SECTION 01
language extensions

SECTION 01
Language extensions

Object initialisation

Parenthesized

```
std::string s("hello");
int m = int();  //default initialization
```

Assignment

```
std::string s = "hello";
int x = 5;
```

Aggregate entities

```
int arr[4]={0,1,2,3};
struct tm today={0};
```

Member Initialiser List

```
struct S { int x; T(): x(0) {} };
```

C++ 98 has at least 4 different initialisation notations:

- Parenthesized initialisation is used for default initialisation of built-in types and non-default initialisation of objects (but not default-intialised objects!)
- Assignment (=) can be used in some cases
- For structures and arrays you can use braces ({})
- Constructors may use a member initialiser list.

Brace initialisation notation

```
class C
{
  int a;
  int b;

public:
  C (int i, int j);
};

C c {0, 0};                          // Object initialisation.
                                     // Equivalent to: C c(0,0);

int* a = new int[3] { 1, 2, 0 };     // Array initialisation


class X
{
  int a[4];

public:
  X() : a{1, 2, 3, 4} {}             // Member array initializer
};
```

C++11 ratifies the initialiser notation, making it (more) consistent for all types.

Default construction

```
class C
{
  int a;
  int b;

public:
  C ();
  C (int i, int j);
};

// C++98-style:
//
C c1;                         // Default constructor
C c2(100, 200);               // Non-default constructor

// C++11-style
//
C c3 {};                      // Default constructor
C c4 {100, 200};              // Non-default constructor
```

In C++98 there is a distinct different syntax between default and non-default construction of an object. The brace notation eliminates this.

Container initialisation

```
// Container initializer

std::vector<string> vs = { "first", "second", "third"};

std::map<string, int> m
{
  {"Dijkstra", 1972},
  {"Scott", 1976},
  {"Wilkes", 1967},
  {"Hamming", 1968}
};
```

Initialiser lists allow STL containers to be initialised in the same way as arrays.

Initialiser lists are not supported by VS 2011 at present.

Non-static member initialisation

```
class C
{
  int a = 7;     // Default member variable initialisers.
  int b = 10;    //

public:
  C();
  C(int i, int j) : a{i}, b{j} { /* ctor body */}
};


C c1 {};         // a = 7; b = 10
C c2 {10, 20};   // a = 10; b = 20
```

C++98 supported the initialisation of static class data members. C++11 extends this idea by allowing non-static members to be given a default value.

If the member variable is not initialised by the class constructor, it will be given its default value.

This notation is equivalent to

```
class C
{
public:
  C(int aInit = 7, int bInit = 10) : a(aInit), b(bInit) {}
private:
  int a, b;
};
```

This feature is currently not supported in VS2010 / 2011

Default functions

```
class A
{
public:
  // Use compiler-supplied constructor
  // and destructor.
  //
  A() = default;
  virtual ~A() = default;
};


int main()
{
  A anAObject;            // Uses default constructor
}
```

The `=default` instructs the compiler to generate the default implementation for the function.

Defaulted functions have two advantages:

- They are more efficient than manual implementations,
- they rid the programmer from the chore of defining those functions manually

Deleted functions

```
class NoCopy
{
public:
  NoCopy& operator= (const NoCopy&) = delete;
  NoCopy (const NoCopy&) = delete;
};


int main()
{
  NoCopy a;
  NoCopy b(a);   // ERROR

  NoCopy c = a;  // ERROR
}
```

Deleted functions are useful for preventing object copying.

Recall that C++ automatically declares a copy constructor and an assignment operator for classes.

To disable copying, declare these two special member functions =`delete`:

Automatic type deduction

C++ is a strongly-typed language

In C++ must explicitly declare the type of all objects
- In some cases this is onerous
- In other cases it is extremely difficult (templates?!)

C++11 allows automatic type deduction

Type deduction

```
auto x = 0;        // x has type int because 0 is an integer
                   // literal

auto c = 'a';      // c has type char

auto d = 0.5;      // double (floating point literal)


int func()
{
  return 0;
}

int main()
{
  auto i = func(); // i has type int (return type from func())
}
```

Automatic type deduction uses the existing mechanism used by compiler for template type-deduction.

Note, the object still has a definitive type. `auto` is not in any way like Visual Basic's `var` type.

Simplifying declarations

```
// C++98
//
void func(const vector<int>& vi)
{
  vector<int>::const_iterator ci = vi.begin();
  // …
}
```

```
// C++11
//
void func(const vector<int>& vi)
{
  auto ci = vi.begin();
  // …
}
```

Automatic type deduction is useful when the type of the object is verbose or when it's automatically generated (for example, in templates).

Limitations of auto

```
void invalid(auto i)     // ERROR - can't use auto
{                        // for function parameters
}


class A
{
  auto _m;               // ERROR - invalid for
};                       // member variables


int main()
{
  auto arr[10];          // ERROR - invalid for
}                        // arrays.
```

To use auto the compiler must have the information to deduce the type (and therefore allocate memory). The above declarations are invalid.

The decltype operator

```
int main()
{
  int i = 10;

  cout << typeid(decltype(i + 1.0)).name() << endl;

  decltype(i++) b;       // b is of type int;
                         // but i is not incremented.
}
```

The `decltype` operator returns the type of the expression; but does not evaluate it.

Problems with return types

```
template<typename T1, typename T2>
<return_type> multiply(T1 x, T2 y)
{
  return x * y;
}
```

We cannot evaluate the return type…

```
// C++98 solution
//
template<typename RT, typename T1, typename T2>
RT multiply(T1 x, T2 y)
{
  return x * y;
}
```

In C++98 there were scoping problems with some expressions. In the above example we want the *<return_type>* to be the type of `(x*y)`. We cannot know what that return type is until we instantiate the template.

The C++98 solution was to supply a third template parameter - the return type of the function. The user was responsible for specifying the return type, and an incorrect choice could lead to truncation (for example supplying an integer return type for two floating point parameters)

Using decltype…

```
template<typename T1, typename T2>
decltype(x * y) multiply(T1 x, T2 y)  // ERROR!
{
  return x * y;
}
```

`decltype` can deduce the type of the expression `(x*y)` but we still have the scoping problem, since x and y aren't in scope yet.

Trailing return types

```
template<typename T1, typename T2>
auto multiply(T1 x, T2 y) -> decltype(x * y)
{
  return x * y;
}
```

Putting the return type after the parameter list allows the parameters to be used to specify the return type.

The keyword `auto` is required to tell the compiler the return type is specified *after* the parameters

Iterating through containers

```
#include <vector>
#include <algorithm>

using namespace std;

void func(int i) { /* Do something with i */ }

int main()
{
  int arr[10];
  vector<int> v(10);

  // Add data to arr...
  // Add data to v...

  for (int i = 0; i < 10; ++i)
  {
    func(arr[i]);
  }

  for_each(v.begin(), v.end(), &func);
}
```

A very common problem is iterating through a container of data, using each element in turn in some calculation.

In C++ there are several idioms for achieving this - the `for` loop or the `for_each` algorithm.

range-for statement

```
void f(std::vector<double>& v)
{
  for (auto x : v)    // x is double
  {
    std::cout << x << std::endl;
  }

  for (auto& x : v) ++x;  // using a reference to allow
                          // us to change the value.
}


for (const auto x : { 1, 2, 3, 5, 8, 13, 21, 34 } )
{
  std::cout << x << std::endl;
}
```

The range-for statement provides iteration through a container of objects. A copy of each object, in turn, is taken from the container and may be used. By taking a reference you can manipulate the object in the container.

The range-for statement can be used with any container that is supported by `std::begin` and `std::end`. These functions return an iterator to the first element and last element in a container, respectively. The functions support all the STL containers and C-style arrays.

Enum classes

```
enum Alert { green, yellow, election, red };   // traditional enum

enum class Color { red, blue };                // Scoped and strongly-typed enum.
                                               // No export of enumerator names
                                               // into enclosing scope.
                                               // No implicit conversion to int

enum class TrafficLight { red, yellow, green };

Alert a = 7;                                   // ERROR (as ever in C++)
Color c = 7;                                   // ERROR: no int -> Color
conversion

int a2 = red;                                  // OK: Alert -> int conversion
int a3 = Alert::red;                           // ERROR in C++98; OK in C++11
int a4 = blue;                                 // ERROR: blue not in scope
int a5 = Color::blue;                          // ERROR: no Color -> int
conversion

Color a6 = Color::blue;                        // OK
```

Enum classes provide strongly-typed enumerations.

An `enum class`, unlike the C++98 `enum` does not export its enumerator names into its enclosing scope, meaning different `enum class` objects can have the same enumerator value, without causing a name issue.

Also `enum class` values cannot be implicitly converted to integers.

Enumeration underlying type

```
// Specify the underlying type:
//
enum EE : unsigned long {EE_ONE = 1, EE_TWO= 2, EE_BIG = 0xFFFFFFF0U};


// Forward declaration of enums:
//
enum class Color_code : char;     // (Forward) declaration
void foobar(Color_code& p);       // Use of forward declaration


// Other code…

// Definition:
//
enum class Color_code : char { red, yellow, green, blue };
```

We can now specify the underlying type of the enumeration (as long as it's an integer type). The default is integer; as with C++98

If the `enum class` type is to be used as a forward reference you can (must) provide the underlying type as part of the declaration.

Compile-time assertion

```
template<typename T1, typename T2>
auto multiply(T1 x, T2 y) -> decltype(x * y)
{
  static_assert(sizeof(decltype(x*y)) <= sizeof(long), "Result too big!");
  return x * y;
}

int main()
{
  multiply(1000L, 1000L);
}
```

```
>------ Build started:
Error C2338: Result too big!
BUILD FAILED
```

`static_assert` allows compile-time checks.

The `static_assert` first parameter must be a constant-expression (as it must be evaluated at compile time)

Key points

C++ Now has a standard method of initialisation for all types

Functions such as the constructor may be removed from use or given a (compiler-defined) default implementation

Automatic type deduction can be deduced by the compiler

range-for allows iteration through a container, given access to each element in turn

Enumerations classes allow a strongly-typed enumeration