An Introduction to C++

Benno List

DESY Summer Students Tutorial

30.7 and 1.8.2007

Introduction I

- C++ is one of the most complicated programming languages around
- FORTRAN is like a VW beetle:
  simple, reliable, easy to master
- C++ is like a Formula 1 racer:
  incredibly powerful, but difficult to drive

FORTAN

C++

Introduction II

- The best way to learn programming is to look at programs
- I'll show many code examples
- In your work, you will mostly start with an example program and adapt it to your needs
  - I concentrate on showing you how to understand what existing programs do
  - Programming languages are like all languages: **You cannot write if you can't read!**
- For reasons of space, examples are ususally not production-quality code!
  - I often omiss (essential!) error checking
  - I often prefer simple code over the most concise code
  - Sometimes I avoid syntactic complications (omit “const”, don't use references) for the sake of brevity and clarity

Hello, World!

Our first C++ program:

Note: C++ is case-sensitive:
cout, Cout and COUT are 3 different things!

file: `hello.C`

```
#include <iostream>

using namespace std;

int main() {
  cout << "Hello, World!\n";
  return 0;
}
```

Reads in file "`iostream`", which declares cout

Without this, we would have to write `std::cout`

This is the main program, returning an integer
Prints out "`Hello, World`", "`\n`" ends the line
returns "0" to the shell: no error

Note: a semicolon ends each statement.

In the shell:

```
$> g++ -o hello hello.C
$> ./hello
Hello, World!
$>
```

`g++` is the compiler, `hello` is the excutable file
execute "`hello`"
yes, it works!

Functions

- In C++: almost everything returns a value
  => no "SUBROUTINE"s in C++, only "FUNCTION"s
- No implicit typing, every function and variable has to be declared

file: `area.h`

```
double area (double radius);
```

Declares the function:
function takes one argument "`radius`" of type "`double`", returns a "`double`" value

file: `area.C`

```
#include "area.h"

double area (double radius) {
  double result = 3.14159276*
    radius*radius;
  return result;
}
```

Includes the declaration file

Defines the function

Note: linebreaks are allowed almost everywhere

Using Functions

file: `calcarea.C`

```
#include <iostream>
using namespace std;
#include "area.h"

int main() {
  cout << "Enter radius: ";
  double radius;
  cin >> radius;
  cout << "Area of circle with radius "
       << radius << " is "
       << area (radius) << endl;
  return 0;
}
```

Includes the declaration files

Note: <> for standard headers,
"" for user headers!

`cin` reads from standard input

In the shell:

```
$> g++ -o calcarea calcarea.C area.C
$> ./calcarea
Enter radius: 1.5
Area of circle with radius 1.5 is 7.06858
$>
```

Basic Types

- Some of the types available in C++

C++ Type	Meaning	Size	Range (appr.)	Resolution
`int, long`	Integer	32 bit	±2147483648	1
`float`	Floating-point	32 bit	$\pm 3 \cdot 10^{\pm 38}$	$1 \cdot 10^{-7}$
`double`	Floating-point	64 bit	$\pm 2 \cdot 10^{\pm 308}$	$2 \cdot 10^{-16}$
`bool`	Boolean value	32 bit (!)	false, true	
`char`	Character, integer	8 bit	-128 - 127	1
`short`	Integer	16 bit	±32768	1
`long long`	Integer	64 bit	$\pm 9 \cdot 10^{18}$	1

Operators I: Arithmetic operators

- Arithmetic operators:

Operator	Meaning
-	Sign Change
*	Multiplication
/	Division
%	Modulus
+	Addition
-	Subtraction

note: no exponentiation! use “pow” function

- Assignment: = evaluates right side, assigns value to left side

```
double radius = 1.5;
double result = 3.14159276*radius*radius;
int i = 1;
i = i + 1;   // now i is 2!
```

Operators II:

- Special cases:

```
int i = 1;

i += 1;
i *= 3;
++i;

int j = ++i;

j = i++;
```

same as `i = i+1;` now `i` is 2
same as `i = i*3;` now `i` is 6
increments `i`. Now `i` is 7.

assigns new value of `i` to `j. => j is now 8.`
called "pre-increment"
assigns ***old*** value to k. => k is now 8, but `i` is 9 !
called "post-increment"

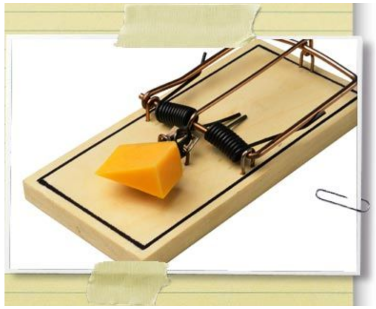
- The operators "+=", "*=" etc work also for `float`, `double` etc.

- Precedence as usual, evaluation from left to right:
  `a = b+2*-c +d%e;` is same as
  `a = (b+(2*(-c))) +(d%e);`

Operators III: Relational Operators

- Relational (comparison) operators: return “false” or “true”

Operator	Meaning
==	Equal
!=	Not equal
<	Less than
<=	Less or equal
>	Greater than
>=	Greater or equal



- Careful: “==” is a comparison, “=” is an assignment!
- In C/C++, an assignment has also a value: the assigned value:
  `a = (b = 7) + 1;` is legal (b becomes 7, a becomes 8)
- Therefore: `if (a=7)`... is also legal, but not what you want!

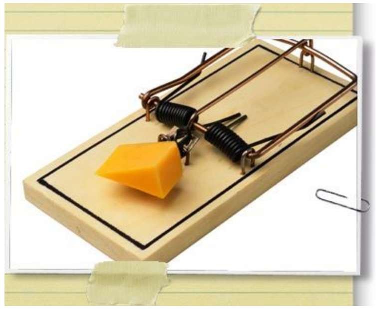
Operators IV: Logical Operators

- Logical operators: used for boolean expressions

Operator	Meaning
!	Not
!=	Exclusive or
&&	And
||	Or

- Bitwise operators: Perform bit-by-bit operations on integer types

Operator	Meaning
~	Bitwise complement
&	Bitwise and
^	Bitwise exclusive or
|	Bitwise or



- Careful! Don't confuse logical and bitwise operators!
  integers can be converted to bool: 0 is false, everything else is true
  => 7 && 8 is true, 7 & 8 is 0 is false!

Operators V: Input and Output

```
#include <iostream>

using namespace std;

int main() {
  int i;
  double d;
  cout << “Enter an integer and a double: “;
  cin >> i >> d;
  cout << “The integer is “ << i
       << “ and the double is “ << d << endl;
  cerr << “This is an error message\n”;
  return 0;
}
```

Every UNIX program has 3 pre-defined inputs/outputs:
`cin` is the standard input.
`cout` is the standard output.
`cerr` is the error output.

“<<” is the output operator.
“>>” is the input operator.

Numerical Functions

- Available from `<cmath>`
  Don't forget "`using namespace std;`"!

Function	Meaning	Remark
`sin(x)`	Sine	
`cos (x)`	Cosine	
`tan (x)`	Tangent	
`asin(x)`	Arc sine	
`acos(x)`	Arc cosine	
`atan(x)`	Arc tangent	$-\pi/2 < \text{Result} < \pi/2$
`atan2(x,y)`	Arc tangent (x/y)	$-\pi < \text{Result} < \pi$
`exp(x)`	Exponential	
`log(x)`	Natural logarithm	
`log10(x)`	Logarithm, base 10	
`abs(x)`	Absolute value	
`sqrt(x)`	Square root	
`pow (x, y)`	x to the power y	only for x >= 0
`pow (x, i)`	x to the integer power	also for x<0

Type Conversions I: Automatic Conversions

C/C++ has many pre-defined type conversions that are applied automatically, when necessary:

- integer types (int, short, char, long long) to floating point types (float, double): gives the same number
  careful: for large integers, the conversion is not exact!
- floating point types to integer types:
  the number is truncated (not rounded!) towards 0:
  1.3 -> 1, 1.7 -> 1, -1.8 -> -1
- Number types to bool: 0 -> false, non-zero -> true
- arithmetic expressions between integers result in integers:
  7/3 -> 2, 4/5 -> 0
- arithmetic expressions between floats (and integers) result in floats:
  1.3*5 -> 6.5, 4.0/5 -> 0.8, 4/5.0 -> 0.8
- Arguments of arithmetic functions are (often) automatically converted:
  sqrt (2) -> 1.41

Type Conversions II: Casts

You can explicitly ask for a type conversion. This is called a **cast**. (Like “casting bronze”)

- C-style casts: (type)expression:

```
double d = 3.7;
int i = (int)d * 2; // i is 3*2=6, not 7!
```

  - **discouraged!!! hard to read, ambiguous**

- C++ style casts:

```
int i = static_cast<int>(d) * 2;
```

  - **the recommended form.**
  - other casts exist (`dynamic_cast, reinterpret_cast, static_cast`)

Control Strutures I: If-then-else

```
double maximum (double a, double b) {
    double result;
    if (a > b) {
      result = a;
    }
    else {
      result = b;
    }
    return result;
  }
```

- condition in parantheses after "`if`"
- note: `result` must be declared *before* the if-block
- multiple statements after `if()` and `else` must be enclosed in curly braces.

Note: no semicolon needed (but allowed) after curly braces

```
  double maximum (double a, double b) {
    double result;
    if (a > b) result = a;
    else result = b;
    return result;
  }
```

for single statements after `if()` and `else`, we don't need the curly braces. (But use them anyway!)

```
  double maximum (double a, double b) {
    double result = (a > b) ? a : b;
    return result;
  }
```

"`?  :`" is a special operator (taking *three* arguments), especially for cases such as this one.

```
  double maximum (double a, double b) {
    return (a > b) ? a : b;
  }
```

The variable `result` is unnecessary.

Control Structures II: while, do-while

```
double power (double x, int n) {
  // evaluates x^n, for nonnegative n
  double result = 1;
  int i = 0;
  while (i < n) {
    result *= x;
    ++i;
  }
  return result;
}

double exponential (double x) {
  /* calculates exp(x)
     exp (x) = 1 + x + x^2/2 + ... x^i/i! */
  double result = 1, xx = 1;
  int i = 1;
  do {
    xx *= x/i;
    result += xx;
    ++i;
  } while (xx > 0.0000001 * result);
  return result;
}
```

By the way: This is a single-line comment

- This block is executed only if i<n; once i >= n, go to next statement
- Block may be executed 0 times (for n == 0)

By the way: This is a multi-line comment

- This block is repeated as long as xx > 0.0000001 * result.
- Block is executed at least once!

Control Structures III: for

```
double power (double x, int n) {
  // evaluates x^n, for nonnegative n
  double result = 1;
  int i = 0;
  while (i < n) {
    result *= x;
    ++i;
  }
  return result;
}

double power (double x, int n) {
  // evaluates x^n, for nonnegative n
  double result = 1;
  for ( int i = 0 ; i < n ; ++i ) {
    result *= x;
  }
  return result;
}
```

- A `for`-loop is exactly equivalent to a `while`-loop
- Just a convenient short-hand notation

More Complicated Data Structures: Classes I

file `Vector.h`:

```
class Vector {
  public:
    double x, y, z;
};
```

Note: Here the semicolon is mandatory!!!

- In a class, several variables (“data members”) can be grouped togther
- “public” means: other parts of the program may access the variable
- A class creates a new variable type!

file `calcVectorLength.h`:

```
double calcVectorLength (Vector v);
```

file `calcVectorLength.C`:

```
#include “Vector.h”
include <cmath>
using namespace std;

double calcVectorLength (Vector v) {
  return sqrt (pow (v.x, 2) +
   pow (v.y, 2)+pow (v.z, 2));
}
```

Here we have to pass only one variable of type `Vector`, instead of 3

Classes II

```
#include "Vector.h"
#include "calcVectorLength.h"
#include <iostream>
using namespace std;


int main() {
  Vector v;
  cout << "Enter three vector components:";
  cin >> v.x >> v.y >> v.z;
  cout << "Length of this vector is "
       << calcVectorLength (v) << endl;
  Vector w = v;
  cout << "Length of vector w is "
       << calcVectorLength (w) << endl;
  return 0;
}
```

- Creates a Vector named v.
- Reads in the components: v.x is x-component of v!
- Calculates the length.
- Creates a new Vector w, which is a copy of v.

Critique:

- Need extra files for calcVectorLength
- How can I create a Vector with defined (x, y, z) in a single step?

Classes III: Function Members / Methods

file `Vector.h`:

```
class Vector {
  public:
    Vector (double x_, double y_, double z_);
    double length();
    double x, y, z;
};
```

- This is a “constructor”
- This calculates the length of a Vector; it is a function: therefore the “()”, but takes no arguments

file `Vector.C`:

```
#include “Vector.h”
#include <cmath>
using namespace std;

Vector::Vector (double x_, double y_, double z_) {
  x = x_; y = y_; z = z_;
}

double Vector::length() {
    return sqrt (pow (x, 2) + pow (y, 2)+pow (z, 2));
}
```

Note: Here we really need the header file, because it declares the layout of the class

Note: in the definition of the function outside the “`class Vector {};`”, we have to give the class name explicitly

Here we use x, y, z directly, without any “v.”!

Classes IV

file vectorlength.C:

```
#include “Vector.h”
#include <iostream>
using namespace std;


int main() {
  double x, y, z;
  cout << “Enter three vector components:”;
  cin >> x >> y >> z;
  Vector v (x, y, z);
  cout << “Length of this vector is “
       << v.length() << endl;
  Vector w = v;
  cout << “Length of vector w is “
       << w.length() << endl;
  return 0;
}
```

- Now we can also create a Vector directly from its components, using the constructor
- Calculates the length.

Critique:

- Maybe storing x, y, z is very inefficient? Maybe we prefer polar coordinates?

Classes V: Private

file `Vector.h`:

```
class Vector {
  public:
    Vector (double x_, double y_, double z_);
    double length();
  private:
    double r, phi, theta;
};
```

- Now we have spherical coordinates.
- The coordinates may not be accessed from outside the class anymore: they are **private**!

file `Vector.C`:

```
#include “Vector.h”
#include <cmath>
using namespace std;
Vector::Vector (double x_, double y_, double z_) {
  r = sqrt (pow (x_, 2) + pow (y_, 2)+pow (z_, 2));
  phi = atan2 (y_, x_);
  theta = (r > 0) ? acos (z_/r) : 0;
}
double Vector::length() {
    return r;
}
```

- Now the constructor is much more complicated.
- But calculating the length is easy!

Classes VI

```
#include “Vector.h”
#include <iostream>
using namespace std;


int main() {
  double x, y, z;
  cout << “Enter three vector components:”;
  cin >> x >> y >> z;
  Vector v (x, y, z);
  cout << “Length of this vector is “
       << v.length() << endl;
  Vector w = v;
  cout << “Length of vector w is “
       << w.length() << endl;
  return 0;
}
```

What has changed in our main program?

NOTHING! It still works!

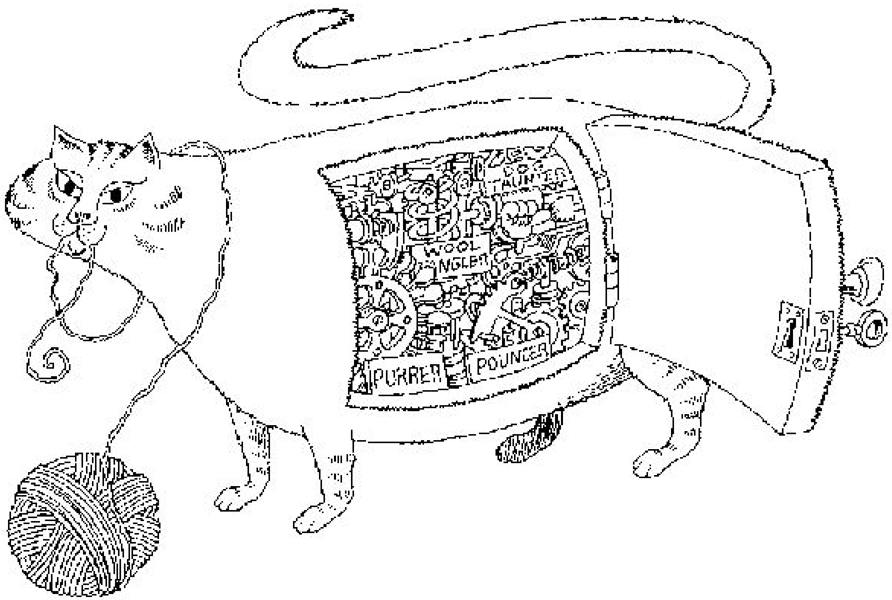
This is GREAT!

This concept is so great,it even has a name: It is called **Encapsulation**

Note: old routine calcVectorLength does not work anymore, because it accesses the data members of Vector directly!

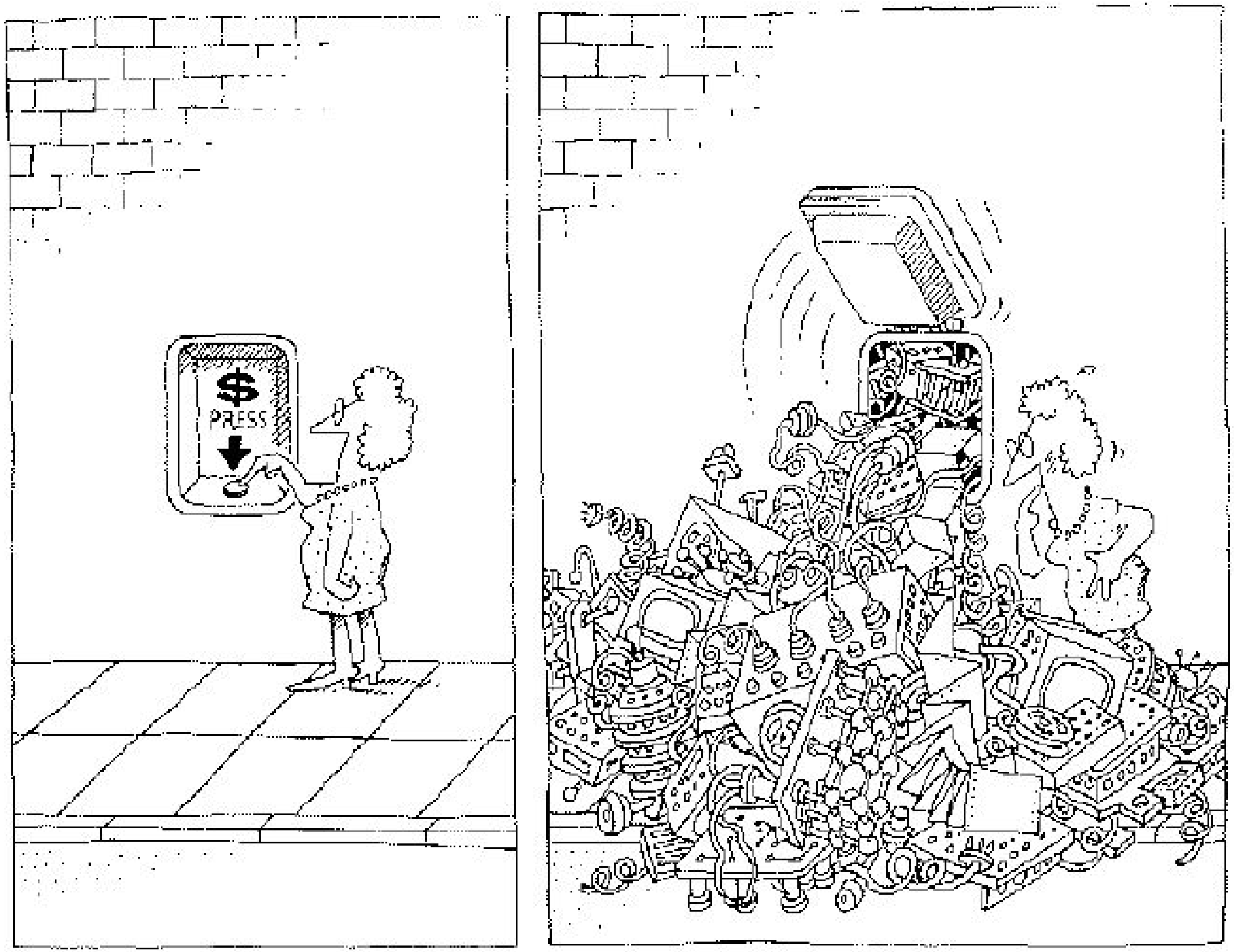
Reflection on Objects and Classes

- Objects: Instances of class variables:
  Vector is a class, v is an Obect
- With classes, we have
  - a close coupling between data and functions that work on the data
  - the possibility to hide ***how*** some piece of code works, we see only ***what*** it does
  - the possibility to divide our code into many small pieces that are individually simple and therefore well to maintain
- Object Oriented Programming is **the** modern way to write programs



Encapsulation hides the details of the implementation of an object.

The Illusion of Simplicity



The task of the software development team is to engineer the illusion of simplicity.

More on Compiling

- Compiler g++: Translates source code (text file) into machine code
- 2 Steps: Compiling and Linking
- Output of compiling step: .o files (object files):
  ```
  $> g++ -c Vector.C
  $> g++ -c vectorlength.C
  ```
  produces files `Vector.o` and `vectorlength.o`
- Output of linking step: executable (no extension)
  ```
  $> g++ -o vectorlength vectorlength.o Vector.o
  ```
  combines the object files `vectorlength.o` and `Vector.o` into the executable file `vectorlength`
- In the linking step, also source files may be used, e.g.
  ```
  $> g++ -o vectorlength vectorlength.C Vector.o
  ```

Archives

- Problem: If we have hundreds of object files, the linking commands gets veeeeeeeery long
- Solution: Collect all the object files (usually without object files that contain a `main()` function) in an archive

```
$> ar r libmyroutines.a Vector.o area.o
```

- Now file `libmyroutines.a` contains the files `Vector.o` and `area.o`; they can be listed with:

```
$> ar t libmyroutines.a
Vector.o
area.o
```

- We can use the archive in the linking step:

```
$> g++ -o vectorlength vectorlength.C libmyroutines.a
```

- Alternatively:

```
$> g++ -o vectorlength vectorlength.C -L. -lmyroutines
```

Recompilation

- Second Problem: If we have hundreds of source files and object files, re-compilation of all routines can take a lot of time
- But if we change `Vector.C`, why should we recompile `area.C`? This is unnecessary!
- Solution: we recompile only Vector.C and replace it in the archive:

```
$> g++ -c Vector.C
$> ar r Vector.o libmyroutines.a
```

  The "r" option (without a "-") tells ar to replace `Vector.o` in `libmyroutine.a`

make

•Third Problem: After an editing session, I may have changed 7 out of 150 .C files. It is very tedious to find out which files to recompile and to do it by hand. **Solution: The make utility**

file Makefile:

```
OBJS=Vector.o area.o
libmyroutines.a: $(OBJS)
	ar r libmyroutines.a $(OBJS)
.C.o:
	g++ -c $< $(CFLAGS)
vectorlength: vectorlength.C libmyroutines.a
	g++ -o vectorlength vectorlength.C
	     -L. -lmyroutines
Vector.o: Vector.h
area.o: area.h
```

`OBJS` is a variable that contains the name of the object files we want to have in the library.

This line says that `libmyroutines.a` depends on all object files. If any of the object files has changed (is newer than `libmyroutines.a`), the library has to be recreated.

This line say how to recreate libmyroutines.a. Note that the command has to be preceeded by a “tab” character, which can be very clumsy to enter in some editors! (`^I` sometines works)

This is a “suffix rule”: It tells make how to make a .C file into an .o file. $< stands for the .C file.

This line says that `Vector.o` also depends on `Vector.h` , not only on `Vector.C`

- Now we can enter in the shell:

```
$> make vectorlength
g++ -c Vector.C
g++ -c area.C
ar r libmyroutines.a Vector.o area.o
g++ -o vectorlength vectorlength.C -L. -lmyroutines
$>
```

Getters and Setters

```
class Vector {
  public:
    Vector (double x_, double y_, double z_);
    double length() const;
    double getX() const;
    double getY() const;
    double getZ() const;
    void setX (double newx);
  private:
    double r, phi, theta;
};
```

This "const" means that getX() does not change the Vector object. We'll hear more about that later.

```
Vector::getX() const {
   return r*cos(phi)*sin(theta);
}

Vector setX (double newx) {
  double newy = getY();
  double newz = getZ();
  r = sqrt (newx*newx + newy*newy + newz*newz);
  phi = atan2 (newy, newx);
  theta = (r > 0) ? acos (newz/r) : 0;
}
```

By using "Getter" and "Setter" methods instead of allowing direct access to the data members, we "decouple" the class `Vector` from its "clients", i.e. from the code that uses `Vector` objects.

If we now want to go back to a Vector representation which internally uses x, y, z, we have to change **only** code in the files `Vector.h` and `Vector.C`. The potentially hundreds of files in which we use `Vector` objects can stay unchanged!

A more complicated class: Particle

file Particle.h:

```
#include “Vector.h”

class Particle {
  public:
    Particle();
    Particle (Vector v_, double m_);
    Vector getMomentum() const;
    double getEnergy() const;
    double getInvariantMass () const;
    double getInvariantMass (Particle p);
  private:
    double px, py, pz, m, e;
};
```

– This is called the “default constructor”

– invariant mass of particle itself
– invariant mass of combination with another particle

Note: we can have several functions with the same name, but different arguments, that do different things!
(This is forbidden in C!)
This is called ***(function) overloading***.

Several Particles: Arrays

Problem: in general, we have several particles in an event

file particlearray.C:

```
#include “Vector.h”
#include “Particle.h”
#include “fillParticles.h”
#include <iostream>
using namespace std;

int main() {
  Particle allParticles[100];
  int n = fillParticles (allParticles);

  for (int i = 0; i < n; ++i) {
    for (int j = i+1; j < n; ++j) {
      cout << “Invariant mass of particles “ << i
           << “ and “ << j << “ is “
           << allParticles[i].getInvariantMass (allParticles[j])
           << endl;
    }
  }
}
```

`allParticles` is an array with 100 Particles.

`fillParticles` somehow fills the array, and returns the number of particles.

Indices start at 0 in C++!

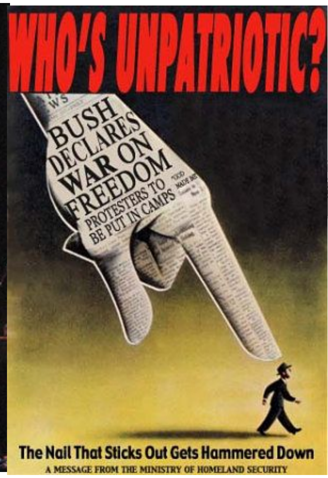
For an array with 100 elements, valid index values are 0 to 99.

Pointers

- A Pointer points to some object anywhere in memory: It contains only the object's memry address, but knows to what kind (class) of object it points to
- We can use this to refer to other objects
- Example: Decay K^0_S -> $\pi^+\pi^-$: we want to point to the 2 possible decay pions, and we may have several pion pairs sharing the same pion candidate

An english pointer

The Pointer Sisters



Another Pointer

"For God's sake, Edwards, put the laser pointer away!"

Pointers can be dangerous!!!

Example: A K0S class

```
#include "Particle.h"

class K0SParticle {
  public:
    K0SParticle (Particle *piplus_, Particle *piminus_);
    getInvariantMass() const;

  private:
    Particle *piplus;
    Particle *piminus;
};

K0SParticle::K0SParticle (Particle *piplus_, Particle *piminus_) {
  piplus = piplus_;
  piminus = piminus_;
}

K0SParticle::getInvariantMass() const {
  return (*piplus).getInvariantMass (*piminus);
}
```

`piplus` is a pointer to a Particle object.
Read: "`*piplus` is a `Particle`".

pointers can be copied without copying the object to which they point

`*piplus` is the object itself.

Using the Kshort class

```
#include “Vector.h”
#include “Particle.h”
#include “K0SParticle.h”
#include <iostream>
using namespace std;

int main() {
  Particle allParticles[100];
  int n = fillParticles (allParticles[100]);

  for (int i = 0; i < n; ++i) {
    for (int j = i+1; j < n; ++j) {
      K0SParticle k0s (&(allParticles[i]), &(allParticles[j]));
      cout << “Invariant mass of K0S is “
           << k0s.getInvariantMass() << endl;
    }
  }
}
```

`k0s` is created here.

`k0s` is destroyed here! (“it goes out of scope”)

Critique:

- How can we store our good K0S candidates? We don't know how many we will get!
- A K0S is also a Particle. It also has similar functions, like `getInvariantMass()`. Can we somehow unify Particle and K0SParticle?

Storing the Kshort Candidates

```
int main() {
  Particle allParticles[100];
  int n = fillParticles (allParticles);
  K0SParticle *allKshorts[10000];

  for (int i = 0; i < 10000; ++i) allKshorts[i] = 0;
  int k0sNumber = 0;
  K0SParticle *k0s;

  for (int i = 0; i < n; ++i) {
    for (int j = i+1; j < n; ++j) {
      k0s = new K0SParticle(&(allParticles[i]), &(allParticles[j]));
      if (abs (k0s->getInvariantMass() - 0.493) < 0.05) {
        allKshorts[k0sNumber] = k0s;
        ++k0sNumber;
      }
      else {
        delete k0s;
      }
    }
  }
  cout << “We have found “ << k0sNumber << “ Kshort candidates.\n”;
}
```

A ***new*** K0SParticle is created here, `k0s` points to it.

We keep the good Kshort candidates

Note: `k02->getInvariantMass()` is just shorthand for `(*k02).getInvariantMass()`

...and throw away the bad Kshort candidates!

A K0SParticle is also a Particle

```
#include "Particle.h"

class K0SParticle: public Particle {
  public:
    K0SParticle (Particle *piplus_, Particle *piminus_);
    getInvariantMass();

  private:
    Particle *piplus;
    Particle *piminus;
};
```

A This means that a K0SParticle is also a Particle.

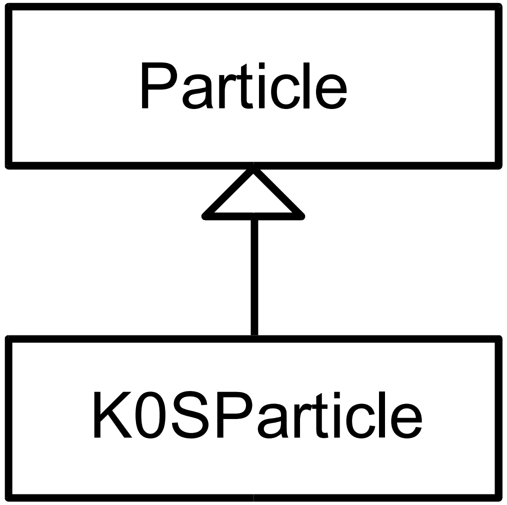
This is called **Inheritance**.

The class “Particle” is called the **base class** of class “K0SParticle”.

Class “K0SParticle” is a **subclass** of class “Particle”.
It “**inherits**” from class Particle, which is the **superclass**.

This is the “UML Diagram” for this relationship →

“UML” stands for “Unified Modeling Language”



Inheritance

```
class Particle {
  public:
    double getPt() { return sqrt(px*px+py*py); }
    double getPhi() { return atan2(py, px); }
    double getInvariantMass() { return sqrt (e*e-px*px-py*py-pz*pz); }
  protected:
    double e, px, py, pz;
};
```

"**protected**" means
"private, but may be accessed from subclasses".

```
class K0SParticle: public Particle {
  public:
    K0SParticle (Particle *piplus_, Particle *piminus_) {
      piplus = piplus_;
      piminus = piminus_;
      e = piplus->e + piminus->e;
      px = piplus->px + piminus->px;
      py = piplus->py + piminus->py;
      pz = piplus->pz + piminus->pz;
    }
  private:
    Particle *piplus;
    Particle *piminus;
};
```

Here we set the properties that are specific for a `K0SParticle`, and those inherited from `Particle`.

Class `K0SParticle` **inherits** `e, px, py, pz` from class `Particle`!

`K0SParticle` also **inherits** `getPt(), getPhi(), getInvariantMass()` from `Particle`!

Inheritance III

```
class Particle {
  public:
    virtual Particle *getDaughter (int i) {
      return 0;
    }
    //...
  protected:
    double e, px, py, pz;
};

class K0SParticle: public Particle {
  public:
     virtual Particle *getDaughter (int i) {
       if (i == 0) return pipus;
       else if (i == 1) return piminus;
       else return 0;
     }
    //...

  private:
    Particle *piplus;
    Particle *piminus;
};
```

A new keyword.
"`virtual`" means that a subclass may implement this method differently.

A more generic Particle:
a particle may have daughter particles into which it decays.
Normally, a particle has no daughters.

A `K0SParticle` has 2 daughters, 0 and 1. Therefore it **overrides** the method `getDaughter` from the base class.

A Simple Jet Class

```
class Jet: public Particle {
  public:
    Jet() {
      ndaughters = 0;
    }
    virual void addParticle (Particle *newDaughter) {
      if (nDaughters >= 100) {
        cerr << “Jet::addParticle: too many daughters!\n”;
      }
      else {
        allDaughters[nDaughters++] = newDaughter;
        e  += newDaughter->e;
        px += newDaughter->px;
        py += newDaughter->py;
        pz += newDaughter->pz;
      }
    }
    virtual Particle *getDaughter (int i) {
       return (i >= 0 && i < nDaughters) ? allDaughters[i] : 0;
     }
  protected:
    int nDaughters;
    Particle *allDaughters[100];
};
```

A simple class for jets; jets are composed of particles, but may also be treated as a pseudo-particle (e.g. a quark!)

Typical C/C++: Doing 2 things at the same time: assigning to `allDaughters[nDaughters]`, incrementing `nDaughters` afterwards.

This is an array of pointers to Particles. Uff!

Using the Jet Class: A Jet Algorithm (à la JADE)

```
int findJets (Particle *particles[], int nParticles, double ycut, double s) {
  int imin, jmin;
  while (nParticles > 1) {
   double mmin = sqrt (s);
   for (int i = 0; i < nParticles; ++i) {
     for (int j = i+1; j < nParticles; ++j) {
       double m = particles[i]->getInvariantMass (particles[j]);
       if (m < mmin) {
         mmin = m; imin = i; jmin = j;
       }
     }
   }
   if (mmin*mmin < ycut*s) {
     Jet *jet = new Jet;
     jet->addParticle (particles[imin]);
     jet->addParticle (particles[jmin]);
     particles[jmin] = particles[--nParticles];
     particles[imin] = jet;
  }
   else break;
  }
  return nParticles;
}
```

Loop over all pairs of particles,
find the pair with the least invariant mass.
For this pair, store the indices i and j.

Combine particles imin and jmin into a new jet;
remove both particles from the list of particles:
- replace particle imin by the new jet,
- replace particle jmin by last particle in the list,
- decrease the number of particles by 1.

← This is the trick!
Because a `Jet` is also a `Particle`,
we may use it wherever a `Particle` is needed!

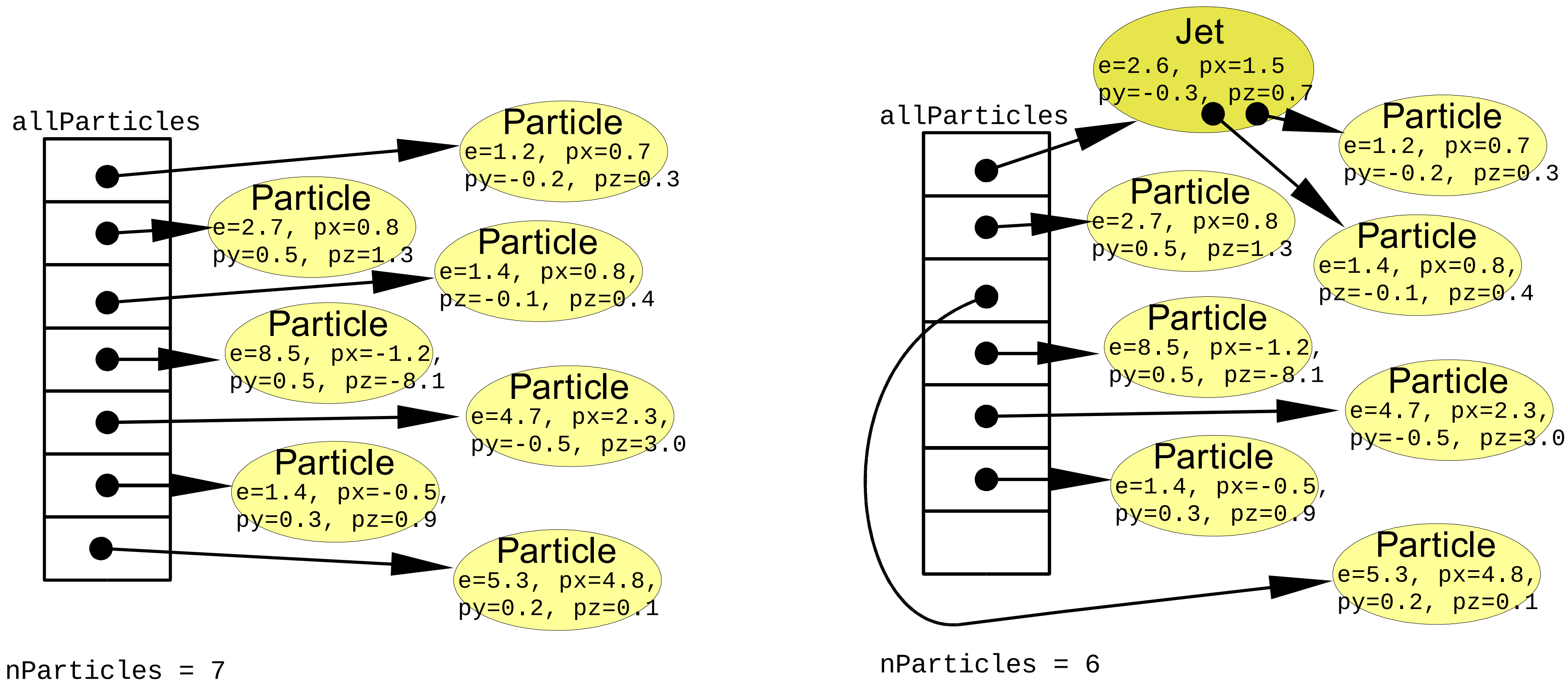
Reflection

- We just saw great things a work:
  One object behaving like an object from a different class!
- A Jet **IsA** special sort of Particle:
  `class Jet: public Particle {...};`
- Therefore, wherever a `Particle` is needed, I can use a `Jet`!
- But a `Jet` also contains more information than an ordinary `Particle`, e.g. the number of `Particle`s that it is composed of.
- What happens to this additional information?

```
Jet *jet = new Jet;
Particle *part = jet;
Jet jetCopy = *jet;
Particle partCopy = *jet;
```

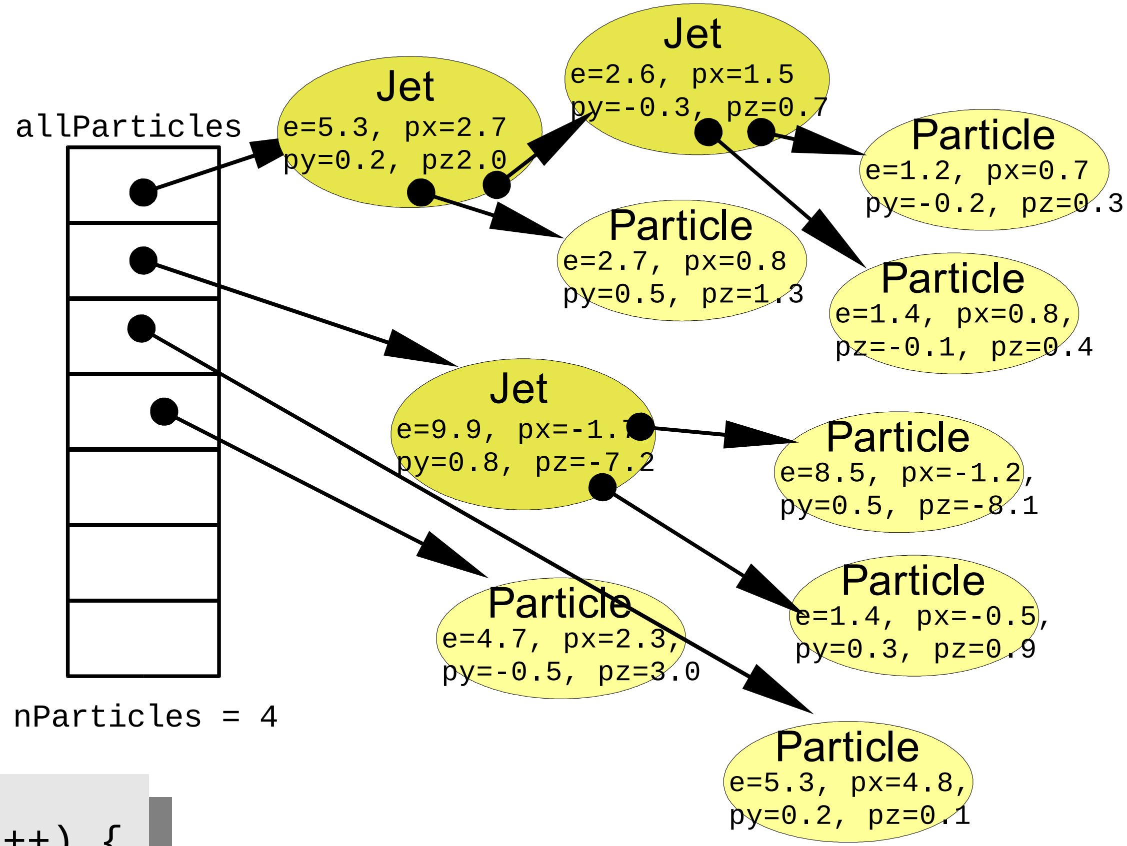
A pointer to a newly created Jet object
Another pointer, pointing to this object
A copy of the Jet object, with all the information
A copy of the Particle info of the Jet, i.e. only e, px, py, pz

The Jet Algorithm at Work



Destructors

- After the Jet finder: a complicated tree.
- All the objects use memory
- If we want to run the the jet finder on many events, we have to free the memory again!




```
class Jet: public Particle {
  public:
    ....
    virtual ~Jet();
};
```

```
Jett::~Jet() {
  for (int i = 0; i < nDaughters; i++) {
    delete allDaughters[i];
  }
}
```

~Jet() is the Destructor of class Jet.
It is called when a variable of class Jet goes out of scope,
or when we explicitly delete an objet of class Jet
which a pointer points to.
The destructor is used to "clean up".

Passing Arguments to Subroutines

- Normal case in C/C++: "**Pass by Value**":
  - Only the value of a variable is passed to a subroutine
  - For objects: a **copy** is passed
  - If we change the object, only a copy is changed => no effect for calling routine!
  - If we pass an object of a subclass (Jet/Particle!), we lose information

```
Jet *jet = new Jet;
Particle *part = jet;
Jet jetCopy = *jet;
Particle partCopy = *jet;
```

- To pass "the object itself", we can pass a pointer to the object:
  - the value of the pointer is the the address of the object
  - the pointer is copied, i.e. the address, but not the object pointed to!

```
Jet *jet = new Jet;
Particle *part = jet;
Jet jetCopy = *jet;
Particle partCopy = *jet;
```

References

- Passing pointers is completely OK, but leads to clumsy notation:

```
void sort (double *d1, double *d2) {
  if (*d2 > *d1) {
    double d = *d1;
    *d1 = *d2;
    *d2 = d;
  }
}
```

```
int main() {
  double a = 2.3;
  double b = 5;
  sort (&a, &b);
  cout << “After sorting: “ << a “ <= “ b << endl;
}
```

- A reference is another name for an object:

```
int main() {
  double a = 2.3;
  double b = 5;
  double& c = a;
  a = 7.5;
  cout << “Value of c: “ << c << endl;
}
```

References II

- With references, our sort function looks much nicer:

```
void sort (double& d1, double& d2) {
  if (d2 > d1) {
    double d = d1;
    d1 = d2;
    d2 = d;
  }
}
```

```
int main() {
  double a = 2.3;
  double b = 5;
  sort (a, b);
  cout << “After sorting: “ << a “ <= “ b << endl;
}
```

- References don't exist in C, only in C++
- Passing a reference is essentially like passing a pointer, but nicer:
  - No copying is involved
  - The reference behaves like the object itself

const

- A function that takes a reference to an object can in principle change the object
- Very often, we want to write functions that only “look” at an object, i.e. get some properties of the object, but do not change the object.
- If we use “const”, we promise not to change the object:

```
double scalarProduct (const Vector& v1, const Vector& v2) {
  return v1.getX()*v2.getX()
       + v1.getY()*v2.getY()
       + v1.getZ()*v2.getZ();
}
```

- But how do we know that getX() does not change the Vector?

```
class Vector {
  public:
    ...
    double getX() const;
};
```

The “const” tells the compiler that getX() may be used for constant objects. It is a promise that getX() will not change the object.

```
double Vector::getX() const {
  return r*cos(phi)*sin(theta);
}
```

In the implementation file, the compiler will report an error if we try to do anything that changes the object, e.g. write

```
r = 1.7;
```

Things we Have not Covered

- operator overloading
- templates
- the standard template library
- much much more...

I'll try to give you a flavour about these things in the next slides.

These things are very useful, but not trivial to use, because we have not covered many technical details in this 2 day boot camp.

But let's see...

A Flavour of Templates

file maximum.h:

```
template<class T>
T maximum (const T& a, const T& b) {
  return (a > b) ? a : b;
}
```

This defines a generic “maximum” function for any data type T that has a “>” operator.
Note that the complete definition is in the header file, there is no .C file!

file trymaximum.C:

```
#include<iostram>
using namespace std;
#include “maximum.h”

int main() {
  double d1, d2;
  cout << “Enter two floating point numbers: “;
  cin >> d1 >> d2;
  cout << “The maximum of “ << d1 << “ and “
       << d2 << “ is “ << maximum (d1, d2) << endl;
  int i1, i2;
  cout << “Enter two integer numbers: “;
  cin >> i1 >> i2;
  cout << “The maximum of “ << i1 << “ and “
       << i2 << “ is “ << maximum (i1, i2) << endl;
  return 0;
}
```

Here we use the new maximum function:

The compiler automatically creates a maximum function from the template that takes two doubles and returns a double.

The compiler automatically creates a different maximum function that takes two integers and returns an integer!

A Flavour of Operator Overloading

file Vector.h:

```
class Vector {
  public:
    ...
    double getX() const;
    double getY() const;
    double getZ() const;
};

Vector operator+ (const Vector& lhs, const Vector& rhs);
```

Here we declare the “+” operator for two Vectors.

file Vector.C:

```
double Vector::getX() const { return r*cos(phi)*sin(theta); }
Vector operator+ (const Vector& lhs, const Vector& rhs) {
  double x = lhs.getX() + rhs.getX();
  double y = lhs.getY() + rhs.getY();
  double z = lhs.getZ() + rhs.getZ();
  return Vector (x, y, z);
}
```

The access functions are simple.

The “+” operator is also straightforward

Now we can write:

```
  Vector v1 (1, 2, 3), v2 (-0.5, 2.3, 0);
  Vector w = v1 + v2;
```

A Flavour of the STL

- STL: Standard Template Library

file numbervector.C:

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

int main() {
  int n;
  cout << “Enter the number of elements: “;
  cin >> n;
  vector<double> allNumbers(n);
  for (int i = 0; i < n; i++) {
    cout << “Enter number “ << i+1 << “: ”;
    cin >> allNumbers[i];
  }
  sort (allNumbers.begin(), allNumbers.end());
  cout << “Here are all numbers in order: \n(”;
  for (int i = 0; i < allNumbers.size()-1; i++) {
    cout << allNumbers[i] << “, “;
  }
  cout << allNumbers[allNumbers.size()-1] << “)\n”;
  return 0;
}
```

`vector<T>` is a template type.
It stores elements of type T. Here T is a double.
Here we create a vector with n elements.

The vector behaves like an array, but it can be copied, resized, sorted etc etc.

Here we sort the vector.

The vector knows its own size! Very useful...

Reserve

RESERVE

Operators I: Arithmetic operators

- Arithmetic operators:

Operator	Meaning	FORTRAN
-	Sign Change	-
*	Multiplication	*
/	Division	/
%	Modulus	MOD
+	Addition	+
-	Subtraction	-

note: no exponentiation (** in FORTRAN)! use "pow" function

- Assignment: = evaluates right side, assigns value to left side

```
double radius = 1.5;
double result = 3.14159276*radius*radius;
int i = 1;
i = i + 1;   // now i is 2!
```

Operators III: Relational Operators

- Relational (comparison) operators: return “false” or “true”

Operator	Meaning	FORTRAN
`==`	Equal	`.EQ.`
`!=`	Not equal	`.NE.`
`<`	less than	`.LT.`
`<=`	less or equal	`.LE.`
`>`	greater than	`.GT.`
`>=`	greater or equal	`.GE.`

- Careful: “==” is a comparison, “=” is an assignment!
- In C/C++, assignment has also a value: the assigned value:
  `a = (b = 7) + 1;` is legal (b becomes 7, a becomes 8)
- Therefore: `if (a=7)...` is also legal, but not what you want!

Operators IV: Logical Operators

- Logical operators: used for boolean expressions

Operator	Meaning	FORTRAN
!	not	`.NOT.`
!=	exclusive or	`.XOR.`
&&	and	`.AND.`
||	or	`.OR.`

- Bitwise operators: Perform bit-by-bit operations on integer types

Operator	Meaning	FORTRAN
~	complement	`INOT`
&	bitwise and	`IAND`
^	bitwise exclusive or	`IEOR`
|	bitwise or	`IOR`

- Careful! Don't confuse logical and bitwise operators!
  integers can be converted to bool: 0 is false, everything else is true
  => 7 && 8 is true, 7 & 8 is 0 is false!

Numerical Functions

- Available from `<cmath>`
  Don't forget "`using namespace std;`"!

Function	Meaning	FORTRAN	Remark
`sin(x)`	Sine	`SIN(X)`	
`cos (x)`	Cosine	`COS(X)`	
`tan (x)`	Tangent	`TAN(X)`	
`asin(x)`	Arc sine	`ASIN(X)`	
`acos(x)`	Arc cosine	`ACOS(X)`	
`atan(x)`	Arc tangent	`ATAN(X)`	$-\pi/2 < \text{Result} < \pi/2$
`atan2(x,y)`	Arc tangent (x/y)	`ATAN2 (X, Y)`	$-\pi < \text{Result} < \pi$
`exp(x)`	Exponential	`EXP(X)`	
`log(x)`	Natural logarithm	`LOG(X)`	
`log10(x)`	Logarithm, base 10	`LOG10(X)`	
`abs(x)`	Absolute value	`ABS(X)`	
`sqrt(x)`	Square root	`SQRT(X)`	
`pow (x, y)`	x to the power y	`X**Y`	only for x >= 0
`pow (x, i)`	x to the integer power	`X**I`	also for x<0