BABAR C++ Course

Paul F. Kunz

Stanford Linear Accelerator Center

Recommended text book:

- John J. Barton and Lee R. Nackman
  Scientific and Engineering **C++**
  Addison-Wesley
  IBSN:0-201-53393-6

No prior knowledge of C assumed

I'm not an expert in C++

Will try to do the dull stuff quickly, then move into OOP and OO design

Running the Examples

Access to source code examples

- use WWW browser: `http://www.slac.stanford.edu/BFROOT/doc/www/ProgC++class.html`
- copy from /nfs/juno/u5/doc/Programming/C++Class/SciEng/
- `cvs co SciEng` (for whole thing)
- `cvs co SciEng/examples/ch2` (for one chapter)

Create `a.out` executable with

- for AIX: `xlC file.C`
- for gcc: `g++ file.C -lm`
- for others: ?

Type `a.out` to run.

Code that uses the *exceptions* feature requires xlC under AIX or gcc 2.7 (when available), otherwise the current gcc should be fine

Comments

Two forms of comments allowed (`ch2/comments.C`)

- Tradition C style

```
/* This is a comment */

/*
 * This is a multiline
 * comment
 */

a = /* ugly comment */ b + c;
```

- New C++ style (also Objective-C)

```
// This is a comment

//
// This is a multiline
// comment
//

a = b + c; // comment after an expression
```

Computing group recommends C++ style

Main program

All programs must have a main

Most trivial is (`ch2/trivial.C`)

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

- `under` UNIX`, suffix is .C or .cc`
- `computing group recommends .cc`
- `main()` is a function called by the OS
- this `main()` takes no arguments
- braces ("`{`" and "`}`") denote body of function
- `main` returns 0 to the OS (success!)
- a statement ends with semi-colon ("`;`"), otherwise completely free form
- same rules as C (except `.c` suffix is used)

C++ Input and Output

Introduce I/O early, so we can run programs from shell and see something happen :-)

Example (`ch2/regurgiate.C`)

```
#include <iostream.h> // preprocessor command

int main() {
    // Read and print three floating point numbers
    float a, b, c;
    cin >> a >> b >> c;  // input
  // output
    cout << a << ", " << b << ", " << c << endl;

    return 0;
}
```

- `iostream.h` is header file containing declarations needed to use C++ I/O system
- `a`, `b`, and `c` are floating point variables (like `REAL*`4)
- `cin >>` reads from `stdin`, *i.e.* the keyboard
- `cout <<` prints to `stdout`, *i.e.* the screen
- `endl` is special variable: the end-of-line (‘`\n`’ in C) Unlike Fortran, you control the end-of-line.

More on I/O

Controlling end-of-line has its advantages

Example (`ch2/intercepts.C`**)**

```
// Print the equation coefficients of a*x + b*y + c = 0
cout << "Coefficients: " << a << ", " << b << ", " << c << endl;

// Compute and print the x-intercept.
cout << "x-intercept: ";
if (a != 0) {
    cout << -c / a << ", "; // a not equal to 0
}
else {
    cout << "none, "; // a is equal to 0
}
```

- an expression can be input to `cout <<`
- we print the result of the expression, or “none” on same line las label.

math.h

Unlike Fortran, there are no instrinsic functions

But there are standard libraries

One must include header file to make library functions available

Example (`ch2/cosang.C`)

```
// Read an angle in degrees and print its cosine.
#include <iostream.h>
#include <math.h>

int main() {

    float angle;      // Angle, in degrees
    cin >> angle;
    cout << cos(angle * M_PI / 180.0 ) << endl;
                      // M_PI is from <math.h>
    return 0;
}
```

- functions can be input to `cout <<`
- do “`man math`” to get list of functions
- useful constants are defined as well
- C shares same library

Variables, Objects, and Types

Consider (`ch2/simple.f`)

```
      INTEGER I
      REAL X
      DATA I/3/, X/10.0/
      CALL S(X, 4.2)
```

- we have three objects with initial value

I:	X:	
INTEGER 3	REAL 10.0	REAL 4.2

Consider (`simple.f`) `S()`

```
      SUBROUTINE S(A, B)
      REAL A, B
      A = B
      END
```

- we have still only three objects, but,

I:	X: A:	B:
INTEGER 3	REAL 10.0	REAL 4.2

- thus `X` gets changed by `S()` in calling routine
- we say: Fortran passes by reference

Declaring types and initializing

Consider (`ch2/simplecpp.C`**)**

```
int i = 3;
float x = 10.0;
```

- variable names must start with letter or "_" and are case sensitive
- initialization can occur on same line
- type declaration is *mandatory* (like having `IMPLICIT NONE` in every file)
- for all of the above, same rules in C
- type declaration must be before first use, but does not have to be before first executable statement

```
int i = 3;
float x = 10.0;
i = i + 1;
int j = i;
```

- general practice is to make type declaration just before first use

Types

Both Fortran and C/C++ have *types*

Fortran	C++ or C
`LOGICAL`	`bool (C++ only)`
`CHARACTER*1`	`char`
`INTEGER*2`	`short`
`INTEGER*4`	`int long`
`REAL*4`	`float`
`REAL*8`	`double`
`COMPLEX`	

- defines the meaning of bits in memory
- defines which machine instructions to use on certain operations
- implicit conversion rules between variables of different type
- `limits.h` gives your the valid range of integer types
- `float.h` gives you the valid range, precision, *etc.* of floating point types
- not all compilers support `bool` type yet
- as with Fortran, watch out on PCs or 64 bit machines

Arithmetic Operators

Both Fortran and C/C++ have *operators*

Fortran	Purpose	C or C++
`X + Y`	`add`	`x + y`
`X - Y`	`subtract`	`x - y`
`X*Y`	`multiply`	`x*y`
`X/Y`	`divide`	`x/y`
`MOD(X,Y)`	`modulus`	`x%y`
`X**Y`	`exponentiations`	`pow(x,y)`
`+X`	`unary plus`	`+x`
`-Y`	`unary minus`	`-y`
	`postincrement`	`x++`
	`preincrement`	`++x`
	`postdecrement`	`x--`
	`predecrement`	`--x`

- `x++` is equivalent to `x = x + 1`
- x++ means use it, then increment it
- `++x` means increment it, then use it.
- sorry, can't do `x**2`; use `x*x` instead
  (for sub-expressions like `(x+y)**2`, we'll see some tricks later)

Exercise

What is the output of (`ch2/prepostfix.C`)

```
#include <iostream.h>
int main() {


int i = 1;
cout << i << ", ";
cout << (++i) << ", ";
cout << i << ", ";
cout << (i++) << ", ";
cout << i << endl;


return 0;
}
```

Should be

```
1, 2, 2, 2, 3
```

Relational Operators

Both Fortran and C/C++ define relational operators

Fortran	Purpose	C or C++
`X .LT. Y`	less than	`x < y`
`X .LE. Y`	less than or equal	`x <= y`
`X .GT. Y`	greater than	`x > y`
`X .GE. Y`	greater than or equal	`x >= y`
`X .EQ. Y`	equal	`x == y`
`X .NE. Y`	not equal	`x != y`

- zero is false and non-zero is true
- Warning: a common mistake is

```
int i, j;
// some code setting i and j
if ( i = j ) {
    // some stuff
}
```

this sets `i = j` and then does some stuff if `j` is non-zero

Logical operators and Values

Both Fortran and C/C++ have logical operations and values

Fortran	Purpose	C or C++
`.FALSE.`	`false value`	`0`
`.TRUE.`	`true value`	`non-zero`
`.NOT. X`	`logical negation`	`!x`
`X .AND. Y`	`logical and`	`x && y`
`X .OR. Y`	`logical inclusive or`	`x || y`

- any expression has a logical result

```
if ( i = j(k) ) {
    // do some stuff
}
```

- `&&` and `||` evaluate from left to right and right hand expression not evaluated if it doesn't need to be
- the following never divides by zero

```
if ( d && (x/d < 10.0) ) {
    // do some stuff
}
```

- if `bool` type is supported, the `true` and `false` exists as constants.

Characters

C/C++ only has single byte characters

Constants of type `char` use single quotes

```
char a = 'a';
char aa = 'A';
```

Use *escape sequence* for unprintable characters and special cases

- `'\n'` for new line
- `'\''` for single quote
- `'\"'` for double quotes
- `'\?'` for question mark
- `'\ddd'` for octal number
- `'\xddd'` for hexadecimal

Bitwise Operators

Both Fortran and C/C++ have bitwise operators

Fortran	Purpose	C/C++
`NOT(I)`	`complement`	`~i`
`IAND(I,J)`	`and`	`i&j`
`IEOR(I,J)`	`exclusive or`	`i^j`
`IOR(I,J)`	`inclusive or`	`i|j`
`ISHFT(I,N)`	`shift left`	`i<<n`
`ISHFT(I,-N)`	`shift right`	`i>>n`

- can be used on any integer type (`char`, `short`, `int`, *etc.*)
- most often used for on-line DAQ and trigger
- also used for unpacking compressed data

Assignment operators

C/C++ has many assignment operators

Fortran	Purpose	C or C++
`X = Y`	`assignment`	`x = y`
`X = X + Y`	`add assignment`	`x += y`
`X = X - Y`	`subtract assignment`	`x -= y`
`X = X*Y`	`multiply assignment`	`x *= y`
`X = X/Y`	`divide assignment`	`x /= y`
`X = MOD(X,Y)`	`modulus assignment`	`x %= y`
`X = ISHFT(X,-N)`	`right shift assignment`	`x >>= n`
`X = ISHFT(X,N)`	`left shift assignment`	`x <<= n`
`X = IAND(X,Y)`	`and assignment`	`x &= y`
`X = IOR(X,Y)`	`or assignment`	`x |= y`
`X = IEOR(X,Y)`	`xor assignment`	`x ^= y`

- takes some time to get use to
- makes code more compact
- summing is example

```
float x=1.0., y = 2.0, z = 3.0;
float sum = x;
sum +=y;
sum +=z;
```

Operator Precedence

Both Fortran and C/C++ use precedence rules to determine order to evaluate expressions

- `z = a*x + b*y + c;` evaluates as you would expect
- can over ride default by use of parentheses
- when in doubt, use parentheses
- make code easy to understand
- don’t make clever use of precedence

`if` Statements

C/C++ `if` statement is analogous to Fortran (`ch2/tempctrl.C`)

```
if (current_temp > maximum_safe_temp) {
    cerr << "EMERGENCY: Too hot--flushing" << endl;
    flushWithWater();
}
```

Any expression that evaluates to numeric value is allowed.

```
if ( !(channel = openChannel(“temperature”)) ) {
    cerr << "Could not open channel" << endl;
    exit(1);
}
```

Braces are optional when single expression is in the block

```
if ( x < 0 )
    x = -x;  // abs(x)
    y = -y;  // always executed
```

- leaves potential for future error
- suggest single expressions remain on same line

`if else` Statements

Analogous to Fortran

```
if ( x < 0 ) {
    y = -x;
} else {
    y = x;
}
```

C/C++ also has condition operator

```
y = (x < 0) ? -x : x; // y = abs(x)
```

- use only for simple expressions
- else code can become unreadable

Also have

```
if ( x < 0 ) {
    y = -x;
} else if (x > 0) {
    y = x;
} else {
    y = 0;
}
```

Coding Styles

C/C++ is free form

Common styles for `if` block are

```
if ( x < 0 ) {
    y = -x;
} else {
    y = x;
}
// or
if ( x < 0 )
{
    y = -x;
}
else
{
    y = x;
}
```

- the computing group considers both acceptable
- the first is more common

`while` loop

C/C++ `while` is when block should be executed zero or more times

General form

```
while (expression) {
    statement
    ...
}
```

- any expression that returns numeric value
- same rules as `if` block for braces
- Fortran equivalent requires `GOTO`

```
10 IF (.NOT. expression ) GOTO 20
   statement
    ...
   GOTO 10
20 CONTINUE
```

`while` Example

Example (`ch2/sqrtTable.C`**)**

```
#include <iostream.h>
#include <iomanip.h>
#include <math.h>

int main() {
  float x;
  while (cin >> x) {
    cout << x << sqrt(x) << endl;
  }
  return 0;
}
```

- reads terminal until end-of-file
- <ctrl>-d is end-of-file for UNIX
- I can not explain how this works until later

`do-while` loop

C/C++ `do-while` is when block should be executed one or more times

General form

```
do {
    statement
    ...
} while(expression);
```

- any expression that returns numeric value
- same rules as `if` block for braces
- Fortran equivalent requires `GOTO`

```
10 CONTINUE
    statement
    ...
   IF(expression)GOTO 10
```

`do-while` Example

Snippet from use of Newton's method
(`ch2/sqrtTable.C`)

```
x = initial_guess;
do {
    dx = f(x) / fprime(x);
    x  -= dx;
} while (abs(dx) > desired_accuracy);
```

`for` loop

C/C++ `for` loop much more general than Fortran `DO` loop

```
for(init-statement; test-expr; increment-expr) {
     statement
     ...
}
```

- the test expression can be any that returns numeric value like `if` block
- function calls and I/O are also allowed

In Fortran

```
   DO 10 I = 1, J, K
    statements
    ...
10 CONTINUE
```

In C or C++

```
for( i = 1; i <= j; i += k )
     statements
     ...
}
```

More Examples

Typically, one sees

```
for(int i = 0; i < count; i++) {
    // statements in loop body
}
```

- where `i` is declared and typed in init-statement

Nested loops might iterate over all pairs with

```
for(int i = 0; i < count - 1; i++) {
    for(int j = i+1; j < count; j++) {
       // statements in loop body
    }
}
```

Use of two running indices might be

```
for(int i = 0, int j = count-1; i < count-1; i++, j--) {
   // statements in loop body
}
```

- separate expressions with commas

`break` and `continue` Statements

Consider following Fortran

```
      DO 100 I = 1, 100
         IF ( I .EQ. J ) GO TO 100
         IF ( I .GT. J ) GO TO 200
            ! do some work
100   CONTINUE
200   CONTINUE
```

- common need to break out of loop and continue to next iteration.

Equivalent C++ code is

```
for ( int i = 0; i < 100; i++ ) {
    if ( i == j ) continue;
    if ( i > j ) break;
    // do some work
}
```

- `continue` goes to next iteration of current loop
- `break` step out of current loop
- `goto` exists in C/C++ but rarely used
- we'll make less use of these constructs in C++, then in either C or Fortran

Arrays

A collection of elements of same type

```
float x[100]; // like REAL*4 X(100) in F77
```

- first element of array is `x[0]`
- last element of array is `x[99]`

Initializing array elements

```
float x[3] = {1.1, 2.2, 3.3};
float y[] = {1.1, 2.2, 3.3, 4.4};
```

- can let the compiler calculate the dimension

Multi-dimensions arrays

```
float m[4][4]; // like REAL*4 M(4,4) in F77
int m [2][3] = { {1,2,3}
                 {4,5,6} };
```

- elements appear row-wise
- Fortran elements appear column-wise
- Thus `m[0][1]` in C/C++ is `M(2,1)` in Fortran
- royal pain to interface C/C++ with Fortran

Example Code and a Test

Multiplying matrices (`ch2/mat3by3.C`)

```
float m[3][3], m1[3][3], m2[3][3];
// Code that initializes m1 and m2 ...

// m = m1 * m2
double sum;
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        sum = 0.0;
        for (int k = 0; k < 3; k++) {
            sum += m1[i][k] * m2[k][j];
        }
        m[i][j] = sum;
    }
}
```

- If you understand this code, then you know enough C/C++ to code the algorithmic part of your code
- At the beginning of this session, the above code would probably have been gibberish
- If you can not understand this code, then I'm going too fast :-(

A Pause for Reflection

What have we learned so far?

- we’ve seen how to do in C/C++ everything you can do in Fortran 77 except `function, COMMON` blocks, and character arrays.
- some aspects of C/C++ are more convenient than Fortran; some are not
- but we’ve seen nothing fundamentally new, things are just different

Next week, we start with the new stuff and we’re not even finished with chapter 2!

In particular, the replacement for `COMMON` blocks is going to be quite different

Plan of the day

Functions

Pointers

More on functions

Functions

Example function (`ch2/coulombsLaw-onefile.C`)

```
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 value
- 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

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

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 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

More on declarations

Data can be external

```
extern double pi;

double pi = 3.1415;
```

- external data is like data in Fortran `COMMON` block
- definition is like Fortran `BLOCK DATA`

Internal declaration (`ch5/expdef.C`)

```
#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
- definition substitutes for declaration within file
- still must come before use

Default Function Arguments

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

Example (`ch5/logof.h`)

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

- can be used like

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

- all arguments to the right of the first default argument must have defaults
- 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 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 avoid double inclusion, write

```
#ifndef COULOMBSLAW_H
#define COULOMBSLAW_H
extern double coulombsLaw(double q1, double q2, double r);
#endif // COULOMBSLAW_H
```

- `#ifndef` is C preprocessor directive saying “if not defined”
- `COULOMBSLAW_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
- BABAR computing group recommends use of `.hh` suffix for C++ header files

The (dreaded) Pointers

A pointer is an object that refers to another object

Declare it thus

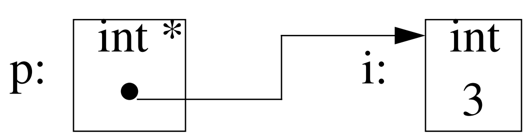
```
int* p;
int *q;
```

- either form can be used
- both mean the same thing

Assign a value to the pointer

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

- read `&` as "address of"
- thus we have



Dereferencing pointers

Consider (`ch2/ptrs.C`)

```
#include <iostream.h>

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;
}
```

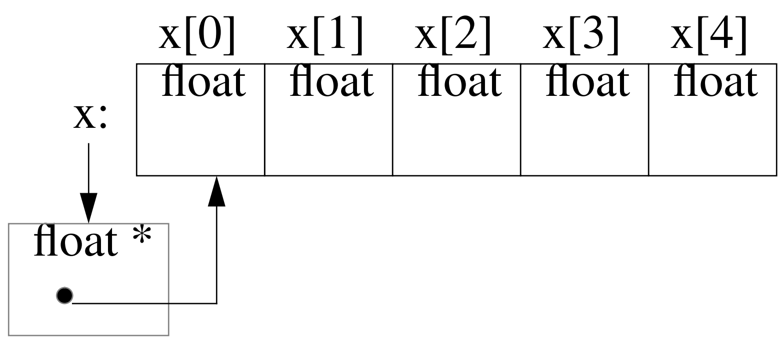
- `*p` derefences pointer to access object pointed at
- `*p` can be used on either side of assignment operator
- if `p` is equal to 0, then pointer is pointing at nothing and is called a *null* pointer.
- dereferencing a null pointer causes a core dump :-(

Pointers and Arrays

Consider

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

Our memory model is



- what does the label `x` mean?
- in Fortran, `x` is the same as `x(1)` in function call
- 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 an array 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 short hand for `*(y+1)`
- integer add, subtract and relational operators are allowed on pointers

What's wrong here?

```
float x[5];
float* y, z;
y = x; // OK
z = x; // WRONG
```

- `z` is a `float` and not a `float *`

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
- get same machine instructions from any decent compiler

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 this style
- don’t do it if you are not comfortable with it.
- compiler will generate same machine instructions either way

Examples of Pointer Arithmetic

Reverse elements of an array
(`ch2/array-reverse.C`)

```
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
(`ch2/array-zero.C`)

```
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 BABAR 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(`ch2/linefit.C`**)**

```
#include <iostream.h>

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];
     }
```

Line fit continued

Part 2 (`ch2/linefit.C`)

```
// 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' };
```

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 (ch2/scope.C)

```
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;
}
```

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

Formal Arguments

Consider(ch2/funcarg.C)

```
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

Formal Arguments

Consider(ch2/funcarg.C)

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

int j = 1;
int k[] = {2, 3, 4};
float y[] = {3.0, 4.0, 5.0};
f(j, k[0], y);
```

- in Fortran passing `k` or `k(1)` to function is the same
- in C/C++ what's the value of `k[0]` after calling `f()`?
- upon calling `f()`, the compiler generated code to initialize the arguments

```
int i = j;
float x = k[0];   // note type conversion
float *a = y; // init pointer to array
```

- thus `k[0]` does not change

References

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

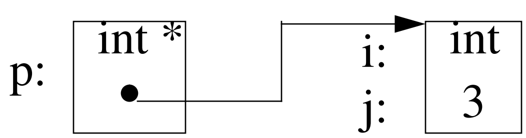
Reference (`ch/simplecpp.C`)

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

- `a` and `b` are called a *reference*
- `a`, `b`, and `x` are all the same memory location
- 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(ch2/funcarg.C)

```
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 `f()`, 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);
```

Recursion

A function can call itself (ch2/Stirling.C)

```
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
```

const Pointer

Consider

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

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

*q = 3.0;  // OOPS
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.

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    abs(int i);
float  abs(float x);
double abs(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

Inline Functions

Sometimes, the time to execute a function is small compared to time to call it

Use `inline` keyword

```
inline double sqr(double x) {
    return x * x;
}
```

- any decent compiler will generate inline code instead of function call if function is small
- but `inline` keyword is only a hint to the compiler
- inline functions are usually defined in header file
- convention puts inline functions in file with `.icc` suffix that is included by header file

Consequences of inline functions

- program might be faster
- program will be larger
- might have poor instruction cache performance and debugging might be harder

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

Classes

B&N: "Scientific and engineering problems are rarely posed directly in terms of the computer's intrinsic types: bits, bytes, integers and floating point numbers"

Shocking statement?

In BABAR tracking code, for example, the problem is posed in terms of…

- tracks
- points
- list of points
- chamber
- cylinders
- layers

C++ with its mechanism of *classes* allows defining new types and the operations on these types

When we do object-oriented programming with C++ we will be writing and using classes

Examples from CLHEP

Class Library for High Energy Physics

Why?

- Provide some classes are specific to HEP
- Encourage code sharing between experiments and between experimentalists and theorists.
- Reduce redundant work

Who?

- Leif Lönnblad, CERN (via DESY and Lund)
- Paul Rensing, CERN (via SLAC)
- Anders Nilsson (Lund), Dag Brück (LIT), and others

Use

- examples of use at `/nfs/juno/u5/doc/programming/C++class/SciEng/examples/clhep`
- `cvs co SciEng/examples/clhep`
- `g++ -I/nfs/juno/u5/src/CLHEP file.C -lm`
- `xlC -I$(BFROOT)/src/CLHEP file.C`

ThreeVector

CLHEP's ThreeVector class (simplified)

```
class Hep3VectorF {
public:
  Hep3VectorF();
  Hep3VectorF(float x, float y, float z);
  Hep3VectorF(const Hep3VectorF &v);
  float x();
  float y();
  float z();
  float phi();
  float cosTheta();
  float mag();
  // much more not shown
private:
  float fx, fy, fz;
};
```

- this is the header file
- keyword `class` starts the declaration which is contained within the `{}`
- sort of like the function prototypes
- an object of a class contains data
- an object of a class has certain functions associated with it

Using a class object

Consider (`clhep/threeVector0.C`)

```
#include <iostream.h>
#include <CLHEP/ThreeVector.h>

int main() {
  float x, y, z;

  while ( cin >> x >> y >> z ) {
    Hep3VectorF aVec(x, y, z);

    cout << "r: " << aVec.mag();
    cout << "  phi: " << aVec.phi();
    cout << "  cos(theta): " << aVec.cosTheta() << endl;
  }
  return 0;
}
```

- `Hep3VectorF aVec(x, y, z);` declares `aVec`, a variable of type `Hep3VectorF` and initializes it
- `aVec.mag()` calls the member function `mag()` of the object
- the “.” is the *class member access operator*
- use “->” access operator when one has pointer to object:

Data members

Look again

```
class Hep3VectorF {
public:
  // member functions

private:
  float fx, fy, fz;
};
```

- `Hep3VectorF` contains 3 data members
- declaration is like any other except no initializers are allowed
- every instance of the class `Hep3VectorF` will have its own 3 data members.

```
Hep3VectorF x(1.0, 0.0, 0.0);
Hep3VectorF y(0.0, 1.0, 0.0);
Hep3VectorF z(0.0, 0.0, 1.0);
```

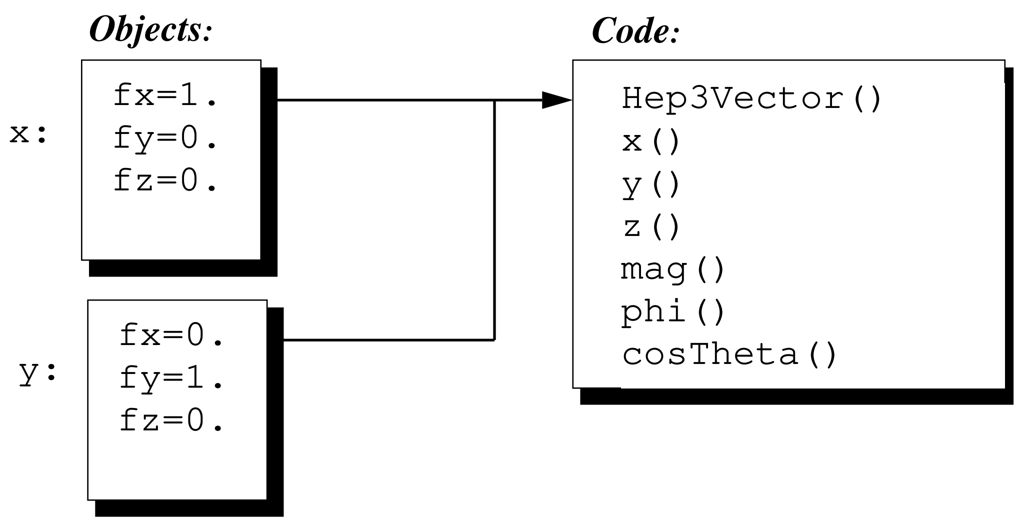
- `Hep3VectorF` is a type
- an object of type `Hep3VectorF` has a value (or state) that is represented by the values of its data members (like complex number)
- the size of a `Hep3Vector` object is likely to be `3*sizeof(float)`

Memory model

Consider

```
Hep3VectorF x(1.0, 0.0, 0.0);
Hep3VectorF y(0.0, 1.0, 0.0);
```

In computer's memory we have



- an object is an instance of a class (type)
- each object has its own data
- one copy of the code for a class is shared by all instances of the class

Use of `private` keyword

We have

```
class Hep3VectorF {
public:
  float mag();
  float x();
  float dummy;
  // member functions

private:
  float fx, fy, fz;
};
```

- the following compiles

```
Hep3VectorF x(1.0, 0.0, 0.0);
cout << x.dummy;
```

- the following does not compile

```
Hep3VectorF x(1.0, 0.0, 0.0);
cout << x.fx;  // WRONG
```

- this is called *data hiding*
- by disallowing direct access, you hide how data is stored.
- one can change how data is stored without breaking user code because you disallowed direct access

Initializing a class object

At least 3 ways we would like to initialize an object

- no initial value

```
Hep3VectorF a;
```

- with three `float` values

```
Hep3VectorF a(1.0, 1.0, 1.0);
```

- copy of another object

```
Hep3VectorF a(1.0, 1.0, 0.0);
Hep3VectorF b = a;
```

- each calls a special member function called a *constructor*

There are three constructors in the class

```
class Hep3VectorF {
public:
  Hep3VectorF();
  Hep3VectorF(float x, float y, float z);
  Hep3VectorF(const Hep3VectorF &v);
  // much more not shown
private:
  float fx, fy, fz;
};
```

Constructor Implementations

The constructor member functions

```
Hep3VectorF::Hep3VectorF(float x, float y, float z){
  fx = x;
  fy = y;
  fz = z;
}

Hep3VectorF::Hep3VectorF(const Hep3Vector &vec) {
  fx = vec.fx;
  fy = vec.fy;
  fz = vec.fz;
}

Hep3VectorF::Hep3VectorF(){
}
```

- called after memory space has been allocated
- when the class name and member name are the same, then the member function is a constructor
- `Foo::bar()` says that `bar()` is a member function of the class `Foo`
- `::` is the *scope resolution operator*

Data Hiding

Violation of private parts?

```
Hep3VectorF::Hep3VectorF(const Hep3Vector &vec) {
  fx = vec.fx;
  fy = vec.fy;
  fz = vec.fz;
}
```

- objects of the same class have access to private data members
- the purpose of data hiding is to hide implementation from other classes
- can’t hide implementation from object of same class
- `const` qualifier says we wouldn’t change argument

Access member functions

The declaration was

```
class Hep3VectorF {
public:
  float x();
  float y();
  float z();
  // much more not shown
private:
  float fx, fy, fz;
};
```

The implementation is

```
float Hep3VectorF::x() {
  return fx;
}
float Hep3VectorF::y() {
  return fy;
}
float Hep3VectorF::z() {
  return fz;
}
```

- inefficient?
- make function in-line
- always ask: “do I want the data to do some work or do I want the object to do the work”

Inline access member functions

Change declaration to

```
inline float Hep3VectorF::x() {
  return fx;
}
inline float Hep3VectorF::y() {
  return fy;
}
inline float Hep3VectorF::z() {
  return fz;
}
```

- any decent compiler should produce code that is just as efficient as direct access
- data hiding is preserved
- implementation needs to be in the header file
- frequently put in file with `.icc` suffix that is included by the header file

More Implementation

Recall

```
class Hep3VectorF {
public:
  float mag();
  float phi();
  float cosTheta();
  // much more not shown
private:
  float fx, fy, fz;
};
```

Implementation

```
inline float Hep3VectorF::mag() {
  return sqrt(fx*fx + fy*fy + fz*fz);
}

inline float Hep3VectorF::phi() {
  return fx == 0.0 && fy == 0.0 ? 0.0 : atan2(fy,fx);
}

inline float Hep3VectorF::cosTheta() {
  float ptot = mag();
  return ptot == 0.0 ? 1.0 : fz/ptot;
}
```

- note how object calls its own member function
- examples of letting object do the work

Design decisions

Fortran style

```
common/points/hits(3,100)
real*4        hits
real*4 x, y, z, r
! do some work
x = hits(1,i) ! or from ZEBRA bank
y = hits(2,i)
z = hits(3,i)
r = sqrt(x*x + y*y + z*z);
```

Another Fortran style

```
common/points/hits(3,100)
real*4        hits
real*4 x, y, z, r
! do some work
x = hits(1,i)
y = hits(2,i)
z = hits(3,i)
r = mag(x, y, z) ! or mag(hits(1,i))
```

Mark II VECSUB style

```
real*4 r
! do some work
r = hitsmag(i);
```

C++ design

C++ style

```
Hep3VectorF hits[100];
// do some work
float r = hits[i].mag();
```

- efficient with inline functions
- don’t need knowledge of data structure
- modular
- re-usable
- later, we’ll get rid of the fixed or dynamic arrays

Homework

Suppose

```
class Hep3VectorF {
public:
  Hep3VectorF();
  Hep3VectorF(float x, float y, float z)
  Hep3VectorF(const Hep3VectorF &v);
  inline float x();
  inline float y();
  inline float z();
  inline float phi();
  inline float cosTheta();
  inline float mag();
private:
  float r, cos_theta, phi;
};
```

- write the implementation for this class
- constructors take x, y, and z as arguments, but must intialize r, cos(theta), and phi data members
- try `clhep/threeVector0.C`; it should still work with this small change

```
// #include <CLHEP/ThreeVector.h>
#include "ThreeVector.h"
```

- write a program to exercise `x()`, `y()`, and `z()` member functions

Another look at `Hep3VectorF`

We'll now look at the real `Hep3VectorF` class and explain those new language elements we need to understand it

```
class Hep3VectorF {
public:
  inline Hep3VectorF(float x=0., float y=0., float z=0.);
  inline Hep3VectorF(const Hep3VectorF&);
  float x() const;
  float y() const;
  float z() const;
  float phi() const;
  float cosTheta() const;
  float mag() const;
  // much more not shown
private:
  float fx, fy, fz;
};
```

- uses default arguments
- `const` keyword after function means no data member of the object will be changed by invoking function
- this `const` is enforced when compiling the class
- the above are obvious, but it will be less obvious with other classes in the future

Initializing syntax

Initialization of built-in types has been extended to be consistent with class initialization

```
int i = 3;  // C style
int j(3);   // C++ class style
```

- the notation `int j(3);` calls the constructor for the built-in type `int`
- it is as if the constructor for built-in types sets the value given as argument
- no such constructor for built-in types exists, except deep inside the compiler

Consider

```
Hep3VectorF x = 1.0;
```

- equivalent to

```
Hep3VectorF tmp(1.0);
Hep3VectorF x = tmp;
```

Member Initializers

The constructor can be implemented like any other member function…

```
Hep3VectorF::Hep3VectorF(float x, float y, float z){
  fx = x;
  fy = y;
  fz = z;
}
```

An alternate from is use of member initializers

```
Hep3VectorF::Hep3VectorF(float x, float y, float z) :
    fx(x), fy(y), fz(z){}
```

- note the `:` preceding the opening `{`
- `fx(x)` is call to the constructor
- if `fx` were not a built-in type, then the constructor for it's class would be invoked; the *reason d'etre* for this notation
- the function body is required, even if empty
- the argument to the initializer can be anything accepted by a constructor, thus can be expressions and calls to member functions

Function Return Types

A function returns a temporary hidden variable that is initialized by the return statement

Consider

```
float f() {
    return 1;
}
float x;
// ...
x = f();
```

- it is as if

```
float tmp = 1;
x = tmp;
```

Consider

```
float & Vector3::x() {
    return fx;
}
// ...
vec.x() = 1.0; // uh?
```

- it is as if

```
float &tmp = vec.fx;
tmp = 1.0;
```

Operators are functions?

Operators can be thought of as functions

```
float add( float a, float b) {
    return a + b;
}
float x, y, z;
//
z = x + y;
z = add(x, y);
```

- `add()` operates on two arguments and returns a result
- the symbol `+` operates on two operands and returns a result

Use of mathematical symbols is more concise and easier to read

```
float add( float a, float b);
float mul( float a, float b);
float a, b, x, y, z;
//
z = add(mul(a, x), mul(b,y));
z = a*x + b*y;
```

C, C++, and Fortran all define operators for built-in types

Operator Functions

An operator function in `Hep3VectorF`

```
class Hep3VectorF {
public:
  inline Hep3VectorF& operator +=(const Hep3VectorF &);
  // more not shown
```

- the name of the function is the word `operator` followed by the operator symbol
- this function is called when

```
Hep3VectorF p, q;
//
q += p;
```

- the function is invoked on `q` ; the left-hand side
- the argument will be `p` ; the right-hand side
- `q += p;` is shorthand for `q.operator+=(p);`
- the function returns a `Hep3VectorF` reference for consistency with built-in types

Operator Function Implementation

Implementation

```
inline Hep3VectorF& Hep3VectorF::operator+=(const Hep3VectorF& p) {
  fx += p.x();
  fy += p.y();
  fz += p.z();
  return *this;
}
```

- does the accumulation as one would expect
- `this` is a keyword representing a pointer to the object’s own self
- `this->fx` is thus equivalent to `fx`
- remember: use `->` instead of `.` when you have a pointer
- or `fx` is shorthand for `this->fx`
- recall that `Hep3VectorF::x()` is an in-line function itself
- `return *this` returns the address of the object, thus the reference

Compare Fortran and C++

Fortran vector sum

```
real p(3), q(3)
! ...
q(1) = q(1) + p(1)
q(2) = q(2) + p(2)
q(3) = q(3) + p(3)
```

C++ vector sum

```
Hep3VectorF p, q;
// ...
q += p;
```

Operator Functions

Essentially all operators can be used for user defined types except `.` , `*` , `::` , `sizeof` and `?:`

Can not define new ones

- sorry, can't do `operator**()` for exponentiation

One should only use when conventional meaning makes sense

```
Hep3Vector p, q;
float z;
// .........
z = p*q; // uh?
```

- is this cross product or dot product?
- `Hep3VectorF` defines it to be dot product

Non-member Operator Function

Consider

```
inline Hep3VectorF operator*(const Hep3VectorF& p, float a) {
  Hep3VectorF q( a*p.x(), a*p.y(), a*p.z() );
  return q;
}
```

- invoked by

```
float scale = 3.0;
Hep3Vector p(1.0);  // unit vector along x axis
Hep3Vector r(0.0, 1,0);
r += p*scale;
```

- note return by value
- need a new object whose value is `x*scale`
- the temporary object is used as argument to `operator+=()` and then discarded
- such temporary objects are generated by Fortran as well

```
real scale, p(3), r(3)
r(1) = r(1) + p(1)*scale
r(2) = r(2) + p(2)*scale
r(3) = r(3) + p(3)*scale
```

Need Symmetric Operator Functions

CLHEP has

```
inline Hep3VectorF operator*(const Hep3VectorF& p, float a) {
  Hep3VectorF q( a*p.x(), a*p.y(), a*p.z() );
  return q;
}
inline Hep3VectorF operator*(float a, const Hep3VectorF& p) {
  Hep3VectorF q( a*p.x(), a*p.y(), a*p.z() );
  return q;
}
```

- second one invoked by

```
float scale = 3.0;
Hep3Vector p(1.0);  // unit vector along x axis
Hep3Vector q(0.0, 1,0);
q += scale*p;
```

- argument matching applies
- must use global function because `scale.operator*(p)` doesn't exist

The Complete List - I

Constructors

```
inline Hep3VectorF(float x=0.0, float y=0.0, float z=0.0);
inline Hep3VectorF(const Hep3VectorF &);
inline Hep3VectorF(const Hep3VectorD &);
```

- also contains conversion constructor

Destructor

```
inline ~Hep3VectorF();
```

- invoked when object is deleted (more next week)

Accessor-like functions

```
inline float x() const;
inline float y() const;
inline float z() const;
inline float mag() const;
inline float mag2() const;
inline float perp() const;
inline float perp2() const;
inline float phi() const;
inline float cosTheta() const;
inline float theta() const;
inline float angle(const Hep3VectorF &) const;
inline float angle(const Hep3VectorD &) const;
inline float perp(const Hep3VectorF &) const;
inline float perp2(const Hep3VectorF &) const;
```

The Complete List - II

Manipulators

```
void rotateX(float);
void rotateY(float);
void rotateZ(float);
void rotate(float angle, const Hep3VectorF & axis);
Hep3VectorF & operator *= (const HepRotation &);
Hep3VectorF & transform(const HepRotation &);
```

Set functions

```
protected:
  inline void setX(float);
  inline void setY(float);
  inline void setZ(float);
```

- `protected:` keyword is like private until we get into inheritance

Output function

```
ostream & operator << (ostream &, const Hep3VectorF &);
```

- allows

```
Hep3VectorF x(1.0);
// ...
cout << x << endl;
```

The Complete List - III

Vector algebra member functions

```
inline float dot(const Hep3VectorF &) const;
inline Hep3VectorF cross(const Hep3VectorF &) const;
inline Hep3VectorF unit() const;
inline Hep3VectorF operator - () const;
```

Vector algebra non-member functions

```
Hep3VectorF operator+(const Hep3VectorF&, const Hep3VectorF&);
Hep3VectorF operator-(const Hep3VectorF&, const Hep3VectorF&);
float operator * (const Hep3VectorF &, const Hep3VectorF &);
Hep3VectorF operator * (const Hep3VectorF &, float a);
Hep3VectorF operator * (float a, const Hep3VectorF &);
```

Assignment operators

```
inline Hep3VectorF & operator = (const Hep3VectorF &);
inline Hep3VectorF & operator = (const Hep3VectorD &);
inline Hep3VectorF & operator += (const Hep3VectorF &);
inline Hep3VectorF & operator += (const Hep3VectorD &);
inline Hep3VectorF & operator -= (const Hep3VectorF &);
inline Hep3VectorF & operator -= (const Hep3VectorD &);
inline Hep3VectorF & operator *= (float);
```

Summary

`Hep3VectorF` implements vector algebra

It was long and tedious to implement

Now that we have it (thank you, Leif and Anders), we can use it and never have to expand these details in our own code

Besides objects of type `int`, `float`, and `double`, we can use operators with objects of type `Hep3VectorF`

We have a new type with higher level of abstraction

Levels of Abstraction in Physics

Do you recognize these equations?

$$\sum_i \frac{\partial E_i}{\partial x_i} = \frac{\partial E_x}{\partial x} + \frac{\partial E_y}{\partial y} + \frac{\partial E_z}{\partial z} = 4\pi\rho$$

$$\sum_i \frac{\partial B_i}{\partial x_i} = \frac{\partial B_x}{\partial x} + \frac{\partial B_y}{\partial y} + \frac{\partial B_z}{\partial z} = 0$$

$$\sum_i \varepsilon_{ijk} \frac{\partial}{\partial x_j} E^k = -\frac{1}{c}\frac{\partial B_i}{\partial t}$$

$$\frac{\partial E_z}{\partial y} - \frac{\partial E_y}{\partial z} = -\frac{1}{c}\frac{\partial B_x}{\partial t}$$

$$\frac{\partial E_x}{\partial \mathrm{z}} - \frac{\partial E_z}{\partial x} = -\frac{1}{c}\frac{\partial B_y}{\partial t}$$

$$\frac{\partial E_y}{\partial x} - \frac{\partial E_x}{\partial y} = -\frac{1}{c}\frac{\partial B_z}{\partial t}$$

Higher Level of Abstraction

Now do you recognize them?

$$\vec{\nabla} \bullet \mathbf{E} = 4\pi\rho$$

$$\vec{\nabla} \times \mathbf{B} = \frac{4\pi}{c}\mathbf{J} + \frac{1}{c}\frac{\partial \mathbf{E}}{\partial t}$$

$$\vec{\nabla} \bullet \mathbf{B} = 0$$

$$\vec{\nabla} \times \mathbf{E} = -\frac{1}{c}\frac{\partial \mathbf{B}}{\partial t}$$

or even

$$\partial_\alpha F^{\alpha\beta} = \frac{4\pi}{c} J^\beta$$

$$\frac{1}{2}\varepsilon^{\alpha\beta\gamma\delta}\partial_\alpha F_{\gamma\delta} = 0 = \partial^\alpha F^{\beta\gamma} + \partial^\beta F^{\gamma\alpha} + \partial^\gamma F^{\alpha\beta}$$

To advance in physics/math, we need higher levels of abstractions, else we get lost in implementation details

C++ allows higher level of abstract as well

Plan of the day

Where are we at?

- session 1: basic language constructs
- session 2: pointers and functions
- session 3: basic class and operator overloading

Today

- design of two types of container classes
- templates
- friend
- nested classes

`SimpleFloatArray` Class

Design and implement an array class with

- run time sizing
- access to element with `x[i]`
- automatic memory management
- automatic copy of array elements
- automatic copy upon assignment
- set all elements of array to a value
- find the current size
- dynamic resizing

Each requirement leads to a member function

There will be some technical issues to learn

Why an array class?

Replace these parts of linefit.C

```
    cin >> n;
    float* x = new float[n];
// munch munch
    sx += x[i];
    delete [] x;
```

with

```
    cin >> n;
    SimpleFloatArray x(n);
// munch munch
    sx += x[i];
//    delete [] x;
```

- to avoid pointers
- to get automatic deletion
- to show how to be able to do

```
SimpleFloatArray x(n);
SimpleFloatArray y = x;
SimpleFloatArray z;
//
z = x    // copy array
x = 0.0; // clears the array
```

SimpleFloatArray Class Definition

The header file (`ch4/SimpleFloatArray.h`)

```
class SimpleFloatArray {
public:
  SimpleFloatArray(int n);           // init to size n
  SimpleFloatArray();                // init to size 0
  SimpleFloatArray(const SimpleFloatArray&); // copy
  ~SimpleFloatArray();                    // destroy
  float& operator[](int i);               // subscript
  int numElts();
  SimpleFloatArray& operator=(const SimpleFloatArray&);
  SimpleFloatArray& operator=(float);     // set values
  void setSize(int n);
private:
  int num_elts;
  float* ptr_to_data;
  void copy(const SimpleFloatArray& a);
};
```

- ~SimpleFloatArray() is the *destructor* member function and is invoked when object is deleted
- `float& operator[](int i)` is the member function invoked when the operator `[]` is used
- `operator=()` is member function invoked when doing assignment: the *copy* assignment
- note private member function

Constructor Implementations

Constructors (`ch4/SimpleFloatarray.C`**)**

```
SimpleFloatArray::SimpleFloatArray(int n) {
    num_elts = n;
    ptr_to_data = new float[n];
}

SimpleFloatArray::SimpleFloatArray() {
    num_elts = 0;
    ptr_to_data = 0; // set pointer to null
}

SimpleFloatArray::SimpleFloatArray(const SimpleFloatArray& a) {
    num_elts = a.num_elts;
    ptr_to_data = new float[num_elts];
    copy(a); // Copy a's elements
}
```

- by implementing the default constructor, we ensure that every instance is in well defined state before it can be used
- must implement copy constructor else the default behavior is member-wise copy which would lead to two array objects sharing the same data

copy Implementation

Terse implementation (ch4/SimpleFloatArray.C)

```
void SimpleFloatArray::copy(const SimpleFloatArray& a) {
  // Copy a's elements into the elements of our array
  float* p = ptr_to_data + num_elts;
  float* q = a.ptr_to_data + num_elts;
  while (p > ptr_to_data) *--p = *--q;
}
```

- uses pointer arithmetic
- uses prefix operators

Fortran style implementation

```
void SimpleFloatArray::copy(const SimpleFloatArray& a) {
  // Copy a's elements into the elements of *this
  for (int i = 0; i < num_elts; i++ ) {
    ptr_to_data[i] = a.ptr_to_data[i];
  }
}
```

- uses array notation on pointer
- uses postfix operator

Destructor Member Function

Implementation (`ch4/SimpleFloatArray.C`**)**

```
SimpleFloatArray::~SimpleFloatArray() {
    delete [] ptr_to_data;
}
```

- one and only one destructor
- function with same name as class with ~ prepended
- no arguments, no return type
- invoked automatically when object goes out of scope
- invoked automatically when object is deleted
- usually responsible for cleaning up any dynamically allocated memory

`operator[]` Member Function

Implementation

```
float& SimpleFloatArray::operator[](int i) {
    return ptr_to_data[i];
}
```

- overloads what `[]` means for object of this type
- returns *reference* to element in array
- since it is a reference, it can be used on right-hand or left-hand side of assignment operator
- this snippet of code will work (`ch4/linefit.C`)

```
int n;
cin >> n;
SimpleFloatArray x(n);
SimpleFloatArray y(n);

for (int i = 0; i < n; i++) {
    cin >> x[i] >> y[i];
}
double sx  = 0.0, sy  = 0.0;
for (i = 0; i < n; i++) {
    sx += x[i];
    sy += y[i];
}
```

- remember, a reference is not a pointer

`operator=` Member Function

Implementation

```
SimpleFloatArray&
SimpleFloatArray::operator=(const SimpleFloatArray& rhs) {
    if ( ptr_to_data != rhs.ptr_to_data ) {
        setSize( rhs.num_elts );
        copy(rhs);
    }
    return *this;
}
```

- `if()` statements tests that array object is not being assigned to itself.
- `this` is a pointer to the object with which the member function was called.
- `this->ptr_to_data` and `ptr_to_data` are equivalent
- must implement else default is member-wise copy leading to two objects sharing the same data

Assignment versus Copy

Copy Constructor

```
SimpleFloatArray::SimpleFloatArray(const SimpleFloatArray& a) {
    num_elts = a.num_elts;
    ptr_to_data = new float[num_elts];
    copy(a); // Copy a's elements
}
```

Assignment operator

```
SimpleFloatArray&
SimpleFloatArray::operator=(const SimpleFloatArray& rhs) {
    if ( ptr_to_data != rhs.ptr_to_data ) {
        setSize( rhs.num_elts );
        copy(rhs);
    }
    return *this;
}
```

Use

```
SimpleFloatArray x(n);
SimpleFloatArray y = x;    // copy constructor
//
z = x    // copy array     // assignment
```

- should not implement one without the other

Scaler assignment

Implementation

```
SimpleFloatArray& SimpleFloatArray::operator=(float rhs) {
    float* p = ptr_to_data + num_elts;
    while (p > ptr_to_data) *--p = rhs;
    return *this;
}
```

- set all elements of array to a value
- invoked by

```
SimpleFloatArray a(10);
a = 0.0; // assignment
```

- not

```
SimpleFloatArray a(10) = 0.0;
```

which depends on matching and conversion rules explained in chapter 5

The remaining implementation

Implementation

```
int SimpleFloatArray::numElts() {
    return num_elts;
}

void SimpleFloatArray::setSize(int n) {
    if (n != num_elts) {
         delete [] ptr_to_data;
         num_elts = n;
         ptr_to_data = new float[n];
    }
}
```

- nothing special here.
- can't resize (no `realloc()`)
- could save old data with re-write of class

Key points

- should supply destructor function so object can delete memory it allocated before it gets deleted itself
- must supply copy constructor and `operator=()` if member-wise copy is not what we want
- should return reference in case where object could be on left hand side of assignment

Class explosion?

Suppose we want `SimpleIntArray`?

Could copy `SimpleFloatArray`, edit everywhere we find `float` and save to create new class

- tedious work
- duplicate code
- we'll want to the same for `double`, `Hep3VectorF`, *etc*.

Could use `void *` instead of `float` and then cast return values.

- only C programmers know what I'm talking about
- bad idea because we lose type safety

If we have `n` data types and `m` things to work with them, we don't want to have to write `n x m` classes

Enter *template* feature of C++ to solve this problem

`SimpleArray` Template Class

Class declaration (`ch4/SimpleArray.h`)

```
template<class T>
class SimpleArray {
public:

    SimpleArray(int n);
    SimpleArray();
    SimpleArray(const SimpleArray<T>&);
    ~SimpleArray();
    T& operator[](int i);
    int numElts();
    SimpleArray<T>& operator=(const SimpleArray<T>&);
    SimpleArray<T>& operator=(T);
    void setSize(int n);
private:
    int num_elts;
    T* ptr_to_data;
    void copy(const SimpleArray<T>& a);
};
```

- `template<>` says what follows is a template for producing a class
- `<class T>` is the template argument
- `T` is arbitrary symbol for some type, either built-in or programmer defined (not necessarily a class)
- line breaking is a style issue

Use of Class Template

Line fit with template class (`ch4/linefit2.C`)

```
void linefit() {

    int n;
    cin >> n;
    SimpleArray<float> x(n);
    SimpleArray<float> y(n);

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

- `SimpleArray<float>` is now a class
- `float` replaced `class T`
- use a template class like any other class
- any type can be used

```
SimpleArray<Hep3VectorF> x(n);
```

Function Templates

Remember (`ch2/doubleSqr.C`)

```
inline double sqr(double x) {
    return x * x;
}
```

Templated version (`SciEng/utils.h`)

```
template<class T>
inline
T sqr(T x) {
    return x * x;
}
```

Now we can do

```
int i = 1;
float f = 3.1;
Hep3VectorF v(1, 1, 1);

cout << sqr(i) << endl;
cout << sqr(f) << endl;
cout << sqr(v) << endl;
```

- using the templated function generates one of the correct type
- without the template function, implicit conversion would happen (details in chapter 5)

List or Array?

`SimpleArray` is fixed in size once created or re-assigned

What we really want is a `List`

- add incrementally objects to a list
- remove objects from a list
- list should resize itself automatically
- provide a means to iterate through the list
- find member of a list
- insert an object at particular point in the list
- sort a list
-
-
-
-

Use of a `List`

Normalizing some numbers to minimum value (`ch6/demoList.C`)

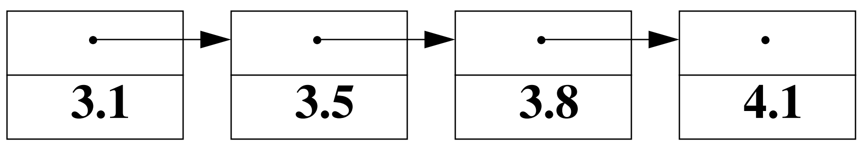
```
int main() {
    // Read list of values and find minimum.
    List<float> list;
    float val;
    float minval = FLT_MAX; // from <float.h>
    while ( cin >> val) {
        if (val < minval) minval = val;
        list.add(val);
    }

    // Normalize values and write out.
    for (ListIterator<float> i(list); i.more(); i.advance()) {
        cout << i.current() - minval << endl;
    }
    return 0;
}
```

- reads until end of file
- finds minimum value
- adds to list
- iterate through list to normalize

Linked List

A popular data structure



Advantages of a list compared to an array

- fast to re-size
- fast to insert

Disadvantage of a list compared to an array

- more memory per element
- slow to random access

The `Node` and `List` classes

Declaration and implementation

```
template <class T>
class Node {
private:
    Node(T x) : link(0), datum(x) {}
  // perhpas more not shown
    Node* link;
    T     datum;
};
```

- uses initializers

Declaration and implementation

```
template<class T>
class List {
public:
    List() : first(0), last(0) {}
    void add(T x) {
        if (first == 0) first = last = new Node(x);
        else last = last->link = new Node(x);
    }
private:
    Node* first;
    Node* last;
};
```

- data members point to first and last nodes in order to quickly add a node to end of list

Problems

Some design issues

- If `Node` class will only be used by `List`, then should it take such a simple name?
- If we always use ListIterator to access data, then do we have to provide three accessor functions?

The answers makes use of two new features:

- nested classes
- `friend` declaration

`List` with nested `node` class

Declaration and implementation (`ch6/List.h`)

```
template<class T>
class List {
public:
    List() : first(0), last(0) {}
    void add(T x) {
        if (first == 0) first = last = new Node(x);
        else last = last->link = new Node(x);
    }
    friend class ListIterator<T>;
private:
    class Node {
    public:
        Node(T x) : link(0), datum(x) {}
        Node* link;
        T     datum;
    };
    Node* first;
    Node* last;
};
```

- not only nested, but private as well
- `Node` as a class name is not visible outside of `List`
- did not have to repeat `template` keyword
- `friend` keyword allows access of private data members to `ListIterator<T>` class

ListIterator class

Declaration and Implementation (`ch6/List.h`)

```
template<class T>
class ListIterator {
public:
    ListIterator(const List<T>& list) : cur(list.first) {}

    Boolean more()    const { return cur != 0;    }
    T       current() const { return cur->datum; }
    void    advance()       { cur = cur->link;    }

private:
    List<T>::Node* cur;
};
```

- violation of private parts?
- In `List` we had

```
    friend class ListIterator<T>;
```

- `List<T>::Node*` scoping is needed because `Node` as a class name is not visible even to a friend
- note that `List` was easier to implement than `SimpleArray`
- `bool` is now a type in C++, but not when the book was written

Iterators

Compare

```
SimpleArray<float> a(n);
// ..
for (int i = 0; i < n; i++) {
    sum += a[i];
}
```

with

```
List<float> list;
// ..
for (ListIterator<float> i(list); i.more(); i.advance()) {
    sum += i.current();
}
```

- `i` is the iterator in both cases
- both initialize `i` to first element
- both use `i` to test for completion
- both increment `i` to next element
- both use `i` to reference element
- the `ListIterator` version is more tolerant to changes
- one could write an iterator for `SimpleArray`

Iterators++

Compare

```
SimpleArray<float> a(n);
// ..
for (int i = 0; i < n; i++) {
    sum += a[i];
}
```

with

```
List<float> list;
// ..
for (ListIterator<float> i(list); i != 0; i++) {
    sum += i.current();
}
```

- implement `operator++()`
- will be the style of STL

Use of Containers

Chamber containing layers

```
class Chamber {
public:
//
private:
    Array<Layer> layers(60);
// ...
}
```

Event containing tracks and clusters

```
class BBEvent {
public:
//
private:
    List<DCTracks> tracks;
    List<CalCluster> clusters;
// ...
}
```

A track keeping its space points

```
class Track {
public:
//
private:
    List<Hep3VectorF *> hits;
// ...
}
```

CLHEP containers

CLHEP offers us

- `HepAList`: array based implementation of list like object
- `HepVectorF`: vector of n dimension
- `HepMatrixF`: general n by n matrix
- `SymMatrixF`: symmetric matrix
- `DiagMatrixF`: diagonal matrix
- double precision version of the above
- all depend on inheritance which we don't know yet

Standard Template Library (STL)

Features

- various types of templated containers
- very much iterator based
- supplies functions that can work with any kind of containers
- very well designed

Status

- contributed by HP labs, Palo Alto
- part of the draft standard since July 1994
- under UNIX, compiles only with xlC
- modified but buggy version works with gcc 2.6.3
- we’ll probably use it
- we’ll defer discussion until we can
- books on it are being written

Plan of the day

Inheritance is last major feature of the language that we need to learn

- used to expressed common implementation
- used to expressed common behavior
- used to expressed common structure

Will divert from the text book in order to introduce HEP specific classes

- Examples from CLHEP
- Examples from Gismo

Recall ThreeVector

CLHEP's ThreeVector class (simplified)

```
class Hep3VectorF {
public:
  Hep3VectorF();
  Hep3VectorF(float x, float y, float z);
  Hep3VectorF(const Hep3VectorF &v);
  inline float x();
  inline float y();
  inline float z();
  inline float phi();
  inline float cosTheta();
  inline float mag();
  // much more not shown
private:
  float fx, fy, fz;
};
```

and some of the implementation

```
inline float Hep3VectorF::x() {
  return fx;
}
inline float Hep3VectorF::mag() {
  return sqrt(fx*fx + fy*fy + fz*fz);
}
```

Recall our test program

The object does the work (`clhep/threeVector0.C`)

```
#include <iostream.h>
#include <CLHEP/ThreeVector.h>

int main() {
  float x, y, z;

  while ( cin >> x >> y >> z ) {
    Hep3VectorF aVec(x, y, z);

    cout << "r: " << aVec.mag();
    cout << "  phi: " << aVec.phi();
    cout << "  cos(theta): " << aVec.cosTheta() << endl;
  }
  return 0;
}
```

including algebraic operators

```
Hep3Vector p, q, r;
float z;
// .........
z = p*q;
r = x + y;
```

Possible 4-Vector Class

Might look like…

```
class HepLorentzVectorF {
public:
  HepLorentzVectorF();
  HepLorentzVectorF(float x, float y, float z, float t);
  HepLorentzVectorF(const HepLorentzVectorF &v);
  inline float x();
  inline float y();
  inline float z();
  inline float t();
  inline float phi();
  inline float cosTheta();
  inline float mag();
  // much more not shown
private:
  float fx, fy, fz, ft;
};
```

Compare with 3-Vector class

- some member functions must be exactly the same
- some member functions are added
- some member functions must be re-implemented
- some data is the same
- one new data item

Another Possible 4-Vector Class

Might look like…

```
class HepLorentzVectorF {
public:
  HepLorentzVectorF();
  HepLorentzVectorF(float x, float y, float z, float t);
  HepLorentzVectorF(const HepLorentzVectorF &v);
  inline float x();
  inline float y();
  inline float z();
  inline float t();
  inline float mag();
  // much more not shown
private:
  Hep3VectorF vec3;
  float ft;
};
```

- `HepLorentzVectorF` *has-a* `Hep3VectorF`
- let 3-vector component do part of the work

```
float HepLorentzVectorF::mag() {
    return sqrt(ft*ft - vec3.mag2() );
}
```

- must still implement functions like

```
float HepLorentzVectorF::x() {
    return vec.x();
}
```

Inheritance

Part of the header file (`CLHEP/LorentzVector.h`)

```
class HepLorentzVectorF : public Hep3VectorF {
public:
  HepLorentzVectorF();
  HepLorentzVectorF(float x, float y, float z, float t);
  HepLorentzVectorF(const HepLorentzVectorF &v);
  HepLorentzVectorF(const Hep3VectorF &p, float t);
  float t();
  float mag();
  // much more not shown
private:
  float ft;
};
```

- `HepLorentzVectorF` *is-a* `Hep3VectorF`
- All public members of `Hep3VectorF` are also public members of `HepLorentzVectorF` by use of keyword `public` in class declaration.
- member function `t()` is added
- member function `mag()` overrides function of same name in `Hep3VectorF`
- constructors take different arguments
- one new data member: `ft`

Use of Lorentz Vector

Consider (`clhep/fourVector0.h`)

```
int main() {
  float x, y, z, t;
  while ( cin >> x >> y >> z >> t ) {
    Hep3VectorF a3Vec(x, y, z);
    HepLorentzVectorF a4Vec(x, y, z, t);

    cout << "3-vector x and mag: "
         << a3Vec.x() << " " << a3Vec.mag() << endl
    cout << "4-vector x and mag: "
         << a4Vec.x() << " " << a4Vec.mag() << endl
  }
  return 0;
}
```

- `HepLorentzVector` behaves like any other class
- how does `a4Vect.x()` work since no member function has been defined?… by inheritance
- `a4Vec.mag()`, however, is completely different from `a3Vect.mag()`
- output of program

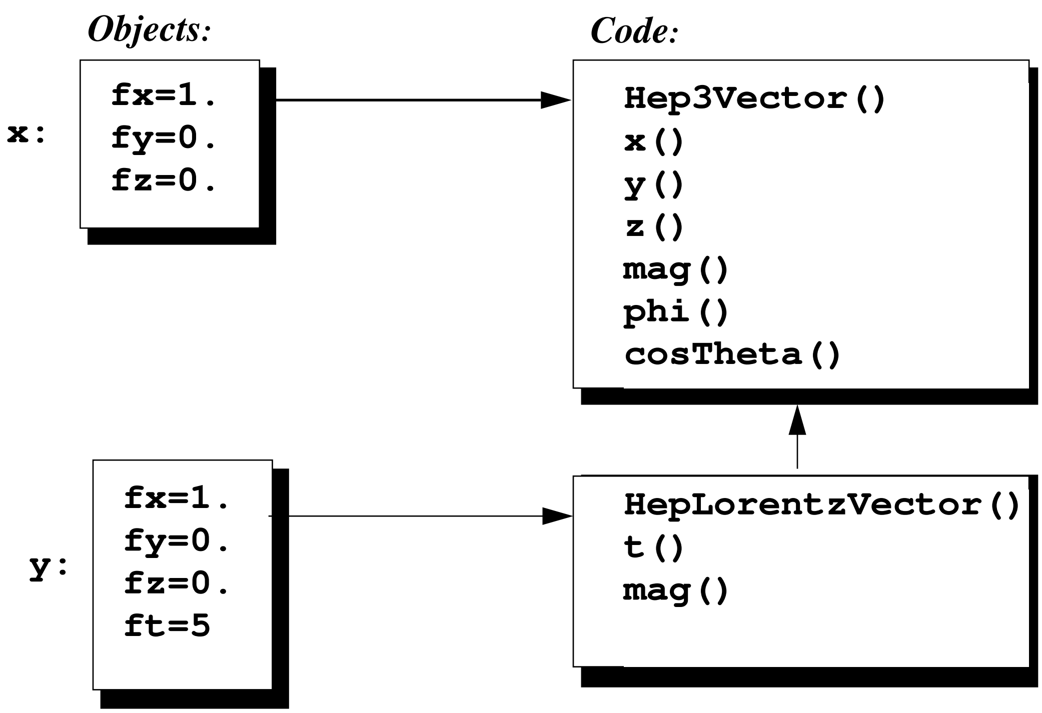
```
hpkaon> a.out
1 1 1 2
3-vector x and mag: 1 1.73205
4-vector x and mag: 1 1
```

Memory model

Consider

```
Hep3VectorF x(1.0, 0.0, 0.0);
HepLorentzVectorF y(1,0, 0,0, 0,0, 5.0);
```

In computer's memory we have



- inheritance of data members
- inheritance of member functions

Look at Implementation

As you might expect

```
inline float HepLorentzVectorF::t() const {
  return ft;
}
```

- the `t()` member function is like we've seen before

This doesn't work

```
inline float HepLorentzVectorF::mag2() const {
  return ft*ft - (fx*fx + fy*fy + fz*fz));
}
```

- `fx`, `fy`, and `fz` were declared `private`
- `private` means access to objects of the same class and `HepLorentzVector` is a different class
- could modify `Hep3VectorF` to

```
class Hep3VectorF {
public:
// same as before
protected:
float x, y, z
}
```

- `protected:` means access to members of the same class and all subclasses

More on Implementation

Keep the base class data members private

```
inline float HepLorentzVectorF::mag2() const {
  return ft*ft - Hep3VectorF::mag2();
}
```

- use scope operator `::` to access function of same name in super class
- now we can re-write `Hep3VectorF` to use `r`, `costheta` and `phi` without needing to re-write `HepLorentzVectorF`

Finally, we have

```
inline float HepLorentzVectorF::mag() const {
  float pp = mag2();
  return pp >= 0.0 ? sqrt(pp) : -sqrt(-pp);
}
```

- did you remember that 4-vector can have negative magnitude?

Even more of Implementation

The dot product

```
inline float
HepLorentzVectorF::dot(const HepLorentzVectorF & p) const {
  return ft*p.t() - z()*p.z() - y()*p.y() - x()*p.x();
}
```

- use of accessor functions `x()`, `y()`, and `z()` because data members are private in the super class
- scope operator `::` not needed because these functions are unique to the base class

The `+=` operator

```
inline HepLorentzVectorF &
HepLorentzVectorF::operator += (const HepLorentzVectorF& p) {
  Hep3VectorF::operator += (p);
  ft += p.t();
  return *this;
}
```

- example of directly calling operator function

Many other functions will not be shown

They implement the vector algebra for Lorentz vectors

What's new?

A Lorentz boost function

```
void HepLorentzVectorF::boost(double bx, double by, double bz){
  double b2 = bx*bx + by*by + bz*bz;
  register double gamma = 1.0 / sqrt(1.0 - b2);
  register double bp = bx*x() + by*y() + bz*z();
  register double gamma2 = b2 > 0 ? (gamma - 1.0)/b2 : 0.0;

  setX(x() + gamma2*bp*bx + gamma*bx*ft);
  setY(y() + gamma2*bp*by + gamma*by*ft);
  setZ(z() + gamma2*bp*bz + gamma*bz*ft);
  ft = gamma*(ft + bp);
}
```

- `register` keyword advises compiler that variable should be optimized in machine registers

Also have

```
inline Hep3VectorD HepLorentzVectorD::boostVector() const {
  Hep3VectorD p(x()/dt, y()/dt, z()/dt);
  return p;
}
inline void HepLorentzVectorF::boost(const Hep3VectorD & p){
  boost(p.x(), p.y(), p.z());
}
```

- double precision version only

A Road Sign

Warning: prototype code is about to follow

- experimental work by Paul Rensing before he left SLAC
- see BABAR Note 169 for a description
- work is left unfinished
- there are some problems that need fixing
- needs to be updated to latest CLHEP release
- there are some ideas for a better design
- pre-dates formation of collaboration

Opportunity for some people who want to do some useful work while learning

It is nevertheless useful to look at this code as is

Particle class

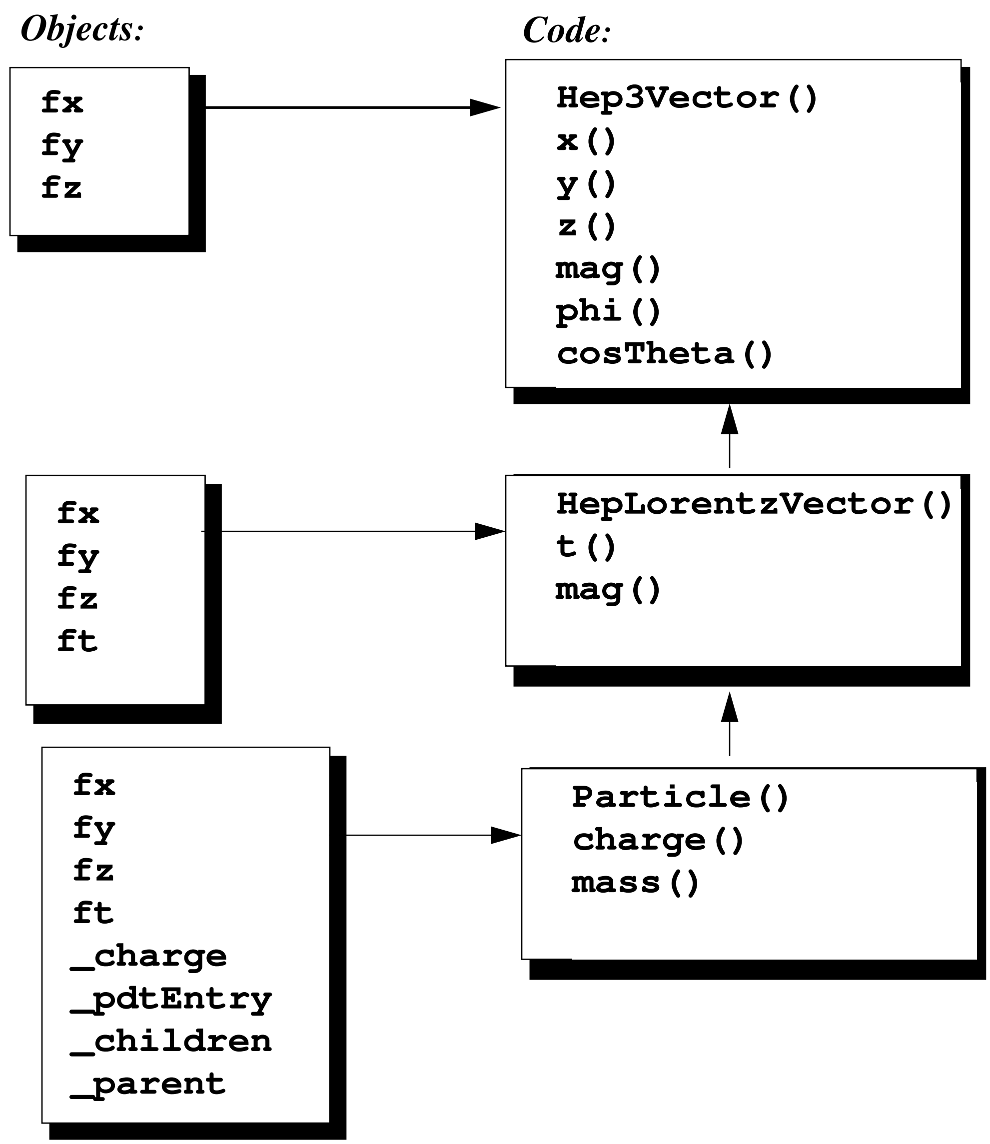
Take Lorentz vector and add to it (`bfast/particle.h`)

```
class Particle : public HepLorentzVector
{
public:
     Particle() {
        _pdtEntry = 0; _charge = 0; _parent = 0; }
     Particle(HepLorentzVector &, PDTEntry *);
     Particle(const Particle &);
     virtual ~Particle() {}
     float charge() const { return _charge; }
     float mass() const { return mag(); }
  // more methods not shown
protected:
     float _charge;  // units of e
     PDTEntry *_pdtEntry;
     HepAList<Particle> _children;
     Particle *_parent;
};
```

- note one can inherit from a class which is an inherited class
- added features are charge, pointer to entry in particle data table, list of children, and pointer to parent
- member functions not shown support added features

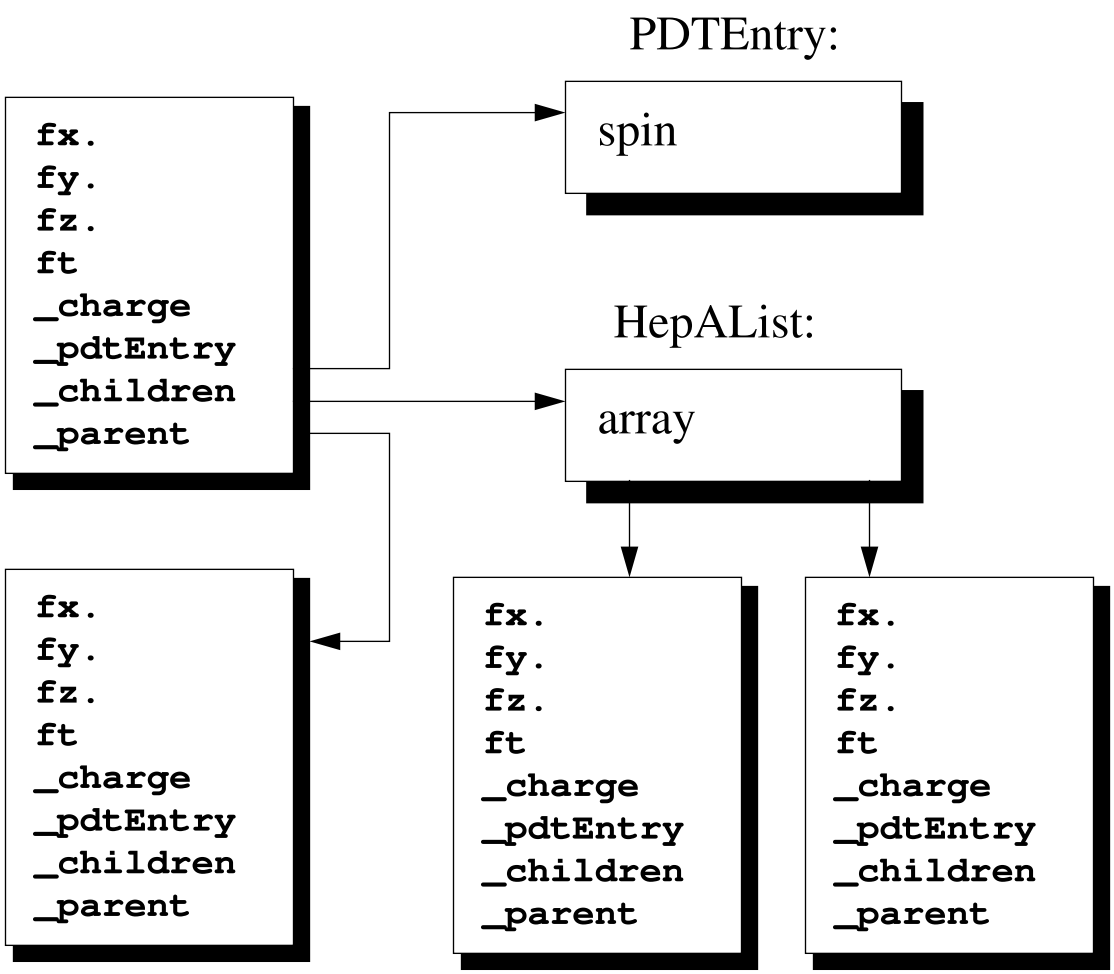
Class Hierarchy

In computer’s memory we have



Object Hierarchy

In computer memory we have



- the class and object hierarchies are different in dimensions

The 3 hierarchies of OOP

- Class hierarchy
- Object hierarchy
- hierarchy of levels of abstraction, *e.g.* float, vector matrix

Multiple Inheritance

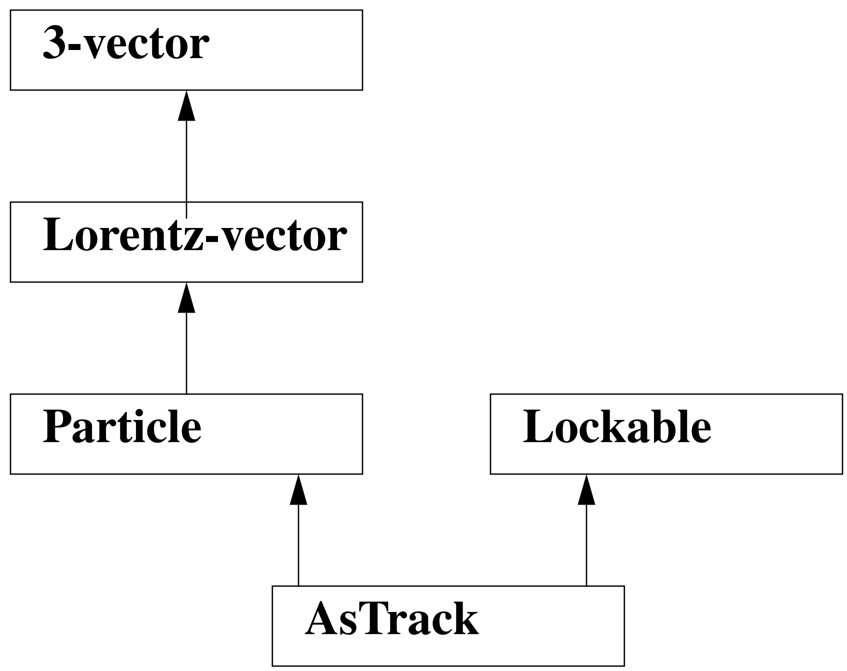
One can inherit from more than one class (`aslund/AsTrack.h`)

```
class AsTrack : public HepLockable, public Particle
{
public:
     AsTrack();
     AsTrack(AsEvent *e, int type, int index);
     AsTrack(const AsTrack &);
     virtual ~AsTrack();
  // more member functions not shown
}
```

- `AsTrack` inherits from both `Particle` and `HepLockable`
- both data members and member functions are inherited from both classes

Class hierarchy

For both data members and functions we have



- `AsTrack` has the functions defined in itself and all of its super classes
- `AsTrack` has data members defined in itself and all of its super classes

`Lockable` class

The header file

```
class HepLockable {
public:
  inline HepLockable();
  inline void lock(int bit);
  inline void unlock(int bit);
  inline int locked() const;
  // more not shown
}
```

- gives property of “in-use” flag to sub-class
- can use when something can only be used once
- will demonstrate use with a combintoric problem
- `HepLockable` is part of CLHEP

`AsTrack`'s constructor

Beginning of constructor (`aslund/AsTrack.cc`)

```
AsTrack::AsTrack(AsEvent *e, int type, int index)
   : Lockable(), Particle()
{
     _type = type;
     _index = index;
     int ftype = type + 1;
     int find = index + 1;
     float p[20];
     trkallc(&ftype, &find, p);

     setX(p[0]);
     setY(p[1]);
     setZ(p[2]);
     setT(p[3]);
     _charge = p[10];
  // more not shown
```

- note calling the constructors of the super classes
- careful: the super class constructors are called in order of the class definition, not necessarily in the order listed in the constructor.
- `trkallc` is a Fortran subroutine that fetches data out of ASLUND's COMMON blocks

Extended Example

Reconstruct *B^0*, *i.e.*

$B^0 \to D^* D^*$

$D^* \to D^0 \pi$

$D^0 \to K^{\pm} \pi^{\mp} \pi^0$

$D^0 \to K^{\pm} \pi^{\mp}$

Procedure

- start with measured tracks of unknown type
- identify K, π, and π^0 candidates.
- try all combinations of $K\pi$ then $K\pi\pi^0$ and make list of D^0s.
  NOTE: ***don't*** use the same measured track twice
- try all combinations of D^0 and π and make list of D^*
  NOTE: ***don't*** use the same measured track twice
- try all combinations of D^*

We'll look at some prototype code to do this analysis

The classes involved

From CLHEP library

- Lorentz Vector
- `HepAList`
- `Lockable`
- `Chooser`: chooses un-locked entities
- `Combiner`: a subclass of Chooser that also adds entities together

From AsLund library

- `AsLund`: the event generator interface to the Fortran
- `AsEvent`: the interface to parts of the event
- `AsTrack` a track
- `AsAnalysis`: base class for an analysis

User supplied

- `main.cc`: the main program
- `dStarAnal`: a subclass of `AsAnalysis`

main.cc

The main program in C++ (`user/main.cc`) (simplified)

```
int main( int argc, char **argv )
{
    dStarAnal work;
    Aslund aslund;
    aslund.initialize();
 // initialize the C++ PDT from the LUND common blocks
    initPDTfromLUND();
    work.initialize();

    AsEvent *e;
    int rc = 0;
    while (rc == 0 && (e = aslund.next())!=0 ) {
        rc = work.analyse(e);
    }
  // final output
    aslund.finish();
    work.finish();
    exit(0);
}
```

- not in official BABAR framework because it was written before the collaboration existed.

dStarAnal::initialize()

The initialization (`user/dStarAnal.cc`) (simplified)

```
void dStarAnal::initialize()
{
    AsAnalysis::initialize();
  // some stuff deleted

    pi0Mass = PDT::mass( PDT::pi0 );
    d0Mass = PDT::mass( PDT::D0 );
    dStarMass = PDT::mass( PDT::Dstarp );
    B0Mass = PDT::mass( PDT::B0 );

     // find the boost vector for the CM system
    float emass = PDT::mass(PDT::e);
    HepLorentzVector e(0.0, 0.0,
                 sqrt(3.1*3.1 - emass*emass), 3.1);
    HepLorentzVector p(0.0, 0.0,
                -sqrt(9.022*9.022 - emass*emass), 9.022);
    HepLorentzVector cm = e + p;
    CMBoost = cm.boostVector();
}
```

- `PDT::pi0` is something we'll have to cover next week
- ditto for `PDT::mass()`

dStarAnal::analyse()

Member function `analyse` of class `dStarAnal` is passed pointer to `AsEvent` object

```
int dStarAnal::analyse( AsEvent *evt )
{
   AsTrack *t; // tmp varible
```

Form list of charged tracks and neutral tracks (*i.e.* cluster in calorimeter)

```
HepAList<AsTrack> chgTracks = evt->posTracks();
chgTracks.append(evt->negTracks());
HepAList<AsTrack> neutTracks = evt->neutTracks();
```

- `chgTracks` is an object of type `HepAList` containing objects of type `AsTrack`
- `AsEvent` objects can return to us various objects which describe the event.

Do particle type identification

Fake the particle with `mcPID object`

```
HepAList<AsTrack> kaons
   = mcPID.doPID(chgTracks, PDT::Kp);
HepAList<AsTrack> pions
   = mcPID.doPID(chgTracks, PDT::pip);
```

- we fake the particle id code with MC truth
- of course, same measured track could be put into both lists
- `PDT` is particle data table object with all properties of all known particles

Identify π^0

Create a list of π^0 by combining neutral (gamma) and cutting on mass

```
HepAList<AsTrack> piZeros;
Combiner<AsTrack> pi0Cand(&neutTracks, &neutTracks);
while ( t = pi0Cand.next() ) {
   if ( fabs(t->mass() - pi0Mass) > 0.025 ) continue;
   piZeros.append(new AsTrack(*t));
}
```

- `pi0Cand` is an object of type `Combiner` which combines objects of type `AsTrack`
- `pi0Cand` is initialized with the same list twice
- `Combiner` when given same list twice, just does obvious nested `for` loops
- `Combiner` *adds* a track from the first list to one in second and returns a new track that is the sum.
- `AsTrack` implements `operator+=` by adding 4-momentum and summing the charge
- for each pi0 we want to save, we create a new `AsTrack` which is a copy of the one made by the `Combiner`

Find D^0s

Find all the $D^0 \rightarrow K\pi$ candidates

```
HepAList<AsTrack> D0s;
Combiner<AsTrack> D0comb1(&kaons, &pions);
while ( t = D0comb1.next() ) {
   if (t->charge() != 0.0) continue;
   if (fabs(t->mass() - d0Mass) > 0.2) continue;
   D0s.append(new AsTrack(*t));
}
```

- `D0comb1` is initialized with two different lists
- `D0comb1.next()` does
  picks next track from `kaon` list
  locks the track
  picks next unlocked track from `pion` list
  create and return pointer to new track which is sum
- thus a measured track is never used twice in one combination
- note we must make copy of track

Find more D^0 and D*

Find D^0->$K\pi\pi^0$

```
Combiner<AsTrack> D0comb2(&kaons, &pions, &piZeros);
while ( t = D0comb2.next() ) {
   if (t->charge() != 0.0) continue;
   if (fabs(t->mass() - d0Mass) > 0.2) continue;
   D0s.append(new AsTrack(*t));
}
```

Find D^*

```
HepAList<AsTrack> Dstars;
Combiner<AsTrack> DsComb(&D0s, &pions);
while ( t = DsComb.next() ) {
   if (fabs(t->mass() - dStarMass) > 0.2) continue;
   Dstars.append(new AsTrack(*t));
}
```

- `Combiner` allows variable number of arguments
- $D^0 \rightarrow K\pi$ and $D^0 \rightarrow K\pi\pi^0$ go into same `D0s` list
- Locking a track from `D0s` list, will lock all its children
- note that the while loops are no more complex then the previous ones

Finally, find all B^0 –> D^* D^*

Use the `Combiner` again to find the B^0s

```
Combiner<AsTrack> B0Comb(&Dstars, &Dstars);
int B0Count = 0;
while ( t = B0Comb.next() ) {
   if (t->charge() != 0) continue;
   if (fabs(t->mass() - B0Mass) > 0.2) continue;
// find B’s momentum in CM frame
   HepLorentzVector b(*t);
   b.boost(CMBoost);
```

- `b` will be the Lorentz vector component of our B^0
- `CMBoost` was calculated in our initialization step
- obviously, can do a lot more things at this stage

Summary

We now know enough C++ to do a physics analysis

Next week we'll look at polymorphic uses of inheritance with examples from Gismo

Then, we'll be pretty much done with learning the language

Are you about to be thrown out of the nest?

Not quite, but…

It's soon time to start some mini-projects using C++

- fix/re-design/extend the analysis classes shown today?
- re-write trackerr?
- tracking code?
- code for prototype tests?
- ?
- ?

Plan of the day

One language feature from chapter 2

Particle data table

Polymorphic inheritance

Enumerations

mnemonic names for integer codes grouped into sets

```
enum Color { red, orange, yellow, green, blue, indigo, violet };

Color c = green;

enum Polygon { triangle = 3, quadrilateral, pentagon };
```

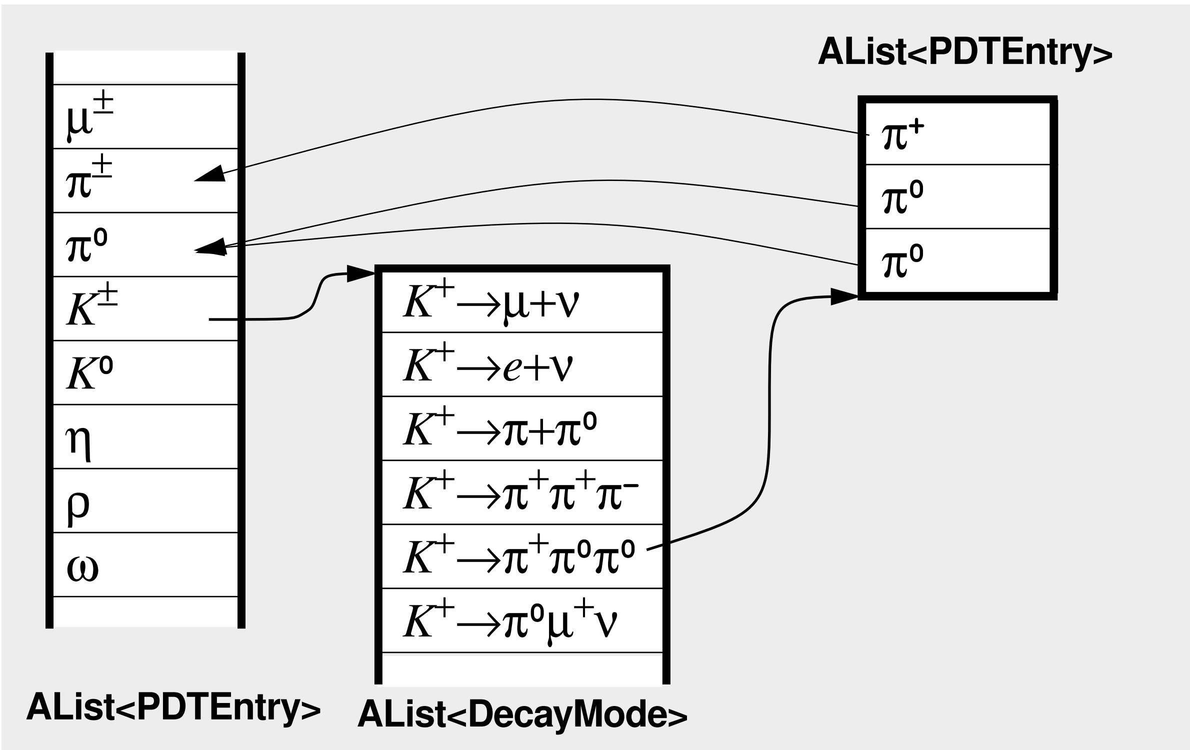
- `Color` is programmer defined type
- `red`, `orange`, *etc* are constants of type `Color`
- `c` is declared as type `color` with inital value of `green`
- `c` can change, but `red`, `orange` *etc* can not
- `enum` values are converted to int when used in arithmetic or logical operations
- default integer values start at 0 and increment by 1
- can override the default.
- but valued stored in variable which is an enumerated type is limited to the values of the `enum`
- uniqueness of the enumerated values is guaranteed
- slightly different from C

Particle Data Table class

History

- written by Paul Rensing at SLAC
- inspired from `ParticleList` class in the MC++ project (a particle generator framework, see Lönnblad, *et al, Comp. Phys. Commun.*, **71**, 1, 1992)
- not part of CLHEP
- found in `bfast` directory

Layout



- `PDT` has one data member: `HepAList<PDTEntry> _entries`
- `PDTEntry` has data members for particle properties and an `AList<DecayMode>` for list of decay modes
- `DecayMode` has data members for branching fraction and an `AList<PDTEntry>` for list of children.
- `AList` entries are pointers or references, not copies

The PDT class

Part of the header file (`bfast/PDT.h`)

```
class PDT
{
public:
// a list of common particles
// the numbers are PDG standard particle codes
  enum IDType {
    e=11, nu_e, mu, nu_mu, tau, nu_tau,
    photon=22, gamma=22, Z_0=23, W,

    pi0=111, rho0=113, a2_0=115,
// many more not shown
  };
// more not shown
};
```

- use an `enum` to give names to standard (PDG) particle codes
- use scope operator to access values from outside the class, *e.g.* `PDT::photon`
- last week we had…

```
HepAList<AsTrack> kaons
   = mcPID.doPID(chgTracks, PDT::Kp);
HepAList<AsTrack> pions
   = mcPID.doPID(chgTracks, PDT::pip);
```

`static` keyword

More of the header

```
class PDT
{
public:
  enum IDType {
// contents not shown not shown
  };
  static float mass(long id);
  static float mass(const char* name);
  static float width(long id);
  static float width(const char* name);
private:
  static HepAList<PDTEntry> _entries;
};
```

- a `static` data member is one that is shared by all instances of the class, *e.g.* a global within the scope of the class
- a `static` member function is one that is global within the scope of the class
- access a data member or member function with scope operator

```
     pi0Mass = PDT::mass( PDT::pi0 );
     d0Mass = PDT::mass( PDT::D0 );
```

- an `enum` behaves like static data members

`PDTEntry` class

Parts of the header file (`bfast/PDTEntry.h`)

```
class PDTEntry {
public:
  const char *name() const {return _name;}
  float  charge() const {return _charge;}
  int idCode() const {return _idCode;}
  float mass() const {return _mass;}
  float width() const {return _width;}
// more not shown
protected:
  char *_name;
  long  _idCode; //Particle Data Group code number
  float _mass;     // nominal mass (GeV)
  float _width;    // width (0 if stable) (GeV)
  float _lifeTime; // c*tau, (cm)
  float _spin;     // spin, in units of hbar
  float _charge;   // charge, in units of e
  float _widthCut; // used to limit range of B-W
  float _sumBR;    // total branching ratio
  HepAList<DecayMode> _decayList;
};
```

- nothing new

DecayMode class

From the header file (`bfast/DecayMode.h`)

```
class DecayMode {
public:
  DecayMode(float bf, HepAList<PDTEntry> *l ) {
    _branchingFraction = bf;
    _children = l;
   }
  float BF() const { return _branchingFraction; }
  const HepAList<PDTEntry> *childList() const {
     return _children; }
protected:
  float  _branchingFraction;
  HepAList<PDTEntry> *_children;
};
```

- again, nothing new

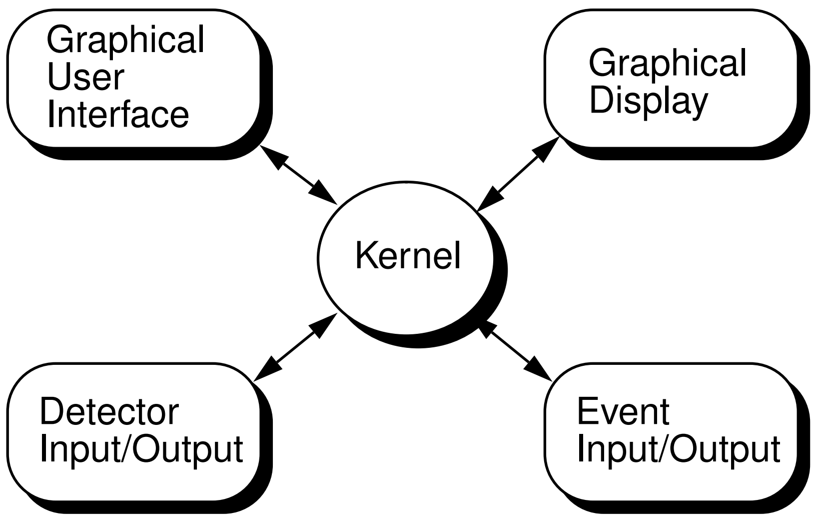
Detector Simulation

What classes are involved?

- 3-vector
- geometry
- track
- detectors
- fields

Gismo

C++ framework for detector simulation and reconstruction



- kernel written as a C++ class library
- kernel is machine independeant
- graphical parts machine dependent (NeXTSTEP and [someday] X-Windows versions)
- input/output left open

Gismo History

Version 0, the prototype

- written by Bill Atwood (SLAC) and Toby Burnett (U Washington)
- used Objective-C language
- completed in Spring 1991

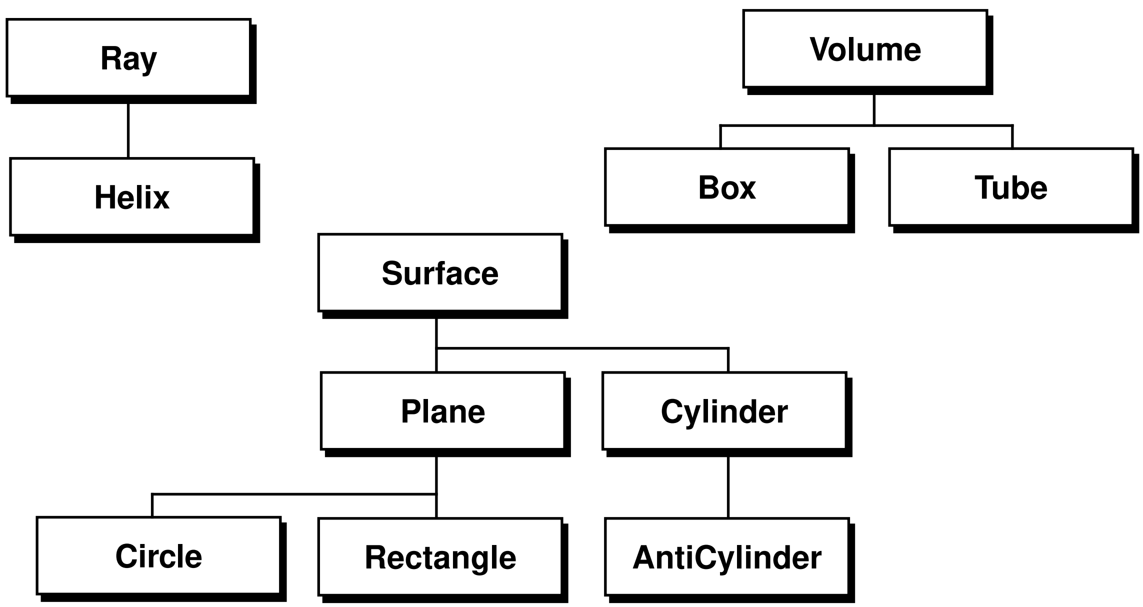
Version 1, current release

- written by Atwood, Burnett, Alan Breakstone (Hawaii), Dave Britton (McGill) and others
- used C++ but without templates and without CLHEP
- first release was summer 1992
- `ftp://ftp.slac.stanford.edu/pub/sources/gismo-0.5.0.tar.Z`
- will show code based on this version

Version 2, under development

- written by Atwood and Burnett
- C++ with templates and CLHEP

Some Gismo Classes



- other Gismo classes are not shown
- we see several independent class hierarchies
- objects from these hierarchies will work together

Let’s browse some of the classes

`Ray` class

Part of the header

```
class Surface;
class Ray
{
public:
  Ray();
  Ray( const ThreeVec& p, const ThreeVec& d );
  virtual ~Ray() {};
  Ray( const Ray& r );
  virtual ThreeVec position( double s ) const
    { return pos + s * dir; }
  const ThreeVec& position() const {return pos;}
  virtual double curvature() const   { return 0.0; }
  virtual double
  distanceToLeaveSurface( const Surface* s, ThreeVec& p ) const;
// more not shown
protected:
  ThreeVec pos;
  ThreeVec dir;
  float arclength;
}
```

- note forward class declaration
- note that a ray is clearly a straight line
- we have some virtual functions whose signifance will be explained shortly

`Helix` class

Part of the header

```
class Helix : public Ray
{
public:
  Helix();
  Helix( const ThreeVec& p, const ThreeVec& d,
         const ThreeVec& a, double r );
  virtual ~Helix() {};
  Helix( const Helix& r );
  virtual ThreeVec position( double step ) const;
  const ThreeVec& position()const
    {return Ray::position();}
  double curvature() const { return 1.0 / rho; }
  virtual double
  distanceToLeaveSurface( const Surface* s, ThreeVec& p ) const;
protected:
  ThreeVec axis;  // helix axis direction (unit vector)
  double rho;     // helix radius, sign significant
  ThreeVec perp;  // perpendicular direction
  double parallel;// component along axis
}
```

- `position` member function is not inline
- we have some more virtual functions

Surface class

Part of the header

```
class Surface
{
protected:
    ThreeVec origin; // origin of Surface
public:
    Surface() : origin() {}
    Surface( const ThreeVec& o ) : origin( o ) {}
    virtual ~Surface() {}
    Surface( const Surface& s ) {
        origin = s.origin; }
    virtual double distanceAlongRay(
        int which_way, const Ray* ry, ThreeVec& p ) const = 0;
    virtual double distanceAlongHelix(
        int which_way, const Helix* hx, ThreeVec& p ) const = 0;
// more not shown
}
```

- we see usually constructors and destructors
- the `distanceAlong` member functions are pure virtual
- an instance of `Surface` can not be instanciated
- `Surface` exists to define an interface

`Plane` class

Part of header

```
class Plane: public Surface
{
public:
  Plane( const Point& origin, const Vector& n );
  Plane( const Point& origin, const Vector& nhat,
         double dist );
// more not shown
private:
   double d;
   // offset from origin to surface

};
```

- `Plane` is infinite since it has no data members to describe boundary

`Circle` class

Part of header

```
class Circle: public Plane
{
public:
  Circle() : Plane() { radius = 1.0; }
  Circle( const ThreeVec& o,
          const ThreeVec& n, double r );
  virtual ~Circle() {}
  Circle( const Circle& c );
  virtual int withinBoundary( const ThreeVec& x )
const;
// more not shown
protected:
  double radius;
}
```

- has data member to describe boundary
- also has member function to give the answer

`Rectangle` class

Part of the header

```
class Rectangle: public Plane
{
public:
  Rectangle();
  Rectangle( const ThreeVec& o, const ThreeVec& n,
             double l, double w, const ThreeVec& la);
  virtual ~Rectangle() {}
  Rectangle( const Rectangle& r );
  virtual int withinBoundary( const ThreeVec& x ) const;
protected:
  double length, width;
  ThreeVec length_axis;
}
```

- data members to describe boundary
- member function to test for boundary
- data member to describe direction

Gismo Volume

Part of the header

```
class Volume
{
// a lot not shown
  virtual double distanceToLeave( const Ray& r,
           ThreeVec& p, const Surface*& s ) const;
protected: // make available to derived classes
  HepAList<Surface> surface_list;
  ThreeVec center; // center of Volume
  double roll, pitch, yaw;
}
```

- `Volume` is a base class with common functionality of all volumes
- it contains a list of surfaces that describe the volume
- it contains a 3-vector for its center and 3 floats for its rotation
- member functions not shown allow one to build abitrary volumes, move them, and rotate it.
- for tracking, key member function is `distanceToLeave`

Part of implementation

The key member function

```
double Volume::distanceToLeave( const Ray& r,
             ThreeVec& p, const Surface *&sf ) const
{
  double d = 0.0, t = FLT_MAX;
  ThreeVec temp ( t, t, t );
  p = temp;
  sf = 0;
  Surface *s = surface_list->first();
  while ( s ) {
    d = r.distanceToLeaveSurface( s, temp );
    if ( ( t > d ) && ( d >= 0.0 ) ) {
      t = d;
      p = temp;
      sf = s;
    }
    s = surface_list->next();
  }
  return t;
}
```

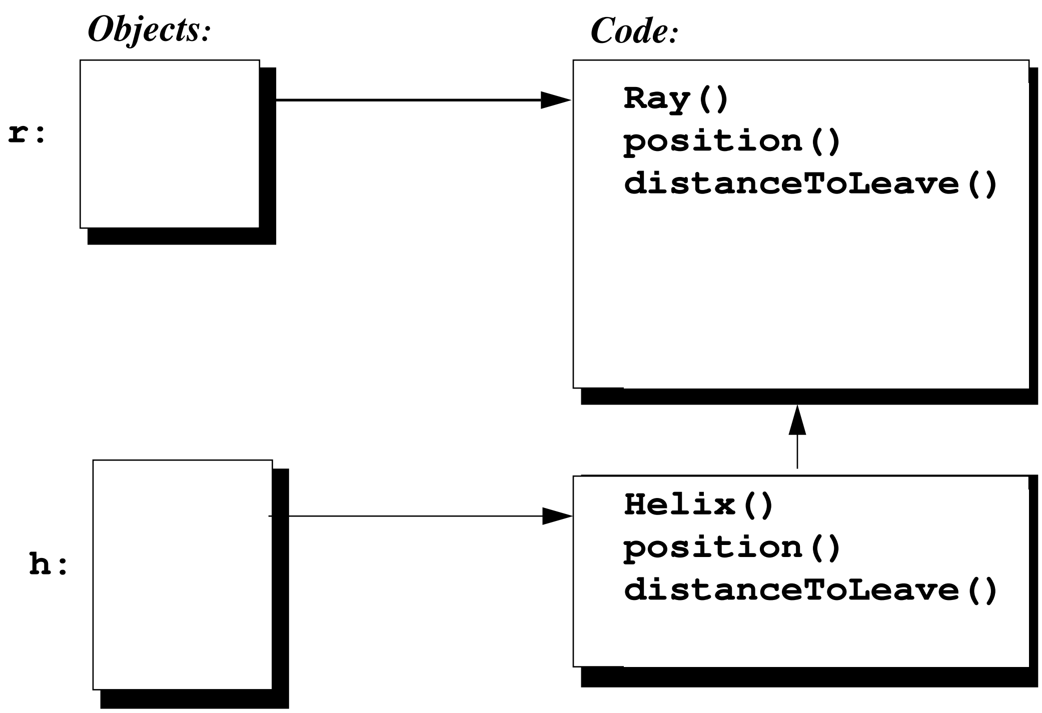
- loop over all surfaces to find the shortest distance
- the `Ray` object appears to do the work
- we don't know if the `Ray` object is-a `Ray` or its `Helix` subclass

Recall Memory model

Consider

```
Ray r;
Helix h;
```

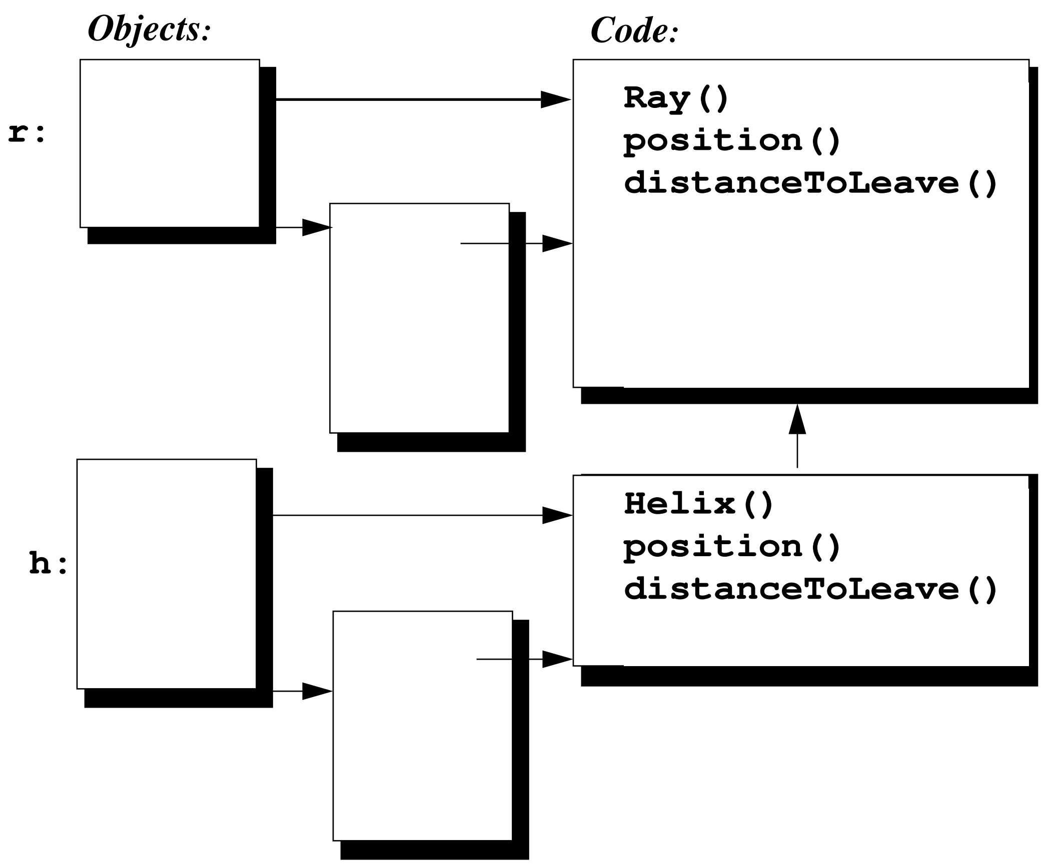
In computer's memory we have



- but now, we want `Volume` to invoke `Helix::distanceToLeaveSurface` if

The virtual function table

Memory model with virtual functions



- virtual member functions are invoked indirectly via the virtual funcation table
- the table contains pointers to the member function
- each class initializes the table with its functions

Back to implementation

We have

```
double Volume::distanceToLeave( const Ray& r,
             ThreeVec& p, const Surface *&sf ) const
{
  double d = 0.0, t = FLT_MAX;
  ThreeVec temp ( t, t, t );
  p = temp;
  sf = 0;
  Surface *s = surface_list->first();
  while ( s ) {
    d = r.distanceToLeaveSurface( s, temp );
    if ( ( t > d ) && ( d >= 0.0 ) ) {
      t = d;
      p = temp;
      sf = s;
    }
    s = surface_list->next();
  }
  return t;
}
```

- compiler creates different machines instructions to invoke a virtual member function
- `distanceToLeaveSurface` was declared `virtual` so correct function gets called

Following the trail

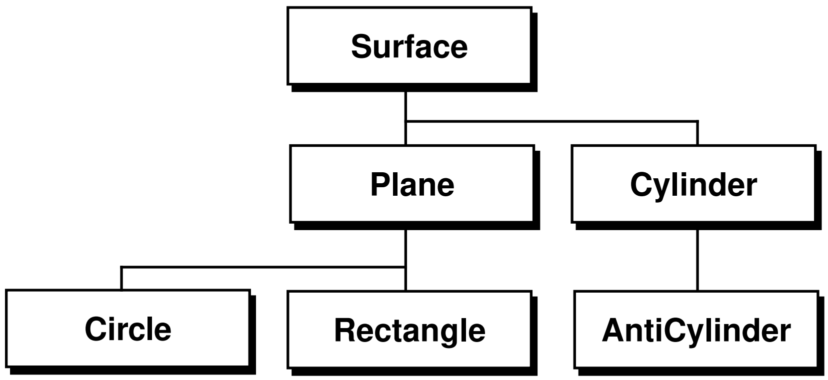
In `Ray` and `Helix` we have

```
double Ray::distanceToLeaveSurface
            ( const Surface* s, ThreeVec& p ) const
{
    return s->distanceAlongRay(  1, this, p );
}
//
double Helix::distanceToLeaveSurface
            ( const Surface* s, ThreeVec& p ) const
{
    return s->distanceAlongHelix(  1, this, p );
}
```

- so `Surface` will do the work
- this design pattern is called the Vistor pattern or the Double-Dispatch pattern
- via the `Ray` or `Helix`, we invoke the correct member function of `Surface` subclass
- recall that these functions were pure virtual in `Surface`

Where's the implementation?

Where will we find `distanceAlongRay`?



- its not in `Surface`
- one implementation in `Plane`
- but we really instanciate objects of type `Circle` or `Rectangle`
- another in `Cylinder`

Implementation

In `Plane`, we have

```
double Plane::distanceAlongRay( int which_way,
       const Ray* ry, ThreeVec& p ) const
{
  double dist = FLT_MAX;
  ThreeVec lv ( FLT_MAX, FLT_MAX, FLT_MAX );
  p = lv;
//  Origin and direction unit vector of Ray.
  ThreeVec x = ry->position();
  ThreeVec dhat = ry->direction( 0.0 );
//  Normal unit vector to Plane.
  ThreeVec nhat = normal();
  double denom = nhat * dhat;
  if ( ( denom * which_way ) <= 0.0 )
    return dist;  // return large distance
  double d = ( ( ( getOrigin() - x ) * nhat ) / denom );
  if ( ( d >= 0.0 ) && ( d < FLT_MAX ) ) {
    dist = d;
    p = ry->position( d );
    if ( withinBoundary( p ) == 0 ) {
      dist = FLT_MAX;
      p = ThreeVec( FLT_MAX, FLT_MAX, FLT_MAX );
    }
  }
  return dist;
}
```

- `withinBoundary()` member function must be in `Circle` or `Rectangle`

As expected

In `Circle` we have

```
int Circle::withinBoundary( const ThreeVec& x ) const
{
  ThreeVec p = x - origin;
  if ( p.magnitude() <= radius )
    return 1;
  else
    return 0;
}
```

In `Rectangle` we have

```
int Rectangle::withinBoundary( const ThreeVec& x ) const
{
  ThreeVec p = x - origin;
  ThreeVec width_axis = norm.cross( length_axis );
  if ( ( fabs( p * length_axis ) <= ( 0.5 * length ) )  &&
       ( fabs( p * width_axis  ) <= ( 0.5 * width  ) ) )
    return 1;
  else
    return 0;
}
```

Virtual destructor

In Volume, we may have

```
Volume::~Volume()
{
  Surface *s = surface_list->first();
  while ( s ) {
    delete s;
    s = surface_list->next();
  }
  delete surface_list;
}
```

- we need to call the destructor for `Circle`, `Plane`, *etc*
- thus we make the destructor virtual for this heirarchy

Summary

Inheritance used for

- used to expressed common implementation
- used to expressed common behavior
- used to expressed common structure

Virtual inheritance allows objects to use abstract base functions with concrete classes

We’re done for a while

Plan of the day

C++ Review

Project discussion

Beyond Language

What's really new is new ways to design our code

- it is hard to teach someone to be a good designer
- it takes experience to become a good designer

A good design is…

- easier to work with
- easier to maintain
- resistant to required changes

Let's review some of the different designs we've seen

Where we've been

The six sessions (LBL/LLNL version)

1) Introduction to basics for C/C++

- different but nothing fundamentally new

2) Functions, pointers and dynamic allocation

- tighter control on calling functions
- pointers allow for dynamically allocated arrays

3) Classes with operator overloading

- group and hide the three elements of a 3-Vector
- letting the object do the work
- adding new type to the language
- code to higher level of abstraction

Where we’ve been

4) Arrays and lists

- design details we must do right
- list versus arrays
- use of iterators
- design with template classes

5) Simple inheritance and use for physics analysis

- re-use existing class by inheriting from it
- multiple inheritance and mix-in classes
- pushing the combinorics into a special class

6) Polymorphic inheritance and use in simulation

- abstract base classes
- polymorphic derived classes
- set of cooperating classes

My Observations

It's hard to get the design right the first time

- design a little
- code a little
- review design and re-design
- iterate

What to look for

- minimize data flow
- look for abstractions
- think about what might need to change

This course hasn't prepared you to immediately do good design

But one has to get started

My Partial List of Projects

Starting with AsLund++

- re-design analysis classes
- augment analysis classes with fitting
- upgrade them to latest release of CLHEP

Simulations and reconstructions

- re-write TRACKERR
- tracking code
- test fitting codes
- resurrect McGrath's Gismo code
- do small simulation with Gismo

Miscellaneous

- convert stand-a-lone codes to C++
- something in the on-line world
- evaluate STL
- write a game?

Object Mentors

Experienced people willing to help

- Dick Blankenbecler (track finding)
- Bob Jacobson (to be confirmed)
- Paul Kunz
- Matt Langston (SLD student)
- Mark McDougald
- Tom Pavel (SLD student)