Classes

check out:

http://www.cplusplus.com/reference/memory/allocator/

http://www.cplusplus.com/reference/memory/allocator/allocate/

http://www.cplusplus.com/reference/memory/allocator/deallocate/

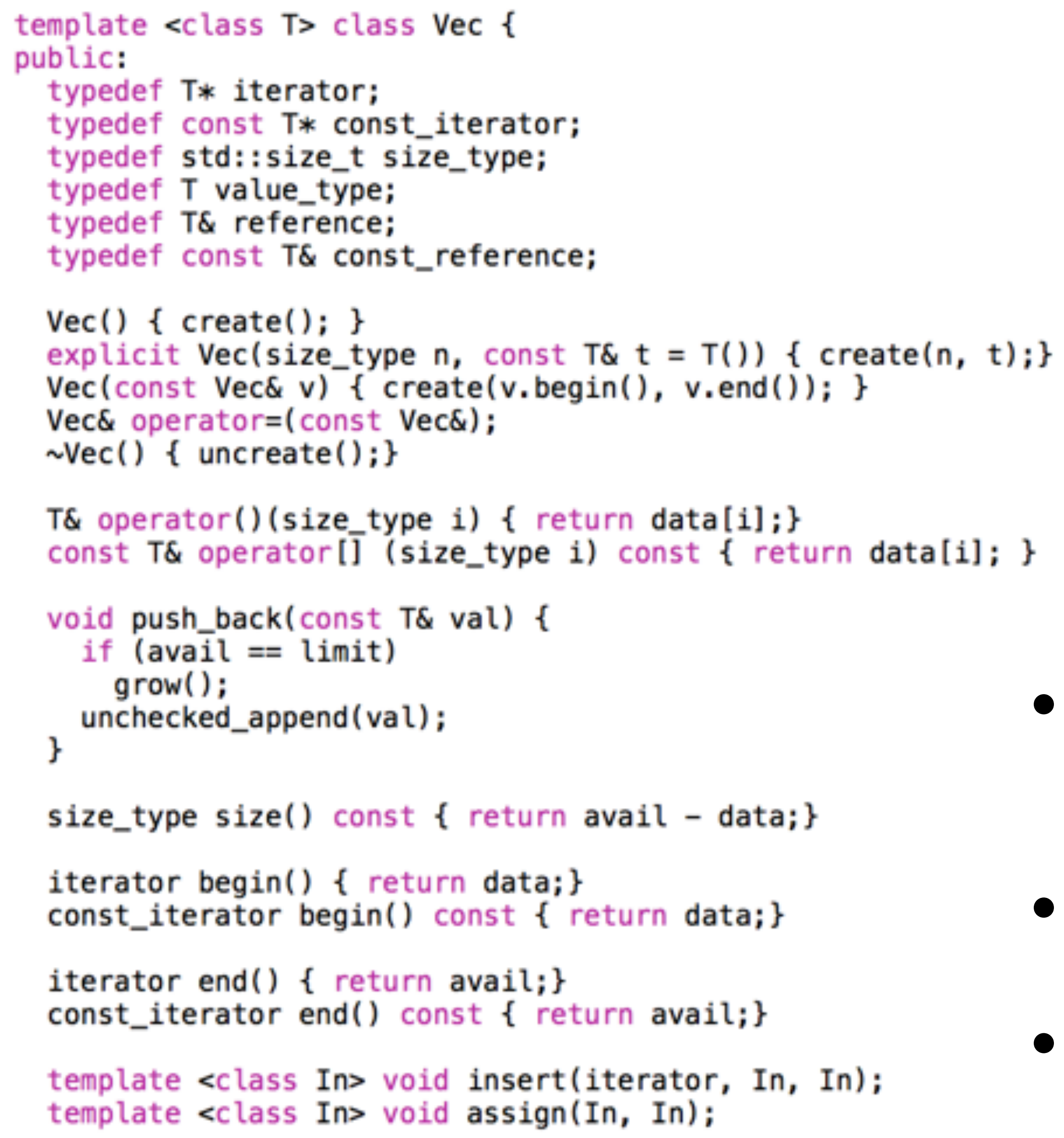
http://www.cplusplus.com/reference/memory/uninitialized_copy/

http://www.cplusplus.com/reference/memory/uninitialized_fill/

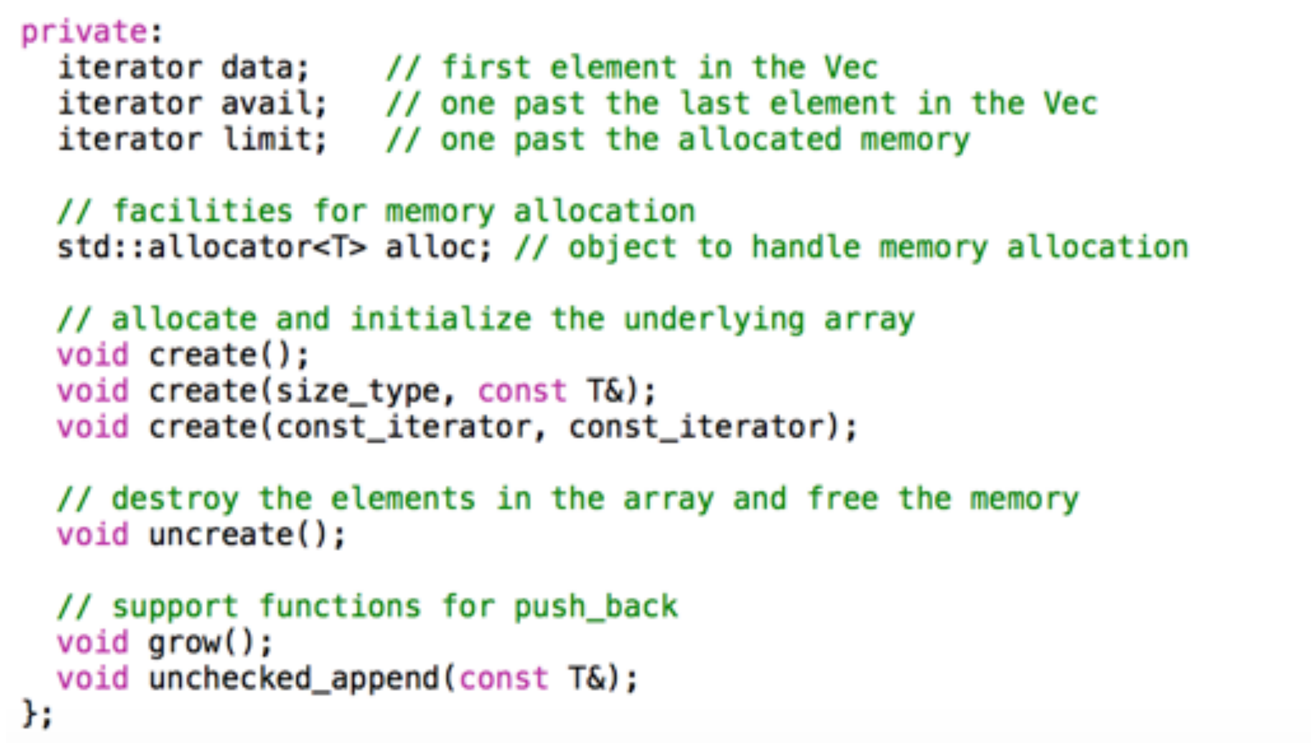
http://www.cplusplus.com/reference/memory/allocator/construct/

Class design

Defining abstract base classes is one of the key features of C++. Following [Accelerated C++] we will implement our own vector class



```
template <class T> class Vec {
public:
  typedef T* iterator;
  typedef const T* const_iterator;
  typedef std::size_t size_type;
  typedef T value_type;
  typedef T& reference;
  typedef const T& const_reference;

  Vec() { create(); }
  explicit Vec(size_type n, const T& t = T()) { create(n, t);}
  Vec(const Vec& v) { create(v.begin(), v.end()); }
  Vec& operator=(const Vec&);
  ~Vec() { uncreate();}

  T& operator()(size_type i) { return data[i];}
  const T& operator[] (size_type i) const { return data[i]; }

  void push_back(const T& val) {
    if (avail == limit)
      grow();
    unchecked_append(val);
  }

  size_type size() const { return avail - data;}

  iterator begin() { return data;}
  const_iterator begin() const { return data;}

  iterator end() { return avail;}
  const_iterator end() const { return avail;}

  template <class In> void insert(iterator, In, In);
  template <class In> void assign(In, In);
```



```
private:
  iterator data;    // first element in the Vec
  iterator avail;   // one past the last element in the Vec
  iterator limit;   // one past the allocated memory

  // facilities for memory allocation
  std::allocator<T> alloc; // object to handle memory allocation

  // allocate and initialize the underlying array
  void create();
  void create(size_type, const T&);
  void create(const_iterator, const_iterator);

  // destroy the elements in the array and free the memory
  void uncreate();

  // support functions for push_back
  void grow();
  void unchecked_append(const T&);
};
```

- default access to variables for class is private : only class members and member functions can access private data.
- A struct is the same as a class but with public default settings
- do not forget the semi-colon ; after the class definition
- following the STL, we defined the 6 typedefs

Class design

A class variable can de declared as

```
Vec<int> v;
```

Its public member functions (and variables) can be accessed as

```
v.push_back(10);
N = v.size();
```

A pointer to a Vec object can also be defined

```
Vec<int> * w;
```

Its public member functions (and variables) can be accessed as

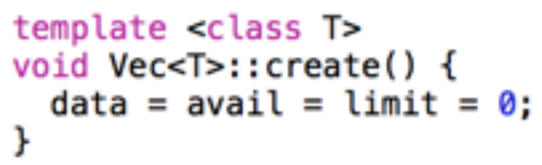
```
w->push_back(10);
w->size();
```

Certain functions such as create() were only declared in the class declaration. Their implementation can be written outside the class body with the scope operator ::

```
template <class T>
void Vec<T>::create () { /* implementation */ }
```

Default Constructor

The line *Vec();* is the default constructor: it takes no arguments. It is called whenever we write *Vec<int> v; It is incorrect to write Vec<int> vc()* because this would make *vc* a function declaration instead of an object declaration: a function that takes no arguments but returns a value of type Vec<int>.
There is no return type. In our example the constructor calls the private function *create()*:



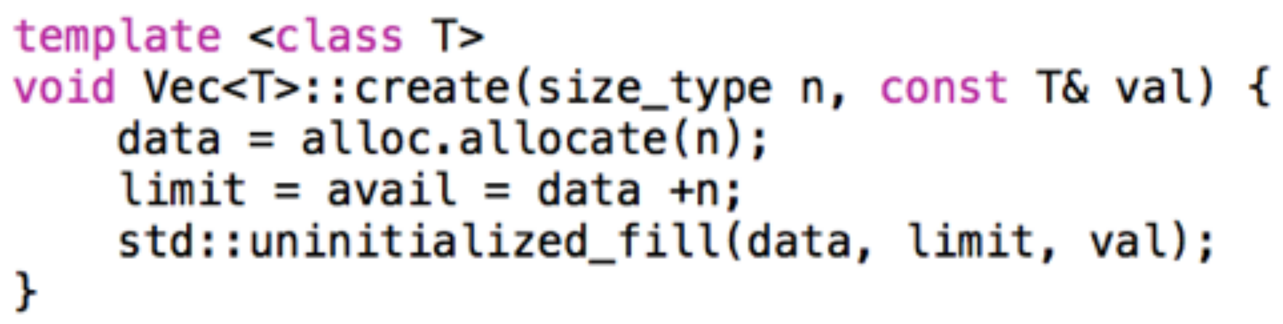
```
template <class T>
void Vec<T>::create() {
  data = avail = limit = 0;
}
```

This initializes (all) the (private) members of the class: the pointers are set to null pointers.

In case no constructor is provided, the compiler will automatically provide one. However, this can not initialize pointers or request dynamic memory etc.
In case specific constructors are provided but no default providers, then the compiler will not generate a default constructor. This could limit how this class can be used; for instance, declaring an array of such a class is more involved.

Specific Constructors

A second type of constructor is provided, thereby *overloading* the constructor:
explicit Vec(size_type n, const T& t = T()) { create (n ,t); }



```
template <class T>
void Vec<T>::create(size_type n, const T& val) {
    data = alloc.allocate(n);
    limit = avail = data +n;
    std::uninitialized_fill(data, limit, val);
}
```

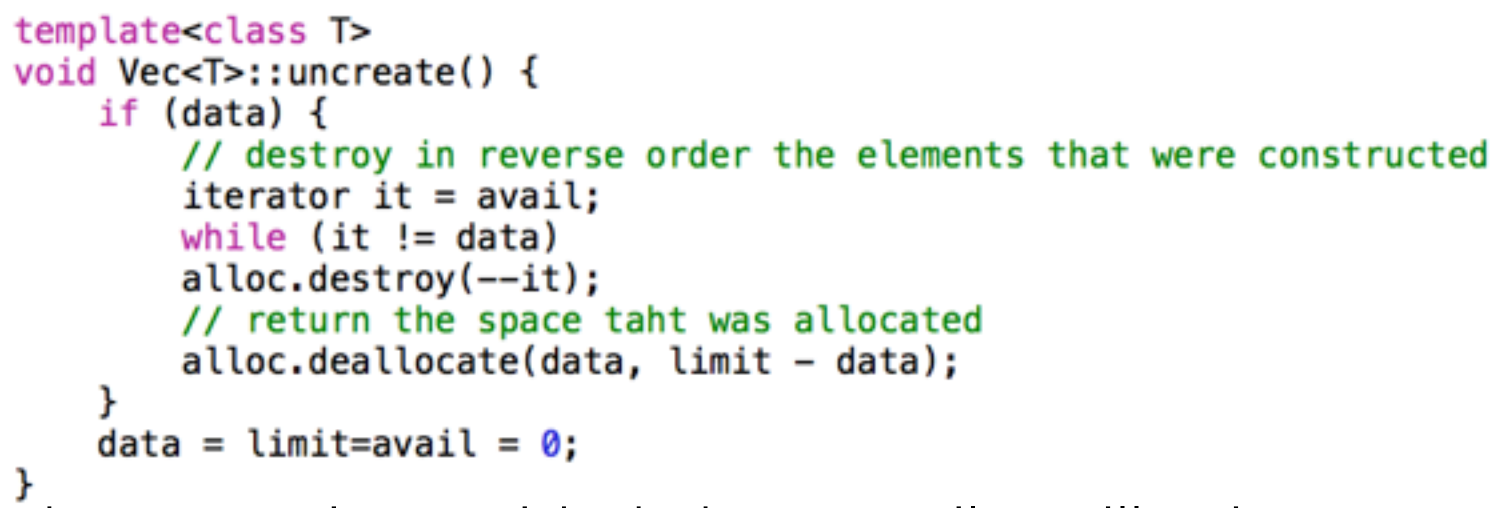
it allocates memory for *n* variables of type *T* and therefore sets the *limit* to this value. Since for a vector it is costly to reshape (when the vector is asked to contain more elements than its maximum size for instance), we try to minimize these operations by allowing *avail* to differ from *limit*. At the moment this is not necessary and *avail* is set equal to *limit*. Finally, the standard library function *uninitialized_fill* is called on the data.

the keyword explicit avoids automatic type conversions and is recommended for constructors with one of more arguments.

This specific version of the constructor will be called when writing *Vec<int> v(100, 1)*;
In this case memory for a 100 int variables is allocated and filled with value 1.

Destructor

When the Vec variable goes out of scope, its destructor is automatically called: *~Vec();* In this case, *~Vec() { uncreate();}* calls the function *uncreate()*



```
template<class T>
void Vec<T>::uncreate() {
    if (data) {
        // destroy in reverse order the elements that were constructed
        iterator it = avail;
        while (it != data)
        alloc.destroy(--it);
        // return the space taht was allocated
        alloc.deallocate(data, limit - data);
    }
    data = limit=avail = 0;
}
```

When no destructor is provided, the compiler will write one automatically (an empty function). However, for dynamic memory we must do the memory clean-up ourselves. In the example above, the destructor destroys the elements one by one in reverse order, deallocates its memory, and finally sets the data, limit and avail pointers back to the null_ptr.

Copy constructor

There is a third line without return type. It is the copy constructor

Vec(const Vec&);

which must take a const-ref to the same class as argument

The copy constructor is invoked whenever objects of the Vec class are copied.

This can occur when a new Vec object is initialized via an existing one

Vec<int> v3(v1);

which (surprisingly!) is identical to:

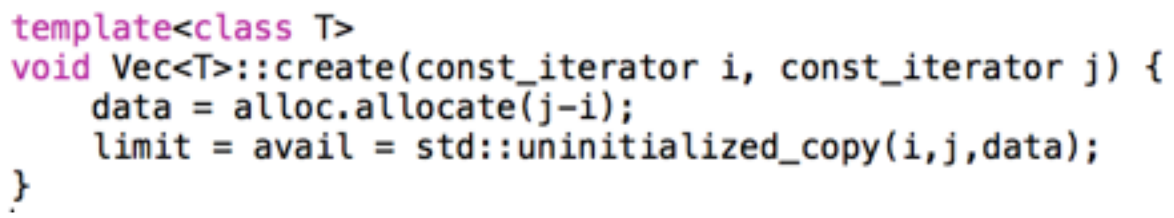
Vec<int> v3 = v1; *// no assignment !!!*

or when copies are made as a consequence of function calls or return arguments:

If no copy constructor is provided, the compiler will generate one automatically with member-wise copying

in our example the copy constructor is implemented as

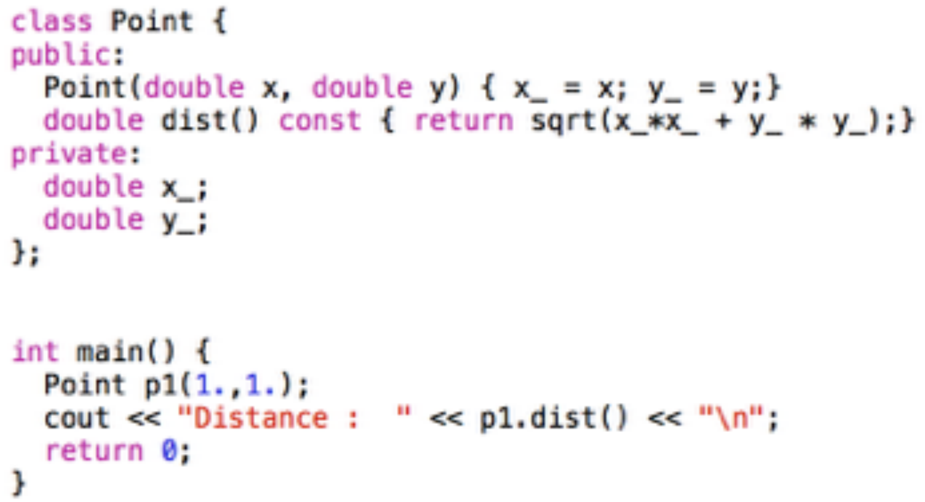
Vec(const Vec& v) { create(v.begin(), v.end());}



```
template<class T>
void Vec<T>::create(const_iterator i, const_iterator j) {
    data = alloc.allocate(j-i);
    limit = avail = std::uninitialized_copy(i,j,data);
}
```

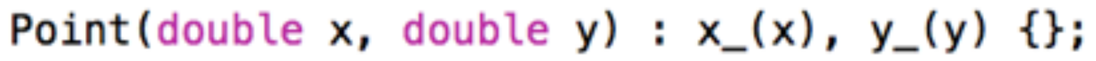
member initialization

We have seen the initialization in the constructor can be carried out as



```
class Point {
public:
  Point(double x, double y) { x_ = x; y_ = y;}
  double dist() const { return sqrt(x_*x_ + y_ * y_);}
private:
  double x_;
  double y_;
};


int main() {
  Point p1(1.,1.);
  cout << "Distance :  " << p1.dist() << "\n";
  return 0;
}
```

Another way of doing this is via *member initialization* : we can replace the constructor by



```
Point(double x, double y) : x_(x), y_(y) {};
```

Note the empty function body {} because all members are already initialized.

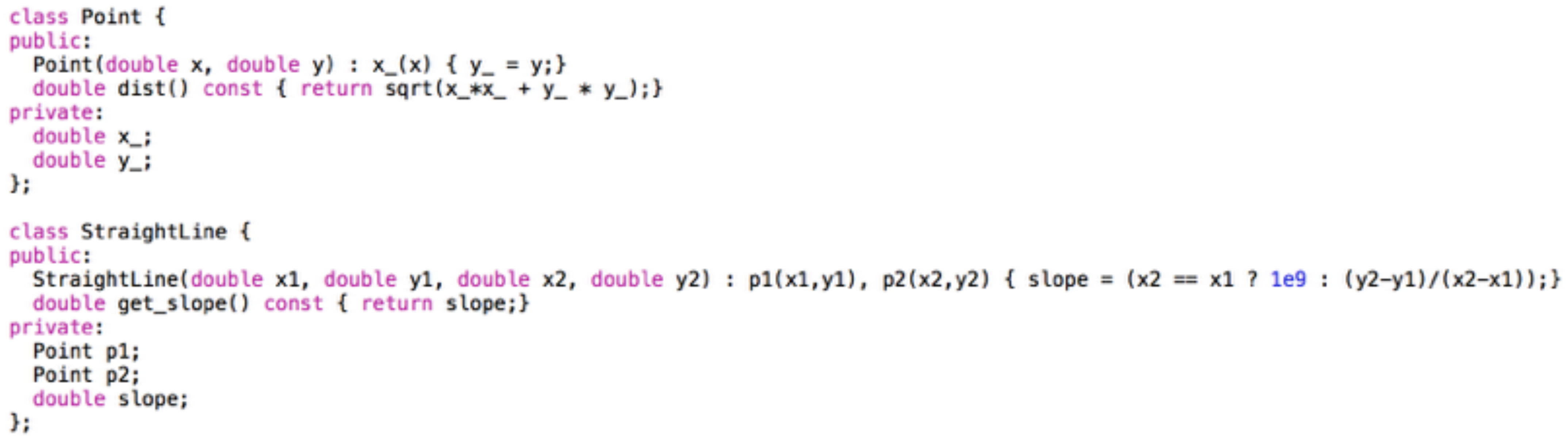
In case the variables after the colon (:) do not appear in the same way as they are declared in the function body, compilers may give a warning

For fundamental types both ways of initialization are equivalent, but for abstract types or more complex types (ie, your own classes) the difference in syntax can be useful: their default constructor might be called, but in case their default constructor does not exist, member initialization is a way to solve this problem.

Also when initializing const variables or references, this is the only possibility

member initialization

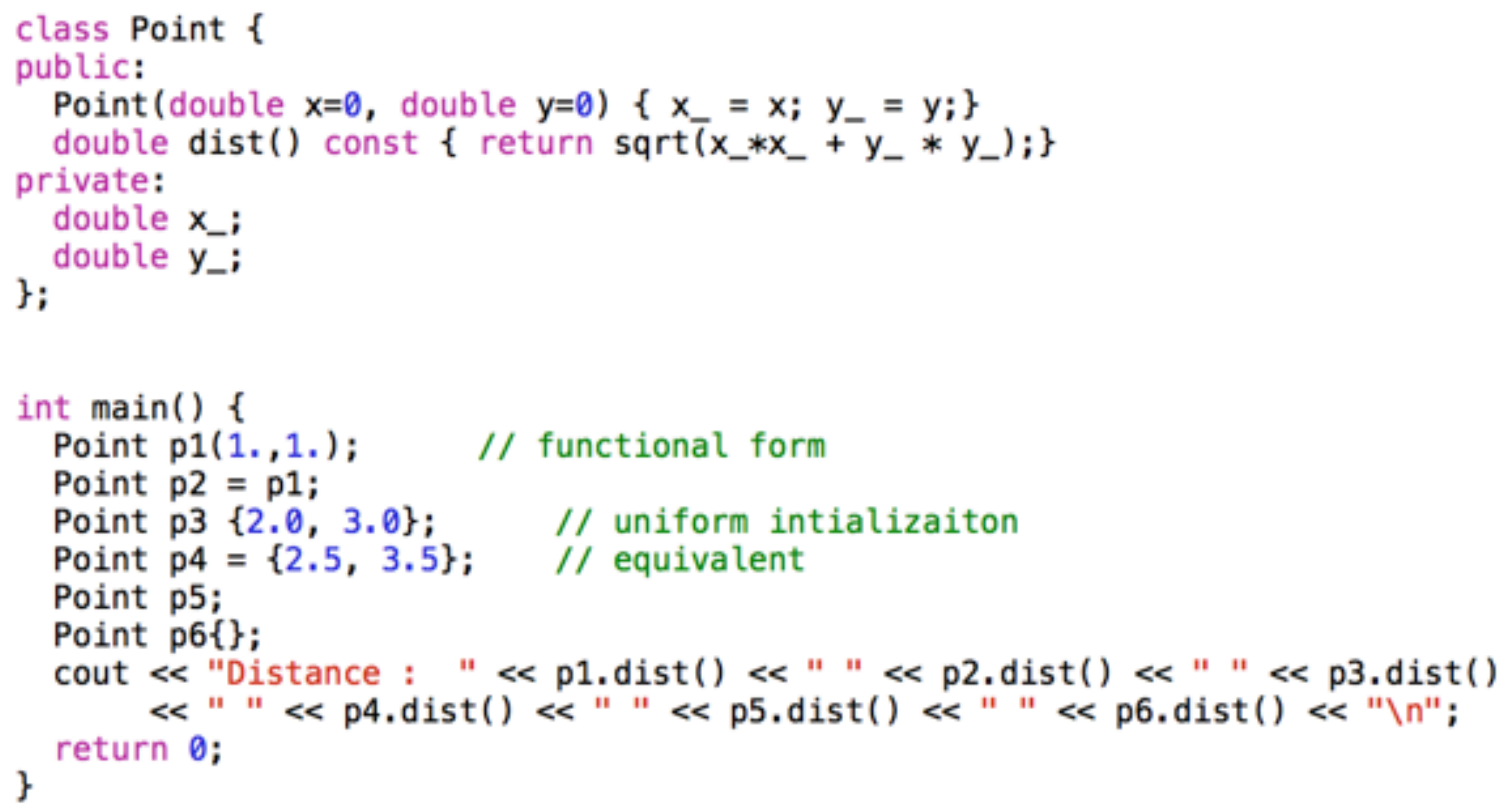
In the example below, class Point has no default constructor, only a constructor that takes two arguments. Class StraightLine has 2 Point members. The only way to make this work is via the member list initialization



```
class Point {
public:
  Point(double x, double y) : x_(x) { y_ = y;}
  double dist() const { return sqrt(x_*x_ + y_ * y_);}
private:
  double x_;
  double y_;
};

class StraightLine {
public:
  StraightLine(double x1, double y1, double x2, double y2) : p1(x1,y1), p2(x2,y2) { slope = (x2 == x1 ? 1e9 : (y2-y1)/(x2-x1));}
  double get_slope() const { return slope;}
private:
  Point p1;
  Point p2;
  double slope;
};
```

uniform initialization

Since C++11 *uniform initialization* is also allowed



```
class Point {
public:
  Point(double x=0, double y=0) { x_ = x; y_ = y;}
  double dist() const { return sqrt(x_*x_ + y_ * y_);}
private:
  double x_;
  double y_;
};


int main() {
  Point p1(1.,1.);      // functional form
  Point p2 = p1;
  Point p3 {2.0, 3.0};      // uniform intializaiton
  Point p4 = {2.5, 3.5};    // equivalent
  Point p5;
  Point p6{};
  cout << "Distance :  " << p1.dist() << " " << p2.dist() << " " << p3.dist()
       << " " << p4.dist() << " " << p5.dist() << " " << p6.dist() << "\n";
  return 0;
}
```

We also see here the example of a default value

Shallow versus deep copy

With abstract objects and dynamic memory we need to think about what we want to copy: the pointers (and the references), or the objects themselves (the values)? The former corresponds to a shallow copy, the latter to a deep copy.

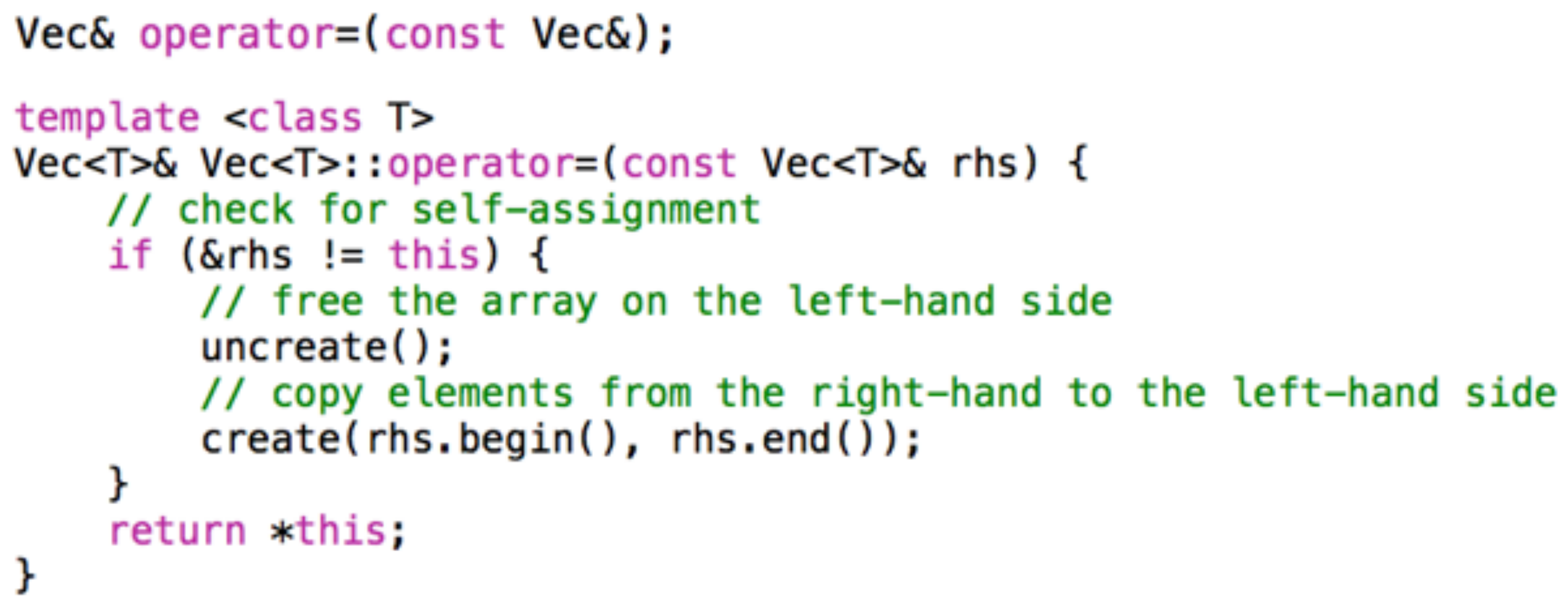
When no copy constructor is provided, the compiler will make one for you but it can only copy the static members and this will always correspond to a shallow copy. In case of a vector we really want to copy the whole object with all the values (deep copy), not the address of the first element (shallow copy)

Assignment operator

The issue of shallow vs deep copies also appears when assigning 2 objects:

Vec<int> v(100,1);
Vec<int> v2;
v2 = v1; *// assignment!*

for non-fundamental types the compiler has no idea how to assign objects to each other. Nevertheless, we would like to equate 2 vectors to each other, that is to overload the operator=



```
Vec& operator=(const Vec&);

template <class T>
Vec<T>& Vec<T>::operator=(const Vec<T>& rhs) {
    // check for self-assignment
    if (&rhs != this) {
        // free the array on the left-hand side
        uncreate();
        // copy elements from the right-hand to the left-hand side
        create(rhs.begin(), rhs.end());
    }
    return *this;
}
```

It takes as argument a const-ref (to an object of the same type) and the result-type is a reference to an object of the class. This means that *v2 = v1;* is understood by the compiler as *v2.operator=(v1);*

Assignment operator

The first line checks for self-assignment. In that case nothing should be done.

The keyword this represents a pointer to the object whose member function is being executed. It is used within a class's member function to refer to the object itself. It is very frequently used to check for self-assignment and in the form return *this when a reference to the object is needed as return type.

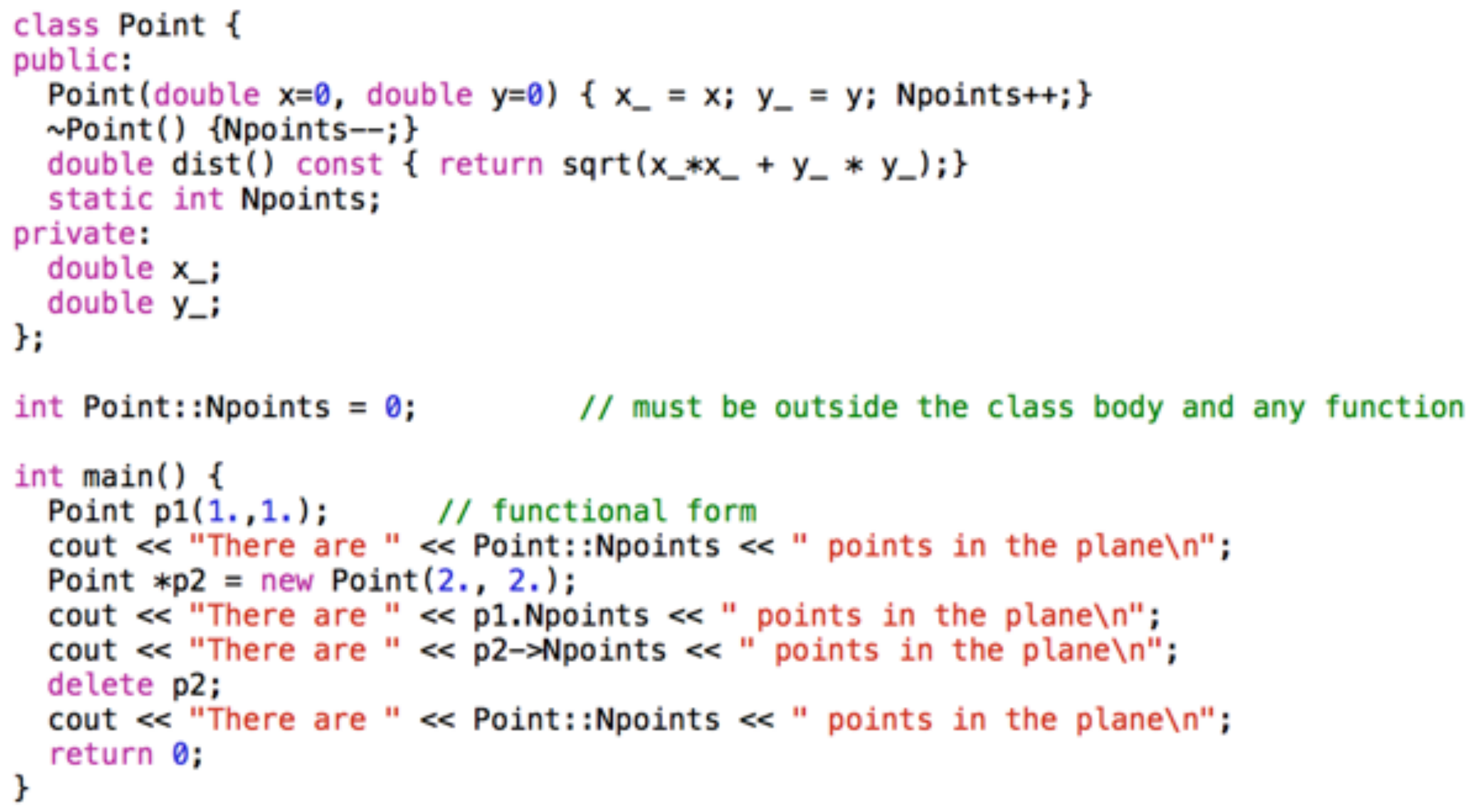
Clearly we want assignments between objects of the same type to be possible. But in our example v2 was not of the same size as v1! So the actual assignment is more involved than a member-wise copying: we first need to make sure that v2 is of the same size as v1 by destroying and freeing what was in v2 before (if anything), and then allocate new memory of the size of v1. Finally, the memberwise copying can be performed.

In case your class consists only of fundamental data types you can let the compiler write an automatic version of operator= for you.

static data members

A static data member of a class is also known as a class variable

- only one value even if there are many objects of the class
- cannot be initialized in the class : must be initialized outside the class
- enjoys class scope operator ::
- also static member functions are allowed, but they can not access non-static data members nor functions (nor is the use of this allowed)



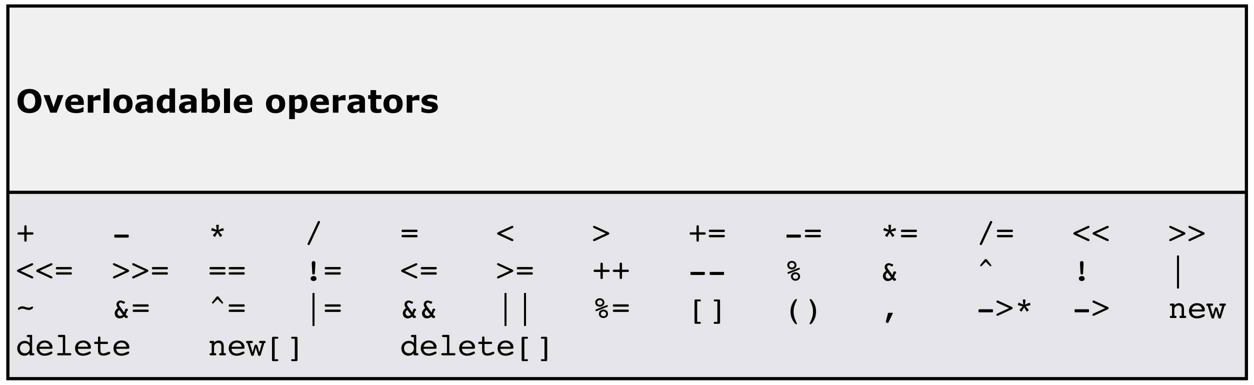
```
class Point {
public:
  Point(double x=0, double y=0) { x_ = x; y_ = y; Npoints++;}
  ~Point() {Npoints--;}
  double dist() const { return sqrt(x_*x_ + y_ * y_);}
  static int Npoints;
private:
  double x_;
  double y_;
};

int Point::Npoints = 0;         // must be outside the class body and any function

int main() {
  Point p1(1.,1.);      // functional form
  cout << "There are " << Point::Npoints << " points in the plane\n";
  Point *p2 = new Point(2., 2.);
  cout << "There are " << p1.Npoints << " points in the plane\n";
  cout << "There are " << p2->Npoints << " points in the plane\n";
  delete p2;
  cout << "There are " << Point::Npoints << " points in the plane\n";
  return 0;
}
```

Design

- A class corresponds to 1 concept and is a collection of all types (members), functions and operators to realize this concept.
- make members private. Access should be via public member functions *T get_name() const;* and *void set_name(const T&);*
- *"data hiding" :* the user is not interested in the implementation of the class and should not care about a correctly functioning class when debugging
- every class must have a (default) constructor, destructor, copy constructor and assignment operator
- understand the difference between copying and assigning
- understand when temporaries are created and copied
- make specific constructors explicit when they have one argument to prevent implicit conversions and copy-initialization
- think carefully about constness : make const whatever member should be const!
- implement iterators and const-iterators when useful
- overloading *T& operator()* and *T& operator[]* can be very useful for type conversions!
- overloading more operators is only meaningful when it is needed and intuitive
- do not forget the ; at the end of the class definition

Operator overloading

We have already seen how we can overload the constructor and the assignment operator. When adding two vectors v1 and v2, we would like to write v1+v2 but the compiler does not know how to interpret this for non-fundamental datatypes. We therefore have to overload the operator+. A list of operators that can be overloaded can be found below:



Overloadable operators												
+	-	*	/	=	<	>	+=	-=	*=	/=	<<	>>
<<=	>>=	==	!=	<=	>=	++	--	%	&	^	!	|
~	&=	^=	|=	&&	||	%=	[]	()	,	->*	->	new
delete		new[]		delete[]								

That does not mean that it is a good idea to overload these operators — we advice not to do so for new, delete, the comma operator, ++, —, ~, … in fact.

There might be good reasons to overload the operators [], (), +, +=, -, -=, ==, !=, < , > though. For instance an overloaded relational operator (<) might come in handy to sort a list of students according to their final grade.

Operator[] and operator()

our own Vec<T> class can be viewed as an encapsulated array of type T (via pointers T*). Usual arrays are indexed via [], and we would like to have this functionality as well. We overload this operator as follows:

```
const T& operator[] (size_type i) const { return data[i]; }
```

A statement of the type v[4] is hence understood as v.operator[](4) by the compiler.

Usually we have to provide also a non-const version in case we want "read/write" access and not just "read" access

```
T& operator[] (size_type i)       { return data[i]; }
```

This version will be called whenever objects are non-const.

The result type should be a reference to an object of type T. This is clear if we want to write an assignment like this: v[4] = 5; which requires that we modify v itself.

In the same spirit we can also overload the operator() for function calls:

```
T& operator()(size_type i) { return data[i];}
```

We could of course have defined the function body in any way we find appropriate.

For the notion of rvalue and lvalue (and move constructors) : see C++11.

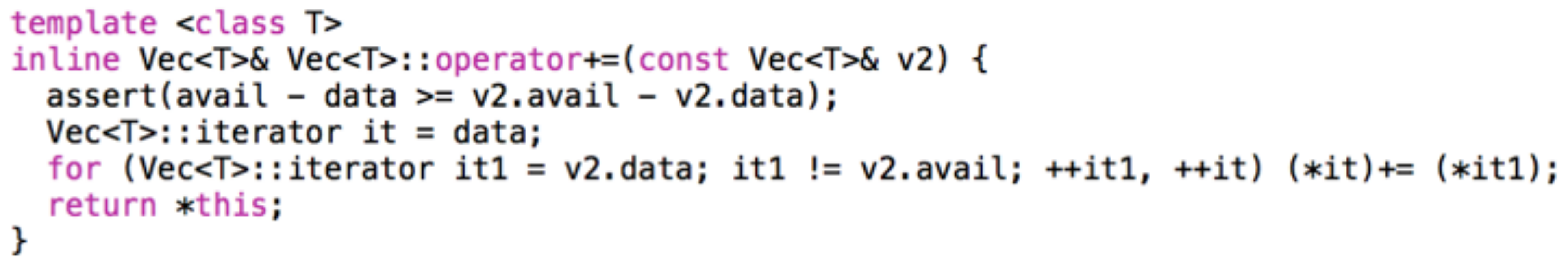
Operator+=

The operator+= adds a vector v2 to an existing vector v1 and overwrites v1.

The compiler understands this statement as v1.operator+=(v2)

The return type is hence a reference to Vec<T>&.

The argument is unchanged during this operation. It can hence be const, and there is no need to copy it, so a reference suffices.



```
template <class T>
inline Vec<T>& Vec<T>::operator+=(const Vec<T>& v2) {
  assert(avail - data >= v2.avail - v2.data);
  Vec<T>::iterator it = data;
  for (Vec<T>::iterator it1 = v2.data; it1 != v2.avail; ++it1, ++it) (*it)+= (*it1);
  return *this;
}
```

As a safety check we did a crude size check via assert. The actual copying is done via iterators. Finally, the reference to the current object via (*this) is taken.

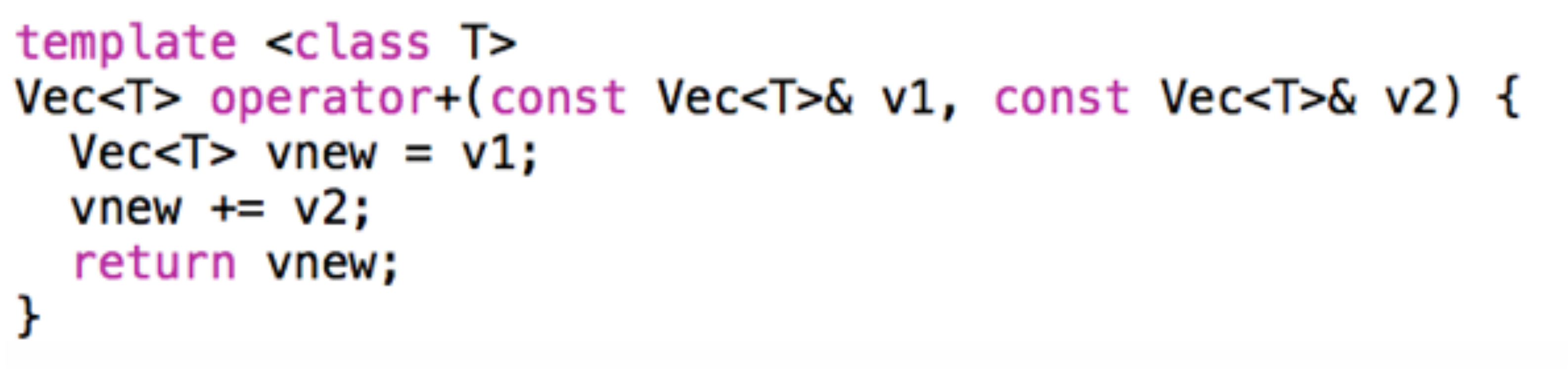
*note: the standard library does not provide operator overloading for the operators +=, +, *= , … for the vector class*

Operator+

We want to be able to sum two vectors (say of the same type T). We know that this addition does not change the value of either operand.

The return value should be a Vec<T> object

There is hence no reason why it should be a class member (this is a design criterion, not a language feature). It is better to implement operator+ in terms of operator+= as follows:



```
template <class T>
Vec<T> operator+(const Vec<T>& v1, const Vec<T>& v2) {
  Vec<T> vnew = v1;
  vnew += v2;
  return vnew;
}
```

We create a new Vec<T> object by initializing a local variable vnew to be a copy of v1 (which uses the copy constructor). Next, we invoke operator+= on vnew to add v2, and then we return vnew (again through calling the copy constructor).

Note that there is a specific order in which the addition is implemented, which is not necessarily commutative (eg, when v1 and v2 have different size).

Operator *=

We wish to be able to multiply our Vec<T> objects with scalars of type T.

```
Vec& operator*=(const T&);
```

The requirements are rather similar to operator+= : the return type should be a reference to a Vec<T>& type, and the operator *= should be a class member.

The function argument is an object of type T, which we can take to be a const. In case T is a complicated object, copying it is costly and unnecessary, so we can take a const T& as argument

```
template <class T>
inline Vec<T>& Vec<T>::operator*=(const T& alpha) {
  for (Vec<T>::iterator it = data; it != avail; ++it) (*it) *= alpha;
  return *this;
}
```

In a similar way as for operator+, the operator* can be defined but taking two different types as arguments (pay again attention to the order of the arguments: for the compiler, multiplying a Vec<T> with a scalar is not the same as multiplying a scalar with an object of type Vec<T>.

Operator <

Standard library algorithms such as std::sort or trees such as std::set are based on a relational ordering of objects (known as the compare concept), where the default is the operator<. For the fundamental data types (int, double, char, …) the meaning of lesser than is usually clear but for our own data types we have to give this a precise meaning.

Let us use the convention that we order our 2-dimensional points as follows: p1 < p2 if (x1 < x2) or ((x1 == x2) and (y1 < y2)).

```
template<class T>
class Point {
public:
  Point(double x=0, double y=0) { x_ = x; y_ = y;}
  Point (const Point& p) { x_ = p.x_; y_ = p.y_;}
  Point& operator=(const Point& p) {x_ = p.x_; y_ = p.y_; return *this;}
  double dist() const { return sqrt(x_*x_ + y_ * y_);}
  double x_;
  double y_;
};
```

```
template <class T>
bool operator<(const Point<T>& p1, const Point<T>& p2) {
  if (p1.x_ < p2.x_) return true;
  if ((p1.x_ == p2.x_) && (p1.y_ < p2.y_)) return true;
  return false;
}
```

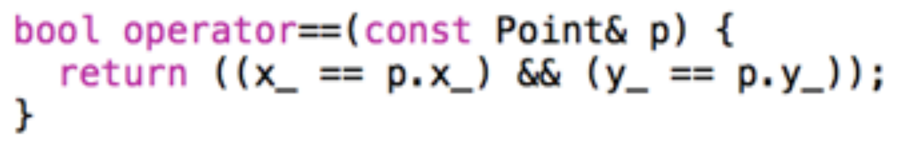
the return type should be bool, the function arguments to const-ref to Point objects; hence, there is no reason to have operator< as a class member function

in C++11 (or later) we can write this in more compact from using std::tie from <tuple> :

```
template <class T>
bool operator<(const Point<T>& p1, const Point<T>& p2) {
  return (tie(p1.x_, p2.x_) < tie(p2.x_, p2.y_));
}
```

Operator==

In a very similar way the operator== can be overloaded: the return type should be bool and the function argument does not change:



```
bool operator==(const Point& p) {
  return ((x_ == p.x_) && (y_ == p.y_));
}
```

friends

A careful reader will have noticed that I made the variable x_ and y_ public such that operator< could have access to them. That looks like bad practice. Is there a more elegant way?

```
public:
  /* as before */

  template<class U>
  friend bool operator<(const Point<U>&, const Point<U>&);

private:
  double x_;
  double y_;
```

We can define operator< as a friend of class Point. Friends have access to all private data members and must be declared in the class declaration.

In case of templates, we see that the syntax is slightly more involved than you might expect: you have to write template<class U> inside the class body — assuming that both are of type T will give a compiler error

This produces a slight security error: U can be a double where T is an int etc. There are ways to avoid this using *forward declarations* (see textbooks)

A very common way for friend access is to overload the operator<< for an output stream:

```
template<class U>
friend std::ostream& operator<<(std::ostream& os, const Point<U>& p) {os << "(" << p.x_ << "," << p.y_ << ")"; return os;}
```

Template specialization

Template specialization for classes works in the expected way:

```
template<class T>
class Point {
public:
  Point(double x=0, double y=0) { x_ = x; y_ = y;}
  double dist() const { return sqrt(x_*x_ + y_ * y_);}
private:
  double x_;
  double y_;
};

template<>
class Point<unsigned int> {
public:
  Point(unsigned int x=0, unsigned y=0) { x_ = x; y_ = y;}
  unsigned int dist() const { return x_ + y_;} // Manhattan distance
private:
  unsigned x_;
  unsigned y_;
};


int main() {
  Point<unsigned int> p(1,1);
  cout << "Distance " << p.dist() << "\n";
  return 0;
}
```

indicates template specialization for specific type

class Point is templated

const

The use of const is very important in C++ and can be found in different contexts:

1. A constant object of a class has restricted access to read-only (member variables and member functions)

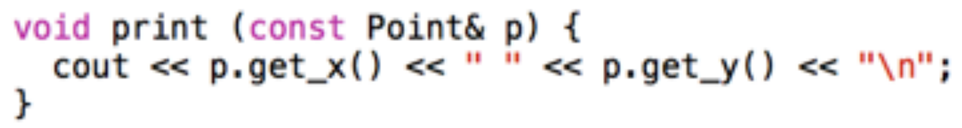
```
class Point {
public:
  Point(double x, double y) { x_ = x; y_ = y;}
  double dist() const { return sqrt(x_*x_ + y_ * y_);}
  double x_;
  double y_;
};


int main() {
  const Point p(5., 4.);        // example of const object of class Point
  //p.x_ = 4.;                // error!
  cout << "Distance :  " << p.dist() << "\n";
  return 0;
}
```

2. The function dist() could only be called because it is a const function (= which does not change any of the class members). In such a case const follows the function name and variable list.
3. The return value of a function can be const

```
const double& get_x() { return x_;}
```

const

Const objects are very common. You should always write const when you don't want to modify objects — this will help you with debugging. Most functions taking classes as parameters take them by const reference, and these can thus only access its const members!



```
void print (const Point& p) {
  cout << p.get_x() << " " << p.get_y() << "\n";
}
```

we already talked about const_iterators.
Also note that declaring a class const, turns the this pointer into a const pointer!

union

A union is a type of class in which only one member (the largest one) is stored at a time. The default access is public

volatile

volatile is a keyword that tells the compiler not to optimize aggressively. It means that the volatile variable could be changed by sources not known to the compiler and hence change

mutable

mutable members inside a const struct or class are treated as non-const. Another use is inside const functions where mutable variables can still be modified. It can be useful when most variables (should) remain constant but only a few need to be updated

inlining

A function call is expensive.

Classes allow inlining: inlined functions are replaced by the actual function body code without performing the function call, which reduces overhead (copying arguments and the result type)

Function definitions inside the class definition are automatically inlined

Functions defined outside the class declaration can be inlined. If the code is sufficiently easy and small, the compiler will successfully manage to inline the function *automatically*.

in C++11 a function declared constexpr is always inlined

Type traits

Suppose we want to compute the average of a std::vector<T>

```
template <class T>
T average(const vector<T>& v) {
  T sum = 0;
  for (unsigned int n=0; n < v.size(); n++) sum += v[n];
  return sum/v.size();
}
```

It has problems for int : let v = (1,2,2) then the result is int(5/3) = 1.

We can do better using type traits

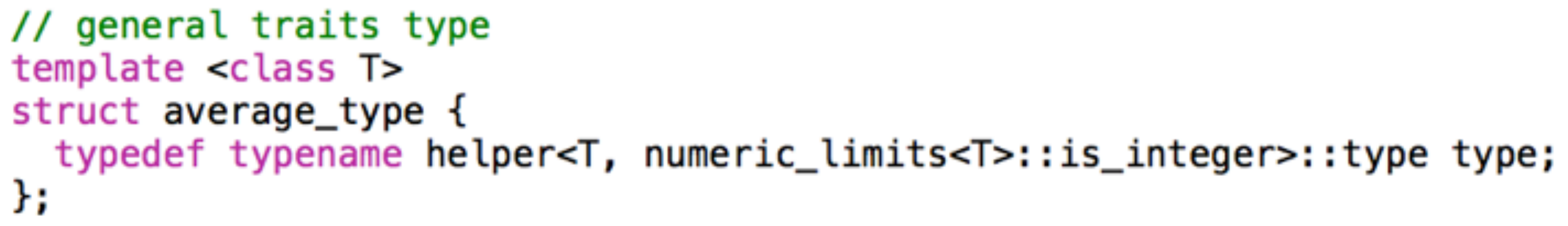
```
// general traits type
template <class T>
struct average_type {
  typedef T type;
};

// special cases
template<>
struct average_type<int> {
  typedef double type;
};

// repeat for all integer types : int32_t, int64_t, ...
```

```
template <class T>
typename average_type<T>::type average(const vector<T>& v) {
  typename average_type<T>::type sum = 0;
  for (unsigned int n=0; n < v.size(); n++) sum += v[n];
  return sum/v.size();
};
```

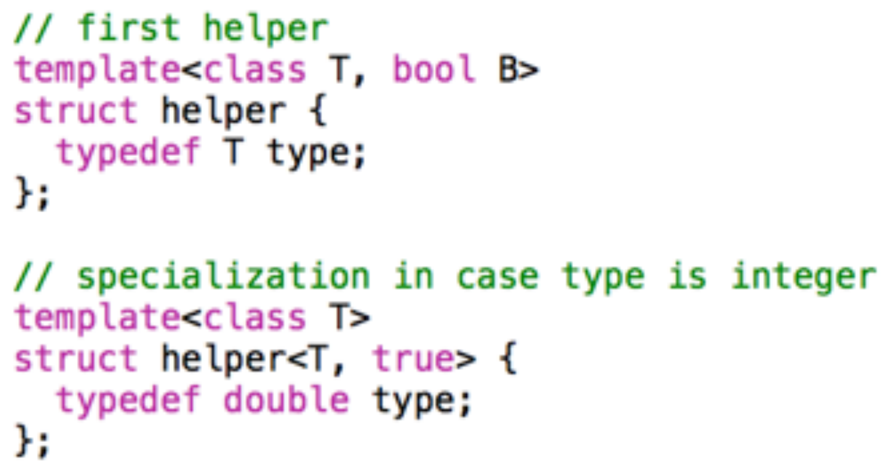
note the use of keyword typename indicating that a type name follows, not a variable.

Type traits

By using the numeric_limits class we can generalize to all integers:



```
// general traits type
template <class T>
struct average_type {
  typedef typename helper<T, numeric_limits<T>::is_integer>::type type;
};
```



```
// first helper
template<class T, bool B>
struct helper {
  typedef T type;
};

// specialization in case type is integer
template<class T>
struct helper<T, true> {
  typedef double type;
};
```

note: C++11 has a traits class and allows for a simpler solution to the above problem

Question

- how would you implement type traits when you want to add 2 vectors of different types such as double and int?