- **Objects with dynamic memory contents**
- **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**
  - **Example:**
    - *class Thing {*
      public:
      void print();
      void set_name(const string& new_name);
      private:
      int ID;
      string name;
      Gizmo the_gizmo;
      *Thing * ptr 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.*
  - **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 initializes 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 member variables from the values in the original object.*
    - *Automagically calls assignment operator for any class-type member variables to assign values from the corresponding variables in the original object.*
- **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!*
  - **Usually you need to write one or more constructors just to get your member variables intialized to their correct values - especially if you have built-in type member variables.**
    - *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.
    - *If you definite **any** constructor at all, the compiler will **not** create a default constructor for you.*

- **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.*
- **Basic Rule**
  - *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.*

Writing the big three

- **Objects that contain a pointer to dynamic memory**
  - *Constructor usually allocates it*
  - *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 deal with dangling pointers 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 Smart_Array*
    - similar in some ways to standard library vector class
    - similar in very 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 arrays to be used like a regular variable type*

- **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
      - 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.*

- **Smart_Array version 1 - Smart_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*
  - **constructof function integer parameter - how many cells in the array**
    - *keep the size, allocate a piece of memory that big*
  - **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[]*
  - **a reader function for the 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 - could just return the value
      - lhs - "lvalue" want to be able to set the value, so reference to the memory location
  - **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 ctor*
    - *object used*
    - *then object deallocated from stack - memory deallocated by dtor automagically*

- **Also, could allocate a Smart_Array object with new, then delete later see Smart_Array_v1p.cpp**
- **Problems with version 1:**
  - **What happens if we default construct a Smart_Array**
    - *int main()*
      {
      Smart_Array a; // ??? error - not allowed
    - *Compiler doesn't suppy a default constructor, because we supplied a constructor.*
    - *Should we provide a default ctor?*
    - *what's a sensible default size?*
      - What's the point of providing one?
    - *Better idea: make it impossible to default declare an object by making its default ctor unavailable*
      - two ways to do this:
        - If you declare any ctor yourself, then the compiler will not create a default ctor for you, so it automatically becomes non-existent and can't be called.
        - Declare the default ctor private
          - Certain design patterns and programming idioms involve private constructors.
          - rule - find the relevant function first, then check on whether access is permitted
          - just need to declare the function and make it private
          - If not supposed to be called by anybody, you don't have to define it, since it won't ever be called - linker won't go looking for it
  - **What happens if you assign one to another?**
    - *sa1 = sa2;*
    - *default assignment operator does memberwise assignment:*
      - sa1.size = sa2.size;
      - sa1.ptr = sa2.ptr;
    - *OK, but two problems:*
      - "copy semantics"
        - sa1 and sa2 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 Smart_Arrays.v1 will behave that way:
          - sa2[i] = 3;
          - sa1 = sa2; // sa1[i] is now 3
          - sa2[i] = 5;
          - should sa1[i] now be 5? NO!
      - dangling pointer and memory leak
        - sa1's ptr value has been overwritten

            - no way to free that memory up now
        - sa1 and sa2 point to the same piece of memory
        - whoever dtor's first will free it
        - second object is pointing to memory that is now deallocated
        - second object to dtor will try to free it again - bad news
- **What happens if you try to call with Smart_Array as function argument and'/or returned value?**
    - *example code*

      ```
      sa2 = squarem (sa1)

      Smart_Array squarem(Smart_Array a)
      {
              Smart_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 sa1's member values*
    - *inside function create b, set its values*
    - *return b - copy b out onto stack,*
        - done with b, dtor it
        - copy tempory stack value into sa2
        - dtor the tempory object - more dangling pointer, memory leak problems
- **"a" is dtored 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**
    - *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*
    - *by making 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 don't supply one, compiler just does memberwise copy*
  - **but might be right idea if doesn't make sense to do call/return by value or assignment**
  - **Smart_Array version 2 - Smart_Array_v2.cpp - fixed so that Smart_Array class is safe to use, but limited and inconvenient**

- **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, but sometimes inefficient, 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 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**
    - *make a copy of original_object's data for this object*
      - allocate enough memory for ths object, copy original_object's data into it
    - *in constructor initializer list, initialize rest of this object's member variables from original_object's values*
    - *Example of copy constructor for Smart_Array*
    - *See Smart_Array example Smart_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 Smart_Array example Smart_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 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; mustbe checked for because if it it does happen, and you do the assignment anyway, heap will get corrupted instantly.
    - *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 Smart_Array*
- **New idiom for assignment operators: copy-swap**
  - *Overall better than traditional assognment operator*
    - A way to get "exception safety" - lhs side object is unchanged if assignment fails.
    - Takes advantage of fact that copy constructor already does most of the work we need.
    - 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 correct.
  - *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.
    - "Exception safety" - result of failure is well-defined unchanged objects.
  - *Swap the "guts" of "this" object with the temp object*
    - Interchange the values of the individual member variables
      - 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.
      - Commonly done with a sometimes useful "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 Smart_Array*

```
// assignment operator overload - copy the data from rhs into lhs object
// return a reference to this object to allow cascaded assignments
Smart_Array& operator= (const Smart_Array& source)
{
	// create a temp copy of source (right-hand side)
	Smart_Array temp(source);
	// swap the guts of this object with the temp;
	swap(temp);
	// return reference to this object as value of the assignment expression
	return *this;
	// destructor deallocates memory of temp that used to belong to this object
}

// swap the member variable values of this object with the other
```

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

// or, using swap function template:

void swap(Smart_Array& other)
{
	swap(size, other.size);
	swap(ptr, other.ptr);
}
```

- **Generalization of copy-swap: 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 ctor), then swap.**

    -
      ```
      void Thing::reset()
      {
      	Thing temp;		// default constructed
      	swap(temp);
      }
      ```