12.010 Computational Methods of Scientific Programming
Lecture 8

Today’s lecture
- Start C/C++
- Basic language features

Web page http://www-gpsg.mit.edu/~lah/12.010
C History and Background

- Origins 1973, Bell Labs
- ANSI C – standardized 1989, X3.159-1989
- Ritchie “C is quirky, flawed and an enormous success”
  - http://cm.bell-labs.com/cm/cs/who/dmr/chist.html
- Compiled language (gcc, cc )
  - Good runtime performance, more control e.g memory utilisation
  - Portability, licensing, versatility
  - C apps: Matlab, Mathematica, + Linux netscape, IE, …
- C++ superset of C i.e. C plus some additional concepts – more on these later
C Variables (and C++)

- Variable names
  - Lower or upper case + lower, upper, digit, _ ...
  - e.g. x, CO2, DENSITY, area_of_polygon
  - Names ARE case sensitive: CO2 and co2 not same
  - Keywords are reserved (also case sensitive)
    - if, for, while, return, int, float ........
Data types and basic arrays

- int, float, double, char, short, uint, long int
- int – 4 byte integer (long = 8 byte), short – 2 byte integer, float 32-bit, double 64-bit, char – 1 byte
- [] for arrays
- Examples
  - int a [10], b[10][10];
  - char c[20];
  - double x, area_of_circle, radius;
- Also macros
  - #define PI 3.14159
- Everything must be declared
- /* */ comments
Executable Statements 1

- Statement terminator is the `;`. All C-statements end with this character (common compile error is to forget to put `;` at end of a statement.
- Assignment
  - `#define PI 3.14159`
    - `double x, radius, area_of_circle;`
    - `radius=2.;`
    - `area_of_circle = PI*radius*radius;`
- Assignment operators:
  - `variable op= expression` is equivalent to
    - `Variable = variable op expression`
- Operators are: `+= -= *= /= %= >>= <<= &= ^= |=`
- Example: `k *= 3+x` is the same as `k=k*(3+x)`
- Some of the operators above `>> << & |` are bit operators and rarely seen. `%` is the modulus operator (a%b is a modulus b; remainder after removing as many b's are possible from a e.g. `7%3 = 1`)
- Multiple `=` and be used on a line e.g., `a=b=c=d;` right to left evaluation
Executables: Conditionals

- Conditional statements are like fortran except no endif statement. The code to be executed in contained in { }’s unless it is just one statement.
  - if ( radius == 0. ) {
    inv_radius = 0.;
  } else {
    inv_radius = 1./radius;
  }
  - We could above used ‘} else inv_radius = 1./radius; ‘
  - If( radius == 0. ) { code }
    else if ( condition ) { code }
  - It is allowed to have to an empty statement by just having ;
    after the if or in a sequence of if else if statements.
Executable Statements 2

• Increment int type by 1 methods in c:
  – Postfix evaluated after expression
  – Prefix evaluated before expression
    int i;
    i = i+1.;
    ++i; /* prefix mode */
    i++; /* postfix mode */
  – When used in an expression prefix mode increments first e.g.,
    c = ++a + ++b; gives difference answer to c = a++ + b++;
  – These commands are used because increment by 1 is a
    machine instruction (faster than load 1 to register and add to
    another register)
• Changing variable type: cast
  – double x; int i;
  – x = (double) i; /* changes integer i to double type)"

Infix notation is the notation used where arguments are entered first and the operator.
Executable Statements 3

- Loops using the “for” construction.
  ```
  int i,j,k;
  double b[10][10];
  k=0;
  for (j=0;j<10;++j) {
    for (i=0;i<10;++i) {
      b[j][i] = (double) k++;
    }
  }
  ```
- Fortran style “do while structure” but the while appears at the end of the construction
  ```
  do { statements; } while (condition);
  ```
Standard libraries

- no math functions, no I/O functions etc are included in standard code. Header files are need to define constants and functions.

```c
#include <math.h>
x = cos(y);
z = cos(PI);
```

```c
#include <stdio.h>
printf("Hello\n");
fprintf(stdout,"Hello\n");
```

`<math.h> == /usr/include/math.h` — C source files
`<stdio.h> == /usr/include/stdio.h`
#include <stdio.h>
#include <math.h>
int i=1;
main()
{
    int j;
    j = 2;
    printf("Hello\n");
fprintf(stdout,"Hello\n");
fprintf(stdout,"pi == %f
\n",M_PI);
fprintf(stdout,"i == %d\n",i);
fprintf(stdout,"j == %d\n",j);
}
Functions

• Definition method. All modules are functions in C and may or may not return a result (type void if no return).
  
  type fname(type arg1, type arg2)
  {
    /* Local variables and executable code */
  }

• Calling a function
  
  fname(arg1, arg2); /* type void call */
  result = fname( arg1, arg2); /* result and fname same type*/

• Prototype defines how a function should be called
  
  type fname(type, type);

• In C, none of the arguments passed to a functions can be changed -- call by value. Addresses can be passed and the values stored at these addresses can be changed.
Function Example

```c
int mymax(float, float); /* Prototype */
main ()
{
    float a,b; int ans;
    a=b=2.;
    ans= mymax(a,b) /* returns 1 if a > b, 2 if b > a, 0 otherwise */
}
int mymax(float a, float b)
{
    if ( a > b ) return 1;
    if ( b > a ) return 2;
    return 0;
}
```

10/05/2010 12.010 Lec 08
Call by reference

```c
int mymax(*float, *float); /* Prototype. The *float is a pointer to
(address of) a floating point number */
main()
{
  float a,b; int ans;
  a=b=2.0;
  ans= mymax(&a,&b); /* 1 if a > b, 2 if b > a, 0 otherwise */
  /* set a and b = to max. value */
}
int mymax(float *a, float *b)
{
  if ( *a > *b ) (*b=*a;return 1;)
  if ( *b > *a ) (*a=*b;return 2;)
  return 0;
}
```

We will return to pointers later.

Only got to here 2006 TAH
Addresses - *, &

- C allows very explicit addressing of memory locations with the concept of “pointers” (points to memory location)

```c
short a; short *ptr_to_a;
a = 1;
ptr_to_a = &a;
```

Computer Memory

```
0x00 0xFFFF
```

![Diagram showing memory allocation and pointer usage]

&a (value stored at &a)
The difference between a standard object module (the result of compiling with the -c option) and a library (such as libraries.a above) is that only the modules that are needed are loaded from the library.

When an object module (.o extent) or a Fortran module (.f extent) is included all the routines from these are linked to the program even if they are not needed.
C Basic Summary

- Origins of C – Compiled language; K&R, ANSI
  - V. versatile i.e Matlab, Mathematica, Fortran compiler, Linux, Netscape cores - all are mostly in C.
- Basic Syntax
  - case sensitive, semi-colon required at end of statements, loops, conditionals (==).
- Simple program
  - Standard libraries (stdio.h, math.h)
  - Calling a function
- Call by reference v. call by value
  - double a; double *ptrToA; ptrToA = &a;
C preprocessor (CPP)

- precompile macros and options; “compiler” proper does not see CPP code.
- Also stand alone cpp; other compilers e.g. .F files fortran – (not in java!)
  - #include - file inclusion
  - #define - macro definition
  - #undef - undefine macro
  - #line - compiler messages line number (not really for general use)
  - #if, #ifdef, #ifndef, #elif, #else, #endif - Conditional compilation
  - __FILE__, __LINE__ (ANSI C).
C preprocessor (CPP)

- `#include "fred.h"` - includes contents of file fred.h in program. `-l cpp` flag sets path to search for fred.h

- `#define PI 3.14159` - substitutes 3.14159 everywhere PI occurs in program source. (except in quotes).

- `#undef PI` - stops substitution

```c
#ifdef PI
    printf("pi is set to %f in file %s\n", PI, __FILE__);
#else
    printf("pi is not set. Line %d file %s\n",
            __LINE__, __FILE__);
#endif
```
C preprocessor (CPP)

- Macros with args
  
  ```c
  #define _getaddress(a) (&a) /* This macro returns address of a */
  main() { double n; double *ptrToN;
     ptrToN = _getaddress(n); }
  ```

- Compiler actually sees code below
  
  ```c
  main() { double n; double *ptrToN;
     ptrToN = &n; }
  ```

- Often used for debugging
  
  ```c
  #ifdef debug
  #define _D(a) a
  #else
  #define _D(a)
  #endif
  ```

Place we are likely to get to in first lecture.
Summary

• C programming language. Similar to fortran in many ways but with:
  – Somewhat less rigid syntax
  – More explicit memory addressing methods
  – “short-cut” ways of doing operations that can be very fast on some CPU’s.
• Next lecture we go into more detail in pointers and call by reference and call by value.