- **Exceptions, Classes, objects with dynamic memory contents,**
  Optional - S 14 and 15 are redundant with C coverage and earlier handouts - skim them if you want another view of these topics.
  • S 13 Exception Handling. Skip 13.3.1, skim 13.4, 13.5.2.4, 13.5.2.5, skip 13.5.3, 13.6.
  • S 16 Classes, skip 16.2.9.4.
  • H: *Incomplete Declarations*
  • H: *C++ Header File Guidelines*
  • H: *Static Members*
  • S 17 Construction, Cleanup, Copy, Move. Skip inheritance-related and initializer-list sections 17.2.3, 17.2.5, 17.3.4, 17.4.2, 17.5.1.2, 17.5.1.4

- ## Lecture Outline - Basic Exceptions

- ## Intro
  - **how to do better error handling**
  - **standard programming problem**
    - *if ignore possibility of errors, programs crash, fail, become hard to use*
    - *but trying to detect and handle errors greatly complicates the code*
    - *every single time something might go wrong, have to check for it*
    - *AND every function that calls a function in which something could go wrong has to deal with it again*
  - **common traditional structure:**
    - *each function returns a code to say whether there is a problem*
    - *each function call must check the returned value to be sure everything is OK*
    - *either way, return a similar code to the caller*
      - Example sketch code using OK/Not-OK return codes

```
int main ()
{
        ....
        if f1(....) {
                cout << "error" << endl;
                do something
                }
        return 0;      // the return code! 1 or 0?
}

bool f1 (.....)
{
        .....
        if (f2(....))
                return true;
        ....
        return false;
}

bool f2 (.....)
{
        .....
        if (f3(....))
                return true;
        ....
        return false
}

bool f3 (.....)
{
        .....
        if (z < 0)
                return true; // something's wrong!
        ....
        return false;
}
```

- **disadvantage:**
  - *lose use of return value (or have to do something even more clunky)*
  - *if' - return all over the place*
- **yuch - there's got to be a better way**
- **sometimes, there is no value to return:**
  - *Example Array class subscripting operator:*
    - See example:
      ```
      // overload the subscripting operator for this class
      int& operator[] (int index)
      {
              if ( index < 0 || index > size - 1) {
                      // this is simple, but there are better actions possible
                      cerr << "Attempt to access Smart_Array with illegal index = "
                              << index << endl;
                      cerr << "Terminating program" << endl;
                      exit(EXIT_FAILURE);
                      }
              return ptr[index];
      }
      ```
    - terminate the program because nowhere to return or check the value:
      - Array a(20);
      - a[21] = 5;
      - what do we check?
      - if(a[21]) ?
- **what to do?**
- **GENERAL IDEA: PROVIDE A SEPARATE FLOW OF CONTROL FOR ERROR SITUATIONS**
  - *most of code can be written as if nothing would go wrong*

- *separate flow of control if something does*
- *Exception concept - an fairly old idea, developed and refined before in e.g. LISP, later C++*

Exception Concept

Basic syntax:

-
  ```
  class X {
          ... whatever you want
          };
  try {
          bunch of statements
          somewhere in here, or in the functions that are called:
                  throw X(); // create and throw an X object
          }
  catch (X& x) { // catch using a reference parameter is recommended
          do something with an X exception
          }
  ... continue processing
          e.g. try again
  ```

What happens

- *Function calls proceed normally*
- *but if a "throw" is executed*
  - stack is "unwound" back up to try block that is followed by the matching catch
  - the catch block is executed
  - execution then continues after the final catch
- *unwinding the stack is equivalent to forcing a return from the function at the point of the throw, and for every function in the calling stack up to the try block*
  - like a return statement magically appears at the point of the throw, and after the call of each function along the way.
- *control is transferred from the point of the throw to the matching catch, with all functions in between immediately returning*

Sketch example

Separate error flow of control now cleans things up!

- *No need to tediously check return values!*
- *Return values can now be used for the real work!*
- *Compare to return-code sketch*

  -
    ```
    class X {
            ... whatever you want the exception class to have in it
            };

    int main ()
    {
            ....
            try {
    ```

```
                ...
                 a= f1(...);
                ...
                }
        // catch block is ignored if no throw
        catch (X& x) {
                cout << "error" << endl;
                do something
                        could quit
                        could change values
                }
        ... continue if desired
}

int f1 (.....)
        {
                .....
                b = f2()
                return i; // get return values back!
        }

int f2 (.....)
        {
                .....
                b = f3()
                return i; // get return values back!
        }

int f3 (.....)
        {
                .....
                if (z < 0)
                        throw X(); // something's wrong!
                return i; // get return values back!
        }
```

Can have more than one kind of exception

Declare them, then catch them

- *class X {*

```
class X {
        ... whatever you want
class Y {
        ... whatever you want
};
try {
        bunch of statements
        somewhere in here, or in the functions that are called:
                throw X();
                or
                throw Y();
        }
catch (X& x) {
        do something with an X exception
        }
catch (Y& y) {
        do something with a Y exception
        }
    ... continue processing
        e.g. try again
```

- *all of the catches are ignored unless there is a matching throw*
- *when catch X is finished, skips over catch Y*

Can catch in more than one place

Can catch, throw something else, rethrow the same exception

-
```
class X {
        ... whatever you want
class Y {
        ... whatever you want
};
try {
        bunch of statements
        try {
                bunch of statements

                somewhere in here, or in the functions that are called:
                        throw X();
                somewhere in here, or in the functions that are called:
                        throw Y();
                }
                catch (X& x) {
                        do something with an X exception
                        throw;// rethrow the same exception
                        or
                        throw Y();     // throw a different exception
                        }
        somewhere in here, or in the functions that are called:
                throw X();
                or
                throw Y();
        }
catch (X& x) {
        do something with an X exception
        }
catch (Y& y) {
        do something with a Y exception
        }
... continue processing
        e.g. try again
```

What happens with uncaught exception?

- **if nobody catches it, there is a default catcher hidden in the run-time environment that catches everything and terminates the program**

you can catch all exceptions with

- *catch (...) { // three dots*
- *cout << "some kind of exception caught" << endl;*

- *}*

In standard C++, memory allocation failure can be caught like this:

catch the bad_alloc exception

-
  ```
  #include <new>
  try {
          code that might allocate too much memory
          }
  catch (bad_alloc& x) {
          cout << "memory allocation failure" << endl;
          // do whatever you want
          }
  ```

Lots of other possibilities

can have a class hierarchy of exception types

- *catch by base class type, etc*

Exceptions can be in class hierarchies, can catch with the base:

- *Tip: always catch an exception object by reference ...*
- *not a bad idea: inherit from std::exception, override virtual char * what() const. Gives a uniform error reporting facility for all exceptions:*

  -
    ```
    try {
            /* stuff */
    }
    catch (exception& x)
    {
            cout << x.what() << endl;
    }
    catch ( ... )
    {
            cout << "unknown exception caught" << endl;
    }
    ```
- *derived classes can build an internal string having whatever info in it that you want, return the .c_str() for what()*

Standard library has some standard exception types that are thrown

- *e.g. bad_alloc*

- **basic idea is easy to use!**

What happens if something goes wrong with constructing an object?

- **e.g. if getting initialization data from a file, what if some of the data is invalid?**
- **note that constructors have no return value, so no obvious way to signal that it didn't work.**

- **without exceptions - only way to handle is to quit trying to construct the object and set some kind of member variable to say the object is no good, and then insist that client code check it before using the object.**
  - *object is actually a zombie - not fully initialized, but walks anyway? What do you do with it?*
- **better approach is to throw an exception - forces an exit from the constructor function**
- **What happens if an exception is thrown during construction of an object?**
  - *Any members that were successfully constructed are destroyed - their destructors are run*
  - *Any memory for the whole object that was allocated is deallocated.*
  - *Control leaves the constructor function at the point of the exception*
- **if an exception is thrown from a constructor, object does not exist!**
  - *Example code:*

    ```
    Thing * p;
    try {                    // try block defines a scope
            Thing t;
            p = new Thing;
            }
    catch(Thing_constructor_failure& x)
            {
                    // what is status of Thing t or the p's pointed to Thing at this point and later?
            }
    ```

  - *What is the status of Thing t and p's pointed-to Thing in the catch block and afterwards?*
    - Thing t is out of scope now - can't refer to it anyway!
    - Guru Sutter describes this in terms of the Monty Python dead parrot sketch.
    - There never was a parrot - it was never alive!
    - Both t, and p's pointed-to thing never existed!
    - p does not point to a valid object, or even a usable object - actually no object at all - don't try to use it in any way, shape, or form
- **What about throwing an exception in a destructor?**
  - *if an exception gets thrown out of a destructor, and we are already unwinding the stack, what is system supposed to do with the TWO exceptions now going on? rule: shouldn't happen!*
    - if happens, terminate - exception handling is officially broken!
  - *if exceptions might get thrown during destruction, you must catch them and deal with them yourself inside the destructor function before returning from it*

Only one thing to watch out for:

- **memory leaks while unwinding the stack**

-
  ```
  class Thing {
  public:
  	Thing ();
  	~Thing() {does something}
  };
  void foo ()
  {
  	int * p = new int[10000];
  	// use p for stuff
  	// use p some more
  	Thing t;
  	Thing * t_ptr;
  	t_ptr = new Thing;
  	goo();
  	...
  	delete[] p;    // we're done with the array
  	delete t_ptr; //done with the Thing
  }
  ```

- **Problem:**
  - *if goo throws an exception, foo is forced to return from the point of the call.*
  - *normal action on a return is to run the destructor function on local variables.*
    - t is a Thing, so its destructor ~Thing() is run
    - t_ptr is a pointer to Thing - it is popped off the stack, but because POINTERS ARE A BUILT-IN TYPE (like int) t_ptr doesn't have distruconstructor, so the memory it is pointing to won't get deallocated. - can have a memory leak
    - same situation with p and the block of 10000 ints we allocated
- **Fixes**
  - *catch all exceptions in foo and deallocate as needed*
  - *better - put such pointers inside an object with a destructor - "smart pointer" - or even vector<int> and make them safer, better - later*

- **Lecture Outline - Basics of Classes, Struct vs Class, How members really work**
- **Basic Concepts**
  - **Idea: User defined type is basis of OOP;**
    - *Built-in types like int, double*
      - declare them: int i, j; double x;
      - operate on them: i = i + j;
      - use them as function arguments, parameters, and return values: foo(i)   i = foo(j);   int foo (int ii) { ...}
      - value of argument is copied into space on stack for parameter
      - return value is held temporarily and then copied back to caller's variable
    - *A type*
      - represents a certain kind of data
      - a certain amount of memory required to represent the data
      - can be used in certain ways, with certain operators
    - *A user-defined type is a custom-made type that you define in order to represent things in the problem domain*
    - *An example (see examples on website for actual code)*
      - CoinMoney - a class that keeps track of money represented in coins - e.g. quarters, nickels, dimes, etc.
      - Sketch implementation:

        -
          ```
          class CoinMoney {
          public:
          CoinMoney () :
                  nickels(0), dimes(0), quarters(0) {}
          CoinMoney (int nickels_, int dimes_, int quarters_) :
                  nickels(nickels_), dimes(dimes_), quarters(quarters_) {}
          // other public functions
          private:
                  int nickels;
                  int dimes;
                  int quarters;
          };
          ```
- **How a class is like a struct, and in fact a struct is a class!**
  - **a C++ class is based on a C struct - what about structs in C++**
  - **"struct" and "class" keywords can be used interchangeably. The only difference is:**
    - *"class" - all members are private by default*

- *"struct" - all members are public by default*
- **in customary usage:**
    - *use "struct" only where you want all members to be public*
        - because that is how the class should work anyway
        - especially if only data members - same way you use a struct in C
        - "POD" class - "plain old data"
    - *use class everywhere else*
        - rule of thumb: Make all members private except for the public interface

How do member functions really work?

- **Three questions:**
    - *Where are the member functions?*
    - *How does compiler know or represent which member function goes with what class?*
    - *How do you get access to the data members without the dot operator?*
- **Member functions are actually just ordinary functions after the Compiler gets through with them.**
- **Compiler essentially rewrites your member functions in the process of compiling them.**
    - *Very first C++ actually did exactly that ... translated early C++ into C which was then compiled.*
- **first, class objects occupy a piece of memory**
- **Remember using a pointer together with a struct - as in P1**
    - *common pattern - function has a first parameter that is a pointer to the data in memory that represents the object - the function works on that object*
    - *CoinMoney * ptr;*
    - *ptr = address of a piece of memory*
    - *ptr->dimes // (*ptr).dimes*
    - *access the int that lies at address in ptr + 4 bytes.*
- **Member functions actually have an implicit parameter, "this", a pointer to the piece of memory that the current object occupies. Compiler compiles this member function**
    -
      ```
      double value()  // in CoinMoney class
      {
              return (5 * nickels + 10 * dimes + 25 * quarters) / 100.;
      }
      ```
- **as if you had written this function:**

- *double value(CoinMoney * const this)*
  {
  *return (5 * this->nickels + 10 * this->dimes + 25 * this->quarters) / 100.;*
  }

in non member code, expression invoking the member function gets rewritten as:

- *x = m1.value(); ==> x = value(&m1);*
- *or if CoinMoney * ptr;*
- *ptr = &m1;*
- *ptr->value() ==> value(ptr);*

Likewise, in member function, invoking another member function gets rewritten:

- *compute_value();*
- *compute_value(this);*

You can use the "this" pointer variable yourself - will see uses of it later.

- *this == a const pointer to the current object, with type <this class> * const*
- **this == the current object itself - dereferencing the pointer.*
- *BUT don't use it if you don't have to - just duplicating what the compiler does, wastes time, error prone, looks ignorant*

How does the compiler keep track of which function goes with what class?

- *Overloading mechanism:*
- *suppose class Gizmo {   int value() {....} };*
- *CoinMoney's value has signature:*
  - value (CoinMoney * this)
- *Gizmo's value has signature:*
  - value (Gizmo * this)
- *Different signatures, different functions!*

How does overloading work? How does it tell those apart?

- *Normally, every function has to have a unique name*
  - linkers assume it is so ...
- *Most C++ implementations actually rewrite the function name to make it include the signature!*
- *value_9CoinMoneyp*

- *value_5Gizmop*
- *called "name mangling" - same old linker logic can be used*
- *Most implementations try to hide the this parameter and the mangled name from you - supposed to be under the hood.*
  - no standard mangling scheme - up to compiler
  - but you will see it from time to time - read through the gobbledegook - you can usually figure out which class and which function is involved

Declaring and defining - member functions and variables

- **Compiler must be told about a class before you can refer to it. Provide class declaration before code refers to objects or members of the class.**
- **But within a class declaration, the code can refer to member variables or member functions before the compiler has seen those members.**
  - *As if the compiler overviews the whole class declaration before, notes the member variables and functions, then goes through an compiles the member function code.*
  - *For this process, all it needs to digest the class declarations is the member variable names and types, and the member function prototypes.*
- **Can define the functions themselves either inside or outside the class declaration.**

  -
    ```
    class Thing {
    void foo ()
            { the code } // defined inside
    };
    ```
  - *but if define function outside, have to tell the compiler which class the function belongs to, using the "scope resolution" operator, ::*

  -
    ```
    class Thing {
    void foo(); // function prototype
    };

    void Thing::foo()
    { the code}
    ```
  - *the function definition can appear anywhere later in the file, or another file altogether (the usual case) since the class declaration and member function prototype tell the compiler everything necessary to compile code that uses the class.*

Constructors

- **Compiler guarantees that constructor will be called when an object comes into being, and destructor when it ceases to exist.**

  -
    ```
    void foo()
    {
            CoinMoney m;      // compiler will insert a constructor call
            . . .
            return;
    }       // m goes out of scope, compiler will insert a destructor call
    ```

- *void foo()*
  ```
  {
          CoinMoney * p = new CoinMoney;      // new will do a constructor call
          . . .
          delete p;      // delete will do a destructor call
  ```

▼ **What do constructors do with member variables?**

- *If you assign them in a constructor function, then that's that.*
- ▼ *If not, two cases:*
  - member variable is built-in type like double, int, etc - nothing happens
  - member variable is a user-defined type, then its DEFAULT CONSTRUCTOR is run
- *member variables are constructed in the order they appear in the class declaration!*

▼ **Short-hand - constructor initializers**

- *CoinMoney() : nickels(0), dimes(0), quarters(0)   {}*
- *CoinMoney(int n, int d, int q) : nickels(n), dimes(d), quarters(q) {}*
- *These are executed in the order that the member variables are declared, not in the order that you list them! If you write initializer values that depend on each other, be very careful to get the order correct - may want to reorder the member variable declarations. Many compilers will warn if the order mismatches.*
- *Requirement : Start using now for all simple initializations*
- *Optional for simple member variables, but essential for other things*

▼ **What happens during construction if a member variable is a user-defined type?**

- *First, what happens if it is a built-in type, like int?*
- *Answer: nothing, unless you do something in the constructor*
- *But if member variable is a class type, its constructor will be called*
- ▼ *e.g. vending machine class*

  -
    ```
    class VendingMachine {
    public:
    VendingMachine()
      {}
     // other members

    private:
            CoinMoney coinbox;
    };
    ```
  - ▼ if you construct a VendingMachine, what happens to the CoinMoney variable?

    -
      ```
      VendingMachine v;
      {...}
      ```

        - answer: default constructed - all zeros, in our example
    - what if you want something else? VendingMachine constructor can arrange it
    - VendingMachine() : coinbox(1,1,1) {...} // compiler will call CoinMoney ctor with those parameters
    - VendingMachine(int n, int d, int q) : coinbox(n, d, q) {...}
- ▼ *THE FOLLOWING WILL NOT WORK, BUT EVERYBODY TRIES IT ONCE!*
    - VendingMachine(int n, int d, int q)

```
:{
        // creates a local variable which is then ignored
        CoinMoney coinbox(n, d, q);
        // OR
        // creates an un-named local Coinmoney object which is then tossed away
        CoinMoney (n, d, q);

        // this works, but why bother with it when ctor initializer will work for you?
        coinbox = CoinMoney(n, d, q);
}
```

▼ **Default function parameters in constructors**

- ▼ *Provides another way to define the default constructor*
    - ▼ the default constructor is one that can be CALLED with no arguments
        - CoinMoney int n = 0, int d = 0, int q = 0) : nickels(n), dimes(d), quarters(q) {}

▼ **New in C++11. A common case: you need more than one constructor function and have several member variables to set up in the same way. How do you avoid the duplication and get a single point of maintenance?**

- *C++98 - write a private helper function that doe the shared initializations; call from constructor body*
- ▼ *"Delegating constructors: - if you invoke another constructor of the same class in the constructor initializer list, that runs on the new object, and intiialization continues with the constructor body. Only the delegating constructor invocation can appear in the initializer list - you can't have any additional constructor initializers in the list.*
    - Example from Stroustrup:

In C++98, if you want two constructors to do the same thing, repeat yourself or call "an init() function." For example:

```
    class X {
        int a;
        validate(int x) { if (0<x && x<=max) a=x; else
throw bad_X(x); }
    public:
        X(int x) { validate(x); }
        X() { validate(42); }
        X(string s) { int x = lexical_cast<int>(s);
validate(x); }
```

```
        // ...
    };
```

Verbosity hinders readability and repetition is error-prone. Both get in the way of maintainability. So, in C++11, we can define one constructor in terms of another:

```
    class X {

        int a;

    public:

        X(int x) { if (0<x && x<=max) a=x; else throw
bad_X(x); }

        X() :X{42} { }

        X(string s) :X{lexical_cast<int>(s)} { }

        // ...

    };
```

New in C++11. In-class member initializers

- *From Stroustrup's FAQ:*
- *In-class member initializers*

In C++98, only static const members of integral types can be initialized in-class, and the initializer has to be a constant expression. These restrictions ensure that we can do the initialization at compile-time. For example:

```
    int var = 7;

    class X {
        static const int m1 = 7;        // ok
        const int m2 = 7;                   // error: not static
        static int m3 = 7;              // error: not const
        static const int m4 = var;          // error: initializer not constant
expression
        static const string m5 = "odd"; // error: not integral type
        // ...
    };
```

The basic idea for C++11 is to allow a non-static data member to be initialized where it is declared (in its class). A constructor can then use the initializer when run-time initialization is needed. Consider:

```
    class A {
    public:
        int a = 7;
    };
```

This is equivalent to:

```
      class A {
      public:
            int a;
            A() : a(7) {}
      };
```

This saves a bit of typing, but the real benefits come in classes with multiple constructors. Often, all constructors use a common initializer for a member:

```
      class A {
      public:
            A(): a(7), b(5), hash_algorithm("MD5"), s("Constructor run") {}
            A(int a_val) : a(a_val), b(5), hash_algorithm("MD5"), s("Constructor run") {}
            A(D d) : a(7), b(g(d)), hash_algorithm("MD5"), s("Constructor run") {}
            int a, b;
      private:
            HashingFunction hash_algorithm;  // Cryptographic hash to be applied to all A
instances
            std::string s;                   // String indicating state in object
lifecycle
      };
```

The fact that hash_algorithm and s each has a single default is lost in the mess of code and could easily become a problem during maintenance. Instead, we can factor out the initialization of the data members:

```
      class A {
      public:
            A(): a(7), b(5) {}
            A(int a_val) : a(a_val), b(5) {}
            A(D d) : a(7), b(g(d)) {}
            int a, b;
      private:
            HashingFunction hash_algorithm{"MD5"};  // Cryptographic hash to be applied
to all A instances
            std::string s{"Constructor run"};       // String indicating state in object
lifecycle
      };
```

If a member is initialized by both an in-class initializer and a constructor, only the constructor's initialization is done (it "overrides" the default). So we can simplify further:

```
      class A {
      public:
            A() {}
            A(int a_val) : a(a_val) {}
            A(D d) : b(g(d)) {}
            int a = 7;
            int b = 5;
      private:
            HashingFunction hash_algorithm{"MD5"};  // Cryptographic hash to be applied
to all A instances
            std::string s{"Constructor run"};       // String indicating state in object
lifecycle
     };
```

See also

- *the C++ draft section "one or two words all over the place"; see proposal.*
- *[N2628=08-0138] Michael Spertus and Bill Seymo*

▼ Const correctness with class members

- **Make member functions const if they don't change the state of the object.**
- **Member variables can be const, but if so, they can only be initialized in the constructor initializer list!**

- **Rare case: a member function is logically const - looks const to the outside world - but does its work by modifying certain private members. For example, saving computation time by caching a result internally.**
    - *Mark member function as const*
    - *Mark modified member variable as mutable - can be modified by a const member function.*
    - *THIS IS RARE - DO NOT USE IN THIS COURSE.*

Inline functions - why reader and writer functions don't hurt.

- **C++ compiler has capability of doing function "in lining"**
    - *can save time and space*
    - *body of a function is inserted at the point of the call*
    - *no function call overhead*
    - *total size of code imay be larger, but code is faster*
- **by default, functions defined inside a class declaration will be inlined (if the compiler can do it)**
- **by custom, only simple functions have their definitions in class declaration**
    - *e.g. readers and writers - they will almost certainly be inlined, so no function call overhead!*
- **complicated function defined outside the class declaration**
    - *put function declaration - prototype - in the class declaration*
        - double value();
        - still a member function, because that is where it is declared
    - *define function outside using "scope resolution operator"*
    - *double CoinMoney::value() {...}*
- **often want to set "don't inline" compiler option while debugging**
    - *e.g. there may be no separate statements to step through or set a break point on*

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*

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.*
- *Note: can copy by creating a new node containing source node's data and pushing it onto the back of the list. Push_back is much more efficient and simpler than calling insert function (which requires order relation to be instantiated as well).*

▼ **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 - push_back each node in try; The push_back code should maintain the list invariant. 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 push_back each item from the source container. The push_back code should maintain the list invariant. If the copy fails inside that push_back function, then the list's destructor does the ceanup automatically and the exception then automatically propogates out of the copy ctor.*