Obscure C++ Features

madebyevan.com/obscure-cpp-features

This page is a collection of obscure C++ features, gathered over the years as I've explored different corners of the language. C++ is very big and I'm always learning more about it. Hopefully you'll learn something from this page even if you already know C++ pretty well. The features below are roughly ordered from least to most obscure.

What square brackets really mean

Accessing an element of an array via `ptr[3]` is actually just short for `*(ptr + 3)`. This can be equivalently written as `*(3 + ptr)` and therefore as `3[ptr]`, which turns out to be completely valid code.

Most vexing parse

The "most vexing parse" is a term coined by Scott Meyers for an ambiguity in C++ declaration syntax that leads to counterintuitive behavior:

```
// Is this:
// 1) A variable of type std::string initialized to a std::string()?
// 2) The declaration of a function that returns a std::string and has one argument,
//    which is a pointer to a function with no arguments that returns a std::string?
std::string foo(std::string());

// Is this:
// 1) A variable of type int initialized to int(x)?
// 2) The declaration of a function that returns an int and has one argument,
//    which is an int named x?
int bar(int(x));
```

The C++ standard requires the second interpretation in both cases, even though the first interpretation is the intuitive one. Programmers can disambiguate by enclosing the initial value of the variable in parentheses:

```
// Parentheses resolve the ambiguity
std::string foo((std::string()));
int bar((int(x)));
```

The reason for the ambiguity in the second case is that `int y = 3;` is equivalent to `int(y) = 3;`.

Alternate operator tokens

The tokens `and` , `and_eq` , `bitand` , `bitor` , `compl` , `not` , `not_eq` , `or` , `or_eq` , `xor` , `xor_eq` , `<%` , `%>` , `<:` , and `:>` can be used instead of the symbols `&&` , `&=` , `&` , `|` , `~` , `!` , `!=` , `||` , `|=` , `^` , `^=` , `{` , `}` , `[` , and `]` . They let you type operators on keyboards that lack the necessary symbols.

Redefining keywords

Redefining keywords via the preprocessor is technically supposed to cause an error but tools allow it in practice. This lets you do fun bug-introducing stuff like `#define true false` or `#define else` . However, there are times it is legitimately useful. For example, if you're using a large library and you need to bypass the C++ access protection mechanism, instead of patching the library you can just turn off access protection before including the headers for the library. Remember to turn the protection back on afterwards!

```
#define class struct
#define private public
#define protected public

#include "library.h"

#undef class
#undef private
#undef protected
```

Note that this may not necessarily work depending on your compiler. C++ only requires instance variables to be laid out sequentially when they are not separated by an access specifier, so the compiler is free to change the memory layout by reordering access specifier groups. For example, a compiler is allowed to move all private members so they come after after all public members. Another potential problem is name mangling; Microsoft's C++ compiler incorporates the access specifier into their name mangling scheme so changing the access specifier will break compatibility with existing compiled code.

Placement new

Placement new is an alternate syntax for the `new` operator that runs in place on an already allocated object, which is assumed to be the correct size and have the correct alignment. This involves setting up the vtable and calling the constructor.

```
#include <iostream>
using namespace std;

struct Test {
  int data;
  Test() { cout << "Test::Test()" << endl; }
  ~Test() { cout << "Test::~Test()" << endl; }
};

int main() {
  // Must allocate our own memory
  Test *ptr = (Test *)malloc(sizeof(Test));

  // Use placement new
  new (ptr) Test;

  // Must call the destructor ourselves
  ptr->~Test();

  // Must release the memory ourselves
  free(ptr);

  return 0;
}
```

Placement new is used when writing custom allocators for performance-critical scenarios. For example, a slab allocator starts with a single large chunk of memory and uses placement new to allocate objects sequentially within the chunk. This avoids memory fragmentation and the overhead of heap traversal that malloc incurs.

Branch on variable declaration

C++ contains a syntactical shorthand for simultaneously declaring a variable and branching on its value. It looks like a single variable declaration and can go in the condition of an `if` or `while` statement.

```
struct Event { virtual ~Event() {} };
struct MouseEvent : Event { int x, y; };
struct KeyboardEvent : Event { int key; };

void log(Event *event) {
  if (MouseEvent *mouse = dynamic_cast<MouseEvent *>(event))
    std::cout << "MouseEvent " << mouse->x << " " << mouse->y << std::endl;

  else if (KeyboardEvent *keyboard = dynamic_cast<KeyboardEvent *>(event))
    std::cout << "KeyboardEvent " << keyboard->key << std::endl;

  else
    std::cout << "Event" << std::endl;
}
```

Ref-qualifiers on member functions

C++11 allows member functions to be overloaded on the value type of the object that will be used for `this` using a ref-qualifier. A ref-qualifier sits in the same position as a cv-qualifier and affects overload resolution depending on if the object for `this` is an lvalue or an rvalue:

```
#include <iostream>

struct Foo {
  void foo() & { std::cout << "lvalue" << std::endl; }
  void foo() && { std::cout << "rvalue" << std::endl; }
};

int main() {
  Foo foo;
  foo.foo(); // Prints "lvalue"
  Foo().foo(); // Prints "rvalue"
  return 0;
}
```

Turing complete template metaprogramming

C++ templates are for compile-time metaprogramming, which means programs that generate other programs. The template system is designed for simple type substitutions but it was discovered by accident during the C++ standardization process that templates are actually powerful enough to perform arbitrary calculations, albeit very awkwardly and inefficiently. Computation is done via template specialization:

```
// Recursive template for general case
template <int N>
struct factorial {
  enum { value = N * factorial<N - 1>::value };
};

// Template specialization for base case
template <>
struct factorial<0> {
  enum { value = 1 };
};

enum { result = factorial<5>::value }; // 5 * 4 * 3 * 2 * 1 == 120
```

C++ templates can be thought of as a functional programming language since they use recursion instead of iteration and contain no mutable state. You can create a variable that holds a type via `typedef` and a variable that holds an `int` via `enum`. Data structures are embedded in types themselves:

```
// Compile-time list of integers
template <int D, typename N>
struct node {
  enum { data = D };
  typedef N next;
};
struct end {};

// Compile-time sum function
template <typename L>
struct sum {
  enum { value = L::data + sum<typename L::next>::value };
};
template <>
struct sum<end> {
  enum { value = 0 };
};

// Data structures are embedded in types
typedef node<1, node<2, node<3, end> > > list123;
enum { total = sum<list123>::value }; // 1 + 2 + 3 == 6
```

While these examples are pretty useless, template metaprogramming enables some useful things like being able to manipulate lists of types. However, the programming language formed by C++ templates has terrible usability, so try to use it sparingly and in small amounts. Template code is hard to read, slow to compile, and very difficult to debug due to incredibly long and cryptic compiler error messages.

Pointer-to-member operators

Pointer-to-member operators let you describe a pointer to a certain member on any instance of a class. There are two pointer-to-member operators, `.*` for values and `->*` for pointers:

```
#include <iostream>
using namespace std;

struct Test {
  int num;
  void func() {}
};

// Notice the extra "Test::" in the pointer type
int Test::*ptr_num = &Test::num;
void (Test::*ptr_func)() = &Test::func;

int main() {
  Test t;
  Test *pt = new Test;

  // Call the stored member function
  (t.*ptr_func)();
  (pt->*ptr_func)();

  // Set the variable in the stored member slot
  t.*ptr_num = 1;
  pt->*ptr_num = 2;

  delete pt;
  return 0;
}
```

This feature is actually really useful, particularly for writing libraries. For example, Boost::Python (a library for binding C++ to Python objects) uses member pointers to easily refer to members when wrapping objects:

```
#include <iostream>
#include <boost/python.hpp>
using namespace boost::python;

struct World {
  std::string msg;
  void greet() { std::cout << msg << std::endl; }
};

BOOST_PYTHON_MODULE(hello) {
  class_<World>("World")
    .def_readwrite("msg", &World::msg)
    .def("greet", &World::greet);
}
```

Keep in mind when using member function pointers that they are different from regular function pointers. Casting between a member function pointer and a regular function pointer will not work. For example, member functions in Microsoft's compilers use an optimized calling convention called thiscall that puts the `this` parameter in the `ecx` register, while normal functions use a calling convention that passes all arguments on the stack.

Also, member function pointers may be up to four times larger than regular pointers. The compiler may need to store the address of the function body, the offset to the correct base (multiple inheritance), the index of another offset in the vtable (virtual inheritance), and maybe even the offset of the vtable inside the object itself (for forward declared types).

```
#include <iostream>

struct A {};
struct B : virtual A {};
struct C {};
struct D : A, C {};
struct E;

int main() {
  std::cout << sizeof(void (A::*)()) << std::endl;
  std::cout << sizeof(void (B::*)()) << std::endl;
  std::cout << sizeof(void (D::*)()) << std::endl;
  std::cout << sizeof(void (E::*)()) << std::endl;
  return 0;
}

// 32-bit Visual C++ 2008:  A = 4, B = 8, D = 12, E = 16
// 32-bit GCC 4.2.1:        A = 8, B = 8, D = 8,  E = 8
// 32-bit Digital Mars C++: A = 4, B = 4, D = 4,  E = 4
```

All member function pointers in the Digital Mars compiler are the same size due to a clever design that generates "thunk" functions to apply the right offsets instead of storing the offsets in the pointer itself.

Static methods on instances

C++ lets you invoke static methods from an instance in addition to invoking them from the type. This lets you change an instance method to a static method without needing to update any call sites.

```
struct Foo {
  static void foo() {}
};

// These are equivalent
Foo::foo();
Foo().foo();
```

Overloading ++ and --

C++ is designed so the function name of custom operators is the operator symbol itself, which works fine in most cases. For example, the unary `-` and binary `-` operators (negation and subtraction) can be distinguished by the argument count. This doesn't work for the unary increment and decrement operators though since they both seem to need the exact same signature. The C++ language has an ugly hack to work around this: the postfix `++` and `--` operators must take a dummy `int` argument as a flag for the compiler to know to make a postfix operator (and yes, only the type `int` works).

```
struct Number {
  Number &operator ++ (); // Generate a prefix ++ operator
  Number operator ++ (int); // Generate a postfix ++ operator
};
```

Operator overloading and evaluation order

Overloading the `,` (comma), `||`, or `&&` operators is very confusing because it destroys the normal evaluation rules. Normally, the comma operator guarantees that the entire left side will be evaluated before evaluation starts on the right side and the `||` and `&&` operators have short-circuit behavior that only evaluates the right side when necessary. However, the overloaded versions of these operators are just function calls, and function calls evaluate their arguments in an unspecified order.

Overloading these operators is just a way to abuse C++ syntax. As an example, I give you a C++ implementation of a Python-style print statement that doesn't need parentheses:

```
#include <iostream>

namespace __hidden__ {
  struct print {
    bool space;
    print() : space(false) {}
    ~print() { std::cout << std::endl; }

    template <typename T>
    print &operator , (const T &t) {
      if (space) std::cout << ' ';
      else space = true;
      std::cout << t;
      return *this;
    }
  };
}

#define print __hidden__::print(),

int main() {
  int a = 1, b = 2;
  print "this is a test";
  print "the sum of", a, "and", b, "is", a + b;
  return 0;
}
```

Functions as template parameters

It's well known that template parameters can be specific integers but they can also be specific functions. This lets the compiler inline calls to that specific function in the instantiated template code for more efficient execution. In the example below, the function `memoize` takes a function as a template parameter and only calls the function for new argument values (old argument values are remembered from the cache).

```
#include <map>

template <int (*f)(int)>
int memoize(int x) {
  static std::map<int, int> cache;
  std::map<int, int>::iterator y = cache.find(x);
  if (y != cache.end()) return y->second;
  return cache[x] = f(x);
}

int fib(int n) {
  if (n < 2) return n;
  return memoize<fib>(n - 1) + memoize<fib>(n - 2);
}
```

Template template parameters

Template parameters can actually have template parameters themselves. This allows you to pass templated types without template parameters when instantiating a template. Say we have the following code:

```
template <typename T>
struct Cache { ... };

template <typename T>
struct NetworkStore { ... };

template <typename T>
struct MemoryStore { ... };

template <typename Store, typename T>
struct CachedStore {
  Store store;
  Cache<T> cache;
};

CachedStore<NetworkStore<int>, int> a;
CachedStore<MemoryStore<int>, int> b;
```

CachedStore puts a cache that holds a certain data type in front of a store that stores the same data type. However, we must repeat the data type (`int` in the code above) when instantiating a CachedStore, once for the store itself and once for CachedStore, and there's no guarantee that the data types are consistent. We really want to just specify the data type once so we can enforce this invariant, but leaving off the type parameter list causes a compile error:

```
// These do not compile because NetworkStore and MemoryStore are missing type parameters
CachedStore<NetworkStore, int> c;
CachedStore<MemoryStore, int> d;
```

Template template parameters let us get the syntax we want. Note that you need to use the `class` keyword for template parameters that themselves have template parameters.

```
template <template <typename> class Store, typename T>
struct CachedStore2 {
  Store<T> store;
  Cache<T> cache;
};

CachedStore2<NetworkStore, int> e;
CachedStore2<MemoryStore, int> f;
```

Function try blocks

Function try blocks exist to catch errors thrown while evaluating a constructor's initializer list. You can't wrap a normal try-catch block around the initializer list because it exists outside the function body. To fix this, C++ allows a try-catch block to serve as the body of a function:

```
int f() { throw 0; }

// Here there is no way to catch the error thrown by f()
struct A {
  int a;
  A::A() : a(f()) {}
};

// The value thrown from f() can be caught if a try-catch block is used as
// the function body and the initializer list is moved after the try keyword
struct B {
  int b;
  B::B() try : b(f()) {
  } catch(int e) {
  }
};
```

Oddly enough, this syntax isn't just limited to constructors but is available for all function definitions.