Copy and move constructors

- it's common for us to want to initialize a new object as a copy of an existing one: hence a copy constructor
- the compiler will also make use of copy constructors for things like pass-by-value and evaluation of expressions involving objects
- sometimes we want to move content from one object to a new one: hence a move constructor
- again, the constructor will sometimes take advantage of move constructors if we've provided them

Copy constructor

- copying contents of one object to another
- pass the original as a parameter to the new one
- typically pass the original by reference (for efficiency) but as a const so it cannot accidentally be altered

```
class example {
   private:
      ...
   public:
      example(const example &orig);
}
```

```
int main()
{
    example e1;
    ...
    example e2(e1);  // copy e1 content into e2
```

Default copy constructor

- a default copy constructor is automatically created, does a field-by-field copy
- works fine if all the fields are simple types

```
class circle {
    private:
        int x, y;
        float radius;
    public:
        circle();
        void set(int xv, int yv, float rv);
};

void circle::set(int xv, int yv, float rv) {
   x = xv; y = yv; radius = rv;
}
```

```
int main()
{
    circle c1;
    c1.set(1,2,3);
    circle c2(c1);
        // implicitly created default copy constructor
        // does c2.x = c1.x
        //         c2.y = c1.y
        //         c2.radius = c1.radius
```

Shallow vs deep copy

- if the original contains more complex types then the default "shallow copy" approach may be inadequate
  - e.g. suppose one field is a pointer for a dynamically allocated array:
    - the default only copies the pointer, thus both objects have pointers to the same array
    - we probably want the new object to have its own full copy of the array

```
class example {
  private:
    int size;
    float* arr;
  ...
};
```

```
int main()
{
    example e1;
    ...
    example e2(e1);
    // sets e2.arr = e1.arr, so they both access
    // the same actual array in memory
```

Creating our own “deep” copy

- need to create a full duplicate of the original

```
class example {
   private:
      int size;
      float *arr;
   public:
      example();
      example(const example& orig);
      ...
};
```

```
example::example(const example& orig)
{
   size = orig.size;
   // create a new array for the copy
   arr = new float[size];
   // copy the contents from the original array
   for (int i = 0; i < size; i++) {
        arr[i] = orig.arr[i];
   }
}
```

Move constructors

- actually move the content from one object into a new one, removing it from the original
- uses && syntaxto reference the original and std::move to invoke the move

```
class example {
   private:
      int size;
      float* arr;
   public:
      example(example &&orig);
      ...
};
```

```
int main()
{
   example e1;
   ... assuming we do stuff to fill e1 ...
   ... then later we want to move e1's content into
   ... a new example, e2...
   example e2 = std::move(e1);
```

move constructor continued

- as with copy constructors, this is most important when dealing with dynamically allocated/complex fields
- want to be sure the move genuinely *moves* the content, removing from original

```
example::example(example &&orig)
{
    size = orig.size;
    arr = orig.arr;
    orig.arr = NULL;
    orig.size = 0;
}
```

aside: rvalues &, lvalues &&

- the & reference syntax is commonly referred to as an lvalue
- the && syntax is commonly referred to as an rvalue
- rvalues can even be used to reference values that are usually only stored temporarily
- the item referenced by the && will actually be maintained in memory as long as the reference variable is in scope

```
// trivial example:
int &&rval = 20;
// usually the 20 would have been dropped from memory by this point,
// but now it will be held there until rval goes out of scope
// ... can be used to keep results of a computation accessible for reuse ...
```