BABAR C++ Course

Paul F. Kunz

Stanford Linear Accelerator Center

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

You need to practice to really learn C++

First two sessions is about the same for C, C++, Objective-C and Java

Preliminaries

Recommended text book:

- John J. Barton and Lee R. Nackman
  Scientific and Engineering C++
  Addison-Wesley
  IBSN: 0-201-53393-6
- http://www.research.ibm.com/xw-SoftwareTechnology-books-SciEng-AboutSciEng.html

Access to source code examples

- use **WWW** browser to text book home page
- copy from /usr/local/doc/C++Class/SciEng/

Create `a.out` executable with

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

Type `a.out` to run.

Some code requires *exceptions* feature

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
```


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`
- under Windows, suffix is `.cpp`
- `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 intrinsic functions

But there are standard libraries

One must include header file to make library functions available at compile time

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 <<`
- see `/usr/include/math.h` 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: [INTEGER 3] X: [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: [INTEGER 3] X: A: [REAL 10.0] B: [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 a letter or "_", and are case sensitive
- initialization can occur on same line
- multiple declarations are allowed
- 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 generate on certain operations
- `limits.h` gives you 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 current value, 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
```

Try changing `++` to `--`

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

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`

- `&&` 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.
- else can do

```
typedef char bool;
bool false = 0; bool true = 1;
```

Characters

C/C++ only has one 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
- `'\xdd'` 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.*)
- right shift might not do sign extension
- 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

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
- also left to right or right to left precedence defined
- 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);
}
```


`if` gotchas

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 ( x < 0 ) x = -x;  // abs(x)
```

Any expression, including assignment

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

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

`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 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/Newton.C`)

```
x = initial_guess;
do {
    dx = f(x) / fprime(x);
    x  -= dx;
} while (fabs(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(i = 0; i < count - 1; i++) {
    for(j = i+1; j < count; j++) {
       // statements in loop body
    }
}
```

Use of two running indices might be

```
for(i = 0, 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 or continue to next iteration.

Equivalent C++ **code is**

```
for (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
```

- access first element of array with `x[0]`
- access last element of array with `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 functions, `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 session, we start with some 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 object
- second token is function name
- argument names are proceeded by their type
- function body is within `{}`
- return statement can be expression or variable
- if keyword `void` is used as return type, then function is like Fortran `SUBROUTINE`
- if no arguments, `void` can be used or leave empty

Function Prototypes

Will this work?

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

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

Will this work?

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

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

Declarations and Definitions

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

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

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

```
extern double sqrt(double);

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

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

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

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

Header files used with definition

In `math.h`, we have declarations

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

In `math.C`, we have definition

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

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

Header files and user code

In `math.h`, we have declarations

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

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

```
#include <math.h>

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

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

Extern Data Declarations

Data can be external

```
extern double aNum;

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

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

Defining extern data

```
double aNum = 1234.5678;

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

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

Static Functions

Static function declaration (`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 of file
- definition substitutes for declaration within file
- still must come before use

Static Data

Consider

```
#include <iostream.h>

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

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

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

Default Function Arguments

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

Example (`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

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

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

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

Functions in C

Function declaration and prototype is the same in C except

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

Header Files

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

Use C preprocessor to avoid to double inclusion

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

- `cpp buils tempoary file for compiler`
- `#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

The (dreaded) Pointers

A pointer is an object that refers to another object

Declare it thus

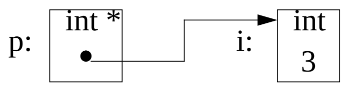
```
int* p;
int *q;
```

- either form can be used; the later is prefered

Assign a value to the pointer

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

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



Watch out!

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


Dereferencing pointers

Consider (`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;
}
```

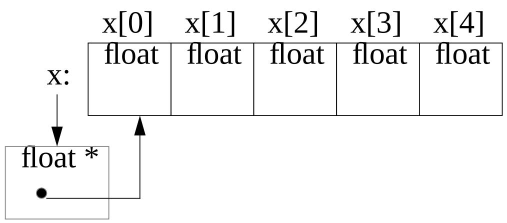
- `*p` derefences pointer to access object pointed at
- `*p` can be used on either side of assignment operator
- if `p` is equal to 0, then pointer is pointing at nothing and is called a *null* pointer.
- dereferencing a null pointer causes a core dump :-(

Pointers and Arrays

Consider

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

Our memory model is



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

Pointer Arithmetic

A pointer can point to element of an array

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

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

Examples

1. Summing an array Fortran style

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

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

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

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

More examples

3. Summing an array with pointer in Fortran style

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

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

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

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

Progression towards C++ style

Fortran style

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

Use add-and-assign operator

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

Use postfix increment operator

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

Combine postfix and dereference

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

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

Examples of Pointer Arithmetic

Reverse elements of an array (`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 HEP code will not be so terse
- in C++, we wouldn’t use pointers as much as in C

Runtime Array Size

In C++, one can dynamically allocate arrays

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

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

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

where the calling routine "owns" the memory

- in C, one does

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

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

```
delete [] x;
```

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

```
free(x);
```


Line fit example

Part 1(`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];
    }
```

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

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' };
```

- dimension of `hello1` is 2

The above is too tedious, so use double quotes

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

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

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

Variable Scope, Initialization, and Lifetime

Consider (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;
}
```

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

`for`-loop Scoping

Consider

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

- note where `i` is declared
- the scope of `i` is the scope just outside the for-loop block
- works for today's UNIX vendor's compilers

Current draft standard

- scope of `i` is *inside* for-loop block
- will need to declare `i` before `for` statement for `i` to have meaning after loop termination
- if declared in `for` statement, will need to repeat it for each `for` statement that follows
- vendor compilers will (eventually) change
- gcc 2.7.x supports draft standard now

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

References

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

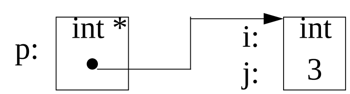
Reference (`ch/simplecpp.C`**)**

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

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

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

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



Reference arguments

Consider(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 `swap()`, it is as if the compiler generated this code to initialize its arguments

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

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

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


Homework

Given this declaration

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

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

Recursion

A function can call itself (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
```

the following is obsolete

```
#define M_PI 3.1415;
```

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

const Pointer

Consider

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

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

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

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

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

`const` function argument

Consider

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

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

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

Function Name Overloading

Pre-Fortran 77 we had

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

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

In C++, one can have

```
int    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

Summary

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

You can also do more than you can do in Fortran

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

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 a detector’s 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?

- started by Leif Lönnblad, Nordiita (via CERN, DESY and Lund)
- Nobu Katayama (KEK) is current editor.

Use

- examples of use at `/usr/local/doc/programming/C++class/SciEng/examples/clhep`
- header files: `/usr/local/lib/include/CLHEP`
- library file for gcc: `/usr/local/lib/libCLHEP.a`

ThreeVector

CLHEP**'s ThreeVector class (simplified)**

```
class Hep3Vector {
public:
  Hep3Vector();
  Hep3Vector(double x, double y, double z);
  Hep3Vector(const Hep3Vector &v);
  double x();
  double y();
  double z();
  double phi();
  double cosTheta();
  double mag();
  // much more not shown
private:
  double dx, dy, dz;
};
```

- this is the declaration in the header file
- keyword `class` starts the declaration which is contained within the `{}`
- class contains member functions
- an object can be an instance of a class
- an object of a class contains data members

Using a class object

Consider (`clhep/threeVector0.C`)

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

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

  while ( cin >> x >> y >> z ) {
    Hep3Vector aVec(x, y, z);

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

- `Hep3Vector aVec(x, y, z);` declares `aVec`, a object of type `Hep3Vector` 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 Hep3Vector {
public:
  // member functions

private:
  double dx, dy, dz;
};
```

- `Hep3Vector` contains 3 data members
- declaration is like any other except no initializers are allowed
- every instance of the class `Hep3Vector` will have its own 3 data members.

```
Hep3Vector x(1.0, 0.0, 0.0);
Hep3Vector y(0.0, 1.0, 0.0);
Hep3Vector z(0.0, 0.0, 1.0);
```

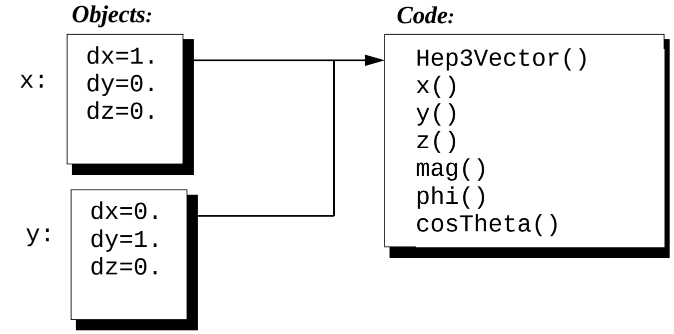
- `Hep3Vector` is a type
- an object of type `Hep3Vector` has a value (or state) that is represented by the values of its data members (like a complex number)
- the size of a `Hep3Vector` object is likely to be `3*sizeof(double)`

Memory model

Consider

```
Hep3Vector x(1.0, 0.0, 0.0);
Hep3Vector 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 members
- one copy of the code for a class is shared by all instances of the class
- hidden argument `this` is how it all works

Use of `private` keyword

We have

```
class Hep3Vector {
public:
  double mag();
  double x();
  double dummy;
  // member functions

private:
  double dx, dy, dz;
};
```

- the following compiles

```
Hep3Vector x(1.0, 0.0, 0.0);
cout << x.dummy;
```

- the following does not compile

```
Hep3Vector x(1.0, 0.0, 0.0);
cout << x.dx;  // 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

```
Hep3Vector a;
```

- with three `double` values

```
Hep3Vector a(1.0, 1.0, 1.0);
```

- copy of another object

```
Hep3Vector a(1.0, 1.0, 0.0);
Hep3Vector b = a;
```

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

There are three constructors in the class

```
class Hep3Vector {
public:
  Hep3Vector();
  Hep3Vector(double x, double y, double z);
  Hep3Vector(const Hep3Vector &v);
  // much more not shown
private:
  double dx, dy, dz;
};
```

Constructor Implementations

The constructor member functions

```
Hep3Vector::Hep3Vector(double x, double y, double z) {
  dx = x;
  dy = y;
  dz = z;
}

Hep3Vector::Hep3Vector(const Hep3Vector &vec) {
  dx = vec.dx;
  dy = vec.dy;
  dz = vec.dz;
}

Hep3Vector::Hep3Vector(){
}
```

- 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*
- note that copy constructor uses a const reference

Data Hiding

Violation of private parts?

```
Hep3Vector::Hep3Vector(const Hep3Vector &vec) {
  dx = vec.dx;
  dy = vec.dy;
  dz = vec.dz;
}
```

- 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 Hep3Vector {
public:
  double x();
  double y();
  double z();
  // much more not shown
private:
  double dx, dy, dz;
};
```

The implementation is

```
double Hep3Vector::x() {
  return dx;
}
double Hep3Vector::y() {
  return dy;
}
double Hep3Vector::z() {
  return dz;
}
```

- 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 double Hep3Vector::x() {
  return dx;
}
inline double Hep3Vector::y() {
  return dy;
}
inline double Hep3Vector::z() {
  return dz;
}
```

- can be used when execution of function body is shorter than time to call and return from function
- any decent compiler should produce inline code instead of function call for above
- inline keyword is just a hint, however
- data hiding is preserved
- implementation needs to be in the header file
- sometimes put in file with `.icc` suffix that is included by the header file (not BaBar practice)
- program could be faster
- program could be larger

More Implementation

Recall

```
class Hep3Vector {
public:
  double mag();
  double phi();
  double cosTheta();
  // much more not shown
private:
  double dx, dy, dz;
};
```

Implementation

```
inline double Hep3Vector::mag() {
  return sqrt(dx*dx + dy*dy + dz*dz);
}

inline double Hep3Vector::phi() {
  return dx == 0.0 && dy == 0.0 ? 0.0 : atan2(dy,dx);
}

inline double Hep3Vector::cosTheta() {
  double ptot = mag();
  return ptot == 0.0 ? 1.0 : dz/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

```
common/points/hits(3,100)
real*4 r
! do some work
r = hitsmag(i)
```

C++ design

C++ style

```
Hep3Vector hits[100];
// do some work
double 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 Hep3Vector {
public:
  Hep3Vector();
  Hep3Vector(double x, double y, double z);
  Hep3Vector(const Hep3Vector &v);
  inline double x();
  inline double y();
  inline double z();
  inline double phi();
  inline double cosTheta();
  inline double mag();
private:
  double 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 `Hep3Vector`

We'll now look at the real `Hep3Vector` class and explain those new language elements we need to understand it

```
class Hep3Vector {
public:
  inline Hep3Vector(double x=0., double y=0., double z=0.);
  inline Hep3Vector(const Hep3Vector&);
  double x() const;
  double y() const;
  double z() const;
  double phi() const;
  double cosTheta() const;
  double mag() const;
  // much more not shown
private:
  double dx, dy, dz;
};
```

- 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

Two forms to invoke copy constructor

```
Hep3Vector x(1.0, 0.0, 0.0);
Hep3Vector y = x;  // C style
Hep3Vector y(x);   // C++ class style
```

- the two are equivalent if argument is same type as object being declared
- both invoke copy constructor
- the `=` form allows user defined conversions when argument is not same type
- both forms allowed for built-in type

Consider

```
Hep3Vector x = 1.0;
```

- might be equivalent to

```
Hep3Vector tmp(1.0);
Hep3Vector x = tmp;
```

- but following has no suprises

```
Hep3Vector x(1.0);
```

Member Initializers

The constructor can be implemented like any other member function…

```
Hep3Vector::Hep3Vector(double x, double y, double z){
  dx = x;
  dy = y;
  dz = z;
}
```

- but data members need to be constructed before assignment
- for `Hep3Vector` the custom constructor would be called

An alternate form is use of member initializers

```
Hep3Vector::Hep3Vector(double x, double y, double z) :
    dx(x), dy(y), dz(z){}
```

- note the `:` preceding the opening `{`
- `dx(x)` notation calls a constructor directly
- which constructor depends on argument matching
- in the above case, it is the copy constructor
- the function body is required, even if empty

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 dx;
}
Vector3 vec;
// ...
vec.x() = 1.0; // uh?
```

- it is as if

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

Operators are functions?

Operators can be thought of as functions

```
double add( double a, double b) {
    return a + b;
}
double 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

```
double add( double a, double b);
double mul( double a, double b);
double 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 `Hep3Vector`

```
class Hep3Vector {
public:
  inline Hep3Vector& operator +=(const Hep3Vector &);
  // more not shown
```

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

```
Hep3Vector 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 `Hep3Vector` reference for consistency with built-in types

```
Hep3Vector p, q, r;
//
r = q += p;
// r.operator=( q.operator+=(p) )
```

Operator Function Implementation

Implementation

```
inline Hep3Vector& Hep3Vector::operator+=(const Hep3Vector& p) {
  dx += p.x();  // could have been dx += p.dx
  dy += p.y();
  dz += p.z();
  return *this;
}
```

- does the accumulation as one would expect
- `this` is a hidden argument that is a pointer to the object's own self
- `this->dx` is thus equivalent to `dx`
- remember: use `->` instead of `.` when you have a pointer
- or `dx` is shorthand for `this->dx`
- recall that `Hep3Vector::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

```
Hep3Vector 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
- and there's no operator one could use with the correct precedence
- can't overload operators for built-in types

One should only use when conventional meaning makes sense

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

- is this cross product or dot product?
- `Hep3Vector` defines it to be dot product

Non-member Operator Function

Consider

```
inline Hep3Vector operator*(const Hep3Vector& p, double a) {
  Hep3Vector q( a*p.x(), a*p.y(), a*p.z() );
  return q;
}
```

- invoked by

```
double 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 Hep3Vector operator*(const Hep3Vector& p, double a) {
  Hep3Vector q( a*p.x(), a*p.y(), a*p.z() );
  return q;
}
inline Hep3Vector operator*(double a, const Hep3Vector& p) {
  Hep3Vector q( a*p.x(), a*p.y(), a*p.z() );
  return q;
}
```

- second one invoked by

```
double 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 Hep3Vector(double x=0.0, double y=0.0, double z=0.0);
inline Hep3Vector(const Hep3Vector &);
```

- also contains conversion constructor

Destructor

```
inline ~Hep3Vector();
```

- invoked when object is deleted (more next session)

Accessor-like functions

```
inline double x() const;
inline double y() const;
inline double z() const;
inline double mag() const;
inline double mag2() const;
inline double perp() const;
inline double perp2() const;
inline double phi() const;
inline double cosTheta() const;
inline double theta() const;
inline double angle(const Hep3Vector &) const;
inline double perp(const Hep3Vector &) const;
inline double perp2(const Hep3Vector &) const;
```

The Complete List - II

Manipulators

```
void rotateX(double);
void rotateY(double);
void rotateZ(double);
void rotate(double angle, const Hep3Vector & axis);
Hep3Vector & operator *= (const HepRotation &);
Hep3Vector & transform(const HepRotation &);
```

Set functions

```
inline void setX(double);
inline void setY(double);
inline void setZ(double);
inline void setMag(double);
inline void setTheta(double);
inline void setPhi(double);
```

Output function

```
ostream & operator << (ostream &, const Hep3Vector &);
```

- allows

```
Hep3Vector x(1.0);
// ...
cout << x << endl;
```


The Complete List - III

Vector algebra member functions

```
inline double dot(const Hep3Vector &) const;
inline Hep3Vector cross(const Hep3Vector &) const;
inline Hep3Vector unit() const;
inline Hep3Vector operator - () const;
```

Vector algebra non-member functions

```
Hep3Vector operator+(const Hep3Vector&, const Hep3Vector&);
Hep3Vector operator-(const Hep3Vector&, const Hep3Vector&);
double operator * (const Hep3Vector &, const Hep3Vector &);
Hep3Vector operator * (const Hep3Vector &, double a);
Hep3Vector operator * (double a, const Hep3Vector &);
```

Assignment operators

```
inline Hep3Vector & operator = (const Hep3Vector &);
inline Hep3Vector & operator += (const Hep3Vector &);
inline Hep3Vector & operator -= (const Hep3Vector &);
inline Hep3Vector & operator *= (double);
```

Summary

`Hep3Vector` 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 `Hep3Vector`

We have a new type with higher level of abstraction

Levels of Abstraction in Physics

Do you recognize these equations?

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

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

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

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

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

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

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

Warning: this will not be a production quality class

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 Declaration

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.
- must implement else default is member-wise copy leading to two objects sharing the same data
- is the behaviour what we expected?

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
SimpleFloatArray z;
//
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 attempts to do both construction and assignment

- might add another constructor function to allocate and assign

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


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`, `Hep3Vector`, *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<Hep3Vector> 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;
Hep3Vector 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`)

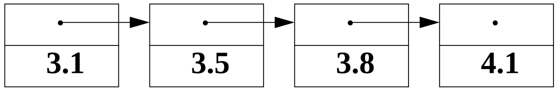
```
int main() {
    // Read list of values and find minimum.
    List<float> list;
    float val;
    float minval = FLT_MAX; // from <float.h>
    while ( cin >> val) {
        if (val < minval) minval = val;
        list.add(val);
    }

    // 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

Warning: this will not be production quality class

`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

Homework

Write a `SimpleArrayIterator<>` class with

- template class to work with `SimpleArray<>` class
- only four member functions: constructor, `advance()`, `current()` and `more()`

We know the behavior, but what are the data members?

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++) {
    sum += *i;
}
```

- implement `operator++()`
- implement the deference operator
- make interator look like pointers

Use of Containers

Chamber containing layers

```
class Chamber {
//
private:
    Array<Layer *> layers;
// ...
}
```

- size is known at compile time

Event containing tracks and clusters

```
class Event {
//
private:
    List<Tracks *> tracks;
// ...
}
```

- size not known at compile time

Why use pointers?

- avoid copying object into list
- needed when same object is reference by multiple lists, *e.g.* tracks can share hits
- but must be careful of memory management

CLHEP containers

HepAList<class T>

- template class
- stores pointers to objects, *i.e.* does not copy objects
- behaves like both list and array
- array based implementation of list like-object
- has associated iterator

HepCList<class T>

- makes copy of objects

`HepVector`

- vector of n dimension
- `stores doubles`
- has mathmatical properties

CLHEP **containers being phased out of BaBar code**

Rogue Wave Collection Classes

Tool.h++ class library

- commerical libary
- 190 classes
- organized as number of different categories

BaBar reconstrction code uses

- `RWTValOrderedVector<>` for copying object
- `RWTPtrOrderedVector<>` for copying pointer to object
- `RWTValDList<>` and `RWTPtrDList<>` when size is not known at compile time

Standard Template Library (STL)

Features

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

Status

- contributed by HP labs, Palo Alto
- part of the draft standard since July 1994
- under UNIX, HP reference version compiles only with IBM's xlC
- hacked version works with gcc
- we'll migrate to it in the future
- 4 books have been written about it (for example, Musser and Saini)

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 (next session)

Recall ThreeVector

CLHEP**'s ThreeVector class (simplified)**

```
class Hep3Vector {
public:
  Hep3Vector();
  Hep3Vector(double x, double y, double z);
  Hep3Vector(const Hep3Vector &v);
  inline double x();
  inline double y();
  inline double z();
  inline double phi();
  inline double cosTheta();
  inline double mag();
  // much more not shown
private:
  double dx, dy, dz;
};
```

and some of the implementation

```
inline double Hep3Vector::x() {
  return dx;
}
inline double Hep3Vector::mag() {
  return sqrt(dx*dx + dy*dy + dz*dz);
}
```

Recall our test program

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

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

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

  while ( cin >> x >> y >> z ) {
    Hep3Vector 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;
double z;
// …
z = p*q;
r = p + q;
```


Possible 4-Vector Class

Might look like…

```
class HepLorentzVector {
public:
  HepLorentzVector();
  HepLorentzVector(double x, double y, double z, double t);
  HepLorentzVector(const HepLorentzVector &v);
  inline double x();
  inline double y();
  inline double z();
  inline double t();
  inline double phi();
  inline double cosTheta();
  inline double mag();
  // much more not shown
private:
  double dx, dy, dz, dt;
};
```

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 HepLorentzVector {
public:
  HepLorentzVector();
  HepLorentzVector(double x, double y, double z, double t);
  HepLorentzVector(const HepLorentzVector &v);
  inline double x();
  inline double y();
  inline double z();
  inline double t();
  inline double mag();
  // much more not shown
private:
  Hep3Vector vec3;
  double dt;
};
```

- `HepLorentzVector` *has-a* `Hep3Vector`
- could also say `HepLorentzVector` is built by aggregation
- or with containment

Possible implementation

Constructors

```
HepLorentzVector::HepLorentzVecor() :
  vec3(), dt(0.0){}

HepLorentzVector::
HepLorentzVector(double x, double y, double z, double t) :
  vec3(x, y, z), dt(t) {}

HepLorentzVector::
HepLorentzVector(const HepLorentzVector &v ) :
  vec3(v.vec3), dt(v.dt) {}
```

- note use of initializers
- must construct data members when constructing class object

Let 3-vector component do part of the work

```
double HepLorentzVector::mag() {
    return sqrt(dt*dt - vec3.mag2() );
}
```

must still implement functions like

```
double HepLorentzVector::x() {
    return vec3.x();
}
```

YAPI

Constructors

```
class HepLorentzVector {
public:
  HepLorentzVector();
  HepLorentzVector(double x, double y, double z, double t);
  HepLorentzVector(const HepLorentzVector &v);
  inline double x();
  inline double y();
  inline double z();
  inline double t();
  inline double mag();
  // much more not shown
private:
  Hep3Vector *vec3;
  double dt;
};
```

- still have containment, but use a pointer
- makes sense in some situations (probably not here)

YAPI implementation

Constructors might be

```
HepLorentzVector::HepLorentzVecor() : dt(0.0)
{
  vec3 = new Hep3Vector(0, 0, 0);
}

HepLorentzVector::
HepLorentzVector(double x, double y, double z, double t) :
  dt(t)
{
  vec3 = new Hep3Vector(x, y, z);
}

HepLorentzVector(const HepLorentzVector &v ) : dt(v.dt)
{
  vec3 = new Hep3Vector(v.vec3); // copy constructor
}
```

- using `new` operator to create one object
- will need to implement destructor!

Inheritance

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

```
class HepLorentzVector : public Hep3Vector {
public:
  HepLorentzVector();
  HepLorentzVector(double x = 0., double y = 0.,
                   double z = 0., double t = 0.);
  HepLorentzVector(const HepLorentzVector &v);
  HepLorentzVector(const Hep3Vector &p, double t);
  double t();
  double mag();
  // much more not shown
private:
  double dt;
};
```

- `HepLorentzVector` *is-a* `Hep3Vector`
- All public members of `Hep3Vector` are also public members of `HepLorentzVector` by use of keyword `public` in class declaration.
- member function `t()` is added
- member function `mag()` overrides function of same name in `Hep3Vector`
- constructors take different arguments
- one new data member: `dt`

Use of Lorentz Vector

Consider (`clhep/fourVector0.h`)

```
int main() {
  double x, y, z, t;
  while ( cin >> x >> y >> z >> t ) {
    Hep3Vector a3Vec(x, y, z);
    HepLorentzVector 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

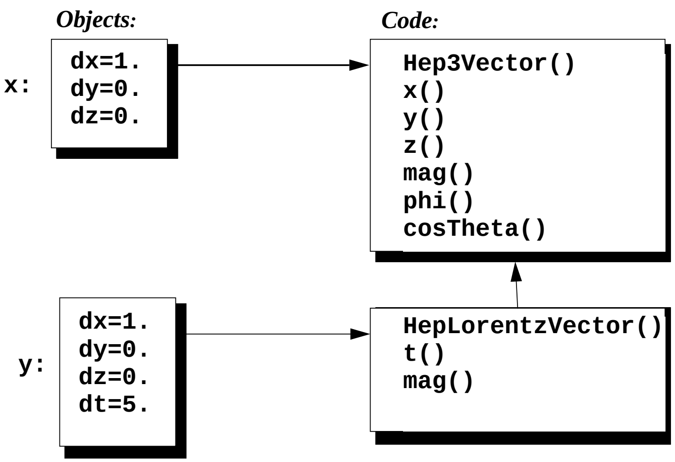
```
hpkaon> a.out
1 1 1 2
3-vector x and mag: 1 1.73205
4-vector x and mag: 1 1
```

Memory model

Consider

```
Hep3Vector x(1.0, 0.0, 0.0);
HepLorentzVector y(1.0, 0.0, 0.0, 5.0);
```

In computer's memory we have



- inheritance of data members
- inheritance of member functions

Constructor Implementations

Constructors

```
HepLorentzVector::
HepLorentzVector(double x, double y, double z, double t) :
  Hep3Vector(x, y, z), dt(t) {}

HepLorentzVector::
HepLorentzVector(const Hep3Vector &v, double t) :
  Hep3Vector(v), dt(t) {}

HepLorentzVector::
HepLorentzVector(const HepLorentzVector &v) :
  Hep3Vector(v), dt(v.dt) {}
```

- super class will be constructed before subclass
- use initializers to direct how to construct superclass

More of Implementation

As you might expect

```
inline double HepLorentzVector::t() const {
  return dt;
}
```

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

This doesn't work

```
inline double HepLorentzVector::mag2() const {
  return dt*dt - (dx*dx + dy*dy + dz*dz);
}
```

- `dx`, `dy`, and `dz` were declared `private`
- `private` means access to objects of the same class and `HepLorentzVector` is a different class
- could modify `Hep3Vector` to

```
class Hep3Vector {
public:
// same as before
protected:
double dx, dy, dz;
}
```

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

More on Implementation

Keep the base class data members private

```
inline double HepLorentzVector::mag2() const {
  return dt*dt - Hep3Vector::mag2();
}
```

- use scope operator `::` to access function of same name in super class
- now we can re-write `Hep3Vector` to use `r`, `costheta` and `phi` without needing to re-write `HepLorentzVector`
- less dependencies between classes is good

Finally, we have

```
inline double HepLorentzVector::mag() const {
  double 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 double
HepLorentzVector::dot(const HepLorentzVector & p) const {
  return dt*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 HepLorentzVector &
HepLorentzVector::operator += (const HepLorentzVector& p) {
  Hep3Vector::operator += (p);
  dt += 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 HepLorentzVector::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*dt);
  setY(y() + gamma2*bp*by + gamma*by*dt);
  setZ(z() + gamma2*bp*bz + gamma*bz*dt);
  dt = gamma*(dt + bp);
}
```

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

Also have

```
inline Hep3Vector HepLorentzVector::boostVector() const {
  Hep3Vector p(x()/dt, y()/dt, z()/dt);
  return p;
}
inline void HepLorentzVector::boost(const Hep3Vector & p){
  boost(p.x(), p.y(), p.z());
}
```

Diagrams

The old ones

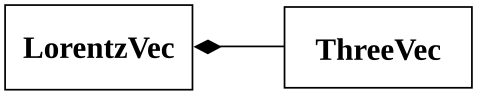
- Booch's "clouds", supported by Rational/Rose
- Rumburgh's OMT

The new one

- UML: Unified Modeling Language
- Booch and Rumburgh working together
- submitted for standardization

Aggration

If we have a *has-a* relationship we draw it thus



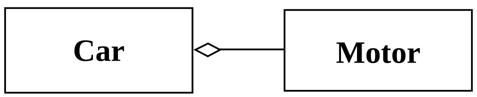
- corresponding code…

```
class LorentzVec {
  // much more not shown
private:
  ThreeVec vec3;
  double dt;
};
```

- `LorenzVec` contains `ThreeVec`
- contained object will be destroyed with the containing object is destroyed

Association

If we have *a association* relationship we draw it thus



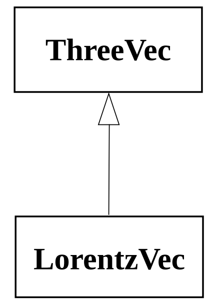
- corresponding code…

```
class Car {
  // much more not shown
private:
  Motor *m;
  };
```

- `not 100% sure just because we have pointer`
- only association if motor is replaceable
- depends on what kind of application this `Car` class is being used for.

Inheritance

If we have *is-a* relationship we draw it thus



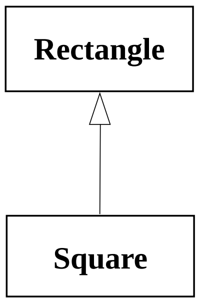
- corresponding code

```
class LorentzVec : public ThreeVec {
  // much more not shown
private:
  double dt;
};
```

- this is class relationship, not object relationship
- don’t be confused with our memory model diagrams
- we say `ThreeVec` is base class and `LorentzVec` is derived class

Bad inheritance

When a square is a rectangle and when it isn't



- corresponding code

```
class Rectangle {
  // much more not shown
  void setLength(float);
  void setHeight(float);
//...
  float length, height;
};
```

- now what's the `Square` going to do about these member functions?
- in math, a square is a subset of all rectangles, but in C++ a `Square` is not a subclass of `Rectangle`

A Possible Particle class

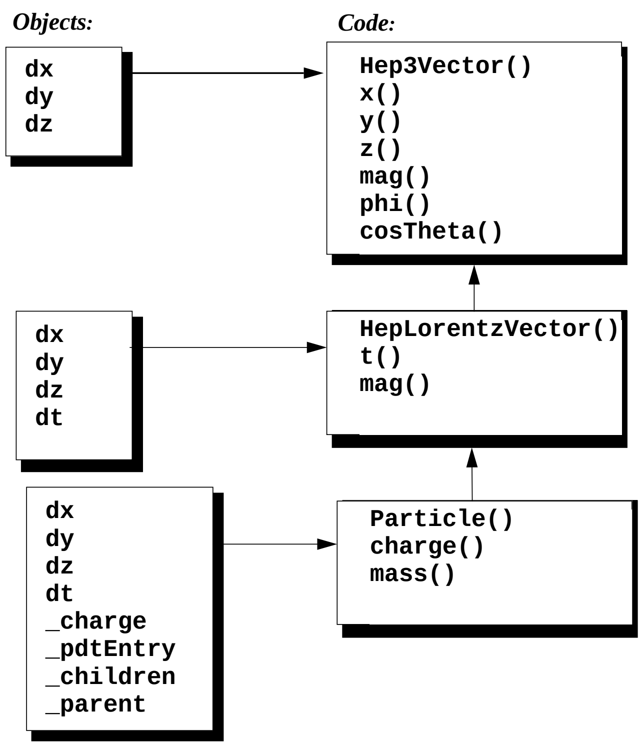
Take Lorentz vector and add to it

```
class Particle : public HepLorentzVector
{
public:
     Particle();
     Particle(HepLorentzVector &, PDTEntry *);
     Particle(const Particle &);
     virtual ~Particle() {}
     float charge() const;
     float mass() const;
  // 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 derived class
- added features are charge, pointer to entry in particle data table, list of children, and pointer to parent
- owns list of children
- _pdtEntry and _parent are pointers because of shared objects
- not very useful class

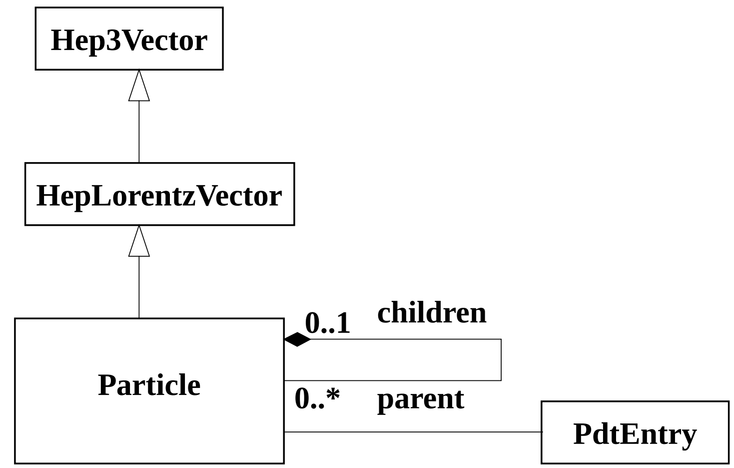
Data Model

In computer's memory we have



Class Diagram

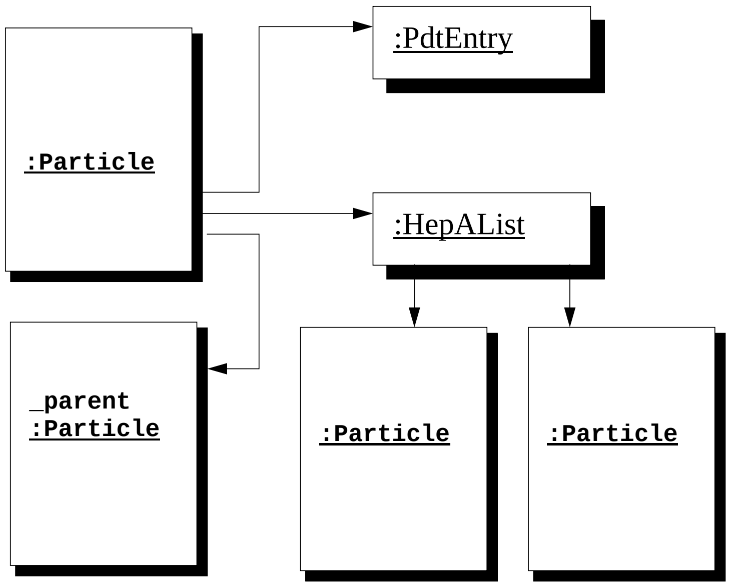
Inheritance and relationships



- `Particle` has 0 to n children and 0 or 1 parents
- `Particle` has association with `PdtEntry`
- we leave the `HepAList<>` out of the picture

Object Hierarchy

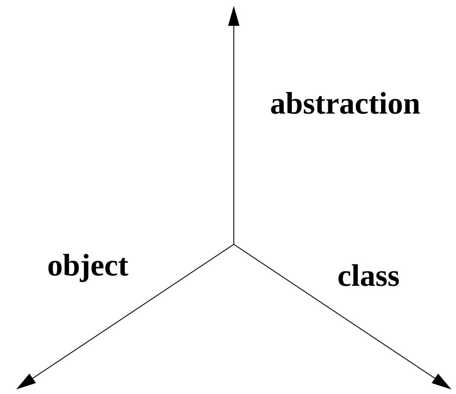
In computer memory we have



- the class and object hierarchies are different in dimensions

The 3 hierarchies of OOP

It's a three dimensional space



- Class hierarchy describes behavior
- Object hierarchy describes data structure
- hierarchy of levels of abstraction, *e.g.* float, vector, lists, arrays, particle, *etc.*

Multiple Inheritance

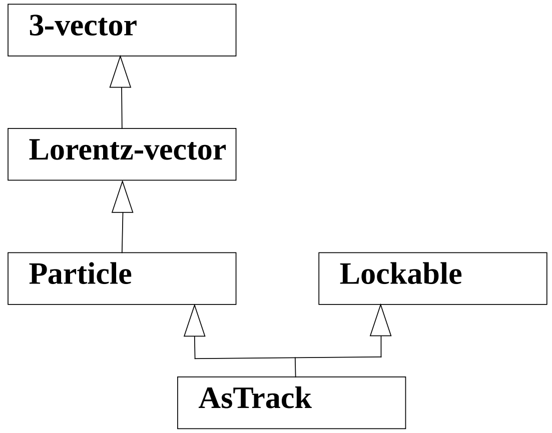
One can inherit from more than one class (`aslund/AsTrack.h`**)**

```
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

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

Summary

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

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

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

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

Plan of the day

Few more language features

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

PdtLund Class

Extract from this class (PDT/PdtLund.hh)

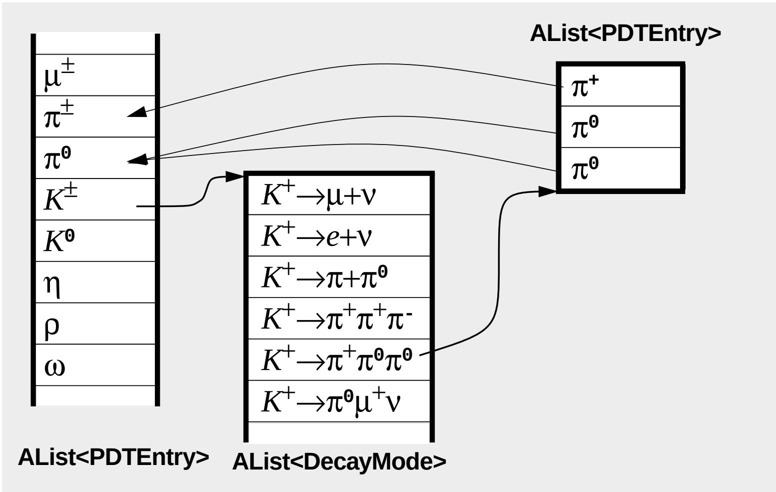
```
class PdtLund
{
public:
// a list of common particles
// the numbers are PDG standard particle codes
  enum LundType {
    e_minus = 11, nu_e, mu_minus, nu_mu,
    e_plus = -11, nu_e_bar = -12
// many more not shown
  };
};
```

- `enum` nested in class
- must use scoping to access outside of class

```
PdtLund::LundType l = PdtLund::e_minus;
```

- the scoping helps the readability and avoids name conflicts
- scope type and constants

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

`static` keyword

Part of the Pdt class declaratin (`PDT/Pdt.hh`)

```
class Pdt
{
public:
  // return entry pointer given particle id or name
  static PdtEntry* lookup(const char *name);
  static PdtEntry* lookup(PdtLund::LundType id);
  static PdtEntry* lookup(PdtGeant::GeantType id);
  static float mass(PdtLund::LundType id);
  static float mass(PdtGeant::GeantType id);
  static float mass(const char* name);
// more not shown
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

```
    mass = Pdt::mass( PdtLund::pi_plus);
```


`PDTEntry` class

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

```
class DecayMode;
class PdtEntry {
public:
  const char *name() const {return _name;}
  float  charge() const {return _charge;}
  float mass() const {return _mass;}
  float width() const {return _width;}
// more not shown
protected:
  char *_name;
  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;
  PdtLund::LundType _lundid;
  PdtGeant::GeantType _geantid;
};
```

- note forward declaration of class

`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;
};
```

- nothing new

Detector Simulation

What classes are involved?

- 3-vector
- geometry
- track
- detectors
- fields
- *etc*

Will take examples from Gismo project

- C++ framework for detector simulation and reconstruction;
- we'll see how it differs from the Fortran *black box* approach, *e.g.* GEANT 3

Gismo History

Version 0, the prototype

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

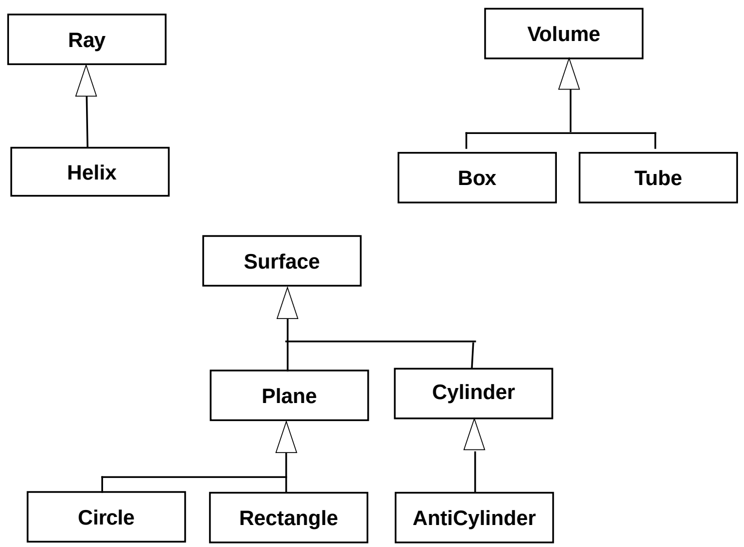
Version 1, previous release

- 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, current version

- written by Atwood and Burnett
- C++ with templates and CLHEP
- `http://www.phys.washington.edu/~burnett/gismo/`

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;
  const ThreeVec& position() const {return pos;}
  virtual double curvature() const;
  virtual double
  distanceToLeaveSurface( const Surface* s, ThreeVec& p ) const;
// more not shown
protected:
  ThreeVec pos;
  ThreeVec dir;
  float arclength;
};
```

- you can pretty well guess the significance of the data members and many of the member functions
- 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;
  double curvature() const { return 1.0 / rho; }
  virtual double
  distanceToLeaveSurface( const Surface* s, ThreeVec& p ) const;
  // many more not shown
protected:
  ThreeVec axis;  // helix axis direction (unit vector)
  double rho;     // helix radius, sign significant
  ThreeVec perp;  // perpendicular direction
  double parallel;// component along axis
};
```

- many member functins must be re-implemented here, so probably a `Helix` is not a `Ray`
- 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;
    virtual int withinBoundary( const ThreeVec& x ) const = 0;
/// more not shown
};
```

- data members can be first in file, but not usual practise
- 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 );
    virtual double distanceAlongRay(
        int which_way, const Ray* ry, ThreeVec& p ) const;
    virtual double distanceAlongHelix(
        int which_way, const Helix* hx, ThreeVec& p ) const;
 // more not shown
private:
   double d;
   // offset from origin to surface
};
```

- `Plane` is infinite since it has no data members to describe boundary
- distance along ray to infinite plane can be calcutated, so implementatin does exist here

`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;
};
```

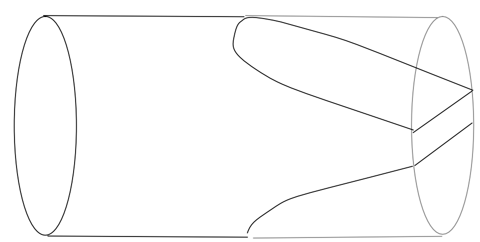
- `Volume` is a base class with common functionality of all volumes
- it contains a list of surfaces that describe the volume
- it contains a 3-vector for its center and 3 doubles for its rotation
- member functions not shown allow one to build abitrary volumes, move them, and rotate it.
- for tracking,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

Subclasses of Volume

Box

```
class Box : Volume
{
  Box( float len, float width, float height);
  Box(const Box &);
  virtual ~Box();
  // very little not shown
};
```

- constructor builds six surfaces, positions them, and adds them to surface list
- hardly any other member functions, nor any data members
- same for Cylinder and other classes
- any one could add a new volume subclass in a smiliar way, for example a light pipe



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;
  HepAListIterator<Surface> iter(surface_list);
  while ( s = iter.next() ) {
    d = r.distanceToLeaveSurface( s, temp );
    if ( ( t > d ) && ( d >= 0.0 ) ) {
       t = d;
       p = temp;
       sf = s;
    }
  }
  return t;
}
```

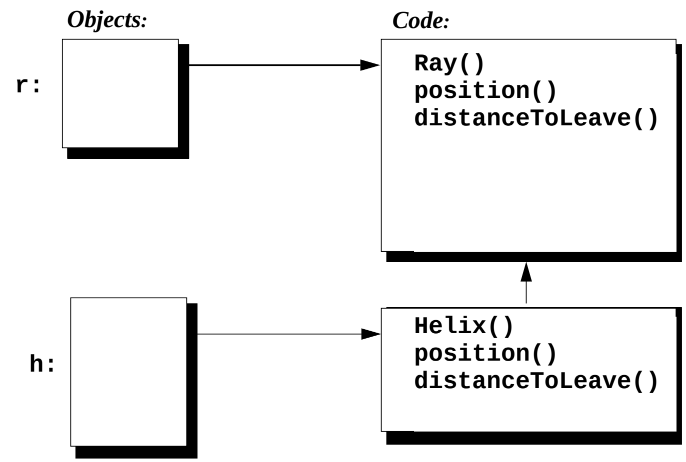
- loop over all surfaces to find the shortest distance
- the `Ray` object appears to do the work
- we don’t know if the `Ray` object is-a `Ray` or the `Helix` subclass

Recall Memory model

Consider

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

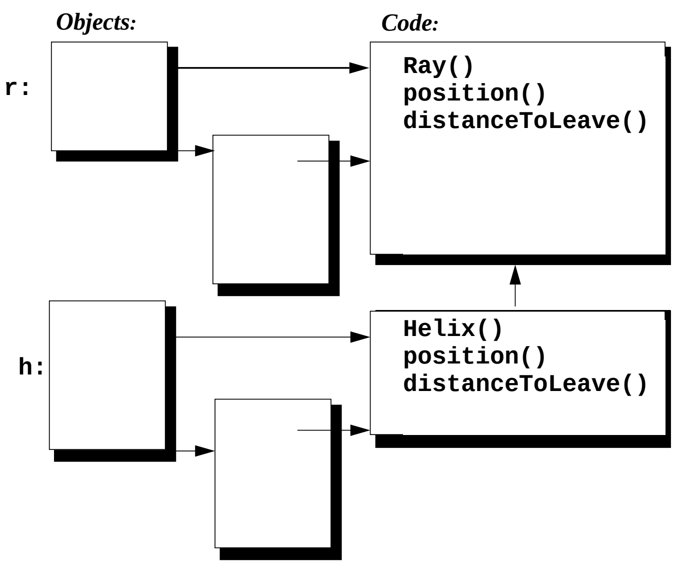
In computer’s memory we have



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

The virtual function table

Memory model with virtual functions



- virtual member functions are invoked indirectly via the virtual function table
- the table contains pointers to the member functions
- 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;
  HepAListIterator<Surface> = iter(Surface_list);
  while ( s = iter.next() ) {
    d = r.distanceToLeaveSurface( s, temp );
    if ( ( t > d ) && ( d >= 0.0 ) ) {
       t = d;
       p = temp;
       sf = s;
    }
  }
  return t;
}
```

- compiler creates different machines instructions to invoke a virtual member function
- `distanceToLeaveSurface` was declared `virtual` so correct function gets called
- can even add another subclass of `Ray` without recompiling this code

Following the trail

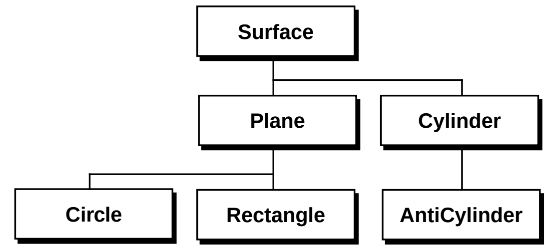
In `Ray` and `Helix` we have

```
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 Visitor 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`?



- it's not in `Surface`
- one implementation in `Plane`
- but we really instansiate 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 );
  ThreeVec nhat = normal(); // Normal to plane
  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`
- example of template pattern

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;
  HepListIterator<Surface> = it(surface_list);
  while ( s = it() ) {
    delete s;
  }
  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!

But…

- its like you’ve heard lectures on how to swim, but now you face the deep end of the pool
- its like you know the rules of the game of chess, but have not yet studied stratgies

Further reading:

- Designing object-oriented C++ applications using the Booch method, Robert C. Martin, ISBN 0-13-203837-4, Prentice Hall
- Design Patterns, Gamma, Helm, Johnson, and Vlissides, ISBN 0-201-63361-2, Addison-Wesley