# EECS2031
# Software Tools

Department of Electrical Engineering
& Computer Science
Lassonde School of Engineering
York University

- Instructor: Gulzar Khuwaja, PhD, PEng
- Email: khuwaja@cse.yorku.ca
  Tel: (416) 736-2100 x 77874
- Course Website:
  https://wiki.eecs.yorku.ca/course_archive/2015-16/S/2031/
- Schedule:
  Lectures: T 6:00 – 8:00 pm Room LSB 105
  Labs: T 4:00 – 6:00 pm Room LAS 1006
  Office Hrs: T 8:00 – 9:00 pm Room LAS 2018

# Grading Details

- Lab Test 1 20%
- Lab Test 2 20%
- Midterm 20%
- Final 40%

# Introduction

Course Content

- C
  - Learn how to write and test C programs
- UNIX (LINUX)
  - Using Unix tools to automate making and testing
  - Unix shell programming

# About the course

- By the end of course, the students are expected to be able to:
  - Use the basic functionality of the Unix shell, such as standard commands and utilities, input/output redirection, and pipes
  - Develop and test shell scripts of significant size
  - Develop and test programs written in C programming language
  - Describe the memory management model of C programming language

# Text

- C Programming: A Modern Approach 2nd edition K.N. King http://knking.com/books/c2/
- The C Programming Language, Kernighan and Ritchie (K&R)
- Practical Programming in the UNIX Environment edited by W. Sturzlinger
- Class notes
- Man pages

# WHY C and UNIX

- Wide use, powerful, and fast
- Both started at AT&T Bell Labs
- UNIX was written in assembly, later changed to C

# WHY C and UNIX

- The first part of the course is C
- The second part shell script (sh)
- We will start with a quick introduction to Unix to be able to start the labs

# Introduction to Unix

- Please check the tutorial at http://www.cs.sfu.ca/~ggbaker/reference/unix/
- The first 6 tutorials

Chapter 1

# Introducing C

# C – A History

- Ken Thompson wrote original version of UNIX
- Thompson designed B language based on BCPL
- Dennis Ritchie began programming in B
- Bell Labs acquired a PDP-11 for UNIX in 1970
- Ritchie developed an extended version called NB and then C
- UNIX was rewtitten in C in 1973

# C – A History

- In 1978 Brian Kernighan and Dennis Ritchie Published their “elite” book and became defacto standard for C known as K&R C
- ANSI completed a standard for C approved in 1989 as ANSI X3.159-1989 known as C89 or C90 (ANSI-C)
- C99 became standard in ISO/IEC 9899:1999

# Languages based on C

- C++ basically object oriented C
- Java based on C syntax, much more restrictive + garbage collection
- C# derived from C++ and Java
- Perl started as scripting language, overtime adopted many features of C

# C: Strengths

- Low level: access to machine level (bytes, addresses, etc) for systems programming. It provides operations that correspond to a computer's built-in instructions
- Small: limited set of features. Relies heavily on a library of standard functions
- Permissive: assumes you know what you are doing so it allows you to a wider degree of freedom

# C: Strengths

- Efficient: run quickly and in limited amounts of memory
- Portable: early association with Unix and ANSI/ISO and portability supporting features
- Powerful: large collection of data types and operators
- Flexible: from systems programming to embedded systems. C imposes very few restrictions
- Standard library contains hundreds of functions for input/output, string handling, storage allocation, etc
- Integration with UNIX

# C: Weaknesses

- Error prone: programming mistakes can not be detected by compiler for its flexibility like assembly
- Difficult to understand: a number of features are not found in other languages and often misunderstood
- Difficult to modify: large programs can be hard to change if not written for maintenance. C lacks features like classes and packages for program division

# Obfuscated C

- 1990's best small program of the annual international obfuscated C code contest
- Program written by Doron O. and Baruch N.
- Prints all solutions to the Eight Queens problem

```
int v,i,j,k,l,s,a[99];
main(){
for(scanf("%d",&s);*a-s;v=a[j*=v]-a[i],k
=i<s,j+=(v=j<s&&(!k&&!!printf(2+"\n\n%c"-(!l<<!j),"
#Q"[l^v?(l^j)&1:2])&&++l || a[i]<s&&v&&v-i+j&&v+i-
j))&&!( l%=s),v|| (i==j?a[i+=k]=0:++a[i])>=s*k&&++a[-
-i])
                ;
}
```

# Tips

- Use tools to make programs more reliable
- Use existing code library to reduce errors and save programming effort
- Adopt a sensible set of coding conventions. It's possible to write a code that is all but unreadable
- Avoid tricks and overly complex code. Shortest solution is often the hardest to comprehend

Chapter 2

# C Fundamentals

# Program: Printing a Pun

```
#include <stdio.h>

int main(void)
{
  printf("To C, or not to C: that is the question.\n");
  return 0;
}
```

# Compiling and Linking

- Before a program can be executed, three steps are usually necessary:
  - ***Preprocessing.*** The ***preprocessor*** obeys commands that begin with # (known as ***directives***)
  - ***Compiling.*** A ***compiler*** then translates the program into machine instructions (***object code***).
  - ***Linking.*** A ***linker*** combines the object code produced by the compiler with any additional code needed to yield a complete executable program.
- The preprocessor is usually integrated with the compiler.

# Compiling and Linking Using `cc`

- To compile and link the `pun.c` program under UNIX, enter the following command in a terminal or command-line window:

  ```
  % cc pun.c
  ```

  The `%` character is the UNIX prompt.
- Linking is automatic when using `cc`; no separate link command is necessary.

# Compiling and Linking Using `cc`

- After compiling and linking the program, `cc` leaves the executable program in a file named `a.out` by default.
- The `-o` option lets us choose the name of the file containing the executable program.
- The following command causes the executable version of `pun.c` to be named `pun`:

```
% cc -o pun pun.c
```

# The GCC Compiler

- GCC is one of the most popular C compilers.
- GCC is supplied with Linux but is available for many other platforms as well.
- Using this compiler is similar to using `cc`:

```
% gcc -o pun pun.c
```

# The General Form of a Simple Program

- Simple C programs have the form

  *directives*

```
int main(void)
{
   statements
}
```

# The General Form of a Simple Program

- C uses `{` and `}` in much the same way that some other languages use words like `begin` and `end`.
- Even the simplest C programs rely on three key language features:
  - Directives
  - Functions
  - Statements

# Directives

- Before a C program is compiled, it is first edited by a preprocessor.
- Commands intended for the preprocessor are called directives.
- Example:

```
#include <stdio.h>
```

- `<stdio.h>` is a ***header*** containing information about C's standard I/O library.

# Directives

- Directives always begin with a # character.
- By default, directives are one line long; there's no semicolon or other special marker at the end.

# Functions

- A ***function*** is a series of statements that have been grouped together and given a name.
- ***Library functions*** are provided as part of the C implementation.
- A function that computes a value uses a `return` statement to specify what value it "returns":

```
return x + 1;
```

# The `main` Function

- The `main` function is mandatory.
- `main` is special: it gets called automatically when the program is executed.
- `main` returns a status code; the value 0 indicates normal program termination.
- If there's no `return` statement at the end of the `main` function, compilers may produce a warning message.

# Statements

- A ***statement*** is a command to be executed when the program runs.
- `pun.c` uses only two kinds of statements. One is the `return` statement; the other is the ***function call.***
- Asking a function to perform its assigned task is known as ***calling*** the function.
- `pun.c` calls `printf` to display a string:

```
printf("To C, or not to C: that is the question.\n");
```

# Comments

- A ***comment*** begins with `/*` and end with `*/`.

```
/* This is a comment */
```

- Comments may appear almost anywhere in a program, either on separate lines or on the same lines as other program text.
- Comments may extend over more than one line.

```
/* Name: pun.c
   Purpose: Prints a bad pun.
   Author: K. N. King */
```

# Comments

- *Warning:* Forgetting to terminate a comment may cause the compiler to ignore part of your program:

```
printf("My ");     /* forgot to close this comment...
printf("cat ");
printf("has ");    /* so it ends here */
printf("fleas");
```

# Comments in C99

- In C99, comments can also be written in the following way:

```
// This is a comment
```

- This style of comment ends automatically at the end of a line.
- Advantages of `//` comments:
  - Safer: there's no chance that an unterminated comment will accidentally consume part of a program.

# Variables and Assignment

- Most programs need a way to store data temporarily during program execution.
- These storage locations are called ***variables.***

# Types

- Every variable must have a ***type.***
- C has a wide variety of types, including `int` and `float`.
- A variable of type `int` (short for *integer*) can store a whole number such as 0, 1, 392, or –2553.

# Types

- A variable of type `float` (short for *floating-point*) can store much larger numbers than an `int` variable.
- Also, a `float` variable can store numbers with digits after the decimal point, like 379.125.
- Drawbacks of `float` variables:
  - Slower arithmetic
  - Approximate nature of `float` values

# Declarations

- Variables must be ***declared*** before they are used.
- Variables can be declared one at a time:

```
int height;
float profit;
```

- Alternatively, several can be declared at the same time:

```
int height, length, width, volume;
float profit, loss;
```

# Declarations

- When `main` contains declarations, these must precede statements:

```
int main(void)
{
    declarations
    statements
}
```

- In C99, declarations don't have to come before statements.

# Printing the Value of a Variable

- `%d` works only for `int` variables; to print a `float` variable, use `%f` instead.
- By default, `%f` displays a number with six digits after the decimal point.
- To force `%f` to display *p* digits after the decimal point, put `.`*p* between `%` and `f`.
- To print the line

  ```
  Profit: $2150.48
  ```

  use the following call of `printf`:

  ```
  printf("Profit: $%.2f\n", profit);
  ```

# Initialization

- Some variables are automatically set to zero when a program begins to execute, but most are not.
- A variable that doesn't have a default value and hasn't yet been assigned a value by the program is said to be ***uninitialized.***
- Attempting to access the value of an uninitialized variable may yield an unpredictable result.
- With some compilers, worse behavior—even a program crash—may occur.

# Reading Input

- `scanf` is the C library's counterpart to `printf`.
- `scanf` requires a ***format string*** to specify the appearance of the input data.
- Example of using `scanf` to read an `int` value:

```
scanf("%d", &i);
/* reads an integer; stores into i */
```

- The `&` symbol is usually (but not always) required when using `scanf`.

# Reading Input

- Reading a `float` value requires a slightly different call of `scanf`:

```
scanf("%f", &x);
```

- `"%f"` tells `scanf` to look for an input value in `float` format (the number may contain a decimal point, but doesn't have to).

# Program: Converting from Fahrenheit to Celsius

- The `celsius.c` program prompts the user to enter a Fahrenheit temperature; it then prints the equivalent Celsius temperature.
- Sample program output:

```
Enter Fahrenheit temperature: 212
Celsius equivalent: 100.0
```

- The program will allow temperatures that aren't integers.

# celsius.c

```
/* Converts a Fahrenheit temperature to Celsius */

#include <stdio.h>

#define FREEZING_PT 32.0f
#define SCALE_FACTOR (5.0f / 9.0f)

int main(void)
{
  float fahrenheit, celsius;

  printf("Enter Fahrenheit temperature: ");
  scanf("%f", &fahrenheit);

  celsius = (fahrenheit - FREEZING_PT) * SCALE_FACTOR;

  printf("Celsius equivalent: %.1f\n", celsius);

  return 0;
}
```

# Identifiers

- Names for variables, functions, macros, and other entities are called ***identifiers.***
- An identifier may contain letters, digits, and underscores, but must begin with a letter or underscore:

  ```
  times10  get_next_char  _done
  ```

  It's usually best to avoid identifiers that begin with an underscore.
- Examples of illegal identifiers:

  ```
  10times  get-next-char
  ```

# Identifiers

- C is ***case-sensitive:*** it distinguishes between upper-case and lower-case letters in identifiers.
- For example, the following identifiers are all different:

```
job   joB   jOb   jOB   Job   JoB   JOb   JOB
```

Chapter 3

# Formatted Input/Output

# The `printf` Function

- The `printf` function must be supplied with a ***format string,*** followed by any values that are to be inserted into the string during printing:

  `printf(`*string*`, `*expr*`1, `*expr*`2, …);`
- The format string may contain both ordinary characters and ***conversion specifications,*** which begin with the `%` character.
- A conversion specification is a placeholder representing a value to be filled in during printing.
  - `%d` is used for `int` values
  - `%f` is used for `float` values

# The `printf` Function

- Ordinary characters in a format string are printed as they appear in the string; conversion specifications are replaced.
- Example:

```
int i, j;
float x, y;

i = 10;
j = 20;
x = 43.2892f;
y = 5527.0f;

printf("i = %d, j = %d, x = %f, y = %f\n", i, j, x, y);
```

- Output:

```
i = 10, j = 20, x = 43.289200, y = 5527.000000
```

# The `printf` Function

- Compilers aren't required to check that the number of conversion specifications in a format string matches the number of output items.
- Too many conversion specifications:

```
printf("%d %d\n", i);    /*** WRONG ***/
```

- Too few conversion specifications:

```
printf("%d\n", i, j);    /*** WRONG ***/
```

# The `printf` Function

- Compilers aren't required to check that a conversion specification is appropriate.
- If the programmer uses an incorrect specification, the program will produce meaningless output:

```
printf("%f %d\n", i, x);  /*** WRONG ***/
```

# Conversion Specifications

- The ***minimum field width*** specifies the minimum number of characters to print.
- If the value to be printed requires fewer characters, it is right-justified within the field.
  - `%4d` displays the number 123 as `•123`. (• represents the space character.)
- If the value to be printed requires more characters, the field width automatically expands to the necessary size.
- Putting a minus sign causes left justification.
  - The specification `%-4d` would display 123 as `123•`.

# Program: Using `printf` to Format Numbers

- The `tprintf.c` program uses `printf` to display integers and floating-point numbers in various formats.

# tprintf.c

```
/* Prints int and float values in various formats */

#include <stdio.h>

int main(void)
{
  int i;
  float x;

  i = 40;
  x = 839.21f;

  printf("|%d|%5d|%-5d|%5.3d|\n", i, i, i, i);
  printf("|%10.3f|%10.3e|%-10g|\n", x, x, x);

  return 0;
}
```

- Output:

```
|40|   40|40   |  040|
|   839.210| 8.392e+02|839.21     |
```

# Escape Sequences

- The `\n` code that used in format strings is called an ***escape sequence.***
- Escape sequences enable strings to contain nonprinting (control) characters and characters that have a special meaning (such as ").
- A partial list of escape sequences:

| | |
|---|---|
| Alert (bell) | `\a` |
| Backspace | `\b` |
| New line | `\n` |
| Horizontal tab | `\t` |

# Escape Sequences

- A string may contain any number of escape sequences:

```
printf("Item\tUnit\tPurchase\n\tPrice\tDate\n");
```

- Executing this statement prints a two-line heading:

```
Item    Unit    Purchase
        Price   Date
```

# Escape Sequences

- Another common escape sequence is `\"`, which represents the `"` character:

```
printf("\"Hello!\"");
  /* prints "Hello!" */
```

- To print a single `\` character, put two `\` characters in the string:

```
printf("\\");
  /* prints one \ character */
```

# The `scanf` Function

- In many cases, a `scanf` format string will contain only conversion specifications:

```
int i, j;
float x, y;

scanf("%d%d%f%f", &i, &j, &x, &y);
```

- Sample input:

```
1 -20 .3 -4.0e3
```

`scanf` will assign 1, –20, 0.3, and –4000.0 to `i`, `j`, `x`, and `y`, respectively.