Introduction to C

Brian Gregor
Research Computing, IS&T
bgregor@bu.edu
Outline

- Goals
- History
- Basic syntax
- Makefiles
- Additional syntax
Goals

- To learn the C language and syntax
- To write simple C programs
- To understand and modify existing C code
Getting started with the training room terminals

- Log on with your BU username
  - If you don’t have a BU username:
  - Username: Choose tutm1-tutm18, tutn1-tutn18
  - Password: on the board.

- On the desktop is a link to MobaXterm. Double click to open it.
Getting started on the SCC

- login using your account to the host `scc2.bu.edu`

- If you don’t have an SCC account use:
  - Username: `tutaN` where N is the number on the room laptop in front of you.
  - Password: on the board

- Type this command to make a folder in your home directory and copy in the tutorial files:

  ```bash
  /scratch/intro_to_c.sh
  ```
Getting started

- We’ll use a straightforward editor called *geany*

- To open or create a file:

  ```sh
gpany filename.c &
  ```
C History

- Developed by Dennis Ritchie at Bell Labs in 1969-73
  - Ancestors: BCPL -> B -> NB
  - Impetus was porting of Unix to a DEC PDP-11
    - PDP-11 had 24kB main memory!
  - See http://cm.bell-labs.com/cm/cs/who/dmr/chist.html
- See *The C Programming Language* by Kernighan & Ritchie (2nd ed.) (aka “K & R”)
- Official ANSI standard published in 1989 (“C89”)
  - Updated in 1999 (“C99”)
- C++ (1985)
  - Author: Bjarne Stroustrup (Bell Labs), 1979, “C with classes”
A word from Dennis Ritchie

- Despite some aspects mysterious to the beginner and occasionally even to the adept, C remains a simple and small language, translatable with simple and small compilers. Its types and operations are well-grounded in those provided by real machines, and for people used to how computers work, learning the idioms for generating time- and space-efficient programs is not difficult.
Compiled vs. Interpreted Languages

- **Interpreted languages**
  - when you type something, e.g., “x=y+z”, it is immediately converted to machine language and executed
  - examples: MATLAB, Python, R
  - advantage
    - interactive, allows fast development
  - disadvantage
    - generally uses more CPU/memory/time for given task
Compiled (cont’d)

- Compiled languages
  - examples: C, C++, Fortran
  - source code is written using a text editor
    - source code does nothing by itself – it’s just text
  - source code must be processed through a compiler, which
    - checks for correct syntax and semantics
    - translates source code into assembly, then assembles (or calls assembler) to produce machine code
    - passes machine code to linker, which creates executable
      - this is the file that you actually run
      - example: .exe file in Windows
      - default name in Unix/Linux: a.out
Behind the Scenes: The Compilation Process

- **header files**
  - `stdio.h`
  - `math.h`

- **source files**
  - `main.c`

**C preprocessor**
- Expanded source code file
  - Normally hidden
  - `gcc -E` to see output

**C compiler**

**Assembler code file**
- Normally hidden
  - `gcc -S` to see output

**Object code file**
- `main.o`

**linker**

**Executable**
- `main`

Command
- `gcc -o main main.c`
C syntax

- C is case-sensitive
- Spaces, linefeeds, etc., don’t matter except within character strings.
- Source lines generally end with semicolons
- Comments
  - notes for humans that are ignored by the compiler
  - C: enclosed by /* */
  - C++: // at beginning of comment
    - many C compilers also accept this syntax
  - Official advice: use them liberally so you can still understand your program next year
    - or next week, depending on your age
Declarations

- Variables and functions are **declared** to have a certain **type**.
- Common types include:
  - int
    - “integer”
      - number with no decimal places: -56, 857436
  - float, double
    - “floating-point”
      - number with decimal: 1.234, 4.0, 7.
    - float: single precision, 32 bits*, ~7 significant digits
    - double: double precision, 64 bits*, ~16 significant digits
  - complex, double complex (since C99)

*on most computers
Declarations (cont’d)

- char
  - “character”
  - enclosed in single quotes
  - ‘x’, ‘$’
  - character string is string of chars enclosed in double quotes
    - “This is a character string.”
Functions

- Source code largely consists of **functions**
  - each one performs some task
  - you write some of them
  - some are supplied, typically in **libraries**

- Every program contains at least one function, called **main**
  - ...but a library doesn’t contain a main function.

- Functions often, though not always, return a value, e.g.:
  - **int**, **float**, **char**, etc.
  - default return value is **int**
  - To be explicit about returning no value, declare as **void**
Functions (cont’d)

- Functions may, but do not have to, take arguments
  - arguments are inputs to the function
    - e.g., \( y = \sin(x) \)
- Code blocks, including entire functions, are enclosed within “curly brackets” \{ \} 
- main function is defined in source code as follows:

  ```c
  int main() {
    function statements
  }
  ```

Note: main is of type “int” because it returns an integer to the operating system. With the ‘bash’ shell in Linux, view this value via “echo $?”. 
Functions (3)

- Style note: some people like to arrange the brackets like so:
  ```
  int main( )
  {
    function statements
  }
  ```

- Either way is fine
  - Friendly advice: be consistent!

- A good editor can do automatic indentation, help you find matching brackets, etc.
How to say “hello, world”: printf

- `printf` is a function, part of C’s standard input/output library, that is used to direct output to the screen, e.g.,
  ```c
  printf("my string");
  ```

- The above syntax does not include a line feed. We can add one with:
  ```c
  printf("my string\n");
  ```
  where `\n` is a special character sequence representing a line feed
printf and stdio.h

- Some program elements, such as library functions like printf, are declared in header files, aka “include files.”
- Syntax*:
  
  ```
  #include <stdio.h> or
  #include “stdio.h”
  ```

- The contents of the named file are presented to the compiler as if you had placed them directly in your source file.
- In the case of printf, “stdio.h” informs the compiler about the arguments it takes, so the compiler can raise a warning or error if printf is called incorrectly.

*Note that the #include statement does not end with ‘;’
Exercise 1

- Write a “hello world” program in an editor
- Program should print a character string
- General structure of code, in order:
  - include the file “stdio.h”
  - define main function
  - use printf to print string to screen
- Save it to the file name hello.c
Compilation

- A compiler is a program that reads source code and converts it to a form usable by the computer/CPU, i.e., machine code.
- Code compiled for a given type of processor will not generally run on other types
  - AMD and Intel (the “x86” and “x86_64” family of CPUs) are compatible
- We’ll use gcc, since it’s free, readily available, and very high quality.
Compilation (cont’d)

- Compilers have numerous options
  - See gcc compiler documentation at http://gcc.gnu.org/onlinedocs/
  - gcc is part of the “GNU compiler collection,” which also includes a C++ compiler (g++), Fortran compiler (gfortran), etc.

- For now, we will simply use the –o option, which allows you to specify the name of the resulting executable

- A quick demo of some other common flags...
Compilation (3)

- In a Unix window:
  ```
  gcc hello.c -o hello
  ```
  - “hello.c” is source file name (compiler input)
  - “hello” is name of executable file (compiler output)

- Compile your code
- If it simply returns a Unix prompt it worked
- If you get error messages, read them carefully and see if you can fix the source code and re-compile
Compilation (4)

- Once it compiles correctly, type the name of the executable
  - `hello`
  - at the Unix prompt, and it will run the program
- should print the string to the screen
Variable Declarations

- Different variable types have different internal representations, so CPUs use different machine instructions for int, float, etc.
- We must tell compiler the types of variables by \textit{declaring} them prior to use.
- Example declarations:
  
  ```
  int i, jmax, k_value;
  float xval, elapsed_time;
  char aletter, bletter;
  ```
Arithmetic

- +, -, *, /
  - No power operator (see next bullet)

Math functions declared in math.h
- pow(x,y) raises x to the y power
- sin, acos, tanh, exp, sqrt, etc.
- for some compilers, need to add –lm flag (that’s a small el) to compile command to access math library
- complex functions declared in complex.h

Exponential notation indicated by letter “e”
e.g., $4.2 \times 10^3$ is expressed as $4.2e3$
Arithmetic (cont’d)

- Computer math
  - The equals sign is used for assignment:
    - Value of variable on left is replaced by value of expression on right
    - Many legal statements are algebraically nonsensical, e.g.,
      \[ i = i + 1; \]
Arithmetic (3)

- **++ and -- operators**
  - These are equivalent:
    
    ```
    i = i+1;
    i++; 
    ```
  - Available as prefix or postfix operator
    
    ```
    j = i++; // assign value of i to j, then increment i
    j = ++i; // increment i, then assign value to j
    ```

- **+= assignment**
  - These are equivalent:
    
    ```
    x = x + 46.3*y;
    x += 46.3*y;
    ```
Arithmetic (4)

- Pure integer arithmetic *truncates* result!
  
  \[ \frac{5}{2} = 2 \]
  \[ \frac{2}{5} = 0 \]

- Can convert types with *cast* operator

  ```
  float xval;
  int i, j;
  xval = (float) i / (float) j;
  ```
A Little More About printf

- To print a value (as opposed to a literal string), must specify a **format**
- For now we will use `%f` for a float and `%d` for an int
  - For floats, to specify 2 digits to the right of the decimal point, use `%.2f`
- Here’s an example of the syntax:
  ```c
  printf("My integer value is %d and my float value is %f \n", ival, fval);
  ```
- The values listed at the end of the printf statement will be embedded at the locations of their respective formats.
Exercise 2

- Write program to convert Celsius temperature to Fahrenheit and print the result.
  - Hard-wire the Celsius value to 100.0
    - We’ll make it an input value in a subsequent exercise
  - Don’t forget to declare all variables
  - Here’s the formula, which you will need to modify appropriately [hint, hint!] for your program:

\[ F = \frac{9}{5}C + 32 \]
**scanf**

- reads from keyboard
- 2 arguments
  - character string describing format, e.g.,
    - `%d` for integer
    - `%f` for float
  - address* of variable into which to put keyboard input
- example
  
  ```c
  int ival;
  scanf("%d", &ival);
  ```

*see next slide*
Address-of Operator

- Every variable has an address in which it is stored in memory
- In C, we sometimes need to use the address of a variable rather than its value
  - Will go into more detail when we discuss pointers
- Address-of operator & returns address of specified variable
  - &ival gives the address of the variable ival
  - rarely need to know actual value of address, just need to use it
Exercise 3

- Modify Celsius program to read value from keyboard
  - Prompt for Celsius value using printf
  - Read value using scanf
  - Rest of program can remain the same as last exercise
Arrays

- Declare arrays using [ ]
  
  float  x[100];
  char  a[25];

- Array indices start at zero
  - Declaration of x above creates locations for x[0] through x[99]

- Multi-dimensional arrays are declared as follows:
  
  int  a[10][20];
Character arrays

- Can’t directly assign character array values:
  ```
  char w[100];
  w = “hello”;
  ```
  This is wrong!

- Need to use `strcpy` function
  - declared in string.h
  ```
  strcpy(w, “hello”);
  ```
Character arrays (cont’d)

- Character strings (char arrays) always end with the null character (\0)
  - You usually don’t have to worry about it as long as you dimension the string 1 larger than its maximum possible length

```c
char name[5];
strcpy(name, "Fred");
```

```c
char name[4];
strcpy(name, "Fred");
```

- bugs: might or might not work,
  (depending on what follows ‘name’ in memory – might corrupt other variables)

```c
works
```
For Loop

- for loop repeats calculation over range of indices
  ```
  for(i=0; i<n; i++) {
    a[i] = sqrt( pow(b[i],2) + pow(c[i],2) );
  }
  ```
- for has 3 control statements separated by semicolons:
  - initialization
  - completion condition (i.e., if true, keep looping)
  - what to do after each iteration
- The body of the for loop follows the control section and is enclosed by curly brackets.
while

- while is a simpler alternative to for:
  ```c
  int i = 0;
  while (i < n) {
    a[i] = sqrt( pow(b[i],2) + pow(c[i],2) );
    i++;
  }
  ```
do

- do is like while, but executes the loop before testing the condition:

```c
int i = 0;
do {
doi = sqrt(pow(b[i],2) + pow(c[i],2));
i++;
} while (i < n);
```

- Note that after the first iteration, the logic of do is identical to while.
**break**

- **break** immediately exits the enclosing loop:

```c
int i = 0;
while (1) {
    a[i] = sqrt( pow(b[i], 2) + pow(c[i], 2) );
    i++;
    if (i >= n) break;
}
```
continue

- `continue` immediately jumps to the top of the enclosing loop:

```c
for (i=0; i<maxindex; i++) {
    if (a[i] == b[i]) continue;
    printf("Mismatch of a and b at index \%d\n", i);
    break;
}
```
Exercise 4

- Write program to:
  - declare
    - two float vectors of length 3
    - integer loop variable
    - float result variable
  - prompt for first vector, then read values using `for` loop
  - prompt for second vector, then read values using `for` loop
  - calculate dot product using `for` loop
  - print the result

- Possible to use “redirection of standard input” to avoid retyping each time:
  - `% echo 1 2 3 4 5 6 | dotprod`
Pointers

- When you **declare** a variable, a location of appropriate size is reserved in memory
- When you set its value, the value is placed in that memory location

```c
float x;
x = 3.2;
```
Pointers (cont’d)

- A pointer is a variable containing a memory address
- Declared using *
  
  ```c
  float *p;
  ```
- Often used in conjunction with address-of operator &
  
  ```c
  float x, *p;
  p = &x;
  ```
float x, *p;
p = &x;
Pointers (4)

- Depending on context, * can also be the dereferencing operator
  - Value stored in memory location pointed to by specified pointer
    \[ *p = 3.2; // \text{“the place pointed to by } p \text{ gets } 3.2” \]
- Common newbie error
  
  ```c
  double *p;
  *p = 3.2;
  ```
  
  Wrong! – p contains an unknown address

  ```c
  float x, *p;
  p = &x;
  *p = 3.2;
  ```
  
  correct

  Pop quiz: what is the value of x after this code runs?
Pointers (5)

- The name of an array is actually a pointer to the memory location of the first element
  - `a[100]`
  - “a” is a pointer to the first element of the array

- These are equivalent:
  - `x[0] = 4.53;`
  - `*x = 4.53;`
Points (6)

- If p is a pointer and n is an integer, the syntax \( p+n \) means to advance the pointer by \( n \) locations*

- These are therefore equivalent:
  
  \[
  x[4] = 4.53; \\
  *(x+4) = 4.53;
  \]

*i.e., for most machines, 4*n bytes for a float, and 8*n bytes for a double*
Pointers (7)

- In multi-dimensional arrays, values are stored in memory with *last* index varying most rapidly:*
  
  \[(a[0][0], a[0][1], a[0][2], \ldots)\]
  
  - Opposite of MATLAB, Fortran, R, et al.

- The two statements in each box are equivalent for an array declared as int a[5][5]:

  \[
  \begin{align*}
  a[0][3] &= 7; \\
  *(a+3) &= 7; \\
  a[1][0] &= 7; \\
  *(a+5) &= 7; \\
  \end{align*}
  \]

  * referred to as "row-major order"
sizeof

- Some functions require size of something in bytes
- A useful function – sizeof(arg)
  - The argument arg can be a variable, an array name, a type
  - Returns no. bytes in arg

```c
float x, y[5];
sizeof(x)       ( 4)
sizeof(y)       (20)
sizeof(float)   ( 4)
```
Dynamic Allocation

- Suppose you need an array, but you don’t know how big it needs to be until run time OR the compiler won’t let you create a really big static array.

- Tried and true method - use `malloc` function:
  ```c
  malloc(n)
  ```
  - `n` is no. bytes to be allocated
  - returns pointer to allocated space
  - declared in stdlib.h

- Many C compilers now accept “float f[n]”, where ‘n’ is determined at runtime.
Dynamic Allocation (cont’d)

- Declare pointer of required type
  ```
  float *myarray;
  ```
- Suppose we need 101 elements in array:
  ```
  myarray = malloc(101*sizeof(float));
  ```
- `free` releases space when it’s no longer needed:
  ```
  free(myarray);
  ```
Exercise 5

- Modify dot-product program to handle vectors of any length
  - Prompt for length of vectors (printf)
  - Read length of vectors from screen (scanf)
  - Dynamically allocate vectors (malloc)
  - Prompt for and read vectors (printf, scanf)
    - use for loop
  - Don’t forget to include stdlib.h, which contains a declaration for the malloc function
  - Note that the vectors will be declared as pointers, not fixed-length arrays
if/else

- Conditional execution of block of source code
- Based on relational operators
  
<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>==</td>
<td>equal</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>!=</td>
<td>not equal</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
if/else (cont’d)

- Condition is enclosed in parentheses, and code block is enclosed in curly brackets:
  
  ```c
  if (x > 0.0 && y > 0.0) {
    printf("x and y are both positive\n");
    z = x + y;
  }
  ```

- Note: curly brackets are optional if there is only a single statement in the code block (but this is a notorious source of bugs):

  ```c
  if (x > 0.0 && y > 0.0)
    z = x + y;
  ```
if/else (3)

- Can have multiple conditions by using else if

```c
if( x > 0.0 && y > 0.0 ) {
    z = 1.0/(x+y);
} else if( x < 0.0 && y < 0.0 ) {
    z = -1.0/(x+y);
} else {
    printf("Error condition\n");
}
```
Conditional expression ?:

- For simple if-else logic, the conditional expression `?:` can be used:
  - if `(a > b)` {
    - `z = a;`
  } else {
    - `z = b;`
  }

  is equivalent to
  - `z = (a > b) ? a : b;`
switch

- For multi-way branches on constant integer values:

```c
int j; …
switch (j) {
case 0:
    printf("Here, j = 0\n");
    break;
case 100:
    printf("Here, j = 100\n");
    break;
default:
    printf("Here, j != 0 && j != 100\n");
    break;
}
```
Functions

- C functions return a single value
- Return type should be declared (default is int)
- Argument types must be declared
- Sample function *definition*:

```c
float sumsqr(float x, float y) {
    float z;
    z = x*x + y*y;
    return z;
}
```
Functions (cont’d)

- Use of `sumsqr` function:
  
  ```
  a = sumsqr(b,c);
  ```

- Call by *value*
  
  - when function is called, copies are made of the arguments
  - copies are accessible within function
  - after function exits, copies no longer exist
Functions (3)

```c
b = 2.0; c = 3.0;
a = sumsqr(b, c);
printf("%f\n", b);
```  

```c
float sumsqr(float x, float y) {
    float z;
    z = x*x + y*y;
    x = 1938.6;  
    return z;
}
```  

will print 2.0

this line has no effect on b
Functions (4)

- If you want to change argument values, pass pointers

```c
int swap(int *i, int *j) {
    int k;
    k = *i;
    *i = *j;
    *j = k;
    return 0;
}
```
Functions (5)

- Let’s examine the following code fragment:
  ```c
  int a, b;
  a = 2; b = 3;
  swap(&a, &b);
  ```

- Memory after setting values of `a` and `b`
Functions (6)

- When function is called, copies of arguments are created in memory

```c
swap(&a, &b);
```

```c
int swap(int *i, int *j){ ...
}
```

- `i, j` are pointers to ints with values `&a` and `&b`
## Functions (7)

- What happens to memory for each line in the function?

```
int k;
```

```
k = *i;
```

### Memory Addresses and Variables

<table>
<thead>
<tr>
<th>Address</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>a</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>24</td>
<td>3</td>
</tr>
<tr>
<td>28</td>
<td>b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>i</td>
</tr>
<tr>
<td>52</td>
<td>j</td>
</tr>
<tr>
<td>56</td>
<td>24</td>
</tr>
<tr>
<td>60</td>
<td>k</td>
</tr>
</tbody>
</table>

```k = *i;```
Functions (8)

\[
\begin{align*}
&\text{address} & \text{variable} \\
&28 & 3 & b \\
&24 & 3 & a \\
&20 & 3 & a \\
&16 & & \\

\text{address} & \text{variable} \\
&28 & 2 & b \\
&24 & 2 & b \\
&20 & 3 & a \\
&16 & & \\

\text{address} & \text{variable} \\
&60 & 2 & k \\
&56 & 24 & j \\
&52 & 20 & i \\
&48 & & \\
\end{align*}
\]

\[\ast i = \ast j;\]

\[\ast j = k;\]
## Functions (9)

```c
return 0;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

![Variable addresses](image)
Exercise 7

- Modify dot-product program to use a function to compute and return the dot product
  - The function definition should go after the includes but *before* the main program in the source file
  - Arguments can be an integer containing the length of the vectors and a pointer to each vector
  - Function should only do dot product, no i/o
  - Do not give function same name as executable
    - I called my executable “dotprod” and the function “dp”
Function Prototypes

- C compiler checks arguments in function calls
  - number
  - type
- Multiple files are compiled separately, so if function definition and function call are not in same file, need means of determining proper arguments, etc.
  - this is done through *function prototypes*
Function Prototypes (cont’d)

- Prototype looks like 1\textsuperscript{st} line of function definition
  - return type
  - name
  - argument types

  \texttt{float dp(int n, float *x, float *y);} \\

- Argument names are optional:

  \texttt{float dp(int, float*, float*)}
Function Prototypes (3)

- Prototypes are often contained in include files

```c
/* mycode.h contains prototype for myfunc */
#include "mycode.h"
int main(){
  ...
  myfunc(x);
  ...
}
```
Basics of Code Management

- Large programs usually consist of multiple files
- Some programmers create a separate file for each function
  - Easier to edit
  - Can recompile one function at a time
- Files can be compiled, but not linked, using –c option; then object files can be linked later
  
  ```
gcc –c mycode.c
gcc –c myfunc.c
gcc –o mycode mycode.o myfunc.o
```
Exercise 8

- Put dot-product function and main program in separate files
- Create header file
  - function prototype
  - .h suffix
  - include at top of file containing main
- Compile, link, and run
Exercise 9a

- At terminal prompt, copy sample makefile from solutions folder, e.g.:
  - `scc1% cp ~/solutions/ex09/Makefile .`
- Modify so it works with the filenames you are using.
- Use **make** to build your code using the new makefile.
Exercise 9b

- Type `make` again
  - should get message that it’s already up to date
- Clean files by typing `make clean`
  - Type `ls` to make sure files are gone
- Type `make` again
  - will rebuild code
- Update time stamp on header file
  - `touch dp.h`
- Type `make` again
  - should recompile main program, but not dot product function
Addendum: Makefile macros

- Macros* can be used in makefiles to make textual substitutions:
  
  ```
  OBJECTS=dotprod.o dp.o
dotprod: $(OBJECTS)
  ```

- Possible to make flexible Makefile template, with small number of macros at top and boilerplate following. (See next slide, and also solutions/ex09/Makefile2.)

*GNU documentation uses the term “variables” for macros
Addendum: Makefile macros (cont’d)

OBJECTS=dp.o dotprod.o
INCLUDE_FILES=dp.h
EXECUTABLE=dotprod

.SUFFIXES:
.SUFFIXES: .c .o
%.o: %.c
gcc -c $<
$(EXECUTABLE): $(OBJECTS)
gcc -o $@ $(OBJECTS)
$(OBJECTS): $(INCLUDE_FILES)
clean:
   rm -f *.o $(EXECUTABLE)

Note: GNU “pattern rule” is being used instead of older “suffix rule”
C Preprocessor

- Initial processing phase before compilation
- Directives start with `#`
- We’ve seen one directive already, `#include`
  - `#include` inserts specified file in place of directive
- Another common directive is `#define`
  
  ```
  #define NAME text
  ```
  - `NAME` is any name you want to use
  - `text` is the text that replaces `NAME` wherever it appears in source code
C Preprocessor (cont’d)

- `#define` often used to define global constants
  ```c
  #define NX 51
  #define NY 201
  ...
  float x[NX][NY];
  ```

- `#define` can also be used to define a macro with substitutable arguments
  ```c
  #define ind(m,n) (m + NY*n)
  k = 5*ind(i,j);   k = 5*(i + NY*j);
  ```
Structures

- Can create a compound data structure, i.e., a group of variables under one name

```c
struct grid{
    int param;
    float x[100][100], y[100][100], z[100][100];
};
```

- Note semicolon at end of definition
Structures (cont’d)

- To declare a variable as a struct
  ```c
  struct grid mygrid1;
  ```

- Components are accessed using `.`
  ```c
  mygrid1.param = 20;
  mygrid1.x[0][0] = 0.0;
  ```

- Or, with struct pointer, access using `-`
  ```c
  struct grid *mygrid1;
  mygrid1 = malloc(sizeof(struct grid));
  mygrid1->param = 20;
  mygrid1->x[0][0] = 0.0;
  ```
typedef

- Typedef can be used to create synonyms for data types. It is often used with struct declarations, e.g.:

```c
struct rvec {
    int veclen;
    float *vec;
};
typedef struct rvec Rvec;
...
Rvec vec1, vec2;
```
i/o

- Often need to read from and write to files rather than screen
- Files are opened with a *file pointer* via a call to the *fopen* function
- File pointer is of type *FILE*, which is defined in stdio.h
- If fopen fails, NULL is returned.
i/o (cont’d)

- fopen takes 2 character-string arguments
  - file name
  - mode
    - “r”   read
    - “w”   write
    - “a”   append

FILE *fp;
fp = fopen(“myfile.d”, “w”);
i/o (3)

- Write to file using `fprintf`
  - Need `stdio.h`

- `fprintf` arguments:
  1. File pointer
  2. Character string containing what to print, including any formats
     - `%f` for float or double
     - `%d` for int
     - `%s` for character string
  3. Variable list corresponding to formats
i/o (4)

- Example:
  
  ```c
  fprintf(fp, "x = \%f\n", x);
  ```

- Read from file using `fscanf`
  - Arguments similar to `fprintf`, but, as with `scanf`, must supply addresses of variables:
    - ```c
      fscanf(fp, \"%f\", &x);
      ```
  - Returns integer equal to # items read (or EOF if error)

- When finished accessing file, close it
  ```c
  fclose(fp);
  ```
Exercise 12

- Modify dot-product code to read inputs, e.g.,
  3 1 2 3 4 5 6

- (size of vector and values for both vectors) from file “inputfile”. You can use a #define for the name; a better approach will be shown in a later exercise.
  - In main function, declare FILE pointer variable fp
  - Use fopen to open file and assign value to fp, and use if to ensure that fp is not equal to 0 (and exit if is).
  - Use fscanf to read file.
  - Note: you no longer need the prompts (printfs) for the vector size and vector data, so comment them out or remove them.
Additional Material

- The following slides are from the previous version of the C tutorial, which was 4 tutorial sessions in length.

- Topics: Makefiles, file I/O, command line arguments.
Makefiles

- Make is a Unix utility to help manage codes
- When you make changes to files, make will
  - automatically deduce which files have been modified and compile them
  - link latest object files
- **Makefile** is a file that tells the *make* utility what to do
- Default name of file is “makefile” or “Makefile”
  - Can use other names if you’d like
- See documentation here:
Makefiles (cont’d)

- Makefile contains different sections with different functions
  - The sections are *not* executed in order
- Comment character is `#`
  - As with source code, use comments freely
Makefiles (3)

- Simple sample makefile

```make
### suffix rule
.SUFFIXES:
.SUFFIXES: .c .o
.c.o:
    gcc -c $*.c

### compile and link
myexe: mymain.o fun1.o fun2.o fun3.o
    gcc -o myexe mymain.o fun1.o fun2.o fun3.o
```
Makefiles (4)

- Most makefiles contain one or more *rules*:
  - target: prerequisites
    - recipe
  - The target is a goal, oftentimes the name of an executable (but can be any name)
  - Prerequisites are files the target depends on
    - E.g., executable requires object files
  - Recipe generally contains means of producing target
  - May have multiple targets in a makefile
    - First target is default
Makefiles (5)

- Have to tell how to create one file type from another with a *suffix rule*
  
  `.c.o:
    gcc -c $*.c`

- The first line indicates that the rule tells how to create a `.o` file from a `.c` file
- The second line tells *how* to create the `.o` file
- `$*` is automatically set to the “stem” of the `.o` filename
- The big space before `gcc` is a tab, and you must use it!
Makefiles (6)

- Define all file suffixes that may be encountered
  .SUFFIXES:  .c  .o

- To override make’s built-in list of suffixes, first use a null .SUFFIXES: line:
  .SUFFIXES:
  .SUFFIXES:  .c  .o
Makefiles (7)

- Revisit sample makefile

### suffix rule
.SUFFIXES:
.SUFFIXES: .c .o
.c.o:
    gcc -c $*.c

### compile and link
myexe: mymain.o fun1.o fun2.o fun3.o
    gcc -o myexe mymain.o fun1.o fun2.o fun3.o
Makefiles (8)

- When you type "make," it will look for a file called "makefile" or "Makefile"
- searches for the first target in the file
- In our example (and the usual case) the object files are prerequisites
- checks suffix rule to see how to create an object file
- In our case, it sees that .o files depend on .c files
- checks time stamps on the associated .o and .c files to see if the .c is newer
- If the .c file is newer it performs the suffix rule
  - In our case, compiles the routine
Makefiles (9)

- Once all the prerequisites are updated as required, it performs the recipe
- In our case it links the object files and creates our executable
- Many makefiles have an additional target, “clean,” that removes .o and other files
  ```
  clean:
    rm -f *.o
  ```
- When there are multiple targets, specify desired target as argument to make
  ```
  make clean
  ```
Makefiles (10)

- Also may want to set up dependencies for header files
  - When header file is changed, files that include it will automatically recompile

- example:
  
  myfunction.o: myincludelfile.h
  
  - if time stamp on .h file is newer than .o file and .o file is required in another dependency, will recompile myfunction.c
  - no recipe is required
Addendum: standard file streams

- **stdin**
  - “standard input” //default = keyboard

- **stdout**
  - “standard output” //default = screen

- **stderr**
  - “standard error” //default = screen

- Can separate standard program output from error messages:
  ```c
  printf("%f %f %f\n", x, y, z);
  ...
  fprintf(stderr, "Error opening %s.\n", filename);
  ```
Binary i/o

- Binary data generally require much less disk space than ASCII data
- [optional in Linux as of C89]: use “b” suffix on mode
  \[
  \text{fp} = \text{fopen}(\text{"myfile.d"}, \text{"wb"});
  \]
- Use `fwrite`, `fread` functions (which take same arguments)
  \[
  \text{float x[100];}
  \text{fwrite( x, sizeof(float), 100, fp );}
  \]
  - Pointer to 1st element
  - No. of elements
  - No. of bytes in each element
  - File pointer
- Note that there is no format specification
  - We’re strictly writing binary, not ASCII, data
Command-Line Arguments

- It’s often useful to pass input values to the executable on the command line, e.g.,
  mycode 41.3 “myfile.d”
- Define `main` with two arguments:
  ```c
  int main(int argc, char *argv[])
  
  1. argc is the number of items on the command line, *including name of executable*
      - “argument count”
  2. argv is an array of character strings containing the arguments
      - “argument values”
      - argv[0] is pointer to executable name
      - argv[1] is pointer to 1st argument, argv[2] is pointer to 2nd argument, etc.
  ```
Command-Line Arguments (cont’d)

- Arguments are character strings, often want to convert them to numbers
- Some handy functions – atoi, atof
  - atoi converts string to integer
  - atof converts string to double
  - They are declared in stdlib.h
  - Example:
    
    ```c
    ival = atoi(argv[2])
    ```
    
    to convert the 2\textsuperscript{nd} argument to an integer
Command-Line Arguments (3)

- Often want to check the value of argc to make sure the correct number of command-line arguments were provided
- If wrong number of arguments, can stop execution with `exit` statement
  - Can exit with status, e.g.:
    ```
    exit(1);
    ```
  - View status by echoing `$?`:
    ```
    % echo $?
    1
    ```
Exercise 14

- Modify dot-product code to accept input filename on command line.
- Declare a character string variable and use strcpy to make copy of argv[1]
  - And remember to #include <string.h> at the top of the file.
- Add test on argc to ensure one command-line argument was provided
  - argc should equal 2 (since the executable name counts)
  - if argc is not equal to 2, print message and exit to stop execution
References

- Lots of books available
  - Kernighan & Ritchie, “The C Programming Language”
- gcc
- If you’d like to move on to C++
  - RCS C++ Programming Tutorial!
    - Week of Feb 5, 2018
  - Good C++ book for scientists:
    - Barton and Nackman, “Scientific and Engineering C++”
  - Quick and dirty C++ book:
    - Liberty, “Teach Yourself C++ in 21 Days”