Functions Synthesized by Compiler

The material in this handout is collected from the following references:

- Sections 7.1 and 13.1 of the text book C++ Primer.
- Various sections of Effective C++.
- Various sections of More Effective C++.
- Microsoft has additional information on explicitly defaulted and deleted functions.

Compiler synthesized member functions

If you don't declare them yourself, C++ compilers will synthesize their own versions of a copy constructor, an assignment operator, a destructor, and a pair of address-of operators. Furthermore, if you don't declare any constructors, they will declare a default constructor for you, too. All these functions will be public. This means if you had defined the following class:

```
class Empty {
// empty by design
};
```

it's the same as if you'd written this:

```
class Empty {
public:
  Empty();                          // default constructor
  Empty(Empty const& rhs);          // copy constructor
  ~Empty();                         // destructor
  Empty& operator=(Empty const& rhs); // assignment operator
  Empty* operator&();               // address-of operators
  Empty const* operator&() const;
};
```

These functions are generated only if they are needed, but it doesn't take much to need them. The following code will cause each function to be generated:

```
Empty const e1;         // default ctor and dtor
Empty e2(e1);           // copy ctor syntax in old and modern C++
Empty e3{e1};           // copy ctor syntax in modern C++
e3 = e1;                // copy assignment
Empty *pe2 = &e2;       // address-of operator (non-const)
Empty const *pe1 = &e1; // address-of operator (const)
```

Given that compilers are synthesizing functions for you, what do these functions do? Well, the default constructor and the destructor don't really do anything. They just enable you to create and destroy objects of the class. The default address-of operators just return the object's address. These functions are effectively defined like this:

```
inline Empty::Empty() {}
inline Empty::~Empty() {}
inline Empty* Empty::operator&() { return this; }
inline Empty const* Empty::operator&() const { return this; }
```

As for the copy constructor and the assignment operator, the official rule is *shallow copy*: *the default copy constructor [copy assignment] performs member-wise copy construction [copy assignment] of non-`static` data members of the class.* That is, if `m` is a non-`static` data member of type `T` in a class `C` and `C` declares no copy constructor [copy assignment], `m` will be copy constructed [assigned] using the copy constructor [copy assignment] defined for `T`, if there is one. If there isn't, this rule will be recursively applied to `m`'s data members until a copy constructor [copy assignment] or built-in type [e.g., `int`, `double`, pointer, etc.] is found. By default, objects of built-in types are copy constructed [assigned] using bitwise copy from the source object to the destination object.

`default`ed functions

Here's an example to understand how synthesized functions behave. Consider the definition of a `NamedInt` class, whose instances are classes allowing you to associate names with `int` values:

```
class NamedInt {
public:
  NamedInt(char const *nam, int val);
  NamedInt(std::string const& nam, int val);

  std::string const& Name() const;
  std::string&       Name();
  int const&         Int() const;
  int&               Int();
private:
  std::string name;
  int ival;
};
```

Because `NamedInt` declares at least one constructor, compilers won't generate a default constructor. However, even though `NamedInt` fails to declare both the copy constructor and the copy assignment, compilers will generate the copy constructor and the copy assignment [if they're needed]:

```
NamedInt ni1{"1st Prime Number", 2};
NamedInt ni2{ni1}; // copy constructor generated
NamedInt ni3{"2nd Prime Number", 3};
ni2 = ni3;         // copy assignment generated
```

The copy constructor generated by the compiler must initialize `ni2.name` and `ni2.ival` using `ni1.name` and `ni1.ival`, respectively. The type of `NamedInt::name` is `string`, and `string` has a copy constructor [which you can verify by examining `string` in the standard library], so `ni2.name` will be initialized by calling the `string` copy constructor with `ni1.name` as its argument. Since the type of `NamedInt::ival` is `int` and no copy constructor is defined for `int`s, `ni2.ival` will be initialized by copying the bits over from `ni1.ival`.

When `ni3` is assigned to `ni2`, the copy assignment generated by the compiler will assign to `ni2.name` and `ni2.ival` using `ni3.name` and `ni3.ival`, respectively. Since `string` has a copy assignment defined, `ni2.name` will be assigned by calling the `string` copy assignment with `ni3.name` as its argument. Since no copy assignment is defined for `int`s, `ni2.ival` will be assigned by copying the bits over from `ni3.ival`.

We can state the obvious [that copy constructor and copy assignment must be generated when needed] to both human readers and compilers by declaring them as `=default`:

```
class NamedInt {
public:
  NamedInt(char const *nam, int val);
  NamedInt(std::string const& nam, int val);

  // these two declarations are for human readers!!!
  NamedInt(NamedInt const&)            = default;
  NamedInt& operator=(NamedInt const&) = default;

  std::string const& Name() const;
  std::string&       Name();
  int const&         Int() const;
  int&               Int();
private:
  std::string name;
  int ival;
};
```

The code for class `NamedInt` that uses keyword `default` can be found here.

`delete`d functions

The copy assignment generated by the compiler for `NamedInt` behaves as expected. But in general, compiler-generated copy assignments behave as expected only when the resulting code is both legal and has a reasonable chance of making sense. If either of these tests fails, compilers will *refuse* to generate an `operator=` for your class.

For example, suppose a class `NamedRefIntConst` were defined like this, where `name` is a *reference* to a string and `ival` is a *read only* `int`. Notice the class is explicitly requesting the compiler to generate the copy constructor and copy assignment when needed:

```
class NamedRefIntConst {
public:
  // This ctor no longer takes a const name, because name is
  // now a reference-to-non-const string.
  NamedRefIntConst(std::string& name, int value);
  // The ctor with char const* as first parameter is gone, because
  // we must have a string to refer to

  NamedRefIntConst(NamedRefIntConst const&)            = default;
  NamedRefIntConst& operator=(NamedRefIntConst const&) = default;

  // Other stuff not relevant to discussion
private:
```

```
  std::string& name;  // this is now a reference
  int const ival;     // this is now const
};
```

Now consider what should happen here:

```
std::string porsche{"Porsche"}, bmw{"Beemer"};
NamedRefIntConst old_car(porsche, 5); // I've owned this car for 5 years
NamedRefIntConst new_car(bmw, 2);     // I've owned this car for 2 years
new_car = old_car; // what should happen to the data members in new_car?
```

Before the assignment, `new_car.name` refers to some `string` object and `old_car.name` also refers to a `string`, though not the same one. How should the assignment affect `new_car.name`? After the assignment, should `new_car.name` refer to the `string` referred to by `old_car.name`, i.e., should the reference itself be modified? However, that is impossible, because C++ doesn't provide a way to make a reference refer to a different object. Alternatively, should the `string` object to which `new_car.name` refers be modified, thus affecting other objects that hold pointers or references to that `string`, i.e., objects not directly involved in the assignment? Is that what the compiler-generated copy assignment should do?

Faced with such a conundrum, C++ refuses to compile the code. If you want to support assignment in a class containing a reference member, you must define the copy assignment yourself. Compilers behave similarly for classes containing `const` members [such as `ival` in class `NamedRefIntConst`]; it's not legal to modify `const` members, so compilers are unsure how to treat them during an implicitly generated copy assignment function.

However note that the compiler will generate a copy constructor to initialize `newest_car`:

```
NamedRefIntConst newest_car{new_car};
```

After the initialization, `newest_car.name` will refer to the same `string` as `new_car` while `newest_car.ival` will be initialized with `new_car.ival`'s value. Although, the code compiles, the fact that members of multiple objects refer to the same `string` is a bit dubious!!!

In the case of class `NamedRefIntConst`, the programmer can explicitly disallow use of compiler-generated copy constructor and copy assignment using keyword `delete`:

```
class NamedRefIntConst {
public:
  // This ctor no longer takes a const name, because name is
  // now a reference-to-non-const string.
  NamedRefIntConst(std::string& name, int value);
  // The ctor with char const* as first parameter is gone, because
  // we must have a string to refer to

  // explicitly disallow any type of copy operation!!!
  NamedRefIntConst(NamedRefIntConst const&)            = delete;
  NamedRefIntConst& operator=(NamedRefIntConst const&) = delete;

  // Other stuff not relevant to discussion
private:
  std::string& name;  // this is now a reference
```

```
  int const ival;     // this is now const
};
```

The code for class `NamedRefIntConst` that uses keyword `delete` can be found here.

Code for `NamedInt`

```
#include <iostream>
#include <string>

class NamedInt {
public:
  NamedInt(char const *nam, int val);
  NamedInt(std::string const& nam, int val);

  NamedInt(NamedInt const&) = default;
  NamedInt& operator=(NamedInt const&) = default;

  std::string const& Name() const;
  std::string&       Name();
  int const&         Int() const;
  int&               Int();
private:
  std::string name;
  int ival;
};

NamedInt::NamedInt(char const *nam, int val)
  : name{nam}, ival{val} { /* empty by design */ }

NamedInt::NamedInt(std::string const& nam, int val)
  : name{nam}, ival{val} { /* empty by design */ }

std::string const& NamedInt::Name() const { return name; }
std::string&       NamedInt::Name()       { return name; }

int const&         NamedInt::Int() const { return ival; }
int&               NamedInt::Int()       { return ival; }

// overload operator << to write NamedInt to output stream ...
std::ostream& operator<<(std::ostream& os, NamedInt const& rhs) {
  os << '(' << rhs.Name() << ", " << rhs.Int() << ')';
  return os;
}

int main() {
  NamedInt ni1{"1st Prime Number", 2};
  NamedInt ni2{ni1}; // copy constructor generated
  NamedInt ni3{"2nd Prime Number", 3};
  ni2 = ni3;         // copy assignment generated
  std::cout << "ni1: " << ni1 << " | ni2: " << ni2 << " | ni3: " << ni3 <<
'\n';
}
```

Code for `NamedRefIntConst`

```cpp
#include <iostream>
#include <string>

class NamedRefIntConst {
public:
  // This ctor no longer takes a const name, because name is
  // now a reference-to-non-const string.
  NamedRefIntConst(std::string& name, int value);

  // The ctor with char const* as first parameter is gone, because
  // we must have a string to refer to

  NamedRefIntConst(NamedRefIntConst const&) = default;
  NamedRefIntConst& operator=(NamedRefIntConst const&) = default;

  // Assume no copy ctor nor copy assignment is declared ...
  std::string const& Name() const;
  std::string&       Name();
  int const&         Int() const;
private:
  std::string& name;  // this is now a reference
  int const ival;     // this is now const
};

NamedRefIntConst::NamedRefIntConst(std::string& nam, int val)
  : name{nam}, ival{val} { /* empty by design */ }

std::string const& NamedRefIntConst::Name() const { return name; }
std::string&       NamedRefIntConst::Name()       { return name; }
int const&         NamedRefIntConst::Int() const { return ival; }

// overload operator << to write NamedRefIntConst to output stream ...
std::ostream& operator<<(std::ostream& os, NamedRefIntConst const& rhs) {
  os << '(' << rhs.Name() << ", " << rhs.Int() << ')';
  return os;
}

int main() {
  std::string bmw("Beemer"), porsche("Porsche");
  NamedRefIntConst new_car(bmw, 2); // I've owned this car for 2 years
  NamedRefIntConst old_car(porsche, 5); // I've owned this car for 5 years

  // copy assignment is deleted and will therefore not compile:
  //new_car = old_car;
  std::cout << "new car: " << new_car << " | old_car: " << old_car << '\n';

  // copy construction is deleted and will therefore not compile:
  //NamedRefIntConst newest_car{old_car};
  //std::cout << "newest car: " << newest_car << '\n';
}
```