A vector class

We will develop a vector class to illustrate some aspects of C++ programming and in some lab exercises. Use of the vector class and C++ I/O facilities in your ray tracer is *entirely optional*. In contrast to lights, materials, and visible objects we will see that O-O implementation is not very natural in many numerical applications and forces us into unnatural asymmetric implementations.

Our C++ vector class will differ from the array based approach we have been using. It is based upon the structured implementation we identified at the start of the course in which the (x, y, z) elements of the vector are explicitly identified as class member variables named x, y and z;

We put the class definition in file vec.h

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

class vec_t
{
  public:
    vec_t();
    vec_t(double, double, double);
  
  private:
    double x;
    double y;
    double z;
};
```

These two lines replace `#include <stdio.h>` when using C++ I/O

Default constructor will set vector to (0,0,0). Other one will set it to values provided
The vec_t constructors

As with the ray tracing routines, we put the implementations of the class methods in vec.cpp.

```cpp
#include "vec.h"

vec_t::vec_t()
{
    cerr << "default constructor" << endl;
    x = y = z = 0.0;
}

vec_t::vec_t(double xi, double yi, double zi)
{
    cerr << "3 parm constructor" << endl;
    x = xi;
    y = yi;
    z = zi;
}
```

The `cerr` function may be passed any number of items of types that it "knows about" separated by `<<` and they will be given a default format and sent to the standard error file. `cout` can be used to send output to `stdout`. The counterpart of these functions is:

```cpp
double d;
cin >> d;
```

It can be used to read a value into a type that it "knows about".

The `<<` and `>>` operators are used for bit shifting in standard C and are overloaded by the `ostream` and `istream` classes. We will see later on how to make `<<` and `>>` know about the vec_t class itself.

Vector operations

In the C++ version of the vector class, our old design can be made to work if we make `vec_sum()`, `vec_diff()`, and so on friends of vec_t. In that way they can remain standard C functions, but have access to the private values x, y, z of any instance of a vector class to which they hold a reference.

```cpp
class vec_t
{
    public:
        vec_t();
        vec_t(double, double, double);
        friend inline void vec_sum(vec_t *, vec_t *, vec_t *);
        friend inline void vec_diff(vec_t *, vec_t *, vec_t *);
    }
```

Note that the `friend` declaration is mandatory for this function to be able to access the private elements of the vec_t. But when declared `friend` we can transform our array based functions to class based in this straight forward way.

```cpp
void vec_sum(vec_t *v1, vec_t *v2, vec_t *v3)
{
    v3->x = v1->x + v2->x;
    v3->y = v1->y + v2->y;
    v3->z = v1->z + v2->z;
}
```
Reference parameters

Standard C supports only *pass by value*. With pass by value, when a structure name is used as a formal parameter/actual argument, the entire structure must be *copied onto the stack*. This becomes increasingly undesirable as structures increase in size. Therefore the standard C solution is to make the formal parameter and actual argument be *pointers to the structure type*. Doing this provides two benefits:

- Only a single word (the pointer) must be pushed onto the stack
- The called function is able to modify elements of the structure.

C++ supports an additional parameter passing technique known as *pass by reference*. Use of pass by reference is signified by the use of `& instead of *` in the formal parameters of the function prototype. In C++ *all* function prototypes must have been defined at the point of invocation so there is no ambiguity regarding what technique is in use.

A benefit of passing references instead of values is:

- Unlike a pointer, it is not possible to do arithmetic on a reference. Thus some of the more error prone aspects of C programming may be avoided.

```cpp
class vec_t
{
    public:
        vec_t();
        vec_t(double, double, double);
        friend void vec_sum(vec_t *, vec_t *, vec_t *);
        friend void vec_sum(vec_t &, vec_t &, vec_t &);
};
```
Implementing overloaded functions

Function overloading is the practice of providing multiple implementations of a function having a single name. Needless to say, excessive use of this technique can produce programs that are virtually indecipherable!

When a function name is overloaded, the implementation that is actually used is the one whose formal parameters match the actual arguments being passed by the caller in both number of arguments and type of argument. Therefore, it is mandatory that each implementation have distinguishable parameters.

The C++ compiler allows even standard C functions to be overloaded. We must provide a function body for every distinct parameterization we wish to allow.

class vec_t
{
   public:
      vec_t();
      vec_t(double, double, double);
      friend inline void vec_sum(vec_t *, vec_t *, vec_t *);
      friend inline void vec_sum(vec_t &, vec_t &, vec_t &);
};

In the body of the implementation the reference parameter is treated as a structure, and not a pointer to a structure.

void vec_sum(vec_t &v1, vec_t &v2, vec_t &v3)
{
   v3.x = v1.x + v2.x;
   v3.y = v1.y + v2.y;
   v3.z = v1.z + v2.z;
}

void vec_sum(vec_t *v1, vec_t *v2, vec_t *v3)
{
   v3->x = v1->x + v2->x;
   v3->y = v1->y + v2->y;
   v3->z = v1->z + v2->z;
}
**Invoking overloaded functions**

On the previous page we created two distinct versions of `vec_sum()`. Fortunately, they produce the same answer here, but there is no requirement that they do so. On this page we look at the problems of how to invoke them and which one gets invoked.

To use pass by reference *the actual argument must be an instance of the class* not a pointer to an instance of the class.

The instance of an overloaded function that is used is the one whose formal parameters (best) match the actual arguments. The matching is straightforward when instances of or pointers to classes are the parameters.

```cpp
int main()
{
    vec_t v1(1.0, 2.0, 3.0);
    vec_t v2(4.0, 5.0, 6.0);
    vec_t v3;
    vec_sum(&v1, &v2, &v3);
    v3.put();

    vec_sum(v1, v2, v3);
    v3.put();
}
```

The parameterized constructor is called here.
The default constructor is called here.
The pointer based implementation is invoked here.
The reference based implementation is invoked here.
A broken function:

We could attempt to add a third implementation of `vec_sum()` which passes parameters by value by copying them onto the stack.

```cpp
class vec_t
{
    public:
        vec_t();
        vec_t(double, double, double);
        friend inline void vec_sum(vec_t *, vec_t *, vec_t *);
        friend inline void vec_sum(vec_t , vec_t , vec_t);
        friend inline void vec_sum(vec_t &, vec_t &, vec_t &);
}
```

To call this version we would also have to use:

```cpp
vec_sum(v1, v2, v3);
```

This would produce the following compile time error. Because the calling sequence shown IS the correct way to invoke either of the bottom two prototypes the compiler can't distinguish which one you want.

```
main.cpp: In function 'int main()':
main.cpp:24: error: call of overloaded 'vec_sum(vec_t&, vec_t&, vec_t&)' is ambiguous
vec.h:68: note: candidates are: void vec_sum(vec_t&, vec_t&, vec_t&)
vec.h:75: note:                 void vec_sum(vec_t, vec_t, vec_t )
acad/cs102/examples/vec ==>
```

The compile time problem could be "fixed" by removing the implementation that used reference parameters and the resulting program would compile fine but just would not work.

*Exercise: Why not?*
A class based approach

We can also implement yet another instance of `vec_sum()` which is a true member function of the class.

At first glance this one looks a bit odd because `v1` seems to have disappeared!! This occurs because class methods are always invoked in the context of an instance of the class. In this case the instance will be `v1`.

```cpp
class vec_t
{
  public:
    vec_t();
    vec_t(double, double, double);
    void vec_sum(const vec_t &v2, vec_t &v3);
    friend inline void vec_sum(vec_t *, vec_t *, vec_t *);
    friend inline void vec_sum(vec_t &, vec_t &, vec_t &);
};
```

The implementation also looks somewhat asymmetric with `v2` and `v3` being explicitly accessed in contrast to the implicit access to `v1`.

```cpp
void vec_t::vec_sum(const vec_t &v2, vec_t &v3)
{
    v3.x = x + v2.x;
    v3.y = y + v2.y;
    v3.z = z + v2.z;
}
```

The asymmetry is also apparent in the invocation.

```cpp
v1.vec_sum(v2, v3);
v3.put();
```
A collection of possible implementations for vec_sum

C++ makes it possible (though not necessarily desirable) to provide implementations that match virtually any parameterization:

```cpp
void vec_sum(const vec_t &v2, vec_t &v3);
void vec_sum(const vec_t *v2, vec_t *v3);
void vec_sum(const vec_t *v2, vec_t &v3);
void vec_sum(const vec_t  v2, vec_t *v3);
```

Exercises: Identify any possible usable prototypes that we may have missed. Identify a prototype different from any of those above that would cause the compile time problem we saw previously.

Identify a prototype that would compile correctly but would not work. Each prototype must have a distinct implementation depending upon whether reference or value pointers are used.

```cpp
void vec_t::vec_sum(const vec_t &v2, vec_t &v3)
{
    v3.x = x + v2.x;
    v3.y = y + v2.y;
    v3.z = z + v2.z;
}

void vec_t::vec_sum(const vec_t *v2, vec_t *v3)
{
    v3->x = x + v2->x;
    v3->y = y + v2->y;
    v3->z = z + v2->z;
}

void vec_t::vec_sum(const vec_t *v2, vec_t &v3)
{
    v3.x = x + v2->x;
    v3.y = y + v2->y;
    v3.z = z + v2->z;
}

void vec_t::vec_sum(const vec_t v2, vec_t *v3)
{
    v3->x = x + v2.x;
    v3->y = y + v2.y;
    v3->z = z + v2.z;
}
```

This is the only place an entire vector must be copied onto the stack.
Implementations *returning* instances of *vec_t*

It is possible in *both* C and C++ to define instances of *vec_sum()* that *return* the answer. Unless you have a *real good* reason for doing so, this is generally a bad idea because in both languages it causes a copy on to and copy off of the stack operation. (In the C language, function overloading (the use of multiple implementations of the same function name) is also illegal).

Here the prototype is declared to *return* an instance of *vec_t*.

```c
vec_t vec_sum(const vec_t &v2);
```

The implementation requires a temporary variable in which the sum is computed. It is important to remember to *return*(tmp);

```c
vec_t vec_t::vec_sum(const vec_t &v2)
{
    vec_t tmp;
    tmp.x = x + v2.x;
    tmp.y = y + v2.y;
    tmp.z = z + v2.z;
    return(tmp);
}
```

To compute the result *v3* = *v2* + *v1* the following C++ code can be used. The "default" structure assignment mechanism takes care of copying the result off of the stack and into *v3*. This overhead is relatively minor for a vector but would be seriously bad for a structure containing a large array.

The syntax of the invocation is not very "natural"

```c
v3 = v2.vec_sum(v1);
```

when compared to the C function

```c
v3 = vec_sum(&v1, &v2);
```
Operator overloading in C++

Most operators can be overloaded. The ones that cannot are

.  .*  ::  ? :  and  sizeof

Operator overloading is actually part of the function overloading mechanism. To overload an operator you simply provide a function of the appropriate name. For example,

<table>
<thead>
<tr>
<th>Operator</th>
<th>Function name</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>operator+</td>
</tr>
<tr>
<td>-</td>
<td>operator-</td>
</tr>
<tr>
<td>*</td>
<td>operator*</td>
</tr>
<tr>
<td>&lt;=</td>
<td>operator&lt;=</td>
</tr>
<tr>
<td>-&gt;</td>
<td>operator-&gt;</td>
</tr>
</tbody>
</table>

Operator overloading is restricted to existing operators. Thus it is not legal to try to overload **

** operator**

The operator functions work "almost" exactly like "regular" functions. They can even be invoked using their operator+ or operator- names. Keeping this fact in mind will remove much of the mystery from how they work and how they must be implemented. The "almost" qualifier above reflects necessity to remember that almost all C operators take either one or two operands. Thus an operator function has at most two parameters.

The C addition operator takes two operands: \( a + b \)

Therefore the operator+ function will have two parameters: the first will represent the left side operand and the second the right side operand.

The C logical not operator takes one operand: \( !value \)

Therefore the operator! function will have one parameter and it will represent the right side operand.

Operator overloading must preserve the "aryness" (unary or binary) nature of the operator. Thus, the \( ! \) operator could be overloaded to compute the length of a single vector but could not be used to compute a dot product. The operators \&, \*, +, - have both binary and unary versions that may be overloaded separately.

It is not possible to change the precedence or associativity of an overloaded operator.

Most operator functions return a value that replaces the operator and its operand(s) in an expression. However, that is not mandatory. For example a side effect operator such as ++ may not need to return a value.

Like "regular functions" all operator functions may be defined as "regular C" friend functions of the class(es) to which their operands belong.

In some but not all cases they may be defined as class member functions instead.
Overloads as *friend* functions

We can write a version of the `vec_sum()` function that uses the + operator.

```cpp
class vec_t
{
    public:
        vec_t();
        vec_t(double, double, double);

        friend void vec_sum(vec_t *, vec_t *, vec_t *);
        friend void vec_sum(vec_t &, vec_t &, vec_t &);
        friend vec_t operator+(const vec_t &, const vec_t &);
};
```

The body of the function is written in exactly the same way that a `vec_sum()` function which returned the sum would be written.

```cpp
vec_t inline operator+(const vec_t &v1, const vec_t &v2)
{
    vec_t v3;
    v3.x = v1.x + v2.x;
    v3.y = v1.y + v2.y;
    v3.z = v1.z + v2.z;
    return(v3);
}
```

The function may then be invoked either by standard use of the + operator or by using its *operator+()* name.

```cpp
vec_t v1(1.0, 2.0, 3.0);
vec_t v2(4.0, 5.0, 6.0);
vec_t *vp = &v2;
vec_t v3;
```

```cpp
v3 = v1 + v2;
v3 = operator+(v1, v2);
```

If we try to invoke operator + as:

```cpp
v3 = v1 + vp;
```

main.cpp:21: error: no match for ‘operator+’ in ‘v1 + vp’
vec.h:98: note: candidates are: vec_t operator+(const vec_t &, const vec_t &)

This occurs because *vp* is a pointer. We can fix this in one of two ways.

1 – Write another instance of the operator+() function in which the second parameter is a `vec_t *`
2 – Just invoke the function as:

```cpp
v3 = v1 + *vp;
```
Overloads as *member* functions

Overloaded operators can be *member functions* instead of friend functions *if the left side operand is an instance of the class* or *if the right side operand of a unary operator such as \(!\) is a class member.*

```cpp
class vec_t
{
    : vec_t operator+(vec_t &rhs);
    vec_t operator+(vec_t *rhs);
    vec_t operator!(void);
    :
};
vec_t vec_t::operator+(vec_t &rhs)
{
    vec_t tmp;
    tmp.x = x + rhs.x;
    tmp.y = y + rhs.y;
    tmp.z = z + rhs.z;
    return(tmp);
}
vec_t vec_t::operator+(vec_t *rhs)
{
    vec_t tmp;
    tmp.x = x + rhs->x;
    tmp.y = y + rhs->y;
    tmp.z = z + rhs->z;
    return(tmp);
}

This demonstrates that it is possible to create a second implementation that takes a pointer to a vector instead of an instance of a vector. The invocation below actually works as intended, *but it is really ugly when you think about it!* *What sense does it make to add a vector and a pointer.*

v3 = v1 + vp;
```
Scaling a vector

In the scaling operation we want to multiply the components of a vector by a double precision value. Since the left side operand is conventionally the double precision value and not a vector the friend function method must be used. Note that it is possible to write the body of the friend function within the class definition. Also note that for this week’s lab assignment, this approach is not allowed.

```cpp
friend vec_t operator*(double val, const vec_t &rhs) {
    vec_t tmp;
    tmp.x = val * rhs.x;
    tmp.y = val * rhs.y;
    tmp.z = val * rhs.z;
    return(tmp);
}
```

We can also write a function that computes the component-wise product:

```cpp
friend vec_t operator*(const vec_t &lhs, const vec_t &rhs) {
    vec_t tmp;
    tmp.x = lhs.x * rhs.x;
    tmp.y = lhs.y * rhs.y;
    tmp.z = lhs.z * rhs.z;
    return(tmp);
}
```

The correct implementation will be chosen by the compiler depending upon the operands.

```cpp
v3 = 5.0 * v2;
v3 = v1 * v2;
```

The following, however, will generate a compile time error:

```cpp
v3 = v2 * 5.0;
```

```
main.cpp: In function ‘int main()’:
main.cpp:38: error: no match for ‘operator*’ in ‘v1 * 5.0e+0’
vec.h:54: note: candidates are: vec_t operator*(double, const vec_t&)
vec.h:63: note: vec_t operator*(const vec_t&, const vec_t&)
```
Further overloading of << and >>

We saw in last week's lab that the overloaded operators << and >> could be used to print and to read numeric and character string values to the stdout and stderr and from the stdin. True to form, they can be further overloaded to print and read a complete vec_t. We want to be able to do something like

```cpp
cout << v1;
```

where v1 is a vector. Since the left hand side is not a vec_t, we must use the friend function form. So in vec.h we include in the vec_t class definition:

```cpp
friend ostream &operator<<(ostream &out, const vec_t &pvec);
```

We provide the implementation in either vec.cpp or inline in the class definition.

```cpp
ostream &operator<<(ostream &out, const vec_t &pvec)
{
    out << pvec.x <<", " << pvec.y <<", " << pvec.z << endl;
    return(out);
}
```

This is the key to cascading application of the << operator.

Note that our new overload just uses the built in overload of << to output each component of the vector along with punctuation and a '\n'.
Cascading cout

Our implementation will also let us print an arbitrary number of vectors:

```cpp
cout << v1 << v2 << v3;
```

This seemingly magic behavior occurs because of two things:

- C++ evaluates a sequence of `<<` operations in *left to right order*
- The operator `<<` function *returns its left side operand as its result.*

So what REALLY happens is: `cout << v1` is evaluated and the value it returns (cout) replaces the operator and its operands in the larger expression leaving:

```cpp
cout << v2 << v3;
```

Then `cout << v2` is evaluated and the value it returns (cout) replaces the operator and its operands in the larger expression leaving:

```cpp
cout << v3;
```

Then `cout << v3` is evaluated and the value it returns is "dropped on the ground", because there is nothing to assign it to.