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++.

For more information, see this Microsoft page 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 that 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
Empty e3{e1};            // copy ctor in modern C++
e3 = e1;                 // assignment operator
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:

```cpp
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 this: *the default copy constructor (assignment operator) performs member-wise copy construction (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 (assignment operator), `m` will be copy constructed (assigned) using the copy constructor (assignment operator) 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 (assignment operator) 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:

```cpp
class NamedInt {
public:
  NamedInt(char const *name, int value);
  NamedInt(std::string const& name, int value);
  // we tell human readers that by design a copy ctor and op=
  // must be generated:
  NamedInt(NamedInt const&) = default;
  NamedInt& operator=(NamedInt const&) = default;
  // other stuff here that is not relevant to our discussion
private:
  std::string nameValue;
  int intValue;
};
```

> ***You can explicitly ask the compiler to generate the synthesized versions of the copy-control members by declaring them as*** `= default`**.**

Because the `NamedInt` classes declare at least one constructor, compilers won't generate default constructors, but because the programmer has explicitly asked for the synthesized versions, the compiler will generate those functions [if they're needed]. Note that if classes fail to declare copy constructors or assignment operators, compilers will generate those functions [if they are needed]. Consider the following call to a copy constructor:

```cpp
NamedInt no1("Smallest Prime Number", 2);
NamedInt no2(no1);      // calls copy constructor
```

The copy constructor generated by the compiler must initialize `no2.nameValue` and `no2.intValue` using `no1.nameValue` and `no1.intValue`, respectively. The type of `nameValue` is `string`, and `string` has a copy constructor [which you can verify by examining [`string`] in the standard library], so `no2.nameValue` will be initialized by calling the `string` copy constructor with `no1.nameValue` as its argument. On the other hand, the type of `NamedInt::intValue` is `int`, and no copy constructor is defined for `int`s, so `no2.intValue` will be initialized by copying the bits over from `no1.intValue`.

The code for class `NamedInt` and to test the keyword `default` can be found here.

`delete`d functions

The compiler-generated assignment operator for `NamedInt` would behave the same way, but in general, compiler-generated assignment operators behave as described 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 `nameValue` is a *reference* to a string and `intValue` is a `int const`:

```
class NamedRefIntConst {
public:
  // this ctor no longer takes a const name, because nameValue is
  // now a reference-to-non-const string. The char const* ctor is
  // gone, because we must have a string to refer to
  NamedRefIntConst(std::string& name, int value);
  // assume no copy ctor nor operator= is declared ...
  // other stuff here that is not relevant to our discussion
private:
  std::string& nameValue;   // this is now a reference
  int const intValue;       // this is now const
};
```

Now consider what should happen here:

```
std::string newCar("Beemer");
std::string oldCar("Porsche");
NamedRefIntConst p(newCar, 2); // I've owned this car for 2 years
NamedRefIntConst s(oldCar, 5); // I've owned this car for 5 years
// what should happen to the data members in p?
p = s;
```

Before the assignment, `p.nameValue` refers to some `string` object and `s.nameValue` also refers to a `string`, though not the same one. How should the assignment affect `p.nameValue`? After the assignment, should `p.nameValue` refer to the `string` referred to by `s.nameValue`, 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 `p.nameValue` 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 assignment operator 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 assignment operator yourself. Compilers behave similarly for classes containing `const` members [such as `intValue` in class `NamedRefIntConst`]; it's not legal to modify `const` members, so compilers are unsure how to treat them during an implicitly generated assignment function.

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

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

  // explicitly disallow any type of copy operation!!!
  NamedRefIntConst(NamedRefIntConst const&) = delete;
  NamedRefIntConst& operator=(NamedRefIntConst const&) = delete;
  // other stuff here that is not relevant to our discussion
private:
  std::string& nameValue;   // this is now a reference
  int const intValue;       // this is now const
};
```

The code to test the `delete` keyword can be found here.

Code for `NamedInt`

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

class NamedInt {
public:
  NamedInt(char const *name, int value)
    : nameValue(name), intValue(value) {}
  NamedInt(std::string const& name, int value)
    : nameValue(name), intValue(value) {}

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

  void setName(std::string const& name) { nameValue = name; }
  std::string const& getName() const { return nameValue; }
  void setInt(int value) { intValue = value; }
  int const& getInt() const { return intValue; }

private:
  std::string nameValue;
  int intValue;
};

std::ostream& operator<<(std::ostream& os, NamedInt const& rhs) {
  os << "(" << rhs.getName() << ", " << rhs.getInt() << ")";
  return os;
}

int main() {
  NamedInt no1("Smallest Prime Number", 2);
  NamedInt no2(no1);      // calls copy constructor
  std::cout << "no1: " << no1 << "\n";
  std::cout << "no2: " << no2 << "\n";

  no2.setName("Next Prime Number Is");
  no2.setInt(3);
  no1 = no2; // calls copy-assignment operator
  std::cout << "no1: " << no1 << "\n";
  std::cout << "no2: " << no2 << "\n";
}
```

Code for `NamedRefIntConst`

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

class NamedRefIntConst {
public:
  // this ctor no longer takes a const name, because nameValue is
  // now a reference-to-non-const string. The char const* ctor is
  // gone, because we must have a string to refer to
  NamedRefIntConst(std::string& name, int value)
    : nameValue{name}, intValue{value} {
      // empty by design
    }

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

  std::string const& getName() const { return nameValue; }
  int                getInt()  const { return intValue; }
  // other stuff here that is not relevant to our discussion
private:
  std::string& nameValue;   // this is now a reference
  const int intValue;       // this is now const
};

std::ostream& operator<<(std::ostream& os, NamedRefIntConst const& rhs) {
  os << "(" << rhs.getName() << ", " << rhs.getInt() << ")";
  return os;
}

int main() {
  std::string newCar("Beemer");
  std::string oldCar("Porsche");
  NamedRefIntConst p(newCar, 2); // I've owned this car for 2 years
  NamedRefIntConst s(oldCar, 5); // I've owned this car for 5

  std::cout << "p: " << p << "\n";
  std::cout << "s: " << s << "\n";

// uncomment to get errors
  //NamedRefIntConst t(s);
  //std::cout << "t: " << t << "\n";
  //p = s;
}
```