Plan of the day

Functions

Pointers

More on functions

Functions

Example function

```
double coulombsLaw(double q1, double q2, double r) {
// Coulomb's law for the force acting on two point charges
// q1 and q2 at a distance r.  MKS units are used.

    double k = 8.9875e9;     // nt-m**2/coul**2
    return k * q1 * q2 / (r * r);

}
int main() {
    cout << coulombsLaw(1.6e-19, 1.6e-19, 5.3e-11)
         << " newtons" << endl;
    return 0;
}
```

- first token is type of returned object
- second token is function name
- argument names are proceeded by their type
- function body is within `{}`
- return statement can be expression or variable
- if keyword `void` is used as return type, then function is like Fortran `SUBROUTINE`
- if no arguments, `void` can be used or leave empty

Function Prototypes

Will this work?

```
int main() {
    cout << coulombsLaw(1.6e-19, 1.6e-19, 5.3e-11)
         << " newtons" << endl;
    return 0;
}
```

- C++ checks types and number of arguments
- does standard type conversions if necessary
- C++ checks return type
- can be compilation error if checks fail or type conversion is not possible

Will this work?

```
extern double coulombsLaw(double q1, double q2, double r);
int main() {
    cout << coulombsLaw(1.6e-19, 1.6e-19, 5.3e-11)
         << " newtons" << endl;
    return 0;
}
```

- `extern` keyword says that the function is external and needs to be included in the link step
- statement ends with `;` where body would have been

Declarations and Definitions

On the one hand, programs must be broken up into units which are compiled separately

- standard functions compiled and put in libraries
- analysis code compiled and linked to library

On the other hand, functions and other externals must be declared before their use.

```
extern double sqrt(double);

double x, y, z, r;
//
r = sqrt(x*x + y*y + z*z);
```

- `sqrt(double)` and `sqrt(double x)` are equivalent in the declaration statement

What would happen if declaration we used did not correspond to function in the library?

To ensure consistency, we force the library function and the declaration we use to share same declaration

Header files used with definition

In `math.h`, we have declarations

```
extern double sqrt(double);
extern double sin(double);
extern double cos(double);
// and many more
```

In `math.c`, we have definition

```
#include <math.h>
double sqrt(double x) {
//
    return result;
}
double sin(double x) {
//
    return result;
}
```

- `#include` is like Fortran include
- declaration in header files is used in compilation of the library function
- any mismatch between declaration and definition is flagged as error.

Header files and user code

In `math.h`, we have declarations

```
extern double sqrt(double);
extern double sin(double);
extern double cos(double);
// and many more
```

in `user.c` we have definition of user code

```
#include <math.h>

double x, y, z, r;
//
r = sqrt(x*x + y*y + z*z);
```

- use same header file in user code
- user code then compiles correctly with implicit conversions as needed

Extern Data Declarations

Data can be external

```
extern double aNum;

int foo() {
  cout << aNum << endl;
  return 0;
}
```

- external data is like data in Fortran `COMMON` block
- rarely used feature in C and even less in C++

Defining extern data

```
double aNum = 1234.5678;

int main() {
  foo();
  return 0;
}
```

- definition must only be done once
- definition is like those in Fortran `BLOCK DATA`

Static Functions

Static function declaration

```
#include <math.h>

static double exp_random(double mu) {
    return -mu * log(random());
}

void simulation1() {
   double x1 = exp_random(2.1);
    // ...
}
```

- `static` keyword means local in scope of file
- definition substitutes for declaration within file
- still must come before use

Static Data

Consider

```
#include <iostream>
using namespace std;

int counter() {
  static int count = 0;
  count++;
  return count;
}

int main() {
  int i;
  i = counter();
  cout << i << ", ";
  i = counter();
  cout << i << endl;
  return 0;
}
```

- static objects retains its value after return from function
- behaves like Fortran local data under VM or VMS
- like Fortran local data under UNIX with `SAVE` option
- rarely used feature

Default Function Arguments

One can specify the value of the arguments not given in the call to a function

Example

```
#include <math.h>
extern double log_of(double x, double base = M_E);
    // M_E in <math.h>
```

- can be used like

```
#include <ch5/logof.h>

x = log_of(y);      // base e
z = log_of(y, 10);  // base 10
```

- all arguments to the right of the first argument with default value must have default values
- once first default value is used, the remaining ones must also be used
- value of the default must be visible to the caller

Functions in C

Function declaration and prototype is the same in C except

- if header inclusion is missing in calling program, then C compiler gives warning and takes default argument types (`long` or `double`) and return type (`int`)
- if header file is included and there is a mismatch between arguments or return type, the C compiler only gives warnings
- you don't see the warnings unless you ask for them (see man pages for their flag)
- `gcc` gives excellent warnings with `-Wall` flag
- ignoring these warnings can be a disaster on some RISC machines
- no default arguments

Header Files

In a large program, it is possible that a header file might get included twice

Use C preprocessor to avoid to double inclusion

```
#ifndef MATH_H
#define MATH_H
extern double sin(double x);
extern double cos(double x);
extern double tan(double x);
// etc
#endif // MATH_H
```

- cpp buils tempoary file for compiler
- `#ifndef` is C preprocessor directive saying “if not defined”
- `MATH_H` is preprocessor macro variable and is upper case by convention
- `#define` defines a macro variable but in this case doesn’t give it a value
- `#endif` ends the `#ifndef`
- this structure seen in all system header files
- same for C and C++

namespace

To avoid conflict of function names from different packages, encapsulate in namespace

```
#ifndef MATH_H
#define MATH_H
namespace std {
extern double sin(double x);
extern double cos(double x);
extern double tan(double x);
// etc
} // end namespace
#endif // MATH_H
```

Now to use, we must do one of following

```
#include <cmath>

y = std::sin ( x ); // tedious
// or
using std::sin; // explicit
y = sin ( x );
// or
using namespace std; // sloppy
y = sin ( x );
```

Good rule: never use `using` in header file, but ok in implementation file

The (dreaded) Pointers

A pointer is an object that refers to another object

Declare it thus

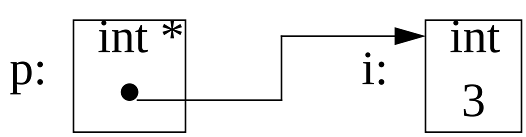
```
int* p;
int *q;
```

- either form can be used; the later is prefered

Assign a value to the pointer

```
int i = 3;
int *p = &i;
```

- read `&` as “address of”
- data model is thus



Watch out!

```
int *p, i;
p = &i; // i is an int
```

Dereferencing pointers

Consider

```
#include <iostream>
using namespace std;

int main() {
  int* p;
  int j = 4;
  p = &j;

  cout << *p << endl;

  *p = 5;
  cout << *p << " " << j << endl;

  if (p != 0) {
    cout << "Pointer p points at " << *p << endl;
  }
  return 0;
}
```

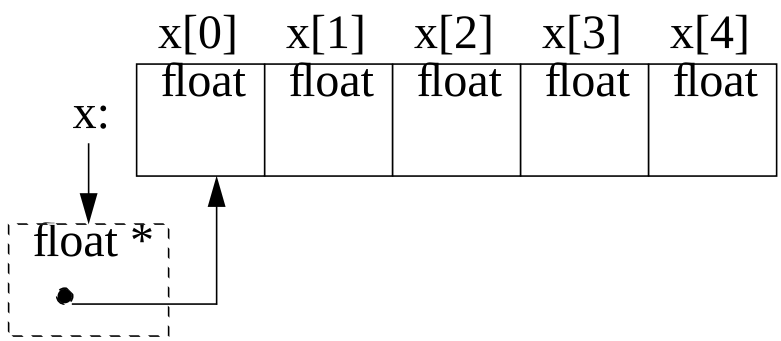
- `*p` derefences pointer to access object pointed at
- `*p` can be used on either side of assignment operator
- if `p` is equal to 0, then pointer is pointing at nothing and is called a *null* pointer.
- dereferencing a null pointer causes a core dump :-(

Pointers and Arrays

Consider

```
float x[5];
```

Our memory model is



- what does the label `x` mean?
- in Fortran, `foo(x)` is the same as `foo( x(1) )` is the same
- in C/C++, `x` is a pointer to the first element
- `*x` and `x[0]` are the same
- `x` and `&x[0]` are the same
- elements of an array can be accessed either way
- but `x` is a label to an array of object, not a pointer object

Pointer Arithmetic

A pointer can point to element of an array

```
float x[5];
float *y = &x[0];
float *z = x;
```

- `y` is a pointer to `x[0]`
- `z` is also a pointer to `x[0]`
- `y+1` is pointer to `x[1]`
- thus `*(y+1)` and `x[1]` access the same object
- `y[1]` is shorthand for `*(y+1)`
- integer add, subtract and relational operators are allowed on pointers

Examples

1. Summing an array Fortran style

```
float  x[5];
double sum;
int    i;
// some code that fills x
sum = 0.0;
for (i = 0; i < 5; i = i + 1) {
  sum = sum + x[i];
}
```

2. Summing an array C++ **style**

```
float  x[5];
// some code that fills x
double sum = 0.0;
for (int i = 0; i < 5; i++) {
  sum += x[i];
}
```

- we declare `sum` just before we need it
- we initialize `sum` with the declaration
- we use `i++` to indicate increment
- we use `sum +=` to indicate accumulation

More examples

3. Summing an array with pointer in Fortran style

```
float  x[5];
float  *y;
double sum;
int    i;
// code to fill x
sum = 0.0;
y = &x[0];
for (i = 0; i < 5; i = i + 1) {
  sum = sum + *y;
  y = y + 1;
}
```

4. Summing an array with pointer in C++ style

```
float  x[5];
// code to fill x
float  *y = x;
double sum = 0.0;
for (int i = 0; i < 5; i++) {
  sum += *y++;
}
```

- delay declaration until need
- use increment operator
- use `+=` assignment operator

Progression towards C++ style

Fortran style

```
sum = sum + *y;
y = y + 1;
```

Use add-and-assign operator

```
sum += *y;
y = y + 1;
```

Use postfix increment operator

```
sum += *y;
y++;
```

Combine postfix and dereference

```
sum += *y++;
```

- it takes some time to get use to writing in this style
- be prepared to read code written by others in this style
- don’t worry about performance issues yet

Examples of Pointer Arithmetic

Reverse elements of an array

```
float x[10];
// ... initialize x ...
float* left  = &x[0];
float* right = &x[9];
while (left < right) {
    float temp = *left;
    *left++  = *right;
    *right-- = temp;
}
```

Set elements of an array to zero

```
float x[10];

float* p = &x[10]; // uh?
while (p != x) *--p = 0.0;
```

- this terse style is typical of experienced C/C++ programmers
- most HEP code will not be so terse
- in C++, we wouldn't use pointers as much as in C

Runtime Array Size

In C++, one can dynamically allocate arrays

```
float* x = new float[n];
```

- `new` is an operator that returns a pointer to the newly created array
- note use of `n`; a variable
- not the same as Fortran's

```
SUBROUTINE F(X,N)
DIMENSION X(N)
```

where the calling routine "owns" the memory

- in C, one does

```
float *x = (float *)malloc( n*sizeof(float) );
```

In C++, to delete a dynamically allocated array one uses the `delete` operator

```
delete [] x;
```

- in C one uses the `free()` function

```
free(x);
```

Line fit example

Part 1

```
#include <iostream>
using namespace std;

void linefit() {
   // Create arrays with the desired number of elements
    int n;
    cin >> n;
    float* x = new float[n];
    float* y = new float[n];


    // Read the data points
    for (int i = 0; i < n; i++) {
        cin >> x[i] >> y[i];
    }

    // Accumulate sums Sx and Sy in double precision
    double sx  = 0.0;
    double sy  = 0.0;
    for (i = 0; i < n; i++) {
        sx += x[i];
        sy += y[i];
    }
```

- note first declaration of `i` carries forward
- will need to change in future

Line fit continued

Part 2

```
// Compute coefficients
    double sx_over_n = sx / n;
    double stt = 0.0;
    double  b = 0.0;
    for (i = 0; i < n; i++) {
        double ti = x[i] - sx_over_n;

        stt += ti * ti;
        b   += ti * y[i];

    }
    b /= stt;
    double a = (sy - sx * b) / n;


    delete [] x;
    delete [] y;



    cout << a << " " << b << endl;
}

int main() {
  linefit();
  return 0;
}
```

Character Strings

Character strings are special case of array and array initialization

```
char hello1[] = { 'H', 'i' };
```

- dimension of `hello1` is 2

The above is too tedious, so use double quotes

```
char hello2[] = "Hi";
```

- the dimension of `hello2` is 3
- the characters are '`H`', '`i`', and '`\0`'
- all string functions in C/C++ library expect the last character to be '`\0`'
- one frequently uses pointers to walk thru a string

```
char hello2[] = "Hi";
int n = 0;
for (char *p = hello2; *p !=0; p++) {
    n++;
}
// n == 2
```

Variable Scope, Initialization, and Lifetime

Consider

```
void f() {
    float temp = 1.1;
    int a;
    int b;
    cin >> a >> b;

    if (a < b) {
        int temp = a;  // This "temp" hides other one
        cout << 2 * temp << endl;
    }// Block ends; local "temp" deleted.
    else {
       int temp = b;  // Another "temp" hides other one
        cout << 3 * temp << endl;
    }

    cout << a * b + temp << endl;
}
```

- every pair of `{}` defines a new scope
- even a pair with out function, if, for, *etc.*
- variables declared in a scope are deleted when execution leaves scope

`for`-loop Scoping

Consider

```
for(int i = 0; i < count; i++) {
    if ( a[i] < 10 ) break;
}
cout << i << endl;
```

- note where `i` is declared
- the scope of `i` is the scope just outside the for-loop block
- used to work with many compilers

Current draft standard

- scope of `i` is *inside* for-loop block
- will need to declare `i` before `for` statement for `i` to have meaning after loop termination
- if declared in `for` statement, will need to repeat it for each `for` statement that follows
- vendor compilers will (eventually) change
- gcc ok, Microsoft?

Formal Arguments

Consider

```
void f(int i, float x, float *a) {
    i = 100;
    x = 101.0;
    a[0] = 0.0;
}

int j = 1;
int k = 2;
float y[] = {3.0, 4.0, 5.0};
f(j, k, y);
```

- what's the value of `j` after calling `f()`?
- C/C++ pass arguments by value, thus `j` and `k` are left unchanged
- `i`, `x`, and `a` are formal arguments and in the scope of `f()`
- upon calling `f()`, it is as if the compiler generated this code to initialize the arguments

```
int i = j;
float x = k;  // note type conversion
float *a = y; // init pointer to array
```

- thus `y[0]` does get set to 0.0

References

A way to reference the same location (C++ only)

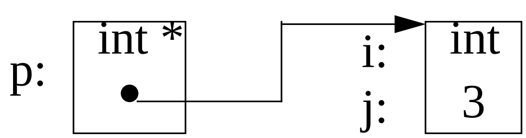
Reference

```
float x = 12.1;
float& a = x;
float &b = x;
```

- `a` and `b` are called a *reference*
- `a`, `b`, and `x` are all labels for the same object
- the position of the "&" is optional
- Don't confuse a reference and a pointer

```
int i = 3;   // data object
int &j = i;  // reference to i
int *p = &i; // pointer to i
```

- `i` *has* an address of a memory location containing `3`
- `j` *has* the *same* address as `i`
- the contents of `p` *is* the address of `i`



Reference arguments

Consider

```
void swap( int &i1, int &i2) {
    int temp = i1;
    i1 = i2;
    i2 = temp;
}
int c = 3;
int d = 4;
swap(c, d);
// c == 4 and d == 3
```

- `swap()` has reference arguments
- upon calling `swap()`, it is as if the compiler generated this code to initialize its arguments

```
int &i1 = c;
int &i2 = d;
```

- thus `i1` and `i2`, the variables in `swap()`'s scope, are aliases for the caller's variables.
- `swap()` behaves like Fortran functions
- C does not have reference; instead you have to write

```
extern void swap(int *i1, int *i2);
swap(&c, &d);
```

Homework

Given this declaration

```
void swap( int &i1, int *i2);
```

- write the function
- show how it is called
- draw a data model showing type and value of the arguments

Recursion

A function can call itself

```
int stirling(int n, int k) {
    if (n < k) return 0;
    if (k == 0 && n > 0) return 0;
    if (n == k) return 1;
   return k * stirling(n-1, k) + stirling(n-1, k-1);
}
```

- each block (function, if, for, *etc.*) creates new scope
- variables are declared and initialized in a scope and deleted when execution leaves scope

Exercise: write a function that computes factorial of a number

More on declarations

We have seen

```
int i;
int j = 3;
float x = 3.14;
```

A `const` declaration

```
const float e = 2.71828;
const float pi2 = 3.1415/2;
```

- a `const` variable can not be changed once it is initialized
- get compiler error if you try.

```
const float pi = 3.1415;
pi = 3.0; // act of congress
```

the following is obsolete

```
#define M_PI 3.1415;
```

- but maintained to be compatible with C
- it is C preprocesor macro (just string subsitution)

const Pointer

Consider

```
const float pi = 3.1415;
float pdq = 1.2345;
const float *p = &pi;
float* const d = &pi; // WRONG
float* const q = &pdq;
const float *const r = &pi;

*p = 3.0;  // WRONG
p = &pdq;  // OK
*p = 3.0;  // still WRONG

*q = 3.0;  // OK
q = &pdq;  // WRONG

*r = 3.0;  // WRONG
r = &pdq;  // WRONG AGAIN
```

- `const` qualifier can refer to what is pointed at (frequent usage)
- `const` qualifier can refer to pointer itself (rare usage)
- `const` qualifier can refer to both (infrequent usage)

`const` function argument

Consider

```
void f(int i, float x, const float *a) {
    i = 100;
    x = 101.0*a[0];  // OK
    a[0] = 0.0;      // WRONG!
}

int j = 1;
int k = 2;
float y[] = {3.0, 4.0, 5.0};
f(j, k, y);
```

- a `const` argument tells user of function that his data wouldn't be changed
- the `const` is enforced when attempting to compile function.
- first aspect of spirit of client/server interface

Function Name Overloading

Pre-Fortran 77 we had

```
INTEGER FUNCTION IABS(I)
INTEGER I
REAL*4 FUNCTION ABS(X)
REAL*4 X
REAL*8 FUNCTION DABS(X)
REAL*8 X
```

- separate functions had different names
- today, intrinsic functions have the same name
- programmer defined functions still must have different names

In C++, one can have

```
int    sqr(int i);
float  sqr(float x);
double sqr(double x);
```

- separate functions with same name
- functions distinguished by their name, and the number and type of arguments
- *name mangling* occurs to create the external symbol seen by the linker

Summary

Now we covered enough C/C++ **so that every thing you can do in Fortran you can now do in** C/C++

You can also do more than you can do in Fortran

Next session we introduce classes and start on the road towards object-oriented programming.