Function Declarations

Everything, also functions, must be declared before use. You may write the declaration separate from the definition of a function. The declaration consists of the return type, function name, and parameter list (the parameters need not be named). This information is called the *prototype* of the function.

```
void print(const int*, size_t); // declaration, goes
                                // in header file

void print(const int* ia, size_t n) { // definition
    ...
}
```

Overloaded Functions

There may be several functions with the same name in the same scope, provided they have different number of parameters or different parameter types (as in Java). Functions cannot be overloaded based only on differences in the return type (but member functions can be overloaded based on `const`).

Use overloading primarily when the number of parameters is different. There are complex rules for overload resolution on parameter types (read section 6.6 in Lippman if you're interested).

Default Arguments

Formal parameters may be given default values. If the corresponding argument is omitted, it is given the default value. Only trailing parameters may be given default values.

```
void move(int from, int to = 0, int by = 1);

move(2, 3); // equivalent to move(2, 3, 1)
move(2);    // equivalent to move(2, 0, 1)
```

Default parameters are often used in constructors (reduces the need for overloading):

```
Complex(double r = 0, double i = 0) : re(r), im(i) {}
```

Pointers to Functions

Functions may be passed as arguments to other functions. A function argument is passed as a *pointer* to a function, and you can define variables of type pointer to function. The syntax is complicated:

```
return_type (*pointer_name) (parameters);
```

Type aliases make definitions easier to read. Example:

```
void f(double x) {
    using DDPtrFunc = double(*)(double);
    DDPtrFunc fcn[10];
    fcn[0] = std::sin;
    ...
    fcn[9] = std::cos;
    for (size_t i = 0; i != 10; ++i) {
        cout << fcn[i](x) << endl;
    }
}
```

Classes

- Basics: members, friends, constructors, type conversions
- Destructors, allocators, reference and value semantics
- Copy control, copy constructors, assignment
- Copying and moving

Class Basics

A `Point` class where coordinates may not be negative:

```
class Point {
public:
    using coord_t = unsigned int;
    Point(coord_t ix, coord_t iy);
    coord_t get_x() const;
    coord_t get_y() const;
    void move_to(coord_t new_x, coord_t new_y);
private:
    coord_t x;
    coord_t y;
};
```

- Note the public type alias: we want users to use that name.
- The accessor functions do not change the state of the object. They shall be declared `const`.

Member Functions

```
Point::Point(coord_t ix, coord_t iy) : x(ix), y(iy) {}

Point::coord_t Point::get_x() const { return x; }

Point::coord_t Point::get_y() const { return y; }

void Point::move_to(coord_t new_x, coord_t new_y) {
    x = new_x;
    y = new_y;
}
```

- Short member functions like these are often *defined* inside the class, not only declared. Such functions are automatically inline.

const Member Functions

Member functions that are not declared const cannot be used for constant objects:

```
void f(const Point& p, Point& q) {
    Point::coord_t x1 = p.get_x(); // ok, get_x is const
    p.move(10, 20);                // wrong, move is not const
    ...
    Point::coord_t x2 = q.get_x(); // ok
    q.move(10, 20);                // ok
    ...
}
```

Some Class Notes

- `this` is a pointer to the current object, as in Java
- A `struct` is the same as a class but all members are public by default. Structs are often used for internal implementation classes.
- If two classes reference each other, you need a class *declaration*:

```
class B; // class declaration, "forward declaration"

class A {
    B* pb;
};

class B {
    A* pa;
};
```

Selective Protection

In addition to the three protection levels `private`, `protected`, and `public`, Java has "package visibility". An attribute or a method with package visibility is considered to be `private` to all classes outside of the current package, `public` to all classes inside the current package. This makes it possible to share information between classes that cooperate closely:

```
package myListPackage;
public class List {
    ListElement first; // package visible
    ...
}

package myListPackage;
public class ListElement {
    ListElement next;  // package visible
    ...
}
```

Friends

Java's package visibility gives all classes in a package extended access rights. The C++ solution is in a way more fine grained: you can hand out access rights to a specific class or to a specific (member or global) function. It is also more coarse grained: you hand out the rights to access *all* members of a class. This is done with a `friend` declaration.

```
class List {
    // the function insert may access private members of List
    friend ListElement::insert(const List&);
};

class ListElement {
    // all members of List may access members of ListElement
    friend class List;
};
```

Friend declarations may be placed anywhere in the class definition. They are *not* part of the class interface (not members).

A Generator Class

This is a "Fibonacci number generator" (the Fibonacci numbers are 1, 1, 2, 3, 5, 8, ...):

```
class Fibonacci {
public:
    Fibonacci();
    unsigned int value(unsigned int n) const;
};

unsigned int Fibonacci::value(unsigned int n) const {
    int nbr1 = 1;
    int nbr2 = 1;
    for (unsigned int i = 2; i <= n; ++i) {
        int temp = nbr1 + nbr2;
        nbr1 = nbr2;
        nbr2 = temp;
    }
    return nbr2;
}
```

Mutable Data Members

We wish to make `Fibonacci` more efficient by using a cache to save previously computed values. We add a `vector<unsigned int>` as a member. When `Fibonacci::value(n)` is called, all Fibonacci numbers up to and including `n` will be saved in the vector.

The problem is that `value` is a `const` function, but it needs to modify the member. To make this possible the member must be declared as *mutable*:

```
class Fibonacci {
    ...
private:
    mutable vector<unsigned int> values;
};
```

Fibonacci Implementation

```
Fibonacci::Fibonacci() {
    values.push_back(1);
    values.push_back(1);
}

unsigned int Fibonacci::value(unsigned int n) const {
    if (n < values.size()) {
        return values[n];
    }
    for (unsigned int i = values.size(); i <= n; ++i) {
        values.push_back(values[i - 1] + values[i - 2]);
    }
    return values[n];
}
```

Initializing Class Members

Class members can be initialized in three ways:

```
class A {
    ...
private:
    int x = 123; // direct initialization, new in C++11
};

A::A(int ix) : x(ix) {}  // constructor initializer

A::A(int ix) { x = ix; } // assignment
```

- The constructor initializer should be preferred.
- Members that are references or const cannot be assigned — *must* use initialization.
- Also members of class types that don't have a default constructor.

Initializing Data Members of Class Types

The initializers in the initializer list are constructor calls, and you may use any of the type's constructors.

```
class Example {
public:
    Example(const string& s) : s1(s), s2("abc") {}
private:
    string s1;
    string s2;
};
```

If you had omitted the initializers and instead written `s1 = s; s2 = "abc";` in the constructor body, the following would have happened:

- `s1` and `s2` would have been initialized as empty strings.
- The assignments would have been performed. This could involve a lot of work: first the empty strings are destroyed, then the strings are copied.

Delegating Constructors C++11

A constructor may delegate some (or all) of its work to another constructor by calling that constructor in the initializer list. Example:

```
class Complex {
public:
    Complex(double r, double i) : re(r), im(i) {}
    Complex(double r) : Complex(r, 0) {}
    Complex() : Complex(0, 0) {}
    ...
};
```

In this example, default parameters could have been used:

```
    Complex(double r = 0, double i = 0) : re(r), im(i) {}
```

Type Conversion and Constructors

A constructor with one parameter, `Classname(parameter)`, means that "if we know the value of the parameter, then we can construct an object of class `Classname`", for example in `Person p("Joe")`.

But this is kind of type conversion (from `string` to `Person`, in this example). Another case where type conversion is used:

```
double x = 2; // know an int value, can construct a double
```

Single-argument constructors are implicitly used for conversion when a value of class type is expected and a value of the parameter type is found.

```
Person p = "Joe"; // compiled to Person p = Person("Joe")

void print(const Person& p) { ... }

print("Bob");    // compiled to print(Person("Bob"));
```

Explicit Constructors

You can disallow implicit conversion to a class type by making the constructor *explicit* (write `explicit Person(const string&)`). Example, where the second constructor should have been explicit:

```
class String {
public:
    String(const char* s); // initialize with C string
    String(size_t n);      // string with room for n characters
};

void f(const String& s) { ... }

f("a"); // reasonable
f('a'); // probably meant "a", but the char is converted to int
        // and an empty string with 97 characters is created
f(String('a')); // allowed even if the constructor is explicit
```

Static Members

Same as `static` in Java. A class that counts the number of objects of the class, using a static attribute:

```
class Counted {
public:
    Counted() { ++nbrObj; }
    ~Counted() { --nbrObj; }
    static unsigned int getNbrObj() { return nbrObj; }
private:
    static unsigned int nbrObj;
};

unsigned int Counted::nbrObj = 0;
```

- A static data member must be defined outside the class.
- The function is accessed with `Counted::getNbrObj()`.

Managing Objects

Recall:

- Most classes need at least one *constructor*. If you don't write a constructor, the compiler synthesizes a default constructor with no parameters that does nothing (almost).
- Classes that manage a resource (dynamic memory, most commonly) need a *destructor*.
- Memory for a stack-allocated object is allocated when a function is entered, and a constructor is called to initialize the object. When the function exits, the destructor is called and the memory for the object is deallocated.
- Memory for a heap-allocated object is allocated and a constructor is called when `new Classname` is executed. The destructor is called and memory deallocated when you `delete` the object.

Example

A class that describes persons and their addresses:

```
class Person {
public:
    Person(const string& nm, const string& ad) : name(nm),
            address(ad) {}
    ...
private:
    string name;
    string address;
};

void f() {
    Person p1("Bob", "42 Elm Street");
    ...
}
```

When the function exists, the object `p1` is destroyed. This means that the attributes also will be destroyed (the `string` destructors are called).

Destructors

A destructor is necessary in a class that acquires resources that must be released, for example dynamic memory. The `Person` class again, but we for some reason use a *pointer* to a dynamic address object instead of the object itself:

```
class Person {
public:
    Person(const string& nm, const string& ad) : name(nm),
            paddress(new string(ad)) {}
    ~Person() { delete paddress; }
    ...
private:
    string name;
    string* paddress;
};
```

Here, there really is no reason to use a dynamically allocated object.

Don't Use Raw Pointers

If you really want to use dynamic memory you should choose a safe pointer instead — in this case a unique pointer. Class `unique_ptr` has a destructor which deletes the object that the pointer is pointing to.

```
class Person {
public:
    Person(const string& nm, const string& ad) : name(nm),
            paddress(new string(ad)) {}
    ...
private:
    string name;
    unique_ptr<string> paddress;
};
```

This is better since you no longer have to worry about the destructor, but it's still unnecessary to use a dynamically allocated object.

Sharing Objects

A valid reason to use a pointer to an address object is that it should be possible for several objects to *share* a common address. But then you must use a safe pointer that keeps count of how many pointers that point to an object, and only deletes the object when that number reaches zero.

```
class Person {
public:
    Person(const string& nm, const shared_ptr<string>& pad)
              : name(nm), paddress(pad) {}
    ...
private:
    string name;
    shared_ptr<string> paddress;
};
```

Example of use on the next slide.

Sharing Example

Example, two persons sharing address:

```
void f() {
    shared_ptr<string> common_address(new string("42 Elm Street"));
    Person p1("Bob", common_address);
    Person p2("Alice", common_address);
    ...
}
```

After `p1` and `p2` have been created, three pointers point to the dynamically allocated string object. When the function exits, `p2`, `p1`, and `common_address` are destroyed, the use count is decremented three times, reaches zero, and the dynamically allocated string is deleted.

Another (better) way to initialize a shared pointer:

```
auto common_address = make_shared<string>("42 Elm Street");
```