Enumerations By default allocate successive integer values from 0

Type Conversion Automatic conversion may occur, generally widens

Grammar Expressions have results, expressions followed by a semicolon are statements
Can separate several expressions with a comma, evaluated left to right with result being the RHS

Globals Variables outside functions are initialised to 0 by default

Externs Can declare but not define using the extern keyword
Extern within a function references a global var
Extern takes on int type by default, you need not spec it
Extern is optional on function declarations
Key point: declaration does not allocate memory

Functions A function definition with no values means that its arguments should not be type checked!
Declare a function with no arguments using void
Can do partial parameter specification with …

Static In the global scope, static does not export the symbol
In the local scope, static means variable retains value

Preprocessor Deletes each occurrence of a backslash followed by \r\n
Replaces comments by " "
Does conditionals
Replaces definitions
Replaces escaped sequences in chars/strings, concats. any adjacent strings
`#define name replacement`
Prefixing a parameter with # places the value in quotes
Placing a ## between two parameters removes any whitespace between them
Override line and filenames with `#line constant filename`
`#error some text`

Arrays Multidimensional with:
`int i[5][10]`
Don't need to pass the first dimension length to a function (i.e. the leftmost)

Function Pointers `int (*compare)(int, int)`

Structures `struct circle c = {12, 23, 5};`

Unions Have size equal to their largest member

Bit Fields `struct foo { int f1 : 2; }`

Misc. Do not have addresses
`const`, `volatile`
`typedef struct llist *LLptr;`
`typedef struct _foo { int bar; } foo;`
`inline`: preserves semantics, esp. still has an address, but must be defined in the same execution unit it is used in

Types Character literals are now of type 'char'
Adds bool type (true, false)
Enumeration defines a new type, not constants, with no implicit convesion

References `int &refi = i[0];`
Implicit type conversion into a temporary is automatic for const references, else error

Functions Overloading
Default arguments: `double v=10.0`, but cannot have defaults before required args

Namespaces Collect related code
`using` keyword

Classes `private`, `protected`, `public`
Structs are the same as classes but have default public access, not private
Constructors have the same name as class, destructors prefixed by a tilde
Statics are per-class
Can define an instance by assignment: does value copy, unless you define a copy constructor (with const reference argument). May also overload assignment
Const member functions

Use constructor notation to initialise class-class variables, must be used on const and reference variables, and to init a base class (by name)
Can make arrays of classes if they have default constructor
Friend functions can be declared in a class (`friend Foo operator*(const Bar&, const Foo&);`)
Non-virtual functions are called on the static type of the variable, pointer, or reference, virtual functions use the vtable
Can declare pure virtual functions with "= 0": this makes the class abstract

Multiple Inheritance

Name clashes must be resolved by explicit class naming (using the namespace operator)
Need virtual base classes in the diamond situation

Operator Overloading

Define outside or within the class body. If you do both the compiler prefers the version outside the class

Memory Management

`new`, `new[]`, `delete`, `delete[]`

Temporary objects not bound to expressions only exist during evaluation of a full expression!

Exceptions

`try`, `throw`, `catch`

Templates

```
template<class T> class Stack { push(T v); }
template<class T> void Stack<T>::push(T v) { }
template<int i> class A { int b[i]; }
template<class T, T val> class B { T blah = val; }
```

Not type checked until instantiation

Template Specialisation

```
template <class T> struct B { }
template <> struct B<A> { }
template<int N> int fact() { return N*fact<N-1>(); }
template<> int fact<1>() { return 1; }
```

Resolving overloads uses the most specialized call

STL

Separates algorithms from data structures (containers) using iterators (input, output, forward (= input + output), bidirectional, random access)
Adaptors modify the interfaces of components (e.g. reverse_iterator)
If a container dosen't support your algorithm your container is wrong ☺
Overload function calls inside classes: this lets you store per-instance data ie. closure!