Basic Facilities

- • S 6. Look for the new C++11 concepts.
  • S 7 but skim 7.3.2.1 raw string literals, skim 7.3.2.2 unicode topics, slow down and read carefully 7.7 on rvalue reference - new in C++11.
  • S 8 skim 8.2.6 on POD, skim 8.2.7 Fields, skim 8.3 Unions introduction, then skip 8.3.1 and 8.3.2.
  • S 9. should be very familiar.
  • S 10 but skim 10.2 calculator example - read for concepts involved, not details.
  • S 11 Skip 11.2.4 on overloading new. 11.3 "Lists" are C++11's initialization lists. Skip 11.4 Lambda Expressions - we'll come back.
  • S 12 Functions. Skip 12.2.3 List Arguments; skip 12.2.4 Unspecified number of arguments. Skip 12.5 and 12.6 - redundant with C coverage.

S 6 Types and Declarations:

char type

- *we'll be using plain char throughout, as recommended (p. 143), and you should avoid doing arithmetic with char values - not needed; let stdlib do the relevant work for you.*

int type

- *we won't be using unsigned types (except via size_t or similar definied types), and plain int only.*

- **literal types - new in C++11 - we won't be using**

declaration terminology

- *optional specifier, base type, declarator, initializer*
- *specifier is non-type modifier*
- *base type is the type*
- *declarator is a name and optional operators: *, * const, &, [], () both prefix and postfix, like use in expressions is the idea*
  - postfix bind tighter than prefix - *kings[] is an array of pointers
  - sometimes need parentheses
- *New in C++11 automatic type deduction in declarations*
  - using the now obsolete "auto" keyword for the type of a variable means to use the type of the initializer.
    - auto i = 42; // i is an int
      auto x = 3.14; // double
      auto x = foo(); // x is whatever type foo() returns
  - especially useful later when types are complicated:
    - map<int, string, Cmp>::const_iterator it = container.begin();
      auto it = container.begin(); // it has whatever iterator type the container begin() returns
  - my advice - save auto for special occasions, especially when type name is either very complex or totally redundant; best to be explicit about simple types for most variables.

scope

- *block - or local scope*
  - function parameter names are actually declared in the outer most block of a function
- *class scope*
  - member names
- *namespace scope*
- *global - outside any function, class, or namespace*
  - global namespace is a namespace
  - scope resolution operator can unhide a global name (p 158)
- *try to avoid hiding names by choosing global or outer scope names carefully*
- *globals have a global scope, can use the scope resolution operator to specify them*
- *names come into scope after the complete declarator and before the initializer*
  - struct Thing * p ... declares the incomplete type struct Thing

initialization

- *if no initializer, and the variable is global or local static (just static actually), it gets initialized to the appropriate flavor of zero; ditto for global or local static structs or arrays. User defined types are default initialized.*
- *arrays and structs can be initialized by lists of values in { }*
- *for user defined types, "function style initializer" from invoking a possibly implicit constructor*
  - Point p(1, 2);
  - note int f(); is a function declaration
  - Point p; would default initialize p, not Point p();
    - Point p(); declares "p" to be a function with no arguments that returns a Point!
- *New in C++11:*
  - uniform 3 with {curly braces} added to the language - can do all initializations with { }, like you can with arrays and structs in C and C++98.
  - Point p1; // default intialize
    Point p1{}; // default intialize - now distriguishable from function declaration syntax!
    Point p{1, 2};
    Point p1 = {1, 2};

    int* a = new int[3] {3, 2, 1}; // peviously no way to do this
  - ```
    // narrowing conversions are an error with {} initialization
        int i{4};
    ```
    ```

```
    int ii = {4};
//    int iii{1.2}; // not allowed
    int iiii = 1.2; // legal, but possible error
    double d{3};
    float f{3};
```

- complex classes like vector can now be initialized with { } analogous to arrays - later

Unnamed objects

Suppose we have a class Point that we can initialize with x, y values as in:

- Point p1(12, 23);
- This declares and defines Point object named "p1" initialized with 12 and 23;

If you leave out the name, then you are declaring an "unnamed" object.

- Point(12, 23);
- This declares and defines a Point object initialized with 12 and 23 that has no name, a temporary object.
- Temporary objects disappear once you leave the "full expression" they are in.
  - Compiler creates these as temporary object in an expression or function call
  - Example

```
string x  ="hello,";
string y = "world";
string z = x + y;

foo(x + y);  // where foo is foo(string s); or foo(const string& s);
```

    - x + y creates a temporary object, used to hold the "hello,world" long enough to initialize z, then it is gone
    - can be a performance issue with user-defined types, but rarely for built-in types
    - C++11 enables "move semantics" that alleviate unnecessary copying of data from temporary objects
  - Examples

```
// new_location is (12, 23) translated by vector1.
Point new_location = Point(12, 23) + vector1;

double distance(Point p1, Point p2);  // calculates distance between two points
d = distance(Point(12, 23), Point(58, 14));

Point get_Point()
{
    /* get x and y values from the user */
    return Point(x, y);  // temporary Point object to copy for return
}
```

- Unnamed objects are very commonly used in some contexts.

- **objects and lvalues**
  - *an "object" is a piece of memory; an "lvalue" is an expression that designates a piece of memory*
    - basic notion - you can store somethig there, or use its address
    - roughly the l in lvalue is for left hand side of an assignment
    - usually it has a name - like a variable name
    - actually, if it has a variable name, it is treated as an lvalue by the compiler
    - but some lvalues can't be used there, and some lvalues refer to a constant
  - *New in C++11 - important role for rvalues*
    - roughly speaking, the r iin rvalue is for right hand side of an assignment
    - temporary unnamed variables are rvalues
      example: x = y + z; // the y + z is in an rvalue
      - it is a place in memory whose value you can always read, but normally it doesn't make any sense to store something there because it will soon disappear.
        - C++98 disallows modifying an rvalue
      - But there are a couple of important occasions for making the lvalue/rvalue distrinction, and modifying what is stored in an rvalue - later!
  - *See chart p 166*
    - objects can have identity; objects can be moved from rather than just copied.
    - 3 of the four combinations are used:
    - lvalue xvalue prvalue
      glvalue rvalue
    - prvalue - pure rvalue, like function return value,, expression value - no name, can be moved from
    - xvalue - has name, but can be moved from (e.g. std::move applied to iit_
    - glvalue - generalized lvalue - has identity either movable or not
    - rvalue - can be moved from
- **lifetime**
  - *automatic*
  - *stastic*
  - *free store (dynamic)*
  - *temporary*
    - if bound to a reference, lifetime of reference, otherwise end of full expresssion, typically automatic.

- **Type aliases**
  - *using mytype = existing_type;*
  - *usually equivalent to a typedef, but more flexible with templates:*
    - tempplate<typename T>
      using Vector = std::vector<T>;
    - /* template <typename T>
      typedef Ordered_list<T> myOL; // error typedef can't be a template
      */
      template <typename T>
      using myOL = Ordered_list<T>;

      template <typename T>
      myOL<T> foo(myOL<T> x)
      {return x;};

S. 7. Pointers, Arrays, References

- S 7 but skim 7.3.2.1 raw string literals, skim 7.3.2.2 unicode topics, slow down and read carefully 7.7 on rvalue reference - new in C++11.

pointers and zero as a pointer value

- *use zero, or better, **nullptr**, instead of NULL*
- *zero takes on a type depending on its context*
- *foo(0) could call either foo(int) or foo(char *) - which?*
- *always use nullptr instead of a 0 pointer value. Using NULL is now doubly-obsolete in C++*

void *

- *should only show up in C++ code at down & dirty low-levels; bad idea otherwise*
- *note static_cast<double *>(pv) example - deliberately ugly*

references

- *must be initialized at point of definition*
- *can't be changed to refer to something else - can't be "reseated"*
- *two ways to think of them*
  - another name for a object
  - constant pointers where the compiler sticks in the & and *'s for you
- *main use is function parameters & return types*
- *can be used otherwise, but rare*
- *tricky because you never operate on a reference, always on the thing it refers to - it really is just another name ...*
- *S's advice is to avoid reference arguments as returned values unless function name makes it obvious that it is going to happen.*
- *returning a reference is a way to let the caller know where to put something - e.g. subscript operator ...*
- *returning a reference can avoid object copying*
- *New in C++11: may be called "lvalue reference" when important to distinguish between rrvalue reference*

const

- *if const, has to be initialized at the point of definition, can't be changed later*
- *specifies how the variable can be used, not how or where it is stored*
- *with pointers, can have a pointer to const, but can still modify it in some other way - example p. 95*

- *with pointers, const can appear in two places:*
  - read right to left for clarity
  - char * const p; constant pointer to characters - can't change contents of p, but can change things where it points
  - char const * p; same as below
  - const char * p; usual form - p is a changable pointer to characters that can't be changed - can't use p to change them, but can change p.
    - Can also change the characters through another pointer to the same place!
  - const char * const p; constant pointer to constant chars
    - Can also change the characters through another pointer to the same place!
- *const as a promise or statement of policy not to modify*
  - compiler enforces this - won't let you put something that is supposed to be const into something that doesn't keep the same promise
- *const in parameter lists*
  - normally not done for call by value, built-in types
    - might see:
    - void foo (const int i)
    - as a way of saying i is read-only for this function.
    - but void foo(int i) allows i to be modified, but won't affect caller's variable, right? It's a copy!
  - commonly done for read-only class objects called by reference to avoid constructor overhead - some objects are big and complex to create and initialize - why do it unnecessarily
    - void foo (const Big_object_type& x);
      - VERY common convention: means I don't want to waste time copying the object, because it is read-only, so let's just refer to the caller's object
    - void foo (Big_object_type& x);
      - This means that the caller's argument will be modified! Use only when that is true!
    - void foo(const Big_object_type x);
      - This means that x will be a copy of the caller's argument will be used in x, but we won't change it. Why use this? Waste time copying it for no good reason?
    - void foo(Big_object_type x);
      - This means that x will be a copy of the caller's argument, and we made the copy because we intend to change it for convenience inside foo.

- otherwise, we would have to explicitly copy it as in:
  ```
  void foo(const Big_object_type& x_in)
  {
          Big_object_type x(x_in); // use copy constructor
          x.modify();
          ....
  }
  ```

- ***CONST CORRECTNESS***
  - specify const everywhere it is logically meaningful to do so
  - gives extra protection on programming errors
  - BUT: Don't make things const that by design, have to be changable!!!
  - write it that way from the beginning.
  - if existing code is made const correct, tends to be viral - "const" spreads through the program.

- **New in C++11 rvalue references**
  - *written with &&*
  - *Micro example:*
    ```
    void foo(int&& rv)
    {
            cout << "foo(int &&) called with rvalue" << rv << endl;
    }

    int x, y;
    …
    foo(x + y);
    ```
  - *allows you to refer to rvalues - normally only used in parameters with certain overloaded functions where the compiler needs to call a different function depending on whether something is an lvalue or an rvalue.*
  - *Also useful for writing function templates that accept rvalues as well as lvalue parameters*
  - *if you declare a variable to have type rvalue rference, that variable is in fact an lvalue within its scope.*
  - *More later on what this is good for - Stroustrup provides some details here*

- **issues with struct names**
  - *forward declaration- incomplete type*
  - *name of a type becomes known immediately after it has been encountered and before the declaration is complete - can use it as long as the name of a member is not involved nor the size*
    - class S;
    - S f(); // function declaration
    - void g(S); // function declaration

    - S* h(S *);

- *in C++, using "struct" or "class" outside of a declaration is not done*
- *can use explicit "struct" and "class" for rare cases when need to disambiguate things that have the same name, but these are best avoided.*

S. 8. Structures, Unions, Enumerations

- S 8 skip 8.2.4, skim 8.2.6 on POD, skim 8.2.7 Fields, skim 8.3 Unions introduction, then skip 8.3.1 and 8.3.2.

S 9, 10 Statements and Expressions:

- **Skim the extended example in 10.2**
- **constexpr (10.4)**
  - *something that the compiler can evaluate*
  - *constexpr often better choice for simple named constants that const variables*
  - *many possibilities for compile-time evaluation*
- **casts**
  - *static_cast converts between related types (e.g. kinds of numbers or pointers in the same hierarchy*
  - *reinterpret_cast will convert unrelated pointer types*
  - *const_cast used when it is necessary to change something that unfortunately was declared const*
  - *dynamic_cast uses run-time information for conversion between types -*
  - *C-style casts are available but should not be used in modern C++ code -- too dangerous and hard to spot, intentions are not clear*
    - does anything that static_cast, reinterpret_cast, and const_cast will do.
- **constructor notation**
  - *function-style casts*
    - can write Type(value), as in
      ```
      double d;
      int i = int(d);
      ```
    - for built in types, T(v) is same as static_cast<T>(v)
    - good usage: for simple numeric type conversions
      - double x = double(my_int_var);
  - *But same notation is also used to initialize objects with constructor functions.*
    - There is a nice consistency here
    - double(a_value) means define an unnamed double variable initialized with a_value, which can then be used for something else.
  - *T() means the default value for type T - if user type, constructs an object of type T, using default constructor, built in type, the default value*
    - int i = int(); // gives value of zero
    - an UNNAMED OBJECT WITH DEFAULT CONSTRUCTION
  - **** Note also that*

- int i(5); is the same as int i = 5;

▼ *New in C++11 - using { } for constructor parameters - can't be mistaken for a function declaration*

- int i(); vs int i{};

▼ **where declarations can appear**

▼ *declarations are statements, and get executed - initialization happens when control goes through*

- static variables are the exception - initialized only once
- doing it this way allows delaying declaration until variable can be initialized, avoid errors or possible inefficiencies

▼ *declarations in conditions of if*

- scope extends from point of declaration until end of statement that condition controls - includes the else
- only a single variable allowed

▼ *declarations in for statements*

- from point of declaration until end of statement
- cf. MSVC++ error in earlier versions - allowed declarations in for, but had the wrong scope.

S 11 "Select Operations"

- S 11 Skip 11.2.4 on overloading new. Skim 11.3 "Lists" are C++11's initialization lists. Skip 11.4 Lambda Expressions - we'll come back.

Free Store new and delete(11.2)

- *free store is more official word than "heap"*
- *what does new/delete do compared with malloc/free?*
  - basically, malloc/free allocate/deallocate with blocks of raw memory, new/delete allocate/deallocate objects in memory
- *malloc*
  - allocates a block of raw memory
  - is given how many bytes you want
    - you use sizeof to determine this
  - allocates a piece of memory at least that size and returns its address to you
  - if can't allocate memory, returns NULL (or zero)
- *free*
  - deallocates a block of raw memory
  - is given an address originally supplied by malloc
  - returns that piece of memory to the pool of free memory, available for later reallocation
- *new*
  - allocates an object
  - figures out how many bytes are needed based on the type you supply
    - does the sizeof itself
  - allocates a peice of memory at least that size
  - if the type you supplied is a class-type that has a constructor, it runs the constructor on that piece of memory with the arguments you supplied (if any)
    - result is an initialized, ready-to-go object living in that piece of memory
  - returns the address of the object (piece of memory) to you.
  - if can't allocate memory, throws a Standard exception, std::bad_alloc
    - If uncaught, program is terminated
- *delete*

- deallocates an object
- is given an address originally supplied by new
- if the supplied pointer is a pointer to a class-type that has a destructor function, it runs the destructor on that piece of memory to "de initialize" or destroy the object
- returns that piece of memory to the free memory pool

▼ *new[]*

- ▼ allocates an array of objects
  - int * a = new int[n]; // allocate memory for n ints as an array
  - Thing * a = new Thing[n]; // allocate memory for an array of n Things
- figures out how much memory is needed by the number of cells you supply and the sizeof of the type of object you specify for each cell
- allocates a piece of memory at least that size
- ▼ if the cells contain a class-type object, then it runs the default constructor on each cell to initialize it.
  - no syntax for specifying a non-default constructor, unfortunately
- returns the address of the first cell to you
- ▼ if can't allocate memory, throws a Standard exception, std::bad_alloc
  - If uncaught, program is terminated

▼ *delete[]*

- deallocates an array of objects
- is given an address originally supplied by new[]
- if the pointer is a pointer to a class-type that has a destructor, it runs the destructor on each cell of the array
- returns the whole array to the pool of free memory

S 12 Functions:

- S 12 Functions. Skip 12.2.3 List Arguments; skip 12.2.4 Unspecified number of arguments. Skip 12.5 and 12.6 - redundant with C coverage.

Introduction

- *arguments are passed using initialization semantics, not assignment semantics*
- *meaning copy constructors are used, not assignment*
  - what's the difference? assignment has to assume that there is already a value in the variable - if is is of class type, might have to be destructed!
- *note use of const & to save copying*
- *can't pass in a constant or literal or must-be-converted type in as a reference, only as a const reference or value*
  - prevents assigning back to a temporary
- *in-line functions*
  - if you ask (by inline declaration), compiler can, at its option, replace a call to the function with an appropriately edited version of the functions code.
    - compiler writers get to decide how much and what they will inline - can get pretty tricky
      - e.g. can't inline a recursive function!
    - can produce considerable speedup if the function is called a gazillion times!
  - note that definition must be available to the compiler!
    - compiler has to have seen not just the prototype, but the actual code.
  - But don't specify inline without good reason - drawbacks:
    - can lead to greater code coupling - tinker with the definition, everybody using it has to recompile
    - can lead to "code bloat" - a long function body gets copied in wherever it appears

Overloaded functions

- *allow use of sensible names instead of having to make up different ones all the time*
- *can be extremely valuable e.g. in overloaded operators, constructors, etc*
- *see for the rules on matching calls to functions*
- *if ambiguous - more than one at the same level or rule, error*
- *overloading can actually help prevent errors*
- *overloading can improve efficiency*
- *return types are not considered in resolution*

- *overloading does not cross scope boundaries - only functions in same scope are considered.*
  - this can be tricky if you've defined your own namespaces - been there
- *HOW DOES OVERLOADING WORK?*
  - name mangling - compiler creates names for functions that include type information about the arguments in a special gobbledegook which you normally don't see - though sometimes you are forced to look at it.
  - result is that every overloaded function ends up with a unique name, so the linker can just do its thing as it did before - using only the function name!
  - "type safe linkage" - avoids silent errors familiar in C world

- **default arguments**
  - *only one declaration of default arguments - can''t repeat*
    - not allowed to make compiler worry about which default value is the "right" one.
    - If compiler sees two default values, it objects, even if they are the same!
    - normally means the default value goes in the function prototype (often in a header file) and not in the function definition

- **New in C++11 function overloading with parameter types including rvalues**
  - *Compiler has rules for determining which overloads are prefered if more than one applies. If rules do not choose just one, then call is ambiguous and code is rejected.*
  - *Consider the following function declarations (see examples for a RvalueRef_demo)*
    - void foo(int i); // 1. call by value parameter
      // void foo(const int i); // legal, meaningful, but usually not used
      void foo(int& i); // 2. call by reference prameter
      void foo(const int& i); // 3. call by const reference or reference to const parameter
      void foo(int&& i); // 4. rvalue reference parameter
      // void foo(const int&& i); // legal but not useful, so not used at this time
  - *consider the following function calls and which declarations they match*
    - foo(42); matches foo(int i), foo(const int& i); foo(int&& i);
    - int int_var;
      foo(int_var); matches foo(int i); foo(int& i); foo(const int&);
    - int int_var;
      int return_an_int(); // gives a pure rvalue
      foo(return_an_int()); matches foo(int i); foo(const int& i); foo(int&& i);
  - *Above calls are ambiguous - rejected by the compiler - can't tell which version of foo to call.*
  - *Situation changes if there is no call by value version foo(int i) in the picture. Compiler applies rules of overloading preference to pick best match.*
  - *Consider the following with no foo(int i) and no foo(int&& i) declared, all calls are unambiguous due to overloading preferences - the C++98 rules apply*

- int int_var = 6;
  const int c_int = 7;
  int return_an_int(); // gives a pure rvalue

  foo(int_var); calls foo(int& i); - you must want to modify the lvalue argument
  foo(c_int); calls foo(const int&i); - you can't modify the constant argumnet
  foo(42); calls foo(const int&i); - you can't modify the constant argumnet
  foo(return_an_int()); calls foo(const int& i) - you can't modify the rvalue argument

▼ *Consider the following with no foo(int i) and but with foo(int&& i) declared, all calls are unambiguous due to compiler preferences, but C++11 rules apply when rvalue parameter type is present*

- int int_var = 6;
  const int c_int = 7;
  int return_an_int(); // gives a pure rvalue

  foo(int_var); calls foo(int& i); - you must want to modify the lvalue argument
  foo(c_int); calls foo(const int&i); - you can't modify the constant argumnet
  foo(42); calls foo(int&& i); - the constant is not an lvalue, so it must be an rvalue, but call copies it into i so you still can't modify the original constant.
  foo(return_an_int()); calls foo(int&& i) - the argument is an rvalue

- *Odd behavior with foo(42) is there to enable "perfect forrwarding" of function template argument types. - Later.*
- *Conclusion: In C++11, if you declare an overload that takes an rvalue reference parameter, compiler assumes you must want it called where possible, and so changes its preferences. Otherwise, the old C++98 rules apply.*
- *We'll see how these perferences are used later*

▼ **Example code**

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


// If this version is not commented out, then all calls to foo are
ambigous and rejected.
/*
void foo(int i)  // 1
{
    cout << "foo with call by value " << i << endl;
    i++;
    cout << "foo's i is now " << i << endl;
}
*/

void foo(int& i) // 2
{
    cout << "foo with call by reference " << i << endl;
    i++;
    cout << "foo's i is now " << i << endl;
}

void foo(const int& i) // 3
{
```

```
    cout << "foo with call by reference to const " << i << endl;
 //   i++; // compile error if present
    cout << "foo's i is now " << i << endl;
}

void foo(int&& i) // 4
{
    cout << "foo with call by rvalue reference " << i << endl;
    i++;
    cout << "foo's i is now " << i << endl;
}

// used to give us a pure rvalue
int return_an_int()
{
    static int i = 10;
    i++;
    return i;
}

int main()
{
    int v_int = 6;
    const int c_int = 7;

    cout << "\ncall foo(v_int);" << endl;
    foo(v_int);
    cout << "v_int is now " << v_int << endl;

    cout << "\ncall foo(c_int);" << endl;
    foo(c_int);
    cout << "c_int is now " << c_int << endl;

    cout << "\ncall foo(42);" << endl;
    foo(42);
    cout << "paranoid: constant 42 is now " << 42 << endl;

    cout << "\ncall foo(return_an_int());" << endl;
    foo(return_an_int());

}
/*
Below, a call that fails to compile is due to either an "ambiguous"
or "no matching function" error.
If there are failed calls, they were commented out to show
which version of foo gets called by the successful calls.
*/

/* Effects of declaring only one version of foo at a time
1. if only foo(int i) declared, compiles, and all cases call foo(int i)

call foo(v_int);
foo with call by value 6
```

```
foo's i is now 7
v_int is now 6

call foo(c_int);
foo with call by value 7
foo's i is now 8
c_int is now 7

call foo(42);
foo with call by value 42
foo's i is now 43
paranoid: constant 42 is now 42

call foo(return_an_int());
foo with call by value 11
foo's i is now 12

2. if only foo(int& i) declared, only call foo(v_int); compiles, rest
fail

call foo(v_int);
foo with call by reference 6
foo's i is now 7
v_int is now 7

3. if only foo(const int& i) declared, all calls compile:

call foo(v_int);
foo with call by reference to const 6
foo's i is now 6
v_int is now 6

call foo(c_int);
foo with call by reference to const 7
foo's i is now 7
c_int is now 7

call foo(42);
foo with call by reference to const 42
foo's i is now 42
paranoid: constant 42 is now 42

call foo(return_an_int());
foo with call by reference to const 11
foo's i is now 11

4. if only foo(int&& i) declared, only foo(42); and
foo(return_an_int()); compile:

call foo(42);
foo with call by rvalue reference 42
foo's i is now 43
paranoid: constant 42 is now 42
```

```
call foo(return_an_int());
foo with call by rvalue reference 11
foo's i is now 12
*/

/* two versions of foo defined:
1 & 2. if foo(int i) and foo(int& i) declared,
foo(v_int); is ambiguous; others compile, and call foo(int i)

call foo(c_int);
foo with call by value 7
foo's i is now 8
c_int is now 7

call foo(42);
foo with call by value 42
foo's i is now 43
paranoid: constant 42 is now 42

call foo(return_an_int());
foo with call by value 11
foo's i is now 12

1 & 3. foo(int i) foo(const int& i)
all calls are ambiguous

1 & 4. foo(int i), foo(int&& i)
foo(42); foo(return_an_int()); are ambiguous

call foo(v_int);
foo with call by value 6
foo's i is now 7
v_int is now 6

call foo(c_int);
foo with call by value 7
foo's i is now 8
c_int is now 7

2 & 3. foo(int& i), foo(const int& i)
all calls compile:

call foo(v_int);
foo with call by reference 6
foo's i is now 7
v_int is now 7

call foo(c_int);
foo with call by reference to const 7
foo's i is now 7
c_int is now 7
```

```
call foo(42);
foo with call by reference to const 42
foo's i is now 42
paranoid: constant 42 is now 42

call foo(return_an_int());
foo with call by reference to const 11
foo's i is now 11

2 & 4. foo(int& i), foo(int&& i)
foo(c_int); fails to compile

call foo(v_int);
foo with call by reference 6
foo's i is now 7
v_int is now 7

call foo(42);
foo with call by rvalue reference 42
foo's i is now 43
paranoid: constant 42 is now 42

call foo(return_an_int());
foo with call by rvalue reference 11
foo's i is now 12

3 & 4. foo(const int& i), foo(int&& i)
all calls compile:

call foo(v_int);
foo with call by reference to const 6
foo's i is now 6
v_int is now 6

call foo(c_int);
foo with call by reference to const 7
foo's i is now 7
c_int is now 7

call foo(42);
foo with call by rvalue reference 42
foo's i is now 43
paranoid: constant 42 is now 42

call foo(return_an_int());
foo with call by rvalue reference 11
foo's i is now 12
*/

/*
Three versions of foo defined
1 & 2 & 3. All calls are ambiguous
```

```
1 & 2 & 4. Only foo(c_int) compiles:
call foo(c_int);
foo with call by value 7
foo's i is now 8
c_int is now 7

1 & 3 & 4. All calls ambiguous.

2 & 3 & 4. All calls compile:

call foo(v_int);
foo with call by reference 6
foo's i is now 7
v_int is now 7

call foo(c_int);
foo with call by reference to const 7
foo's i is now 7
c_int is now 7

call foo(42);
foo with call by rvalue reference 42
foo's i is now 43
paranoid: constant 42 is now 42

call foo(return_an_int());
foo with call by rvalue reference 11
foo's i is now 12
*/

/*
All 4 versions defined: All calls are ambiguous.
*/
```