# Accelerated C++

p.23 size_type, eg. std::string::size_type or
std::vector<int>::size_type

p.47
```
typedef vector<double>::size_type vec_sz;
```

p.47
```
streamsize prec = cout.precision();
<ios>
```

p.57 Reading from an input stream in
```
istream& read(istream &in, vector<double> &vec){
        if(in){
                vec.clear();
                double x;
                while (in >> x)
                    vec.push_back(x)
                in.clear();
        }
        return in;
}
```

p.68 Header and source file partitioning

p.73 Error types: logic_error, domain_error, invalid_argument,
length_error, out_of_range, runtime_error,
range_error, overflow_error, underflow_error

p.94
```
vector<string> vec, other;
  vec.insert(vec.end(), other.begin(), other.end());
```

p.98 <cctype> header for manipulating character data
isspace(c), isalpha(c), isdigit(c), isalnum(c),
ispunct(c), isupper(c), islower(c),
toupper(c), tolower(c)

p.115 <numeric> accumulate(v.begin(), v.end(), 0.0);
where the third arg determines the return type.

p.112 <algorithm> vector<double> grades;

transform(students.begin(), students.end(),
back_inserter(grades), grade_aux);

p.110 <algorithm> find(homework.begin(), homework.end(), $\underset{\text{val}}{0}$);

p.102 <algorithm> copy(v.begin(), v.end(), back_inserter(res));

p.116 <algorithm> vector<double> nonzero;

remove_copy(homework.begin() homework.end(),
back_inserter(nonzero), 0);

p.117 remove(b, e, t), remove_copy(b, e, d, t)
remove_if(b, e, p), remove_copy_if(b, e, d, p)

"If" variant uses a predicate/functor, p, instead of a val, t.

p.119 <algorithm> stable_partition(students.begin(), students.end(), pgrade);

p.120 Crucial fact to understand algorithms and containers:
"Algorithms act on container elements - they do not act on containers."

p.121 <u>List of algorithms</u>.

p.146 <u>Iterator types</u>:

i.) Input - sequential read-only.
ii.) Output - sequential write-only.
iii.) Forward - sequential read and write.
iv.) Bidirectional - sequential forwards and backwards read-write.
v.) Random access - random, non-sequential forwards and backwards read-write.

p.148 <algorithm> bool binary-search(Fwd.begin(), Fwd.end(), (const T&val);

p.151 <iterator> istream_iterator<T>(istream-type &s)

```
copy(istream_iterator<int>(cin), istream_iterator<int>(),
                                 back-inserter(vec));

copy(vec.begin(), vec.end(), ostream-iterator<int>(cout, " "));
```

p.172 <u>Function pointers and typedef for function pointers</u>

Eg. `double (*analysis)(const vector<int> &) ;`

```
typedef double (*analysis)(const vector<int> &) ;

analysis get_analysis_ptr() ;          (Modern)

double (*get_analysis_ptr())(const vector<int> &)     (Arcane)
```

p.175 <cstddef> ptrdiff-t signed integer type for pointer arithmetic

p.176 <u>String literals</u> are null terminated with '\0' char.
<cstring> strlen() returns the number of chars in a string literal (or other null terminated) array of chars, not counting the null at the end.

p.180 Output streams cout, cerr, clog.
cout and clog employ buffering; whilst cerr does not.

p.191 When defining your own container class, remember to implement:

```
typedef T value_type;
typedef T& reference_type;
typedef const T& const_reference;
typedef ptrdiff_t difference_type;
typedef size_t size_type;
typedef T* iterator;
typedef const T* const_iterator;
```

p.199 <u>Assignment is not initialisation</u>. Assignment (operator=) always obliterates a previous value, initialisation never does so. Rather, initialisation involves creating a new object and giving it a value at the same time.

```
string url = "www.google.co.uk"        // initialisation
string x;                              // initialisation
 x = url;                              // assignment
```

When we use = to give initial value, we invoke copy constructor. The compiler will call the string constructor that takes a const char*. That constructor can construct url directly, or construct an unnamed temporary, and then call the copy constructor to construct url as a copy of that temporary.

p.201 <u>Rule of Three</u>: i) copy constructor, ii) assignment, iii) destructor.

p.204 <memory> allocator<T> member functions:

```
T* allocate (size_t);
void deallocate (T*, size_t);
void construct (T*, const T&);
void destroy (T*);
```

<memory> allocator<T>

p.207 Non-member functions:

```
void uninitialized_fill (Fwd, Fwd, const T&);

Fwd uninitialized_copy (In, In, Fwd);
```

p.217/ p.251 <u>Friend classes and functions</u>. Makes no difference whether it follows a public or private label. Friendship is neither inherited nor transitive, friends of friends and classes derived from friends have no special priviledges.

p.220/ p.225 <u>Automatic conversions</u> once non-explicit constructor available which takes a single argument of appropriate type, or conversion operator of the form "operator typename();" Conversion operators must be member functions.

p.220 Symmetric binary operators should be non-member functions. Asymmetric assignment binary operators (eg. "+=") should be member functions. Also, if an operator changes the data of an existing object, it should be a member function.

p.223 void *type "universal pointer" can point to any type of object but cannot be dereferenced because the object type to yield is unknown. But permits conversion to bool.

Eg. istream cin defines conversion to void* rather than to an arithmetic type or bool. This prevents mistakes of the type

```
int x;
cin << x;
```

which would otherwise convert cin to bool, convert to int, then shift bitwise left by x bits.

p.235 Virtual only applies when a function is called through a reference or pointer. After all, calling a function on an ordinary object means we know the exact type of the object. The phrase "dynamic binding" captures the notion that functions may be bound at runtime, as opposed to "static binding" that happens at compile time. Virtual label is automatically inherited.

p.246 Ordinarily, when a derived class redefines a function from the base class, it does so exactly - the parameter list and the return type are identical. However, if the base-class function returns a pointer (or reference) to a base class, then the derived-class function can return a pointer (or reference) to a corresponding derived class.

p.255 Managing memory: Handle class, Ref_handle class, Ptr class.

p.256 Overloading operator->()

$x \to y$ equivalent to $(x.\text{operator}\to())\to y$