A Very Short tour of C++

Day 2

Problems are hardly ever posed directly in terms of computer intrinsic types – `int, float` etc. Instead we talk about tracks, lines, points, detectors, and so on.
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. This makes the approach lot more intuitive and appealing. In addition, formalisms in C++ allows convenient handling of large codes, developed by a number of developers.

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

<u>Object-Oriented Programming</u>:
Classic "procedural" programming languages before C++ (such as C) often focused on the question "What should the program do next?" The way you structure a program in these languages is:
Split it up into a set of tasks and subtasks
Make functions for the tasks
Instruct the computer to perform them in sequence

With large amounts of data and / or large numbers of tasks, this approach leads to complex programs that are difficult to maintain.
Consider the task of modeling the operation of a car. Such a program would have lots of separate variables storing information on various car parts, and there'd be no way to group together all the code that relates to, say, the wheels. You can have number of variables grouped together in `structures`, but there is no way to group the functionalities related to these data. It's hard to keep all these variables and the connections between all the functions in mind.

To manage this complexity, it's nicer to package up self-sufficient, modular pieces of code. People think of the world in terms of *interacting objects*: we'd talk about interactions between the steering wheel, the pedals, the wheels, etc. OOP allows programmers to pack away details into neat, self-contained boxes (objects) so that they can think of the objects more abstractly and focus on the interactions between them.

There are lots of definitions for OOP, but 3 primary features of it are:

Encapsulation: grouping related data and functions together as objects and defining an interface to those objects

Inheritance: allowing code to be reused between related types

Polymorphism: allowing a value to be one of several types, and determining at runtime which functions to call on it based on its type.

One way to think about what happens in an object-oriented program is that we define what objects exist and what each one knows, and then the objects send messages to each other (by calling each other's methods) to exchange information and tell each other what to do.

Encapsulation just refers to packaging related stuff together. We'll shortly see how to package up data and the operations it supports in C++: with classes.
If someone hands us a class, we do not need to know how it actually works to use it; all we need to know about is its public methods/data – its interface. This is often compared to operating a car: when you drive, you don't care how the steering wheel makes the wheels turn; you just care that the interface the car presents (the steering wheel) allows you to accomplish your goal. If you think of the analogy about objects being boxes with buttons you can push, you can also think of the interface of a class as the set of buttons each instance of that class makes available. Interfaces abstract away the details of how all the operations are actually performed, allowing the programmer to focus on how objects will use each other's interfaces – how they interact.
This is why C++ makes you specify `public` and `private` access specifiers: by default, it assumes that the things you define in a class are internal details which someone using your code should not have to worry about. The practice of hiding away these details from client code is called "data hiding," or making your class a "black box."

Inheritance allows us to define hierarchies of related classes.
Imagine we're writing an inventory program for vehicles, including cars and trucks. We could write one class for representing cars and an unrelated one for representing trucks, but we'd have to duplicate the functionality that all vehicles have in common. Instead, C++ allows us to specify the common code in a `Vehicle` class, and then specify that the `Car` and `Truck` classes share this code.
The `Vehicle` class can be as follows:

```
class Vehicle
{
protected:
  string license ;
  int year;
public:
  Vehicle( const string &myLicense, const int myYear)
  :license(myLicense), year(myYear) {}
  const string getDesc() const
  {return license + " from " + stringify(year);}
  const string &getLicense() const {return license;}
  const int getYear() const {return year;}
};
```

A few notes on this code:

1) The standard `string` class was briefly touched upon in yesterday's class. Recall that `string`s can be appended to each other with the `+` operator (similar to the `strcat` function). `protected` is largely equivalent to `private`. We'll discuss the differences shortly.

2) The code demonstrates member initializer syntax. When defining a constructor, you sometimes want to initialize certain members, particularly `const` members, even before the constructor body. You simply put a colon before the function body, followed by a comma-separated list of items of the form `dataMember(initialValue)`.

3) Assumes the existence of some function `stringify` for converting numbers to `string`s (something like `sprintf` in `C`).

Now we want to specify that `Car` will inherit the `Vehicle` code, but with some additions. This is accomplished in line 1 below:

```
class Car : public Vehicle
{ // Makes Car inherit from Vehicle
string style;

public :
  Car( const string &myLicense, const int myYear, const string &myStyle)
  : Vehicle(myLicense, myYear), style(myStyle) {}
  const string &getStyle() {return style;}
};
```

Now class `Car` has all the data members and methods of `Vehicle`, as well as a `style` data member and a `getStyle` method.

Class `Car` *inherits* from class `Vehicle`. This is equivalent to saying that `Car` is a *derived class*, while `Vehicle` is its *base class*. You may also hear the terms *subclass* and *superclass* instead.

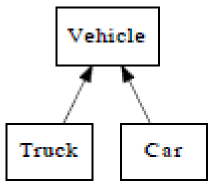
Notes on the code:

1) Don't worry for now about why we stuck the `public` keyword in there.

2) Note how we use member initializer syntax to call the base-class constructor. We need to have a complete `Vehicle` object constructed before we construct the components added in the `Car`.

3) If you do not explicitly call a base-class constructor using this syntax, the default base-class constructor will be called.

Similarly, we could make a `Truck` class that inherits from `Vehicle` and shares its code. This would give a class hierarchy like the following:



Class hierarchies are generally drawn with arrows pointing from derived classes to base classes.

Is-a vs. Has-a:
There are two ways we could describe some class `A` as depending on some other class `B`:
Every `A` object *has a* `B` object. For instance, every `Vehicle` *has a* `string` object(called `license`).
Every instance of `A` *is a* `B` instance. For instance, every `Car` *is a* `Vehicle`, as well.

Inheritance allows us to define "is-a" relationships, but it should not be used to implement "has-a" relationships. It would be a design error to make `Vehicle` inherit from `string` because every `Vehicle` has a license; a `Vehicle` is not a `string`. "has-a" relationships should be implemented by declaring data members, not by inheritance.

Overriding Methods:
We might want to generate the description for `Car`s in a different way from generic `Vehicle`s. To accomplish this, we can simply redefine the `getDesc` method in `Car`, as below. Then, when we call `getDesc` on a `Car` object, it will use the redefined function. Redefining in this manner is called *overriding* the function.

```
class Car : public Vehicle
{
string style ;
public :
  Car( const string &myLicense, const int myYear, const string &myStyle)
  : Vehicle(myLicense, myYear), style(myStyle) {}

  const string getDesc() // Overriding this member function
  {
  return stringify(year) + ’ ’ + style + ": " + license;
}

  const string &getStyle() {return style;}
};
```

Programming by Difference:
In defining derived classes, we only need to specify what's different about them from their base classes. This powerful technique is called *programming by difference*.
Inheritance allows only overriding methods and adding new members and methods. We cannot remove functionality that was present in the base class.

Access Modifiers and Inheritance:
If we'd declared `year` and `license` as `private` in `Vehicle`, we wouldn't be able to access them even from a derived class like `Car`. To allow derived classes but not outside code to access data members and member functions, we must declare them as `protected`.

The `public` keyword used in specifying a base class (e.g., `class Car : public Vehicle {...}`)gives a limit for the visibility of the inherited methods in the derived class. Normally you should just use `public` here, which means that inherited methods declared as `public` are still `public` in the derived class. Specifying `protected` would make inherited methods, even those declared `public`, have at most `protected` visibility.

Polymorphism:

Polymorphism means "many shapes." It refers to the ability of one object to have many types. If we have a function that expects a `Vehicle` object, we can safely pass it a `Car` object, because every `Car` is also a `Vehicle`. Likewise for references and pointers: anywhere you can use a `Vehicle *`, you can use a `Car *`.

`virtual` Functions:

There is still a problem. Take the following example:

```
Car c("Vanity", 2003);
Vehicle *vPtr = &c;
Cout << vPtr->getDesc();
```

(The `->` notation on line 3 just dereferences and gets a member. `ptr->member` is equivalent to `(*ptr).member`.)

Because `vPtr` is declared as a `Vehicle *`, this will call the `Vehicle` version of `getDesc`, even though the object pointed to is actually a `Car`. Usually we'd want the program to select the correct function at runtime based on which kind of object is pointed to. We can get this behavior by adding the keyword `virtual` before the method definition:

```
class Vehicle
{
...
  virtual const string getDesc()
  {
  ...
  }
};
```

With this definition, the code above would correctly select the `Car` version of `getDesc`.

Selecting the correct function at runtime is called *dynamic dispatch*. This matches the whole OOP idea – we're sending a message to the object and letting it figure out for itself what actions that message actually means it should take.

Because references are implicitly using pointers, the same issues apply to references:

```
Car c("Vanity", 2003);
Vehicle &v = c;
cout << v.getDesc();
```

This will only call the `Car` version of `getDesc` if `getDesc` is declared as `virtual`.

Once a method is declared `virtual` in some class `C`, it is virtual in every derived class of `C`, even if not explicitly declared as such. However, it is a good idea to declare it as `virtual` in the derived classes anyway for clarity.

Pure `virtual` Functions:

Arguably, there is no reasonable way to define `getDesc` for a generic `Vehicle` –only derived classes really need a definition of it, since there is no such thing as a generic vehicle that isn't also a car, truck, or the like. Still, we do want to require every derived class of `Vehicle` to have this function.

We can omit the definition of `getDesc` from `Vehicle` by making the function *pure virtual* via the following odd syntax:

```
class Vehicle {
...
virtual const string getDesc() = 0; // Pure virtual
};
```

The `=0` indicates that no definition will be given. This implies that one can no longer create an instance of `Vehicle`; one can only create instances of `Car`s, `Truck`s, and other derived classes which do implement the `getDesc` method. `Vehicle` is then an abstract class – one which defines only an interface, but doesn't actually implement it, and therefore cannot be instantiated.

Multiple Inheritance:

Unlike many object-oriented languages, C++ allows a class to have multiple base classes:

```
class Car : public Vehicle , public InsuredItem
{
...
};
```

This specifies that `Car` should have all the members of both the `Vehicle` and the `InsuredItem` classes.

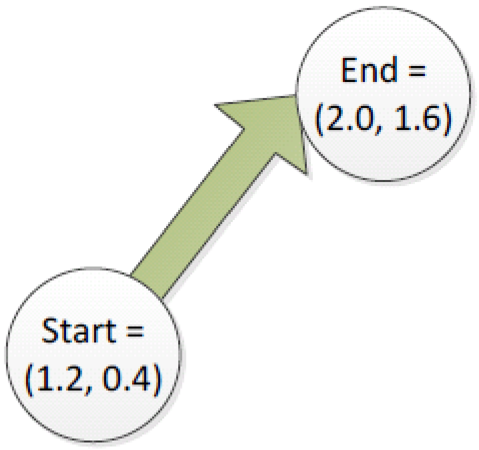
Multiple inheritance is tricky and potentially dangerous:

If both `Vehicle` and `InsuredItem` define a member `x`, you must remember to disambiguate which one you're referring to by saying `Vehicle::x` or `InsuredItem::x`.

If both `Vehicle` and `InsuredItem` inherited from the same base class, you'd end up with two instances of the base class within each `Car` (a "dreaded diamond" class hierarchy). There are ways to solve this problem, but it can get messy.

In general, avoid multiple inheritance unless you know exactly what you're doing.

Consider a two-dimensional vector
• In the context of geometry, a vector consists of 2 points: a start and a finish
• Each point itself has an x and y coordinate



<u>Structures in C</u>
```
typedef struct
  {
  double x;
  double y;
  } Point2D;



typedef struct
  {
  Point2D Start;
  Point2D End;
  } Vector2D;
```

<u>Classes in C++</u>
A user-defined datatype which groups together related pieces of information.
```
class Point2D
  {
  public:
    double x;
    double y;
  };

class Vector2D
  {
  public:
    Point2D Start, End;
  };
```
This indicates that the new data-types we're defining are called `Point2D` and `Vector2D`
Fields indicate what related pieces of information our data-type consists of – another word for *field* is *member*

Unlike in an array, different member of a class / structure can be of different type.
```
class SchoolStudent
```

```
  {
  public:
    char *name;
    int ID;
    float Score;
  };
```

An *instance* is an occurrence of a class.
Different *instances* can have their own set of values in their *fields*.

If you wanted to represent 2 different students (who can have different names and IDs), you would use 2 instances of `SchoolStudent`

```
int main(void)
{
SchoolStudent student1;
SchoolStudent student2;
}
```

Note that the two instances have only been declared, but not defined, i.e., we have not spelt out the details of `student1` and `student2`
To access fields of instances, use `variable.fieldName`

```
int main(void)
{
SchoolStudent student1;
SchoolStudent student2;

student1.name = "YourName"; student1.ID = 1; student1.Score = 100.0;
student2.name = "YourFriendsName"; student2.ID = 2; student2.Score = 100.0;
cout << "student1 name is" << student1.name << endl;
cout << "student1 id is" << student1.ID << endl;
cout << "student1 score is" << student1.Score << endl;
cout << "student2 name is" << student2.name << endl;
cout << "student2 id is" << student2. ID << endl;
cout << "student2 score is" << student2.Score << endl;
}
```

Going back to the vectors, we can have the following:

```
int main(void)
{
Vector vec1;
vec1.Start.x = 3.0;
vec1.Start.y = 4.0;
vec1.End.x = 5.0;
vec1.End.y = 6.0;

Vector vec2;
vec2.Start = vec1.Start;// Assign one instance to another to copy all fields
vec2.Start.x = 7.0;
}
```

Classes, naturally, can be passed to functions. As discussed yesterday, passing by value passes a copy of the class instance to the function and changes aren't preserved. If you want to reflect the changes made in the called function in the original calling function, you need to use references:

```
class Point2D
{
public: double x, y;
};

class Vector2D
{
public: Point2D start, end;
};

void offsetVector(Vector &v, double offsetX, double offsetY)
{
v.start.x += offsetX;
v.end.x += offsetX;
v.start.y += offsetY;
v.end.y += offsetY;
}

void printVector(Vector v)
{
cout << "(" << v.start.x << "," << v.start.y << ") -> (" << v.end.x <<
"," << v.end.y << ")" << endl;
}

int main(void)
{
Vector2D vec;

vec.start.x = 1.2; vec.end.x = 2.0; vec.start.y = 0.4; vec.end.y = 1.6;
offsetVector(vec, 1.0, 1.5);
printVector(vec); // (2.2,1.9) -> (3.8,4.3)
}
```

Observe how some functions are closely associated with a particular class. This leads us to *methods*: functions which are part of a class. As an analogy, we can say that *method*s are "buttons" on each box (instance), which do things when pressed

```
Vector2D vec1, vec2;
vec1.start.x = 1.2; vec1.end.x = 2.0;
vec1.start.y = 0.4; vec1.end.y = 1.6;
vec1.start.x = 1.2; vec1.end.x = 2.0;
vec1.start.y = 0.4; vec1.end.y = 1.6;

vec1.print();      // which instance (box), which method (button)
vec2.offset(1.0, 1.5);  // instance called vec2, methods called offset
```

Methods implicitly pass the current instance.

An example implementation can be as follows:

```
class Vector2D
{
public:
  Point2D start, end;
  void offset(double offsetX, double offsetY)
    {
    start.offset(offsetX, offsetY);
    end.offset(offsetX, offsetY);
    }
  void print()
    {
    start.print();
    cout << " -> ";
    end.print();
    cout << endl;
    }
};
class Point2D
{
public:
  double x, y;
  void offset(double offsetX, double offsetY)
  {
  x += offsetX; y += offsetY;
  }
  void print()
  {
  cout << "(" << x << "," << y << ")";
  }
};
```

A better implementation would be splitting the declarations and definitions separate. First, the header file:

```
// vector.h - header file
class Point2D
{
public:
  double x, y;
  void offset(double offsetX, double offsetY);
  void print();
};
class Vector2D
{
public:
  Point2D start, end;
  void offset(double offsetX, double offsetY);
  void print();
};
```

Then, the source code containing implementation of methods:

```
#include "vector.h"
// vector.cpp - method implementation
void Point2D::offset(double offsetX, double offsetY)
{
x += offsetX; y += offsetY;
}
void Point2D::print()
{
cout << "(" << x << "," << y << ")";
}
void Vector2D::offset(double offsetX, double offsetY)
{
start.offset(offsetX, offsetY);
end.offset(offsetX, offsetY);
}
void Vector2D::print()
{
start.print();
cout << " -> ";
end.print();
cout << endl;
}
```

As you can expect, manually initializing the fields can be extremely tedious. Here comes the role of constructors.

<u>Constructors:</u>
Methods that are called automatically when an instance of a class is created.

```
class Point2D
{
public:
  double x, y;
  Point2D()
  {
  x = 0.0; y = 0.0; cout << "Point instance created" << endl;
  }
};

int main(void)
{
Point2D p; // Point instance created
// p.x is 0.0, p.y is 0.0
}
```

Constructors can also accept parameters:

```
class Point2D
{
public:
  double x, y;
  Point2D(double nx, double ny)
  {
  x = nx; y = ny; cout << "2-parameter constructor" << endl;
  }
};

int main(void)
{
Point2D p(2.0, 3.0); // 2-parameter constructor
// p.x is 2.0, p.y is 3.0
}
```

Naturally, with the blessings of Function Name Overloading, we can have multiple constructors in a given class (in fact, we almost always have):

```
class Point2D
{
public:
  double x, y;
  Point2D()
  {
  x = 0.0; y = 0.0; cout << "default constructor" << endl;
  }
  Point2D(double nx, double ny)
  {
  x = nx; y = ny; cout << "2-parameter constructor" << endl;
  }
};

int main(void)
```

```
{
Point2D p; // default constructor
// p.x is 0.0, p.y is 0.0)
Point2D q(2.0, 3.0); // 2-parameter constructor
// q.x is 2.0, q.y is 3.0)
}
```

We have unknowingly used another constructor that is provided by the compiler as soon as a class is created – the default copy constructor that assigns the state of source instance onto the destination instance by copying all fields:

```
class Point2D
{
public:
  double x, y;
  Point2D()
  {
  x = 0.0; y = 0.0; cout << "default constructor" << endl;
  }
  Point2D(double nx, double ny)
  {
  x = nx; y = ny; cout << "2-parameter constructor" << endl;
  }
};

int main(void)
{
Point2D q(1.0, 2.0); // 2-parameter constructor
Point2D r = q; // r.x is 1.0, r.y is 2.0)
}
```

If for some reason, you do not want all the fields to be copied onto the destination instance, you can write your own copy constructor (which is possible with the default constructor, as well, in case you want to change the default behavior, or have a default despite having constructors with arguments). Consider the following amusing snippet:

```
class SchoolStudent
{
public:
  int studentID;
  char *name;
  SchoolStudent()
  {
  studentID = 0;
  name = "";
  }
};

int main(void)
{
SchoolStudent student1;
student1.studentID = 98;
char n[] = "foo";
student1.name = n;

SchoolStudent student2 = student1;
student2.name[0] = 'b';
cout << student1.name; // boo
}
```

As a remedy, we can write our own copy constructor:

```
class SchoolStudent
{
public:
  int studentID;
  char *name;
  SchoolStudent()
  {
  studentID = 0;
  name = "";
  }
  SchoolStudent(SchoolStudent &o)
  {
  studentID = o.studentID;
  name = strdup(o.name);// try “man strdup” to see what is the big difference
  }
};
```

An example involving two-dimensional points could be as follows (which, however, is equivalent to the default copy constructor, with an announcement)

```
class Point2D
{
public:
  double x, y;
  Point2D(double nx, double ny)
  {
  x = nx; y = ny; cout << "2-parameter constructor" << endl;
  }
  Point2D(Point &o)
  {
  x = o.x; y = o.y; cout << "custom copy constructor" << endl;
  }
};

int main(void)
{
Point2D q(1.0, 2.0); // 2-parameter constructor
Point2D r = q; // custom copy constructor
// r.x is 1, r.y is 2
}
```

Copy constructors are examples where we pass arguments by reference.

When making a class instance, the default constructor of its fields are invoked

```
class Integer
{
public:
  int val;
  Integer()
  {
  val = 0; cout << "Integer default constructor" << endl;
  }
};
class IntegerWrapper
{
public:
  Integer val;
  IntegerWrapper()
  {
  cout << "IntegerWrapper default constructor" << endl;
  }
};
int main()
{
IntegerWrapper q;
}
```

The output will be as follows:

```
Integer default constructor
IntegerWrapper default constructor
```

If a constructor with parameters is defined, the default constructor is no longer available. That can be very inconvenient while declaring arrays. The way out is to (i) create a separate 0-argument constructor, or, (ii) use default arguments

```
class Integer
{
public:
  int val;
  Integer(int v = 0)
  {
  val = v;
  }
};
int main()
{
Integer i; // ok
Integer j(3); // ok
}
```

If a method and an argument in a class have the same name, we need to use `this`, which is a pointer to the current instance (`this` is of type pointer to the relevant class).

```
class Integer
{
public:
  int val;
  Integer(int val = 0)
  {
  this->val = val;
  }
  void setVal(int val)
  {
  this->val = val;
  }
};
```

One copy of the code class is shared by all instances of the class and, thus, only one copy of the member functions is available. So, how does the correct method for a given object gets invoked? Once again, the hidden argument `this` takes care. It translates as a call to the member function of the class in question with the first argument as pointer to the specific object

Access modifiers:

Define where your fields/methods can be accessed from

`public`: can be accessed from anywhere

`private`: can only be accessed within the class

Use `getters` and `setters` to allow read and write accesses to private fields

```
class Point2D
{
private:
  double x, y;
public:
  Point(double nx, double ny)
  {
  x = nx; y = ny;
  }
  double getX() { return x; }
  double setX(double a) {x = a;}
  double getY() { return y; }
  double setY(double b) {y = b};
};

int main(void)
{
Point2D p(2.0,3.0);

// p.x = 5.0;      // not allowed
p.setX(5.0);       // ok

// cout << p.x << endl; // not allowed

cout << p.getX() << endl; // allowed
}
```

`struct`: `public` by default

`class`: `private` by default

Allocation of memory

We have already discussed ways of allocating memory through declaration of variables. Here is a little more detail:

Whenever we declare a new variable (int x), memory is allocated
When can this memory be freed up (so it can be used to store other variables)? The memory is freed when the variable goes out of scope. Thus, when a variable goes out of scope, that memory is no longer guaranteed to store the variable's value

The `new` operator
Another way to allocate memory, where the memory will remain allocated until you manually de-allocate it. The `new` operator returns a pointer to the newly allocated memory

```
int *x = new int;
```

Terminology note:
–If using `int x;` the allocation occurs on a region of memory called *the stack*
–If using `new int;` the allocation occurs on a region of memory called *the heap*

The `delete` operator de-allocates memory that was previously allocated using `new`. It takes a pointer to the memory location as the argument.

```
int *x = new int; // use memory allocated by new; x is a pointer to int!
delete x;   // delete memory allocated by new
```

As discussed before, allocating arrays of variable sizes and freeing them can be very conveniently done with these operators:
If we use `new[]` to allocate arrays, they can have variable size, and we can de-allocate such arrays with `delete[]`

```
int numItems;
cout << "how many items?";
cin >> numItems;

int *arr = new int[numItems];
…

delete[] arr;
```

It is important to note that `new` can be used to create new instances of a class. It invokes the appropriate constructors. On the other hand, `delete` can be used to free the memory allocated by `new`.

```
class Point
{
public:
  int x, y;
  Point(int nx, int ny)
  {
```

```
  x = nx;
  y = ny;
  cout << "2-arg constructor" << endl;
  }
};

int main()
{
Point *p = new Point(2, 4);

delete p;
}
```

The output will be as follows:

```
2-arg constructor
```

Destructors

Destructor is called when the class instance gets de-allocated

(1) If stack-allocated, when it goes out of scope

```
class Point
{
public:
  int x, y;
  Point()
  {
  cout << "constructor invoked" << endl;
  }
  ~Point()
  {
  cout << "destructor invoked" << endl;
  }
};

int main()
{
if (true)
  {
  Point p;
  }

cout << "p out of scope" << endl;
}
```

Output:

```
constructor invoked
destructor invoked
p out of scope
```

(2) If allocated with `new`, when `delete` is called

```
class Point
{
public:
  int x, y;
  Point()
  {
  cout << "constructor invoked" << endl;
  }
  ~Point()
  {
  cout << "destructor invoked" << endl;
  }
};

int main()
{
Point *p = new Point;
delete p;
}
```

Output:

```
constructor invoked
destructor invoked
```

Let us now consider the Three Vector Class available in the Class Library for High Energy Physics (CLHEP) in bits and pieces.

The declaration (simplified) of this class is as follows:

```
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;
};
```

As you can see, there are three constructors. They are implemented as follows:

```
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(){
}
```

Note that

- `Foo::bar()` says that `bar()` is a member function of the class `Foo`
- `::` is the *scope resolution operator*
- Copy constructor uses a `const` reference

Inline:

In the `Hep3Vector` class, the data members `dx, dy, dz` are private and we need `getters` and `setters`, as dicussed above. Usually, small functions such as these `getters` and `setters` are made `inline` and put in the header file declaring the class. Consider the following declaration:

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

The implementation for the methods could be:

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

This approach, however, is inefficient. The following is preferred in which the declaration itself is modified:

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

Points to be noted:
• 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
• program could be faster
• program could be larger

<u>Member initializers</u>:
There are a couple of approaches:
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 `{`. You can use the semicolon to initialize a member when a method is called. It is usually used to initialize constants in the constructor. You can use the semicolon to initialize a member when a method is called. This makes sure that all members have a correct value assigned to them, in the most efficient way. They allow better exception handling too. It is advisable to use them, and initialize all your member variables explicitly this way.
• `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

Operator Overloading:

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) )
```

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

Essentially all operators can be used for user defined types except `"."` , `".*"` , `"::"` , `"sizeof"` and `"?:"`. However, it is not possible to overload operators for built-in types.

The Complete List:

Constructors
```
Hep3Vector(double x=0.0, double y=0.0, double z=0.0);
Hep3Vector(const Hep3Vector &);
```
• also contains conversion constructor

Destructor
```
~Hep3Vector();
```
• invoked when object is deleted

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;
```

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;
```

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);
```