Integrating R and C/C++
Day 2

Robert Putnam - putnam@bu.edu
Katia Oleinik - koleinik@bu.edu
Information Services and Technology
Introduction

- Welcome back!
- Agenda
  - Day two
    - C wrap-up
      - Review dot product code, then touch on functions/prototyping, make, struct, cpp, and I/O.
  - Optimization (C, C with GSL)
  - Metropolis (R, C, R+C, R+Rcpp)
  - LM (Rcpp+RcppGSL)
  - Your applications

For future reference, slides and code available here: http://www.bu.edu/tech/research/training/tutorials/list/
dotprod.c

#include <stdio.h>
#include <stdlib.h>

int main() {
    int i, veclen;
    float *v1, *v2, d;
    printf("Please input size of vectors: ");
    scanf("%d", &veclen);
    v1 = malloc(veclen*sizeof(float));
    v2 = malloc(veclen*sizeof(float));
    printf("Please input vector #1: ");
    for(i=0;i<veclen;i++) {
        scanf("%f", v1+i);
    }
    printf("Please input vector #2: ");
    for(i=0;i<veclen;i++) {
        scanf("%f", v2+i);
    }
    d = 0.0;
    for(i=0;i<veclen;i++) {
        dp += *(v1+i) * *(v2+i);
    }
    printf("Dot product = %7.2f\n", d);
}
if/else

- Conditional execution of block of source code
- Based on relational operators
  
  - `<`       less than
  - `>`       greater than
  - `==`      equal
  - `<=`      less than or equal
  - `>=`      greater than or equal
  - `!=`      not equal
  - `&&`      and
  - `||`      or
if/else (cont’d)

- Condition is enclosed in parentheses
- 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;
}
```
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”);
}
```
Functions

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

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


Functions (cont’d)

- Use of sumsqr function:
  \[ a = \text{sumsqr}(b, c); \]
- Call by value
  - when function is called, copies are made of the arguments
  - scope of copies is scope of function
    - after return from function, copies no longer exist
Functions (3)

\[
b = 2.0; \quad c = 3.0;
\]
\[
a = \text{sumsqr}(b, c);
\]
\[
\text{printf}(%"f", b); \quad \text{will print 2.0}
\]

float sumsqr(float x, float y) {
    float z;
    z = x*x + y*y;
    \textcolor{red}{x = 1938.6; \quad \text{this line has no effect on } b}
    return z;
}
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;
}
```
Exercise 7

- Modify dot-product program to use a function to compute 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”

- [solution](#)
Function Prototypes

- C compiler checks arguments in function definition and calls
  - number
  - type
- If definition and call are in different files, compiler needs more information to perform checks
  - this is done through function prototypes
Function Prototypes (cont’d)

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

  \begin{verbatim}
  float dp(int n, float *x, float *y);
  \end{verbatim}

- Argument names are optional:

  \begin{verbatim}
  float dp(int, float*, float*);
  \end{verbatim}
Function Prototypes (3)

- Prototypes are often contained in include files
  
  /* mycode.h contains prototype for myfunc */
  
  #include "mycode.h"
  
  int main()
  {
    ...
    myfunc(x);
    ...
  }
Basics of Code Management

- Large codes 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
- solution
Makefiles

- `make` is a Unix utility to help manage codes.
- When you make changes to files, it 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.
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

### 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)

- Have to define all file suffixes that may be encountered
  .SUFFIXES: .c .o

- Just to be safe, delete any default suffixes first with a null .SUFFIXES: command
  .SUFFIXES:
  .SUFFIXES: .c .o
Makefiles (5)

- Have to tell how to create one file suffix 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 the root of the file name
- The big space before gcc is a tab, and you must use it!
Finally, everything falls together with the definition of a **recipe**

```
  target: prerequisites
  recipe
```

- The target is any name you choose
  - Often use name of executable

- Prerequisites are files that are required by other files
  - e.g., executable requires object files

- Recipe tells what you want the makefile to do

- May have multiple targets in a makefile
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 command
  ```
  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: myincludefile.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
Exercise 9a

- Create a makefile for your dot product code
- Include 2 targets
  - create executable
  - clean
- Include header dependency (see previous slide)
- Delete old object files and executable manually
  - `rm *.o dotprod`
- Build your code using the makefile
- [solution](#)
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
C Preprocessor

- Initial processing phase before compilation
- Directives start with 
- We’ve seen one directive already, #include
  - simply includes 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];
  ```

- Also handy to specify precision
  
  ```c
  #define REAL double
  ...
  REAL x, y;
  ```
C Preprocessor (3)

- Since `#define` is often placed in header file, and header will be included in multiple files, this construct is commonly used:
  ```c
  #ifndef REAL
  #define REAL double
  #endif
  ```

- This basically says “If REAL is not defined, go ahead and define it.”
C Preprocessor (3)

- Can also check values using the `#if` directive
- In the current exercise code, the function `fabsf` is used, but that is for floats. For doubles, the function is `fabs`. We can add this to dp.h file:

```c
#if REAL == double
#define ABS fabs
#else
#define ABS fabsf
#endif
```
C Preprocessor (4)

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

- Be careful to use ( ) when required!
  - without ( ) above example would come out wrong
    ```c
    k = 5*i + NY*j  // wrong!
    ```
Structures

- Can package a number of variables under one name
  ```c
  struct grid{
    int nvals;
    float x[100][100], y[100][100], jacobian[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.nvals = 20;
  mygrid1.x[0][0] = 0.0;
  ```

- Handy way to transfer lots of data to a function
  ```c
  int calc_jacobian(struct grid mygrid1){…
  ```
i/o

- Often need to read/write data from/to files rather than screen
- File is associated with a file pointer through a call to the fopen function
- File pointer is of type FILE, which is defined in stdio.h.
i/o (cont’d)

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

```c
FILE *fp;
fp = fopen("myfile.d", "w");
```

Note: NULL is returned on error
i/o (3)

- Write to file using `fprintf`
  - Need `stdio.h`
- `fprintf` has 3 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)

- Special character `\n` produces new line (carriage return & line feed)
  - Often used in character strings
    
    "This is my character string.\n"

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

- Read from file using `fscanf`
  - arguments same as `fprintf`
  - Return type = int: EOF on error, otherwise # items read

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

- Modify dot-product code to read inputs (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 the next exercise.)

- solution
Command-Line Arguments

- It's often convenient to type some inputs on the command line along with the executable name, 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"
     - `argc[0]` is pointer to executable name
     - `argc[1]` is pointer to 1st argument, `argc[2]` is pointer to 2nd argument, etc.
Command-Line Arguments (cont’d)

- Arguments are character strings. We often want to convert them to numbers.
- Some handy functions:
  - atoi converts string to integer
  - atof converts string to double
  - They live in stdlib.h
  - arguments are pointers to strings, so you would use, for 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.:
    ```c
    exit(1);
    ```
  - With csh shell, view status by echoing `$status`:
    ```c
    % echo $status
    1
    ```
Exercise 14

- Modify dot-product code to enter a maximum vector length as a command-line argument (and give an error if the value read from the file exceeds it).
- Use `atoi`
- Add test on `argc` to make sure a command-line argument was provided
  - `argc` should equal 2, since the executable name counts
  - if `argc` is not equal to 2, print message and `return` to stop execution

solution
R -> C Agenda

- Benchmark/profile R code
  - Is it a good candidate for speedup? Tools: system.time, Rprof(), cmpfile, etc.

- Convert to C standalone

- Modify C code to be callable from R
  - [http://cran.r-project.org/doc/manuals/R-exts.html](http://cran.r-project.org/doc/manuals/R-exts.html)

- Use Rcpp for simpler R<->C interface
R->C: Using the .Call interface

- C functions called from R will receive pointers to R objects. These pointers are called SEXPs (for "S expression pointer", which shows R's roots in the language S).
- Macros and functions are provided in R header files (R.h and Rdefines.h [or Rinternals.h]) which provide access to the data pointed to by SEXPs.
- C functions called from R must return a SEXP (or R_NilValue).
- If a C function called from R creates new R objects, those objects must be PROTECTed from being reaped by the R garbage collector.
R->C: Using the .Call interface (cont.)

- Use Rprintf instead of printf, and don't include stdio.h.
- Don’t call exit (as this will stop your R session).
- Compile at the command line:
  - R CMD SHLIB file.c
- Load into R
  - > dyn.load(“file.so”)
- Use .Call interface
  - > .Call(“myfun”, arg1, arg2,...)

Note: There is another R->C interface (“.C”), which we are not covering. It has largely been superceded by .Call.
Exercise

- Write “hello, world” using the .Call interface
  - Include R.h and Rdefines.h
  - Use Rprintf
  - Return R_NilValue
Survey

- Please fill out the course survey at http://scv.bu.edu/survey/tutorial_evaluation.html