CSE2031 Software Tools

SU 2012 - 2013

The Course
CSE2031 Software Tools

- Lecture: LSB 105. Tuesdays 18:00 – 20:00
- Labs: LAS (CSEB) 1006
  - Lab 01 Tuesdays 16:00 – 18: 00
  - Lab 02 Wednesdays 17:00 – 19:00
- Course website: **www.cse.yorku.ca/course/2031**
  - Visit frequently for announcements, due dates, solutions etc …

The instructor

Hui Wang
- Office: LAS (CSEB) 2010
- Email: **huiwang@cse.yorku.ca**
- Home page: www.cse.yorku.ca/~huiwang
- Office hours
  - Tuesdays 17:30 pm – 18:00 pm LAS (CSEB) 2010 (take elevator)
  - Tuesdays after class 20:00-20:30 Classroom or LAS (CSEB) 2010
  - Wednesdays after lab 19:00-20:00 Lab or LAS (CSEB) 2010
  - By appointment in special cases

- CSE2031 student
- CSE2031 instructor

Course content

- Introduces software tools that are useful for building applications and are useful in the software development process.

- You will be exposed to the layers between a programming language and the operating system and the CPU.

The course covers the following topics:
- ANSI-C
  - Learning how to write, test and debug C programs
  - (C Basics, stdio, pointers, memory management, ANSI-C libraries)

- Unix (Linux) operating system
  - Using Unix tools to automate compilation, execution and testing
  - Shell programming under Unix -- Bourne (again) shell, filters and pipes)

- Testing and debugging

Course objective

- By the end of the course, you should be able to
  - Write modest-sized programs in C
  - Write programs using UNIX shell scripting language
  - Test and debug C and other programs
  - Use Unix utilities for fast solving practical problems.

Why 'software tools'

- CSE 1020 → CSE 1030
- Basic programming skills / concepts
  - read API specification (client), implement API (implementer).
- → CSE 2011
  - classical data structures for developing algorithms (not only for Java)
- Java gives you a "safe" programming environment (VM)
  - higher level

- Now good time to learn practical programming skills
  - know how lower layers work. Better understanding how software develops and works, as well as testing debugging skills
- Laying foundation for future courses, careers, researches .....

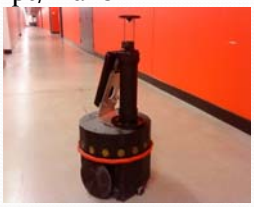
Why C and Unix

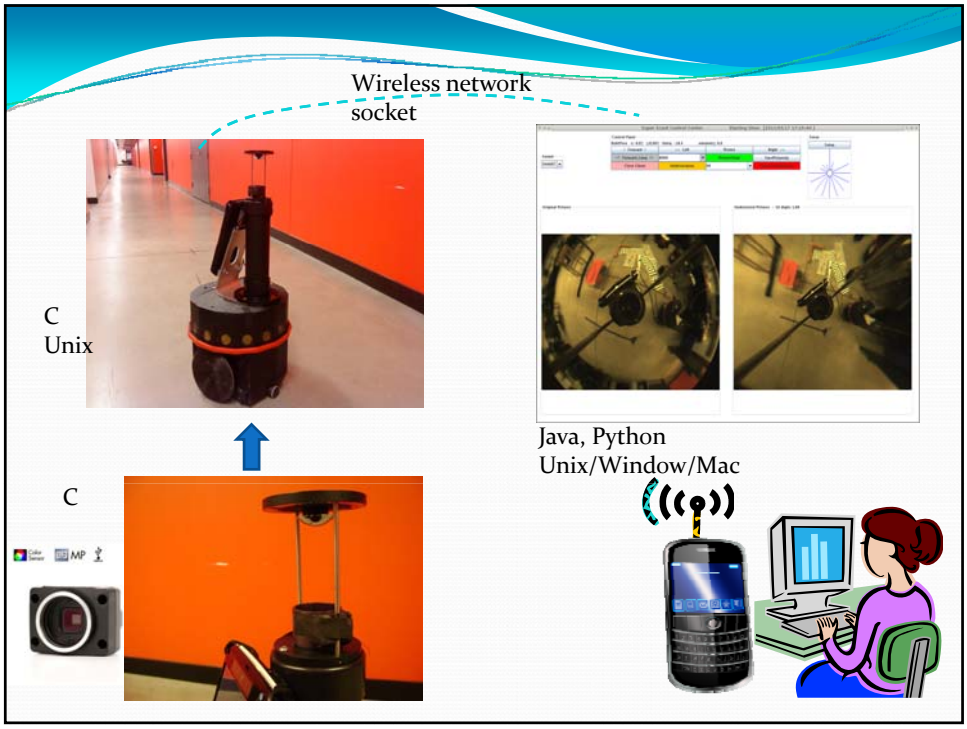
- C and Unix are closely related
  - Both started at AT&T Bell labs
  - Unix was written in assembly language, later changed to C
- Right platforms to teach skills necessary for practical program development
  - C: Expose students to underlying layers
    - C's ability to handle low-level activities (direct memory access, memory allocation, BitWise operations, etc)
    - Safety layers not present (C has poor error detection and significantly fewer safeguards than Java)
      - A good language to learn testing and debugging
  - Unix: Good environment to learn systematic testing

Application

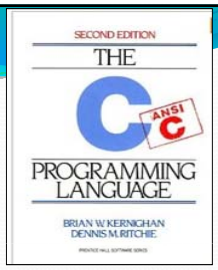
- Network, image processing .....
- Another example -- Driving robots
  - Robot side: Unix (Ubuntus), C
  - Socket programming, shell script, make file ....
  - Add a camera, still c
    - Integrate with the existing C code -- script/make







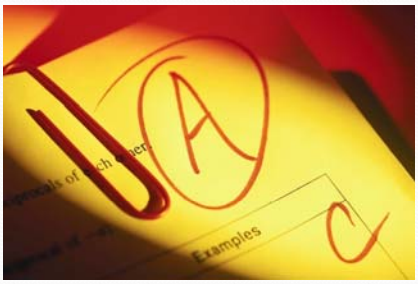
Textbooks



- **The C Programming Language (2nd Ed)**
  - *B. W. Kernighan and D. M. Ritchie (KR)*
  - *ISBN 0-13-110362-8*
  - Short and well written, covering C89/ANSI-C, this is the classic C book.
- **Practical Programming in the UNIX Environment**
  - *Edited by W. Sturzlinger*
  - *ISBN 0-536-74996-5*
  - This is a compendium of material from two other texts. Tailored for you!
- List of recommended books on course website
- Additional online material will be provided

Administrivia

- Grading Scheme (tentative)
  - 5 % – Weekly labs (9)
  - 12% – Assignments (3)
  - 35% – Midterm test
    - 25% –Written
    - 10% – Lab tests
  - 45~50% – Final exam
    - 35% –Written
    - 15% – Lab tests



  - Maybe a few in-class quizzes 3%

Administrivia

- Weekly Labs: Wednesdays, and next Tuesday
  - A set of 2-3 **small problems is posted earlier**. They are similar to & serve as the basis to the lab questions.
  - One or two are chosen in the lab
  - Some are in "lab test" mode, no web, submit locally
  - Good learning process. We are there to help.
  - Each lab is worth about 0.5 ~ 1%.

  - Use as tutorial, ask questions to TA or me!
  - Some labs are cancelled, check schedule. Will announce

Administrivia

- Assignments
  - **Medium programming problems.**
  - Students have around 2-3 weeks to complete and submit an assignment.
  - Can work in a group of up to two.

Administrivia

- Tests and Exam
  - Lab tests I, II (mid-term, final)
    - Small to medium-size programming problems
  - Written midterm and final tests
- Test and Exam Policy
  - You are allowed to miss a test/exam only under extraordinary circumstances.
  - If the reason is sickness, your doctor must fill in the Attending Physician's Statement form. Only this form, completely and properly filled, will be accepted.
  - There are NO make up tests. The weight of a missed test will be transferred to another or the final exam.

Useful suggestions

- When sending emails to the instructor or TA, please indicate "CSE 2031" in the subject line (e.g., "CSE 2031 questions")
- Use your CSE email! Check/forward your CSE email!
- Read the lecture notes and the textbook before and again after each lecture. Notes will be finalized after class – check back!
- Visit website frequently. Probably the only way I announce things to you outside lecture and labs
- Come to the class, come to the lab

- Work on the posted lab exercises before coming to the scheduled lab sessions.
- Practice, practice, and practice!

Challenging but doable course

- C bit shifting, pointers, memory allocation
- Unix shell syntax bit strange

Acknowledgement

- Many thanks to those who have helped me, including (but not limited to)

Matt Robinson
Hamzeh Roumani
Bil Tzerpos
Hui Jiang
Wolfgang Stuerzlinger
Gordon Turpin
Mokhtar Aboelaze
Przemyslaw Pawluk
U. T. Nguyen
Aijun An
Erich Leung
.....

- Any questions so far?

Overview of C KR ch1.1-8,ch7.1-4

- Powerful
  - A widely used general purpose programming language
  - with high-level constructs and ability to handle low-level activities (direct memory access, memory allocation, BitWise operations, etc).
- Small, efficient and fast
  - It produces efficient programs.
- Predecessor of modern object oriented languages
  - Many languages derived from C (e.g., C++, Java) C –> C++ --> Java
- Good to learn
  - A good (but not easy) language to learn testing and debugging (C has poor error detection and significantly fewer safeguards than Java)
- ANSI-C (C89) standard by *American National Standard Institute*

Overview of C

- C -- C++ Java

- Topics of C
  - Something same as or similar to Java
    - Variable, data type, operator (arithmetic, relational, logical etc), operation precedence, function (declare, define, call), Basic IO, expressions, flow control, strings, arrays, structures ….
  - Something totally different from Java
    - Pre-processing, header files, pointers, pointer arithmetic, memory allocation, system call ….

Overview of C

C vs Java

- Java-like (actually Java has a C-like syntax),
  - `int, double, int i =2;`
  - `void afunction (int a),`
  - Flow control -- `if else, for…, while, do while, switch`, even recursion

- Some differences
  - No classes Procedure-oriented vs. Object Oriented -- no inheri, ploy
  - No garbage collection
  - No exceptions (try - catch)
  - No type Boolean
  - No type String
  - No //, only /* */ multi line – for ANSI C
  - Declare variable at the block beginning
  - Has pointers
  - Can do memory allocation and de-allocation
  - .....

Additional resources on website

Topics of C

- Basic I/O - Chapters 1 and 7
- Variables, Types and operators - Chapter 2
- Control flow - Chapter 3
- Functions - Chapter 4
- Arrays and pointers - Chapter 5
- Dynamic memory allocation - Section 7.8.5
- Structures - Chapter 6
- Unions, enumeration - Chapter 6
- I/O, files - Chapters 7 and 8

First C program -- what it looks like

#include <iostream> in C++

```c
#include <stdio.h>
/*import standard io header files*/

/* salute the world */

int main(int argc, char** argv)
main()
{
   printf("Hello, world\n");
}
```

hello.c / first.c

```java
import java.lang.....
/*import library functions*/

class Hello
{
   // salute the world
   public static void main(String[] args)
   {
     // System.out.println("Hello World!");
     System.out.printf ("Hello World\n");
   }
}
```

Hello.java

Another C program

```c
#include  <stdio.h>

int sum (int i, int j){
    int k;
    k = i+j;
    return k;
}

/* main */
main()
{  int x=2, y=3;
   printf("%d + %d = %d\n", x,y, sum(x,y));
}
```

cal.c

```java
import java.lang.....

class Cal {
  static int sum(int i, int j){
     int k;
     k = i + j;
     return k;
   }

   // world
    public static void main(String[] args)
    {  int x=2, y=3;
       System.out.printf("%d + %d = %d\n", x,y,
sum(x,y));
    }
}
```

Cal.java

C basics

- The first program – what it looks like
- **Compile and run C program**
- Basic syntax
  - Comments
  - Variables
  - Functions
  - Basic IO functions
  - Expression
  - Statements
  - Preprocessing: # include, # define

Compiling and running a C program (general)

- C programs (source code) are in files ending with ".c" --- hello.c
- To compile a C program (in general):

```
% cc hello.c
```

- This create an executable program named **a.out** (in the current directory)

```
indigo 342 % cc hello.c
indigo 343 % a.out
Hello, world
```

Compiling and running on Linux (our lab)

- gcc -- GNU C and C++ compiler, Only c compiler for Linux.
- C99
  - variable declared as long as before use.
  - // comment
- On our Linux machines, **cc** is a symbolic (soft) link to **gcc**

```
red 289 % which cc
/usr/bin/cc
red 290 % which gcc
/usr/bin/gcc
```

```
red 291 % file /usr/bin/gcc
/usr/bin/gcc: ELF 64-bit LSB executable, x86-64,
inked (uses shared libs), for GNU/Linux 2.6.18,
red 292 % file /usr/bin/cc
/usr/bin/cc: symbolic link to `gcc'
```

```
indigo 302 % gcc -o hello hello.c
indigo 303 % hello
Hello, world
```

⟺

```
indigo 344 % cc -o hello hello.c
indigo 345 % hello
Hello, world
```

```
indigo 307 % man gcc

NAME
       gcc - GNU project C and C++ compiler

SYNOPSIS
       gcc [-c|-S|-E] [-std=standard]
           [-g] [-pg] [-Olevel]
           [-Wwarn...] [-pedantic]
           [-Idir...] [-Ldir...]
           [-Dmacro[=defn]...] [-Umacro]
           [-foption...] [-mmachine-option...]
           [-o outfile] infile...

       Only the most useful options are listed here; see below for the
       remainder.  g++ accepts mostly the same options as gcc.

DESCRIPTION
       When you invoke GCC, it normally does preprocessing, compilation,
       assembly and linking.
```

How C programs are compiled

- C executables are built in three stages

Handles
#include
#define

Converts C code into binary processor instructions -- **`hello.o`**

Puts multiple files together, linking **`hello.o`** with system libraries, and creates an executable program e.g., **`a.out`**

remove **`hello.o`**

- You also need basic knowledge of unix/linux command.
- **`cd, cd .., ls -l, rm, mkdir, cp, mv, pwd, < >`** .....
- **`submit`**

- If you need to recap, see the guided lab tour (for CSE1020) on the course website.
  - You can ignore file protection (**`chmod`**) for now

Work off campus?

- Remotely: Use Putty (and the like) to connect to our server (red) and work online
- Locally (@home, laptop): use any windows compiler that compiles C codes
  - Good choice can be GCC compiler called MinGW → cmd/dos/gcc → a.exe)
  - Visual studio, Cygwin ….
  - For MAC, something come with Xcode ?! easier?
- **Note: all submitted work must**
  - **at least compile in our system (C99)**
  - **follows ANSI-C (C89)**
- Kind suggestion for home work: done most (e.g. 90%) of the work. Then wrap up and compile in our lab for final version
  - Be careful with file transfer – dos2unix

C basics

- Compile and run C program
- Basic syntax
  - **Comments**
  - Variables
  - Functions
  - Basic IO functions
  - Expression
  - Statements
  - Preprocessing: # include, # define

Comments

- ANSI-C /* comment */
- Span multiple lines /* .....
  ..... */
- May not be nested /* /* */ */
- Good practice to comment. But don't write trivial ones

- // may not work. Depend on compiler.
  - Ok in C99 (our lab).
  - But avoid it! For portability.

C basics

- Compile and running Comments
- Basic syntax
  - Comments
  - **Variables**
  - Functions
  - Basic IO functions
  - Expression
  - Statements
  - Preprocessing: # include, # define

C variables

- Store data, whose value can change.
  - Declaration and initialization.
    - `int x;  x = 5;`
    - `int x =5; x = 9;`
- Variable names
  - combinations of letters (including underscore character _), and numbers.
  - that do not start with a number; avoid starting with _;
  - are not a keyword.
  - upper and lower case letters are distinct (`x ≠ X`).
- Examples: Identify valid and invalid variable names
  - `abc, aBc, abc5, aA3_ , my_index`
  - `5sda, _360degrees,_temp, char, struct, while`

C variables

4 basic types in C (how many in Java?)

- **char -- characters**
- **int -- integers**
- **float -- single precision floating point numbers**
- **double -- double precision floating point**

We will further study and discuss other types later.

C variables

- `int x;  x= 20;` or `int i = 20;`
- `double d = 223.3;`
- `char c = 'b'  c='\n'` (new line) `c='\t'` (tab).

- Sequence of characters forms strings "this is a string"
  - `printf("hello world\n");`
- But `String s = "hello"` does NOT work in C.
  - No string type. Array of chars. Will look at it later

One thing to get adapted from Java
(among many other things)

C basics

- Compile and running Comments
- Basic syntax
  - Comments
  - Variables
  - **Functions**
  - Basic IO functions
  - Expression
  - Statements
  - Preprocessing: # include, # define

functions

```
return_type  functionName (parameter type name, ……)
{block}
```

```
int main(){…}

int sum (int i, int j)
{
  int s = i + j;
  return s;              /*  return i+j;  */
}

void display (int i){
  printf("this is %d", i);
}
```

- Similar to Java's (static) methods ? Sort of but not really. Be careful

functions

- Must be declared or defined before we can use – different from Java
- Declaration (prototype) -- describe arguments and return type but not the code
  - `int sum (int i, int j)` or `int sum(int, int)`
  - `void display(int i)` or `void display(int)`
- Definition – describe arguments and return value, and gives the code

```
int sum (int i, int j){
   return i+j;          /*  int s = i + j; return s; */
}

void display (int i)
{ printf("this is %d", i);}
```

- **<stdio.h>** contains declaration (prototype) for **printf()**, **scanf()** etc. --- that why we "include" it

functions

```
#include <stdio.h>

/* function definition */
int sum (int i, int j){
   return i+j;
}

main()
{
  int x =2, y=3;
  printf( "%d and %d = %d\n", x,y, sum(x,y));
}
```

/* Contains declaration (prototype) of printf() */

functions

```
# include <stdio.h>

/* function declaration */
int sum(int, int);          /* int sum(int a, int b) */

main()
{
  int x =2, y=3;
  printf( "%d and %d = %d\n", x,y, sum(x,y));
}

/* function definition */
int sum (int i, int j){
   return i+j;
}
```

/* Contains declaration (prototype) of printf() */

C basics

- Compile and running Comments
- Basic syntax
  - Comments
  - Variables
  - Functions
  - **Basic IO functions**
  - Expression
  - Statements
  - Preprocessing: # include, # define

Basic I/O functions <stdio.h>

- Every program has a Standard Input: keyboard
- Every program has a Standard Output: screen
  - Can use redirection Unix < inputFile > outputFile

- **int printf (char *format, arg1, ...);**
  - Format and prints arguments on standard output (screen or > outputFile)
  - **printf("This is a test %d \n", x)**
- **int scanf (char *format, arg1, ...);**
  - Formatted input from standard input (keyboard or < inputFile)
  - **scanf("%x %d", &x, &y)**

Others

- **int getchar();**
  - Reads and returns the next char on standard input (keyboard or < inputFile)
- **int putchar(int c)**
  - Write the character c on standard output (screen or > outputFile)

/* format string */

- `printf("This is a test %d \n", x)`
  - Returns number of chars printed
  - back in Java 1.5 (year 2005)

/* conversion specification */

- Format string contains: 1) regular chars 2) conversion specifications
  - **%d next argument is an integer -- decimal**
  - **%s next argument is a ``string``**
  - **%c next argument is a character**
  - **%f is a floating point number**

- `printf("Add %d to %f get %f (%s)\n", 2, 3.5, 5.5, "yes");`

```
red% Add 2 to 3.5 you get 5.5 (yes)
```

- `int x;`
- `scanf("%d", &x)`
  - opposite to **printf**
  - Wait for standard input (keyboard or < inputFile), then assign input value to **x**

- Format string contains: 1) regular chars 2) conversion specifications
  - **%d** next argument is an integer -- decimal
  - **%s** next argument is a string
  - **%c** next argument is a character
  - **%f** is a floating point number

- **&x** → memory address of **x**. Details later. Take as it is for now.

```
#include <stdio.h>


main()
{
 int a; int b;
 printf("pls enter the number" );

 scanf( "%d",  &a);    /* assign value to a */

 b = a * 2;
 printf("double of the input is %d\n", b);
 }
```

- **&x** → memory address (pointer) to **x**. Details later. Take as it is for now.

```
#include <stdio.h>

int sum (int, int) /* function declaration (prototype) */

main()
{
 int a, b;
 printf("pls enter two numbers separated by blank: " );

 scanf( "%d  %d",  &a,&b);     /* assign value to a b  */

 printf("input %d and %d. sum is %d\n", a, b, sum(a,b));
}

int sum (int i, int j){
  return i+j;            /*  int s = i + j; return s; */
}
```

getchar, putchar (Ch 1.5)

- **`int getchar(void)`**
  - To read one character at a time from the *standard input*
  - returns the next input char each time it is called;
  - returns EOF when it encounters end of file.
  - EOF input: Ctrl-d (Unix) or Ctrl-z (Windows).
  - EOF value defined in <stdio.h> is -1.

- **`int putchar(int c)`**
  - Puts the character c on the *standard output*
  - returns the character written;
  - returns EOF if an error occurs.

getchar, putchar (ch 1.5)

- **copy.c**

```
#include <stdio.h>

main(){
 int c;
 c = getchar();
 while(c != EOF)
 {
   putchar(c);
   c = getchar();/*read next*/
 }
}
```

```
indigo 309 % a.out
hello ↵
hello
how are you ↵
how are you
^D
indigo 310 %
```

Actually get/put chars line by line!

getchar, putchar

- **char, line counting**

```
#include <stdio.h>

main(){
 int c, cC, lC;
 cC = lC = 0;

 c = getchar(); /* read 1 char */
 while(c != EOF)
 {
   cC ++;
   putchar(c);
   if (c == '\n') /*a newline char*/
     lC ++;

   c= getchar(); /* read again */
 }
 printf("cC:%d lC:%d\n",cC,lC);
}
```

```
indigo 337 % a.out
hello ↵
hello
how are you ↵
how are you
i am good ↵
i am good
^D
cC:28  lC:3
```

C basics

- Compile and running Comments
- Basic syntax
  - Comments
  - Variables
  - Functions
  - Basic IO functions
  - **Expression**
  - **Statements**
  - Preprocessing: # include, # define

Expression

- Formed by combine **operands** (variable, constants and function calls), using **operators**
- Has return values -- always
  - x+1
  - i<20 flase: 0 true: >+1
  - sum (i+j)
  - x=k+ sum(i,j)

Statements

- Program to execute
  - Ended with a ;
- Expression statement
  - Self: declaration statement `int x;`
  - Assignment statement `x=4;`
- Function call statement `printf("the result is %d");`
- Flow control statement (ch3)
  - `if else, for(), while, do while, switch`

Statements

- In ANSI-C: declarations can only appear at the **start** of block

```
{
  int i, j;
  i = 0;
  j = i+ 1
}
```

```
{
   int i;
   i = 0;

    int j;
   j = i+ 1
}
```

- C99 removed this restriction.
  - Legal in C99 (lab)

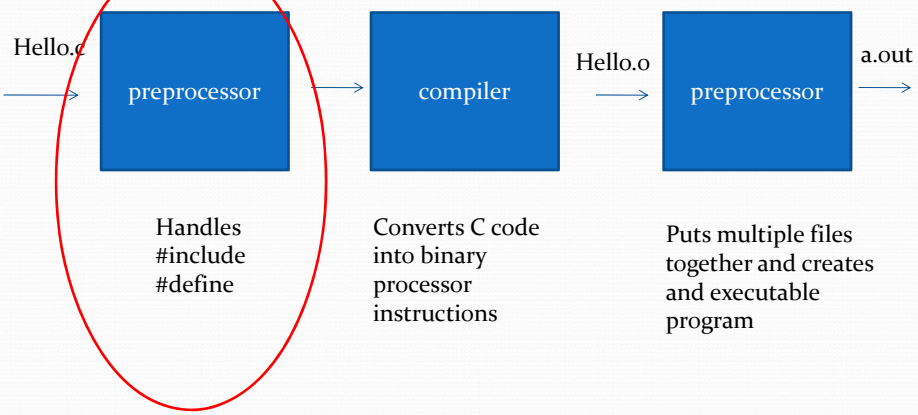
C basics

- Compile and running Comments
- Basic syntax
  - Comments
  - Variables
  - Functions
  - Basic IO functions
  - Expression
  - Statements
  - **Preprocessing: # include, # define**

```
indigo 307 % man gcc

NAME
       gcc - GNU project C and C++ compiler

SYNOPSIS
       gcc [-c|-S|-E] [-std=standard]
           [-g] [-pg] [-Olevel]
           [-Wwarn...] [-pedantic]
           [-Idir...] [-Ldir...]
           [-Dmacro[=defn]...] [-Umacro]
           [-foption...] [-mmachine-option...]
           [-o outfile] infile...

       Only the most useful options are listed here; see below for the
       remainder.  g++ accepts mostly the same options as gcc.

DESCRIPTION
       When you invoke GCC, it normally does preprocessing, compilation,
       assembly and linking.
```

C preprocessor #include #define

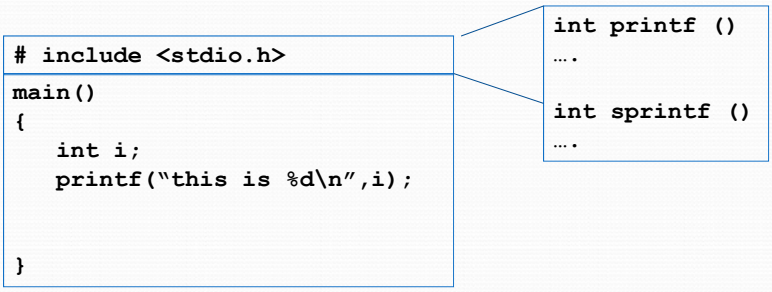
- C executables are built in three stages



preprocessing #include #define

- # include <file> -- include <stdio.h> library header file
- # include“file” -- include “file.h” programmer defined

- “includes” another file in the current file as if contents were part of the current file
- Textual replace. Nothing fancy
- File .h **.header** file
- Usually # define
- Declaration of variable, functions -- an example



define directive

- Syntax # define name value
  - Name called symbolic constant, conventionally written in upper case
  - Value can be any sequence of characters

```
#define   N    100
main() {
int  i  = 10 + N;
}
```

```
main() {
int  i  = 10 + 100;
}
```

  - Textual replacement
    - try yourself by `cpp file.c` or `cpp file.c > outputFile`

Summary and ``future work``

- Today: course introduction. C basics
  - Variables
  - Functions
  - Basic IO functions
    - **`scanf & printf, getchar & putchar`**
  - See an example code. How more comfortable you are?
- Next time (next week):
  - Finishes C basic syntax
  - C data, type, operators (Ch 2)
  - C flow controls (Ch 3)

Announcement

- Get textbooks and read KR Ch 1 (getchar/putchar). Ch3 (flow control). Ch7.4 (scanf).
- Read guided lab tour for CSE1020.
- Try the code in Ch1 and notes about basic IO (printf, scanf, getchar, putchar)!

- Labs starts this Wednesday (and next Tuesday)
- Learn/review guided lab tour
- Pre-lab exercise posted soon. Please try it yourself – only way to learn a programming language. Especially for something unsafe like C.

- Why Unix?
- Widely used
- Widely used operating system with time-sharing, multi-tasking, and
- multi-user. First written in assembler in 1969, rewritten in C in 1973 {
- portable!
- Performance
- stability,
- security and
- predictability
- Idea used in many other systems
- Many systems derived from it
- Linux is a clone of Unix
- embedded OS