- **Stroustrup review -**
- **S 4 Types and Declarations:**
  - **declaration terminology (4.9.1)**
    - *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
  - **scope (4.9.4)**
    - *block - or local scope*
    - *function parameter names are actually declared in the outer most block of a function*
    - *global - outside any function, class, or namespace*
    - *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 (4.9.5)**
    - *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!
  - **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.
        - Often used to create a temporary object in an expression or function call
        - 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
}
```

      - Example

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

        - 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
    - Unnamed objects are very commonly used in some contexts.
- **objects and lvalues (4.9.6)**
  - *an "object" is a piece of memory; an "lvalue" is an expression that designates a piece of memory*
    - roughly the l in lvalue is for left hand side of an assignment
    - but some lvalues can't be used there, and some lvalues refer to a constant
- **advice 4.10.**
  - *consistent naming style - mixed case, lower with underscores, start class or type names with a capital letter.*

- **S 5 Pointers, Arrays,**
  - **Structures: pointers and zero as a pointer value (5.1)**
    - *zero takes on a type depending on its context*
    - *use zero instead of NULL*
  - **references (5.5)**
    - *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*
  - **const (5.4)**
    - *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.

- **void * (5.6)**
  - *should only show up in C++ code at down & dirty low-levels; bad idea otherwise*
  - *note static_cast<double *>(pv) example - deliberately ugly*
- **issues with struct names (p. 103-4)**
  - *forward declaration (p. 103 center example), 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.*
- **5.8. advice**

- **S 6 Expressions and Statements:**
  - **Skim the extended example in 6.1, because he refers to it many places later.**
  - **new and delete(6.2.6)**
    - *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
      - 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
- **casts 6.2.7**
    - *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 6.2.8**
    - *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;
- **where declarations can appear(6.3.1, 6.3.2.1, 6.3.3.1),**
    - *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.
- *6.4*
    - his advice on comments makes sense
- *6.5 advice*
    - - ignore #13 - very advanced topic

- **S 7 Functions:**
  - **Introduction to 7.2**
    - *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 (7.4)**
    - *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 p. 149 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 p. 150-151.*
    - *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 (7.5)**
    - *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!

    - often means the default value goes in the function prototype (often in a header file) and not in the function definition
- **7.9 advice**

- **S 9 Source Files and Programs:**
  - **The one-definition rule (9.2.3)**
    - *A class, enumeration, template, etc must be defined exactly once in a program*
    - *more exactly:*
      - two definitions are accepted as the same unique definition iff
        - they appear in different translation units
        - they are token-for-token identical
        - the meanings of the tokens are the same
    - *the ODR is difficult for a compiler to enforce - if violated, get subtle errors - see examples p. 204*
    - *proper header file discipline (see handouts) will help avoid problems.*
    - *linker will disallow duplicate functions of same signature*
  - **program startup and termination (9.4)**
    - *Non local variables - statics, globals, class statics, are initialized before main is started. In each translation unit, in the order defined. If no initial value, then initialized to the default value for its type, e.g. 0 for int*
      - built-in type initialized before class types
    - *no guaranteed order of initialization between translation units.*
    - *termination: return from main, exit(), abort(), throwing uncaught exception*
    - *abort gives no chance of cleanup, exit does some. Throwing uncaught exception usually does better clean up and allows somebody else to handle the problem*
  - **9.5. advice**