FEABHAS

EMBEDDING SOFTWARE COMPETENCE

Bitesize Modern C++

An introduction to the new features in Modern C++ that are useful for programmers currently using C++98.

These chapters will help you to get started and to write more elegant and maintainable programs.

Author: Glennan Carnie, Technical Consultant at Feabhas

Bitesize Modern C++

Introduction

The C++11 standard marked a fundamental change to the C++ language. A range of new language constructs and library features were added and, as a result, new idioms were developed. Bjarne Stroustrup, originator of C++, referred to it as “feeling like a completely new language”.

In 2014 a revision of the standard was released with refinements to C++11 features and some new constructs, designed to improve the usability of the language; another revision is planned for 2017.

Traditionally, C++ programmers refer to their language version by the standard release year (so, C++98, C++03) but, as C++11 marked such a fundamental shift in the programming style of C++ (and as revisions are coming out more frequently) people are starting to refer to C++ post-2011 as 'Modern C++'

In the embedded world Modern C++ has had comparatively little take-up, possibly due to a lack of embedded compiler support. Slowly, though, this is beginning to change.

There is a huge amount of material available on Modern C++. It has been explored, examined and dissected by experts for nearly five year. However, for the C++ programmer just starting to transition to Modern C++ a lot of this material can be somewhat overwhelming. For many it can be difficult to find a simple place just to start.

And that's the idea behind these articles.

We wanted to present simple introductions to the new features in Modern C+; ones that are useful for programmers currently using C++98. These are not the big, headline features - lambda expressions, move semantics, etc. - but the 'little' additions that can make you more productive and your programs to be more elegant and maintainable.

The other thing we wanted to avoid was TMI – Too Much Information. These articles are not meant to be definitive; they don't explore every last quirk and corner-case. You won't be an expert after reading these articles. Their purpose is simply to get you started; to give you a jumping-off place for all the excellent, detailed information out there.

Enjoy!

Contents

Chapter	Page
Introduction	2
Automatic type deduction	4
Constant expressions	7
static_assert	10
enum class	11
nullptr	13
Using aliases	15
Uniform initialisation syntax	16
std::initializer_list	19
Range-for loops	22
override and final	24
noexcept	26
std::array	28
Smart pointers	31
About Feabhas	36

Automatic type deduction

C++ is a statically-typed language, that is, you must explicitly declare the type of every object before use. The type of an object specifies:

- The amount of memory occupied
- How the memory (bits) is interpreted
- The operations allowable on the object

An object's type is fixed at declaration – unless the programmer chooses to circumvent the type system using a cast.

Often for C++ objects specifying the type can be onerous:

```
namespace lib
{
  class UserDefinedType{};
}

int main()
{
  std::vector<lib::UserDefinedType> container;
  std::vector<lib::UserDefinedType>::const_iterator it =
                                          container.cbegin(); // Phew...
...
}
```

C++11 allows automatic type-deduction to simplify the creation and maintenance of code.

The auto keyword has been appropriated from C (where it was a storage specifier indicating a local – 'automatic' – variable) that is almost never used. In C++11 auto now means 'deduce the type of an object'

Let's start with some simple (but not particularly useful) examples.

```
int func()
{
  return 0;
}


int main()
{
  auto a = 0;           // a => int
  auto b = 'a';         // b => char
  auto c = 0.5;         // c => double
  auto d = func();      // d => int (return type from func())
}
```

The compiler uses the type of the initialiser to establish the type of the new object. Note, the object still has a definitive (and fixed) type. Type-deduction does not lead to 'duck typing'.

(One might reasonably argue that there is very little benefit in auto-deducing built-in types)

To use auto the compiler must have the information to deduce the type (and therefore allocate memory). Therefore, there are some (obvious?) limitations:

```
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
}
```

The mechanism used by the compiler for type-deduction is the same one used to deduce parameter types for template functions. Because of this, the object-type deduced for an auto object may not be *exactly* what the programmer expects.

The rules are as follows:

- The deduced type will be a new object.
- If the initialiser is a reference-to-object, the reference is ignored (since a reference is just an alias for an object).
- Any *const-volatile* (cv)-qualifiers on the initialiser are ignored (since the cv-qualifiers are a property of the initialiser, not the new object).
- Arrays are degenerated into pointer types.

```
const int  c = 100;
const int& c_ref = c;
int        array[100];

auto i = c;               // i => int  (cv-qualifiers ignored)
auto j = c_ref;           // j => int  (reference ignored)
auto k = array;           // k => int* (array degeneration)
```

We can cv-qualify the deduced type separately, if required. The cv-qualifiers are applied to the new object.

```
const auto l = c;        // l => int (then cv-qualified)
const auto m = array;    // m => int* const (ptr is const)
```

If we reference-qualify the auto-deduced type the rules change:

- The deduced type will now be a reference the initialiser object.
- The cv-qualifiers of the initialiser will be maintained on the deduced type (since you cannot have a non-const reference to a const object).
- If the initialiser is an array the deduced type is a reference-to-array type, not a pointer.

```
auto& n = c;          // n => const int&
auto& o = c-ref;      // o => const int&
auto& p = array;      // p => int(&)[100]
```

Returning to our original example, we can simplify the code (considerably) with the use of `auto`.

```
namespace lib
{
  class UserDefinedType {};
}


int main()
{
  std::vector<lib::UserDefinedType> container;

  auto it = container.cbegin();      // Much simpler!
  ...
}
```

Additionally, the use of `auto` allows us a degree of flexibility in our code for future modifications:

```
namespace lib
{
class UserDefinedType {};
}


int main()
{
  std::list<lib::UserDefinedType> container; // <= new container...

  auto it = container.cbegin();              // ...but no change here
  ...
}
```

Back to Contents

Constant expressions

A *constant expression* is an expression that can be evaluated at compile-time. The `const` qualifier gives a weak guarantee of a *constant expression* – a `const`-qualified type may notbe changed after initialisation but that does not guarantee it will be initialised at compile-time. For example:

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

  ...

  const int sz = size;          // Non-modifiable, but run-time initialized
  double array[sz];             // FAIL!
}
```

C++11 introduces a strong form of *constant expression*, `constexpr`, which also expands the capabilities of compile-time evaluation.

constexpr objects

A `constexpr` variable is essentially the same as qualifying the type as `const` with the additional requirement that its initialiser(s) must be known at compile-time. Any object type that could be made `const` can be made a `constexpr` object. As expected, the object must be initialised on definition. Note that literals are, by definition, *constant expressions*.

```
int main()
{
  int sz;
  constexpr int array_sz1 = 100;  // OK, 100 is a literal
  constexpr int array_sz2 = sz;   // FAIL – sz is not a constexpr

  double array[array_sz1];        // OK – array_sz1 known at compile-time
}
```

Since `constexpr` types have their value known at compile-time they can be placed in read-only memory; and, for built-in types (and provided you don't take their address) the compiler doesn't even have to store the object. We'll look at user-defined types in a moment.

constexpr functions

So far, so what?

The real power of *constant expressions* comes from the fact that not only objects but *functions* may be qualified as `constexpr`.

A `constexpr` function is one that may be evaluated at compile-time; and hence used in the definition of other *constant expressions*:

```
constexpr int getval()
{
  return 100;
}

constexpr int calc(int val)
{
  return val * getval();  // Calls another constexpr function
}


int main()
{
  double array[calc(2)];  // OK – calc() is a constexpr
  ...
}
```

Of course, there are some limitations with `constexpr` functions:

- They must have exactly one return statement
- They cannot have local variables
- They cannot have conditional expressions
- They cannot throw exceptions
- They cannot include a `goto` statement
- All parameters must be (capable of being) `constexpr` objects

Put simply, in order to be a `constexpr` function everything the function needs to be computed must be known at compile-time. Making a function a *constant expression* implies it is `inline`.

(Note: C++14 relaxes some of these requirements with respect to local variables and conditionals).

Marking a function as a `constexpr` doesn't limit it to only be usable at compile-time. They may still be used at run time:

```
int main()
{
  int a = 10;       // Non-constexpr object
  int b = calc(a);  // Evaluated at run-time
}
```

ROM-able types

The use of `constexpr` for functions extends to member functions, too. Marking a member function as a `constexpr` implicitly makes it a `const` function, meaning it cannot change any of the attributes of the class.

This is also true of constructors. Making a class's constructor a `constexpr` allows objects of that type to themselves be `constexpr` objects.

```
class Rommable
{
public:
  constexpr Rommable(int val) : data { val } { }
  constexpr int get() { return data; }

private:
  int data;
};


int main()
{
  constexpr Rommable romObject{ calc(100) }; // ctor is constexpr

  int val = romObject.get();                 // Read-only function
}
```

Since everything required to construct the `Rommable` object is known at compile-time it can be constructed in read-only memory.

In order to be truly ROM-able there are some additional requirements we must satisfy (over and above the requirements for a `constexpr` function):

- The class cannot have any virtual functions (virtual functions cannot be *constant expressions*)
- The class cannot have a virtual base class
- All base classes and all `const` members must be initialised by the constructor

Back to Contents

static_assert

C's `assert` library is a useful tool for catching invalid invariants (conditions that *must* holdtrue in order for your system to operate as specified) in your program. The big problem with `assert` is that it's a run-time check; in many cases the best you can do is restart the system or put it into a quiescent state.

In a lot of cases the (faulty) invariants could be detected at compile-time but in C++98 there was no mechanism for doing so. The pre-processor has an `assert`-like mechanism - `#error` – but the pre-processor has no knowledge of your C++ code so it cannot evaluate program invariants.

In C++11 a compile-time assert mechanism was added: `static_assert`. `static_assert` acts like run-time `assert` except that the checked-condition must be a *constant-expression* (that is, an expression that can be evaluated at compile-time); the checked-condition must be a Boolean expression, or an expression that can be converted to a boolean.

```
constexpr int calc(const int val)
{
  return val * sizeof(int);
}


int main()
{
  constexpr int sz = calc(20);
  static_assert(sz < 10, "Array size is too big");

  ...

  double array[sz];   // We know we're OK here...
}
```

Back to Contents

enum class

Enumerated types in C++ give a trivial simulation of *symbolic types* – that is, objects whose instances have unique, human-readable values. In C++ enumerations are essentially named integers that are either assigned values implicitly by the compiler or explicitly by the programmer (or a combination of both) C++ enum types inherit their semantics from C with some additions:

- enum objects are now first-class types
- enums may be implicitly converted to integers; but the reverse is not true

```
enum Colour { RED, GOLD, GREEN, BLUE };  // No typedef required
enum Metal  { GOLD, SILVER, BRONZE };    // Oops re-definition of GOLD!

Colour c = RED;                          // Colour is a new first-
                                         // class type

Metal m = GOLD;                          // <= FAIL! - this is Colour's
                                         // GOLD, not Metal's!

int a = c;                               // OK
c = 7;                                   // Not allowed
```

Another characteristic illustrated in the above code is the need for all enumerated symbols to be (globally) unique. This can be both frustrating and difficult to maintain for larger systems.

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.

```
enum class Colour { RED, GOLD, GREEN, BLUE };
enum class Metal  { GOLD, SILVER, BRONZE };

Colour c1 = RED;                         // FAIL – RED not defined
Colour c2 = Colour::RED;                 // OK

Metal m1 = Colour::GOLD;                 // FAIL (Obviously!)
Metal m2 = Metal::GOLD;                  // No redefinition now

int a = c2;                              // Now not allowed
int b = static_cast<int>(c2);            // Explicit cast required.
c = 7;                                   // Not allowed (as before)
```

One of the more confusing aspects of C++98 enumerationss was their size – although the type of an enum was defined (`int`) its size was implementation defined and required only to be a type big enough to hold the largest enumerated value. This means that an enumeration could be 8, 16, 32 or even 64 bits in size; and could change if new `enum` values were specified. This could (and probably has!) cause

alignment issues in embedded systems.

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.

```
enum EE : unsigned long {EE_ONE = 1, EE_TWO= 2, EE_BIG = 0xFFFFFFF0U};
```

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.

```
// Forward declaration of enum class:
//
enum class Color_code : char;


// Use of forward declaration
//
void foobar(const Color_code& p);

// Other code...

// Definition:
//
enum class Color_code : char { RED, YELLOW, BLUE, GREEN };
```

Back to Contents

nullptr

What's the value of a null pointer?

- `0`
- `NULL`
- `NUL`

No doubt you've been involved in the (always heated) discussions about which is the correct one (By the way: if you said `NUL` you need to take yourself to one side and give yourself a stern talking to).

The arguments tend to go something like this:

- `0` is the only 'well-known' value a pointer can be set to that can be checked.
- `NULL` is more explicit than just writing zero (even though it is just a macro definition wrapper)

The problem with using `0` or `NULL` is that they are, in fact, integers and that can lead to unexpected behaviours when function overloading occurs

```
void func(int i);
void func(int* ptr);

int main()
{
  func(NULL);  // Which version of func() gets called?
}
```

Based on what we've just discussed it should be pretty straightforward to see that the `int` overload will be called. (This rather weakens the argument that `NULL` is more explicit – explicitly confusing in this case!)

It gets worse: Implementations are free to define `NULL` as any integer type they like. In a 32-bit system it might seem reasonable to set `NULL` to the same size as a pointer:

```
#define NULL 0UL // 32-bit addresses
```

Sadly, this just adds confusion to our code:

```
void func(int i);
void func(int* ptr);
void func(long l);

int main()
{
  func(NULL);  // calls func(long)
}
```

In case you're a C programmer who's looked at this (and is feeling pretty smug at the moment) you may define `NULL` as follows:

```
#define NULL (void*)0L
```

Unfortunately, this doesn't work either, because a `void*` cannot be implicitly converted to an `int`, `long` or `int*` (or any other type) in C++.

In C++11, the answer to the question of the value of a pointer is much more simple: it is *always* `nullptr`.

`nullptr` is a keyword that represents any 'empty' pointer (and also any pointer-like objects; for example smart pointers) A `nullptr` is not an integer type, or implicitly convertible to one (it's actually of type `nullptr_t`). In our code, using `nullptr` instead of `NULL` always gives the results we expect.

```
void func(int i);
void func(int* ptr);
void func(long l);

int main()
{
  func(nullptr);    // calls func(int*);
}
```

Back to Contents

Using aliases

In a C++ program it is common to create type aliases using `typedef`. A type alias is not a new type, simply a new name for an existing declaration. Used carefully, `typedef` can improve the readability of code – particularly when dealing with complex declarations.

```
class ADT
{
public:
  void op(int);
};

template<typename T1, typename T2>     // Template class with two
class TC {};                           // template parameters

typedef unsigned long uint32_t;        // 1 - C-style type alias

typedef void (ADT::*ptr_mem_fn)(int)   // 2 - Pointer to member
                                       //     function on class ADT

typedef TC<int, int> MyTC;             // 3 - Alias for template TC,
                                       //     specifying explicit
                                       //     template parameters
```

In C++11 `typedef` can be replaced with a *using-alias*. This performs the same function as a `typedef`; although the syntax is (arguably) more readable. A using-alias can be used wherever a `typedef` could be used.

```
using uint32_t  = unsigned long;       // Same as (1)

using ptr_mem_fn = void(ADT::*)(int);  // Same as (2)

using MyTC = TC<int, int>;             // Same as (3)
```

Using-aliases have the advantage that they can also be templates, allowing a partial substitution of template parameters.

```
template<typename T>                   // Template alias, specifying T2
using MyTC = TC<T, int>;               // as explicitly int. cf (3)
```

Back to Contents

Uniform initialisation syntax

C++98 has a frustratingly large number of ways of initialising an object.

```
int i ;                     // Uninitialised built-in type
int j = 10;                 // Initialised built-in type
int k(10);                  // Initialised built-in type

int array[] = {1, 2, 3 };   // Aggregate initialisation
char str[] = "Hello";       // String literal initialisation

X x1;                       // Default constructor
X x2(10.7);                 // Non-default constructor
X x3 = x2;                  // Copy constructor
X x4 = 10.7;                // Copy-constructor (with elision)
```

(Note: not all these initialisations may be valid at the same time, or at all. We're interested in the syntax here, not the semantics of the class `X`).

One of the design goals in C++11 was *uniform initialisation syntax*. That is, wherever possible, to use a consistent syntax for initialising any object. The aim was to make the language more consistent, therefore easier to learn (for beginners), and leading to less time spent debugging.

To that end they added *brace-initialisation* to the language.

As the name would suggest, brace-initialisation uses braces (`{}`) to enclose initialiser values. So extending the above examples:

```
int i { };                  // Default initialised built-in type
int j { 10 };               // Initialised built-in type

X x1 { };                   // Default constructor
X x2 { 10.7 };              // Non-default constructor
X x3 { x2 };                // Copy constructor
X x4 = { 1, 3, 5, 7 };      // Construct as aggregate type
```

There are a couple of highlights from the above code.

- Integer `i` is default-initialised (with the value `0`). This is equivalent to C++03's (*much* more confusing):

  ```
  int i = int();
  ```

- `x1` is explicitly default-constructed. This alleviates the 'classic' mistake made by almost all C++ programmers at some point in their career:

  ```
  X x1();       // Declares a function, x1, that takes
                // no parameters and returns an X object
  ```

- Brace-initialisation also alleviates *C++'s Most Vexing Parse* as well. For those not familiar with it, here it is:

  ```
  ADT ad1( ADT() );
  ```

  Most programmers read this as "create an object, `adt`, and initialise it with a temporary object, `ADT()`". Your compiler, however, following the C++ parsing rules, reads it as "`adt` is a *function declaration* for a function returning an `ADT` object, and taking a (pointer to) a function with zero parameters, returning an `ADT` object." Don't believe me? (http://en.wikipedia.org/wiki/Most_vexing_parse)

  With brace-initialisation, this problem goes away:

  ```
  ADT adt1 { ADT {} };
  ```

  The compiler *cannot* parse the above except as "create an object, `adt`, and initialise it with a temporary object, `ADT{}`"

- Classes may be initialised in the same way as the built-in aggregate types using a `std::initializer_list` constructor overload; which we'll cover in the next article (so don't worry about it for now).

The uniform initialisation syntax goal means that brace-initialisation can be used *anywhere* an object must be initialised. This includes the member initialisation list:

```
class ADT
{
public:
  ADT( double x, double y) : x_pos { x }, y_pos { y } { /* ctor body */ }

private:
  double x_pos;
  double y_pos;
};
```

C++11 also introduced the ability to default-initialise class members. The code:

```
class ADT
{
public:
  ADT() : x_pos { 0.0 }, y_pos { 0.0 } {}

private:
  double x_pos;
  double y_pos;
};
```

Can be re-written as:

```
class ADT
{
public:
  ADT() = default;          // Tells the compiler to keep
                            // the default ctor if other ctors
                            // are added.
private:
  double x_pos { 0.0 };     // Use these values if no other
  double y_pos { 0.0 };     // initialiser value is specified.
};


int main()
{
  ADT adt1 { };             // adt1.x_pos => 0.0
                            // adt1.y_pos => 0.0
}
```

Back to Contents

std::initializer_list

An *aggregate type* in C++ is a type that can be initialised with a brace-enclosed list of initialisers. C++ contains three basic aggregate types, inherited from C:

- arrays
- structures
- unions

Since one of the design goals of C++ was to emulate the behaviour of built-in types it seems reasonable that you should be able to initialise user-defined aggregate types (containers, etc.) in the same way.

```
class ADT { /* ... */ };


class Aggregate                 // Acts like an array of ADT
{
public:
  Aggregate();
  ADT& operator[](int index);  // Allows array-like access

private:
  ADT data[16];
};


int main()
{
  ADT array[16]  = { ADT{1}, ADT{2}, ADT{3} };      // OK
  Aggregate aggr = { ADT{1}, ADT{2}, ADT{3} };      // FAIL!
}
```

A `std::initializer_list` is a template class that allows a user-defined type to become anaggregate type.

When initialiser list syntax is used the compiler generates a `std::initializer_list` object containing the initialisation objects. A `std::initializer_list` is a simple container class that may be queried for its size; or iterated through.

```
#include <initializer_list>

class Aggregate
{
public:
  Aggregate() = default;
  Aggregate(std::initializer_list<ADT> init);
  ADT& operator[](int index) { return data[index]; }

private:
  ADT data[16];
};
```

```
Aggregate::Aggregate(std::initializer_list<ADT> init)
{
  int index = 0;
  std::initializer_list<ADT>::iterator it;

  for(it = init.begin(); it != init.end(); ++it)
  {
    data[index++] = *it;
  }
}
```

If the class contains a constructor that takes a `std::initializer_list` as a parameter, this constructor is invoked and the `std::initializer_list` object passed.

```
int main()
{
  // Creates std::initializer_list<ADT>
  //
  Aggregate aggr = { ADT{1}, ADT{2}, ADT{3} };
}
```

There is some syntactic sugar at work here – the lack of brackets (`[]`) in the declaration of `aggr` forces the compiler to construct the `std::initializer_list` (then call `aggr`'s constructor) rather than creating an array of three `Aggregate` objects.

At this point we have to insert some words of caution. Since initialiser lists use brace-initialisation syntax there can be ambiguity over which constructor gets called. To resolve this, the compiler uses the following rules:

If a class has constructors overloaded for `T` and `std::initializer_list<T>` the compiler will always prefer the `std::initializer_list` overload; except in the case of an empty initialiser, where it will prefer the default constructor (if there is one)

To override the compiler's brace-initialisation resolution rules you have to resort to using C++98 parenthesis-initialisation syntax.

```
class X
{
public:
  X();
  X(int init_val);
  X(std::initializer_list<int> init_vals);
};


int main()
{
  X x1 { 10, 20 };  // Calls X::X(std::initializer_list<int>) with 2 elements
  X x2 { 10 };      // Calls X::X(std::initializer_list<int>) with 1 element
  X x3 ( 10 );      // Calls X::X(int)
  X x4 { };         // Calls X::X()
  X x5 ( );         // Function declaration!
}
```

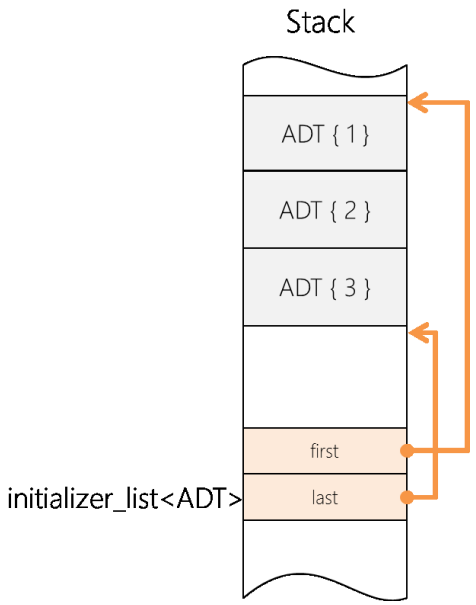
Initialiser lists can begin to look like so much magic and 'handwavium', so a brief look at an implementation of `std::initializer_list` is useful to dispel the mysticism:

```
template <typename T>
class initializer_list
{
public:
  initializer_list(const T* fst, const T* lst) : first{fst}, last{lst} {}

  const T* begin() { return first; }
  const T* last()  { return last;  }
  size_t   size()  { return static_cast<size_t>(last - first);  }

private:
  const T* first;
  const T* last;
};
```

When the compiler creates an `std::initializer_list` the elements of the list are constructed on the stack (or in static memory, depending on the scope of the `initializer_list`).



The compiler then creates the `initializer_list` object that holds the address of the first element and one-past-the-end of the last element. Note that the `initializer_list` is very small (two pointers) so can be passed by copy without a huge overhead; it does not pass the initialiser objects themselves. Once the `initializer_list` has been copied the receiver can access the elements and do whatever needs to be done with them.

Since C++11 all the STL containers support `initializer_list` construction; so now lists and vectors can be initialised in the same way as built-in arrays

```
class ADT { /*... */ };

int main()
{
  // Push back three ADT objects
  //
  vector<ADT> v = { ADT{ 1 }, ADT { 2 }, ADT { 3 } };
  ...
}
```

Back to Contents

Range-for loops

If you're using container classes in your C++ code (and you probably should be) then one of the things you're going to want to do (a *lot*) is iterate through the container accessing each member in turn.

Without resorting to STL algorithms we could use a for-loop to iterate through the container:

```
#include <vector>
#include <iostream>

using std::vector;
using std::cin;
using std::cout;

int main()
{
  vector<int> v; int val;

  // Add some data...
  //
  while(cin >> val) v.push_back(val);


  for(vector<int>::iterator it = v.begin();  // Read-write iterator.
      it != v.end();                         // Non-inclusive range.
      ++it)                                  // Always pre-increment!
  {
    // Manipulate vector element...
    //
    cout << static_cast<char>(*it + '0');
  }
}
```

If the above is baffling to you there are plenty of useful little tutorials on the STL on the Internet (For example: http://cs.brown.edu/~jak/proglang/cpp/stltut/tut.html)

We could simply the iterator declaration in C++11 using `auto`:

```
...
for(auto it = v.begin(); it != v.end(); ++it)
{
  // Manipulate vector element...
  //
  cout << static_cast<char>(*it + '0');

}
```

(See the article on `auto` type-deduction for details of how it works). However, there's a nicer syntactic sugar to improve our code: the *range-for* loop:

```
for(auto& item : v)
{
  // Manipulate vector element...
  //
  cout << static_cast<char>(item + '0');
}
```

The semantics of the *range-for* are: For every element in the container, `v`, create a reference to each element in turn, `item`. The above code is semantically equivalent to the following:

```
for (auto it_ = std::begin(v); it_ != std::end(v); ++it_)
{
  auto& item = *it_;
  cout << static_cast<char>(item +'0');
}
```

Look familiar?

Not only does this save you some typing but, because it's the compiler that's generating the code it has a lot more potential for optimisation (for example, the compiler may be able to decide that the `end()` iterator is not invalidated in the body of the *range-for* statement, therefore it can be read once before the loop; or the compiler may choose to unroll the loop; etc.)

In case you were wondering `std::begin()` and `std::end()` are adapter functions that return an iterator to the first element in the supplied container and an iterator to one-past-the-end, respectively. For most STL containers they simply call *`cont`*`.begin()` and *`cont`*`.end()`; but the functions are overloaded to handle built-in arrays and other container-like objects (see below)

Range-for loops are not limited to STL containers. They can also work with built-in arrays:

```
int main()
{
  int array[10];
  int init{};

  for(auto& i : array)  // i must be a reference to
  {                     // manipulate array elements
    i = init++;
  }
  // ...
}
```

And also initialiser lists:

```
int main()
{
  for(auto i : { 1, 5, 7, 9, 11 } ) // i is a copy of each element.
  {
    cout << i << " ";
  }
}
```

Back to Contents

override and final

override specifier

In C++98 using polymorphic types can sometimes lead to head-scratching results:

```
class Base
{
public:
  virtual void op(int i);
};


class Derived : public Base
{
public:
  virtual void op(long i);
};


void usePolymorphicObject(Base& b)
{
  p(100L);  // Calls Derived::op(), right?...
}


int main()
{
  Derived d;
  usePolymorphicObject(d);
}
```

On the face of it this code looks sound; indeed it will compile with no errors or warnings. However, when it runs the `Base` version of `op()` will be executed!

The reason? `Derived`'s version of `op()` is not actually an override of `Base::op` since `int` and `long` are considered different types (it's actually a conversion between an `int` and a `long`, not a promotion).

The compiler is more than happy to let you overload functions in the `Derived` class interface; but in order to call the overload you would need to (dynamic) cast the Base class object in `usePolymorphicObject()`.

In C++11 the `override` specifier is a compile-time check to ensure you are, in fact, overriding a base class method, rather than simply overloading it.

```
class Base
{
public:
  virtual void op(int i);
};


class Derived : public Base
{
public:
  virtual void op(long i) override;
};
```

```
ERROR: 'Derived::op': method with override specifier
       'override' did not override any base class methods
```

Final specifier

In some cases you want to make a virtual function a 'leaf' function – that is, no derived class can override the method. The `final` specifier provides a compile-time check for this:

```
class Base
{
public:
  virtual void op(int i);
};


class Derived : public Base
{
public:
  virtual void op(int i) override final;
};


class MoreDerived : public Derived
{
public:
  virtual void op(int i) override;  // <= Oops!
};
```

```
ERROR:  'Derived::op': function declared as 'final'
         cannot be overridden by 'MoreDerived::op'
```

Back to Contents

noexcept

We have some basic problems when trying to define error management in C:

- There is no "standard" way of reporting errors. Each company / project / programmer has a different approach
- Given the basic approaches, you cannot guarantee the error will be acted upon.
- There are difficulties with error propagation; particularly with nested calls.

The C++ exception mechanism gives us a facility to deal with run-time errors or fault conditions that make further execution of a program meaningless.

In C++98 it is possible to specify in a function declaration which exceptions a function may throw.

```
class Analog
{
public:
  double get_value(void);
  void   display(void)     throw();
  void   set_value(double) throw(char*,  Sensor_Failed);

private:
  double value;
};
```

The above function declarations state:

- `get_value()` can throw any exception. This is the default.
- `display()` will not throw *any* exceptions.
- `set_value()` can throw exceptions of *only* of type `char*` and `Sensor_Failed`; it cannot throw exceptions of *any other type.*

This looks wonderful, but compilers (can) only partially check exception specifications at compile-time for compliancy.

```
extern void process(void);                  // May throw ANY exception
void check(void) throw(std::out_of_range); // May throw only
                                            // std::out_of_range


void check(void) throw(std::out_of_range)
{
  process();                               // compiler does not check if
                                           // process(), or one of its
                                           // subordinates, only throws
                                           // std::out_of_range!
}
```

If `process()` throws an exception of any type other than `std::out_of_range` this will cause the exception handling mechanism – at run-time – to call the function `std::unexpected()` which, by default, calls `std::terminate()` (although its behaviour can – and probably should - be replaced).

Because of the limitations of compile-time checking, for C++11 the exception specification was simplified to two cases:

- A function may propagate any exception; as before, the default case
- A function may not throw *any* exceptions.

Marking a function as throwing no exceptions is done with the exception specifier, `noexcept`.

(If you read the noexcept documentation you'll see it can take a boolean constant-expression parameter. This parameter allows (for example) template code to conditionally restrict the exception signature of a function based on the properties of its parameter type. `noexcept` on its own is equivalent to `noexcept(true)`. The use of this mechanism is beyond the scope of this article.)

```
void mightThrow();                // Could throw any exception
void doesNotThrow() noexcept;     // Does not throw any exception
```

On the face of it, the following function specifications look semantically identical – both state that the function will not throw any exceptions:

```
void old_style() throw();

void new_style() noexcept;
```

The difference is in the run-time behaviour and its consequences for optimisation. With the `throw()` specification, if the function (or one of its subordinates) throws an exception, the exception handling mechanism must unwind the stack looking for a 'propagation barrier' – a (set of) `catch` clauses. Here, the exception specification is checked and, if the exception being thrown doesn't match the provided specification, `std::unexpected()` is called.

However, `std::unexpected()` can *itself* throw an exception. If the exception thrown by `std::enexpected()` is valid for the current exception specification, exception propagation and stack unwinding continues as before.

This means that there is little opportunity for optimisation by the compiler for code using a `throw()` specification. In contrast, in the case of a `noexcept` function specification `std::terminate()` is called immediately, rather than `std::unexpected()`. Because of this, the compiler has the opportunity to not have to unwind the stack during an exception, allowing it a much wider range of optimisations.

In general, then, if you know your function will *never* throw an exception, prefer to specify it as `noexcept`, rather than `throw()`.

Back to Contents

std::array

C++98 inherited C's only built-in container, the array. Arrays of non-class types behave in exactly the same way as they do in C. For class types, when an array is constructed the default constructor is called on each element in the array.

```
class Position
{
public:
  Position = default;
  Position(double lon, double lat);

private:
  double longitude { 0.0 };
  double latitude  { 0.0 };
};


int main()
{
  Position track[5];  // Default constructor is called for each element
}
```

Explicitly initialising objects in an array is one of the few times you can explicitly call a class' constructor.

```
int main()
{
  Position track[5] =
  {
    Position { 0.0,   0.0  },
    Position { 90.0,  45.0 },
    Position { 180.0, 90.0 }
  };
}
```

For `track[]`, the non-default constructor is called for first three elements, followed by the default (no parameter) constructor for the last two elements; hence they are `0.0`.

(Note the performance implications of this – five constructor calls will be made whether you explicitly initialise the objects or not.)

Arrays are referred to as 'degenerate' containers. They are basically a contiguous sequence of memory, pointers, and some syntactic sugar. This can lead to some self-delusion on the part of the programmer.

```
#define array_sizeof(a) (sizeof(a) / sizeof(a[0]))


void process(Position track[5])
{
  size_t sz = array_sizeof(track);  // Probably 0!
  track++;                          // <= Compiles! Eh?!
}


int main()
{
  Position track[5];
  size_t sz = array_sizeof(track);  // Yields 5, as expected
  track++;                          // FAIL – as expected
}
```

Despite the fact that the declaration of `process()` appears to specify an array of five `Position` objects, it is in fact a simple `Position*` that is passed. This explains why the `array_sizeof` macro fails (since the size of a `Position` is greater than the size of a pointer!). It also explains why we can increment the array name (which *should* be a constant) – as it is in `main()`).

In C++11, use of 'raw' arrays is undesirable; and there are more effective alternatives.

`std::array` is fixed-size contiguous container. The class is a template with two parameters – the type held in the container; and the size.

```
#include <array>

int main()
{
  std::array<Position, 10> track;  // Default ctor called on all
                                   // 10 Position objects

  cout << a[5]  << endl;           // No bounds-checking
  cout << a.at(5) << endl;         // Bounds-checked access
}
```

`std::array` does not perform any dynamic memory allocation. Basically, it is a thin wrapper around C-style arrays. Memory is allocated – as with built-in arrays – on the stack or in static memory. Because of this, and unlike `std::vector`, `std::array`s cannot be resized.

If C-style notation is used there is no bounds-checking on the `std::array`; however, if the `at()` function is used an exception (`std::out_of_range`) will be thrown if an attempt is made to access outside the range of the array.

std::arrays have the advantage (over the built-in array) that they support all the facilities required by the STL algorithms so they can be used wherever a vector or list (etc.) could be used; without the overhead of dynamic memory management.

```
int main()
{
  std::array<Position, 10> track =   // Note:
  {                                  // NOT using std::initializer_list!
     Position {   0.0,  0.0 },       // The internal 'raw' array is being
     Position {  90.0, 45.0 },       // initialised here (exactly as above)
     Position { 180.0, 90.0 }
  };

  for(auto& pos : track)
  {
    cout << pos.asString() << endl;
  }
}
```

Finally, because container types are classes (not syntactic sugar) they can be passed around the system like any other object.

```
void process(std::array<Position, 10>& track)
{
  cout << "Processing " << track.size() << " elements" << endl;

  for(auto& pos : track)
  {
    cout << pos.asString() << endl;
  }

}


int main()
{
  std::array<Position, 10> track;

  process(track);  // Pass just as with any other object.
}
```

Back to Contents

Smart pointers

The dynamic creation and destruction of objects was always one of the bugbears of C. It required the programmer to (manually) control the allocation of memory for the object, handle the object's initialisation then ensure that the object was safely cleaned-up after use and its memory returned to the heap. Because many C programmers weren't educated in the potential problems (or were just plain lazy or delinquent in their programming) C got a reputation in some quarters for being an unsafe, memory-leaking language.

Things didn't significantly improve in C++. We replaced `malloc` and `free` with `new` and `delete`; but the memory management issue remained.

```
class X { /* ... */ };

void func(X* theX)
{
  theX = new X;
}


int main()
{
  X *px = new X;
  func(px);       // Oops! Memory leak!

  delete px;
}
```

I concede – the code above is trivial and stupid but I suspect if I looked around I could find similar (or even worse!) examples in actual production code.

Languages such as Java and C# solved this problem by taking memory management out of the hands of the programmer and using a garbage collector mechanism to ensure memory is cleaned up when not in use.

In Modern C++ they have chosen not to go down this route but make use of C++'s Resource Acquisition Is Initialisation (RAII) mechanism and encapsulate dynamic object creation / destruction within smart pointers.

A smart pointer is basically a class that has the API of a 'raw' pointer. In Modern C++ we have four classes for dynamic object management:

std::auto_ptr
Single-owner managed pointer, from C++98; now deprecated

std::shared_ptr
A reference-counted pointer, introduced in C++98 TR1

std::unique_ptr
Single-owner managed pointer which replaces (the now deprecated) `auto_ptr`

std::weak_ptr
Works with `shared_ptr` in situations where circular references could be a problem

Avoid using std::auto_ptr

`std::auto_ptr` was introduced in C++98 as a single-owner resource-managed smart pointer. That is, only one `auto_ptr` can ever be pointing at the resource.

`auto_ptr` objects have the peculiarity of taking ownership of the pointers assigned to them: An `auto_ptr` object that has ownership over one element is in charge of destroying the element it points to and to deallocate the memory allocated to it when itself is destroyed. The destructor does this by calling operator `delete` automatically.

```
#include <memory>

class X
{
  public: void op();
};

int main()
{
  std::auto_ptr<X> p1{new X};
  std::auto_ptr<X> p2{new X};

  p1->op();
  p1 = p2;    // Take ownership, rather than copy.

  p2->op();   // What happens here?!
}
```

When an assignment operation takes place between two `auto_ptr` objects, ownership is transferred, which means that the object losing ownership is set to no longer point to the element (it is set to `nullptr`).

This could lead to unexpected null pointer dereferences - an unacceptable consequence for most (if not all) systems. Therefore, we recommend avoiding the use of `auto_ptr`. It has now been deprecated in C++11 (and replaced with the much more consistent `std::unique_ptr`)

Use std::unique_ptr for single ownership

`std::unique_ptr` allows single ownership of a resource. A `std::unique_ptr` is an RAII wrapper around a 'raw' pointer, therefore occupies no more memory (and is generally almost as fast) as using a raw pointer. Unless you need more complex semantics, `unique_ptr` is your go-to smart pointer.

`unique_ptr` does not allow copying (by definition); but it does support move semantics, so you can explicitly transfer ownership of the resource to another `unique_ptr`.

```
using namespace std;

int main()
{
  unique_ptr<X> p1{ new X };      // NB: You can't do:
                                  // std::unique_ptr<X> p1 = new X;

  auto p2 = make_unique<X>();     // Preferred mechanism for dynamic
                                  // objects.

  p1->op();

  unique_ptr<X> p3{ p1 };         // ERROR - Copy construction
  unique_ptr<X> p4{ move(p1) };   // OK - move constructor called

  p1 = p2;                        // ERROR – Can't copy.
  p1 = move(p2);                  // p1 => p2
                                  // p2 => nullptr
}
```

The utility function `make_unique<T>()` hides away the memory allocation and is the preferred mechanism for dynamically creating objects. `make_unique<T>()` is not officially supported in C++11; but it is part of C++14 and is supported by many C++11-compliant compilers. (Its omission appears to have been a mistake in the C++11 standard.)

For sharing a resource, use std::shared_ptr

`std::shared_ptr` is a reference-counted smart pointer.

Creating a new dynamic object also creates a new associated management structure that holds (amongst other things) a reference count of the number of `shared_ptr`s currently 'pointing' at the object.

Each time a `shared_ptr` is copied the reference count is incremented. Each time one of the pointers goes out of scope the reference count on the resource is decremented. When the reference count is zero (that is, the last `shared_ptr` referencing the resource goes out of scope) the resource is deleted.

`std::shared_ptr`s have a higher overhead (in memory and code) than `std::unique_ptr` but they come with more sophisticated behaviours (like the ability to be copied).

```
void func(std::shared_ptr<X> p)  // Pass-by-value; copy made on call
{                                // ref count incremented
  p->op();
}                                // parameter object deleted;
                                 // ref count decremented
```

```
int main()
{
  auto ptr = std::make_shared<X>(100); // Creates a new shared_ptr<X>.

  ptr -> op();
  func(ptr);                          // Copy ptr; increment ref count.
                                      // Last shared_ptr goes out of
}                                     // scope; delete memory.
```

Once again, the standard library provides a utility function `make_shared<T>()` for creating shared dynamic objects; and, once again, this is the preferred mechanism.

Avoid circular dependencies with std::weak_ptr

A `std::weak_ptr` is related to a `std::shared_ptr`. Think of a `weak_ptr` as a 'placeholder' for a `shared_ptr`. `std::weak_ptr`s are useful if you need to break cyclic dependencies between `shared_ptr`s (A topic that is outside the scope of this article)

When you create a `weak_ptr` it must be constructed with an extant `shared_ptr`. It then becomes a 'placeholder' for that `shared_ptr`. You can store `weak_ptr`s, copy and move them, but doing so has no effect the reference count on the resource.

```
void func(std::weak_ptr<int> p);              // Note p is pass-by-value.

int main()
{
  auto shared = std::make_shared<int>(100);  // shared ctor: count = 1
  std::weak_ptr<int> weak{shared};           // weak ctor:   count = 1

  *weak = 200;                               // ERROR!

  func(weak);                                // weak copy:   count = 1
}                                            // weak dtor:   count = 1
                                             // shared dtor: count = 0
```

Notice you cannot directly use a `weak_ptr`. You must convert it back to a `shared_ptr` first. `weak_ptr`s have a method, `lock()`, that creates (in effect) a copy of the original `shared_ptr`, which can then be accessed.

```
void func(std::weak_ptr<int> p)              // p:                 count = 1
{
  if(!p.expired())                          // Is p valid?
  {
    auto temp = p.lock();                    // Return shared_ptr: count = 2
    *temp = 200;=
  }                                          // temp dtor:         count = 1
}                                            // p dtor:            count = 1
```

Since `weak_ptr`s can have a different lifetime to their associated `shared_ptr` there is a chance the original `shared_ptr` could go out of scope (and delete its resource) before the `weak_ptr` is destroyed. A `weak_ptr` can therefore be invalid - that is, referencing a resource that is no longer viable. You should use the `expired()` method on the `weak_ptr` to see if it is still valid, before attempting to access it; alternatively, calling `lock()` on an expired `weak_ptr` will return `nullptr` if it has expired.

Back to Contents

About Feabhas

Feabhas is the UK's leading independent provider of training and consultancy for real-time embedded systems development and software competency. It provides on-site team development, public training for individual engineers, consultancy and mentoring, as well as pre- and post-course assessments.

Feabhas was formed in 1995 and has trained over 15,000 engineers worldwide to date, helping them to improve their embedded software competency and reduce software development times and costs.

Feabhas is an ARM Approved Training Centre and can also help with developing software standards e.g. DO-178C, ISO 26262, IEC 62304, EN 50128 etc., graduate training program and re-skilling from other disciplines.

Glennan Carnie is a Technical Consultant at Feabhas. He is an embedded systems and software engineer with over 20 years' experience in high-integrity systems for the defence and aerospace industry. He has also worked in systems development and support in the banking industry and the pre-press digital artwork delivery industry.

Glennan's training repertoire includes courses to an Advanced level for C, C++, UML, SysML, software testing and embedded software engineering.

He is also a frequent Blogger (Sticky bits) and has particular interest in raising the quality standards of software engineering and software process development, including: the development methodologies and processes, configuration management (revision control, change management, etc.), testing, validation and verification, and project management.

Back to Contents