- **Objects with dynamic memory contents**
  - **Some terminology needed for two ways in which the contents of one object are supplied by another.**
  - **Copy - one object gets a copy of the data in another object**
    - *Two notions of copy applied to member variables:*
      - shallow copy - we simply copy the values of the declared member variables.
      - deep copy - we copy the data referred to by member variables (especially data referred to by a pointer).
    - *Copy construction - create and initialize an object containing a copy of an existing object's data*
      - Thing t2(t1); // create t2 and initialize it to have a copy of the contents of t1;
        Thing t2 = t1; // same effect - not an assignment because t2 is a newly defined object
    - *Copy assignment - discard the current contents of lhs object and change it to a copy of rhs's contents*
      - Thing t1, t2;
        . . . // stuff is done to t1 and t2

        t1 = t2; // whatever was in t1 is gone, replaced by a copy of t2;
  - **Another concept - instead of copying the data, we *move* the data (new in C++11)**
    - *always a "shallow" move - no such thing as a "deep" move - only member variable values are changed.*
    - *move construction - new object gets the existing object's data, original object left in an "empty" state*
    - *move assignment - lhs gets the data in rhs; rhs left in "empty" state*
    - *Moving data is much faster than copying data, and works just fine if original owner of the data is no longer going to be used for anything*
      - But original object must be left in a state in which it can be destroyed correctly - like an "empty" state, or some other valid contents.
      - Also, good if it is left in a state where it can be assigned to - usually the case with a valid state.
  - **A final concept - *swap* of member variable values in two objects**
    - *a.swap(b) - the value of member variables of a are replaced with those of b, and vice-versa. Object a now has b's values, and b now has a's original values.*
    - *implement as a member function so it has access to all of the member variables.*
  - **We'll start with traditional copy construction and copy assignment**
- **The compiler will automatically create certain class member functions for you!**
  - **If you don't define these, the compiler will create ones for you - "compiler-supplied" member functions: Defaut constructor, destructor, copy constructor, assignment operator**
    **New in C++11: move constructor, move assignment operator**
  - **Example:**
    - *class Thing {*
      *public:*
      *void print();*
      *oid set_name(const string& new_name);*
      *private:*
      *int ID;*
      *string name;*
      *Gizmo the_gizmo;*
      *Thing * buddy_ptr;*
      *};*
  - **Compiler-supplied constructor - a default constructor - no arguments**
    - *Compiler automatically calls it for Thing thing1; Thing * p = new Thingl*

- *Does nothing, nada, zero, zilch for member variables of built-in types.*
  - built-in types: ints, doubles, chars, etc., especially: all pointer types (char*, int*, int**, Thing*, whatever).
- *Automagically calls the default constructors for any class-type member variables that have one.*
- *But in C++11, new possibility:*
  - Thing t{}; // does {} initialization on ALL members, including built-in types
    - eg. int x{}; initializes x to 0
    - to t's ID member gets initialized to 0, buddy_ptr to nullptr;
  - But can't be sure user will intialize a Thing this way (at least for some years), so best to supply a default ctor of your own.

- **Compiler-supplied destructor**
  - *Compiler automatically calls it when a Thing goes out of scope, or deleted*
  - *Does nothing, nada, zero, zilch with member variables of built-in types.*
  - *Automagically calls the destructors for any class-type member variables that have one.*
- **Compiler-supplied copy constructor**
  - *Explicit copy: Thing clone_thing(existing_thing);*
    *implicit: call or return by value: void foo (Thing t); Thing foo().*
    *Compiler calls it to create the copy on the call stack or return value location.*
  - *Simply nitializes built-in type member variables with the values in the original object.*
  - *Automagically copy-constructs any class-type member variables from the corresponding variables in the original object.*
- **Compiler-supplied assignment operator**
  - *Simply assigns built-in type lhs member variables from the values in the rhs object.*
  - *Automagically calls assignment operator for any class-type member variables to assign values from the corresponding variables in the rhs object.*
- **Come back to the C++11 move constructor and assignment operator**

## When do you need to define constructors, destructors, and the assignment operator?

- **If the compiler-supplied versions will do what you need, then you don't have to define them!**
  - *And you shouldn't define them- only do it when you need to - defining these unnecessarily is a source of errors!*
  - *The most reliable code is code that doesn't exist! It can't be wrong!*
- **Usually you need to write one or more constructors with parameters just to get your member variables intialized to their desired values.**
  - *In your constructor, if you don't explicitly initialize class-type member variables, compiler will automagically call the default constructor for them - isn't that handy!*
    - e.g. no need to explicitly initialize a std::string member variable to the empty string.
    - shouldn't do it - just clutter
  - *example:*
    - Thing(int ID_, const string& name_, Thing * buddy_ptr_) : ID(ID_), name(name_), buddy_ptr(buddy_ptr_) {} // the_gizmo is default_constructed
  - *If you definite **any** constructor at all, the compiler will **not** create a default constructor for you.*
    - e.g
    - Thing() : id(0), name("unidentified"), buddy_ptr(nullptr) {}
- **Rule of Three: "Law of the Big Three"**

- *If you need to write a destructor, then you almost certainly need to write your own copy constructor and assignment operator.*
- *Why a destructor? Because the object owns something that got allocated, and it needs to be released when the object goes away.*
- *If so, then the ownership is going to get confused if the object is copy and assigned by the compiler-supplied default memberwise assignment.*
- *So if you write a destructor, also write copy constructor and assignment operator - or rule them out by making them unavailable.*

- **Basic Rule of Virtue Rewarded**
  - *If your new class has only member variables that already have correct destruction, copy, and assignment behavior then you do not need to write them for this new class!*
  - *With good use of the Standard Library, and careful design of your own custom components, you rarely have to define the big three, and when you do, it is usually very easy.*

## The "rule of three"

- **Objects that contain a pointer to dynamic memory**
  - *Constructor usually allocates the memory*
  - *Some additional member functions involved*
  - *Destructor - deallocate the memory*
    - automatically called when either a local variable goes out of scope - e.g. function returns, or when a dynamically allocated variable is deallocated with delete
  - *Copy constructor, assignment operator prevent dangling pointers, double deletion, or memory leak possibilities*
  - *An example of issues involved with an class that allocates/deallocates some resource*
    - e.g. I/O device, network connection, etc.
  - *See example code for Array*
    - similar in some ways to standard library vector class, new Array class
    - similar in many ways to your String class for Project 2
  - *start with reminder of several limitations about C/C++ built-in arrays*
  - *show how build a class that allows an array-like type to be used like a regular variable type*

- **Digression - Why built-in arrays can be a pain**
  - **Very fast, but very dumb**
    - *Some non-standard implementations make them a tad more convenient, but still clunky.*
  - **sized at compile time - no flexibility (at least until C99)**
    - *traditional workaround: make way big enough, then use subset*
  - **have to keep track of size of array/or how much in use separately**
  - **index is not checked for valid range**
    - *with very careful code, can eliminate as a problem, but if index value is supplied externally (e.g. by user), must always check explicitly*
  - **no call by value**
    - *arrays are always passed by reference -*
      - array name corresponds to start of area in memory - pass in by pointer to first cell
      - idea: avoid copying data
      - but no way to pass an array without issue of whether caller will modify it
  - **no return by value**
    - *if you want a subroutine to put values into an array, you have to supply the array*
      - call by reference
      - subroutine puts values there
  - **can't be assigned - have to copy cells explicitly**
  - **Sometimes efficiency is great, but ease of programming is a good idea also**
    - *dynamically allocating memory gives more flexibility, but you have to keep track of the pointer, remember to deallocate it, etc.*

- **Array version 1 - Array_v1.cpp**
  - **A class of objects that can be used like an array, but also behave like regular variables.**
  - **Example encapsulates a dynamically allocated array of integers**
    - *memory is automatically freed when object is no longer in use*
    - *write this class once, use everywhere you need something better than built in array*
  - **First version - missing some key pieces! But get started.**
  - **two private member variables**
    - *a size - how many cells in the array*
    - *a pointer to integers - a poitner to the dynamically allocated memory for the array*
      - ```
        class Array {
        public:
        private:
            int size;
            int* ptr;
        };
        ```
  - **Default constructor - callable with no parameters - construct an empty Array**
    - *size and pointer are 0*
    - ```
          Array() : size(0), ptr(nullptr) {}
      ```
  - **constructof function integer parameter - how many cells in the array**
    - *keep the size, allocate a piece of memory that big*
    - ```
          Array(int size_) : size(size_) {
              if(size <= 0)
                  throw Array_Exception(size, "size must be greater than
      0");
              ptr = new int[size];
              }
      ```
  - **When object is deallocated, free the memory**
    - *when popped off a function call stack*
      - program termination, inside a function
      - when delete is used (later)
    - *compiler puts in a call to DESTRUCTOR function for you*
      - name is class name with a tilde in front
      - no return type - like constructor, you don't call it, compiler does it for you
    - *destructor will deallocate the memory with delete[]*
    - ```
          ~Array() {
              delete[] ptr;
              }
      ```
  - **a reader function for the size**
    - ```
          int get_size() const {return size;}
      ```
  - **an overloaded subscripting operator**
    - *we will check the index*
      - pull the plug if out of range - other approaches later
      ```

- *return a reference to the cell of the array*
  - can use on both the left and right hand size of an assignment
  - rhs - fetch the value - compiler knows it needs to be the const version
    - ```
      const int& operator[] (int index) const {
          if ( index < 0 || index > size - 1) {
              // throw a bad-subscript exception
              throw Array_Exception(index, "Index out of range");
              }
          return ptr[index];
          }
      ```
  - lhs - "lvalue" want to be able to set the value, so reference to the memory location
    - ```
      int& operator[] (int index) {
          if ( index < 0 || index > size - 1) {
              // throw a bad-subscript exception
              throw Array_Exception(index, "Index out of range");
              }
          return ptr[index];
          }
      ```
- **See example code**
  - *ask user for size*
  - *create an object using the size*
  - *fill it up - subscripted object returns reference to corresponding place in the internal array*
  - *ask user for an index*
    - subscript operator checks it for correct value
- **see constructor/destructor call**
  - *call zap*
  - *local object created on stack, memory allocated by constructor*
  - *object used*
  - *then object deallocated from stack - memory deallocated by destructor automagically*
- **Also, could allocate a Array object with new, then delete later see Array_v1p.cpp**

## Problems with version 1:

- **What happens if you assign one to another?**
  - *a1 = a2;*
  - *default assignment operator does memberwise assignment:*
    - a1.size = a2.size;
    - a1.ptr = a2.ptr;
  - *OK, but two problems:*
    - "copy semantics"
      - a1 and a2 share the same data - is this what we mean by assignment?
      - not usually
        - int2 = 3;
        - int1 = int2; // int1 is now 3

        - int2 = 5;
        - is int1 now 5? NO!
      - but Array.v1 will behave that way:
        - a2[i] = 3;
        - a1 = a2; // sa1[i] is now 3
        - a2[i] = 5;
        - should a1[i] now be 5? NO!
    - dangling pointer and memory leak
      - a1's ptr value has been overwritten
        - no way to free that memory up now
      - a1 and a2 point to the same piece of memory
      - whoever's destructor runs first will free it
      - second object is pointing to memory that is now deallocated
      - second object's destructor will try to free it again - bad news - why? memory allocation/ deallocation is fast but dumb! Will get confused by a double delete. Debug mode often available that keeps track of allocations and deletions and complains if they don't match up. But takes run time!
- **What happens if you try to call with Array as function argument and'/or returned value?**
  - *example code*
    *a2 = squarem (a1)*

    ```
    Array squarem(Array a)
    {
        Array b(a.get_size());

        for (int i = 0; i < a.get_size(); i++)
            b[i] = a[i] * a[i];

        return b;
    }
    ```
  - *function call stack loaded as normally with a copy of the argument:*
  - *"a" object has copy of a1's member values*
  - *inside function create b, set its values*
  - *return b - copy b out onto stack,*
    - done with b, destroy it
    - copy tempory stack value into a2
    - destroy the tempory object - more dangling pointer, memory leak problems
- **"a" is destroyed on way out - problem since it was copy of sa1's values**

- **How to prevent problems with copy and assignment**
  - **Fix by making assignment operator and COPY CONSTRUCTOR private or otherwise rule them out**
    - *compiler's rule - find the relevant function first, then check on whether access is permitted*
    - *just need to declare the function and make it private*
    - *don't have to define it, since it won't ever be called - linker won't go looking for it*
  - **assignment operator - compiler will seek to apply this, discover it can't, and signal an error**
  - **Copy constructor**
    - *X(const X&);*
    - *describes how to make a initialize an object as a COPY of another object*
    - *in function call, a copy of the object is made and put on the stack as the function parameter*
    - *in returned value, a copy of the returned object is made and put on the stack somewhere*
    - *If we make it private, we are saying that an object can not be used in a function call argument or as a returned value*
    - *avoids dangling pointer/memory leak problem*
    - *could still call by reference if we wanted to - no copy involved*
    - *if we don't supply one, compiler just does memberwise copy*
  - **Making it private might be right idea if doesn't make sense to do call/return by value or assignment.**
  - **In C++11, instead of making private, add = delete; to the declaration to show that the member function should be "suppressed" - not automatically declared and defined, so again it can't be called.**
  - **Array version 2 - Array_v2.cpp - fixed so that Array class is safe to use, but limited and inconvenient**
    - ```
          // C++11 style for forbidding copy and assignment
          Array(const Array& source) = delete;
          Array& operator= (const Array& source) = delete;

      private:
          // C++98 (or C++03) style for forbidding copy and assignment
          //  Array(const Array& source); // forbid use of copy
      constructor
          //  Array& operator= (const Array& source); // forbid use of
      assignment operator
      ```
      ```

- **How to provide copy and assignment**
  - **Have to decide what meaning of copy and assignment should be**
  - **e.g. if obj1 = obj2 or obj1 is a copy of obj2, after copy or assignment**
    - *contain same set of values*
    - *independent - changing one does not affect the other*
    - *independent lifetimes - either object can be destroyed without affecting other*
  - **simple way to do this is to give each object its own copy of the data**
- **How to define the copy constructor**
  - **Form of copy constructor prototype is**
    - *class-name(const class-name& original_object);*
    - *notice that the argument is by const reference! Can't define copying with a function that requres copying of its argument!*
  - **Notice we are initializing a new object!**
    - *Be sure ALL member variables are properly initialized!*
    - *Getting values for initialization from the object being copied.*
  - **Basic scheme**
    - *in constructor initializer list, initialize rest of this object's appropriate member variables from original_object's values*
    - *make a copy of original_object's data for this object*
      - allocate enough memory for ths object, copy original_object's data into it
    - *Example of copy constructor for Array*

```
Array(const Array& original) : size(original.size),
ptr(new int[size])
{
    for (int i = 0; i < size; i++)
        ptr[i] = original.ptr[i];
}
```

    - *See Array example Array_v3.cpp*
- **How to provide assignment**
  - **Form of overloaded assignment operator prototype is**
    - *class-name& operator= (const class-name& rhs);*
    - *define as a member function*
      - the left-hand-side is "this" object
      - the argument of the overloaded operator function is the right-hand-side
  - **Two approaches to defining the assignment function, one traditional, the other new.**
  - **See Array example Array_v3.cpp**
  - **Traditional assignment operator**
    - *First check for aliasing - make sure that the l.h.s. and r.h.s. are different objects*
      - Compare their addresses - every individual object lives at a distinct address in memory, by definition!
      - if(this != &rhs) // if different object, proceed with assignment; if not, do nothing

- Might seem bizarre - it is very rare for the objects to be the same, but can happen if pointers and references being used a lot; must be checked for because if it it does happen, and you do the assignment anyway, heap will get corrupted instantly.
- Checking for it every time actually is inefficient because it almost never happens - there is a better way to handle this.

- *If the objects are different, copy the data*
  - Deallocate the memory pointed to by the lhs
  - Allocate a new peice of memory big enough to hold the data from the rhs and Set the lhs pointer to point to it
  - Copy the rhs data into the new lhs memory space
  - Set the other lhs member variables to the rhs values
- *Return a reference to "this" object*
  - Return *this;  // always the last line of a normal assignment operator definition
  - Allows "chaining" of assignments like for built-in types:
    - thing3 = thing2 = thing1;
- *Example of traditional assignment operator for  Array*

```
Array& operator= (const Array& rhs) {
    if(this != &rhs) {
        delete[] ptr;
        size = rhs.size;
        ptr = new int[size];
        for (int i = 0; i < size; i++)
            ptr[i] = rhs.ptr[i];
        }
    return *this;
}
```

- **Better idiom for assignment operators: copy-swap**
  - *Overall better than traditional  assognment operator*
    - Provides "exception safety" - lhs side object is unchanged if assignment fails.
      - e.g. if memory allocation for copy of data fails.
    - Takes advantage of fact that copy constructor and destructor already does almost all of the work we need to do.
    - Don't waste time checking for aliasing when it almost never happens.
      - This approach wastes time only if the rare case happens, and the result is still correct.
  - *Step 1. Use copy constructor to construct a termporary object that is a copy of the rhs.*
    - Reuse the code!
    - if this throws an exception, we exit the assignment operator, but then "this" object is unchanged!
    - Basic and strong "Exception safety" - result of failure is no memory leak, and unchanged objects.
  - *Step 2. Swap the "guts" of "this" object with the temp object*
    - Interchange the values of the individual member variables.
      - Study this carefully!
      - Using code that cannot throw an exception, so if we get to this point, we will succeed.
      - While tedious to write, is usually very fast because only built-in types are involved.

        - Common for modern classes to have a "swap" member function - see Standard Library containers - they all have it!
      - Now "this" object has the new copy, and temp object has what "this" object used to have.
    - *Return *this*
    - *That's all!*
      - Destructor will automatically clean up the temp object, thereby freeing the resources that used to belong to "this" object.
    - *Example of copy-swap for Array*

```
Array& operator= (const Array& rhs)
{
    Array temp(rhs);
    swap(temp);
    return *this;
}

void swap(Array& other) noexcept
{
    int t_size = size;
    size = other.size;
    other.size = t_size;
    int * t_ptr = ptr;
    ptr = other.ptr;
    other.ptr = t_ptr;
}

// using std::swap function template
void swap(Array& other) noexcept
{
    swap(size, other.size);
    swap(ptr, other.ptr);
}
```

- **Further swap-based ideas: create and swap**
  - **Reuse constructor/destructor code more widely, making consistent behavior easier to code - common in Standard Library classes.**
  - **For assignment from another type: if you have a constructor for the other type, then create a temp object from it and swap.**
    - *Thing& operator= (const OtherType& rhs)*
      *{*
      *Thing temp(rhs);*
      *swap(temp);*
      *return *this;*
      *}*
  - **For "reset" or clear - put this object back into its default state - create an empty object (the default constructor), then swap.**
    - *void Thing::reset()*
      *{*
      *Thing temp; // default constructed*
      *swap(temp);*
      *}*

- **When does the Copy Constructor get called? Compiler often "elides" it!**
  - **Technically, copy constructor gets called in the following cases:**
    - *Explicit copy construction*
      - Thing another_thing(existing_thing);
      - Can be involved in initialization:
        - string s = "abc";
          - not an assignment, an initialization.
          - rhs has to be same type as new object being declared:
          - create a temporary unnamed string initialized with "abc", then copy-construct s from it.
          - temporary object is then destroyed.
    - *Call-by-value*
      - void foo(Thing t);
        ...
        Thing my_thing;
        …
        foo(my_thing);
        ...
      - make a copy of caller's argument and push it on the stack, where it becomes function's parameter variable; gets destroyed when function returns.
    - *Return-by-value*
      - Thing foo()
        {
        Thing result(blah, blah); // or otherwise get data into result;
        return result;
        }

        ...
        Thing a_thing;
        …
        a_thing = fool();
      - if an object is returned from a function, the function typically creates a local variable for the object, puts the data in it, and returns it. The compiler sets aside a special place on function call stack in the caller's stack area to hold the returned value before calling the function. The compiler puts in a call to the copy constructor to copy the returned value into that special place before the function finally returns. The local object is destroyed when the function returns.
      - The caller typically asigns the returned value to another variable, and the returned value is destroyed when control leaves the assignment statement (full expression).
  - **HOWEVER, for a long time C++ Compilers have been allowed to do an optimization by default!**
    - *Copy constructor elision … elision means "leave out" - compiler can elide redundant copy constructor calls*
    - *Initialization:*
      - string s = "abc" - just give the "abc" to the string constructor directly instead of build and copy.
      - a no- brainer
    - *Call by value using a temporary*
      - void foo(string s);
        …
        string s1, s2;

…
foo(s1+s2);

- compiler will make space on the stack for a temporary unnamed string object to hold the concatenation of s1 and s2. Why copy this out into another space to be foo's argument s? Instead build it where s will be, saving a call to copy constructor!

- *return by value - so common and important that it has a name: Returned Value Optimization (RVO)*
  - If the compiler can tell that a local object is going to be the return value, instead of building it locally, build it in the special return value place on the stack so it is already where it needs to be!
  - In above example, "result" is built outside foo's stack frame so it doesn't have to be copied out!
  - almost universal in C++ compilers!
- *In some compilers you can turn this optimization off so that you can see "by the book" copy construction going on.*
  - in gcc/g++, use -fno-elide-constructors option
- *The optimization is allowed by the Standard even if some side effects are missing because the copy constructor did not get called.*
  - E.g. the demonstration messages in some of the Array examples.

- **New in C++11: move construction and assignment!**
  - **Key idea: A lot of times, we do work in copy construction or assignment that is wasted because we are copying data from an object that is going to go away - it is a temporary object and is slated for destruction right away. Basic idea is to MOVE the data instead of copying it.**
    - *Actually we are "stealing" the data and leaving something destructible or assignable in its place.*
    - *move construction and assignment steals the data instead of copying it*
    - *we now call regular assignment "copy assignment"*
  - **Move assignment**
    - *assigning from an object that is going to get destroyed*
      - s = s1 + s2; // rhs is temporary object
      - vector<string> make_big_vector(); // create and return a vector<string> in a temporary object
        ...
        vector<string> vs;
        ...
        vs = make_big_vector(); // assign from a temp vector object (even if RVO happened)
    - *Basic idea: why copy data from an object that is going to be deleted? We could just "steal" the data - the object doesn't need it anymore!*
      - We just have to make sure the object could be destroyed correctly (or assigned to correctly, in some cases).
  - **New type: rvalue references**
    - *How can we tell that the rhs object is going to go away? The compiler knows!*
    - *lvalue - roughly speaking, something that can be on lhs of an assignment - or "location value" it has a name or location where you can put something. variables are lvalues. Can also appear in rhs of an assignment, obviously.*
    - *rvalue - roughly speaking, somethat that can only appear on the rhs of an assignment - a "read only" value. Temporary unnamed variables are rvalues.*
    - *New type in C++11 - an "rvalue reference" - written as && - can be called only if argument can be treated as an rvalue.*
    - *We can define an overloaded version of operator= that only gets called if rhs is an rvalue; this can move the data; copy operator= will copy the data if rhs is not an rvalue.*
    - *Overloading rules are pretty kinky when rvalue references are involved. More specifically:*
      - suppose we call foo(something); where something is an lvalue or rvalue.

- If you define both foo(X& x) and foo(X&& x)
  - then compiler will choose the first one for an lvalue, and second for an rvalue.
- If you define only foo(X& x)
  - then foo can be called only on lvalues, not rvalues - an rvalue is not allowed to bind to a reference to non-const (previous C++98 rule).
- if you define only foo(const X& x)
  - then foo will be called on both lvalues and rvalues
- if you define only foo(X&&),
  - foo can be called on rvalues, but not on lvalues
- if you define both foo(const X& x) and foo(X&& x)
  - then foo(const X& x) will be called on lvalues, and foo(X&&) will be called on rvalues
  - this is the combination we want!
- if you define foo(X x) - call by value
  - works for foo(anything) -compiler will consider the other possibilities ambiguous -
  - not usually an issue because to implement copy and move functions, we only declare constX& and X&& versions

- **Implementing move assignment**
  - *form of move assignment operator overload function:*
    *class-name& operator= (class-name&& rhs);*
  - *example - for Array, if we just swap the guts then rhs will deallocate our original stuff, and we get the data that was inside the original rhs. Like copy-swap, but no copy!* ***steal-by-swap!***
  - *Example move assignment operator - see Array_final*
    - ```
      // move assignment just swaps rhs with this.
      Array& operator= (Array&& rhs) {
          swap(rhs);
          return *this;
      }
      ```
  - *When temporary rhs object is destroyed, our original data gets deallocated. Perfect!*
    - Note that after the swap, the rhs destructor will have valid data to destroy, and in fact that object could also be assigned to correctly as well.
  - *Compiler will automatically call copy-swap operator= if rhs is an lvalue, the steal-by-swap operator= if rhs is an rvalue!*
  - *We get a potentially huge performance improvement with no change to the client code!*
- **Move construction**
  - *Similarly, if we copy-construct from an rvalue, we are copying data from an object that is going to be destroyed right away - what a waste when we could steal the data instead!*
  - *Same implementation concept: move constructor takes an rvalue reference parameter; compiler calls it instead of regular copy constructor if source is an rvalue.*
  - *Often can be simpler than move assignment because we don't have anything that needs deallocation in the object being initialized. We just have to leave the source in a deletable/assignable state.*
- **Implementing move construction**
  - *form of move constructor*
    *class-name(class-name&& source);*
  - *see Array_final example*
    - ```
      Array(Array&& original) : size(original.size), ptr(original.ptr)
      {
      ```

```
        original.size = 0;
        // delete[] of 0 pointer defined as "do nothing"
        original.ptr = nullptr;
    }
```

- *Potential big performance boost with no change in client code!*
- *Compiler will automatically call regular copy constructor if source is an lvalue, the move version that steals the data if rhs is an rvalue!*
- *However, if you like swap logic, consider the following implementation*
  - ```
    Array(Array&& original) : size(0), ptr(nullptrr)
    {
        swap(original);
    }
    ```

- **HOWEVER move construction is hard to see taking effect because normally happens only when copy construction would happen. Compilers often elide copy construction, so move construction is often not visible unless you turn off constructor elision.**
  - *One special case: a function returns its call-by-value parameter by value. Compiler is not allowed to try to build these objects in the same place, so is forced to copy/move construct its result - can't elide that constructor!*
  - ```
    Array increment_cells(Array a)
    {
        int n = a.get_size();
        for(int i = 0; i < n; i++) {
            a[i] = a[i] + 1;
            }
        return a;
    }
    // copy or move constructor will definitely get called on the
    return of a!
    ```
- **What if class type member variables are involved - how do you implement move construction or assignment with them?**
  - *You won't automatically get a move because once inside the move function, a named variable for the source is involved, and these are themselves lvalues, not rvalues. (Whoa!)*
  - *rule of thumb: if it has a name in that scope, it's an lvalue!*
  - *Example:*
    - class Gizmo {

      private:
      std::string name;
      Thing the_Thing;
      int id;
      };

    - *Consider move constructor:*
      - Gizmo g(foo()); // foo returns a Gizmo, used to initialize g - move constructor!
      - the compiler might see an rvalue and know to call this constructor, but once inside the function, the rvalue becomes a location with a name:
      - Gizmo::Gizmo(Gizmo&& original) : //original is a named location, so it is an lvalue in the function
        name(original.name), // original.name is an lvalue also - so we do a copy, not a move
        the_Thing(original. the_Thing), // ditto
        id(original.id) // a built-in type so it doesn't matter whether we copy or move
        {}

- OOPS - we ended up copying the member data, not moving it!
- *Idea: tell the compiler that it should try to apply the move constructor instead - cast these to rvalues:*
  - use static_cast<T&&>() to tell compiler to treat as an rvalue
    - this creates an unnamed temporary object of rvalue reference type!
  - if there is a constructor that takes an rvalue, then it will get called, otherwise copy constructor is called;
  - Gizmo::Gizmo(const Gizmo&& original) : //original is a name location, so it is an lvalue in the function
    name(static_cast<std::string&&>(original.name)), // std::string has move constructor, so gets called
    the_Thing(static_cast<Thing&&>(original. the_Thing), // move ctor if defined, copy ctor if not.
    id(original.id) // a built-in type so it doesn't matter whether we copy or move
    {}
  - instead of static_cast, use std::move() which is a function template that casts its argument to an rvalue no matter what the referfence-type status of it is. Does same thing here as the static_cast, but is more expressive and works correctly in other situations.
    - defined in <utility>
  - Gizmo::Gizmo(const Gizmo&& original) : //original is a name location, so it is an lvalue in the function
    name(std::move(original.name)), // std::string has move constructor, so gets called
    the_Thing(std::move(original. the_Thing), // move ctor if defined, copy ctor if not.
    id(original.id) // a built-in type so it doesn't matter whether we copy or move
    {}
- *For move assignment operator, do similarly:*
  - Gizmo& Gizmo::operator= (Gizmo&& rhs)
    {
    name = std::move(rhs.name); // calls string's move assignment
    the_Thing = std::move(rhs.the_Thing); // calls Thing's move assignment
    id = rhs.id; // just simple assignment
    }

- **What does the compiler generate for you, and how should you choose what you want your class to have?**
  - **If you don't tell the compiler what you want, it will also supply default versions of move construction and assignment along with copy construction and assignment.**
    - *default move construction: if original is an rvalue, does memberwise move initialization from original.*
    - *default move assignment: if original is an rvalue, does memberwise move assignment from original.*
    - *members with built-in types: move assignment/construction is same as copy assignment/construction*
    - *members of class types: move assignment/construction using whatever those classes have.*
  - **If your class manages no resources itselt, and if it has class-type member variables that have correct copy and move behavior, then compiler supplied defaults should be just fine!**
  - **Finer control if needed:**
    - *Can mix and match in special cases*
    - *declare and define which ones you need special treatment for*
    - *declare others with = delete; to tell compiler to not create it for you, and you aren't going to define it.*
    - *declare others with = default; to tell compiler explicitly to create the default version.*
  - **Compiler follows this rule:**

- *If you explicitly specify destructor, or any move or any copy, the compiler will not generate any moves by default.*
  - explicitly specify -> declare it, define it, say = default or =delete
  - any move -> move constructor or move assignment
  - any copy -> copy constructor or copy asignment
- *current deprecated rule: if you explicitly specify destructor, or any move or any copy, the compiler will generate default version of undeclared copy operations. (Backwards compatibility).*

- **Replace old rule of three with new rule of five:**
  - *Copy constructor*
  - *Copy assignment*
  - *move constructor*
  - *move assignment*
  - *destructor*
- **If you have to explicitly specify one of these, you need to think about and probably explicitly specify all five.**

## Exception safety concepts

- **What happens if an exception is thrown during construction, assignment, or modificaton of an object?**
  - *Usually will happen due to some time of contruction failure - e.g. when trying to make a copy of an object, something fails.*
    - Usual example: memory exhaustion, but it could be something else - like failure to establish network connections.
  - *Possibility: things are left in a mess! Ugh!*
  - *Idea of exception safety - class members makes some guarantees.*
- **Basic guarantee:**
  - *Class invariants maintained so that it can be successfully destroyed, assigned to, etc. - sitll a valid object.*
  - *No resources are leaked - e.g. no memory leaked.*
- **Strong guarantee:**
  - *If exception thrown while trying to modify an object, not only is basic guarantee met, but object is left in the original state.*
- **No throw guarantee**
  - *Some operations on the object are guaranteed not to throw an exception - means that exception won't leave the function. If violated by throwing, program is terminated, thus enforcing the guarantee. Means caller never has to worry about an exception coming out of the function.*
  - *Done by specifying noexcept on the function.*
  - *Note: By definition, destructors are not allowed to throw exceptions - no exception can leave a destructor - termination is the result.*
- **Copy/swap implementation for copy assignment operator for Array (or your String) provides both basic and strong guarantee.**
- **Array::swap() (or your String::swap()) should make the no-throw guarantee.**
- **Basic technique:**
  - *First do the work that might fail (e.g.. the copy part of copy/swap).*
  - *Then do the rest of the work that won't fail. (the swap part of copy/swap)*
  - *If the object isn't modified in the first part, then get the strong guarantee and part of the basic guarantee - invariant perserved*

- *Getting the no-leak part is easy in the Array or Strng example, can be harder in other cases.*
- *If other member variables constructed already, then compiler will call their destructors as needed.*

- **When it can be tricky: If constructing or modifying the object involves a series of operations on separate objects, any one of which might fail. If partly successful, have to undo the partial sucesses.**
  - *E.g list container copy constructor - might successfully copy first three of 5 nodes, then fail. Have to clean up the first three, leave object in original state.*
- **Technique: local try-catch while maintaining invariant**
  - *put a try-catch everything around code where operation might fail, make sure each operation maintains invariant; in catch, destroy everything that was created.*
  - *E.g. list containeer copy ctor - in catch, walk the list and destroy the nodes already created. Works if list structure was kept good in the process.*
- **Better technique: RAII with an object whose destrutor cleans up.**
  - *arrange so that the code creating the series of new objects is operating inside an object where invarieants are maintained and whose destructor will automatically clean up any objects if an exception is thrown from the inside.*
  - *E.g. list container copy ctor: declare a local list container variable, and insert each objecct from the source container. The insert code should maintain the list invariant. If the copy fails inside that insert function, then the list's destructor does the ceanup automatically and the exception this propogates out of the copy ctor.*