOOP

The indentation tells you visually that there are three major actions happening:

- a branch decision (`if`)
- an outer `for` loop
- an inner `for` loop

Functions*

How do they help?

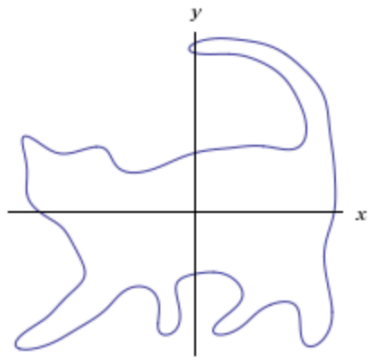
How do they work?

*Also known as subroutines

Result:

$$x(t) = -\frac{721\sin(t)}{4} + \frac{196}{3}\sin(2t) - \frac{86}{3}\sin(3t) - \frac{131}{2}\sin(4t) + \frac{477}{14}\sin(5t) + 27\sin(6t) - \frac{29}{2}\sin(7t) + \frac{68}{5}\sin(8t) + \frac{1}{10}\sin(9t) + \frac{23}{4}\sin(10t) - \frac{19}{2}\sin(12t) - \frac{85}{21}\sin(13t) + \frac{2}{3}\sin(14t) + \frac{27}{5}\sin(15t) + \frac{7}{4}\sin(16t) + \frac{17}{9}\sin(17t) - 4\sin(18t) - \frac{1}{2}\sin(19t) + \frac{1}{6}\sin(20t) + \frac{6}{7}\sin(21t) - \frac{1}{8}\sin(22t) + \frac{1}{3}\sin(23t) + \frac{3}{2}\sin(24t) + \frac{13}{5}\sin(25t) + \sin(26t) - 2\sin(27t) + \frac{3}{5}\sin(28t) - \frac{1}{5}\sin(29t) + \frac{1}{5}\sin(30t) + \frac{2337\cos(t)}{8} - \frac{43}{5}\cos(2t) + \frac{322}{5}\cos(3t) - \frac{117}{5}\cos(4t) - \frac{26}{5}\cos(5t) - \frac{23}{3}\cos(6t) + \frac{143}{4}\cos(7t) - \frac{11}{\cdot}\cos(8t) - \frac{31}{\cdot}\cos(9t) - \frac{13}{\cdot}\cos(10t) - \frac{9}{\cdot}\cos(11t) + \frac{41}{\cdot}\cos(12t) + 8\cos(13t) - \frac{31}{8}\cos(7t) + \frac{9}{10}\cos(8t) - \frac{119}{4}\cos(9t) - \frac{17}{2}\cos(10t) + \frac{22}{3}\cos(11t) + \frac{15}{4}\cos(12t) - \frac{5}{2}\cos(13t) + \frac{19}{6}\cos(14t) + \frac{7}{4}\cos(15t) + \frac{31}{4}\cos(16t) - \cos(17t) + \frac{11}{10}\cos(18t) - \frac{2}{3}\cos(19t) + \frac{13}{3}\cos(20t) - \frac{5}{4}\cos(21t) + \frac{2}{3}\cos(22t) + \frac{1}{4}\cos(23t) + \frac{5}{6}\cos(24t) + \frac{3}{4}\cos(26t) - \frac{1}{2}\cos(27t) - \frac{1}{10}\cos(28t) - \frac{1}{3}\cos(29t) - \frac{1}{19}\cos(30t)$$

Plot:



(plotted for t from 0 to 2π)

How can you make your programs more “tidy”?

- Why “tidy”?
  - Shorten the length of the program.
    - Easier to read and understand the program.
    - Humans have limited memory.
  - Organize like tasks and separate from the rest of the code.
    - Ability to reuse these small snippets of code.

Functions - What are Functions?

- A function is a group of statements that together perform a task.
- That task is usually small and specific.
- A function consists of two parts:
- A function **declaration** consists of the function's **name**, **return type**, and **parameters** (number and type).
- A function **definition** provides the actual body (variables, statements) of the function.

```
returnType functionName (parameter list) {
   body of the function
}
```

```
int isLeapYear ( int year ) {
…
}
```

Definition

```
int isLeapYear ( int );
```

Declaration

Parts of a Function

- **Name** – This is the name of the function. This is used by other programs to “call” the function.
- **Parameters** – When a function is called, values are passed into it via the parameters.
  - type, order, and number of parameters
  - a function may contain no parameters

Parts of a Function

- **Return Type** – if the function returns a value then its data type must be defined.
- Some functions do **not** return a value.
  - If this is the case, then the return_type is the keyword `void`.

Function Declarations

```
int max ( int num1, int num2 );
```

- In a function definition only the types of the parameter are required, so

  ```
  int max(int, int);
  ```

  is also a valid declaration.

- But why do we need function declarations?

When the function **max** is called
what are the values of **n1** and **n2**?

max function

```
int max ( int n1, int n2 ) {
    n1 = n1 + 1;
    n2 = n2 + 1;
    if ( n1 > n2 ) {
        return ( n1 );
    } else {
        return ( n2 );
    }
}
……
num1 = 2;
num2 = 4;
maxNum = max(num1, num2);
```

Calling the **max** function

What are the values of **num1** and **num2**
after the **max** function has returned?

Where do you “put” functions?

- You can put functions in one of two places:

  1. Before the main() in the source file containing the mainline.

  2. In a different source file.

  - A function declaration is required when you **define** a function in one source file and you **call** that function in another file.

    - In this case, the called function’s declaration must be at the top of the source file calling the function.

testMax1.c

```
#include <stdio.h>

int max ( int n1, int n2 ) {
    if ( n1 > n2 ) {
        return ( n1 );
    } else {
        return ( n2 );
    }
}

int main ( int argc, char *argv[] ) {
    int num1, num2;
    num1 = atoi (argv[1]);
    num2 = atoi (argv[2]);
    printf ( "The max of %d and %d is %d\n",
              num1, num2, max(num1, num2) );
    return(0);
}
```

$ gcc -ansi testMax1.c -o testMax1

```
int max ( int n1, int n2 ) {
    if ( n1 > n2 ) {
        return ( n1 );
    } else {
        return ( n2 );
    }
}
```

max.c

$ gcc -ansi -c max.c

$ gcc -ansi testMax.c max.o -o testMax

```
#include <stdio.h>
int max ( int, int );

int main ( int argc, char *argv[] ) {
    int num1, num2;
    num1 = atoi (argv[1]);
    num2 = atoi (argv[2]);
    printf ( "The max of %d and %d is %d\n",
                num1, num2, max(num1, num2) );
    return(0);
}
```

testMax.c

```
/* Check to see if the start year is a leap year */
if (y1%4 == 0 || (y1%100 == 0 && y1%400 == 0)) {
   leap1 = 1;
}
```

Function : leapYear
Information needed : year - integer
Return : leap yes (1) or no (0) - integer

```
int leapYear ( int year ) {
  if (year%4 == 0 || (year%100 == 0 && year%400 == 0)){
    return ( 1 );
  } else {
    return ( 0 );
  }
}
```

```
if ( leapYear ( y1 ) == 1 ) { …
```

```
/* Check if start date is a proper date */
if (m1 < 1 || m1 > 12 || d1 < 1 ||
                         d1 > monthLength[m1-1] ) {
   printf ("Start date is not a proper date\n");
   return ( 1 );
}

/* Check if end date is a proper date */
if ( m2 < 1 || m2 > 12 || d2 < 1 ||
                        d2 > monthLength[m2-1] ) {
   printf ("End date is not a proper date\n");
   return ( 1 );
}
```

Both of these sections of code are doing the same task and thus are candidates to be replaced by a function.

Valid Date Checker

- Information needed:
  - Day, Month, `monthLength[]`
- What should be passed into the function?
- What about `monthLength[]`?
  - Advantage: the main program does not have to store `monthLength[]` since it is not used elsewhere in the code.
  - What about leap year?

Valid Date Checker

- What about leap year?
  - Could send information about leap year after using the leap year function
  - Could send the year and have this function also use the leap year function

```
int validDateChecker ( int day, int month, int year ) {

  /* The number of days in each month */
  int monthLength[12] = {31,28,31,30,31,30,31,31,30,31,30,
                         31};

  if ( leapYear ( year ) == 1 ) {
     monthLength[1] = 29;
  }

  if ( month < 1 || month > 12 || day < 1 ||
                  day > monthLength[month-1] ) {
     return ( -1 );
  } else {
     return ( 0 );
  }

}
```

```
int dayOfYear ( int day, int month, int year ) {

  /* Day of Year - non-Leap year */
  int nonDoY[12] = {1,32,60,91,121,152,182,213,244,274,
                    305,335};

  /* Day of Year - Leap year */
  int leapDoY[12] = {1,32,61,92,122,153,183,214,245,
                     275,306,336};

 if ( leapYear ( year ) == 0 ) {
     return ( nonDoY[month-1] + day - 1 );
  } else {
     return ( leapDoY[month-1] + day - 1 );
  }

}
```

Calling the Function

- To use a function, you have to "**invoke**" it or "**call**" it.
- You do this by using its name and giving it the required parameters and handling the return value (if any).

```
retValue = functionName (param1, param2);
```

- But how does C handle "**passing**" the parameters from one function (*e.g.* `main`) to another one (*e.g.* `functionName`).

Passing Parameters

- There are two ways to pass parameters to a function:
- Pass by **Value**
  - The value of the parameter in the calling routine is copied into the value of the parameter in the function.
  - Changes made to the parameter inside the function have no effect on the value of the variable in the calling routine.

Passing Parameters

- Pass by Reference
- This method tells the function where the actual parameters are in memory.
- Inside the function, the actual calling routine's variables are used. The variables in the function are the **actual** variables in the calling routine. It does this by means of **pointers**.
- Any changes made to the parameters in the function affect those variables in the calling routine.

When the function **max** is called
what are the values of **n1** and **n2**?

max function

```
int max ( int *n1, int *n2 ) {
    *n1 = *n1 + 1;
    *n2 = *n2 + 1;
    if ( *n1 > *n2 ) {
        return ( *n1 );
    } else {
        return ( *n2 );
    }
}
……
num1 = 2;
num2 = 4;
maxNum = max(&num1, &num2);
```

Calling the **max** function

What are the values of **num1** and **num2**
after the **max** function has returned?